Stanford | ONLINE
STATEMENT OF ACCOMPLISHMENT

April 1, 2017

Dan Rimniceanu

has successfully completed a free online offering of

Algorithms: Design and Analysis

This is an undergraduate level course on the design and analysis of algorithms. The main topics are: asymptotic analysis, divide and conquer algorithms, sorting and searching, basic randomized algorithms, graph search, shortest paths, heaps, search trees, and hash tables. In order to earn a Statement of Accomplishment, participants were required to score at least 70% on 6 problem sets, 6 programming assignments, and 1 final exam.

Tim Roughgarden
Associate Professor of Computer Science
Stanford University

Stanford | ONLINE
STATEMENT OF ACCOMPLISHMENT

September 5, 2017

Dan Rimniceanu

has successfully completed a free online offering of

Algorithms: Design and Analysis, Part 2

This course covers greedy algorithms, including applications to minimum spanning trees and Huffman codes; dynamic programming, including applications to sequence alignment and shortest-path problems; and exact and approximation algorithms for NP-complete problems. In order to earn a Statement of Accomplishment, participants were required to score at least 70% on 6 problem sets, 6 programming assignments, and 1 final exam.

Tim Roughgarden
Associate Professor of Computer Science
Stanford University

Authenticity can be verified at **https://verify.lagunita.stanford.edu/SOA/e40649277d9e47d1a30950f36afa8dd9**

CS261: Exercise Set #1

For the week of January 4–8, 2016

Instructions:

(1) *Do not turn anything in.*

(2) The course staff is happy to discuss the solutions of these exercises with you in office hours or on Piazza.

(3) While these exercises are certainly not trivial, you should be able to complete them on your own (perhaps after consulting with the course staff or a friend for hints).

Exercise 1

Suppose we generalize the maximum flow problem so that there are *multiple* source vertices $s_1, \ldots, s_k \in V$ and sink vertices $t_1, \ldots, t_\ell \in V$. (As usual, the rest of the input is a directed graph with integer edge capacities.) You should assume that no vertex is both a source and sink, that source vertices have no incoming edges, and that sink vertices have no outgoing edges. A flow is defined as before: a nonnegative number f_e for each $e \in E$ such that capacity constraints are obeyed on every edge and such that conservation constraints hold at all vertices that are neither a source nor a sink. The value of a flow is the total amount of outgoing flow at the sources: $\sum_{i=1}^{k} \sum_{e \in \delta^+(s_i)} f_e$.

Prove that the maximum flow problem in graphs with multiple sources and sinks reduces to the single-source single-sink version of the problem. That is, given an instance of the multi-source multi-sink version of the problem, show how to (i) produce a single-source single-sink instance such that (ii) given a maximum flow to this single-source single-sink instance, you can recover a maximum flow of the original multi-source multi-sink instance. Your implementations of steps (i) and (ii) should run in linear time. Include a brief proof of correctness.

[Hint: consider adding additional vertices and/or edges.]

Exercise 2

In lecture we've focused on the maximum flow problem in directed graphs. In the *undirected* version of the problem, the input is an undirected graph $G = (V, E)$, a source vertex $s \in V$, a sink vertex $t \in V$, and a integer capacity $u_e \geq 0$ for each edge $e \in E$.

Flows are defined exactly as before, and remain directed. Formally, a *flow* consists of two nonnegative numbers f_{uv} and f_{vu} for each (undirected) edge $(u, v) \in E$, indicating the amount of traffic traversing the edge in each direction. Conservation constraints (flow in = flow out) are defined as before. Capacity constraints now state that, for every edge $e = (u, v) \in E$, the total amount of flow $f_{uv} + f_{vu}$ on the edge is at most the edge's capacity u_e. The value of a flow is the net amount $\sum_{(s,v) \in E} f_{sv} - \sum_{(v,s) \in E} f_{vs}$ going out of the source.

Prove that the maximum flow problem in undirected graphs reduces to the maximum flow problem in directed graphs. That is, given an instance of the undirected problem, show how to (i) produce an instance of the directed problem such that (ii) given a maximum flow to this directed instance, you can recover a maximum flow of the original undirected instance. Your implementations of steps (i) and (ii) should run in linear time. Include a brief proof of correctness.

[Hint: consider bidirecting each edge.]

Exercise 3

For every positive integer U, show that there is an instance of the maximum flow problem with edge capacities in $\{1, 2, \ldots, U\}$ and a choice of augmenting paths so that the Ford-Fulkerson algorithm runs for at least U iterations before terminating. The number of vertices and edges in your networks should be bounded above by constant, independent of U. (This shows that the algorithm is only "pseudopolynomial.")

[Hint: use a network similar to the examples discussed in lecture.]

Exercise 4

Consider the special case of the maximum flow problem in which every edge has capacity 1. (This is called the *unit-capacity* case.) Explain why a suitable implementation of the Ford-Fulkerson algorithm runs in $O(mn)$ time in this special case. (As always, m denotes the number of edges and n the number of vertices.)

Exercise 5

Consider a directed graph $G = (V, E)$ with source s and sink t for which each edge e has a positive integral capacity u_e. For a flow f in G, define the "layered graph" L_f as in Lecture #2, by computing the residual graph G_f and running breadth-first search (BFS) in G_f starting from s, aborting once the sink t is reached, and retaining only the forward edges. (Recall that a forward edge in BFS goes from layer i to layer $(i + 1)$, for some i.)

Recall from Lecture #2 that a *blocking flow* in a network is a flow that saturates at least one edge on each s-t path. Prove that for every flow f and every blocking flow g in L_f, the shortest-path distance between s and t in the new residual graph G_{f+g} is strictly larger than that in G_f.

CS261: Exercise Set #2

For the week of January 11–15, 2016

Instructions:

(1) *Do not turn anything in.*

(2) The course staff is happy to discuss the solutions of these exercises with you in office hours or on Piazza.

(3) While these exercises are certainly not trivial, you should be able to complete them on your own (perhaps after consulting with the course staff or a friend for hints).

Exercise 6

In the *s-t directed edge-disjoint paths problem*, the input is a directed graph $G = (V, E)$, a source vertex s, and a sink vertex t. The goal is to output a maximum-cardinality set of edge-disjoint s-t paths $P_1, \ldots, P_k$. (I.e., P_i and P_j should share no edges for each $i \neq j$, and k should be as large as possible.)

Prove that this problem reduces to the maximum flow problem. That is, given an instance of the disjoint paths problem, show how to (i) produce an instance of the maximum flow problem such that (ii) given a maximum flow to this instance, you can compute an optimal solution to the disjoint paths instance. Your implementations of steps (i) and (ii) should run in linear and polynomial time, respectively. (Can you achieve linear time also for (ii)?) Include a brief proof of correctness.

[Hint: for (ii), make use of your solution to Problem 1 (from Problem Set #1).]

Exercise 7

In the *s-t directed vertex-disjoint paths problem*, the input is a directed graph $G = (V, E)$, a source vertex s, and a sink vertex t. The goal is to output a maximum-cardinality set of internally vertex-disjoint s-t paths $P_1, \ldots, P_k$. (I.e., P_i and P_j should share no vertices other than s and t for each $i \neq j$, and k should be as large as possible.) Give a polynomial-time algorithm for this problem.

[Hint: reduce the problem either directly to the maximum flow problem or to the edge-disjoint version solved in the previous exercise.]

Exercise 8

In the *(undirected) global minimum cut problem*, the input is an undirected graph $G = (V, E)$ with a nonnegative capacity u_e for each edge $e \in E$, and the goal is to identify a cut (A, B) — i.e., a partition of V into non-empty sets A and B — that minimizes the total capacity $\sum_{e \in \delta(S)} u_e$ of the cut edges. (Here, $\delta(A)$ denotes the edges with exactly one endpoint in A.)

Prove that this problem reduces to solving $n-1$ maximum flow problems in undirected graphs.[1] That is, given an instance the global minimum cut problem, show how to (i) produce $n-1$ instances of the maximum flow problem (in undirected graphs) such that (ii) given maximum flows to these $n-1$ instances, you can compute an optimal solution to the global minimum cut instance. Your implementations of steps (i) and (ii) should run in polynomial time. Include a brief proof of correctness.

[1] And hence to solving $n-1$ maximum flow problems in directed graphs.

Exercise 9

Extend the proof of Hall's Theorem (end of Lecture #4) to show that, for every bipartite graph $G = (V \cup W, E)$ with $|V| \leq |W|$,

$$\text{maximum cardinality of a matching in } G = \min_{S \subseteq V} \left[|V| - (|S| - |N(S)|) \right].$$

Exercise 10

In lecture we proved a bound of $O(n^3)$ on the number of operations needed by the Push-Relabel algorithm (where each iteration, we select the highest vertex with excess to Push or Relabel) before it terminates with a maximum flow. Give an implementation of this algorithm that runs in $O(n^3)$ time.

[Hints: first prove the running time bound assuming that, in each iteration, you can identify the highest vertex with positive excess in $O(1)$ time. The hard part is to maintain the vertices with positive excess in a data structure such that, summed over all of the iterations of the algorithm, only $O(n^3)$ total time is used to identify these vertices. Can you get away with just a collection of buckets (implemented as lists), sorted by height?]

CS261: Exercise Set #3

For the week of January 18–22, 2016

Instructions:

(1) *Do not turn anything in.*

(2) The course staff is happy to discuss the solutions of these exercises with you in office hours or on Piazza.

(3) While these exercises are certainly not trivial, you should be able to complete them on your own (perhaps after consulting with the course staff or a friend for hints).

Exercise 11

Recall that in the maximum-weight bipartite matching problem, the input is a bipartite graph $G = (V \cup W, E)$ with a nonnegative weight w_e per edge, and the goal is to compute a matching M that maximizes $\sum_{e \in M} w_e$.

In the minimum-cost perfect bipartite matching problem, the input is a bipartite graph $G = (V \cup W, E)$ such that $|V| = |W|$ and G contains a perfect matching, and a nonnegative cost c_e per edge, and the goal is to compute a perfect matching M that minimizes $\sum_{e \in M} c_e$.

Give a linear-time reduction from the former problem to the latter problem.

Exercise 12

Suppose you are given an undirected bipartite graph $G = (V \cup W, E)$ and a positive integer b_v for every vertex $v \in V \cup W$. A *b-matching* is a subset $M \subseteq E$ of edges such that each vertex v is incident to at most b_v edges of M. (The standard bipartite matching problem corresponds to the case where $b_v = 1$ for every $v \in V \cup W$.)

Prove that the problem of computing a maximum-cardinality bipartite b-matching reduces to the problem of computing a (standard) maximum-cardinality bipartite matching in a bigger graph. Your reduction should run in time polynomial in the size of G and in $\max_{v \in V \cup W} b_v$.

Exercise 13

A graph is *d-regular* if every vertex has d incident edges. Prove that every d-regular bipartite graph is the union of d perfect matchings. Does the same statement hold for d-regular non-bipartite graphs?

[Hint: Hall's theorem.]

Exercise 14

Prove that the minimum-cost perfect bipartite matching problem reduces, in linear time, to the minimum-cost flow problem defined in Lecture #6.

Exercise 15

In the *edge cover* problem, the input is a graph $G = (V, E)$ (not necessarily bipartite) with no isolated vertices, and the goal is to compute a minimum-cardinality subset $F \subseteq E$ of edges such every vertex $v \in V$ is the endpoint of at least one edge in F. Prove that this problem reduces to the maximum-cardinality (non-bipartite) matching problem.

CS261: Exercise Set #4

For the week of January 25–29, 2016

Instructions:

(1) *Do not turn anything in.*

(2) The course staff is happy to discuss the solutions of these exercises with you in office hours or on Piazza.

(3) While these exercises are certainly not trivial, you should be able to complete them on your own (perhaps after consulting with the course staff or a friend for hints).

Exercise 16

In Lecture #7 we noted that the maximum flow problem translates quite directly into a linear program:

$$\max \sum_{e \in \delta^+(s)} f_e$$

subject to

$$\sum_{e \in \delta^-(v)} f_e - \sum_{e \in \delta^+(v)} f_e = 0 \qquad \text{for all } v \neq s, t$$

$$f_e \leq u_e \qquad \text{for all } e \in E$$

$$f_e \geq 0 \qquad \text{for all } e \in E.$$

(As usual, we are assuming that s has no incoming edges.) In Lecture #8 we considered the following alternative linear program, where $\mathcal{P}$ denotes the set of s-t paths of G:

$$\max \sum_{P \in \mathcal{P}} f_P$$

subject to

$$\sum_{P \in \mathcal{P} : e \in P} f_P \leq u_e \qquad \text{for all } e \in E$$

$$f_P \geq 0 \qquad \text{for all } P \in \mathcal{P}.$$

Prove that these two linear programs always have equal optimal objective function value.

Exercise 17

In the *multicommodity flow problem*, the input is a directed graph $G = (V, E)$ with k source vertices $s_1, \ldots, s_k$, k sink vertices $t_1, \ldots, t_k$, and a nonnegative capacity u_e for each edge $e \in E$. An s_i-t_i pair is called a *commodity*. A *multicommodity flow* if a set of k flows $\mathbf{f}^{(1)}, \ldots, \mathbf{f}^{(k)}$ such that (i) for each $i = 1, 2, \ldots, k$, $\mathbf{f}^{(i)}$ is an s_i-t_i flow (in the usual max flow sense); and (ii) for every edge e, the total amount of flow (summing over all commodities) sent on e is at most the edge capacity u_e. The *value* of a multicommodity flow is the sum of the values (in the usual max flow sense) of the flows $\mathbf{f}^{(1)}, \ldots, \mathbf{f}^{(k)}$.

Prove that the problem of finding a multicommodity flow of maximum-possible value reduces in polynomial time to solving a linear program.

Exercise 18

Consider a primal linear program (P) of the form

$$\max \mathbf{c}^T \mathbf{x}$$

subject to

$$\begin{aligned} \mathbf{A}\mathbf{x} &= \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0}. \end{aligned}$$

The recipe from Lecture #8 gives the following dual linear program (D):

$$\min \mathbf{b}^T \mathbf{y}$$

subject to

$$\begin{aligned} \mathbf{A}^T\mathbf{y} &\geq \mathbf{c} \\ \mathbf{y} &\in \mathbb{R}. \end{aligned}$$

Prove weak duality for primal-dual pairs of this form: the (primal) objective function value of every feasible solution to (P) is bounded above by the (dual) objective function value of every feasible solution to (D).[1]

Exercise 19

Consider a primal linear program (P) of the form

$$\max \mathbf{c}^T \mathbf{x}$$

subject to

$$\begin{aligned} \mathbf{A}\mathbf{x} &\leq \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

and corresponding dual program (D)

$$\min \mathbf{b}^T \mathbf{y}$$

subject to

$$\begin{aligned} \mathbf{A}^T\mathbf{y} &\geq \mathbf{c} \\ \mathbf{y} &\geq \mathbf{0}. \end{aligned}$$

Suppose $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ are feasible for (P) and (D), respectively. Prove that if $\hat{\mathbf{x}}, \hat{\mathbf{y}}$ do not satisfy the complementary slackness conditions, then $\mathbf{c}^T\hat{\mathbf{x}} \neq \mathbf{b}^T\hat{\mathbf{y}}$.

Exercise 20

Recall the linear programming relaxation of the minimum-cost bipartite matching problem:

$$\min \sum_{e \in E} c_e x_e$$

[1] In Lecture #8, we only proved weak duality for primal linear programs with only inequality constraints (and hence dual programs with nonnegative variables), like those in Exercise 19.

subject to

$$\sum_{e \in \delta(v)} x_e = 1 \qquad \text{for all } v \in V \cup W$$

$$x_e \geq 0 \qquad \text{for all } e \in E.$$

In Lecture #8 we appealed to the Hungarian algorithm to prove that this linear program is guaranteed to have an optimal solution that is 0-1. This point of this exercise is to give a direct proof of this fact, without recourse to the Hungarian algorithm.

(a) By a *fractional solution*, we mean a feasible solution to the above linear program such that $0 < x_e < 1$ for some edge $e \in E$. Prove that, for every fractional solution, there is an even cycle C of edges with $0 < x_e < 1$ for every $e \in C$.

(b) Prove that, for all ϵ sufficiently close to 0 (positive or negative), adding ϵ to x_e for every other edge of C and subtracting ϵ from x_e for the other edges of C yields another feasible solution to the linear program.

(c) Show how to transform a fractional solution $\mathbf{x}$ into another fractional solution $\mathbf{x}'$ such that: (i) $\mathbf{x}'$ has fewer fractional coordinates than $\mathbf{x}$; and (ii) the objective function value of $\mathbf{x}'$ is no larger than that of $\mathbf{x}$.

(d) Conclude that the linear programming relaxation above is guaranteed to possess an optimal solution that is 0-1 (i.e., not fractional).

CS261: Exercise Set #5

For the week of February 1–5, 2016

Instructions:

(1) *Do not turn anything in.*

(2) The course staff is happy to discuss the solutions of these exercises with you in office hours or on Piazza.

(3) While these exercises are certainly not trivial, you should be able to complete them on your own (perhaps after consulting with the course staff or a friend for hints).

Exercise 21

Consider the following linear programming relaxation of the maximum-cardinality matching problem:

$$\max \sum_{e \in E} x_e$$

subject to

$$\sum_{e \in \delta(v)} x_e \leq 1 \qquad \text{for all } v \in V$$

$$x_e \geq 0 \qquad \text{for all } e \in E,$$

where $\delta(v)$ denotes the set of edges incident to vertex v.

We know from Lecture #9 that for bipartite graphs, this linear program always has an optimal 0-1 solution. Is this also true for non-bipartite graphs?

Exercise 22

Let $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^m$ be a set of n m-vectors. Define C as the *cone* of $\mathbf{x}_1, \ldots, \mathbf{x}_n$, meaning all linear combinations of the $\mathbf{x}_i$'s that use only nonnegative coefficients:

$$C = \left\{ \sum_{i=1}^{n} \lambda_i \mathbf{x}_i \,:\, \lambda_1, \ldots, \lambda_n \geq 0 \right\}.$$

Suppose $\alpha \in \mathbb{R}^m$, $\beta \in \mathbb{R}$ define a "valid inequality" for C, meaning that

$$\alpha^T \mathbf{x} \geq \beta$$

for every $\mathbf{x} \in C$. Prove that

$$\alpha^T \mathbf{x} \geq 0$$

for every $\mathbf{x} \in C$, so α and 0 also define a valid inequality.

[Hint: Show that $\beta > 0$ is impossible. Then use the fact that if $\mathbf{x} \in C$ then $\lambda \mathbf{x} \in C$ for all scalars $\lambda \geq 0$.]

Exercise 23

Verify that the two linear programs discussed in the proof of the minimax theorem (Lecture #10),

$$\max v$$

subject to

$$\begin{aligned} v - \sum_{i=1}^{m} a_{ij} x_i &\leq 0 \qquad \text{for all } j = 1, \ldots, n \\ \sum_{i=1}^{m} x_i &= 1 \\ x_i &\geq 0 \qquad \text{for all } i = 1, \ldots, m \\ v &\in \mathbb{R}, \end{aligned}$$

and

$$\min w$$

subject to

$$\begin{aligned} w - \sum_{j=1}^{n} a_{ij} y_j &\geq 0 \qquad \text{for all } i = 1, \ldots, m \\ \sum_{j=1}^{n} y_j &= 1 \\ y_j &\geq 0 \qquad \text{for all } j = 1, \ldots, n \\ w &\in \mathbb{R}, \end{aligned}$$

are both feasible and are dual linear programs. (As in lecture, $\mathbf{A}$ is an $m \times n$ matrix, with a_{ij} specifying the payoff of the row player and the negative of the payoff of the column player when the former chooses row i and the latter chooses column j.)

Exercise 24

Consider a linear program with n decision variables, and a feasible solution $\mathbf{x} \in \mathbb{R}^n$ at which less than n of the constraints hold with equality (i.e., the rest of the constraints hold as strict inequalities).

(a) Prove that there is a direction $\mathbf{y} \in \mathbb{R}^n$ such that, for all sufficiently small $\epsilon > 0$, $\mathbf{x} + \epsilon\mathbf{y}$ and $\mathbf{x} - \epsilon\mathbf{y}$ are both feasible.

(b) Prove that at least one of $\mathbf{x} + \epsilon\mathbf{y}, \mathbf{x} - \epsilon\mathbf{y}$ has objective function value at least as good as $\mathbf{x}$.

[Context: these are the two observations that drive the fact that a linear program with a bounded feasible region always has an optimal solution at a vertex. Do you see why?]

Exercise 25

Recall from Problem #12(e) (in Problem Set #2) the following linear programming formulation of the s-t shortest path problem:

$$\min \sum_{e \in E} c_e x_e$$

subject to

$$\sum_{e \in \delta^+(S)} x_e \geq 1 \qquad \text{for all } S \subseteq V \text{ with } s \in S,\, t \notin S$$

$$x_e \geq 0 \qquad \text{for all } e \in E.$$

Prove that this linear program, while having exponentially many constraints, admits a polynomial-time separation oracle (in the sense of the ellipsoid method, see Lecture #10).

CS261: Exercise Set #6

For the week of February 8–12, 2016

Instructions:

(1) *Do not turn anything in.*

(2) The course staff is happy to discuss the solutions of these exercises with you in office hours or on Piazza.

(3) While these exercises are certainly not trivial, you should be able to complete them on your own (perhaps after consulting with the course staff or a friend for hints).

Exercise 26

In the online-decision making problem (Lecture #11), suppose that you know in advance an upper bound Q on the sum of squared rewards ($\sum_{t=1}^{T}(r^t(a))^2$) for every action $a \in A$. Explain how to modify the multiplicative weights algorithm and analysis to obtain a regret bound of $O(\sqrt{Q \log n} + \log n)$.

Exercise 27

Consider the thought experiment sketched at the end of Lecture #11: for a zero-sum game specified by the $n \times n$ matrix $\mathbf{A}$:

- At each time step $t = 1, 2, \ldots, T = \frac{4 \ln n}{\epsilon^2}$:
  - The row and column players each choose a mixed strategy ($\mathbf{p}^t$ and $\mathbf{q}^t$, respectively) using their own copies of the multiplicative weights algorithm (with the action set equal to the rows or columns, as appropriate).
  - The row player feeds the reward vector $\mathbf{r}^t = \mathbf{A}\mathbf{q}^t$ into (its copy of) the multiplicative weights algorithm. (This is just the expected payoff of each row, given that the column player chose the mixed strategy $\mathbf{q}^t$.)
  - The column player feeds the reward vector $\mathbf{r}^t = -(\mathbf{p}^t)^T\mathbf{A}$ into the multiplicative weights algorithm.

Let

$$v = \frac{1}{T}\sum_{t=1}^{T}(\mathbf{p}^t)^T\mathbf{A}\mathbf{q}^t$$

denote the time-averaged payoff of the row player. Use the multiplicative weights guarantee for the row and column players to prove that

$$v \geq \left(\max_{\mathbf{p}} \mathbf{p}^T\mathbf{A}\hat{\mathbf{q}}\right) - \epsilon$$

and

$$v \leq \left(\min_{\mathbf{q}} \hat{\mathbf{p}}^T\mathbf{A}\mathbf{q}\right) + \epsilon,$$

respectively, where $\hat{\mathbf{p}} = \frac{1}{T}\sum_{t=1}^{T}\mathbf{p}^t$ and $\hat{\mathbf{q}} = \frac{1}{T}\sum_{t=1}^{T}\mathbf{q}^t$ denote the time-averaged row and column strategies.

[Hint: first consider the maximum and minimum over all deterministic row and column strategies, respectively, rather than over all mixed strategies $\mathbf{p}$ and $\mathbf{q}$.]

Exercise 28

Use the previous exercise to prove the minimax theorem:

$$\max_{\mathbf{p}} \left(\min_{\mathbf{q}} \mathbf{p}^T \mathbf{A} \mathbf{q} \right) = \min_{\mathbf{q}} \left(\max_{\mathbf{p}} \mathbf{p}^T \mathbf{A} \mathbf{q} \right)$$

for every zero-sum game $\mathbf{A}$.

Exercise 29

There are also other notions of regret. One useful one is *swap regret*, which for an action sequence $a^1, \ldots, a^T$ and a reward vector sequence $r^1, \ldots, r^T$ is defined as

$$\max_{\delta: A \to A} \sum_{t=1}^{T} r^t(\delta(a^t)) - \sum_{t=1}^{T} r^t(a^t)$$

where the maximum ranges over all functions from A to itself. Thus the swap regret measures how much better you could do in hindsight by, for each action a, switching your action from a to some other action (on the days where you previously chose a). Prove that, even with just 3 actions, the swap regret of an action sequence can be arbitrarily larger (as $T \to \infty$) than the standard regret (as defined in Lecture #11).[1]

Exercise 30

At the end of Lecture #12 we showed how to use the multiplicative weights algorithm (as a black box) to obtain a $(1-\epsilon)$-approximate maximum flow in $O(\frac{OPT^2}{\epsilon^2} \log n)$ iterations in networks where all edges have capacity 1. (We are ignoring the outer loop that does binary search on the value of OPT.) Extend this idea to obtain the same result for maximum flow instances in which every edge capacity is at least 1.

[Hint: if $\{\ell_e^*\}_{e \in E}$ is an optimal dual solution, with value $OPT = \sum_{e \in E} c_e \ell_e^*$, then obtain a distribution by scaling each $c_e \ell_e^*$ down by OPT. What are the relevant edge lengths after this scaling?]

[1] Despite this, there are algorithms (a bit more complicated than multiplicative weights, but still reasonably simple) that guarantee swap regret sublinear in T.

CS261: Exercise Set #7

For the week of February 15–19, 2016

Instructions:

(1) *Do not turn anything in.*

(2) The course staff is happy to discuss the solutions of these exercises with you in office hours or on Piazza.

(3) While these exercises are certainly not trivial, you should be able to complete them on your own (perhaps after consulting with the course staff or a friend for hints).

Exercise 31

Recall Graham's algorithm from Lecture #13: given a parameter m (the number of machines) and n jobs arriving online with processing times $p_1, \ldots, p_n$, always assign the current job to the machine that currently has the smallest load. We proved that the schedule produced by this algorithm always has makespan (i.e., maximum machine load) at most twice the minimum possible in hindsight.

Show that for every constant $c < 2$, there exists an instance for which the schedule produced by Graham's algorithm has makespan more than c times the minimum possible.

[Hint: Your bad instances will need to grow larger as c approaches 2.]

Exercise 32

In Lecture #13 we considered the *online Steiner tree* problem, where the input is a connected undirected graph $G = (V, E)$ with nonnegative edge costs c_e, and a sequence $t_1, \ldots, t_k \in V$ of "terminals" arrive online. The goal is to output a subgraph that spans all the terminals and has total cost as small as possible. In lecture we only considered the *metric* special case, where the graph G is complete and the edge costs satisfy the triangle inequality. (I.e., for every triple $u, v, w \in V$, $c_{uw} \leq c_{uv} + c_{vw}$.) Show how to convert an α-competitive online algorithm for the metric Steiner tree problem into one for the general Steiner tree problem.[1]

[Hint: Define a metric instance where the edges represent paths in the original (non-metric) instance.]

Exercise 33

Give an infinite family of instances (with the number k of terminals tending to infinity) demonstrating that the greedy algorithm for the online Steiner tree problem is $\Omega(\log k)$-competitive (in the worst case).

Exercise 34

Let $G = (V, E)$ be an undirected graph that is connected and Eulerian (i.e., all vertices have even degree). Show that G admits an Euler tour — a (not necessarily simple) cycle that uses every edge exactly once. Can you turn your proof into an $O(m)$-time algorithm, where $m = |E|$?

[Hint: Induction on $|E|$.]

[1] This extends the $2 \ln k$ competitive ratio given in lecture to the general online Steiner tree problem.

Exercise 35

Consider the following online matching problem in general, not necessarily bipartite graphs. No information about the graph $G = (V, E)$ is given up front. Vertices arrive one-by-one. When a vertex $v \in V$ arrives, and $S \subseteq V$ are the vertices that arrived previously, the algorithm learns about all of the edges between v and vertices in S. Equivalently, after i time steps, the algorithm knows the graph $G[S_i]$ induced by the set S_i of the first i vertices.

Give a $\frac{1}{2}$-competitive online algorithm for this problem.

CS261: Exercise Set #8

For the week of February 22–26, 2016

Instructions:

(1) *Do not turn anything in.*

(2) The course staff is happy to discuss the solutions of these exercises with you in office hours or on Piazza.

(3) While these exercises are certainly not trivial, you should be able to complete them on your own (perhaps after consulting with the course staff or a friend for hints).

Exercise 36

Recall the MST heuristic for the Steiner tree problem — in Lecture #15, we showed that this is a 2-approximation algorithm. Show that, for every constant $c < 2$, there is an instance of the Steiner tree problem such that the MST heuristic returns a tree with cost more than c times that of an optimal Steiner tree.

Exercise 37

Recall the greedy algorithm for set coverage (Lecture #15). Prove that for every $k \geq 1$, there is an example where the value of the greedy solution is at most $1 - (1 - \frac{1}{k})^k$ times that of an optimal solution.

Exercise 38

Recall the MST heuristic for the metric TSP problem — in Lecture #16, we showed that this is a 2-approximation algorithm. Show that, for every constant $c < 2$, there is an instance of the metric TSP problem such that the MST heuristic returns a tour with cost more than c times the minimum possible.

Exercise 39

Recall Christofides's $\frac{3}{2}$-approximation algorithm for the metric TSP problem. Prove that the analysis given in Lecture #16 is tight: for every constant $c < \frac{3}{2}$, there is an instance of the metric TSP problem such that Christofides's algorithm returns a tour with cost more than c times the minimum possible.

Exercise 40

Consider the following variant of the traveling salesman problem (TSP). The input is an undirected complete graph with edge costs. These edge costs need *not* satisfy the triangle inequality. The desired output is the minimum-cost cycle, not necessarily simple, that visits every vertex *at least* once.

Show how to convert a polynomial-time α-approximation algorithm for the metric TSP problem into a polynomial-time α-approximation algorithm for this (non-metric) TSP problem with repeated visits allowed.

[Hint: Compare to Exercise 32.]

CS261: Exercise Set #9

For the week of February 29–March 4, 2016

Instructions:

(1) *Do not turn anything in.*

(2) The course staff is happy to discuss the solutions of these exercises with you in office hours or on Piazza.

(3) While these exercises are certainly not trivial, you should be able to complete them on your own (perhaps after consulting with the course staff or a friend for hints).

Exercise 41

Recall the Vertex Cover problem from Lecture #17: the input is an undirected graph $G = (V, E)$ and a non-negative cost c_v for each vertex $v \in V$. The goal is to compute a minimum-cost subset $S \subseteq V$ that includes at least one endpoint of each edge.

The natural greedy algorithm is:

- $S = \emptyset$
- while S is not a vertex cover:
  - add to S the vertex v minimizing (c_v/# newly covered edges)
- return S

Prove that this algorithm is not a constant-factor approximation algorithm for the vertex cover problem.

Exercise 42

Recall from Lecture #17 our linear programming relaxation of the Vertex Cover problem (with nonnegative edge costs):

$$\min \sum_{v \in V} c_v x_v$$

subject to

$$x_v + x_w \geq 1 \qquad \text{for all edges } e = (v, w) \in E$$

and

$$x_v \geq 0 \qquad \text{for all vertices } v \in V.$$

Prove that there is always a *half-integral* optimal solution $\mathbf{x}^*$ of this linear program, meaning that $x_v^* \in \{0, \frac{1}{2}, 1\}$ for every $v \in V$.

[Hint: start from an arbitrary feasible solution and show how to make it "closer to half-integral" while only improving the objective function value.]

Exercise 43

Recall the primal-dual algorithm for the vertex cover problem — in Lecture #17, we showed that this is a 2-approximation algorithm. Show that, for every constant $c < 2$, there is an instance of the vertex cover problem such that this algorithm returns a vertex cover with cost more than c times that of an optimal vertex cover.

Exercise 44

Prove *Markov's inequality*: if X is a non-negative random variable with finite expectation and $c > 1$, then

$$\mathbf{Pr}[X \geq c \cdot \mathbf{E}[X]] \leq \frac{1}{c}.$$

Exercise 45

Let X be a random variable with finite expectation and variance; recall that $\mathrm{Var}[X] = \mathbf{E}\big[(X - \mathbf{E}[X])^2\big]$ and $\mathrm{StdDev}[X] = \sqrt{\mathrm{Var}[X]}$. Prove *Chebyshev's inequality*: for every $t > 1$,

$$\mathbf{Pr}[|X - \mathbf{E}[X]\,| \geq t \cdot \mathrm{StdDev}[X]] \leq \frac{1}{t^2}.$$

[Hint: apply Markov's inequality to the (non-negative!) random variable $(X - \mathbf{E}[X])^2$.]

CS261: A Second Course in Algorithms
Lecture #1: Course Goals and Introduction to Maximum Flow*

Tim Roughgarden†

January 5, 2016

1 Course Goals

CS261 has two major course goals, and the courses splits roughly in half along these lines.

1.1 Well-Solved Problems

This first goal is very much in the spirit of an introductory course on algorithms. Indeed, the first few weeks of CS261 are pretty much a direct continuation of CS161 — the topics that we'd cover at the end of CS161 at a semester school.

Course Goal 1 Learn efficient algorithms for fundamental and well-solved problems.

There's a collection of problems that are flexible enough to model many applications and can also be solved quickly and exactly, in both theory and practice. For example, in CS161 you studied shortest-path algorithms. You should have learned all of the following:

1. The formal definition of one or more variants of the shortest-path problem.
2. Some famous shortest-path algorithms, like Dijkstra's algorithm and the Bellman-Ford algorithm, which belong in the canon of algorithms' greatest hits.
3. Applications of shortest-path algorithms, including to problems that don't explicitly involve paths in a network. For example, to the problem of planning a sequence of decisions over time.

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

The study of such problems is top priority in a course like CS161 or CS261. One of the biggest benefits of these courses is that they prevent you from reinventing the wheel (or trying to invent something that doesn't exist), instead allowing you to stand on the shoulders of the many brilliant computer scientists who preceded us. When you encounter such problems, you already have good algorithms in your toolbox and don't have to design one from scratch. This course will also give you practice spotting applications that are just thinly disguised versions of these problems.

Specifically, in the first half of the course we'll study:

1. the maximum flow problem;
2. the minimum cut problem;
3. graph matching problems;
4. linear programming, one the most general polynomial-time solvable problems known.

Our algorithms for these problems with have running times a bit bigger than those you studied in CS161 (where almost everything runs in near-linear time). Still, these algorithms are sufficiently fast that you should be happy if a problem that you care about reduces to one of these problems.

1.2 Not-So-Well-Solved Problems

Course Goal 2 Learns tools for tackling not-so-well-solved problems.

Unfortunately, many real-world problems fall into this camp, for many different reasons. We'll focus on two classes of such problems.

1. NP-hard problems, for which we don't expect there to be any exact polynomial-time algorithms. We'll study several broadly useful techniques for designing and analyzing heuristics for such problems.
2. Online problems. The anachronistic name does not refer to the Internet or social networks, but rather to the realistic case where an algorithm must make irrevocable decisions without knowing the future (i.e., without knowing the whole input).

We'll focus on algorithms for NP-hard and online problems that are guaranteed to output a solution reasonably close to an optimal one.

1.3 Intended Audience

CS261 has two audiences, both important. The first is students who are taking their final algorithms course. For this group, the goal is to pack the course with essential and likely-to-be-useful material. The second is students who are contemplating a deeper study of algorithms. With this group in mind, when the opportunity presents itself, we'll discuss

recent research developments and give you a glimpse of what you'll see in future algorithms courses. For this second audience, CS261 has a third goal.

Course Goal 3 Provide a gateway to the study of advanced algorithms.

After completing CS261, you'll be well equipped to take any of the many 200- and 300-level algorithms courses that the department offers. The pace and difficulty level of CS261 interpolates between that of CS161 and more advanced theory courses.

When you speak to audience, it's good to have one or a few "canonical audience members" in mind. For your reference and amusement, here's your instructor's mental model for canonical students in courses at different levels:

1. CS161: a constant fraction of the students do not want to be there, and/or hate math.
2. CS261: a self-selecting group of students who like algorithms and want to learn much more about them. Students may or may not love math, but they shouldn't hate it.
3. CS3xx: geared toward students who are doing or would like to do research in algorithms.

2 Introduction to the Maximum Flow Problem

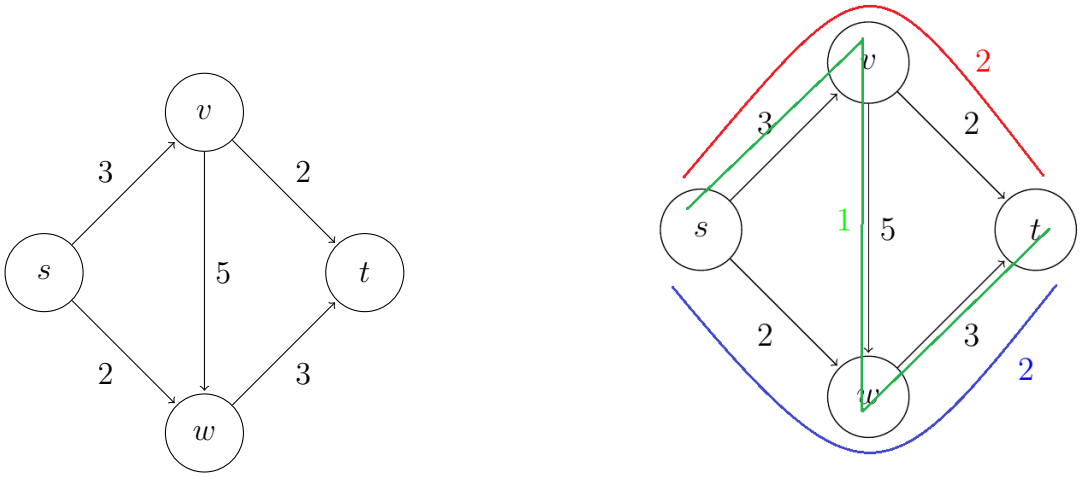

Figure 1: (a, left) Our first flow network. Each edge is associated with a capacity. (b, right) A sample flow of value 5, where the red, green and blue paths have flow values of 2, 1, 2 respectively.

2.1 Problem Definition

The maximum flow problem is a stone-cold classic in the design and analysis of algorithms. It's easy to understand intuitively, so let's do an informal example before giving the formal

definition.

The picture in Figure 1(a) resembles the ones you saw when studying shortest paths, but the semantics are different. Each edge is labeled with a *capacity*, the maximum amount of stuff that it can carry. The goal is to figure out how much stuff can be pushed from the vertex s to the vertex t.

For example, Figure 1(b) exhibits a method of pushing five units of flow from s to t, while respecting all edges' capacities. Can we do better? Certainly not, since at most 5 units of flow can escape s on its two outgoing edges.

Formally, an instance of the maximum flow problem is specified by the following ingredients:

- a directed graph G, with vertices V and directed edges E;[1]
- a *source* vertex $s \in V$;
- a *sink* vertex $t \in V$;
- a nonnegative and integral capacity u_e for each edge $e \in E$.

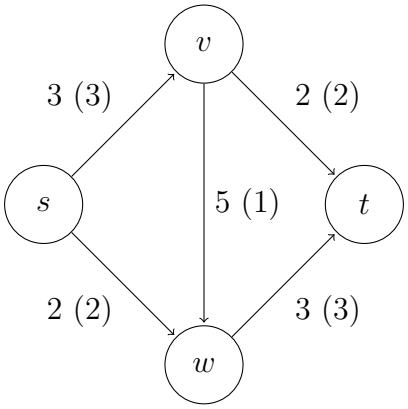

Figure 2: Denoting a flow by keeping track of the amount of flow on each edge. Flow amount is given in brackets.

.

Since the point is to push flow from s to t, we can assume without loss of generality that s has no incoming edges and t has no outgoing edges.

Given such an input, the feasible solutions are the *flows* in the network. While Figure 1(b) depicts a flow in terms of several paths, for algorithms, it works better to just keep track of the amount of flow on each edge (as in Figure 2).[2] Formally, a flow is a nonnegative vector $\{f_e\}_{e \in E}$, indexed by the edges of G, that satisfies two constraints:

[1] All of our maximum flow algorithms can be extended to undirected graphs; see Exercise Set #1.

[2] Every flow in this sense arises as the superposition of flow paths and flow cycles; see Problem #1.

Capacity constraints: $f_e \leq u_e$ for every edge $e \in E$;

Conservation constraints: for every vertex v other than s and t,

$$\text{amount of flow entering } v = \text{amount of flow exiting } v.$$

The left-hand side is the sum of the f_e's over the edge incoming to v; likewise with the outgoing edges for the right-hand side.

The objective is to compute a *maximum flow* — a flow with the maximum-possible *value*, meaning the total amount of flow that leaves s. (As we'll see, this is the same as the total amount of flow that enters t.)

2.2 Applications

Why should we care about the maximum flow problem? Like all central algorithmic problems, the maximum flow problem is useful in its own right, plus many different problems are really just thinly disguised version of maximum flow. For some relatively obvious and literal applications, the maximum flow problem can model the routing of traffic through a transportation network, packets through a data network, or oil through a distribution network.[3] In upcoming lectures we'll prove the less obvious fact that problems ranging from bipartite matching to image segmentation reduce to the maximum flow problem.

2.3 A Naive Greedy Algorithm

We now turn our attention to the design of efficient algorithms for the maximum flow problem. A priori, it is not clear that any such algorithms exist (for all we know right now, the problem is NP-hard).

We begin by considering greedy algorithms. Recall that a greedy algorithm is one that makes a sequence of myopic and irrevocable decisions, with the hope that everything somehow works out at the end. For most problems, greedy algorithms do not generally produce the best-possible solution. But it's still worth trying them, because the ways in which greedy algorithms break often yields insights that lead to better algorithms.

The simplest greedy approach to the maximum flow problem is to start with the all-zero flow and greedily produce flows with ever-higher value. The natural way to proceed from one to the next is to send more flow on some path from s to t (cf., Figure 1(b)).

[3] A flow corresponds to a steady-state solution, with a constant rate of arrivals at s and departures at t. The model does not capture the time at which flow reaches different vertices. However, it's not hard to extend the model to also capture temporal aspects as well.

A Naive Greedy Algorithm

initialize $f_e = 0$ for all $e \in E$
repeat
  search for an s-t path P such that $f_e < u_e$ for every $e \in P$
  `// takes` $O(|E|)$ `time using BFS or DFS`
  if no such path **then**
    halt with current flow $\{f_e\}_{e \in E}$
  else
    let $\Delta = \underbrace{\min_{e \in P} \overbrace{(u_e - f_e)}^{\text{room on } e}}_{\text{room on } P}$
    for all edges e of P **do**
      increase f_e by Δ

Note that the path search just needs to determine whether or not there is an s-t path in the subgraph of edges e with $f_e < u_e$. This is easily done in linear time using your favorite graph search subroutine, such as breadth-first or depth-first search. There may be many such paths; for now, we allow the algorithm to choose one arbitrarily. The algorithm then pushes as much flow as possible on this path, subject to capacity constraints.

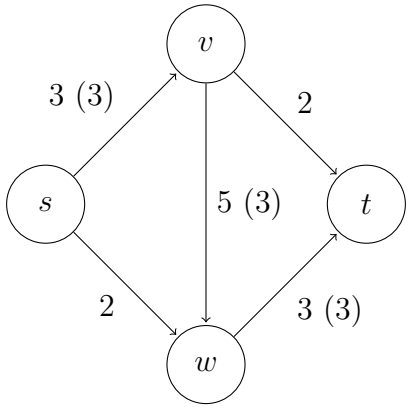

Figure 3: Greedy algorithm returns suboptimal result if first path picked is s-v-w-t.

This greedy algorithm is natural enough, but does it work? That is, when it terminates with a flow, need this flow be a maximum flow? Our sole example thus far already provides a negative answer (Figure 3). Initially, with the all-zero flow, all s-t paths are fair game. If the algorithm happens to pick the zig-zag path, then $\Delta = \min\{3, 5, 3\} = 3$ and it routes 3 units of flow along the path. This saturates the upper-left and lower-right edges, at which point there is no s-t path such that $f_e < u_e$ on every edge. The algorithm terminates at this

point with a flow with value 3. We already know that the maximum flow value is 5, and we conclude that the naive greedy algorithm can terminate with a non-maximum flow.[4]

2.4 Residual Graphs

The second idea is to extend the naive greedy algorithm by allowing "undo" operations. For example, from the point where this algorithm gets stuck in Figure 3, we'd like to route two more units of flow along the edge (s, w), then *backward* along the edge (v, w), undoing 2 of the 3 units we routed the previous iteration, and finally along the edge (v, t). This would yield the maximum flow of Figure 1(b).

Figure 4: (a) original edge capacity and flow and (b) resultant edges in residual network.

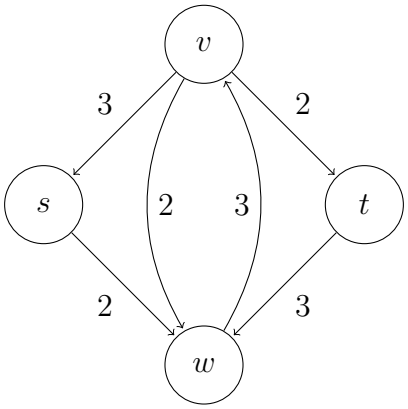

Figure 5: Residual network of flow in Figure 3.

We need a way of formally specifying the allowable "undo" operations. This motivates the following simple but important definition, of a *residual network*. The idea is that, given a graph G and a flow f in it, we form a new flow network G_f that has the same vertex set of G and that has two edges for each edge of G. An edge $e = (v, w)$ of G that carries flow f_e and has capacity u_e (Figure 4(a)) spawns a "forward edge" (u, v) of G_f with capacity $u_e - f_e$ (the room remaining) and a "backward edge" (w, v) of G_f with capacity f_e (the amount

[4]It does compute what's known as a "blocking flow;" more on this next lecture.

of previously routed flow that can be undone). See Figure 4(b).[5] Observe that s-t paths with $f_e < u_e$ for all edges, as searched for by the naive greedy algorithm, correspond to the special case of s-t paths of G_f that comprise only forward edges.

For example, with G our running example and f the flow in Figure 3, the corresponding residual network G_f is shown in Figure 5. The four edges with zero capacity in G_f are omitted from the picture.[6]

2.5 The Ford-Fulkerson Algorithm

Happily, if we just run the natural greedy algorithm in the current residual network, we get a correct algorithm, the *Ford-Fulkerson algorithm*.[7]

Ford-Fulkerson Algorithm

initialize $f_e = 0$ for all $e \in E$
repeat
  search for an s-t path P in the current residual graph G_f such that every edge of P has positive residual capacity
  `// takes` $O(|E|)$ `time using BFS or DFS`
  if no such path **then**
    halt with current flow $\{f_e\}_{e \in E}$
  else
    let $\Delta = \min_{e \in P}$ (e's residual capacity in G_f)
    `// augment the flow` f `using the path` P
    for all edges e of G whose corresponding forward edge is in P **do**
      increase f_e by Δ
    for all edges e of G whose corresponding reverse edge is in P **do**
      decrease f_e by Δ

For example, starting from the residual network of Figure 5, the Ford-Fulkerson algorithm will augment the flow by units along the path $s \to w \to v \to t$. This augmentation produces the maximum flow of Figure 1(b).

We now turn our attention to the correctness of the Ford-Fulkerson algorithm. We'll worry about optimizing the running time in future lectures.

[5] If G already has two edges (v, w) and (w, v) that go in opposite directions between the same two vertices, then G_f will have two parallel edges going in either direction. This is not a problem for any of the algorithms that we discuss.

[6] More generally, when we speak about "the residual graph," we usually mean after all edges with zero residual capacity have been removed.

[7] Yes, it's the same Ford from the Bellman-Ford algorithm.

2.6 Termination

We claim that the Ford-Fulkerson algorithm eventually terminates with a feasible flow. This follows from two invariants, both proved by induction on the number of iterations.

First, the algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow. This is clearly true initially. The parameter Δ is defined so that no flow value f_e becomes negative or exceeds the capacity u_e. For the conservation constraints, consider a vertex v. If v is not on the augmenting path P in G_f, then the flow into and out of v remain the same. If v is on P, with edges (x, v) and (v, w) belonging to P, then there are four cases, depending on whether or not (x, v) and (v, w) correspond to forward or reverse edges. For example, if both are forward edges, then the flow augmentation increases both the flow into and the flow out of v increase by Δ. If both are reverse edges, then both the flow into and the flow out of v decrease by Δ. In all four cases, the flow in and flow out change by the same amount, so conservation constraints are preserved.

Second, the Ford-Fulkerson algorithm maintains the property that every flow amount f_e is an integer. (Recall we are assuming that every edge capacity u_e is an integer.) Inductively, all residual capacities are integral, so the parameter Δ is integral, so the flow stays integral.

Every iteration of the Ford-Fulkerson algorithm increase the value of the current flow by the current value of Δ. The second invariant implies that $\Delta \geq 1$ in every iteration of the Ford-Fulkerson algorithm. Since only a finite amount of flow can escape the source vertex, the Ford-Fulkerson algorithm eventually halts. By the first invariant, it halts with a feasible flow.[8]

Of course, all of this applies equally well to the naive greedy algorithm of Section 2.3. How do we know whether or not the Ford-Fulkerson algorithm can also terminate with a non-maximum flow? The hope is that because the Ford-Fulkerson algorithm has more path eligible for augmentation, it progresses further before halting. But is it guaranteed to compute a maximum flow?

2.7 Optimality Conditions

Answering the following question will be a major theme of the first half of CS261, culminating with our study of linear programming duality.

HOW DO WE KNOW WHEN WE'RE DONE?

For example, given a flow, how do we know if it's a maximum flow? Any correct maximum flow algorithm must answer this question, explicitly or implicitly. If I handed you an allegedly maximum flow, how could I convince you that I'm not lying? It's easy to convince someone that a flow is *not* maximum, just by exhibiting a flow with higher value.

[8] The Ford-Fulkerson algorithm continues to terminate if edges' capacities are rational numbers, not necessarily integers. (Proof: scaling all capacities by a common number doesn't change the problem, so we can clear denominators to reduce the rational capacity case to the integral capacity case.) It is a bizarre mathematical curiosity that the Ford-Fulkerson algorithm need not terminate with edges' capacities are irrational.

Returning to our original example (Figure 1), answering this question didn't seem like a big deal. We exhibited a flow of value 5, and because the total capacity escaping s is only 5, it's clear that there can't be any flow with high value. But what about the network in Figure 6(a)? The flow shown in Figure 6(b) has value only 3. Could it really be a maximum flow?

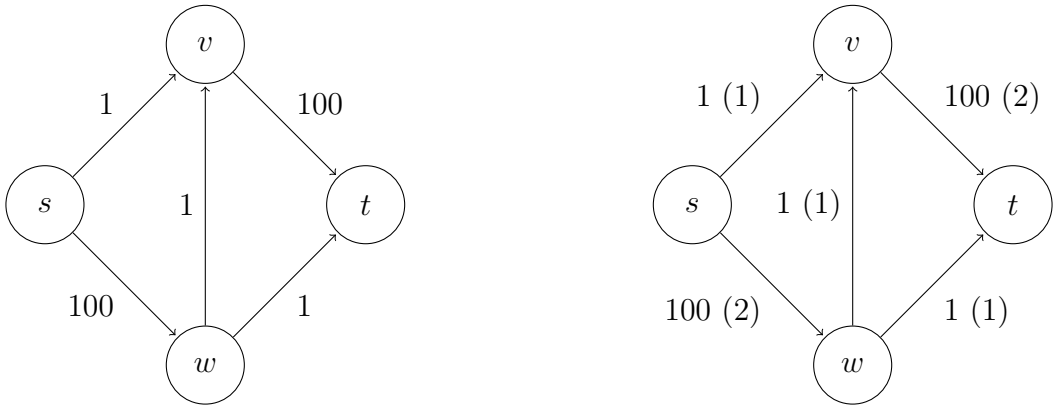

Figure 6: (a) A given network and (b) the alleged maximum flow of value 3.

We'll tackle several fundamental computational problems by following a two-step paradigm.

Two-Step Paradigm

1. Identify "optimality conditions" for the problem. These are sufficient conditions for a feasible solution to be an optimal solution. This step is structural, and not necessarily algorithmic. The optimality conditions vary with the problem, but they are often quite intuitive.

2. Design an algorithm that terminates with the optimality conditions satisfied. Such an algorithm is necessarily correct.

This paradigm is a guide for proving algorithms correct. Correctness proofs didn't get too much airtime in CS161, because almost all of them are straightforward inductions — think of MergeSort, or Dijkstra's algorithm, or any dynamic programming algorithm. The harder problems studied in CS261 demand a more sophisticated and principle approach (with which you'll get plenty of practice).

So how would we apply this two-step paradigm to the maximum flow problem? Consider the following claim.

Claim 2.1 (Optimality Conditions for Maximum Flow) *If f is a flow in G such that the residual network G_f has no s-t path, then the f is a maximum flow.*

This claim implements the first step of the paradigm. The Ford-Fulkerson algorithm, which can only terminate with this optimality condition satisfied, already provides a solution to the second step. We conclude:

Corollary 2.2 *The Ford-Fulkerson algorithm is guaranteed to terminate with a maximum flow.*

Next lecture we'll prove (a generalization of) the claim, derive the famous maximum-flow/minimum-cut problem, and design faster maximum flow algorithms.

CS261: A Second Course in Algorithms
Lecture #10: The Minimax Theorem and Algorithms for Linear Programming*

Tim Roughgarden†

February 4, 2016

1 Zero-Sum Games and the Minimax Theorem

1.1 Rock-Paper Scissors

Recall rock-paper-scissors (or roshambo). Two players simultaneously choose one of rock, paper, or scissors, with rock beating scissors, scissors beating paper, and paper beating rock.[1]

Here's an idea: what if I made you go first? That's obviously unfair — whatever you do, I can respond with the winning move.

But what if I only forced you to commit to a *probability distribution* over rock, paper, and scissors? (Then I respond, then nature flips coins on your behalf.) If you prefer, imagine that you submit your code for a (randomized) algorithm for choosing an action, then I have to choose my action, and then we run your algorithm and see what happens.

In the second case, going first no longer seems to doom you. You can protect yourself by randomizing uniformly among the three options — then, no matter what I do, I'm equally likely to win, lose, or tie. The *minimax theorem* states that, in general games of "pure competition," a player moving first can always protect herself by randomizing appropriately.

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

[1]Here are some fun facts about rock-paper-scissors. There's a World Series of RPS every year, with a top prize of at least $50K. If you watch some videos of them, you will see pure psychological welfare. Maybe this explains why some of the same players seem to end up in the later rounds of the tournament every year.

There's also a robot hand, built at the University of Tokyo, that plays rock-paper-scissors with a winning probability of 100% (check out the video). No surprise, a very high-speed camera is involved.

1.2 Zero-Sum Games

A *zero-sum game* is specified by a real-valued matrix $m \times n$ matrix $\mathbf{A}$. One player, the row player, picks a row. The other (column) player picks a column. Rows and columns are also called *strategies*. By definition, the entry a_{ij} of the matrix $\mathbf{A}$ is the row player's payoff when she chooses row i and the column player chooses column j. The column player's payoff in this case is defined as $-a_{ij}$; hence the term "zero-sum." In effect, a_{ij} is the amount that the column player pays to the row player in the outcome (i, j). (Don't forget, a_{ij} might be negative, corresponding to a payment in the opposite direction.) Thus, the row and column players prefer bigger and smaller numbers, respectively.

The following matrix describes the payoffs in the Rock-Paper-Scissors game in our current language.

	Rock	Paper	Scissors
Rock	0	-1	1
Paper	1	0	-1
Scissors	-1	1	0

1.3 The Minimax Theorem

We can write the expected payoff of the row player when payoffs are given by an $m \times n$ matrix $\mathbf{A}$, the row strategy is $\mathbf{x}$ (a distribution over rows), and the column strategy is $\mathbf{y}$ (a distribution over columns), as

$$\begin{aligned}\sum_{i=1}^{m}\sum_{j=1}^{n}\mathbf{Pr}[\text{outcome } (i,j)]\, a_{ij} &= \sum_{i=1}^{m}\sum_{j=1}^{n}\underbrace{\mathbf{Pr}[\text{row } i \text{ chosen}]}_{=x_i}\cdot\underbrace{\mathbf{Pr}[\text{column } j \text{ chosen}]}_{=y_j}a_{ij}\\ &= \mathbf{x}^\top\mathbf{A}\mathbf{y}.\end{aligned}$$

The first term is just the definition of expectation, and the first equality holds because the row and column players randomize independently. That is, $\mathbf{x}^\top\mathbf{A}\mathbf{y}$ is just the expected payoff to the row player (and negative payoff to the second player) when the row and column strategies are $\mathbf{x}$ and $\mathbf{y}$.

In a two-player zero-sum game, would you prefer to commit to a mixed strategy before or after the other player commits to hers? Intuitively, there is only a first-mover disadvantage, since the second player can adapt to the first player's strategy. The minimax theorem is the amazing statement that *it doesn't matter*.

Theorem 1.1 (Minimax Theorem) *For every two-player zero-sum game* $\mathbf{A}$*,*

$$\max_{\mathbf{x}}\left(\min_{\mathbf{y}}\mathbf{x}^\top\mathbf{A}\mathbf{y}\right) = \min_{\mathbf{y}}\left(\max_{\mathbf{x}}\mathbf{x}^\top\mathbf{A}\mathbf{y}\right). \tag{1}$$

On the left-hand side of (1), the row player moves first and the column player second. The column player plays optimally given the strategy chosen by the row player, and the row

player plays optimally anticipating the column player's response. On the right-hand side of (1), the roles of the two players are reversed. The minimax theorem asserts that, under optimal play, the expected payoff of each player is the same in the two scenarios.

For example, in Rock-Paper-Scissors, both sides of (1) are 0 (with the first player playing uniformly and the second player responding arbitrarily). When a zero-sum game is asymmetric and skewed toward one of the players, both sides of (1) will be non-zero (but still equal). The common number on both sides of (1) is called the *value* of the game.

1.4 From LP Duality to Minimax

Theorem 1.1 was originally proved by John von Neumann in the 1920s, using fixed-point-style arguments. Much later, in the 1940s, von Neumann proved it again using arguments equivalent to strong LP duality (as we'll do here). This second proof is the reason that, when a very nervous George Dantzig (more on him later) explained his new ideas about linear programming and the simplex method to von Neumann, the latter was able, off the top of his head, to immediately give an hour-plus response that outlined the theory of LP duality.

We now proceed to derive Theorem 1.1 from LP duality. The first step is to formalize the problem of computing the best strategy for the player forced to go first.

Looking at the left-hand side (say) of (1), it doesn't seem like linear programming should apply. The first issue is the nested min/max, which is not allowed in a linear program. The second issue is the quadratic (nonlinear) character of $\mathbf{x}^\top \mathbf{A}\mathbf{y}$ in the decision variables $\mathbf{x}, \mathbf{y}$. But we can work these issues out.

A simple but important observation is: the second player never needs to randomize. For example, suppose the row player goes first and chooses a distribution $\mathbf{x}$. The column player can then simply compute the expected payoff of each column (the expectation with respect to $\mathbf{x}$) and choose the best column (deterministically). If multiple columns are tied for the best, the it is also optimal to randomized arbitrarily among these; but there is no need for the player moving second to do so.

In math, we have argued that

$$\begin{aligned} \max_{\mathbf{x}} \left(\min_{\mathbf{y}} \mathbf{x}^T \mathbf{A}\mathbf{y} \right) &= \max_{\mathbf{x}} \left(\min_{j=1}^{n} \mathbf{x}^T \mathbf{A}\mathbf{e}_j \right) \\ &= \max_{\mathbf{x}} \left(\min_{j=1}^{n} \sum_{i=1}^{m} a_{ij} x_i \right), \end{aligned} \tag{2}$$

where $\mathbf{e}_j$ is the jth standard basis vector, corresponding to the column player deterministically choosing column j.

We've solved one of our problems by getting rid of $\mathbf{y}$. But there is still the nested max/min. Here we recall a trick from Lecture #7, that a minimum or maximum can often be simulated by additional variables and constraints. The same trick works here, in exactly the same way.

Specifically, we introduce a decision variable v, intended to be equal to (2), and

$$\max v$$

subject to

$$\begin{aligned} v - \sum_{i=1}^{m} a_{ij} x_i &\leq 0 \qquad \text{for all } j = 1, \ldots, n \\ \sum_{i=1}^{m} x_i &= 1 \\ x_1, \ldots, x_m \geq 0 \quad &\text{and} \quad v \in \mathbb{R}. \end{aligned} \tag{3}$$

Note that this is a linear program. Rewriting the constraints (3) in the form

$$v \leq \sum_{i=1}^{m} a_{ij} x_i \qquad \text{for all } j = 1, \ldots, n$$

makes it clear that they force v to be at most $\min_{j=1}^{n} \sum_{i=1}^{m} a_{ij} x_i$.

We claim that if $(v^*, \mathbf{x}^*)$ is an optimal solution, then $v^* = \min_{j=1}^{n} \sum_{i=1}^{m} a_{ij} x_i$. This follows from the same arguments used in Lecture #7. As already noted, by feasibility, v^* cannot be larger than $\min_{j=1}^{n} \sum_{i=1}^{m} a_{ij} x_i^*$. If it were strictly less, then we can increase v^* slightly without destroying feasibility, yielding a better feasible solution (contradicting optimality).

Since the linear program explicitly maximizes v over all distributions $\mathbf{x}$, its optimal objective function value is

$$v^* = \max_{\mathbf{x}} \left(\min_{j=1}^{n} \mathbf{x}^\top \mathbf{A} \mathbf{e}_j \right) = \max_{\mathbf{x}} \left(\min_{\mathbf{y}} \mathbf{x}^\top \mathbf{A} \mathbf{y} \right). \tag{4}$$

Thus we can compute with a linear program the optimal strategy for the row player, when it moves first, and the expected payoff obtained (assuming optimal play by the column player).

Repeating the exercise for the column player gives the linear program

$$\min w$$

subject to

$$\begin{aligned} w - \sum_{j=1}^{n} a_{ij} y_j &\geq 0 \qquad \text{for all } i = 1, \ldots, m \\ \sum_{j=1}^{n} y_j &= 1 \\ y_1, \ldots, y_n \geq 0 \quad &\text{and} \quad w \in \mathbb{R}. \end{aligned}$$

At an optimal solution $(w^*, \mathbf{y}^*)$, $\mathbf{y}^*$ is the optimal strategy for the column player (when going first, assuming optimal play by the row player) and

$$w^* = \min_{\mathbf{y}} \left(\max_{i=1}^{m} e_i^\top \mathbf{A}\mathbf{y} \right) = \min_{\mathbf{y}} \left(\max_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{y} \right). \tag{5}$$

Here's the punch line: *these two linear programs are duals*. This can be seen by looking up our recipe for taking duals (Lecture #8) and verifying that these two linear programs conform to the recipe (see Exercise Set #5). For example, the one unrestricted variable (v or w) corresponds to the one equality constraint in the other linear program ($\sum_{j=1}^{n} y_j = 1$ or $\sum_{i=1}^{m} x_i = 1$, respectively).

Strong duality implies that $v^* = w^*$; in light of (4) and (5), the minimax theorem follows directly.[2]

2 Survey of Linear Programming Algorithms

We've established that linear programs capture lots of different problems that we'd like to solve. So how do we efficiently solve a linear program?

2.1 The High-Order Bit

If you only remember one thing about linear programming, make it this:

> Linear programs can be solved efficiently, in both theory and practice.

By "in theory," we mean that linear programs can be solved in polynomial time in the worst-case. By "in practice," we mean that commercial solvers routinely solve linear programs with input size in the millions. (Warning: the algorithms used in these two cases are not necessarily the same.)

2.2 The Simplex Method

2.2.1 Backstory

In 1947 George Dantzig developed both the general formalism of linear programming and also the first general algorithm for solving linear programs, the *simplex method*.[3] Amazingly, the simplex method remains the dominant paradigm today for solving linear programs.

[2] The minimax theorem is obviously interesting its own right, and it also has applications in algorithms, specifically to proving lower bounds on what randomized algorithms can do.

[3] Dantzig spent the final 40 years of his career at Stanford (1966-2005). You've probably heard the story about a student who is late to class, sees two problems written on the blackboard, assumes they're homework problems, and then goes home and solves them, not realizing that they are the major open questions in the field. (A partial inspiration for *Good Will Hunting*, among other things.) Turns out this story is not apocryphal: it was Dantzig, as a PhD student in the late 1930s, in a statistics course at UC Berkeley.

2.2.2 Geometry

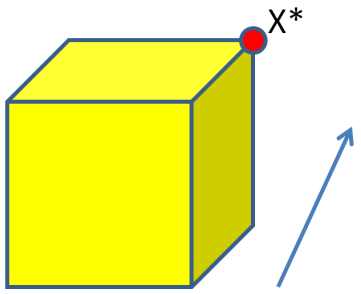

Figure 1: Illustration of a feasible set and an optimal solution x^*. We know that there always exists an optimal solution at a vertex of the feasible set, in the direction of the objective function.

In Lecture #7 we developed geometric intuition about what it means to solve a linear program, and one of our findings was that there is always an optimal solution at a vertex (i.e., "corner") of the feasible region (e.g., Figure 1).[4] This observation implies a finite (but bad) algorithm for linear programming. (This is not trivial, since there are an infinite number of feasible solutions.) The reason is that every vertex satisfies at least n constraints with equality (where n is the number of decision variables). Or contrapositively: for a feasible solution $\mathbf{x}$ that satisfies at most $n-1$ constraints with equality, there is a direction along which moving $\mathbf{x}$ continues to satisfy these constraints, and moving $\mathbf{x}$ locally in either direction on this line yields two feasible points whose midpoint is $\mathbf{x}$. But a vertex of a feasible region cannot be written as a non-trivial convex combination of other feasible points.[5] See also Exercise Set #5. The finite algorithm is then: enumerate all (finitely many) subsets of n linearly independent constraints, check if the unique point of $\mathbb{R}^n$ that satisfies all of them is a feasible solution to the linear program, and remember the best feasible solution found in this way.

The simplex algorithm also searches through the vertices of the feasible region, but does so in a smarter and more principled way. The basic idea is to use local search — if there is a "neighboring" vertex which is better, move to it, otherwise halt. The idea of neighboring vertices should be clear from Figure 1 — two endpoints of an "edge" of the feasible region. In general, we can define two different vertices to be neighboring if and only if they satisfy $n-1$ common constraints with equality. Moving from one vertex to a neighbor then just involves swapping out one of the old tight constraints for a new tight constraint; each such swap (also called a *pivot*) corresponds to a "move" along an edge of the feasible region.[6]

[4] There are a few edge cases, including unbounded or empty feasible regions, which can be handled and which we'll ignore here.

[5] Making all of this completely precise is somewhat annoying. But everything your geometric intuition suggests about these statements is indeed true.

[6] One important issue is "degeneracy," meaning a vertex that satisfies strictly more than n constraints

In an iteration of the simplex method, the current vertex may have multiple neighboring vertices with better objective function value. The choice of which of these to move to is known as a *pivot rule*.

2.2.3 Correctness

The simplex method is guaranteed to terminate at an optimal solution.[7] The intuition for this fact should be clear from Figure 1 — since the objective function is linear and the feasible region is convex, if no "local move" from a vertex is improving, then there should be no direction at all within the feasible region that leads to a better solution. Formally, the simplex method "knows that it's done" by, at termination, exhibiting a a feasible dual solution such that the complementary slackness conditions hold (see Lecture #9). Indeed, the proof that the simplex method is guaranteed to terminate with an optimal solution provides another proof of strong LP duality.

In terms of our three-step design paradigm (Lecture #9), we can think of the simplex method as maintaining primal feasibility and the complementary slackness conditions and working toward dual feasibility.[8]

2.2.4 Worst-Case Running Time

As mentioned, the simplex method is very fast in practice, and routinely solves linear programs with hundreds of thousands or even millions of variables and constraints. However, it is a bizarre mathematical fact that the worst-case running time of the simplex method is exponential in the input size. To understand the issue, first note that the number of vertices of a feasible region can be exponential in the dimension (e.g., the 2^n vertices of the n-dimensional hypercube). Much harder is constructing a linear program where the simplex method actually visits all of the vertices of the feasible region. Such an example was given by Klee and Minty in the early 1970s (25 years after simplex was invented). Their example is a "squashed" version of an n-dimensional hypercube. Such exponential lower bounds are known for all natural deterministic pivot rules.[9]

The number of iterations required by the simplex method is also related to one of the most famous open problems in combinatorial geometry, the *Hirsch conjecture*. This conjecture concerns the "diameter of polytopes," meaning the diameter of the graph derived from the

with equality. (E.g., in the plane, this would be 3 constraints whose boundaries meet at a common point.) In this case, a constraint swap can result in staying at the same vertex. There are simple ways to avoid cycling, however, which we won't discuss here.

[7] Assuming that the linear program is feasible and has a finite optimum. If not, the simplex method correctly detects which of these cases the linear program falls in.

[8] How does the simplex method find the initial primal feasible point? For some linear programs this is easy (e.g., the all-0 vector is feasible). In general, one can add an additional variable, highly penalized in the objective function, to make finding an initial feasible point trivial.

[9] Interestingly, some randomized pivot rules (e.g., among the neighboring vertices that are better, pick one at random) require, in expectation, at most $\approx 2^{\sqrt{n}}$ iterations to converge on every instance. There are now nearly matching upper and lower bounds on the required number of iterations for all the natural randomized rules.

skeleton of the polytope (with vertices and edges of the polytope inducing, um, vertices and edges of the graph). The conjecture asserts that the diameter is always at most linear (in the number of variables and constraints). The best known upper bound on the worst-case diameter of polytopes is "quasi-polynomial" (of the form $\approx n^{\log n}$), due to Kalai and Kleitman in the early 1990s. Since the trajectory of the simplex method is a walk along the edges of the feasible region, the number of iterations required (for a worst-case starting point and objective function) is at least the polytope diameter. Put differently, sufficiently good upper bounds on the number of iterations required by the simplex method (for some pivot rule) would automatically yield progress on the Hirsch conjecture.

2.2.5 Average-Case and Smoothed Running Time

The worst-case running time of the wildly practical simplex method poses a real quandary for the mathematical analysis of algorithms. Can we "correct" the theory so that it better reflects reality?

In the 1980s, a number of researchers (Borgwardt, Smale, Adler-Karp, etc.) showed that the simplex method (with a suitable pivot rule) runs in polynomial time "on average" with respect to various distributions over linear programs. Note that it is not at all obvious how to define a "random linear program." Indeed, many natural attempts lead to linear programs that are almost always infeasible.

At the start of the 21st century, Spielman and Teng proved that the simplex method has polynomial "smoothed complexity." This is like a robust version of an average-cases analysis. The model is to take a worst-case initial linear program, and then to randomly perturb it a small amount. The main result here is that, for every initial linear program, in expectation over the perturbed version of the linear program, the running time of simplex is polynomial in the input size. The take-away being that bad examples for the simplex method are both rare and isolated, in a precise sense. See the instructor's CS264 course ("Beyond Worst-Case Analysis") for much more on smoothed analysis.

2.3 The Ellipsoid Method

2.3.1 Worst-Case Running Time

The *ellipsoid method* was originally proposed (by Shor and others) in the early/mid-1970s as an algorithm for nonlinear programming. In 1979 Khachiyan proved that, for linear programs, the algorithm is actually guaranteed to run in polynomial time. This was the first-ever polynomial-time algorithm for linear programming, a big enough deal at the time to make the front page of the New York Times (if below the fold).

The ellipsoid method is very slow in practice — usually multiple orders of magnitude slower than the fastest methods. How can a polynomial-time algorithm be so much worse than the exponential-time simplex method? There are two issues. First, the degree in the polynomial bounding the ellipsoid method's running time is pretty big (like 4 or 5, depending on the implementation details). Second, the performance of the ellipsoid method

on "typical cases" is generally close to its worst-case performance. This is in sharp contrast to the simplex method, which almost always solves linear programs in time far less than its worst-case (exponential) running time.

2.3.2 Separation Oracles

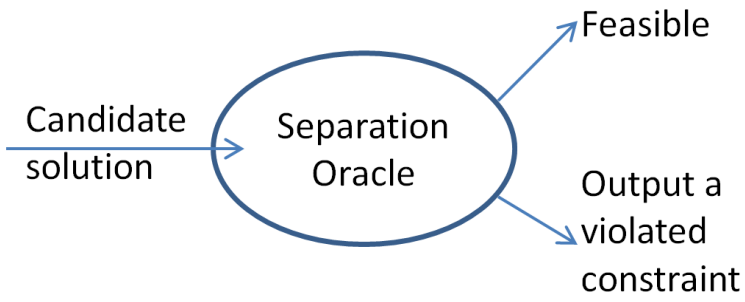

Figure 2: The responsibility of a separation oracle.

The ellipsoid method is uniquely useful for proving theorems — for establishing that other problems are worst-case polynomial-time solvable, and thus are at least efficiently solvable in principle. The reason is that the ellipsoid method can solve some linear programs with n variables and an exponential (in n) number of constraints in time polynomial in n. How is this possible? Doesn't it take exponential time just to read in all of the constraints? For other linear programming algorithms, yes. But the ellipsoid method doesn't need an explicit description of the linear program — all it needs is a helper subroutine known as a *separation oracle*. The responsibility of a separation oracle is to take as input an allegedly feasible solution $\mathbf{x}$ to a linear program, and to either verify feasibility (if $\mathbf{x}$ is indeed feasible) or produce a constraint violated by $\mathbf{x}$ (otherwise). See Figure 2. Of course, the separation oracle should also run in polynomial time.[10]

How could one possibly check an exponential number of constraints in polynomial time? You've actually already seen some examples of this. For example, recall the dual of the path-based linear programming formulation of the maximum flow problem (Lecture #8):

$$\min \sum_{e \in E} u_e \ell_e$$

[10]Such separation oracles are also useful in some practical linear programming algorithms: in "cutting plane methods," for linear programs with a large number of constraints (where the oracle is used in the same way as in the ellipsoid method); and in the simplex method for linear programs with a large number of variables (where the oracle is used to generate *variables* on the fly, a technique called "column generation").

subject to

$$\sum_{e \in P} \ell_e \geq 1 \qquad \text{for all } P \in \mathcal{P} \tag{6}$$

$$\ell_e \geq 0 \qquad \text{for all } e \in E.$$

Here $\mathcal{P}$ denotes the set of s-t flow paths of a maximum flow instance (with edge capacities u_e). Since a graph can have an exponential number of s-t paths, this linear program has a potentially exponential number of constraints.[11] But, it has a polynomial-time separation oracle. The key observation is: at least one constraint is violated if and only if

$$\min_{P \in \mathcal{P}} \sum_{e \in P} \ell_e < 1.$$

Thus, the separation oracle is just Dijkstra's algorithm! In detail: given an allegedly feasible solution $\{\ell_e\}_{e \in E}$ to the linear program, the separation oracle first checks that each ℓ_e is nonnegative (if $\ell_e < 0$, it returns the violated constraint $\ell_e \geq 0$). If the solution passes this test, then the separation oracle runs Dijkstra's algorithm to compute a shortest s-t path, using the ℓ_e's as (nonnegative) edge lengths. If the shortest path has length at least 1, then all of the constraints (6) are satisfied and the oracle reports "feasible." If the shortest path P^* has length less than 1, then it returns the violated constraint $\sum_{e \in P^*} \ell_e \geq 1$. Thus, we can solve the above linear program in polynomial time using the ellipsoid method.[12]

2.3.3 How the Ellipsoid Method Works

Here is a sketch of how the ellipsoid method works. The first step is to reduce optimization to feasibility. That is, if the objective is $\max \mathbf{c}^T\mathbf{x}$, one replaces the objective function by the constraint $\mathbf{c}^T\mathbf{x} \geq M$ for some target objective function value M. If one can solve this feasibility problem in polynomial time, then one can solve the original optimization problem using binary search on the target objective M.

There's a silly story about how to hunt a lion in the Sahara. The solution goes: encircle the Sahara with a high fence and then bifurcate it with another fence. Figure out which side has the lion in it (e.g., looking for tracks), and recurse. Eventually, the lion is trapped in such a small area that you know exactly where it is.

[11] For example, consider the graph $s = v_1, v_2, \ldots, v_n = t$, with two parallel edges directed from each v_i to v_{i+1}.

[12] Of course, we already know how to solve this particular linear program in polynomial time — just compute a minimum s-t cut (see Lecture #8). But there are harder problems where the only known proof of polynomial-time solvability goes through the ellipsoid method.

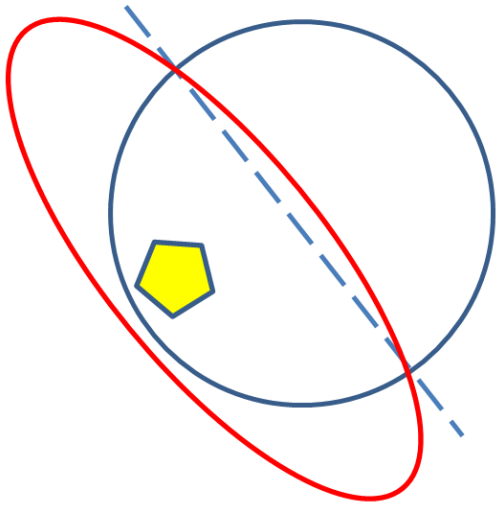

Figure 3: The ellipsoid method first initializes a huge sphere (blue circle) that encompasses the feasible region (yellow pentagon). If the ellipsoid center is not feasible, the separation oracle produces a violated constraint (dashed line) that splits the ellipsoid into two regions, one containing the feasible region and one that does not. A new ellipsoid (red oval) is drawn that contains the feasible half-ellipsoid, and the method continues recursively.

Believe it or not, this story is a pretty good cartoon of how the ellipsoid method works. The ellipsoid method maintains at all times an ellipsoid which is guaranteed to contain the entire feasible region (Figure 3). It starts with a huge sphere to ensure the invariant at initialization. It then invokes the separation oracle on the center of the current ellipsoid. If the ellipsoid center is feasible, then the problem is solved. If not, the separation oracle produces a constraint satisfied by all feasible points that is violated by the ellipsoid center. Geometrically, the feasible region and the ellipsoid center are on opposite sides of the corresponding halfspace boundary (Figure 3). Thus we know we can recurse on the appropriate half-ellipsoid. Before recursing, however, the ellipsoid method redraws a new ellipsoid that contains this half-ellipsoid (and hence the feasible region).[13] Elementary but tedious calculations show that the volume of the current ellipsoid is guaranteed to shrink at a certain rate at each iteration, and this yields a polynomial bound on the number of iterations required. The algorithm stops when the current ellipsoid is so small that it cannot possibly contain a feasible point (given the precision of the input data).

Now that we understand how the ellipsoid method works at a high level, we see why it can solve linear programs with an exponential number of constraints. It never works with an explicit description of the constraints, and just generates constraints on the fly on a "need to know" basis. Because it terminates in a polynomial number of iterations, it only ever

[13] Why the obsession with ellipsoids? Basically, they are the simplest shapes that can decently approximate all shapes of polytopes ("fat" ones, "skinny" one, etc.). In particular, every ellipsoid has a well defined and easy-to-compute center.

generates a polynomial number of constraints.[14]

2.4 Interior-Point Methods

While the simplex method works "along the boundary" of the feasible region, and the ellipsoid method works "outside in," the third and again quite different paradigm of *interior-point methods* works "inside out." There are many genres of interior-point methods, beginning with Karmarkar's algorithm in 1984 (which again made the New York Times, this time above the fold). Perhaps the most popular are "central path" methods. The idea is, instead of maximizing the given objective $\mathbf{c}^T\mathbf{x}$, to maximize

$$\mathbf{c}^T\mathbf{x} - \lambda \cdot \underbrace{f(\text{distance between } \mathbf{x} \text{ and boundary})}_{\text{barrier function}},$$

where $\lambda \geq 0$ is a parameter and f is a "barrier function" that blows up (to $+\infty$) as its argument goes to 0 (e.g., $\log \frac{1}{z}$). Initially, one sets λ so big that the problem becomes easy (when $f(x) = \log \frac{1}{z}$, the solution is the "analytic center" of the feasible region, and can be computed using e.g. Newton's method). Then one gradually decreases the parameter λ, tracking the corresponding optimal point along the way. (The "central path" is the set of optimal points as λ varies from ∞ to 0.) When $\lambda = 0$, the optimal point is an optimal solution to the linear program, as desired.

The two things you should know about interior-point methods are: (i) many such algorithms run in time polynomial in the worst case; and (ii) such methods are also competitive with the simplex method in practice. For example, one of Matlab's LP solvers uses an interior-point algorithm.

There are many linear programs where interior-point methods beat the best simplex codes (especially on larger LPs), but also vice versa. There is no good understanding of when one is likely to outperform the other. Despite the fact that it's 70 years old, the simplex method remains the most commonly used linear programming algorithm in practice.

[14] As a sanity check, recall that every vertex of a feasible region in $\mathbb{R}^n$ is the unique point satisfying some subset of n constraints with equality. Thus in principle there's always n constraints the are sufficient to describe one feasible point (given a separation oracle to verify feasibility). The magic of the ellipsoid method is that, even though a priori it has no idea which subset of constraints is the right one, it always finds a feasible point while generating only a polynomial number of constraints.

CS261: A Second Course in Algorithms
Lecture #11: Online Learning and the Multiplicative Weights Algorithm*

Tim Roughgarden†

February 9, 2016

1 Online Algorithms

This lecture begins the third module of the course (out of four), which is about *online algorithms*. This term was coined in the 1980s and sounds anachronistic there days — it has nothing to do with the Internet, social networks, etc. It refers to computational problems of the following type:

An Online Problem

1. The input arrives "one piece at a time."
2. An algorithm makes an irrevocable decision each time it receives a new piece of the input.

For example, in job scheduling problems, one often thinks of the jobs as arriving online (i.e., one-by-one), with a new job needing to be scheduled on some machine immediately. Or in a graph problem, perhaps the vertices of a graph show up one by one (with whatever edges are incident to previously arriving vertices). Thus the meaning of "one piece at a time" varies with the problem, but it many scenarios it makes perfect sense. While online algorithms don't get any airtime in an introductory course like CS161, many problems in the real world (computational and otherwise) are inherently online problems.

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

2 Online Decision-Making

2.1 The Model

Consider a set A of $n \geq 2$ actions and a time horizon $T \geq 1$. We consider the following setup.

Online Decision-Making

At each time step $t = 1, 2, \ldots, T$:

- a decision-maker picks a probability distribution $\mathbf{p}^t$ over her actions A
- an adversary picks a reward vector $\mathbf{r}^t : A \to [-1, 1]$
- an action a^t is chosen according to the distribution $\mathbf{p}^t$, and the decision-maker receives reward $r^t(a^t)$
- the decision-maker learns $\mathbf{r}^t$, the entire reward vector

An *online decision-making algorithm* specifies for each t the probability distribution $\mathbf{p}^t$, as a function of the reward vectors $\mathbf{r}^1, \ldots, \mathbf{r}^{t-1}$ and realized actions $a^1, \ldots, a^{t-1}$ of the first $t-1$ time steps. An *adversary* for such an algorithm $\mathcal{A}$ specifies for each t the reward vector $\mathbf{r}^t$, as a function of the probability distributions $\mathbf{p}^1, \ldots, \mathbf{p}^t$ used by $\mathcal{A}$ on the first t days and the realized actions $a^1, \ldots, a^{t-1}$ of the first $t-1$ days.

For example, A could represent different investment strategies, different driving routes between home and work, or different strategies in a zero-sum game.

2.2 Definitions and Examples

We seek a "good" online decision-making algorithm. But the setup seems a bit unfair, no? The adversary is allowed to choose each reward function $\mathbf{r}^t$ *after* the decision-maker has committed to her probability distribution $\mathbf{p}^t$. With such asymmetry, what kind of guarantee can we hope for? This section gives three examples that establish limitations on what is possible.[1]

The first example shows that there is no hope of achieving reward close to that of the best action sequence in hindsight. This benchmark $\sum_{t=1}^{T} \max_{a \in A} r^t(a)$ is just too strong.

Example 2.1 (Comparing to the Best Action Sequence) Suppose $A = \{1, 2\}$ and fix an arbitrary online decision-making algorithm. Each day t, the adversary chooses the reward vector $\mathbf{r}^t$ as follows: if the algorithm chooses a distribution $\mathbf{p}^t$ for which the probability on action 1 is at least $\frac{1}{2}$, then $\mathbf{r}^t$ is set to the vector $(-1, 1)$. Otherwise, the adversary sets $\mathbf{r}^t$ equal

[1] In the first half of the course, we always sought algorithms that are always correct (i.e., optimal). In an online setting, where you have to make decisions without knowing the future, we expect to compromise on an algorithm's guarantee.

to $(1, -1)$. This adversary forces the expected reward of the algorithm to be nonpositive, while ensuring that the reward of the best action sequence in hindsight is T.

Example 2.1 motivates the following important definitions. Rather than comparing the expected reward of an algorithm to that of the best action *sequence* in hindsight, we compare it to the reward incurred by the best *fixed action* in hindsight. In words, we change our benchmark from $\sum_{t=1}^{T} \max_{a \in A} r^t(a)$ to $\max_{a \in A} \sum_{t=1}^{T} r^t(a)$.

Definition 2.2 (Regret) Fix reward vectors $\mathbf{r}^1, \ldots, \mathbf{r}^T$. The *regret* of the action sequence $a^1, \ldots, a^T$ is

$$\underbrace{\max_{a \in A} \sum_{t=1}^{T} r^t(a)}_{\text{best fixed action}} - \underbrace{\sum_{t=1}^{T} r^t(a^t)}_{\text{our algorithm}}. \tag{1}$$

We'd like an online decision-making algorithm that achieves low regret, as close to 0 as possible (and negative regret would be even better).[2] Notice that the worst-possible regret in $2T$ (since rewards lie in $[-1, 1]$). We think of regret $\Omega(T)$ as an epic fail for an algorithm.

What is the justification for the benchmark of the best fixed action in hindsight? First, simple and natural learning algorithms can compete with this benchmark. Second, achieving this is non-trivial: as the following examples make clear, some ingenuity is required. Third, competing with this benchmark is already sufficient to obtain many interesting applications (see end of this lecture and all of next lecture).

One natural online decision-making algorithm is *follow-the-leader* , which at time step t chooses the action a with maximum cumulative reward $\sum_{u=1}^{t-1} r^u(a)$ so far. The next example shows that follow-the-leader, and more generally every deterministic algorithm, can have regret that grows linearly with T.

Example 2.3 (Randomization Is Necessary for No Regret) Fix a deterministic online decision-making algorithm. At each time step t, the algorithm commits to a single action a^t. The obvious strategy for the adversary is to set the reward of action a^t to 0, and the reward of every other action to 1. Then, the cumulative reward of the algorithm is 0 while the cumulative reward of the best action in hindsight is at least $T(1 - \frac{1}{n})$. Even when there are only 2 actions, for arbitrarily large T, the worst-case regret of the algorithm is at least $\frac{T}{2}$.

For randomized algorithms, the next example limits the rate at which regret can vanish as the time horizon T grows.

Example 2.4 ($\sqrt{(\ln n)/T}$ Regret Lower Bound) Suppose there are $n = 2$ actions, and that we choose each reward vector $\mathbf{r}^t$ independently and equally likely to be $(1, -1)$ or $(-1, 1)$. No matter how smart or dumb an online decision-making algorithm is, with respect to this random choice of reward vectors, its expected reward at each time step is exactly 0 and its

[2]Sometimes this goal is referred to as "combining expert advice" — if we think of each action as an "expert," then we want to do as well as the best expert.

expected cumulative reward is thus also 0. The expected cumulative reward of the best fixed action in hindsight is $b\sqrt{T}$, where b is some constant independent of T. This follows from the fact that if a fair coin is flipped T times, then the expected number of heads is $\frac{T}{2}$ and the standard deviation is $\frac{1}{2}\sqrt{T}$.

Fix an online decision-making algorithm $\mathcal{A}$. A random choice of reward vectors causes $\mathcal{A}$ to experience expected regret at least $b\sqrt{T}$, where the expectation is over both the random choice of reward vectors and the action realizations. At least one choice of reward vectors induces an adversary that causes $\mathcal{A}$ to have expected regret at least $b\sqrt{T}$, where the expectation is over the action realizations.

A similar argument shows that, with n actions, the expected regret of an online decision-making algorithm cannot grow more slowly than $b\sqrt{T \ln n}$, where $b > 0$ is some constant independent of n and T.

3 The Multiplicative Weights Algorithm

We now give a simple and natural algorithm with optimal worst-case expected regret, matching the lower bound in Example 2.4 up to constant factors.

Theorem 3.1 *There is an online decision-making algorithm that, for every adversary, has expected regret at most* $2\sqrt{T \ln n}$.

An immediately corollary is that the number of time steps needed to drive the expected time-averaged regret down to a small constant is only logarithmic in the number of actions.[3]

Corollary 3.2 *There is an online decision-making algorithm that, for every adversary and* $\epsilon > 0$, *has expected time-averaged regret at most* ϵ *after at most* $(4 \ln n)/\epsilon^2$ *time steps.*

In our applications in this and next lecture, we will use the guarantee in the form of Corollary 3.2.

The guarantees of Theorem 3.1 and Corollary 3.2 are achieved by the *multiplicative weights (MW)* algorithm.[4] Its design follows two guiding principles.

No-Regret Algorithm Design Principles

1. Past performance of actions should guide which action is chosen at each time step, with the probability of choosing an action increasing in its cumulative reward. (Recall from Example 2.3 that we need a randomized algorithm to have any chance.)

[3] Time-averaged regret just means the regret, divided by T.

[4] This and closely related algorithms are sometimes called the multiplicative weight update (MWU) algorithm, Polynomial Weights, Hedge, and Randomized Weighted Majority.

2. The probability of choosing a poorly performing action should decrease at an exponential rate.

The first principle is essential for obtaining regret sublinear in T, and the second for optimal regret bounds.

The MW algorithm maintains a weight, intuitively a "credibility," for each action. At each time step the algorithm chooses an action with probability proportional to its current weight. The weight of each action evolves over time according to the action's past performance.

Multiplicative Weights (MW) Algorithm

initialize $w^1(a) = 1$ for every $a \in A$

for each time step $t = 1, 2, \ldots, T$ **do**

use the distribution $\mathbf{p}^t := \mathbf{w}^t/\Gamma^t$ over actions, where $\Gamma^t = \sum_{a \in A} w^t(a)$ is the sum of the weights

given the reward vector $\mathbf{r}^t$, for every action $a \in A$ use the formula $w^{t+1}(a) = w^t(a) \cdot (1 + \eta r^t(a))$ to update its weight

For example, if all rewards are either -1 or 1, then the weight of each action a either goes up by a $1 + \eta$ factor or down by a $1 - \eta$ factor. The parameter η lies between 0 and $\frac{1}{2}$, and is chosen at the end of the proof of Theorem 3.1 as a function of n and T. For intuition, note that when η is close to 0, the distributions $\mathbf{p}^t$ will hew close to the uniform distribution. Thus small values of η encourage exploration. Large values of η correspond to algorithms in the spirit of follow-the-leader. Thus large values of η encourage exploitation, and η is a knob for interpolating between these two extremes. The MW algorithm is obviously simple to implement, since the only requirement is to update the weight of each action at each time step.

4 Proof of Theorem 3.1

Fix a sequence $\mathbf{r}^1, \ldots, \mathbf{r}^T$ of reward vectors.[5] The challenge is that the two quantities that we care about, the expected reward of the MW algorithm and the reward of the best fixed action, seem to have nothing to do with each other. The fairly inspired idea is to relate both of these quantities to an intermediate quantity, namely the sum $\Gamma^{T+1} = \sum_{a \in A} w^{T+1}(a)$ of the actions' weights at the conclusion of the MW algorithm. Theorem 3.1 then follows from some simple algebra and approximations.

[5] We're glossing over a subtle point, the difference between "adaptive adversaries" (like those defined in Section 2) and "oblivious adversaries" which specify all reward vectors in advance. Because the behavior of the MW algorithm is independent of the realized actions, it turns out that the worst-case adaptive adversary for the algorithm is in fact oblivious.

The first step, and the step which is special to the MW algorithm, shows that the sum of the weights Γ^t evolves together with the expected reward earned by the MW algorithm. In detail, denote the expected reward of the MW algorithm at time step t by ν^t, and write

$$\nu^t = \sum_{a \in A} p^t(a) \cdot r^t(a) = \sum_{a \in A} \frac{w^t(a)}{\Gamma^t} \cdot r^t(a). \tag{2}$$

Thus we want to lower bound the sum of the ν^t's.

To understand Γ^{t+1} as a function of Γ^t and the expected reward (2), we derive

$$\begin{aligned} \Gamma^{t+1} &= \sum_{a \in A} w^{t+1}(a) \\ &= \sum_{a \in A} w^t(a) \cdot (1 + \eta r^t(a)) \\ &= \Gamma^t(1 + \eta \nu^t). \end{aligned} \tag{3}$$

For convenience, we'll bound from above this quantity, using the fact that $1 + x \leq e^x$ for all real-valued x.[6] Then we can write

$$\Gamma^{t+1} \leq \Gamma^t \cdot e^{\eta \nu^t}$$

for each t and hence

$$\Gamma^{T+1} \leq \underbrace{\Gamma^1}_{=n} \prod_{t=1}^{T} e^{\eta \nu^t} = n \cdot e^{\eta \sum_{t=1}^{T} \nu^t}. \tag{4}$$

This expresses a lower bound on the expected reward of the MW algorithm as a relatively simple function of the intermediate quantity Γ^{T+1}.

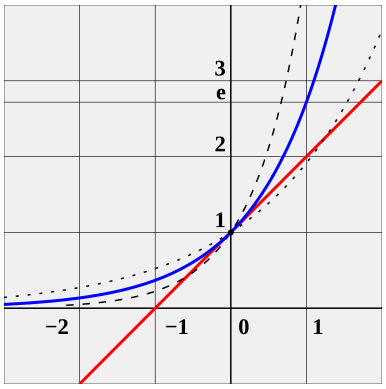

Figure 1: $1 + x \leq e^x$ for all real-valued x.

[6] See Figure 1 for a proof by picture. A formal proof is easy using convexity, a Taylor expansion, or other methods.

The second step is to show that if there is a good fixed action, then the weight of this action single-handedly shows that the final value Γ^{T+1} is pretty big. Combining with the first step, this will imply the the MW algorithm only does poorly if every fixed action is bad.

Formally, let OPT denote the cumulative reward $\sum_{t=1}^{T} r^t(a^*)$ of the best fixed action a^* for the reward vector sequence. Then,

$$\begin{aligned} \Gamma^{T+1} &\geq w^{T+1}(a^*) \\ &= \underbrace{w^1(a^*)}_{=1} \prod_{t=1}^{T} (1 + \eta r^t(a^*)). \end{aligned} \tag{5}$$

OPT is the sum of the $r^t(a^*)$'s, so we'd like to massage the expression above to involve this sum. Products become sums in exponents. So the first idea is to use the same trick as before, replacing $1 + x$ by e^x. Unfortunately, we can't have it both ways — before we wanted an upper bound on $1 + x$, whereas now we want a lower bound. But looking at Figure 1, it's clear that the two function are very close to each other for x near 0. This can made precise through the Taylor expansion

$$\ln(1 + x) = x - \tfrac{x^2}{2} + \tfrac{x^3}{3} - \tfrac{x^4}{4} + \cdots .$$

Provided $|x| \leq \frac{1}{2}$, we can obtain a lower bound on $\ln(1 + x)$ by throwing out all terms but the first two, and doubling the second term to compensate. (The magnitudes of the rest of the terms can be bounded above by the geometric series $\frac{x^2}{2}(\frac{1}{2} + \frac{1}{4} + \cdots)$, so the extra $-\frac{x^2}{2}$ term blows them all away.)

Since $\eta \leq \frac{1}{2}$ and $|r^t(a^*)| \leq 1$ for every t, we can plug this estimate into (5) to obtain

$$\begin{aligned} \Gamma^{T+1} &\geq \prod_{t=1}^{T} e^{\eta r^t(a^*) - \eta^2 (r^t(a*))^2} \\ &\geq e^{\eta OPT - \eta^2 T}, \end{aligned} \tag{6}$$

where in (6) we're just using the crude estimate $(r^t(a^*))^2 \leq 1$ for all t.

Through (4) and (6), we've connected the cumulative expected reward $\sum_{t=1}^{T} \nu^t$ of the MW algorithm with the reward OPT of the best fixed auction through the intermediate quantity Γ^{T+1}:

$$n \cdot e^{\eta \sum_{t=1}^{T} \nu^t} \geq \Gamma^{T+1} \geq e^{\eta OPT - \eta^2 T}$$

and hence (taking the natural logarithm of both sides and dividing through by η):

$$\sum_{t=1}^{T} \nu^t \geq OPT - \eta T - \frac{\ln n}{\eta}. \tag{7}$$

Finally, we set the free parameter η. There are two error terms in (7), the first one corresponding to inaccurate learning (higher for larger learning rates), the second corresponding to overhead before converging (higher for smaller learning rates). To equalize the two terms,

we choose $\eta = \sqrt{(\ln n)/T}$. (Or $\eta = \frac{1}{2}$, if this is smaller.) Then, the cumulative expected reward of the MW algorithm is at most $2\sqrt{T \ln n}$ less than the cumulative reward of the best fixed action. This completes the proof of Theorem 3.1.

Remark 4.1 (Unknown Time Horizons) The choice of η above assumes knowledge of the time horizon T. Minor modifications extend the multiplicative weights algorithm and its regret guarantee to the case where T is not known a priori, with the "2" in Theorem 3.1 replaced by a modestly larger constant factor.

5 Minimax Revisited

Recall that a two-player zero-sum game can be specified by an $m \times n$ matrix $\mathbf{A}$, where a_{ij} denotes the payoff of the row player and the negative payoff of the column player when row i and column j are chosen. It is easy to see that going first in a zero-sum game can only be worse than going second — in the latter case, a player has the opportunity to adapt to the first player's strategy. Last lecture we derived the minimax theorem from strong LP duality. It states that, provided the players randomize optimally, it makes no difference who goes first.

Theorem 5.1 (Minimax Theorem) *For every two-player zero-sum game* $\mathbf{A}$*,*

$$\max_{\mathbf{x}} \left(\min_{\mathbf{y}} \mathbf{x}^\top \mathbf{A} \mathbf{y} \right) = \min_{\mathbf{y}} \left(\max_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} \mathbf{y} \right). \tag{8}$$

We next sketch an argument for deriving Theorem 5.1 directly from the guarantee provided by the multiplicative weights algorithm (Theorem 3.1). Exercise Set #6 asks you to provide the details.

Fix a zero-sum game $\mathbf{A}$ with payoffs in $[-1, 1]$ and a value for a parameter $\epsilon > 0$. Let n denote the number of rows or the number of columns, whichever is larger. Consider the following thought experiment:

- At each time step $t = 1, 2, \ldots, T = \frac{4 \ln n}{\epsilon^2}$:
  - The row and column players each choose a mixed strategy ($\mathbf{p}^t$ and $\mathbf{q}^t$, respectively) using their own copies of the multiplicative weights algorithm (with the action set equal to the rows or columns, as appropriate).
  - The row player feeds the reward vector $\mathbf{r}^t = \mathbf{A}\mathbf{q}^t$ into (its copy of) the multiplicative weights algorithm. (This is just the expected payoff of each row, given that the column player chose the mixed strategy $\mathbf{q}^t$.)
  - Analogously, the column player feeds the reward vector $\mathbf{r}^t = -(\mathbf{p}^t)^T \mathbf{A}$ into the multiplicative weights algorithm.

Let

$$v = \frac{1}{T}\sum_{t=1}^{T}(\mathbf{p}^t)^T\mathbf{A}\mathbf{q}^t$$

denote the time-averaged payoff of the row player. The first claim is that applying Theorem 3.1 (in the form of Corollary 3.2) to the row and column players implies that

$$v \geq \left(\max_{\mathbf{p}} \mathbf{p}^T\mathbf{A}\hat{\mathbf{q}}\right) - \epsilon$$

and

$$v \leq \left(\min_{\mathbf{q}} \hat{\mathbf{p}}^T\mathbf{A}\mathbf{q}\right) + \epsilon,$$

respectively, where $\hat{\mathbf{p}} = \frac{1}{T}\sum_{t=1}^{T}\mathbf{p}^t$ and $\hat{\mathbf{q}} = \frac{1}{T}\sum_{t=1}^{T}\mathbf{q}^t$ denote the time-averaged row and column strategies.

Given this, a short derivation shows that

$$\max_{\mathbf{p}}\left(\min_{\mathbf{q}}\mathbf{p}^T\mathbf{A}\mathbf{q}\right) \geq \min_{\mathbf{q}}\left(\min_{\mathbf{p}}\mathbf{p}^T\mathbf{A}\mathbf{q}\right) - 2\epsilon.$$

Letting $\epsilon \to 0$ and recalling the easy direction of the minimax theorem ($\max_{\mathbf{p}}\min_{\mathbf{q}}\mathbf{p}^\top\mathbf{A}\mathbf{q} \leq \min_{\mathbf{q}}\max_{\mathbf{p}}\mathbf{p}^\top\mathbf{A}\mathbf{q}$) completes the proof.

CS261: A Second Course in Algorithms
Lecture #12: Applications of Multiplicative Weights to Games and Linear Programs*

Tim Roughgarden†

February 11, 2016

1 Extensions of the Multiplicative Weights Guarantee

Last lecture we introduced the multiplicative weights algorithm for online decision-making. You don't need to remember the algorithm details for this lecture, but you should remember that it's a simple and natural algorithm (just one simple update per action per time step). You should also remember its regret guarantee, which we proved last lecture and will use today several times as a black box.[1]

Theorem 1.1 *The expected regret of the multiplicative weights algorithm is always at most $2\sqrt{T \ln n}$, where n is the number of actions and T is the time horizon.*

Recall the definition of regret, where A denotes the action set:

$$\underbrace{\max_{a \in A} \sum_{t=1}^{T} r^t(a)}_{\text{best fixed action}} - \underbrace{\sum_{t=1}^{T} r^t(a^t)}_{\text{our algorithm}}.$$

The expectation in Theorem 1.1 is over the random choice of action in each time step; the reward vectors $\mathbf{r}^1, \ldots, \mathbf{r}^T$ are arbitrary.

The regret guarantee in Theorem 1.1 applies not only with with respect to the best fixed action in hindsight, but more generally to the best fixed probability distribution in

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

[1]This lecture is a detour from our current study of online algorithms. While the multiplicative weights algorithm works online, the applications we discuss today are not online problems.

hindsight. The reason is that, in hindsight, the best fixed action is as a good as the best fixed distribution over actions. Formally, for every distribution $\mathbf{p}$ over A,

$$\sum_{t=1}^{T}\sum_{a\in A} p_a \cdot r^t(a) = \sum_{a\in A} \underbrace{p_a}_{\text{sum to 1}} \underbrace{\left(\sum_{t=1}^{T} r^t(a)\right)}_{\leq \max_b \sum_t r^t(b)} \leq \max_{b\in A} \sum_{t=1}^{T} r^t(b).$$

We'll apply Theorem 1.1 in the following form (where time-averaged just means divided by T).

Corollary 1.2 *The expected time-averaged regret of the multiplicative weights algorithm is at most ϵ after at most $(4\ln n)/\epsilon^2$ time steps.*

As noted above, the guarantee of Corollary 1.2 applies with respect to any fixed distribution over the actions.

Another useful extension is to rewards that lie in $[-M, M]$, rather than in $[-1, 1]$. This scenario reduces to the previous one by scaling. To obtain time-averaged regret at most ϵ:

1. scale all rewards down by M;
2. run the multiplicative weights algorithm until the time-averaged expected regret is at most $\frac{\epsilon}{M}$;
3. scale everything back up.

Equivalently, rather than explicitly scaling the reward vectors, one can change the weight update rule from $w^{t+1}(a) = w^t(a)(1+\eta r^t(a))$ to $w^{t+1}(a) = w^t(a)(1+\frac{\eta}{M} r^t(a))$. In any case, Corollary 1.2 implies that after $T = \frac{4M^2 \ln n}{\epsilon^2}$ iterations, the time-averaged expected regret is at most ϵ.

2 Minimax Revisited (Again)

Last lecture we sketched how to use the multiplicative weights algorithm to prove the minimax theorem (details on Exercise Set #6). The idea was to have both the row and the column player play a zero-sum game repeatedly, using their own copies of the multiplicative weights algorithm to choose strategies simultaneously at each time step. We next discuss an alternative thought experiment, where the players move sequentially at each time step with only the row player using multiplicative weights (the column player just best responds). This alternative has similar consequences and translates more directly into interesting algorithmic applications.

Fix a zero-sum game $\mathbf{A}$ with payoffs in $[-M, M]$ and a value for a parameter $\epsilon > 0$. Let m denote the number of rows of $\mathbf{A}$. Consider the following thought experiment, in which the row player has to move first and the column player gets to move second:

Thought Experiment

- At each time step $t = 1, 2, \ldots, T = \frac{4M^2 \ln m}{\epsilon^2}$:
  - The row player chooses a mixed strategy $\mathbf{p}^t$ using the multiplicative weights algorithm (with the action set equal to the rows).
  - The column player responds optimally with the deterministic strategy $\mathbf{q}^t$.[2]
  - If the column player chooses column j, then set $r^t(i) = a_{ij}$ for every row i, and feed the reward vector $\mathbf{r}^t$ into the multiplicative weights algorithm. (This is just the payoff of each row in hindsight, given the column player's strategy at time t.)

We claim that the column player get at least its minimax payoff, and the row player gets at least its minimax payoff minus ϵ.

Claim 1: In the thought experiment above, the negative time-averaged expected payoff of the column player is at most

$$\max_{\mathbf{p}} \left(\min_{\mathbf{q}} \mathbf{p}^T \mathbf{A} \mathbf{q} \right).$$

Note that the benchmark used in this claim is the more advantageous one for the column player, where it gets to move second.[3]

Proof: The column player only does better than its minimax value because, not only does the player get to go second, but the player can tailor its best responses on each day to the row player's mixed strategy on that day. Formally, we let $\hat{\mathbf{p}} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{p}^t$ denote the time-averaged row strategy and $\mathbf{q}^*$ an optimal response to $\hat{\mathbf{p}}$ and derive

$$\begin{aligned} \max_{\mathbf{p}} \left(\min_{\mathbf{q}} \mathbf{p}^T \mathbf{A} \mathbf{q} \right) &\geq \min_{\mathbf{q}} \hat{\mathbf{p}}^T \mathbf{A} \mathbf{q} \\ &= \hat{\mathbf{p}}^T \mathbf{A} \mathbf{q}^* \\ &= \frac{1}{T} \sum_{t=1}^{T} (\mathbf{p}^t)^T \mathbf{A} \mathbf{q}^* \\ &\geq \frac{1}{T} \sum_{t=1}^{T} (\mathbf{p}^t)^T \mathbf{A} \mathbf{q}^t, \end{aligned}$$

with the the last inequality following because $\mathbf{q}^t$ is an optimal response to $\mathbf{p}^t$ for each t. (Recall the column player wants to minimize $\mathbf{p}^T \mathbf{A} \mathbf{q}$.) Since the last term is the negative

[2] Recall from last lecture that the player who goes second has no need to randomize: choosing a column with the best expected payoff (given the row player's strategy $\mathbf{p}^t$) is the best thing to do.

[3] Of course, we've already proved the minimax theorem, which states that it doesn't matter who goes first. But here we want to reprove the minimax theorem, and hence don't want to assume it.

time-averaged payoff of the column player in the thought experiment, the proof is complete. ■

Claim 2: In the thought experiment above, the time-averaged expected payoff of the row player is at least

$$\min_{\mathbf{q}} \left(\max_{\mathbf{p}} \mathbf{p}^T \mathbf{A} \mathbf{q} \right) - \epsilon.$$

We are again using the stronger benchmark from the player's perspective, here with the row player going second.

Proof: Let $\hat{\mathbf{q}} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{q}^t$ denote the time-averaged column strategy. The multiplicative weights guarantee, after being extended as in Section 1, states that the time-averaged expected payoff of the row player is within ϵ of what it could have attained using any fixed mixed strategy $\mathbf{p}$. That is,

$$\begin{aligned} \frac{1}{T} \sum_{t=1}^{T} (\mathbf{p}^t)^T \mathbf{A} \mathbf{q}^t &\geq \max_{\mathbf{p}} \left(\frac{1}{T} \sum_{t=1}^{T} \mathbf{p}^T \mathbf{A} \mathbf{q}^t \right) - \epsilon \\ &= \max_{\mathbf{p}} \mathbf{p}^T \mathbf{A} \hat{\mathbf{q}} - \epsilon \\ &\geq \min_{\mathbf{q}} \left(\max_{\mathbf{p}} \mathbf{p}^T \mathbf{A} \mathbf{q} \right) - \epsilon. \end{aligned}$$

■

Letting $\epsilon \to 0$, Claims 1 and 2 provide yet another proof of the minimax theorem. (Recalling the "easy direction" that $\max_{\mathbf{p}} \min_{\mathbf{q}} \mathbf{p}^T \mathbf{A} \mathbf{q} \leq \min_{\mathbf{q}} \max_{\mathbf{q}} \mathbf{p}^T \mathbf{A} \mathbf{q}$.) The next order of business is to translate this thought experiment into fast algorithms for approximately solving linear programs.

3 Linear Classifiers Revisited

3.1 Recap

Recall from Lecture #7 the problem of computing a linear classifier — geometrically, of separating a bunch of "+"s and "-"s with a hyperplane (Figure 1).

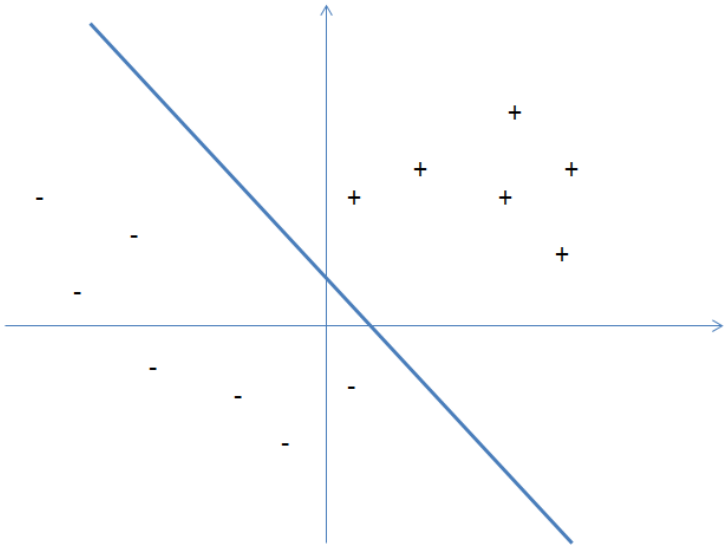

Figure 1: We want to find a linear function that separates the positive points (plus signs) from the negative points (minus signs)

Formally, the input consists of m "positive" data points $\mathbf{p}^1, \ldots, \mathbf{p}^m \in \mathbb{R}^d$ and m' "negative" data points $\mathbf{q}^1, \ldots, \mathbf{q}^{m'} \in \mathbb{R}^d$. This corresponds to labeled data, with the positive and negative points having labels +1 and -1, respectively.

The goal is to compute a linear function $h(\mathbf{z}) = \sum_{j=1}^{d} a_j z_j + b$ (from $\mathbb{R}^d$ to $\mathbb{R}$) such that

$$h(\mathbf{p}^i) > 0$$

for all positive points and

$$h(\mathbf{q}^i) < 0$$

for all negative points. In Lecture #7 we saw how to compute a linear classifier (if one exists) via linear programming. (It was almost immediate; the only trick was to introduce an additional variable to turn the strict inequality constraints into the usual weak inequality constraints.)

We've said in the past that linear programs with 100,000s of variables and constraints are usually no problem to solve, and sometimes millions of variables and constraints are also doable. But as you probably know from your other computer science courses, in many cases we're interested in considerably larger data sets. Can we compute a linear classifier faster, perhaps under some assumptions and/or allowing for some approximation? The multiplicative weights algorithm provides an affirmative answer.

3.2 Preprocessing

We first execute a few preprocessing steps to transform the problem into a more convenient form.

First, we can force the intercept b to be 0. The trick is to add an additional $(d+1)$th variable, with the new coefficient a_{d+1} corresponding to the old intercept b. Each positive and negative data point gets a new $(d+1)$th coordinate, equal to 1. Geometrically, we're now looking for a hyperplane separating the positive and negative points that passes through the origin.

Second, if we multiply all the coordinates of each negative point $\mathbf{y}^i \in \mathbb{R}^{d+1}$ by -1, then we can write the constraints as

$$h(\mathbf{x}^i), h(\mathbf{y}^i) > 0$$

for all positive and negative data points. (For this reason, we will no longer distinguish positive and negative points.) Geometrically, we're now looking for a hyperplane (through the origin) such that all of the data points are on the same side of the hyperplane.

Third, we can insist that every coefficient a_j is nonnegative. (Don't forget that the coordinates of the $\mathbf{x}^i$'s can be both positive and negative.) The trick here is to make two copies of every coordinate (blowing up the dimension from $d+1$ to $2d+2$), and interpreting the two coefficients a'_j, a''_j corresponding to the jth coordinate as indicating the coefficient $a_j = a'_j - a''_j$ in the original space. For this to work, each entry $\mathbf{x}^i_j$ of a data point is replaced by two entries, $\mathbf{x}^i_j$ and $-\mathbf{x}^i_j$. Geometrically, we're now looking for a hyperplane, through the origin and with a normal vector in the nonnegative orthant, with all the data points on the same side (and the same side as the normal vector).

For the rest of this section, we use d to denote the number of dimensions after all of this preprocessing (i.e., we redefine d to be what was previously $2d+2$).

3.3 Assumption

We assume that the problem is feasible — that there is a linear function of the desired type. Actually, we assume a bit more, that there is a solution with some "wiggle room."

Assumption: There is a coefficient vector $\mathbf{a}^* \in \mathbb{R}^d_+$ such that:

1. $\sum_{j=1}^d a^*_i = 1$; and
2. $\sum_{j=1}^d a^*_j x^i_j > \underbrace{\epsilon}_{\text{"margin"}}$ for all data points $\mathbf{x}^i$.

Note that if there is any solution to the problem, then there is a solution satisfying the first condition (just by scaling the coefficients). The second condition insists on wiggle room after normalizing the coefficients to sum to 1.

Let M be such that $|x^i_j| \leq M$ for every i and j. The running time of our algorithm will depend on both ϵ and M.

3.4 Algorithm

Here is the algorithm.

1. Define an action set $A = \{1, 2, \ldots, d\}$, with actions corresponding to coordinates.
2. For $t = 1, 2, \ldots, T = \frac{4M^2 \ln d}{\epsilon^2}$:
   (a) Use the multiplicative weights algorithm to generate a probability distribution $\mathbf{a}^t \in \mathbb{R}^d$ over the actions/coordinates.
   (b) If $\sum_{j=1}^d a_j^t x_j^i > 0$ for every data point $\mathbf{x}^i$, then halt and return $\mathbf{a}^t$ (which is a feasible solution).
   (c) Otherwise, choose some data point $\mathbf{x}^i$ with $\sum_{j=1}^d a_j^t x_j^i \leq 0$, and define a reward vector $\mathbf{r}^t$ with $r^t(j) = x_j^i$ for each coordinate j.
   (d) Feed the reward vector $\mathbf{r}^t$ into the multiplicative weights algorithm.

To motivate the choice of reward vector, suppose the coefficient vector $\mathbf{a}^t$ fails to have a positive inner product $\sum_{j=1}^d a_j^t x_j^i$ with the data point $\mathbf{x}^i$. We want to nudge the coefficients so that this inner product will go up in the next iteration. (Of course we might screw up some other inner products, but we're hoping it'll work out OK in the end.) For coordinates j with $x_j^i > 0$ we want to increase a_j; for coordinates with $x_j^i < 0$ we want to do the opposite. Recalling the multiplicative weight update rule ($w^{t+1}(a) = w^t(a)(1 + \eta r^t(a))$), we see that the reward vector $\mathbf{r}^t = \mathbf{x}^i$ will have the intended effect.

3.5 Analysis

We claim that the algorithm above halts (necessarily with a feasible solution) by the time it gets to the final iteration T.

In the algorithm, the reward vectors are nefariously defined so that, at every time step t, the inner product of $\mathbf{a}^t$ and $\mathbf{r}^t$ is non-positive. Viewing $\mathbf{a}^t$ as a probability distribution over the actions $\{1, 2, \ldots, d\}$, the means that the expected reward of the multiplicative weights algorithm is non-positive at every time step, and hence its time-averaged expected reward is at most 0.

On the other hand, by assumption (Section 3.3), there exists a coefficient vector (equivalently, distribution over $\{1, 2, \ldots, d\}$) $\mathbf{a}^*$ such that, at every time step t, the expected payoff of playing $\mathbf{a}^*$ would have been $\sum_{j=1}^d a_j^* r^t(j) \geq \min_{i=1}^m \sum_{j=1}^d a_j^* x_j^i > \epsilon$.

Combining these two observations, we see that as long as the algorithm has not yet found a feasible solution, the time-averaged regret of the multiplicative weights subroutine is strictly more than ϵ. The multiplicative weights guarantee says that after $T = \frac{4M^2 \ln d}{\epsilon^2}$, the time-averaged regret is at most ϵ.[4] We conclude that our algorithm halts, with a feasible linear classifier, within T iterations.

[4] We're using the extended version of the guarantee (Section 1), which holds against every fixed distribution (like $\mathbf{a}^*$) and not just every fixed action.

3.6 Interpretation as a Zero-Sum Game

Our last two topics were a thought experiment leading to minimax payoffs in zero-sum games and an algorithm for computing a linear classifier. *The latter is just a special case of the former.*

To translate the linear classifier problem to a zero-sum game, introduce one row for each of the d coordinates and one column for each of the data points $\mathbf{x}^i$. Define the payoff matrix $\mathbf{A}$ by

$$\mathbf{A} = \left[a_{ji} = x_j^i\right]$$

Recall that in our thought experiment (Section 2), the row player generates a strategy at each time step using the multiplicative weights algorithm. This is exactly how we generate the coefficient vectors $\mathbf{a}^1, \ldots, \mathbf{a}^T$ in the algorithm in Section 3.4. In the thought experiment, the column player, knowing the row player's distribution, chooses the column that minimizes the expected payoff of the row player. In the linear classifier context, given $\mathbf{a}^t$, this corresponds to picking a data point $\mathbf{x}^i$ that minimizes $\sum_{j=1}^d a_j^t x_j^i$. This ensures that a violated data point (with nonpositive dot product) is chosen, provided one exists. In the thought experiment, the reward vector $\mathbf{r}^t$ fed into the multiplicative weights algorithm is the payoff of each row in hindsight, given the column player's strategy at time t. With the payoff matrix $\mathbf{A}$ above, this vector corresponds to the data point $\mathbf{x}^i$ chosen by the column player at time t. These are exactly the reward vectors used in our algorithm for computing a linear classifier.

Finally, the assumption (Section 3.3) implies that the value of the constructed zero-sum game is bigger than ϵ (since the row player could always choose $\mathbf{a}^*$). The regret guarantee in Section 2 translates to the row player having time-averaged expected payoff bigger than 0 once T exceeds $\frac{4M^2 \ln m}{\epsilon^2}$. The algorithm has no choice but to halt (with a feasible solution) before this time.

4 Maximum Flow Revisited

4.1 Multiplicative Weights and Linear Programs

We've now seen a concrete example of how to approximately solve a linear program using the multiplicative weights algorithm, by modeling the linear program as a zero-sum game and then applying the thought experiment from Section 2. The resulting algorithm is extremely fast (faster than solving the linear program exactly) provided the margin ϵ is not overly small and the radius M of the ℓ_∞ ball enclosing all of the data points $\mathbf{x}_j^i$ is not overly big.

This same idea — associating one player with the decision variables and a second player with the constraints — can be used to quickly approximate many other linear programs. We'll prove this point by considering one more example, our old friend the maximum flow problem. Of course, we already know some pretty good algorithms (faster than linear programs) for maximum flow problems, but the ideas we'll discuss extend also to multicommodity flow problems (see Exercise Set #6 and Problem Set #3), where we don't know any exact algorithms that are significantly faster than linear programming.

4.2 A Zero-Sum Game for the Maximum Flow Problem

Recall the primal-dual pair of linear programs corresponding to the maximum flow and minimum cut problems (Lecture #8):

$$\max \sum_{P \in \mathcal{P}} f_P$$

subject to

$$\underbrace{\sum_{P \in \mathcal{P} : e \in P} f_P}_{\text{total flow on } e} \leq 1 \qquad \text{for all } e \in E$$

$$f_P \geq 0 \qquad \text{for all } P \in \mathcal{P}$$

and

$$\min \sum_{e \in E} \ell_e$$

subject to

$$\sum_{e \in P} \ell_e \geq 1 \qquad \text{for all } P \in \mathcal{P}$$

$$\ell_e \geq 0 \qquad \text{for all } e \in E,$$

where $\mathcal{P}$ denotes the set of s-t paths. To reduce notation, here we'll only consider the case where all edges have unit capacity ($u_e = 1$). The general case, with u_e's on the right-hand side of the primal and in the objective function of the dual, can be solved using the same ideas (Exercise Set #6).[5]

We begin by defining a zero-sum game. The row player will be associated with edges (i.e., dual variables) and the column player with paths (i.e., primal variables). The payoff matrix is

$$\mathbf{A} = \left[a_{eP} = \begin{cases} 1 \text{ if } e \in P \\ 0 \text{ otherwise} \end{cases} \right]$$

Note that all payoffs are 0 or 1. (Yes, this a huge matrix, but we'll never have to write it down explicitly; see the algorithm below.)

Let OPT denote the optimal objective function value of the linear programs. (The same for each, by strong duality.) Recall that the value of a zero-sum game is defined as the expected payoff of the row player under optimal play by both players ($\max_{\mathbf{x}} \min_{\mathbf{y}} \mathbf{x}^T \mathbf{A} \mathbf{y}$ or, equivalently by the minimax theorem, $\min_{\mathbf{y}} \max_{\mathbf{x}} \mathbf{x}^T \mathbf{A} \mathbf{y}$).

Claim: The value of this zero-sum game is $\frac{1}{OPT}$.

[5] Although the running time scales quadratically with ratio of the maximum and minimum edge capacities, which is not ideal. One additional idea ("width reduction"), not covered here, recovers a polynomial-time algorithm for general edge capacities.

Proof: Let $\{\ell_e^*\}_{e \in E}$ be an optimal solution to the dual, with $\sum_{e \in E} \ell_e^* = OPT$. Obtain x_e's from the ℓ_e^*'s by scaling down by OPT — then the x_e's form a probability distribution. If the row player uses this mixed strategy $\mathbf{x}$, then each column $P \in \mathcal{P}$ results in expected payoff

$$\sum_{e \in P} x_e = \frac{1}{OPT} \sum_{e \in P} \ell_e^* \geq \frac{1}{OPT},$$

where the inequality follows the dual feasibility of $\{\ell_e^*\}_{e \in E}$. This shows that the value of the game is at least $\frac{1}{OPT}$.

Conversely, let $\mathbf{x}$ be an optimal strategy for the row player, with $\min_{\mathbf{y}} \mathbf{x}^T \mathbf{A} \mathbf{y}$ equal to the game's value v. This means that, no matter what strategy the column player chooses, the row player's expected payoff is at least v. This translates to

$$\sum_{e \in P} x_e \geq v$$

for every $P \in \mathcal{P}$. Thus $\{x_e/v\}_{e \in E}$ is a dual feasible solution, with objective function value $(\sum_{e \in E} x_e)/v = 1/v$. Since this can only be larger than OPT, $v \leq \frac{1}{OPT}$. ■

4.3 Algorithm

For simplicity, assume that OPT is known.[6] Translating the thought experiment from Section 2 to this zero-sum game, we get the following algorithm:

1. Associate an action with each edge $e \in E$.
2. For $t = 1, 2, \ldots, T = \frac{4OPT^2 \ln |E|}{\epsilon^2}$:
   (a) Use the multiplicative weights algorithm to generate a probability distribution $\mathbf{x}^t \in \mathbb{R}^E$ over the actions/edges.
   (b) Let P^t be a column that minimizes the row player's expected payoff (with the expectation with respect to $\mathbf{x}^t$). That is,

$$P^t \in \operatorname*{argmin}_{P \in \mathcal{P}} \sum_{e \in P} x_e^t. \tag{1}$$

   (c) Define a reward vector $\mathbf{r}^t$ with $r^t(e) = 1$ for $e \in P^t$ and $r^t(e) = 0$ for $e \notin P^t$ (i.e., the P^tth column of $\mathbf{A}$). Feed the reward vector $\mathbf{r}^t$ into the multiplicative weights algorithm.

[6] For example, embed the algorithm into an outer loop that uses successive doubling to "guess" the value of OPT (i.e., take $OPT = 1, 2, 4, 8, \ldots$ until the algorithm succeeds).

4.4 Running Time

An important observation is that this algorithm never explicitly writes down the payoff matrix $\mathbf{A}$. It maintains one weight per edge, which is a reasonable amount of state. To compute P^t and the induced reward vector $\mathbf{r}^t$, all that is needed is a subroutine that solves (1) — that, given the x_e^t's, returns a shortest s-t path (viewing the x_e^t's as edge lengths). Dijkstra's algorithm, for example, works just fine.[7] Assuming Dijkstra's algorithm is implemented in $O(m \log n)$ time, where m and n denote the number of edges and vertices, respectively, the total running time of the algorithm is $O(\frac{OPT^2}{\epsilon^2} m \log m \log n)$. (Note that with unit capacities, $OPT \leq m$. If there are no parallel edges, then $OPT \leq n-1$.) This is comparable to some of the running times we saw for (exact) maximum flow algorithms, but more importantly these ideas extend to more general problems, including multicommodity flow.

4.5 Approximate Correctness

So how do we extract an approximately optimal flow from this algorithm? After running the algorithm above, let $P^1, \ldots, P^T$ denote the sequence of paths chosen by the column player (the same path can be chosen multiple times). Let f^t denote the flow that routes OPT units of flow on the path P^t. (Of course, this probably violates the edge capacity constraints.) Finally, define $f^* = \frac{1}{T}\sum_{t=1}^{T} f^t$ as the "time-average" of these path flows. Note that since each f^t routes OPT units of flow from the source to the sink, so does f^*. But is f^* feasible?

Claim: f^* routes at most $1+\epsilon$ units of flow on every edge.

Proof: We proceed by contradiction. If f^* routes more than $1+\epsilon$ units of flow on the edge e, then more than $(1+\epsilon)T/OPT$ of the paths in $P^1, \ldots, P^T$ include the edge e. Returning to our zero-sum game $\mathbf{A}$, consider the row player strategy $\mathbf{z}$ that deterministically plays the edge e. The time-averaged payoff to the row player, in hindsight given the paths chosen by the column player, would have been

$$\frac{1}{T}\sum_{t=1}^{T} \mathbf{z}^T \mathbf{A} \mathbf{y}^t = \frac{1}{T} \sum_{t\,:\,e \in P^t} 1 > \frac{1+\epsilon}{OPT}.$$

The row player's guarantee (Claim 1 in Section 2) then implies that

$$\frac{1}{T}\sum_{t=1}^{T} (\mathbf{x}^t)^T \mathbf{A} \mathbf{y}^t \geq \frac{1}{T}\sum_{t=1}^{T} \mathbf{z}^T \mathbf{A} \mathbf{y}^t - \frac{\epsilon}{OPT} > \frac{1+\epsilon}{OPT} - \frac{\epsilon}{OPT} = \frac{1}{OPT}.$$

But this contradicts the guarantee that the column player does at least as well as the minimax value of the game (Claim 2 in Section 2), which is $\frac{1}{OPT}$ by the Claim in Section 4.2. ■

Scaling down f^* by a factor of $1+\epsilon$ yields a feasible flow with value at least $OPT/(1+\epsilon)$.

[7] This subroutine is precisely the "separation oracle" for the dual linear program, as discussed in Lecture #10 in the context of the ellipsoid method.

CS261: A Second Course in Algorithms
Lecture #13: Online Scheduling and Online Steiner Tree*

Tim Roughgarden†

February 16, 2016

1 Preamble

Last week we began our study of online algorithms with the multiplicative weights algorithm for online decision-making. We also covered (non-online) applications of this algorithm to zero-sum games and the fast approximation of certain linear programs. This week covers more "traditional" results in online algorithms, with applications in scheduling, matching, and more.

Recall from Lecture #11 what we mean by an online problem.

An Online Problem

1. The input arrives "one piece at a time."
2. An algorithm makes an irrevocable decision each time it receives a new piece of the input.

2 Online Scheduling

A canonical application domain for online algorithms is scheduling, with jobs arriving online (i.e., one-by-one). There are many algorithms and results for online scheduling problems; we'll cover only what is arguably the most classic result.

2.1 The Problem

To specify an online problem, we need to define how the input arrives at what action must be taken at each step. There are m identical machines on which jobs can be scheduled;

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

these are known up front. Jobs then arrive online, one at a time, with job j having a known processing time p_j. A job must be assigned to a machine immediately upon its arrival.

A *schedule* is an assignment of each job to one machine. The *load* of a machine in a schedule is the sum of the processing times of the jobs assigned to it. The *makespan* of a schedule is the maximum load of any machine. For example, see Figure 1.

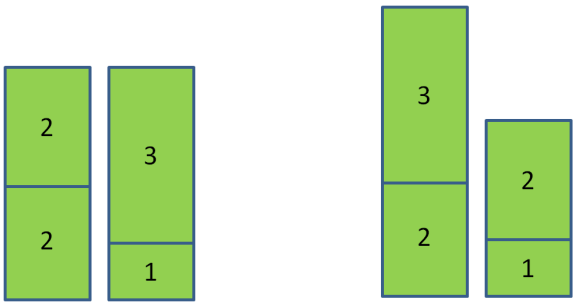

Figure 1: Example of makespan assignments. (a) has makespan 4 and (b) has makespan 5.

We consider the objective function of minimizing the makespan. This is arguably the most practically relevant scheduling objective. For example, if jobs represent pieces of a task to be processed in parallel (e.g., MapReduce/Hadoop jobs), then for many tasks the most important statistic is the time at which the last job completes. Minimizing this last completion time is equivalent to minimizing the makespan.

2.2 Graham's Algorithm

We analyze what is perhaps the most natural approach to the problem, proposed and analyzed by Ron Graham 50 years ago.

Graham's Scheduling Algorithm

when a new job arrives, assign it to the machine that currently has the smallest load (breaking ties arbitrarily)

We measure the performance of this algorithm against the strongest-possible benchmark, the minimum makespan in hindsight (or equivalently, the optimal clairvoyant solution).[1] Since the minimum makespan problem is NP-hard, this benchmark is both omniscient about the future and also has unbounded computational power. So any algorithm that does almost as well is a pretty good algorithm!

[1]Note that the "best fixed action" idea from online decision-making doesn't really make sense here.

2.3 Analysis

In the first half of CS261, we were always asking "how do we know when we're done (i.e., optimal)?" This was the appropriate question when the goal was to design an algorithm that always computes an optimal solution. In an online problem, we don't expect any online algorithm to always compute the optimal-in-hindsight solution. We expect to compromise on the guarantees provided by online algorithms with respect to this benchmark.

In the first half of CS261, we were obsessed with "optimality conditions" — necessary and sufficient conditions on a feasible solution for it to be an optimal solution. In the second half of CS261, we'll be obsessed with *bounds* on the optimal solution — quantities that are "only better than optimal." Then, if our algorithm's performance is not too far from our bound, then it is also not too far from the optimal solution.

Where do such bounds come from? For the two case studies today, simple bounds suffice for our purposes. Next lecture we'll use LP duality to obtain such bounds — this will demonstrate that the same tools that we developed to prove the optimality of an algorithm can also be useful in proving approximate optimality.

The next two lemmas give two different simple lower bounds on the minimum-possible makespan (call it OPT), given m machines and jobs with processing times $p_1, \ldots, p_n$.

Lemma 2.1 (Lower Bound #1)

$$OPT \geq \max_{j=1}^{n} p_j.$$

Lemma 2.1 should be clear enough — the biggest job has to go somewhere, and wherever it is assigned, that machine's load (and hence the makespan) will be at least as big as the size of this job.

The second lower bound is almost as simple.

Lemma 2.2 (Lower Bound #2)

$$OPT \geq \frac{1}{m} \sum_{j=1}^{n} p_j.$$

Proof: In every schedule, we have

$$\begin{aligned} \text{maximum load of a machine} &\geq \text{average load of a machine} \\ &= \frac{1}{m} \sum_{j=1}^{n} p_j. \end{aligned}$$

■

These two lemmas imply the following guarantee for Graham's algorithm.

Theorem 2.3 *The makespan of the schedule output by Graham's algorithm is always at most twice the minimum-possible makespan (in hindsight).*

In online algorithms jargon, Theorem 2.3 asserts that Graham's algorithm is *2-competitive*, or equivalently has a *competitive ratio* of at most 2.

Theorem 2.3 is tight in the worst case (as $m \to \infty$), though better bounds are possible in the (often realistic) special case where all jobs are relatively small (see Exercise Set #7).

Proof of Theorem 2.3: Consider the final schedule produced by Graham's algorithm, and suppose machine i determines the makespan (i.e., has the largest load). Let j denote the last job assigned to i. Why was j assigned to i at that point? It must have been that, at that time, machine i had the smallest load (by the definition of the algorithm). Thus prior to j's assignment, we had

$$\begin{aligned} \text{load of } i &= \text{minimum load of a machine (at that time)} \\ &\leq \text{average load of a machine (at that time)} \\ &= \frac{1}{m}\sum_{k=1}^{j-1} p_k. \end{aligned}$$

Thus,

$$\begin{aligned} \text{final load of machine } i &\leq \underbrace{\frac{1}{m}\sum_{k=1}^{j-1} p_k}_{\leq OPT} + \underbrace{p_j}_{\leq OPT} \\ &\leq 2OPT, \end{aligned}$$

with the last inequality following from our two lower bounds on OPT (Lemma 2.1 and 2.2). ■

Theorem 2.3 should be taken as a representative result in a very large literature. Many good guarantees are known for different online scheduling algorithms and different scheduling problems.

3 Online Steiner Tree

We have two more case studies in online algorithms: the online Steiner tree problem (this lecture) and the online bipartite matching problem (next lecture).[2]

3.1 Problem Definition

In the online Steiner tree problem:

[2]Because the benchmark of the best-possible solution in hindsight is so strong, for many important problems, all online algorithm have terrible competitive ratios. In these cases, it is important to change the setup so that theory can still give useful advice about which algorithm to use. See the instructor's CS264 course ("beyond worst-case analysis") for much more on this. In CS261, we'll cherrypick a few problems where there *are* natural online algorithms with good competitive ratios.

- an algorithm is given in advance a connected undirected graph $G = (V, E)$ with a nonnegative cost $c_e \geq 0$ for each edge $e \in E$;
- "terminals" $t_1, \ldots, t_k \in V$ arrive online (i.e., one-by-one).

The requirement for an online algorithm is to maintain at all times a subgraph of G that spans all of the terminals that have arrived thus far. Thus when a new terminal arrives, the algorithm must connect it to the subgraph-so-far. Think, for example, of a cable company as it builds new infrastructure to reach emerging markets. The gold standard is to compute the minimum-cost subgraph that spans all of the terminals (the "Steiner tree").[3] The goal of an online algorithm is to get as close as possible to this gold standard.

3.2 Metric Case vs. General Case

A seemingly special case of the online Steiner tree problem is the *metric* case. Here, we assume that:

1. The graph G is the complete graph.[4]
2. The edges satisfy the *triangle inequality*: for every triple $u, v, w \in V$ of vertices,

$$c_{uw} \leq c_{uv} + c_{vw}.$$

The triangle inequality asserts that the shortest path between any two vertices is the direct edge between the vertices (which exists, since G is complete) — that is, adding intermediate destinations can't help. The condition states that one-hop paths are always at least as good as two-hop paths; by induction, one-hop paths are as good as arbitrary paths between the two endpoints.

For example, distances between points in a normed space (like Euclidean space) satisfy the triangle inequality. Fares for airline tickets are a non-example: often it's possible to get a cheaper price by adding intermediate stops.

It turns out that the metric case of the online Steiner tree problem is no less general than the general case.

Lemma 3.1 *Every α-competitive online algorithm for the metric case of the online Steiner tree problem can be transformed into an α-competitive online algorithm for the general online Steiner tree problem.*

Exercise Set #7 asks you to supply the proof.

[3] Since costs are nonnegative, this is a tree, without loss of generality.

[4] By itself, this is not a substantial assumption — one could always complete an arbitrary graph with super-high-cost edges.

3.3 The Greedy Algorithm

We'll study arguably the most natural online Steiner tree algorithm, which greedily connects a new vertex to the subgraph-so-far in the cheapest-possible way.[5]

Greedy Online Steiner Tree

initialize $T \subseteq E$ to the empty set
for each terminal arrival t_i, $i = 2, \ldots, k$ **do**
    add to T the cheapest edge of the form (t_i, t_j) with $j < i$

For example, in the 11th iteration of the algorithm, the algorithm looks at the 10 edges between the new terminal and the terminals that have already arrived, and connects the new terminal via the cheapest of these edges.[6]

3.4 Two Examples

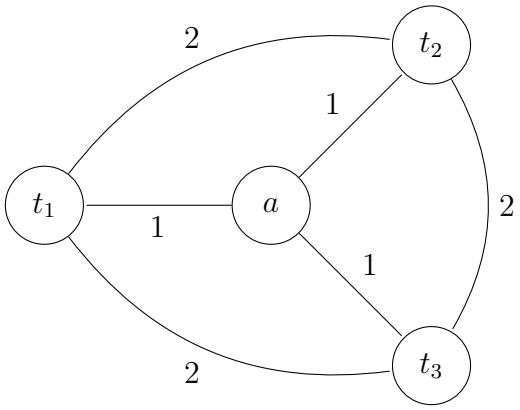

Figure 2: First example.

For example, consider the graph in Figure 2, with edge costs as shown. (Note that the triangle inequality holds.) When the first terminal t_1 arrives, the online algorithm doesn't have to do anything. When the second terminal t_2 arrives, the algorithm adds the edge (t_1, t_2), which has cost 2. When terminal t_3 arrives, the algorithm is free to connect it to either t_1 or t_2 (both edges have cost 2). In any case, the greedy algorithm constructs a

[5] What else could you do? An alternative would be to build some extra infrastructure, hedging against the possibility of future terminals that would otherwise require redundant infrastructure. This idea actually beats the greedy algorithm in non-worst-case models (see CS264).

[6] This is somewhat reminiscent of Prim's minimum-spanning tree algorithm. The difference is that Prim's algorithm processes the vertices in a greedy order (the next vertex to connect is the closest one), while the greedy algorithm here is online, and has to process the terminals in the order provided.

subgraph with total cost 4. Note that the optimal Steiner tree in hindsight has cost 3 (the spokes).

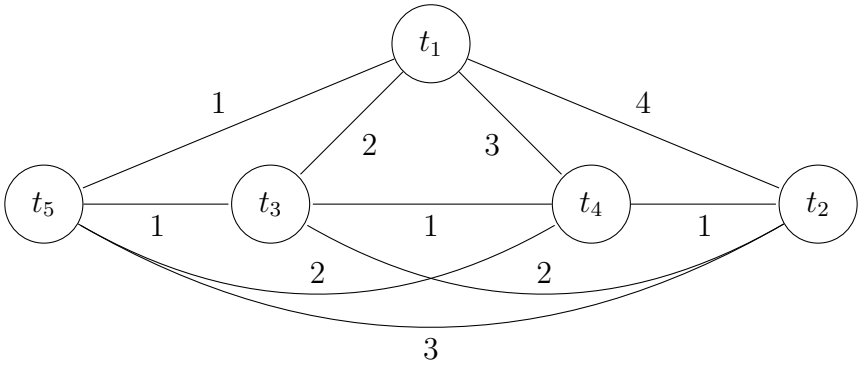

Figure 3: Second example.

For a second example, consider the graph in Figure 3. Again, the edge costs obey the triangle inequality. When t_1 arrives, the algorithm does nothing. When t_2 arrives, the algorithm adds the edge (t_1, t_2), which has cost 4. When t_3 arrives, there is a tie between the edges (t_1, t_3) and (t_2, t_3), which both have cost 2. Let's say that the algorithm picks the latter. When terminals t_4 and t_5 arrive, in each case there are two unit-cost options, and it doesn't matter which one the algorithm picks. At the end of the day, the total cost of the greedy solution is $4 + 2 + 1 + 1 = 8$. The optimal solution in hindsight is the path graph t_1-t_5-t_3-t_4-t_2, which has cost 4.

3.5 Lower Bounds

The second example above shows that the greedy algorithm cannot be better than 2-competitive. In fact, it is not constant-competitive for any constant.

Proposition 3.2 *The (worst-case) competitive ratio of the greedy online Steiner tree algorithm is $\Omega(\log k)$, where k is the number of terminals.*

Exercise Set #7 asks you to supply the proof, by extending the second example above.

The following result is harder to prove, but true.

Proposition 3.3 *The (worst-case) competitive ratio of every online Steiner tree algorithm, deterministic or randomized, is $\Omega(\log k)$.*

3.6 Analysis of the Greedy Algorithm

We conclude the lecture with the following result.

Theorem 3.4 *The greedy online Steiner tree algorithm is* $2 \ln k$*-competitive, where* k *is the number of terminals.*

In light of Proposition 3.3, we conclude that the greedy algorithm is an optimal online algorithm (in the worst case, up to a small constant factor).

The theorem follows easily from the following key lemma, which relates the costs incurred by the greedy algorithm to that of the optimal solution in hindsight.

Lemma 3.5 *For every* $i = 1, 2, \ldots, k-1$*, the* i*th most expensive edge in the greedy solution* T *has cost at most* $2OPT/i$*, where* OPT *is the cost of the optimal Steiner tree in hindsight.*

Thus, the most expensive edge in the greedy solution has cost at most $2OPT$, the second-most expensive edge costs at most OPT, the third-most at most $2OPT/3$, and so on. Recall that the greedy algorithm adds exactly one edge in each of the $k-1$ iterations after the first, so Lemma 3.5 applies (with a suitable choice of i) to each edge in the greedy solution.

To apply the key lemma, imagine sorting the edges in the final greedy solution from most to least expensive, and then applying Lemma 3.5 to each (for successive values of $i = 1, 2, \ldots, k-1$). This gives

$$\text{greedy cost} \leq \sum_{i=1}^{k-1} \frac{2OPT}{i} = 2OPT \sum_{i=1}^{k-1} \frac{1}{i} \leq (2 \ln k) \cdot OPT,$$

where the last inequality follows by estimating the sum by an integral.

It remains to prove the key lemma.

Proof of Lemma 3.5: The proof uses two nice tricks, "tree-doubling" and "shortcutting," both of which we'll reuse later when we discuss the Traveling Salesman Problem.

We first recall an easy fact from graph theory. Suppose H is a connected multi-graph (i.e., parallel copies of an edge are OK) in which every vertex has even degree (a.k.a. an "Eulerian graph"). Then H has an *Euler tour*, meaning a closed walk (i.e., a not-necessarily-simple cycle) that uses every edge exactly once. See Figure 4. The all-even-degrees condition is clearly necessary, since if the tour visits a vertex k times then it must have degree $2k$. You've probably seen the proof of sufficiency in a discrete math course; we leave it to Exercise Set #7.[7]

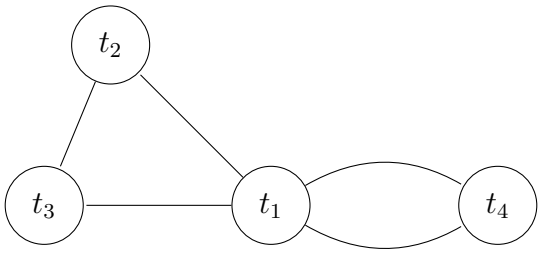

Figure 4: Example graph with Euler tour t_1-t_2-t_3-t_1-t_4-t_1.

[7]Basically, you just peel off cycles one-by-one until you reach the empty graph.

Next, let T^* be the optimal Steiner tree (in hindsight) spanning all of the terminals $t_1, \ldots, t_k$. Let $OPT = \sum_{e \in T^*} c_e$ denote its cost. Obtain H from T^* by adding a second copy of every edge (Figure 5). Obviously, H is Eulerian (every vertex degree got doubled) and $\sum_{e \in H} c_e = 2OPT$. Let C denote an Euler tour of H. C visits each of the terminals at least one, perhaps multiple times, and perhaps visits some other vertices as well. Since C uses every edge of H once, $\sum_{e \in C} c_e = 2OPT$.

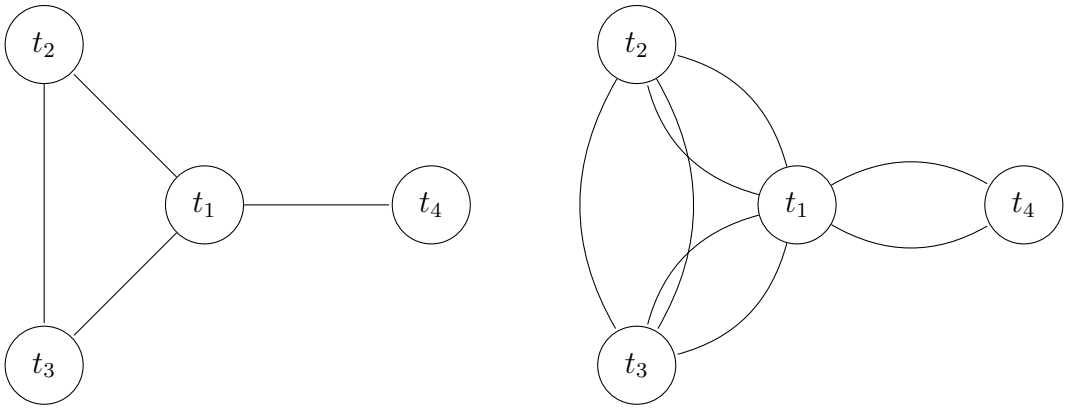

Figure 5: (a) Before doubling edges and (b) after doubling edges.

Now fix a value for the parameter $i \in \{1, 2, \ldots, k-1\}$ in the lemma statement. Define the "connection cost" of a terminal t_j with $j > 1$ as the cost of the edge that was added to the greedy solution when t_j arrived (from t_j to some previous terminal). Sort the terminals in hindsight in nonincreasing order of connection cost, and let $s_1, \ldots, s_i$ be the first (most expensive) i terminals. The lemma asserts that the cheapest of these has connection cost at most $2OPT/i$. (The ith most expensive terminal is the cheapest of the i most expensive terminals.)

The tour C visits each of $s_1, \ldots, s_i$ at least once. "Shortcut" it to obtain a simple cycle C_i on the vertex set $\{s_1, \ldots, s_i\}$ (Figure 6). For example, if the first occurrences of the terminals in C happen to be in the order $s_1, \ldots, s_i$, then C_i is just the edges $(s_1, s_2), (s_2, s_3), \ldots, (s_i, s_1)$. In any case, the order of terminals on C_i is the same as that of their first occurrences in C. Since the edge costs satisfy the triangle inequality, replacing a path by a direct edge between its endpoints can only decrease the cost. Thus $\sum_{e \in C_i} c_e \leq \sum_{e \in C} c_e = 2OPT$. Since C_i only has i edges,

$$\underbrace{\min_{e \in C_i} c_e}_{\text{cheapest edge}} \leq \underbrace{\frac{1}{i} \sum_{e \in C_i} c_e}_{\text{average edge cost}} \leq 2OPT/i.$$

Thus some edge $(s_h, s_j) \in C_i$ has cost at most $2OPT/i$.

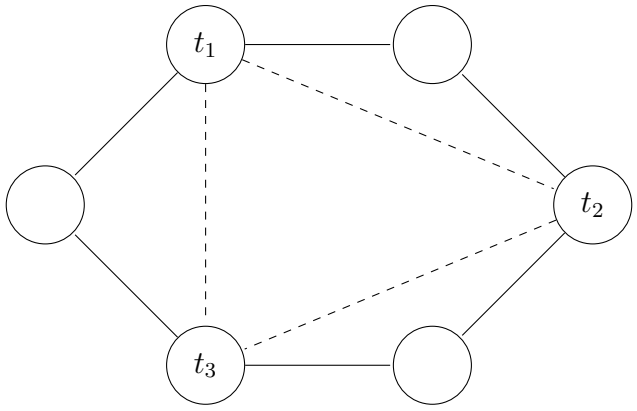

Figure 6: (a) Solid edges represent original edges, and dashed edge represent edges after shortcutting from t_1 to t_2, t_2 to t_3, t_3 to t_1 has been done.

Consider whichever of s_h, s_j arrives later in the online ordering, say s_j. Since s_h arrived earlier, the edge (s_h, s_j) is one option for connecting s_j to a previous terminal; the greedy algorithm either connects s_j via this edge or by one that is even cheaper. Thus at least one vertex of $\{s_1, \ldots, s_i\}$, namely s_j, has connection cost at most $2OPT/i$. Since these are by definition the terminals with the i largest connection costs, the proof is complete. ■

CS261: A Second Course in Algorithms
Lecture #14: Online Bipartite Matching*

Tim Roughgarden†

February 18, 2016

1 Online Bipartite Matching

Our final lecture on online algorithms concerns the *online bipartite matching* problem. As usual, we need to specify how the input arrives, and what decision the algorithm has to make at each time step. The setup is:

- The left-hand side vertices L are known up front.
- The right-hand side vertices R arrive online (i.e., one-by-one). A vertex $w \in R$ arrives together with all of the incident edges (the graph is bipartite, so all of w's neighbors are in L).
- The only time that a new vertex $w \in R$ can be matched is immediately on arrival.

The goal is to construct as large a matching as possible. (There are no edge weights, we're just talking about maximum-cardinality bipartite matching.) We'd love to just wait until all of the vertices of R arrive and then compute an optimal matching at the end (e.g., via a max flow computation). But with the vertices of R arriving online, we can't expect to always do as well as the best matching in hindsight.

This lecture presents the ideas behind optimal (in terms of worst-case competitive ratio) deterministic and randomized online algorithms for online bipartite matching. The randomized algorithm is based on a non-obvious greedy algorithm. While the algorithms do not reference any linear programs, we will nonetheless prove the near-optimality of our algorithms by exhibiting a feasible solution to the dual of the maximum matching problem. This demonstrates that the tools we developed for proving the optimality of algorithms (for max flow, linear programming, etc.) are more generally useful for establishing the approximate optimality of algorithms. We will see many more examples of this in future lectures.

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

Online bipartite matching was first studied in 1990 (when online algorithms were first hot), but a new 21st-century killer application has rekindled interest on the problem over the past 7-8 years. (Indeed, the main proof we present was only discovered in 2013!)

The killer application is Web advertising. The vertices of L, which are known up front, represent advertisers who have purchased a contract for having their ad shown to users that meet specified demographic criteria. For example, an advertiser might pay (in advance) to have their ad shown to women between the ages of 25 and 35 who live within 100 miles of New York City. If an advertiser purchased 5000 views, then there will be 5000 corresponding vertices on the left-hand side. The right-hand side vertices, which arrive online, correspond to "eyeballs." When someone types in a search query or accesses a content page (a new opportunity to show ads), it corresponds to the arrival of a vertex $w \in R$. The edges incident to w correspond to the advertisers for whom w meets their targeting criteria. Adding an edge to the matching then corresponds to showing a given ad to the newly arriving eyeball. Both Google and Microsoft (and probably other companies) employ multiple people whose primary job is adapting and fine-tuning the algorithms discussed in this lecture to generate as much as revenue as possible.

2 Deterministic Algorithms

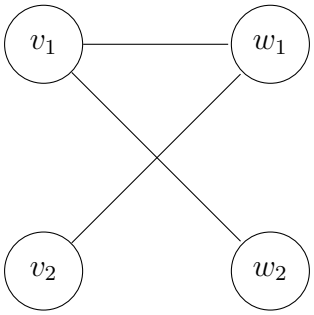

Figure 1: Graph where no deterministic algorithm has competitive ratio better than $\frac{1}{2}$.

We first observe that no deterministic algorithm has a competitive ratio better than $\frac{1}{2}$. Consider the example in Figure 1. The two vertices v_1, v_2 on the left are known up front, and the first vertex w_1 to arrive on the right is connected to both. Every deterministic algorithm picks either the edge (v_1, w_1) or (v_2, w_1).[1] In the former case, suppose the second vertex w_2 to arrive is connected only to v_1, which is already matched. In this case the online algorithm's solution has 1 edge, while the best matching in hindsight has size 2. The other case is symmetric. Thus for every deterministic algorithm, there is an instance where the matching it outputs is at most $\frac{1}{2}$ times the maximum possible in hindsight.

[1]Technically, the algorithm could pick neither, but then its competitive ratio would be 0 (what if no more vertices arrive?).

The obvious greedy algorithm has a matching competitive ratio of $\frac{1}{2}$. By the "obvious algorithm" we mean: when a new vertex $w \in R$ arrives, match w to an arbitrary unmatched neighbor (or to no one, if it has no unmatched neighbors).

Proposition 2.1 *The deterministic greedy algorithm has a competitive ratio of $\frac{1}{2}$.*

Proof: The proposition is easy to prove directly, but here we'll give a more-sophisticated-than-necessary proof because it introduces ideas that we'll build on in the randomized case. Our proof uses a dual feasible solution as an upper bound on the size of a maximum matching. Recall the relevant primal-dual pair ((P) and (D), respectively):

$$\max \sum_{e \in E} x_e$$

subject to

$$\begin{aligned} \sum_{e \in \delta(v)} x_e &\leq 1 \qquad \text{for all } v \in L \cup R \\ x_e &\geq 0 \qquad \text{for all } e \in E, \end{aligned}$$

and

$$\min \sum_{v \in L \cup R} p_v$$

subject to

$$\begin{aligned} p_v + p_w &\geq 1 \qquad \text{for all } (v, w) \in E \\ p_v &\geq 0 \qquad \text{for all } v \in L \cup R. \end{aligned}$$

There are some minor differences with the primal-dual pair that we considered in Lecture #9, when we discussed the minimum-cost perfect matching problem. First, in (P), we're maximizing cardinality rather than minimizing cost. Second, we allow matchings that are not perfect, so the constraints in (P) are inequalities rather than equalities. This leads to the expected modifications of the dual: it is a minimization problem rather than a maximization problem, therefore with greater-than-or-equal-to constraints rather than less-than-or-equal-to constraints. Because the constraints in the primal are now inequality constraints, the dual variables are now nonnegative (rather than unrestricted).

We use these linear programs (specifically, the dual) only for the analysis; the algorithm, remember, is just the obvious greedy algorithm. We next define a "pre-dual solution" as follows: for every $v \in L \cup R$, set

$$q_v = \begin{cases} \frac{1}{2} & \text{if greedy matches } v \\ 0 & \text{otherwise.} \end{cases}$$

The q's are defined in hindsight, purely for the sake of analysis. Or if you prefer, we can imagine initializing all of the q_v's to 0 and then updating them in tandem with the execution

of the greedy algorithm — when the algorithm adds a vertex (v, w) to its matching, we set both q_v and q_w to $\frac{1}{2}$. (Since the chosen edges form a matching, a vertex has its q-value set to $\frac{1}{2}$ at most once.) This alternative description makes it clear that

$$|M| = \sum_{v \in L \cup R} q_v, \tag{1}$$

where M is the matching output by the greedy algorithm. (Whenever one edge is added to the matching, two vertices have their q-values increased to $\frac{1}{2}$.)

Next, observe that for every edge (v, w) of the final graph $(L \cup R, E)$, at least one of q_v, q_w is $\frac{1}{2}$ (if not both). For if $q_v = 0$, then v was not matched by the algorithm, which means that w had at least one unmatched neighbor when it arrived, which means the greedy algorithm matched it (presumably to some other unmatched neighbor) and hence $q_w = \frac{1}{2}$.

This observation does not imply that $\mathbf{q}$ is a feasible solution to the dual linear program (D), which requires a sum of at least 1 from the endpoints of every edge. But it does imply that after scaling up $\mathbf{q}$ by a factor 2 to obtain $\mathbf{p} = 2\mathbf{q}$, $\mathbf{p}$ is feasible for (D). Thus

$$|M| = \frac{1}{2} \underbrace{\sum_{v \in L \cup R} p_v}_{\text{obj fn of } \mathbf{p}} \geq \frac{1}{2} \cdot OPT,$$

where OPT denotes the size of the maximum matching in hindsight. The first equation is from (1) and the definition of $\mathbf{p}$, and the inequality is from weak duality (when the primal is a maximization problem, every feasible dual solution provides an upper bound on the optimum). ■

3 Online Fractional Bipartite Matching

3.1 The Problem

We won't actually discuss randomized algorithms in this lecture. Instead, we'll discuss a deterministic algorithm for the *fractional* bipartite matching problem. The keen reader will object that this is a stupid idea, because we've already seen that the fractional and integral bipartite matching problems are really the same.[2] While it's true that fractions don't help the optimal solution, *they do help an online algorithm*, intuitively by allowing it to "hedge." This is already evident in our simple bad example for deterministic algorithms (Figure 1). When w_1 shows up, in the integral case, a deterministic online algorithm has to match w_1 fully to either v_1 or v_2. But in the fractional case, it can match w_1 50/50 to both v_1 and v_2. Then when w_2 arrives, with only one neighbor on the left-hand side, it can at least be matched with a fractional value of $\frac{1}{2}$. The online algorithm produces a fractional matching

[2] In Lecture #9 we used the correctness of the Hungarian algorithm to argue that the fractional problem always has a 0-1 optimal solution (since the algorithm terminates with an integral solution and a dual-feasible solution with same objective function value). See also Exercise Set #5 for a direct proof of this.

with value $\frac{3}{2}$ while the optimal solution has size 2. So this only proves a bound of $\frac{3}{4}$ of the best-possible competitive ratio, leaving open the possibility of online algorithms with competitive ratio bigger than $\frac{1}{2}$.

3.2 The Water Level (WL) Algorithm

We consider the following "Water Level," algorithm, which is a natural way to define "hedging" in general.

Water-Level (WL) Algorithm

Physical metaphor:
think of each vertex $v \in L$ as a water container with a capacity of 1
think of each vertex $w \in R$ as a source of one unit of water

when $w \in R$ arrives:
drain water from w to its neighbors, always preferring the containers with the lowest current water level, until either
(i) all neighbors of w are full; or
(ii) w is empty (i.e., has sent all its water)

See also Figure 2 for a cartoon of the water being transferred to the neighbors of a vertex w. Initially the second neighbor has the lowest level so w only sends water to it; when the water level reaches that of the next-lowest (the fifth neighbor), w routes water at an equal rate to both the second and fifth neighbors; when their common level reaches that of the third neighbor, w routes water at an equal rate to these three neighbors with the lowest current water level. In this cartoon, the vertex w successfully transfers its entire unit of water (case (ii)).

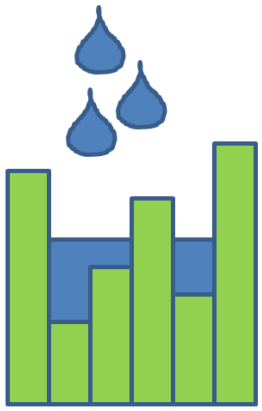

Figure 2: Cartoon of water being transferred to vertices.

For example, in the example in Figure 1, the WL algorithm replicates our earlier hedging, with vertex w_1 distributing its water equally between v_1 and v_2 (triggering case (ii)) and vertex w_2 distributing $\frac{1}{2}$ units of water to its unique neighbor (triggering case (i)).

This algorithm is natural enough, but all you'll have to remember for the analysis is the following key property.

Lemma 3.1 (Key Property of the WL Algorithm) *Let $(v, w) \in E$ be an edge of the final graph G and $y_v = \sum_{e \in \delta(v)} x_e$ the final water level of the vertex $v \in L$. Then w only sent water to containers when their current water level was y_v or less.*

Proof: Fix an edge (v, w) with $v \in L$ and $w \in R$. The lemma is trivial if $y_v = 1$, so suppose $y_v < 1$ — that the container v is not full at the end of the WL algorithm. This means that case (i) did not get triggered, so case (ii) was triggered, so the vertex w successfully routed all of its water to its neighbors. At the time when this transfer was completed, all containers to which w sent some water have a common level ℓ, and all other neighbors of w have current water level at least ℓ (cf., Figure 2). At the end of the algorithm, since water levels only increase, all neighbors of w have final water level ℓ or more. Since w only sent flow to containers when their current water level was ℓ or less, the proof is complete. ■

3.3 Analysis: A False Start

To prove a bound on the competitive ratio of the WL algorithm, a natural idea is to copy the same analysis approach that worked so well for the integral case (Proposition 2.1). That is, we define a pre-dual solution in tandem with the execution of the WL algorithm, and then scale it up to get a solution feasible for the dual linear program (D) in Section 2.

Idea #1:

- initialize $q_v = 0$ for all $v \in L \cup R$;
- whenever the amount x_{vw} of water sent from w to v goes up by Δ, increase both q_v and q_w by $\Delta/2$.

Inductively, this process maintains at all times that the value of the current fractional matching equals $\sum_{v \in L \cup R} q_v$. (Whenever the matching size increases by Δ, the sum of q-values increases by the same amount.)

The hope is that, for some constant $c > \frac{1}{2}$, the scaled-up vector $\mathbf{p} = \frac{1}{c}\mathbf{q}$ is feasible for (D). If this is the case, then we have proved that the competitive ratio of the WL algorithm is at least c (since its solution value equals c times the objective function value $\sum_{v \in L \cup R} p_v$ of the dual feasible solution $\mathbf{p}$, which in turn is an upper bound on the optimal matching size).

To see why this doesn't work, consider the example shown in Figure 3. Initially there are four vertices on the left-hand side. The first vertex $w_1 \in R$ is connected to every vertex of L, so the WL algorithm routes one unit of water evenly across the four edges. Now every container has a water level of $\frac{1}{4}$. The second vertex $w_2 \in R$ is connected to v_2, v_3, v_4. Since all neighbors have the same water level, w_2 splits its unit of water evenly between the three

containers, bringing their water levels up to $\frac{1}{4}+\frac{1}{3}=\frac{7}{12}$. The third vertex $w_3 \in R$ is connected only to v_3 and v_4. The vertex splits its water evenly between these two containers, but it cannot transfer all of its water; after sending $\frac{5}{12}$ units to each of v_3 and v_4, both containers are full (triggering case (i)). The last vertex $w_4 \in R$ is connected only to v_4. Since v_4 is already full, w_4 can't get rid of any of its water.

The question now is: by what factor to we have to scale up $\mathbf{q}$ to get a feasible solution $\mathbf{p} = \frac{1}{c}\mathbf{q}$ to (D)? Recall that dual feasibility boils down to the sum of p-values of the endpoints of every edge being at least 1. We can spot the problem by examining the edge (v_4, w_4). The vertex v_4 got filled, so its final q-value is $\frac{1}{2}$ (as high as it could be with the current approach). The vertex w_4 didn't participate in the fractional matching at all, so its q-value is 0. Since $q_{v_4} + q_{w_4} = \frac{1}{2}$, we would need to scale up by 2 to achieve dual feasibility. This does not improve over the competitive ration of $\frac{1}{2}$.

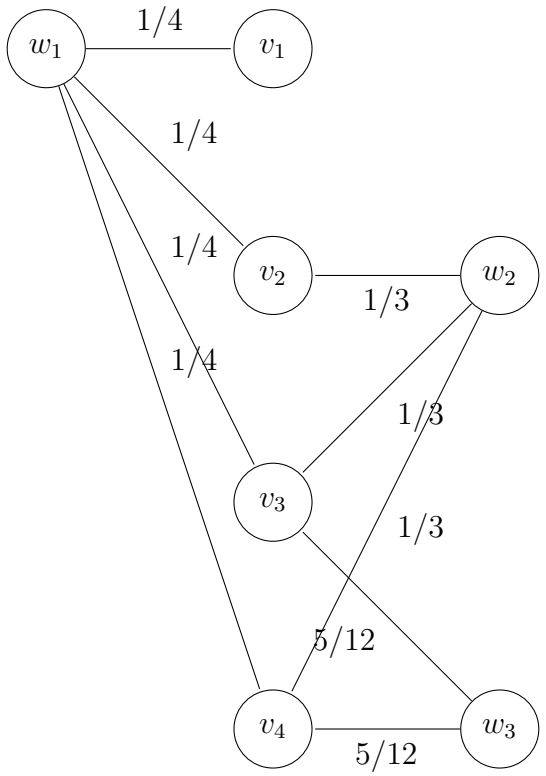

Figure 3: Example showcasing why Idea #1 does not work.

On the other hand, the solution computed by the WL algorithm for this example, while not optimal, is also not that bad. Its value is $1 + 1 + \frac{5}{6} + 0 = \frac{17}{6}$, which is substantially bigger than $\frac{1}{2}$ times the optimal solution (which is 4). Thus this is a bad example only for the analysis approach, and not for the WL algorithm itself. Can we keep the algorithm the same, and just be smarter with its analysis?

3.4 Analysis: The Main Idea

Idea #2: when the amount x_{vw} of water sent from w to v goes up by Δ, split the increase *unequally* between q_v and q_w.

To see the motivation for this idea, consider the bottom edge in Figure 3. The WL algorithm never sends any water on any edge incident to w_4, so it's hard to imagine how its q-value will wind up anything other than 0. So if we want to beat $\frac{1}{2}$, we need to make sure that v_4 finishes with a q-value bigger than $\frac{1}{2}$. A naive fix for this example would be to only increase the q-values for vertices of L, and not from R; but this would fail miserably if w_1 were the only vertex to arrive (then all q-values on the left would be $\frac{1}{4}$, all those on the right 0). To hedge between the various possibilities, as a vertex $v \in L$ gets more and more full, we will increase its q-value more and more quickly. Provided it increases quickly enough as v becomes full, it is conceivable that v could end up with a q-value bigger than $\frac{1}{2}$. Summarizing, we'll use unequal splits between the q-values of the endpoints of an edge, with the splitting ratio evolving over the course of the algorithm.

There are zillions of ways to split an increase of Δ on x_{vw} between q_v and q_w (as a function of v's current water level). The plan is to give a general analysis that is parameterized by such a "splitting function," and solve for the splitting function that leads to the best competitive ratio. Don't forget that all of this is purely for the analysis; the algorithm is always the WL algorithm.

So fix a nondecreasing "splitting function" $g : [0,1] \to [0,1]$. Then:

- initialize $q_v = 0$ for all $v \in L \cup R$;
- whenever the amount x_{vw} of water sent from w to v goes up by an infinitesimal amount dz, and the current water level of v is $y_v = \sum_{e \in \delta(v)} x_e$:
  - increase q_v by $g(y_v)dz$;
  - increase q_w by $(1 - g(y_v))dz$.

For example, if g is the constant function always equal to 0 (respectively, 1), then only the vertices of R (respectively, vertices of L) receive positive q-values. If g is the constant function always equal to $\frac{1}{2}$, then we recover our initial analysis attempt, with the increase on an edge split equally between its endpoints.

By construction, no matter how we choose the function g, we have

$$\text{current value of WL fractional matching} = \text{current value of } \sum_{v \in L \cup R} q_v,$$

at all times, and in particular at the conclusion of the algorithm.

For the analysis (parameterized by the choice of g), fix an arbitrary edge (v, w) of the final graph. We want a worst-case lower bound on $q_v + q_w$ (hopefully, bigger than $\frac{1}{2}$).

For the first case, suppose that at the termination of the WL algorithm, the vertex $v \in L$ is full (i.e., $y_v = \sum_{e \in \delta(v)} x_e = 1$). At the time that v's current water level was z, it accrued q-value at rate $g(z)$. Integrating over these accruals, we have

$$q_v + q_w \geq q_v = \int_0^1 g(z)dz. \tag{2}$$

(It may seem sloppy to throw out the contribution of $q_w \geq 0$, but Figure 3 shows that when v is full it might well be the case that some of its neighbors have q-value 0.) Note that the bigger the function g is, the bigger the lower bound in (2).

For the second case, suppose that v only has water level $y_v < 1$ at the conclusion of the WL algorithm. It follows that w successfully routed its entire unit of water to its neighbors (otherwise, the WL algorithm would have routed more water to the non-full container v). Here's where we use the key property of the WL algorithm (Lemma 3.1): whenever v sent water to a container, the current water level of that container was as most y_v. Thus, since the function g is nondecreasing, whenever v routed any water, it accrued q-value at rate at least $1 - g(y_v)$. Integrating over the unit of water sent, we obtain

$$q_w \geq \int_0^1 (1 - g(y_v))dz = 1 - g(y_v).$$

As in the first case, we have

$$q_v = \int_0^{y_v} g(z)dz$$

and hence

$$q_v + q_w \geq \left(\int_0^{y_v} g(z)dz \right) + 1 - g(y_v). \tag{3}$$

Note the lower bound in (3) is generally larger for smaller functions g (since $1 - g(y_v)$ is bigger). This is the tension between the two cases.

For example, if we take g to be identically 0, then the lower bounds (2) and (3) read 0 and 1, respectively. With g identically equal to 1, the values are reversed. With g identically equal to $\frac{1}{2}$, as in our initial attempt, the right-hand sides of both (2) and (3) are guaranteed to be at least $\frac{1}{2}$ (though not larger).

3.5 Solving for the Optimal Splitting Function

With our lower bounds (2) and (3) on the worst-case value of $q_v + q_w$ for an edge (v, w), our task is clear: we want to solve for the splitting function g that makes the minimum of these two lower bounds as large as possible. If we can find a function g such that the right-hand sides of (2) and (3) (for any $y_v \in [0, 1]$) are both at least c, then we will have proved that the WL algorithm is c-competitive. (Recall the argument: the value of the WL matching is $\sum_v q_v$, and $\mathbf{p} = \frac{1}{c}\mathbf{q}$ is a feasible dual solution, which is an upper bound on the maximum matching.)

Solving for the best nondecreasing splitting function g may seem an intimidating prospect — there are an infinite number of functions to choose from. In situations like this, a good strategy is to "guess and check" — try to develop intuition for what the right answer might look like and then verify your guess. There are many ways to guess, but often in an optimal analysis there is "no slack anywhere" (since otherwise, a better solution could take advantage of this slack). In our context, this corresponds to guessing that the optimal function g equalizes the lower bound in (2) with that in (3), and with the second lower bound tight simultaneously for all values of $y_v \in [0,1]$. There is no a priori guarantee that such a g exists, and if such a g exists, its optimality still needs to be verified. But it's still a good strategy for generating a guess.

Let's start with the guess that the lower bound in (3) is the same for all values of $y_v \in [0,1]$. This means that

$$\left(\int_0^{y_v} g(z)dz\right) + 1 - g(y_v),$$

when viewed as a function of y_v, is a constant function. This means its derivative (w.r.t. y_v) is 0, so

$$g(y_v) - g'(y_v) = 0,$$

i.e., the derivative of g is the same as g.[3] This implies that $g(z)$ has the form $g(z) = ke^z$ for a constant $k > 0$. This is great progress: instead of an infinite-dimensional g to solve for, we now just have the single parameter k to solve for.

Now let's use the guess that the two lower bounds in (2) and (3) are the same. Plugging ke^z into the lower bound in (2) gives

$$\int_0^1 ke^z dz = k\left[e^z\right]_0^1 = k(e-1),$$

which gets larger with k. Plugging ke^z into the lower bound in (3) gives (for any $y \in [0,1]$)

$$\int_0 ke^z dz + 1 - ke^y = k(e^y - 1) + 1 - ke^y = 1 - k.$$

This lower bound is independent of the choice of y — we knew that would happen, it's how we chose $g(z) = ke^z$ – and gets larger with smaller k. Equalizing the two lower bounds of $k(e-1)$ and $1-k$ and solving for k, we get $k = \frac{1}{e}$, and so the splitting function is $g(y) = e^{y-1}$. (Thus when a vertex $v \in L$ is empty it gets a $\frac{1}{e}$ share of the increase of an incident edge; the share increases as v gets more full, and approaches 100% as v becomes completely full.) Our lower bounds in (2) and (3) are then both equal to

$$1 - \frac{1}{e} \approx 63.2\%.$$

This proves that the WL algorithm is $(1 - \frac{1}{e})$-competitive, a significant improvement over the more obvious $\frac{1}{2}$-competitive algorithm.

[3] I don't know about you, but this is pretty much the only differential equation that I remember how to solve.

3.6 Epilogue

In this lecture we gave a $(1 - \frac{1}{e})$-competitive (deterministic) online algorithm for the online *fractional* bipartite matching problem. The same ideas can be used to design a randomized online algorithm for the original *integral* online bipartite matching problem that always outputs a matching with expected size at least $1 - \frac{1}{e}$ times the maximum possible. (The expectation is over the random coin flips made by the algorithm.) The rough idea is to set things up so that the probability that a given edge is included the matching plays the same role as its fractional value in the WL algorithm. Implementing this idea is not trivial, and the details are outlined in Problem Set #4.

But can we do better? Either with a smarter algorithm, or with a smarter analysis of these same algorithms? (Recall that being smarter improved the analysis of the WL algorithm from a $\frac{1}{2}$ to a $1-\frac{1}{e}$.) Even though $1-\frac{1}{e}$ may seem like a weird number, the answer is negative: *no* online algorithm, deterministic or randomized, has a competitive ratio better than $1 - \frac{1}{e}$ for maximum bipartite matching. The details of this argument are outlined in Problem Set #3.

CS261: A Second Course in Algorithms
Lecture #15: Introduction to Approximation Algorithms*

Tim Roughgarden†

February 23, 2016

1 Coping with NP-Completeness

All of CS161 and the first half of CS261 focus on problems that can be solved in polynomial time. A sad fact is that many practically important and frequently occurring problems do not seem to be polynomial-time solvable, that is, are NP-hard.[1]

As an algorithm designer, what does it mean if a problem is NP-hard? After all, a real-world problem doesn't just go away after you realize that it's NP-hard. The good news is that NP-hardness is not a death sentence — it doesn't mean that you can't do anything practically useful. But NP-hardness does throw the gauntlet to the algorithm designer, and suggests that compromises may be necessary. Generally, more effort (computational and human) will lead to better solutions to NP-hard problems. The right effort vs. solution quality trade-off depends on the context, as well as the relevant problem size. We'll discuss algorithmic techniques across the spectrum — from low-effort decent-quality approaches to high-effort high-quality approaches.

So what are some possible compromises? First, you can restrict attention to a relevant special case of an NP-hard problem. In some cases, the special case will be polynomial-time solvable. (Example: the Vertex Cover problem is NP-hard in general graphs, but on Problem Set #2 you proved that, in bipartite graphs, the problem reduces to max flow/min cut.) In other cases, the special case remains NP-hard but is still easier than the general case. (Example: the Traveling Salesman Problem in Lecture #16.) Note that this approach requires non-trivial human effort — implementing it requires understanding and articulating

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

[1] I will assume that you're familiar with the basics of NP-completeness from your other courses, like CS154. If you want a refresher, see the videos on the Course site.

whatever special structure your particular application has, and then figuring out how to exploit it algorithmically.

A second compromise is to spend more than a polynomial amount of time solving the problem, presumably using tons of hardware and/or restricting to relatively modest problem sizes. Hopefully, it is still possible to achieve a running time that is faster than naive brute-force search. While NP-completeness is sometimes interpreted as "there's probably nothing better than brute-force search," the real story is more nuanced. Many NP-complete problems can be solved with algorithms that, while running in exponential time, are significantly faster than brute-force search. Examples that we'll discuss later include 3SAT (with a running time of $(4/3)^n$ rather than 2^n) and the Traveling Salesman Problem (with a running time of 2^n instead of $n!$). Even for NP-hard problems where we don't know any algorithms that provably beat brute-force search in the worst case, there are almost always speed-up tricks that help a lot in practice. These tricks tend to be highly dependent on the particular application, so we won't really talk about any in CS261 (where the focus is on general techniques).

A third compromise, and the one that will occupy most of the rest of the course, is to relax correctness. For an optimization problem, this means settling for a feasible solution that is only approximately optimal. Of course one would like the approximation to be as good as possible. Algorithms that are guaranteed to run in polynomial time and also be near-optimal are called *approximation algorithms*, and they are the subject of this and the next several lectures.

2 Approximation Algorithms

In approximation algorithm design, the hard constraint is that the designed algorithm should run in polynomial time on every input. For an NP-hard problem, assuming $P \neq NP$, this necessarily implies that the algorithm will compute a suboptimal solution in some cases. The obvious goal is then to get as close to an optimal solution as possible (ideally, on every input).

There is a massive literature on approximation algorithms — a good chunk of the algorithms research community has been obsessed with them for the past 25+ years. As a result, many interesting design techniques have been developed. We'll only scratch the surface in our lectures, and will focus on the most broadly useful ideas and problems.

One take-away from our study of approximation algorithms is that the entire algorithmic toolbox that you've developed during CS161 and CS261 remains useful for the design and analysis of approximation algorithms. For example, greedy algorithms, divide and conquer, dynamic programming, and linear programming all have multiple killer applications in approximation algorithms (we'll see a few). And there are other techniques, like local search, which usually don't yield exact algorithms (even for polynomial-time solvable problems) but seem particularly well suited for designing good heuristics.

The rest of this lecture sets the stage with four relatively simple approximation algorithms for fundamental NP-hard optimization problems.

2.1 Example: Minimum-Makespan Scheduling

We've already seen a couple of examples of approximation algorithms in CS261. For example, recall the problem of minimum-makespan scheduling, which we studied in Lecture #13. There are m identical machines, and n jobs with processing times $p_1, \ldots, p_n$. The goal is to schedule all of the jobs to minimize the makespan (the maximum load, where the load of a machine is the sum of the processing times of the jobs assigned to it) — to balance the loads of the machines as evenly as possible.

In Lecture #13, we studied the online version of this problem, with jobs arriving one-by-one. But it's easy to imagine applications where you get to schedule a batch of jobs all at once. This is the offline version of the problem, with all n jobs known up front. This problem is NP-hard.[2]

Recall Graham's algorithm, which processes the jobs in the given (arbitrary) order, always scheduling the next job on the machine that currently has the lightest load. This algorithm can certainly be implemented in polynomial time, so we can reuse it as a legitimate approximation algorithm for the offline problem. (Now the fact that it processes the jobs online is just a bonus.) Because it always produces a schedule with makespan at most twice the minimum possible (as we proved in Lecture #13), it is a *2-approximation algorithm*. The factor "2" here is called the *approximation ratio* of the algorithm, and it plays the same role as the competitive ratio in online algorithms.

Can we do better? We can, by exploiting the fact that an (offline) algorithm knows all of the jobs up front. A simple thing that an offline algorithm can do that an online algorithm cannot is sort the jobs in a favorable order. Just running Graham's algorithm on the jobs in order from largest to smallest already improves the approximation ratio to $\frac{4}{3}$ (a good homework problem).

2.2 Example: Knapsack

Another example that you might have seen in CS161 (depending on who you took it from) is the Knapsack problem. We'll just give an executive summary; if you haven't seen this material before, refer to the videos posted on the course site.

An instance of the Knapsack problem is n items, each with a value and a weight. Also given is a capacity W. The goal is to identify the subset of items with the maximum total value, subject to having total weight at most W. The problem gets its name from a silly story of a burglar trying to fill up a sack with the most valuable items. But the problem comes up all the time, either directly or as a subroutine in a more complicated problem — whenever you have a shared resource with a hard capacity, you have a knapsack problem.

Students usually first encounter the Knapsack problem as a killer application of dynamic programming. For example, one such algorithm, which works as long as all item weights

[2]For the most part, we won't bother to prove any NP-hardness results in CS261. The NP-hardness proofs are all of the exact form that you studied in a course like CS154 — one just exhibits a polynomial-time reduction from a known NP-hard problem to the current problem. Many of the problems that we study were among the first batch of NP-complete problems identified by Karp in 1972.

are integers, runs in time $O(nW)$. Note that this is not a polynomial-time algorithm, since the input size (the number of keystrokes needed to type in the input) is only $O(n \log W)$. (Writing down the number W only takes $\log W$ digits.) And in fact, the knapsack problem is NP-hard, so we don't expect there to be a polynomial-time algorithm. Thus the $O(nW)$ dynamic programming solution is an example of an algorithm for an NP-hard problem that beats brute-force search (unless W is exponential in n), while still running in time exponential in the input size.

What if we want a truly polynomial-time algorithm? NP-hardness says that we'll have to settle for an approximation. A natural greedy algorithm, which processes the items in order of value divided by size ("bang-per-buck") achieves a $\frac{1}{2}$-approximation, that is, is guaranteed to output a feasible solution with total value at least 50% times the maximum possible.[3] If you're willing to work harder, then by rounding the data (basically throwing out the lower-order bits) and then using dynamic programming (on an instance with relatively small numbers), one obtains a $(1-\epsilon)$-approximation, for a user-specified parameter $\epsilon > 0$, in time polynomial in n and $\frac{1}{\epsilon}$. (By NP-hardness, we expect the running time to blow up as ϵ gets close to 0.) This is pretty much the best-case scenario for an NP-hard problem — arbitrarily close approximation in polynomial time.

2.3 Example: Steiner Tree

Next we revisit the other problem that we studied in Lecture #13, the Steiner tree problem. Recall that the input is an undirected graph $G = (V, E)$ with a nonnegative cost $c_e \geq 0$ for each edge $e \in E$. Recall also that there is no loss of generality in assuming that G is the complete graph and that the edge costs satisfy the triangle inequality (i.e., $c_{uw} \leq c_{uv} + c_{vw}$ for all $u, v, w \in V$); see Exercise Set #7. Finally, there is a set $R = \{t_1, \ldots, t_k\}$ of vertices called "terminals." The goal is to compute the minimum-cost subgraph that spans all of the terminals. We previously studied this problem with the terminals arriving online, but the offline version of the problem, with all terminals known up front, also makes perfect sense.

In Lecture #13 we studied the natural greedy algorithm for the online Steiner tree problem, where the next terminal is connected via a direct edge to a previously arriving terminal in the cheapest-possible way. We proved that the algorithm always computes a Steiner tree with cost at most $2 \ln k$ times the best-possible solution in hindsight. Since the algorithm is easy to implement in polynomial time, we can equally well regard it as a $2 \ln k$-approximation algorithm (with the fact that it processes terminals online just a bonus). Can we do something smarter if we know all the terminals up front?

As with job scheduling, better bounds are possible in the offline model because of the ability to sort the terminals in a favorable order. Probably the most natural order in which to process the terminals is to always process next the terminal that is the cheapest to connect to a previous terminal. If you think about it a minute, you realize that this is equivalent to running Prim's MST algorithm on the subgraph induced by the terminals. This motivates:

[3] Technically, to achieve this for every input, the algorithm takes the better of this greedy solution and the maximum-value item.

The MST heuristic for metric Steiner tree: output the minimum spanning tree of the subgraph induced by the terminals.

Since the Steiner tree problem is NP-hard and the MST can be computed in polynomial time, we expect this heuristic to produce a suboptimal solution in some cases. A concrete example is shown in Figure 1, where the MST of $\{t_1, t_2, t_3\}$ costs 4 while the optimal Steiner tree has cost 3. (Thus the cost can be decreased by spanning additional vertices; this is what makes the Steiner tree problem hard.) Using larger "wheel" graphs of the same type, it can be shown that the MST heuristic can be off by a factor arbitrarily close to 2 (Exercise Set #8). It turns out that there are no worse examples.

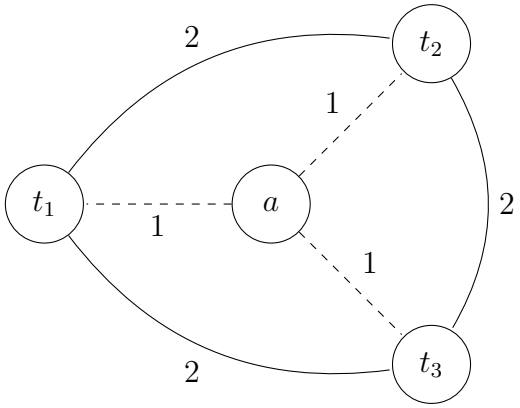

Figure 1: MST heuristic will pick $\{t_1, t_2\}, \{t_2, t_3\}$ but best Steiner tree (dashed edges) is $\{a, t_1\}, \{a, t_2\}, \{a, t_3\}$.

Theorem 2.1 *In the metric Steiner tree problem, the cost of the minimum spanning tree of the terminals is always at most twice the cost of an optimal solution.*

Proof: The proof is similar to our analysis of the online Steiner tree problem (Lecture #13), only easier. It's easier to relate the cost of the MST heuristic to that of an optimal solution than for the online greedy algorithm — the comparison can be done in one shot, rather then on an edge-by-edge basis.

For the analysis, let T^* denote a minimum-cost Steiner tree. Obtain H from T^* by adding a second copy of every edge (Figure 2(a)). Obviously, H is Eulerian (every vertex degree got doubled) and $\sum_{e \in H} c_e = 2OPT$. Let C denote an Euler tour of H — a (non-simple) closed walk using every edge of H exactly once. We again have $\sum_{e \in C} c_e = 2OPT$.

The tour C visits each of $t_1, \ldots, t_k$ at least once. "Shortcut" it to obtain a simple cycle $\widehat{C}$ on the vertex set $\{t_1, \ldots, t_k\}$ (Figure 2(b)); since the edge costs satisfy the triangle inequality, this only decreases the cost. $\widehat{C}$ minus an edge is a spanning tree of the subgraph induced by R that has cost at most $2OPT$; the MST can only be better. ■

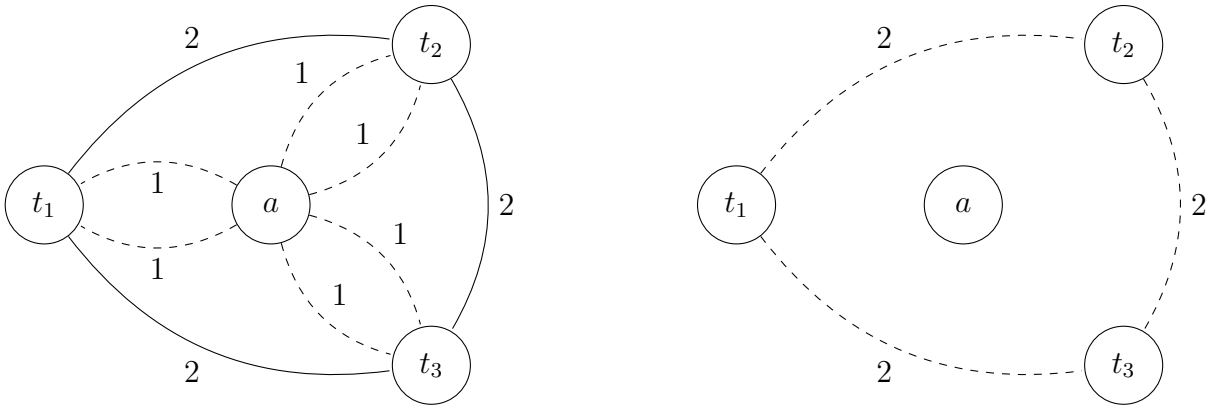

Figure 2: (a) Adding second copy of each edge in T^* to form H. Note H is Eulerian. (b) Shorting cutting edges $(\{t1, a\}, \{a, t2\})$, $(\{t2, a\}, \{a, t3\})$, $(\{t3, a\}, \{a, t1\})$ to $\{t1, t2\}, \{t2, t3\}, \{t3, t1\}$ respectively.

2.4 Example: Set Coverage

Next we study a problem that we haven't seen before, *set coverage*. This problem is a killer application for greedy algorithms in approximation algorithm design. The input is a collection $S_1, \ldots, S_m$ of subsets of some ground set U (each subset described by a list of its elements), and a budget k. The goal is to pick k subsets to maximize the size of their union (Figure 3). All else being equal, bigger sets are better for the set coverage problem. But it's not so simple — some sets are largely redundant, while others are uniquely useful (cf., Figure 3).

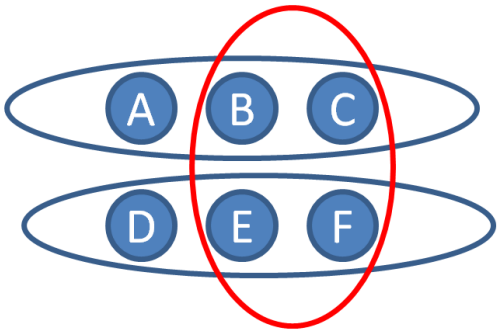

Figure 3: Example set coverage problem. If $k = 2$, we should pick the blue sets. Although the red set is the largest, picking it is redundant.

Set coverage is a basic problem that comes up all the time (often not even disguised). For example, suppose your start-up only has the budget to hire k new people. Each applicant can be thought of as a set of skills. The problem of hiring to maximize the number of distinct

skills required is a set coverage problem. Similarly for choosing locations for factories/fire engines/Web caches/artisinal chocolate shops to cover as many neighborhoods as possible. Or, in machine learning, picking a small number of features to explain as much as the data as possible. Or, in HCI, given a budget on the number of articles/windows/menus/etc. that can be displayed at any given time, maximizing the coverage of topics/functionality/etc.

The set coverage problem is NP-hard. Turning to approximation algorithms, the following greedy algorithm, which increases the union size as much as possible at each iteration, seems like a natural and good idea.

Greedy Algorithm for Set Coverage

for $i = 1, 2, \ldots, k$: **do**
    compute the set A_i maximizing the number of new elements covered (relative to $\cup_{j=1}^{i-1} A_j$)
return $\{A_1, \ldots, A_k\}$

This algorithm can clearly be implemented in polynomial time, so we don't expect it to always compute an optimal solution. It's useful to see some concrete examples of what can go wrong. example.

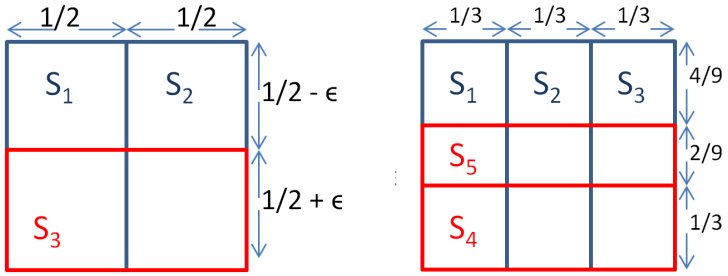

Figure 4: (a) Bad example when $k = 2$ (b) Bad example when $k = 3$.

For the first example (Figure 4(a)), set the budget $k = 2$. There are three subsets. S_1 and S_2 partition the ground set U half-half, so the optimal solution has size $|U|$. We trick the greedy algorithm by adding a third subset S_3 that covers slightly more than half the elements. The greedy algorithm then picks S_3 in its first iteration, and can only choose one of S_1, S_2 in the second iteration (it doesn't matter which). Thus the size of the greedy solution is $\approx \frac{3}{4}|U|$. Thus even when $k = 2$, the best-case scenario would be that the greedy algorithm is a $\frac{3}{4}$-approximation.

We next extend this example (Figure 4(b)). Take $k = 3$. Now the optimal solution is S_1, S_2, S_3, which partition the ground set into equal-size parts. To trick the greedy algorithm in the first iteration (i.e., prevent it from taking one of the optimal sets S_1, S_2, S_3), we add a set S_4 that covers slightly more than $\frac{1}{3}$ of the elements and overlaps evenly with S_1, S_2, S_3. To trick it again in the second iteration, note that, given S_4, choosing any of S_1, S_2, S_3 would

cover $\frac{1}{3} \cdot \frac{2}{3} \cdot |U| = \frac{2}{9}|U|$ new elements. Thus we add a set S_5, disjoint from S_4, covering slightly more than a $\frac{2}{9}$ fraction of U. In the third iteration we allow the greedy algorithm to pick one of S_1, S_2, S_3. The value of the greedy solution is $\approx |U|(\frac{1}{3} + \frac{2}{9} + \frac{1}{3}\frac{4}{9}) = \frac{19}{27}|U|$. This is roughly 70% of $|U|$, so it is a worse example for the greedy algorithm than the first

Exercise Set #8 asks you to extend this family of bad examples to show that, for all k, the greedy solution could be as small as

$$1 - \left(1 - \frac{1}{k}\right)^k$$

times the size of an optimal solution. (Note that with $k = 2, 3$ we get $\frac{3}{4}$ and $\frac{19}{27}$.) This expression is decreasing with k, and approaches $1 - \frac{1}{e} \approx 63.2\%$ in the limit (since $1 - x$ approaches e^{-x} for x going to 0, recall Figure 5).[4]

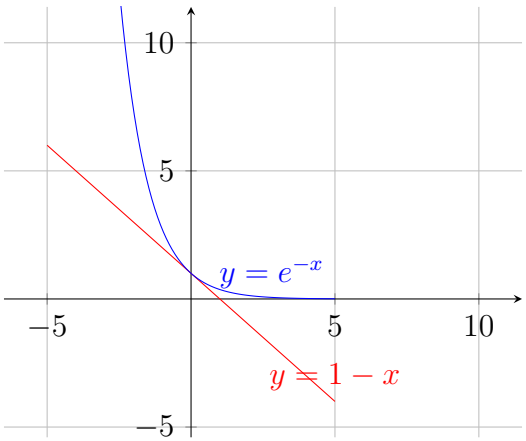

Figure 5: Graph showing $1 - x$ approaching e^{-x} for small x.

These examples show that the following guarantee is remarkable.

Theorem 2.2 *For every $k \geq 1$, the greedy algorithm is a $(1 - (1 - \frac{1}{k})^k)$-approximation algorithm for set coverage instances with budget k.*

Thus there are no worse examples for the greedy algorithm than the ones we identified above. Here's what's even more amazing: under standard complexity assumptions, there is *no* polynomial-time algorithm with a better approximation ratio![5] In this sense, the greedy algorithm is an optimal approximation algorithm for the set coverage problem.

We now turn to the proof of Theorem 2.2. The following lemma proves a sense in which the greedy algorithm makes healthy progress at every step. (This is the most common way to analyze a greedy algorithm, whether for exact or approximate guarantees.)

[4] There's that strange number again!

[5] As k grows large, that is. When k is a constant, the problem can be solved optimally in polynomial time using brute-force search.

Lemma 2.3 *Suppose that the first $i-1$ sets $A_1, \ldots, A_{i-1}$ computed by the greedy algorithm cover ℓ elements. Then the next set A_i chosen by the algorithm covers at least*

$$\frac{1}{k}\left(OPT - \ell\right)$$

new elements, where OPT is the value of an optimal solution.

Proof: As a thought experiment, suppose that the greedy algorithm were allowed to pick k new sets in this iteration. Certainly it could cover $OPT - \ell$ new elements — just pick all of the k subsets in the optimal solution. One of these k sets must cover at least $\frac{1}{k}(OPT - \ell)$ new elements, and the set A_i chosen by the greedy algorithm is at least as good. ■

Now we just need a little algebra to prove the approximation guarantee.

Proof of Theorem 2.2: Let $g_i = |\cup_{j=1}^{i} A_i|$ denote the number of elements covered by the greedy solution after i iterations. Applying Lemma 2.3, we get

$$g_k = (g_k - g_{k-1}) + g_{k-1} \geq \frac{1}{k}(OPT - g_{k-1}) + g_{k-1} = \frac{OPT}{k} + \left(1 - \frac{1}{k}\right) g_{k-1}.$$

Applying it again we get

$$g_k \geq \frac{OPT}{k} + \left(1 - \frac{1}{k}\right)\left(\frac{OPT}{k} + \left(1 - \frac{1}{k}\right) g_{k-2}\right) = \frac{OPT}{k} + \left(1 - \frac{1}{k}\right)\frac{OPT}{k} + \left(1 - \frac{1}{k}\right)^2 g_{k-3}.$$

Iterating, we wind up with

$$g_k \geq \frac{OPT}{k}\left[1 + \left(1 - \frac{1}{k}\right) + \left(1 - \frac{1}{k}\right)^2 + \cdots + \left(1 - \frac{1}{k}\right)^{k-1}\right].$$

(There are k terms, one per iteration of the greedy algorithm.) Recalling from your discrete math class the identity

$$1 + z + z^2 + \cdots + z^{k-1} = \frac{1 - z^k}{1 - z}$$

for $z \in (0, 1)$ — just multiply both sides by $1 - z$ to verify — we get

$$g_k \geq \frac{OPT}{k} \cdot \frac{1 - (1 - \frac{1}{k})^k}{1 - (1 - \frac{1}{k})} = OPT\left[1 - \left(1 - \frac{1}{k}\right)^k\right],$$

as desired. ■

2.5 Influence Maximization

Guarantees for the greedy algorithm for set coverage and various generalizations were already known in the 1970s. But just over the last dozen years, these ideas have taken off in the data mining and machine learning communities. We'll just mention one representative and influential (no pun intended) example, due to Kempe, Kleinberg, and Tardos in 2003.

Consider a "social network," meaning a directed graph $G = (V, E)$. For our purposes, we interpret an edge (v, w) as "v influences w." (For example, maybe w follows v on Twitter.)

We next posit a simple model of how an idea/news item/meme/etc. "goes viral," called a "cascade model."[6]

- Initially the vertices in some set S are "active," all other vertices are "inactive." Every edge is initially "undetermined."
- While there is an active vertex v and an undetermined edge (v, w):
  - with probability p, edge (v, w) is marked "active," otherwise it is marked "inactive;"
  - if (v, w) is active and w is inactive, then mark w as active.

Thus whenever a vertex gets activated, it has the opportunity to active all of the vertices that it influences (if they're not already activated). Note that once a vertex is activated, it is active forevermore. A vertex can get multiple chances to be activated, corresponding to the number of its influencers who get activated. See Figure 6. In the example, note that a vertex winds up getting activated if and only if there is a path of activated edges from v to it.

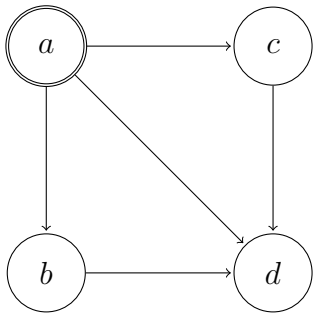

Figure 6: Example cascade model. Initially, only a is activated. b (and similarly c) can get activated by a with probability p. d has a chance to get activated by either a, b or c.

The *influence maximization problem* is, given a directed graph $G = (V, E)$ and a budget k, to compute the subset $S \subseteq V$ of size k that maximizes the expected number of active vertices at the conclusion of the cascade, given that the vertices of S are active at the beginning.

[6]Such models were originally proposed in epidemiology, to understand the spread of diseases.

(The expectation is over the coin flips made for the edges.) Denote this expected value for a set S by $f(S)$.

There is a natural greedy algorithm for influence maximization, where at each iteration we increase the function f as much as possible.

Greedy Algorithm for Influence Maximization

$S = \emptyset$
for $i = 1, 2, \ldots, k$: **do**
    add to S the vertex v maximizing $f(S \cup \{v\})$
return S

The same analysis we used for set coverage can be used to prove that this greedy algorithm is a $(1 - (1 - \frac{1}{k})^k)$-approximation algorithm for influence maximization. The greedy algorithm's guarantee holds for every function f that is "monotone" and "submodular," and the function f above is one such example (it is basically a convex combination of set coverage functions). See Problem Set #4 for details.

CS261: A Second Course in Algorithms
Lecture #16: The Traveling Salesman Problem*

Tim Roughgarden†

February 25, 2016

1 The Traveling Salesman Problem (TSP)

In this lecture we study a famous computational problem, the *Traveling Salesman Problem (TSP)*. For roughly 70 years, the TSP has served as the best kind of challenge problem, motivating many different general approaches to coping with NP-hard optimization problems. For example, George Dantzig (who you'll recall from Lecture #10) spent a fair bit of his time in the 1950s figuring out how to use linear programming as a subroutine to solve ever-bigger instances of TSP. Well before the development of NP-completeness in 1971, experts were well aware that the TSP is a "hard" problem in some sense of the word.

So what's the problem? The input is a complete undirected graph $G = (V, E)$, with a nonnegative cost $c_e \geq 0$ for each edge $e \in E$. By a *TSP tour*, we mean a simple cycle that visits each vertex exactly once. (Not to be confused with an Euler tour, which uses each edge exactly once.) The goal is to compute the TSP tour with the minimum total cost. For example, in Figure 1, the optimal objective function value is 13.

The TSP gets its name from a silly story about a salesperson who has to make a number of stops, and wants to visit them all in an optimal order. But the TSP definitely comes up in real-world scenarios. For example, suppose a number of tasks need to get done, and between two tasks there is a setup cost (from, say, setting up different equipment or locating different workers). Choosing the order of operations so that the tasks get done as soon as possible is exactly the TSP. Or think about a scenario where a disk has a number of outstanding read requests; figuring out the optimal order in which to serve them again corresponds to TSP.

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

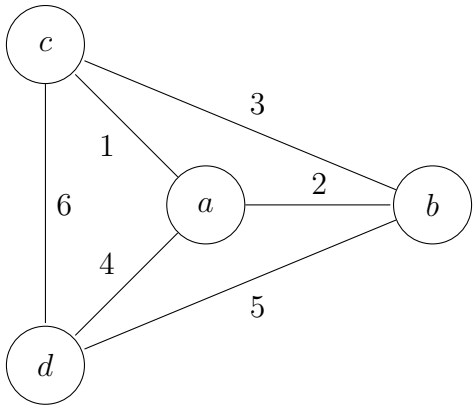

Figure 1: Example TSP graph. Best TSP tour is *a*-*c*-*b*-*d*-*a* with cost 13.

The TSP is hard, even to approximate.

Theorem 1.1 *If $P \neq NP$, then there is no α-approximation algorithm for the TSP (for any α).*

Recall that an α-approximation algorithm for a minimization problem runs in polynomial time and always returns a feasible solution with cost at most α times the minimum possible.

Proof of Theorem 1.1: We prove the theorem using a reduction from the Hamiltonian cycle problem. The Hamiltonian cycle problem is: given an undirected graph, does it contain a simple cycle that visits every vertex exactly once? For example, the graph in Figure 2 does not have a Hamiltonian cycle.[1] This problem is NP-complete, and usually one proves it in a course like CS154 (e.g., via a reduction from 3SAT).

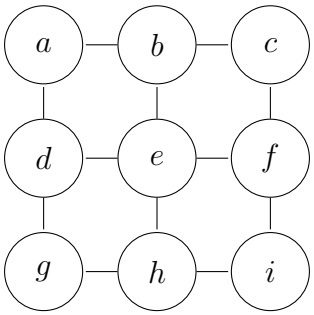

Figure 2: Example graph without Hamiltonian cycle.

[1]While it's generally difficult to convince someone that a graph has no Hamiltonian cycle, in this case there is a slick argument: color the four corners and the center vertex green, and the other four vertices red. Then every closed walk alternates green and red vertices, so a Hamiltonian cycle would have the same number of green and red vertices (impossible, since there are 9 vertices).

For the reduction, we need to show how to use a good TSP approximation algorithm to solve the Hamiltonian cycle problem. Given an instance $G = (V, E)$ of the latter problem, we transform it into an instance $G' = (V', E', c)$ of TSP, where:

- $V' = V$;
- E' is all edges (so (V', E') is the complete graph);
- for each $e \in E'$, set

$$c_e = \begin{cases} 1 & \text{if } e \in E \\ > \alpha \cdot n & \text{if } e \notin E, \end{cases}$$

  where n is the number of vertices and α is the approximation factor that we want to rule out.

For example, in Figure 2, all the edges of the grid get a cost of 1, and all the missing edges get a cost greater than αn.

The key point is that there is a one-to-one correspondence between the Hamiltonian cycles of G and the TSP tours of G' that use only unit-cost edges. Thus:

(i) If G has a Hamiltonian cycle, then there is a TSP tour with total cost n.

(ii) If G has no Hamiltonian cycle, then every TSP tour has cost larger than αn.

Now suppose there were an α-approximation algorithm $\mathcal{A}$ for the TSP. We could use $\mathcal{A}$ to solve the Hamiltonian cycle problem: given an instance G of the problem, run the reduction above and then invoke $\mathcal{A}$ on the produced TSP instance. Since there is more than an α factor gap between cases (i) and (ii) and $\mathcal{A}$ is an α-approximation algorithm, the output of $\mathcal{A}$ indicates whether or not G is Hamiltonian. (If yes, then it must return a TSP tour with cost at most αn; if no, then it can only return a TSP tour with cost bigger than αn.) ■

2 Metric TSP

2.1 Toward a Tractable Special Case

Theorem 1.1 indicates that, to prove anything interesting about approximation algorithms for the TSP, we need to restrict to a special case of the problem. In the *metric TSP*, we assume that the edge costs satisfy the triangle inequality (with $c_{uw} \leq c_{uv} + c_{vw}$ for all $u, v, w \in V$). We previously saw the triangle inequality when studying the Steiner tree problem (Lectures #13 and #15). The big difference is that in the Steiner tree problem the metric assumption is without loss of generality (see Exercise Set #7) while in the TSP it makes the problem significantly easier.[2]

The metric TSP problem is still NP-hard, as shown by a variant of the proof of Theorem 1.1. We can't use the big edge costs αn because this would violate the triangle inequality.

[2]This is of course what we're hoping for, because the general case is impossible to approximate.

But if we use edge costs of 2 for edges not in the given Hamiltonian cycle instance G, then the triangle inequality holds trivially (why?). The optimal TSP tour still has value at most n when G has a Hamiltonian cycle, and value at least $n+1$ when it does not. This shows that there is no exact polynomial-time algorithm for metric TSP (assuming $P \neq NP$). It does not rule out good approximation algorithms, however. And we'll see next that there are pretty good approximation algorithms for metric TSP.

2.2 The MST Heuristic

Recall that in approximation algorithm design and analysis, the challenge is to relate the solution output by an algorithm to the optimal solution. The optimal solution itself is often hard to get a handle on (its NP-hard to compute, after all), so one usually resorts to bounds on the optimal objective function value — quantities that are "only better than optimal." Here's a simple lower bound for the TSP, with or without the triangle inequality.

Lemma 2.1 *For every instance $G = (V, E, c)$, the minimum-possible cost of a TSP tour is at least the cost of a minimum spanning tree (MST).*

Proof: Removing an edge from the minimum-cost TSP tour yields a spanning tree with only less cost. The minimum spanning tree can only have smaller cost. ■

Lemma 2.1 motivates using the MST as a starting point for building a TSP tour — if we can turn the MST into a tour without suffering too much extra cost, then the tour will be near-optimal. The idea of transforming a tree into a tour should ring some bells — recall our online (Lecture #13) and offline (Lecture #15) algorithms for the Steiner tree problem. We'll reuse the ideas developed for Steiner tree, like doubling and shortcutting, here for the TSP. The main difference is that while these ideas were used only in the *analysis* of our Steiner tree algorithms, to relate the cost of our algorithm's tree to the minimum-possible cost, here we'll use these ideas in the *algorithm* itself. This is because, in TSP, we have to output a tour rather than a tree.

MST Heuristic for Metric TSP

compute the MST T of the input G
construct the graph H by doubling every edge of T
compute an Euler tour C of H
`// every` $v \in V$ `is visited at least once in` C
shortcut repeated occurrences of vertices in C to obtain a TSP tour

When we studied the Steiner tree problem, steps 2–4 were used only in the analysis. But all of these steps, and hence the entire algorithm, are easy to implement in polynomial (even near-linear) time.[3]

[3]Recall from CS161 that there are many fast algorithms for computing a MST, including Kruskal's and Prim's algorithms.

Theorem 2.2 *The MST heuristic is a 2-approximation algorithm for the metric TSP.*

Proof: We have

$$\begin{aligned} \text{cost of our TSP tour} &\leq \text{cost of } C \\ &= \sum_{e \in H} c_e \\ &= 2 \sum_{e \in T} c_e \\ &\leq 2 \cdot \text{cost of optimal TSP tour,} \end{aligned}$$

where the first inequality holds because the edge costs obey the triangle inequality, the second equation holds because the Euler tour C uses every edge of H exactly once, the third equation follows from the definition of H, and the final inequality follows from Lemma 2.1. ■

The analysis of the MST heuristic in Theorem 2.2 is tight — for every constant $c < 2$, there is a metric TSP instance such that the MST heuristic outputs a tour with cost more than c times that of an optimal tour (Exercise Set #8).

Can we do better with a different algorithm? This is the subject of the next section.

2.3 Christofides's Algorithm

Why were we off by a factor of 2 in the MST heuristic? Because we doubled every edge of the MST T. Why did we double every edge? Because we need an Eulerian graph, to get an Euler tour that we can shortcut down to a TSP tour. But perhaps it's overkill to double every edge of the MST. Can we augment the MST T to get an Eulerian graph without paying the full cost of an optimal solution?

The answer is yes, and the key is the following slick lemma. It gives a second lower bound on the cost of an optimal TSP tour, complementing Lemma 2.1.

Lemma 2.3 *Let $G = (V, E)$ be a metric TSP instance. Let $S \subseteq V$ be an even subset of vertices and M a minimum-cost perfect matching of the (complete) graph induced by S. Then*

$$\sum_{e \in M} c_e \leq \frac{1}{2} \cdot OPT,$$

where OPT denotes the cost of an optimal TSP tour.

Proof: Fix S. Let C^* denote an optimal TSP tour. Since the edges obey the triangle inequality, we can shortcut C^* to get a tour C_S of S that has cost at most OPT. Since $|S|$ is even, C_S is a (simple) cycle of even length (Figure 3). C_S is the union of two disjoint perfect matchings (alternate coloring the edges of C_S red and green). Since the sum of the costs of these matchings is that of C_S (which is at most OPT), the cheaper of these two matchings has cost at most $OPT/2$. The minimum-cost perfect matching of S can only be cheaper. ■

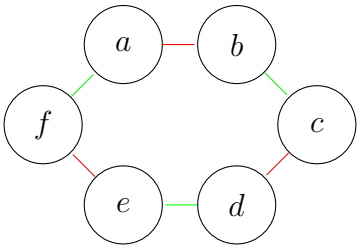

Figure 3: C_S is a simple cycle of even length representing union of two disjoint perfect matchings (red and green).

Lemma 2.3 brings us to *Christofides's algorithm*, which differs from the MST heuristic only in substituting a perfect matching computation in place of the doubling step.

Christofides's Algorithm

compute the MST T of the input G
compute the set W of vertices with odd degree in T
compute a minimum-cost perfect matching M of W
construct the graph H by adding M to T
compute an Euler tour C of H
`//` every $v \in V$ `is visited at least once in` C
shortcut repeated occurrences of vertices in C to obtain a TSP tour

In the second step, the set W always has even size. (The sum of the vertex degrees of a graph is double the number of edges, so there cannot be an odd number of odd-degree vertices.) In the third step, note that the relevant matching instance is the graph induced by W, which is the complete graph on W. Since this is not a bipartite graph (at least if $|W| \geq 4$), this is an instance of *nonbipartite* matching. We haven't covered any algorithms for this problem, but we mentioned in Lecture #6 that the ideas behind the Hungarian algorithm (Lecture #5) can, with additional ideas, be extended to also solve the nonbipartite case in polynomial time. In the fourth step, there may be edges that appear in both T and M. The graph H contains two copies of such edges, which is not a problem for us. The last two steps are the same as in the MST heuristic. Note that the graph H is indeed Eulerian — adding the matching M to T increases the degree of each vertex $v \in W$ by exactly one (and leaves other degrees unaffected), so $T + M$ has all even degrees.[4] This algorithm can be implemented in polynomial time — the overall running time is dominated by the matching computation in the third step.

Theorem 2.4 *Christofides's algorithm is a $\frac{3}{2}$-approximation algorithm for the metric TSP.*

[4]And as usual, H is connected because T is connected.

Proof: We have

$$\begin{aligned}
\text{cost of our TSP tour} &\leq \text{cost of } C \\
&= \sum_{e \in H} c_e \\
&= \underbrace{\sum_{e \in T} c_e}_{\leq OPT \text{ (Lem 2.1)}} + \underbrace{\sum_{e \in M} c_e}_{\leq OPT/2 \text{ (Lem 2.3)}} \\
&\leq \frac{3}{2} \cdot \text{cost of optimal TSP tour},
\end{aligned}$$

where the first inequality holds because the edge costs obey the triangle inequality, the second equation holds because the Euler tour C uses every edge of H exactly once, the third equation follows from the definition of H, and the final inequality follows from Lemmas 2.1 and 2.3. ■

The analysis of Christofides's algorithm in Theorem 2.4 is tight — for every constant $c < \frac{3}{2}$, there is a metric TSP instance such that the algorithm outputs a tour with cost more than c times that of an optimal tour (Exercise Set #8).

Christofides's algorithm is from 1976. Amazingly, to this day we still don't know whether or not there is an approximation algorithm for metric TSP better than Christofides's algorithm. It's possible that no such algorithm exists (assuming $P \neq NP$, since if $P = NP$ the problem can be solved optimally in polynomial time), but it is widely conjecture that $\frac{4}{3}$ (if not better) is possible. This is one of the biggest open questions in the field of approximation algorithms.

3 Asymmetric TSP

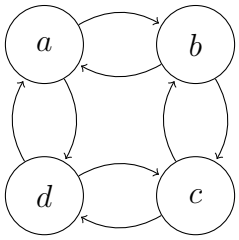

Figure 4: Example ATSP graph. Note that edges going in opposite directions need not have the same cost.

We conclude with an approximation algorithm for the *asymmetric TSP (ATSP)* problem, the directed version of TSP. That is, the input is a complete directed graph, with an edge

in each direction between each pair of vertices, and a nonnegative cost $c_e \geq 0$ for each edge (Figure 4). The edges going in opposite directions between a pair of vertices need not have the same cost.[5] The "normal" TSP is equivalent to the special case in which opposite edges (between the same pair of vertices) have the same cost. The goal is to compute the directed TSP tour — a simple directed cycle, visiting each vertex exactly once — with minimum-possible cost. Since the ATSP includes the TSP as a special case, it can only harder (and appears to be strictly harder). Thus we'll continue to assume that the edge costs obey the triangle inequality ($c_{uw} \leq c_{uv} + c_{vw}$ for every $u, v, w \in V$) — note that this assumption makes perfect sense in directed graphs as well as undirected graphs.

Our high-level strategy mirrors that in our metric TSP approximation algorithms.

1. Construct a not-too-expensive Eulerian directed graph H.

2. Shortcut H to get a directed TSP tour; by the triangle inequality, the cost of this tour is at most $\sum_{e \in H} c_e$.

Recall that a directed graph H is *Eulerian* if (i) it is strongly connected (i.e., for every v, w there is a directed path from v to w and also a directed path from w to v); and (ii) for every vertex v, the in-degree of v in H equals its out-degree in H. Every directed Eulerian graph admits a directed Euler tour — a directed closed walk that uses every (directed) edge exactly once. Assumptions (i) and (ii) are clearly necessary for a graph to have a directed Euler tour (since one enters and exists a vertex the same number of times). The proof of sufficiency is basically the same as in the undirected case (cf., Exercise Set #7).

The big question is how to implement the first step of constructing a low-cost Eulerian graph. In the metric case, we used the minimum spanning tree as a starting point. In the directed case, we'll use a different subroutine, for computing a *minimum-cost cycle cover*.

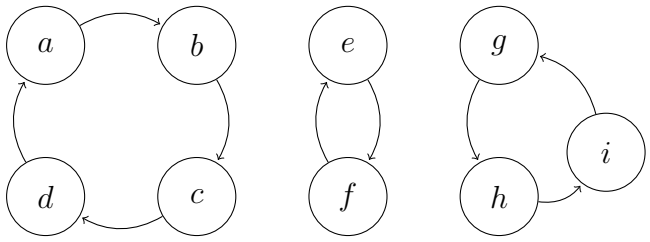

Figure 5: Example cycle cover of vertices.

A *cycle cover* of a directed graph is a collection of $C_1, \ldots, C_k$ of directed cycles, each with at least two vertices, such that each vertex $v \in V$ appears in exactly one of the cycles. (This is, the cycles partition the vertex set.) See Figure 5. Note that directed TSP tours

[5] Recalling the motivating scenario of scheduling the order of operations to minimize the overall setup time, it's easy to think of cases where the setup time between task i and task j is not the same as if the order of i and j are reversed.

are exactly the cycle covers with $k = 1$. Thus, the minimum-cost cycle cover can only be cheaper than the minimum-cost TSP tour.

Lemma 3.1 *For every instance $G = (V, E, c)$ of ATSP, the minimum-possible cost of a directed TSP tour is at least that of a minimum-cost cycle cover.*

The minimum-cost cycle cover of a directed graph can be computed in polynomial time. This is not obvious, but as a student in CS261 you're well-equipped to prove it (via a reduction to minimum-cost bipartite perfect matching, see Problem Set #4).

Approximation Algorithm for ATSP

initialize $F = \emptyset$
initialize G to the input graph
while G has at least 2 vertices **do**
    compute a minimum-cost cycle cover $C_1, \ldots, C_k$ of the current G
    add to F the edges in $C_1, \ldots, C_k$
    for $i = 1, 2, \ldots, k$ **do**
        delete from G all but one vertex from C_i
compute a directed Euler tour C of $H = (V, F)$
`// H is Eulerian, see discussion below`
shortcut repeated occurrences of vertices on C to obtain a TSP tour

For the last two steps of the algorithm to make sense, we need the following claim.

Claim: The graph $H = (V, F)$ constructed by the algorithm is Eulerian.

Proof: Note that $H = (V, F)$ is the union of all the cycle covers computed over all iterations of the while loop. We prove two invariants of (V, F) over these iterations.

First, the in-degree and out-degree of a vertex are always the same in (V, F). This is trivial at the beginning, when $F = \emptyset$. When we add in the first cycle cover to F, every vertex then has in-degree and out-degree equal to 1. The vertices that get deleted never receive any more incoming or outgoing edges, so they have the same in-degree and out-degree at the conclusion of the while loop. The undeleted vertices participate in the cycle cover computed in the second iteration; when this cycle cover is added to H, the in-degree and out-degree of each vertex in (V, F) increases by 1 (from 1 to 2). And so on. At the end, the in- and out-degree of a vertex v is exactly the number of while loop iterations in which it participated (before getting deleted).

Second, at all times, for all vertices v that have been deleted so far, there is a vertex w that has not yet been deleted such that (V, F) contains both a directed path from v to w and from w to v. That is, in (V, F), every deleted vertex can reach and be reached by some undeleted vertex.

To see why this second invariant holds, consider the first iteration. Every deleted vertex v belongs to some cycle C_i of the cycle cover, and some vertex w on C_i was left undeleted. C_i

contains a directed path from v to w and vice versa, and F contains all of C_i. By the same reasoning, every vertex v that was deleted in the second iteration has a path in (V, F) to and from some vertex w that was not deleted. A vertex u that was deleted in the first iteration has, at worst, paths in (V, F) to and from a vertex v deleted in the second iteration; stitching these paths together with the paths from v to an undeleted vertex w, we see that (V, F) contains a path from u to this undeleted vertex w, and vice versa. In the final iteration of the while loop, the cycle cover contains only one cycle C. (Otherwise, at least 2 vertices would not be deleted and the while loop would continue.) The edges of C allow every vertex remaining in the final iteration to reach every other such vertex. Since every deleted vertex can reach and be reached by the vertices remaining in the final iteration, the while loops concludes with a graph (V, F) where everybody can reach everybody (i.e., which is strongly connected). ■

The claim implies that our ATSP algorithm is well defined. We now give the easy argument bounding the cost of the tour it produces.

Lemma 3.2 *In every iteration of the algorithm's main while loop, there exists a directed TSP tour of the current graph G with cost at most OPT, the minimum cost of a TSP tour in the original input graph.*

Proof: Shortcutting the optimal TSP tour for the original graph down to one on the current graph G yields a TSP tour with cost at most OPT (using the triangle inequality). ■

By Lemmas 3.1 and 3.2:

Corollary 3.3 *In every iteration of the algorithm's main while loop, the cost of the edges added to F is at most OPT.*

Lemma 3.4 *There are at most $\log_2 n$ iterations of the algorithm's main while loop.*

Proof: Recall that every cycle in a cycle cover has, by definition, at least two vertices. The algorithm deletes all but one vertex from each cycle in each iteration, so it deletes at least one vertex for each vertex that remains. Since the number of remaining vertices drops by a factor of at least 2 in each iteration, there can only be $\log_2 n$ iterations. ■

Corollary 3.3 and Lemma 3.4 immediately give the following.

Theorem 3.5 *The ATSP algorithm above is a $\log_2 n$-approximation algorithm.*

This algorithm is from the early 1980s, and progress since then has been modest. The best-known approximation algorithm for ATSP has an approximation ratio of $O(\log n / \log \log n)$, and even this improvement is only from 2010! Another of the biggest open questions in all of approximation algorithms is: is there a constant-factor approximation algorithm for ATSP?

CS261: A Second Course in Algorithms
Lecture #17: Linear Programming and Approximation Algorithms*

Tim Roughgarden†

March 1, 2016

1 Preamble

Recall that a key ingredient in the design and analysis of approximation algorithms is getting a handle on the optimal solution, to compare it to the solution return by an algorithm. Since the optimal solution itself is often hard to understand (it's NP-hard to compute, after all), this generally entails bounds on the optimal objective function value — quantities that are "only better than optimal." If the output of an algorithm is within an α factor of this bound, then it is also within an α factor of optimal.

So where do such bounds on the optimal objective function value come from? Last week, we saw a bunch of ad hoc examples, including the maximum job size and the average load in the makespan-minimization problem, and the minimum spanning tree for the metric TSP. Today we'll see how to use linear programs and their duals to generate systematically such bounds. Linear programming and approximation algorithms are a natural marriage — for example, recall that dual feasible solutions are by definition bounds on the best-possible (primal) objective function value. We'll see that some approximation algorithms explicitly solve a linear program; some use linear programming to guide the design of an algorithm without ever actually solving a linear program to optimality; and some use linear programming duality to analyze the performance of a natural (non-LP-based) algorithm.

2 A Greedy Algorithm for Set Cover (Without Costs)

We warm up with a solution that builds on our set coverage greedy algorithm (Lecture #15) and doesn't require linear programming at all. In the *set cover* problem, the input is a list

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

$S_1, \ldots, S_m \subseteq U$ of sets, each specified as a list of elements from a ground set U. The goal is to pick as few sets as possible, subject to the constraint their union is all of U (i.e., that they form a set cover). For example, in Figure 1, the optimal solution comprises of picking the blue sets.

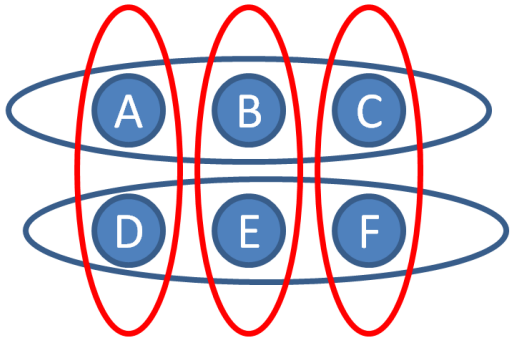

Figure 1: Example set coverage problem. The optimal solution comprises of picking the blue sets.

In the set coverage problem (Lecture #15), the input included a parameter k. The hard constraint was to pick at most k sets, and subject to this the goal was to cover as many elements as possible. Here, the constraint and the objective are reversed: the hard constraint is to cover all elements and, subject to this, to use as few sets as possible. Potential applications of the set cover problem are the same as for set coverage, and which problem is a better fit for reality depends on the context. For example, if you are choosing where to build fire stations, you can imagine that it's a hard constraint to have reasonable coverage of all of the neighborhoods of a city.

The set cover problem is NP-hard, for essentially the same reasons as the set coverage problem. There is again a tension between the size of a set and how "redundant" it is with other sets that might get chosen anyway.

Turning to approximation algorithms we note that the greedy algorithm for set coverage makes perfect sense for set cover. The only difference is in the stopping condition — rather than stopping after k iterations, the algorithm stops when it has found a set cover.

Greedy Algorithm for Set Cover (No Costs)

$\mathcal{C} = \emptyset$
while $\mathcal{C}$ not a set cover **do**
    add to $\mathcal{C}$ the set S_i which covers the largest number of new elements
    `// elements covered by previously chosen sets don't count`
return $\mathcal{C}$

The same bad examples from Lecture #15 show that the greedy algorithm is not in general optimal. In the first example of that lecture, the greedy algorithm uses 3 sets even

though 2 are enough; in the second lecture, it uses 5 sets even though 3 are enough. (And there are worse examples than these.) We next prove an approximation guarantee for the algorithm.

Theorem 2.1 *The greedy algorithm is a $\ln n$-approximation algorithm for the set cover problem, where $n = |U|$ is the size of the ground set.*

Proof: We can usefully piggyback on our analysis of the greedy algorithm for the set coverage problem (Lecture #15). Consider a set cover instance, and let OPT denote the size of the smallest set cover. The key observation is: the current solution after OPT iterations of the set cover greedy algorithm is the same as the output of the set coverage greedy algorithm with a budget of $k = OPT$. (In both cases, in every iteration, the algorithm picks the set that covers the maximum number of new elements.) Recall from Lecture #15 that the greedy algorithm is a $(1 - \frac{1}{e})$-approximation algorithm for set coverage. Since there is a collection of OPT sets covering all $|U|$ elements, the greedy algorithm, after OPT iterations, will have covered at least $(1 - \frac{1}{e})|U|$ elements, leaving at most $|U|/e$ elements uncovered. Iterating, every OPT iterations of the greedy algorithm will reduce the number of uncovered elements by a factor of e. Thus all elements are covered within $OPT \log_e n = OPT \ln n$ iterations. Thus the number of sets chosen by the greedy algorithm is at most $\ln n$ times the size of an optimal set cover, as desired. ■

3 A Greedy Algorithm for Set Cover (with Costs)

It's easy to imagine scenarios where the different sets of a set cover instance have different costs. (E.g., if sets model the skills of potential hires, different positions/seniority may command different salaries.) In the general version of the set cover problem, each set S_i also has a nonnegative cost $c_i \geq 0$. Since there were no costs in the set coverage problem, we can no longer piggyback on our analysis there — we'll need a new idea.

The greedy algorithm is easy to extend to the general case. If one set costs twice as much as another, then to be competitive, it should cover at least twice as many elements. This idea translates to the following algorithm.

Greedy Algorithm for Set Cover (With Costs)

$\mathcal{C} = \emptyset$
while $\mathcal{C}$ not a set cover **do**
    add to $\mathcal{C}$ the set S_i with the minimum *ratio*

$$r_i = \frac{c_i}{\#\text{ newly covered elements}} \tag{1}$$

return $\mathcal{C}$

Note that if all of the c_i's are identical, then we recover the previous greedy algorithm — in this case, minimizing the ratio is equivalent to maximizing the number of newly covered elements. In general, the ratio is the "average cost per-newly covered element," and it makes sense to greedily minimize this.

The best-case scenario is that the approximation guarantee for the greedy algorithm does not degrade when we allow arbitrary set costs. This is indeed the case.

Theorem 3.1 *The greedy algorithm is a $\approx \ln n$-approximation algorithm for the general set cover problem (with costs), where $n = |U|$ is the size of the ground set.*[1]

To prove Theorem 3.1, the first order of business is to understand how to make use of the greedy nature of the algorithm. The following simple lemma, reminiscent of a lemma in Lecture #15 for set coverage, addresses this point.

Lemma 3.2 *Suppose that the current greedy solution covers ℓ elements of the set S_i. Then the next set chosen by the algorithm has ratio at most*

$$\frac{c_i}{|S_i| - \ell}. \tag{2}$$

Indeed, choosing the set S_i would attain the ratio in (2); the ratio of the set chosen by the greedy algorithm can only be smaller.

For every element $e \in U$, define

$$q_e = \text{ratio of the first set chosen by the greedy algorithm that covers } e.$$

Since the greedy algorithm terminates with a set cover, every element has a well-defined q-value.[2] See Figure 2 for a concrete example.

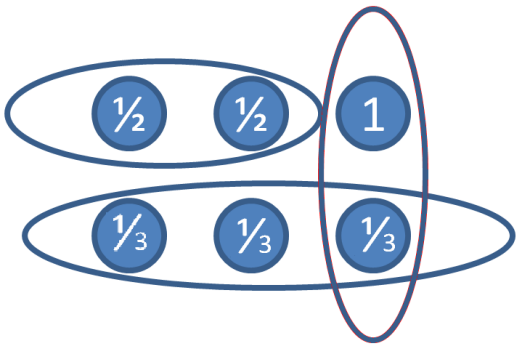

Figure 2: Example set with q-value of the elements.

[1] Inspection of the proof shows that the approximation ratio is $\approx \ln s$, where $s = \max_i |S_i|$ is the maximum size of an input set.

[2] The notation is meant to invoke the q-values in our online bipartite matching analysis (Lecture #13); as we'll see, something similar is going on here.

Corollary 3.3 *For every set S_i, the jth element e of S_i to be covered by the greedy algorithm satisfies*

$$q_e \leq \frac{c_i}{|S_i| - (j-1)}. \tag{3}$$

Corollary 3.3 follows immediately from Lemma 3.2 in the case where the elements of S_i are covered one-by-one (with $j-1$ playing the role of ℓ, for each j). In general, several elements of S_i might be covered at once. (E.g., the greedy algorithm might actually pick S_i.) But in this case the corollary is only "more true" — if j is covered as part of a batch, then the number of uncovered elements in S_i before the current selection was $j-1$ or less. For example, in Figure 2, Corollary 3.3 only asserts that the q-values of the largest set are at most $\frac{1}{3}$, $\frac{1}{2}$, and 1, when in fact all are only $\frac{1}{3}$. Similarly, for the last set chosen, Corollary 3.3 only guarantees that the q-values are at most $\frac{1}{2}$ and 1, while in fact they are $\frac{1}{3}$ and 1.

We can translate Corollary 3.3 into a bound on the sum of the q-values of the elements of a set S_i:

$$\begin{aligned}\sum_{e \in S_i} q_e &\leq \frac{c_i}{|S_i|} + \frac{c_i}{|S_i|-1} + \cdots + \frac{c_i}{2} + \frac{c_i}{1} \\ &\approx c_i \ln |S_i| \qquad (4) \\ &\leq c_i \ln n, \qquad (5)\end{aligned}$$

where $n = |U|$ is the ground set size.[3]

We also have

$$\sum_{e \in U} q_e = \text{cost of the greedy set cover.} \tag{6}$$

This identity holds inductively at all times. (If e has not been covered yet, then we define $q_e = 0$.) Initially, both sides are 0. When a new set S_i is chosen by the greedy algorithm, the right-hand side goes up by c_i. The left-hand side also increases, because all of the newly covered elements receive a q-value (equal to the ratio of the set S_i), and this increase is

$$r_i \cdot (\#\text{ of newly covered elements}) = c_i.$$

(Recall the definition (1) of the ratio.)

Proof of Theorem 3.1: Let $\{S_1^*, \ldots, S_k^*\}$ denote the sets of an optimal set cover, and OPT

[3]Our estimate $\sum_{j=1}^{|S_i|} \frac{1}{j} \approx \ln |S_i|$ in (4), which follows by approximating the sum by an integral, is actually off by an additive constant less that 1 (known as "Euler's constant"). We ignore this additive constant for simplicity.

its cost. We have

$$\begin{aligned}\text{cost of the greedy set cover} &= \sum_{e \in U} q_e \\ &\leq \sum_{i=1}^{k} \sum_{e \in S_i^*} q_e \\ &\leq \sum_{i=1}^{k} c_i \ln n \\ &= OPT \cdot \ln n,\end{aligned}$$

where the first equation is (6), the first inequality follows because $S_1^*, \ldots, S_k^*$ form a set cover (each $e \in U$ is counted at least once), and the second inequality from (5). This completes the proof. ■

Our analysis of the greedy algorithm is tight. To see this, let $U = \{1, 2, \ldots, n\}$, $S_0 = U$ with $c_0 = 1 + \epsilon$ for small ϵ, and $S_i = \{i\}$ with cost $c_i = \frac{1}{i}$ for $i = 1, 2, \ldots, n$. The optimal solution (S_0) has cost $1 + \epsilon$. The greedy algorithm chooses $S_n, S_{n-1}, \ldots, S_1$ (why?), for a total cost of $\sum_{i=1}^{n} \frac{1}{i} \approx \ln n$.

More generally, the approximation factor of $\approx \ln n$ cannot be beaten by *any* polynomial-time algorithm, no matter how clever (under standard complexity assumptions). In this sense, the greedy algorithm is optimal for the set cover problem.

4 Interpretation via Linear Programming Duality

Our proof of Theorem 3.1 is reasonably natural — using the greedy nature of the algorithm to prove the easy Lemma 3.2 and then compiling the resulting upper bounds via (5) and (6) — but it still seems a bit mysterious in hindsight. How would one come up with this type of argument for some other problem?

We next re-interpret the proof of Theorem 3.1 through the lens of linear programming duality. With this interpretation, the proof becomes much more systematic. Indeed, it follows exactly the same template that we already used in Lecture #13 to analyze the WaterLevel algorithm for online bipartite matching.

To talk about a dual, we need a primal. So consider the following linear program (P):

$$\min \sum_{i=1}^{m} c_i x_i$$

subject to

$$\begin{aligned}\sum_{i \,:\, e \in S_i} x_i &\geq 1 \qquad \text{for all } e \in U \\ x_i &\geq 0 \qquad \text{for all } S_i.\end{aligned}$$

The intended semantics is for x_i to be 1 if the set S_i is chosen in the set cover, and 0 otherwise.[4] In particular, every set cover corresponds to a 0-1 solution to (P) with the same objective function value, and conversely. For this reason, we call (P) a *linear programming relaxation* of the set cover problem — it includes all of the feasible solutions to the set cover instance (with the same cost), in additional to other (fractional) feasible solutions. Because the LP relaxation minimizes over a superset of the feasible set covers, its optimal objective function value ("fractional OPT") can only be smaller than that of a minimum-cost set cover ("OPT"):

$$\text{fractional } OPT \leq OPT.$$

We've seen a couple of examples of LP relaxations that are guaranteed to have optimal 0-1 solutions — for the minimum s-t cut problem (Lecture #8) and for bipartite matching (Lecture #9). Here, because the set cover problem is NP-hard and the linear programming relaxation can be solved in polynomial time, we don't expect the optimal LP solution to always be integral. (Whenever we get lucky and the optimal LP solution is integral, it's handing us the optimal set cover on a silver platter.) It's useful to see a concrete example of this. In Figure 3, the ground set has 3 elements and the sets are the subsets with cardinality 2. All costs are 1. The minimum cost of a set cover is clearly 2 (no set covers everything). But setting $x_i = \frac{1}{2}$ for every set yields a feasible fractional solution with the strictly smaller objective function value of $\frac{3}{2}$.

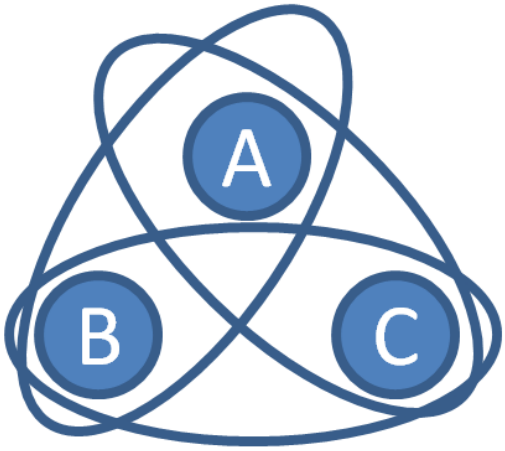

Figure 3: Example where all sets have cost 1. Optimal set cover is clearly 2, but there exists a feasible fraction with value $\frac{3}{2}$ by setting all $x_i = \frac{1}{2}$.

Deriving the dual (D) of (P) is straightforward, using the standard recipe (Lecture #8):

$$\max \sum_{e \in U} p_e$$

[4] If you're tempted to also include the constraints $x_i \leq 1$ for every S_i, note that these will hold anyways at an optimal solution.

subject to

$$\sum_{e \in S_i} p_e \leq c_i \qquad \text{for every set } S_i$$

$$p_e \geq 0 \qquad \text{for every } e \in U.$$

Lemma 4.1 *If $\{p_e\}_{e \in E}$ is a feasible solution to (D), then*

$$\sum_{e \in U} p_e \leq \textit{fractional } OPT \leq OPT.$$

The first inequality follows from weak duality — for a minimization problem, every feasible dual solution gives (by construction) a lower bound on the optimal primal objective function value — and second inequality follows because (P) is a LP relaxation of the set cover problem.

Recall the derivation from Section 3 that, for every set S_i,

$$\sum_{e \in S_i} q_e \leq c_i \ln n;$$

see (5). Looking at the constraints in the dual (D), the purpose of this derivation is now transparent:

Lemma 4.2 *The vector $\mathbf{p} := \frac{\mathbf{q}}{\ln n}$ is feasible for the dual (D).*

As such, the dual objective function value $\sum_{e \in U} p_e$ provides a lower bound on the minimum cost of a set cover (Lemma 4.1).[5] Using the identity (6) from Section 3, we get

$$\text{cost of the greedy set cover} = \ln n \cdot \sum_{e \in U} p_e \leq \ln n \cdot OPT.$$

So, while one certainly doesn't need to know linear programming to come up with the greedy set cover algorithm, or even to analyze it, linear programming duality renders the analysis transparent and reproducible for other problems. We next examine a couple of algorithms whose design is explicitly guided by linear programming.

5 A Linear Programming Rounding Algorithm for Vertex Cover

Recall from Problem Set #2 the vertex cover problem: the input is an undirected graph $G = (V, E)$ with a nonnegative cost c_v for each vertex $v \in V$, and the goal is to compute a minimum-cost subset $S \subseteq V$ that contains at least one endpoint of every edge. On

[5] This is entirely analogous to what happened in Lecture #13, for maximum bipartite matching: we defined a vector $\mathbf{q}$ with sum equal to the size of the computed matching, and we scaled up $\mathbf{q}$ to get a feasible dual solution and hence an upper bound on the maximum-possible size of a matching.

Problem Set #2 you saw that, in bipartite graphs, this problem reduces to a max-flow/min-cut computation. In general graphs, the problem is NP-hard.

The vertex cover problem can be regarded as a special case of the set cover problem. The elements needing to be covered are the edges. There is one set per vertex v, consisting of the edges incident to v (with cost c_v). Thus, we're hoping for an approximation guarantee better than what we've already obtained for the general set cover problem. The first question to ask is: does the greedy algorithm already have a better approximation ratio when we restrict attention to the special case of vertex cover instances? The answer is no (Exercise Set #9), so to do better we'll need a different algorithm.

This section analyzes an algorithm that explicitly solves a linear programming relaxation of the vertex cover problem (as opposed to using it only for the analysis). The LP relaxation (P) is the same one as in Section 4, specialized to the vertex cover problem:

$$\min \sum_{v \in V} c_v x_v$$

subject to

$$\begin{aligned} x_v + x_w &\geq 1 \qquad \text{for all } e = (v, w) \in E \\ x_v &\geq 0 \qquad \text{for all } v \in V. \end{aligned}$$

There is a one-to-one and cost-preserving correspondence between 0-1 feasible solutions to this linear program and vertex covers. (We won't care about the dual of this LP relaxation until the next section.)

Again, because the vertex cover problem is NP-hard, we don't expect the LP relaxation to always solve to integers. We can reinterpret the example from Section 4 (Figure 3) as a vertex cover instance — the graph G is a triangle (all unit vertex costs), the smallest vertex cover has size 2, but setting $x_v = \frac{1}{2}$ for all three vertices yields a feasible fractional solution with objective function value $\frac{3}{2}$.

LP Rounding Algorithm for Vertex Cover

compute an optimal solution $\mathbf{x}^*$ to the LP relaxation (P)
return $S = \{v \in V : x_v^* \geq \frac{1}{2}\}$

The first step of our new approximation algorithm computes an optimal (fractional) solution to the LP relaxation (P). The second step transforms this fractional feasible solution into an integral feasible solution (i.e., a vertex cover). In general, such a procedure is called a *rounding algorithm*. The goal is to round to an integral solution without affecting the objective function value too much.[6] The simplest approach to LP rounding, and a

[6]This is analogous to our metric TSP algorithms, where we started with an infeasible solution that was only better than optimal (the MST) and then transformed it into a feasible solution (i.e., a TSP tour) with suffering too much extra cost.

common heuristic in practice, is to round fractional values to the nearest integer (subject to feasibility). The vertex cover problem is a happy case where this heuristic gives a good worst-case approximation guarantee.

Lemma 5.1 *The LP rounding algorithm above outputs a feasible vertex cover S.*

Proof: Since the solution $\mathbf{x}^*$ is feasible for (P), $x_v^* + x_w^* \geq 1$ for every $(v, w) \in E$. Hence, for every $(v, w) \in E$, at least one of x_v^*, x_w^* is at least $\frac{1}{2}$. Hence at least one endpoint of every edge is included in the final output S. ■

The approximation guarantee follows from the fact that the algorithm pays at most twice what the optimal LP solution $\mathbf{x}^*$ pays.

Theorem 5.2 *The LP rounding algorithm above is a 2-approximation algorithm.*

Proof: We have

$$\begin{aligned} \underbrace{\sum_{v \in S} c_v}_{\text{cost of alg's soln}} &\leq \sum_{v \in V} c_v (2x_v^*) \\ &= 2 \cdot \text{fractional } OPT \\ &\leq 2 \cdot OPT, \end{aligned}$$

where the first inequality holds because $v \in S$ only if $x_v^* \geq \frac{1}{2}$, the equation holds because $\mathbf{x}^*$ is an optimal solution to (P), and the second inequality follows because (P) is a LP relaxation of the vertex cover problem. ■

6 A Primal-Dual Algorithm for Vertex Cover

Can we do better than Theorem 5.2? In terms of worst-case approximation ratio, the answer seems to be no.[7] But we can still ask if we can improve the running time. For example, can we get a 2-approximation algorithm without explicitly solving the linear programming relaxation? (E.g., for set cover, we used linear programs only in the analysis, not in the algorithm itself.)

Our plan is to use the LP relaxation (P) and its dual (below) to guide the decisions made by our algorithm, without ever solving either linear program explicitly (or exactly). The dual linear program (D) is again just a specialization of that for the set cover problem:

$$\max \sum_{e \in E} p_e$$

[7] Assuming the "Unique Games Conjecture," a significant strengthening of the $P \neq NP$ conjecture, there is no $(2 - \epsilon)$-approximation algorithm for vertex cover, for any constant $\epsilon > 0$.

subject to

$$\sum_{e \in \delta(v)} p_e \leq c_v \qquad \text{for every } v \in V$$

$$p_e \geq 0 \qquad \text{for every } e \in E.$$

We consider the following algorithm, which maintains a dual feasible solution and iteratively works toward a vertex cover.

Primal-Dual Algorithm for Vertex Cover

initialize $p_e = 0$ for every edge $e \in E$
initialize $S = \emptyset$
while S is not a vertex cover **do**
    pick an edge $e = (v, w)$ with $v, w \notin S$
    increase p_e until the dual constraint corresponding to v or w goes tight
    add the vertex corresponding to the tight dual constraint to S

In the while loop, such an edge $(v, w) \in E$ must exist (otherwise S would be a vertex cover). By a dual constraint "going tight," we mean that it holds with equality. It is easy to implement this algorithm, using a single pass over the edges, in linear time. This algorithm is very natural when you're staring at the primal-dual pair of linear programs. Without knowing these linear programs, it's not clear how one would come up with it.

For the analysis, we note three invariants of the algorithm.

(P1) $\mathbf{p}$ is feasible for (D). This is clearly true at the beginning when $p_e = 0$ for every $e \in E$ (vertex costs are nonnegative), and the algorithm (by definition) never violates a dual constraint in subsequent iterations.

(P2) If $v \in S$, then $\sum_{e \in \delta(v)} p_e = c_v$. This is obviously true initially, and we only add a vertex to S when this condition holds for it.

(P3) If $p_e > 0$ for $e = (v, w) \in E$, then $|S \cap \{v, w\}| \leq 2$. This is trivially true (whether or not $p_e > 0$).

Furthermore, by the stopping condition, at termination we have:

(P4) S is a vertex cover.

That is, the algorithm maintains dual feasibility and works toward primal feasibility. The second and third invariants should be interpreted as an approximate version of the complementary slackness conditions.[8] The second invariant is exactly the first set of complemen-

[8]Recall the complementary slackness conditions from Lecture #9: (i) whenever a primal variable is nonzero, the corresponding dual constraint is tight; (ii) whenever a dual variable is nonzero, the corresponding primal constraint is tight. Recall that the complementary slackness conditions are precisely the conditions under which the derivation of weak duality holds with equality. Recall that a primal-dual pair of feasible solutions are both optimal if and only if the complementary slackness conditions hold.

tary slackness conditions — it says that a primal variable is positive (i.e., $v \in S$) only if the corresponding dual constraint is tight. The second set of exact complementary slackness conditions would assert that whenever $p_e > 0$ for $e = (v, w) \in E$, the corresponding primal constraint is tight (i.e., exactly one of v, w is in S). These conditions will not in general hold for the algorithm above (if they did, then the algorithm would always solve the problem exactly). They do hold approximately, in the sense that tightness is violated only by a factor of 2. This is exactly where the approximation factor of the algorithm comes from.

Since the algorithm maintains dual feasibility and approximate complementary slackness and works toward primal feasibility, it is a *primal-dual* algorithm, in exactly the same sense as the Hungarian algorithm for minimum-cost perfect bipartite matching (Lecture #9). The only difference is that the Hungarian algorithm maintains exact complementary slackness and hence terminates with an optimal solution, while our primal-dual vertex cover algorithm only maintains approximate complementary slackness, and for this reason terminates with an approximately optimal solution.

Theorem 6.1 *The primal-dual algorithm above is a 2-approximation algorithm for the vertex cover problem.*

Proof: The derivation is familiar from when we derived weak duality (Lecture #8). Letting S denote the vertex cover returned by the primal-dual algorithm, OPT the minimum cost of a vertex cover, and "fractional OPT" the optimal objective function value of the LP relaxation, we have

$$\begin{aligned}
\sum_{v \in S} c_v &= \sum_{v \in S} \sum_{e \in \delta(v)} p_e \\
&= \sum_{e=(v,w) \in E} p_e \cdot |S \cap \{v, w\}| \\
&\leq 2 \sum_{e \in E} p_e \\
&\leq 2 \cdot \text{fractional } OPT \\
&\leq 2 \cdot OPT.
\end{aligned}$$

The first equation is the first (exact) set of complementary slackness conditions (P2), the second equation is just a reversal of the order of summation, the first inequality follows from the approximate version of the second set of complementary slackness conditions (P3), the second inequality follows from dual feasibility (P1) and weak duality, and the final inequality follows because (P) is an LP relaxation of the vertex cover problem. This completes the proof. ■

CS261: A Second Course in Algorithms
Lecture #18: Five Essential Tools for the Analysis of Randomized Algorithms*

Tim Roughgarden†

March 3, 2016

1 Preamble

In CS109 and CS161, you learned some tricks of the trade in the analysis of randomized algorithms, with applications to the analysis of QuickSort and hashing. There's also CS265, where you'll learn more than you ever wanted to know about randomized algorithms (but a great class, you should take it). In CS261, we build a bridge between what's covered in CS161 and CS265. Specifically, this lecture covers five essential tools for the analysis of randomized algorithms. Some you've probably seen before (like linearity of expectation and the union bound) while others may be new (like Chernoff bounds). You will need these tools in most 200- and 300-level theory courses that you may take in the future, and in other courses (like in machine learning) as well. We'll point out some applications in approximation algorithms, but keep in mind that these tools are used constantly across all of theoretical computer science.

Recall the standard probability setup. There is a *state space* Ω; for our purposes, Ω is always finite, for example corresponding to the coin flip outcomes of a randomized algorithm. A *random variable* is a real-valued function $X : \Omega \to \mathbb{R}$ defined on Ω. For example, for a fixed instance of a problem, we might be interested in the running time or solution quality produced by a randomized algorithm (as a function of the algorithm's coin flips). The *expectation* of a random variable is just its average value, with the averaging weights given by a specified probability distribution on Ω:

$$\mathbf{E}[X] = \sum_{\omega \in \Omega} \mathbf{Pr}[\omega] \cdot X(\omega).$$

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

An *event* is a subset of Ω. The *indicator random variable* for an event $E \subseteq \Omega$ takes on the value 1 for $\omega \in E$ and 0 for $\omega \notin E$. Two events E_1, E_2 are *independent* if their probabilities factor: $\mathbf{Pr}[E_1 \wedge E_2] = \mathbf{Pr}[E_1] \cdot \mathbf{Pr}[E_2]$. Two random variables X_1, X_2 are independent if, for every x_1 and x_2, the events $\{\omega : X_1(\omega) = x_1\}$ and $\{\omega : X_2(\omega) = x_2\}$ are independent. In this case, expectations factor: $\mathbf{E}[XY] = \mathbf{E}[X] \cdot \mathbf{E}[Y]$. Independence for sets of 3 or more events or random variables is defined analogously (for every subset, probabilities should factor). Probabilities and expectations generally don't factor for non-independent random variables, for example if E_1, E_2 are complementary events (so $\mathbf{Pr}[E_1 \wedge E_2] = 0$).

2 Linearity of Expectation and MAX 3SAT

2.1 Linearity of Expectation

The first of our five essential tools is *linearity of expectation*. Like most of these tools, it somehow manages to be both near-trivial and insanely useful. You've surely seen it before.[1] To remind you, suppose $X_1, \ldots, X_n$ are random variables defined on a common state space Ω. Crucially, *the X_i's need not be independent*. Linearity of expectation says that we can freely exchange expectations with summations:

$$\mathbf{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbf{E}[X_i]\,.$$

The proof is trivial — just expand the expectations as sums over Ω, and reverse the order of summation.

The analogous statement for, say, products of random variables is not generally true (when the X_i's are not independent). Again, just think of two indicator random variables for complementary events.

As an algorithm designer, why should you care about linearity of expectation? A typical use case works as follows. Suppose there is some complex random variable X that we care about — like the number of comparisons used by QuickSort, or the objective function value of the solution returned by some randomized algorithm. In many cases, it is possible to express the complex random variable X as the sum $\sum_{i=1}^{n} X_i$ of much simpler random variables $X_1, \ldots, X_n$, for example indicator random variables. One can then analyze the expectation of the simple random variables directly, and exploit linearity of expectation to deduce the expected value of the complex random variable of interest. You should have seen this recipe in action already in CS109 and/or CS161, for example when analyzing QuickSort or hash tables. Remarkably, linearity of expectation is already enough to derive interesting results in approximation algorithms.

[1] When I teach CS161, out of all the twenty lectures, exactly one equation gets a box drawn around it for emphasis — linearity of expectation.

2.2 A $\frac{7}{8}$-Approximation Algorithm for MAX 3SAT

An input of *MAX 3SAT* is just like an input of 3SAT — there are n Boolean variables $x_1, \ldots, x_n$ and m clauses. Each clause is the disjunction ("or") of 3 literals (where a literal is a variable or its negation). For example, a clause might have the form $x_3 \vee \neg x_6 \vee \neg x_{10}$. For simplicity, assume that the 3 literals in each clause correspond to distinct variables. The goal is to output a truth assignment (an assignment of each x_i to { true, false }) that satisfies the maximum-possible number of clauses. Since 3SAT is the special case of checking whether or not the optimal objective function value equals m, MAX 3SAT is an NP-hard problem.

A very simple algorithm has a pretty good approximation ratio.

Theorem 2.1 *The expected number of clauses satisfied by a random truth assignment, chosen uniformly at random from all 2^n truth assignments, is $\frac{7}{8}m$.*

Since the optimal solution can't possibly satisfy more than m clauses, we conclude that the algorithm that chooses a random assignment is a $\frac{7}{8}$-approximation (in expectation).

Proof of Theorem 2.1: Identify the state space Ω with all 2^n possible truth assignments (with the uniform distribution). For each clause j, let X_j denote the indicator random variable for the event that clause j is satisfied. Observe that the random variable X that we really care about, the number of satisfied clauses, is the sum $\sum_{j=1}^{n} X_j$ of these simple random variables. We now follow the recipe above, analyzing the simple random variables directly and using linearity of expectation to analyze X. As always with an indicator random variable, the expectation is just the probability of the corresponding event:

$$\mathbf{E}[X_j] = 1 \cdot \mathbf{Pr}[X_j = 1] + 0 \cdot \mathbf{Pr}[X_j = 0] = \mathbf{Pr}[\text{clause } j \text{ satisfied}]\,.$$

The key observation is that clause j is satisfied by a random assignment with probability exactly $\frac{7}{8}$. For example, suppose the clause is $x_1 \vee x_2 \vee x_3$. Then a random truth assignment satisfies the clause unless we are unlucky enough to set each of x_1, x_2, x_3 to false — for all of the other 7 combinations, at least one variable is true and hence the clause is satisfied. But there's nothing special about this clause — for any clause with 3 literals corresponding to distinct variables, only 1 of the 8 possible assignments to these three variables fails to satisfy the clause.

Putting the pieces together and using linearity of expectation, we have

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{j=1}^{m} X_j\right] = \sum_{j=1}^{m} \mathbf{E}[X_j] = \sum_{j=1}^{m} \tfrac{7}{8} = \tfrac{7}{8}m,$$

as claimed. ■

If a random assignment satisfies $\frac{7}{8}m$ clauses on average, then certainly some truth assignment does as well as this average.[2]

[2]It is not hard to derandomize the randomized algorithm to compute such a truth assignment deterministically in polynomial time, but this is outside the scope of this lecture.

Corollary 2.2 *For every 3SAT formula, there exists a truth assignment satisfying at least 87.5% of the clauses.*

Corollary 2.2 is counterintuitive to many people the first time they see it, but it is a near-trivial consequence of linearity of expectation (which itself is near-trivial!).

Remarkably, and perhaps depressingly, there is no better approximation algorithm: assuming $P \neq NP$, there is no $(\frac{7}{8} + \epsilon)$-approximation algorithm for MAX 3SAT, for any constant $\epsilon > 0$. This is one of the major results in "hardness of approximation."

3 Tail Inequalities

If you only care about the expected value of a random variable, then linearity of expectation is often the only tool you need. But in many cases one wants to prove that an algorithm is good not only on average, but is also good almost all the time ("with high probability"). Such high-probability statements require different tools.

The point of a *tail inequality* is to prove that a random variable is very likely to be close to its expected value — that the random variable "concentrates." In the world of tail inequalities, there is always a trade-off between how much you assume about your random variable, and the degree of concentration that you can prove. This section looks at the three most commonly used points on this trade-off curve. We use hashing as a simple running example to illustrate these three inequalities; the next section connects these ideas back to approximation algorithms.

3.1 Hashing

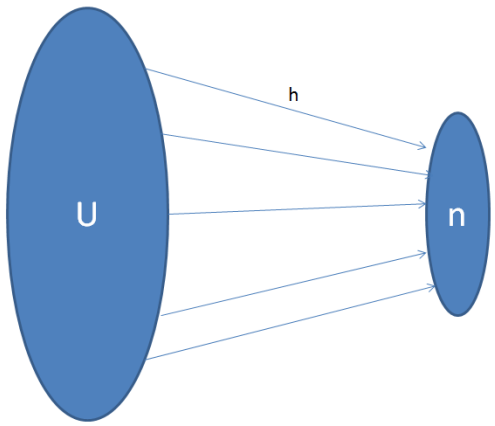

Figure 1: a hash function h that maps a large universe U to a relatively smaller number of buckets n.

Throughout this section, we consider a family $\mathcal{H}$ of hash functions, with each $h \in \mathcal{H}$ mapping a large universe U to a relatively small number of "buckets" $\{1, 2, \ldots, n\}$ (Figure 1). We'll be thinking about the following experiment, which should be familiar from CS161: an adversary picks an arbitrary data set $S \subseteq U$, then we pick a hash function $h \in \mathcal{H}$ uniformly at random and use it to hash all of the elements of S. We'd like these objects to be distributed evenly across the buckets, and the maximum load of a bucket (i.e., the number of items hashing to it) is a natural measure of distance from this ideal case. For example, in a hash table with chaining, the maximum load of a bucket governs the worst-case search time, a highly relevant statistic.

3.2 Markov's Inequality

For now, all we assume about $\mathcal{H}$ is that each object is equally likely to map to each bucket (though not necessarily independently).

(P1) For every $x \in U$ and $i \in \{1, 2, \ldots, n\}$, $\mathbf{Pr}_{h \in \mathcal{H}}[h(x) = i] = \frac{1}{n}$.

This property is already enough to analyze the expected load of a bucket. For simplicity, suppose that the size $|S|$ of the data set being hashed equals the number of buckets n. Then, for any bucket i, by linearity of expectation (applied to indicator random variables for elements getting mapped to i), its expected load is

$$\sum_{x \in S} \underbrace{\mathbf{Pr}[h(x) = i]}_{=1/n \text{ by (P1)}} = \frac{|S|}{n} = 1. \tag{1}$$

This is good — the expectations seem to indicate that things are balanced on average. But can we prove a concentration result, stating that loads are close to these expectations?

The following tail inequality gives a weak bound but applies under minimal assumptions; it is our second (of 5) essential tools for the analysis of randomized algorithms.

Theorem 3.1 (Markov's Inequality) *If X is a non-negative random variable with finite expectation, then for every constant $c \geq 1$,*

$$\mathbf{Pr}[X \geq c \cdot \mathbf{E}[X]] \leq \frac{1}{c}.$$

For example, such a random variable is at least 10 times its expectation at most 10% of the time, and is at least 100 times its expectation at most 1% of the time. In general, Markov's inequality is useful when a constant probability guarantee is good enough. The proof of Markov's inequality is easy, and we leave it to Exercise Set #9.[3]

[3]Both hypotheses are necessary. For example, random variables that are equally likely to be M or $-M$ exhibit no concentration whatsoever as $M \to \infty$.

We now apply Markov's inequality to the random variable equal to the load of our favorite bucket i. We can choose any $c \geq 1$ we want in Theorem 3.1. For example, choosing $c = n$ and recalling that the relevant expectation is 1 (assuming $|S| = n$), we obtain

$$\mathbf{Pr}[\text{load of i } \geq n] \leq \frac{1}{n}.$$

The good news is that $\frac{1}{n}$ is not a very big number when n is large. But let's look at the event we're talking about: the load of i being at least n means that *every single element* of S hashes to i. And this sounds crazy, like it should happen much less often than $1/n$ of the time. (If you hash 100 things into a hash table with 100 buckets, would you really expect everything to hash to the same bucket 1% of the time?)

If we're only assuming the property (P1), however, it's impossible to prove a better bound. To see this, consider the set $\mathcal{H} = \{h(x) = i \,:\, i = 1, 2, \ldots, n\}$ of constant hash functions, each of which maps all items to the same bucket. Observe that $\mathcal{H}$ satisfies property (P1). But the probability that all items hash to the bucket i is indeed $\frac{1}{n}$.

3.3 Chebyshev's Inequality

A totally reasonable objection is that the example above is a stupid family of hash function that no one would ever use. So what about a good family of hash functions, like those you studied in CS161? Specifically, we now assume:

(P2) for every pair $x, y \in U$ of distinct elements, and every $i, j \in \{1, 2, \ldots, n\}$,

$$\mathbf{Pr}_{h \in \mathcal{H}}[h(x) = i \text{ and } h(y) = j] = \frac{1}{n^2}.$$

That is, when looking at only two elements, the joint distribution of their buckets is as if the function h is a totally random function. (Property (P1) asserts an analogous statement when looking at only a single element.) A family of hash functions satisfying (P2) is called a *pairwise* or *2-wise* independent family. This is almost the same as (and for practical purposes equivalent to) the notion of "universal hashing" that you saw in CS161. The family of constant hash functions (above) clearly fails to satisfy property (P2).

So how do we use this stronger assumption to prove sharper concentration bounds? Recall that the *variance* $\text{Var}[X]$ of a random variable is its expected squared deviation from its mean $\mathbf{E}[(X - \mathbf{E}[X])^2]$, and that the *standard deviation* is the square root of the variance. Assumption (P2) buys us control over the variance of the load of a bucket. *Chebyshev's inequality*, the third of our five essential tools, is the inequality you want to use when the best thing you've got going for you is a good bound on the variance of a random variable.

Theorem 3.2 (Chebyshev's Inequality) *If X is a random variable with finite expectation and variance, then for every constant $t \geq 1$,*

$$\mathbf{Pr}[|X - \mathbf{E}[X]| > t \cdot \mathit{StdDev}[X]] \leq \frac{1}{t^2}.$$

For example, the probability that a random variable differs from its expectation by at least two standard deviations is at most 25%, and the probability that it differs by at least 10 standard deviations is at most 1%. Chebyshev's inequality follows easily from Markov's inequality; see Exercise Set #9.

Now let's go back to the load of our favorite bucket i, where a data set $S \subseteq U$ with size $|S| = n$ is hashed using a hash function h chosen uniformly at random from $\mathcal{H}$. Call this random variable X. We can write

$$X = \sum_{y \in S} X_y,$$

where X_y is the indicator random variable for whether or not $h(y) = i$. We noted earlier that, by (P1), $\mathbf{E}[X] = \sum_{y \in S} \frac{1}{n} = 1$.

Now consider the variance of X. We claim that

$$\text{Var}[X] = \sum_{y \in S} \text{Var}[X_y], \tag{2}$$

analogous to linearity of expectation. Note that this statement is *not* true in general — e.g., if X_1 and X_2 are indicator random variables of complementary events, then $X_1 + X_2$ is always equal to 1 and hence has variance 0. In CS109 you saw a proof that for independent random variables, variances add as in (2). If you go back and look at this derivation — seriously, go look at it — you'll see that the variance of a sum equals the sum of the variances of the summands, plus correction terms that involve the covariances of pairs of summands. The covariance of independent random variables is zero. Here, we are only dealing with pairwise independent random variables (by assumption (P2)), but still, this implies that the covariance of any two summands is 0. We conclude that (2) holds not only for sums of independent random variables, but also of pairwise independent random variables.

Each indicator random variable X_y is a Bernoulli variable with parameter $\frac{1}{n}$, and so $\text{Var}[X_y] = \frac{1}{n}(1 - \frac{1}{n}) \leq 1$. Using (2), we have $\text{Var}[X] = \sum_{y \in S} \text{Var}[X_y] \leq n \cdot \frac{1}{n} = 1$. (By contrast, when $\mathcal{H}$ is the set of constant hash functions, $\text{Var}[X] \approx n$.)

Applying Chebyshev's inequality with $t = n$ (and ignoring "+1" terms for simplicity), we obtain

$$\mathbf{Pr}_{h \in \mathcal{H}}[X \geq n] \leq \frac{1}{n^2}.$$

This is a better bound than what we got from Markov's inequality, but it still doesn't seem that small — when hashing 10 elements into 10 buckets, do you really expect to see all of them in a single bucket 1% of the time? But again, without assuming more than property (P2), we can't do better — there exist families of pairwise independent hash functions such that all elements hash to the same bucket with probability $\frac{1}{n^2}$; showing this is a nice puzzle.

3.4 Chernoff Bounds

In this section we assume that:

(P3) all $h(x)$'s are uniformly and independently distributed in $\{1, 2, \ldots, n\}$. Equivalently, h is completely random function.

How can we use this strong assumption to prove sharper concentration bounds?

The fourth of our five essential tools for analyzing randomized algorithms is the *Chernoff bounds*. They are the centerpiece of this lecture, and are used all the time in the analysis of algorithms (and also complexity theory, machine learning, etc.).

The point of the Chernoff bounds is to prove sharp concentration for sums of independent and bounded random variables.

Theorem 3.3 (Chernoff Bounds) *Let $X_1, \ldots, X_n$ be random variables, defined on the same state space and taking values in $[0,1]$, and set $X = \sum_{j=1}^{n} X_j$. Then:*

(i) for every $\delta > 0$,

$$\mathbf{Pr}[X > (1+\delta)\mathbf{E}[X]] < \left(\frac{e}{1+\delta}\right)^{(1+\delta)\mathbf{E}[X]}.$$

(ii) for every $\delta \in (0,1)$,

$$\mathbf{Pr}[X < (1-\delta)\mathbf{E}[X]] < e^{-\delta^2\mathbf{E}[X]/2}.$$

The key thing to notice in Theorem 3.3 is that the deviation probability decays exponentially in both the factor of the deviation $(1+\delta)$ and the expectation of the random variable ($\mathbf{E}[X]$). So if either of these quantities is even modestly big, then the deviation probability is going to be very small.[4]

We could prove Theorem 3.3 in 30 minutes or less, but the right place to spend time on the proof is a randomized algorithms class (like CS265). So we'll just use the Chernoff bounds as a "black box" — this is how almost everybody thinks about them, anyways. It's notable that, of our five essential tools for the analysis of randomized algorithms, only the Chernoff bounds require a non-trivial proof. We'll only use part (i) in this lecture, but (ii) is also useful in many situations. An analog of Theorem 3.3 for random variables that are nonnegative and bounded (not necessarily in $[0,1]$) follows from a simple scaling argument. The independence assumption can be relaxed, for example to negatively correlated random variables, although the proof then requires a bit more work.

Now let's apply the Chernoff bounds to analyze the number of items hashing to our favorite bucket i, under the assumption (P3) that h is a uniformly random function. Again using X_y to denote the indicator random variable for the event that $h(y) = i$, we see that $X = \sum_{y \in S} X_y$ is now the sum of independent 0-1 random variables, and hence is right in the wheelhouse of the Chernoff bounds. For example, setting $1 + \delta = \ln n$ and recalling that $\mathbf{E}[X] = 1$, Theorem 3.3 implies that

$$\mathbf{Pr}[X > \ln n] < \left(\frac{e}{\ln n}\right)^{\ln n}. \tag{3}$$

To interpret this bound, note that $(\frac{1}{e})^{\ln n} = \frac{1}{n}$. More generally, a constant less than one raised to a logarithmic power yields an inverse polynomial. Now $\frac{e}{\ln n}$ is smaller than any

[4] For the first bound (i), it is common to state the tighter probability upper bound of $[e^{\delta}/(1+\delta)^{(1+\delta)}]^{\mathbf{E}[X]}$, which is useful in applications where δ is small. The simpler bound here suffices for all of our applications.

constant as n grows large, and hence the probability bound in (3) is smaller than any inverse polynomial. Notice how much better this is than what we could prove using Markov's or Chebyshev's inequality — we're looking at a much smaller deviation ($\ln n$ instead of n) yet obtaining a much smaller probability bound (smaller than any inverse polynomial).

Theorem 3.3 even implies that

$$\mathbf{Pr}\left[X > \frac{3\ln n}{\ln\ln n}\right] \leq \frac{1}{n^2}, \tag{4}$$

as you should verify. Why $\ln n/\ln\ln n$? Because this is roughly the solution to the equation $x^x = n$ (this is relevant in Theorem 3.3 because of the $(1+\delta)^{-(1+\delta)}$ term). Again, this is a huge improvement over what we obtained using Markov's and Chebyshev's inequalities. For a more direct comparison, note that Chernoff bounds imply that the probability $\mathbf{Pr}[X \geq n]$ is at most an inverse exponential function of n (as opposed to an inverse polynomial function).

3.5 The Union Bound

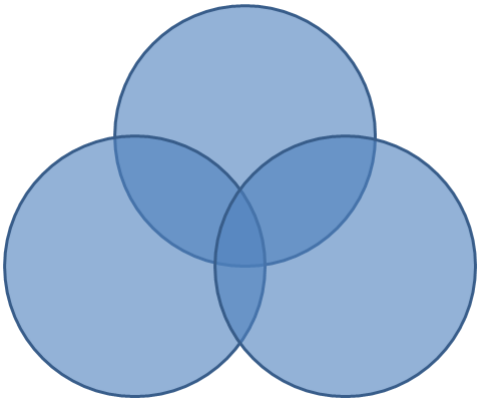

Figure 2: Area of union is bounded by sum of areas of the circles.

Our fifth essential analysis tool is the *union bound*, which is not a tail inequality but is often used in conjunction with tail inequalities. The union bound just says that for events $E_1, \ldots, E_k$,

$$\mathbf{Pr}[\text{at least once of } E_i \text{ occurs}] \leq \sum_{i=1}^{k} \mathbf{Pr}[E_i]\,.$$

Importantly, the events are completely arbitrary, and do not need to be independent. The proof is a one-liner. In terms of Figure 2, the union bound just says that the area (i.e., probability mass) in the union is bounded above by the sum of the areas of the circles. The bound is tight if the events are disjoint; otherwise the right-hand side is larger, due to double-counting. (It's like inclusion-exclusion, but without any of the correction terms.) In

applications, the events $E_1, \ldots, E_k$ are often "bad events" that we're hoping don't happen; the union bound says that as long as each event occurs with low probability and there aren't too many events, then with high probability none of them occur.

Returning to our running hashing example, let E_i denote the event that bucket i receives a load larger than $3 \ln n / \ln \ln n$. Using (4) and the union bound, we conclude that with probability at least $1 - \frac{1}{n}$, *none* of the buckets receive a load larger than $3 \ln n / \ln \ln n$. That is, the maximum load is $O(\log n / \log \log n)$ with high probability.[5]

3.6 Chernoff Bounds: The Large Expectation Regime

We previously noted that the Chernoff bounds yield very good probability bounds once the deviation $(1+\delta)$ or the expectation ($\mathbf{E}[X]$) becomes large. In our hashing application above, we were in the former regime. To illustrate the latter regime, suppose that we hash a data set $S \subseteq U$ with $|S| = n \ln n$ (instead of $\ln n$). Now, the expected load of every bucket is $\ln n$. Applying Theorem 3.3 with $1 + \delta = 4$, we get that, for each bucket i,

$$\mathbf{Pr}[\text{load on } i \text{ is } > 4 \ln n] \leq \left(\frac{e}{4}\right)^{4 \ln n} \leq \frac{1}{n^2}.$$

Using the union bound as before, we conclude that with high probability, no bucket receives a load more than a small constant factor times its expectation.

Summarizing, when loads are light there can be non-trivial deviations from expected loads (though still only logarithmic). Once loads are even modestly larger, however, the buckets are quite evenly balanced with high probability. This is a useful lesson to remember, for example in load-balancing applications (in data centers, etc.).

4 Randomized Rounding

We now return to the design and analysis of approximation algorithms, and give a classic application of the Chernoff bounds to the problem of low-congestion routing.

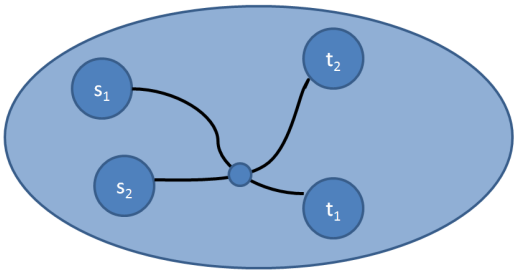

Figure 3: Example of edge-disjoint path problem. Note that vertices can be shared, as shown in this example.

[5]There is also a matching lower bound (up to constant factors).

If the *edge-disjoint paths* problems, the input is a graph $G = (V, E)$ (directed or undirected) and source-sink pairs $(s_1, t_1), \ldots, (s_k, t_k)$. The goal is to determine whether or not there is an s_i-t_i path P_i for each i such that no edge appears in more than one of the P_i's. See Figure 3. The problem is NP-hard (for directed graphs, even when $k = 2$).

Recall from last lecture the linear programming rounding approach to approximation algorithms:

1. Solve an LP relaxation of the problem. (For an NP-hard problem, we expect the optimal solution to be fractional, and hence not immediately meaningful.)

2. "Round" the resulting fractional solution to a feasible (integral) solution, hopefully without degrading the objective function value by too much.

Last lecture applied LP rounding to the vertex cover problem. For the edge-disjoint paths problem, we'll use *randomized* LP rounding. The idea is to interpret the fractional values in an LP solution as specifying a probability distribution, and then to round variables to integers randomly according to this distribution.

The first step of the algorithm is to solve the natural linear programming relaxation of the edge-disjoint paths problem. This is just a multicommodity flow problem (as in Exercise Set #5 and Problem Set #3). In this relaxation the question is whether or not it is possible to send simultaneously one unit of (fractional) flow from each source s_i to the corresponding sink t_i, where every edge has a capacity of 1. 0-1 solutions to this multicommodity flow problem correspond to edge-disjoint paths. As we've seen, this LP relaxation can be solved in polynomial time. If this LP relaxation is infeasible, then we can conclude that the original edge-disjoint paths problem is infeasible as well.

Assume now that the LP relaxation is feasible. The second step rounds each s_i-t_i pair independently. Consider a path decomposition (Problem Set #1) of the flow being pushed from s_i to t_i. This gives a collection of paths, together with some amount of flow on each path. Since exactly one unit of flow is sent, we can interpret this path decomposition as a probability distribution over s_i-t_i paths. The algorithm then just selects an s_i-t_i path randomly according to this probability distribution.

The rounding step yields paths $P_1, \ldots, P_k$. In general, they will not be disjoint (this would solve an NP-hard problem), and the goal is to prove that they are approximately disjoint in some sense. The following result is the original and still canonical application of randomized rounding.

Theorem 4.1 *Assume that the LP relaxation is feasible. Then with high probability, the randomized rounding algorithm above outputs a collection of paths such that no edge is used by more than*

$$\frac{3 \ln m}{\ln \ln m}$$

of the paths, where m is the number of edges.

The outline of the proof is:

1. Fix an edge e. The expected number of paths that include e is at most 1. (By linearity of expectation, it is precisely the amount of flow sent on e by the multicommodity flow relaxation, which is at most 1 since all edges were given unit capacity.)

2. Like in the hashing analysis in Section 3.6,

$$\mathbf{Pr}\left[\#\text{ paths on } e > \frac{3\ln m}{\ln\ln m}\right] \leq \frac{1}{m^2},$$

   where m is the number of edges. (Edges are playing the role of buckets, and s_i-t_i pairs as items.)

3. Taking a union bound over the m edges, we conclude that with all but $\frac{1}{m}$ probability, every edge winds up with at most $3\ln m/\ln\ln m$ paths using it.

Zillions of analyses in algorithms (and theoretical computer science more broadly) use this one-two punch of the Chernoff bound and the union bound.

Interestingly, for directed graphs, the approximation guarantee in Theorem 4.1 is optimal, up to a constant factor (assuming $P \neq NP$). For undirected graphs, there is an intriguing gap between the $O(\log n/\log\log n)$ upper bound of Theorem 4.1 and the best-know lower bound of $\Omega(\log\log n)$ (assuming $P \neq NP$).

5 Epilogue

To recap the top 5 essential tools for the analysis of randomized algorithms:

1. *Linearity of expectation.* If all you care about is the expectation of a random variable, this is often good enough.

2. *Markov's inequality.* This inequality usually suffices if you're satisfied with a constant-probability bound.

3. *Chebyshev's inequality.* This inequality is the appropriate one when you have a good handle on the variance of your random variable.

4. *Chernoff bounds.* This inequality gives sharp concentration bounds for random variables that are sums of independent and bounded random variables (most commonly, sums of independent indicator random variables).

5. *Union bound.* This inequality allows you to avoid lots of bad low-probability events.

All five of these tools are insanely useful. And four out of the five have one-line proofs!

CS261: A Second Course in Algorithms
Lecture #19: Beating Brute-Force Search*

Tim Roughgarden†

March 8, 2016

A popular myth is that, for NP-hard problems, there are no algorithms with worst-case running time better than that of brute-force search. Reality is more nuanced, and for many natural NP-hard problems, there are algorithms with (worst-case) running time much better than the naive brute-force algorithm (albeit still exponential). This lecture proves this point by revisiting three problems studied in previous lectures: vertex cover, the traveling salesman problem, and 3-SAT.

1 Vertex Cover and Fixed-Parameter Tractability

This section studies the special case of the vertex cover problem (Lecture #18) in which every vertex has unit weight. That is, given an undirected graph $G = (V, E)$, the goal is to compute a minimum-cardinality subset $S \subseteq V$ that contains at least one endpoint of every edge.

We study the problem of checking whether or not a vertex cover instance admits a vertex cover of size at most k (for a given k). This problem is no easier than the general problem, since the latter reduces to the former by trying all possible values of k. Here, you should think of k as "small," for example between 10 and 20. The graph G can be arbitrarily large, but think of the number of vertices as somewhere between 100 and 1000. We'll show how to beat brute-force search for small k. This will be our only glimpse of "parameterized algorithms and complexity," which is a vibrant subfield of theoretical computer science.

The naive brute-force search algorithm for checking whether or not there is a vertex cover of size at most k is: for every subset $S \subseteq V$ of k vertices, check whether or not S is a vertex cover. The running time of this algorithm scales as $\binom{n}{k}$, which is $\Theta(n^k)$ when k is small. While technically polynomial for any constant k, there is no hope of running this algorithm unless k is extremely small (like 3 or 4).

If we aim to do better, what can we hope for? Better than $\Theta(n^k)$ would a running time of the form $\text{poly}(n) \cdot f(k)$, where the dependence on k and on n can be separated, with

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

the latter dependence only polynomial. Even better would be a running time of the form $\text{poly}(n) + f(k)$ for some function k. Of course, we'd like the $\text{poly}(n)$ term to be as close to linear as possible. We'd also like the function $f(k)$ to be as small as possible, but because the vertex cover problem is NP-hard for general k, we expect $f(k)$ to be at least exponential in k. An algorithm with such a running time is called *fixed-parameter tractable (FPT)* with respect to the parameter k.

We claim that the following is an FPT algorithm for the minimum-cardinality vertex cover problem (with budget k).

FPT Algorithm for Vertex Cover

set $S = \{v \in V : \deg(v) \geq k+1\}$
set $G' = G \setminus S$
set G'' equal to G' with all isolated vertices removed
if G'' has more than k^2 edges **then**
    return "no vertex cover with size $\leq k$"
else
    compute a minimum-size vertex cover T of G'' by brute-force search
    return "yes" if and only if $|S| + |T| \leq k$

We next explain why the algorithm is correct. First, notice that if G has a set cover S of size at most k, then every vertex with degree at least $k+1$ must be in S. For if such a vertex v is not in S, then the other endpoint of each of the (at least $k+1$) edges incident to v must be in the vertex cover; but then $|S| \geq k+1$. In the second step, G' is obtained from G by deleting S and all edges incident to a vertex in S. The edges that survive in G' are precisely the edges not already covered by S. Thus, the vertex covers of size at most k in G are precisely the sets of the form $S \cup T$, where T is a vertex cover of G' size at most $k - |S|$. Given that every vertex cover with size at most k contains the set S, there is no loss in discarding the isolated vertices of G' (all incident edges of such a vertex in G are already covered by vertices in S). Thus, G has a vertex cover of size at most k if and only if G'' has a vertex cover of size at most $k - |S|$. In the fourth step, if G'' has more than k^2 edges, then it cannot possibly have a vertex cover of size at most k (let alone $k - |S|$). The reason is that every vertex of G'' has degree at most k (all higher-degree vertices were placed in S), so each vertex of G'' can only cover k edges, so G'' has a vertex cover of size at most k only if it has at most k^2 edges. The final step computes the minimum-size vertex cover of G'' by brute force, and so is clearly correct.

Next, observe that in the final step (if reached), the graph G'' has at most k^2 edges (by assumption) and hence at most $2k^2$ vertices (since every vertex of G'' has degree at least 1). It follows that the brute-force search step can be implemented in $2^{O(k^2)}$ time. Steps 1–4 can be implemented in linear time, so the overall running time is $O(m) + 2^{O(k^2)}$, and hence the algorithm is fixed-parameter tractable. In FPT jargon, the graph G'' is called a *kernel* (of size $O(k^2)$), meaning that the original problem (on an arbitrarily large graph, with a given budget k) reduces to the same problem on a graph whose size depends only on k. Using

linear programming techniques, it is possible to show that every unweighted vertex cover instance actually admits a kernel with size only $O(k)$, leading to a running time dependence on k of $2^{O(k)}$ rather than $2^{O(k^2)}$. Such singly-exponential dependence is pretty much the best-case scenario in fixed-parameter tractability.

Just as some problems admit good approximation algorithms and others do not (assuming $P \neq NP$), some problems (and parameters) admit fixed-parameter tractable algorithms while others do not (under appropriate complexity assumptions). This is made precise primarily via the theory of "$W[1]$-hardness," which parallels the familiar theory of NP-hardness. For example, the independent set problem, despite its close similarity to the vertex cover problem (the complement of a vertex cover is an independent set and vice versa), is $W[1]$-hard and hence does not seem to admit a fixed-parameter tractable algorithm (parameterized by the size of the largest independent set).

2 TSP and Dynamic Programming

Recall from Lecture #16 the traveling salesman problem (TSP): the input is a complete undirected graph with non-negative edge weights, and the goal to compute the minimum-cost TSP tour, meaning a simple cycle that visits every vertex exactly once. We saw in Lecture #16 that the TSP problem is hard to even approximate, and for this reason we focused on approximation algorithms for the (still NP-hard) special case of the metric TSP. Here, we'll give an exact algorithm for TSP, and we won't even assume that the edges satisfy the triangle inequality.

The naive brute-force search algorithm for TSP tries every possible tour, leading to a running time of roughly $n!$, where n is the number of vertices. Recall that $n!$ grows considerably faster than any function of the form c^n for a constant c (see also Section 3). Naive brute-force search is feasible with modern computers only for n in the range of 12 or 13. This section gives a dynamic programming algorithm for TSP that runs in $O(n^2 2^n)$ time. This extends the "tractability frontier" for n into the 20s. One drawback of the dynamic programming algorithm is that it also uses exponential space (unlike brute-force search). It is an open question whether or not there is an exact algorithm for TSP that has running time $O(c^n)$ for a constant $c > 1$ and also uses only a polynomial amount of space. Two take-aways from the following algorithm are: (i) TSP is another fundamental NP-hard problem for which algorithmic ingenuity beats brute-force search; and (ii) your algorithmic toolbox (here, dynamic programming) continues to be extremely useful for the design of exact algorithms for NP-hard problems.

Like any dynamic programming algorithm, the plan is to solve systematically a collection of subproblems, from "smallest" to "largest," and then read off the final answer from the biggest subproblems. Coming up with right subproblems is usually the hardest part of designing a dynamic programming algorithm. Here, in the interests of time, we'll just cut to the chase and state the relevant subproblems.

Let $V = \{1, 2, \ldots, n\}$ be the vertex set. The algorithm populates a two-dimensional array A, with one dimension indexed by a subset $S \subseteq V$ of vertices and the other dimension

indexed by a single vertex j. At the end of the algorithm, the entry $A[S, j]$ will contain the cost of the minimum-cost path that:

(i) visits every vertex $v \in S$ exactly once (and no other vertices);

(ii) starts at the vertex 1 (so 1 better be in S);

(iii) ends at the vertex j (so j better be in S).

There are $O(n2^n)$ subproblems. Since the TSP is NP-hard, we should not be surprised to see an exponential number of subproblems.

After solving all of the subproblems, it is easy to compute the cost of an optimal tour in linear time. Since $A[\{1, 2, \ldots, n\}, j]$ contains the length of the shortest path from 1 to j that visits every vertex exactly once, we can just "guess" (i.e., do brute-force search over) the vertex preceding 1 on the tour:

$$OPT = \min_{j=2}^{n} \left(\underbrace{A[\{1, 2, \ldots, n\}, j]}_{\text{path from 1 to } j} + \underbrace{c_{j1}}_{\text{last hop}} \right).$$

Next, we need a principled way to solve all of the subproblems, using solutions to previously solved "smaller" subproblems to quickly solve "larger" subproblems. That is, we need a *recurrence* relating the solutions of different subproblems. So consider a subproblem $A[S, j]$, where the goal is to compute the minimum cost of a path subject to (i)–(iii) above. What must the optimal solution look like? If we only knew the penultimate vertex k on the path (right before j), then we would know what the path looks like: it would be the cheapest possible path visiting each of the vertices of $S \setminus \{j\}$ exactly once, starting at 1, and ending at k (why?), followed of course by the final hop from k to j. Our recurrence just executes brute-force search over all of the legitimate choices of k:

$$A[S, j] = \min_{k \in S \setminus \{1, j\}} \left(A[S \setminus \{j\}, k] + c_{kj} \right).$$

This recurrence assumes that $|S| \geq 3$. If $|S| = 1$ then $A[S, j]$ is 0 if $S = \{1\}$ and $j = 1$ and is $+\infty$ otherwise. If $|S| = 2$, then the only legitimate choice of k is 1.

The algorithm first solves all subproblems with $|S| = 1$, then all subproblems with $|S| = 2$, ..., and finally all subproblems with $|S| = n$ (i.e., $S = \{1, 2, \ldots, n\}$). When solving a subproblem, the solutions to all relevant smaller subproblems are available for constant-time lookup. Each subproblem can thus be solved in $O(n)$ time. Since there are $O(n2^n)$ subproblems, we obtain the claimed running time bound of $O(n^2 2^n)$.

3 3SAT and Random Search

3.1 Schöning's Algorithm

Recall from last lecture that a 3SAT formula involves n Boolean variables $x_1, \ldots, x_n$ and m clauses, where each clause is the disjunction of three literals (where a literal is a variable or

its negation). Last lecture we studied MAX 3SAT, the optimization problem of satisfying as many of the clauses as possible. Here, we'll study the simpler decision problem, where the goal is to check whether or not there is a assignment that satisfies all m clauses. Recall that this is the canonical example of an NP-complete problem (cf., the Cook-Levin theorem).

Naive brute-force search would try all 2^n truth assignments. Can we do better than exhaustive search? Intriguingly, we can, with a simple algorithm and by a pretty wide margin. Specifically, we'll study Schöning's random search algorithm (from 1999). The parameter T will be determined later.

Random Search Algorithm for 3SAT (Version 1)

repeat T times (or until a satisfying assignment is found):
    choose a truth assignment **a** uniformly at random
    repeat n times (or until a satisfying assignment is found):
        choose a clause C violated by the current assignment **a**
        choose one the three literals from C uniformly at random, and
         modify **a** by flipping the value of the corresponding variable
         (from "true" to "false" or vice versa)
if a satisfying assignment was found **then**
    return "satisfiable"
else
    return "unsatisfiable"

And that's it![1]

3.2 Analysis (Version 1)

We give three analyses of Schöning's algorithm (and a minor variant), each a bit more sophisticated and establishing a better running time bound than the last. The first observation is that the algorithm never makes a mistake when the formula is unsatisfiable — it will never find a satisfying assignment (no matter what its coin flips are), and hence reports "unsatisfiable." So what we're worried about is the algorithm failing to find a satisfying assignment when one exists. So for the rest of the lecture, we consider only satisfiable instances. We use $\mathbf{a}^*$ to denote a reference satisfying assignment (if there are many, we pick one arbitrarily). The high-level idea is to track the "Hamming distance" between $\mathbf{a}^*$ and our current truth assignment $\mathbf{a}$ (i.e., the number of variables with different values in $\mathbf{a}$ and $\mathbf{a}^*$). If this Hamming distance ever drops to 0, then $\mathbf{a} = \mathbf{a}^*$ and the algorithm has found a satisfying assignment.

[1] A little backstory: an analogous algorithm for 2SAT (2 literals per clause) was studied earlier by Papadimitriou. 2SAT is polynomial-time solvable — for example, it can be solved in linear time via a reduction to computing the strongly connected components of a suitable directed graph. Papadimitriou's random search algorithm is slower but still polynomial ($O(n^2)$), with the analysis being a nice exercise in random walks (covered in the instructor's Coursera videos).

A simple observation is that, if the current assignment $\mathbf{a}$ fails to satisfy a clause C, then $\mathbf{a}$ assigns at least one of the three variables in C a different value than $\mathbf{a}^*$ does (as $\mathbf{a}^*$ satisfies the clause). Thus, when the random search algorithm chooses a variable of a violated clause to flip, there is at least a $1/3$ chance that the algorithm chooses a "good variable," the flipping of which decreases the Hamming distance between $\mathbf{a}$ and $\mathbf{a}^*$ by one. (If $\mathbf{a}$ and $\mathbf{a}^*$ differ on more than one variable of C, then the probability is higher.) In the other case, when the algorithm chooses a "bad variable," where $\mathbf{a}$ and $\mathbf{a}^*$ give it the same value, flipping the value of the variable in $\mathbf{a}$ increases the Hamming distance between $\mathbf{a}$ and $\mathbf{a}^*$ by 1. This happens with probability at most $2/3$.[2]

All of the analyses proceed by identifying simple sufficient conditions for the random search algorithm to find a satisfying assignment, bounding below the probability that these sufficient conditions are met, and then choosing T large enough that the algorithm is guaranteed to succeed with high probability.

To begin, suppose that the initial random assignment $\mathbf{a}$ chosen in an iteration of the outer loop differs from the reference satisfying assignment $\mathbf{a}^*$ in k variables. A sufficient condition for the algorithm to succeed is that, in every one of the first k iterations of the inner loop, the algorithm gets lucky and flips the value of a variable on which $\mathbf{a}, \mathbf{a}^*$ differ. Since each inner loop iteration has a probability of at least $1/3$ of choosing wisely, and the random choices are independent, this sufficient condition for correctness holds with probability at least 3^{-k}. (The algorithm might stop early if it stumbles on a satisfying assignment other than $\mathbf{a}^*$; this is obviously fine with us.)

For our first analysis, we'll use a sloppy argument to analyze the parameter k (the distance between $\mathbf{a}$ and $\mathbf{a}^*$ at the beginning of an outer loop iteration). By symmetry, $\mathbf{a}$ agrees with $\mathbf{a}^*$ on at least half the variables (i.e., $k \leq n/2$) with probability at least $1/2$. Conditioning on this event, we conclude that a single outer loop iteration successfully finds a satisfying assignment with probability at least $p = \frac{1}{2 \cdot 3^{n/2}}$. Hence, the algorithm finds a satisfying assignment in one of the T outer loop iterations except with probability at most $(1-p)^T \leq e^{-pT}$.[3] If we take $T = \frac{d \ln n}{p}$ for a constant $d > 0$, then the algorithm succeeds except with inverse polynomial probability $\frac{1}{n^d}$. Substituting for p, we conclude that

$$T = \Theta\left((\sqrt{3})^n \log n\right)$$

outer loop iterations are enough to be correct with high probability. This gives us an algorithm with running time $O((1.74)^n)$, which is already significantly better than the 2^n dependence in brute-force search.

[2] The fact that the random process is biased toward moving farther away from $\mathbf{a}^*$ is what gives rise to the exponential running time. In the case of 2SAT, each random move is at least as likely to decrease the distance as increase the distance, which in turn leads to a polynomial running time.

[3] Recall the useful inequality $1 + x \leq e^x$ for all $x \in \mathbb{R}$, used also in Lectures #11 (see the plot there) and #15.

3.3 Analysis (Version 2)

We next give a refined analysis of the same algorithm. The plan is to count the probability of success for all values of the initial distance k, not just when $k \leq n/2$ (and not assuming the worst case of $k = n/2$).

For a given choice of $k \in \{1, 2, \ldots, n\}$, what is the probability that the initial assignment $\mathbf{a}$ and $\mathbf{a}^*$ differ in their values to exactly k variables? There is one such assignment for each of the $\binom{n}{k}$ choices of a set S of k out of n variables. (The corresponding assignment $\mathbf{a}$ agrees with $\mathbf{a}^*$ on S and disagrees with $\mathbf{a}^*$ outside of S.) Since all truth assignments are equally likely (probability 2^{-n} each),

$$\mathbf{Pr}[\text{dist}(\mathbf{a}, \mathbf{a}^*) = k] = \binom{n}{k} 2^{-n}.$$

We can now lower bound the probability of success of an outer loop iteration by conditioning on k:

$$\begin{aligned}
\mathbf{Pr}[\text{success}] &= \sum_{k=0}^{n} \mathbf{Pr}[\text{dist}(\mathbf{a}, \mathbf{a}^*) = k] \cdot \mathbf{E}[\text{success} \mid \text{dist}(\mathbf{a}, \mathbf{a}^*) = k] \\
&\geq \sum_{k=0}^{n} \binom{n}{k} 2^{-n} \left(\frac{1}{3}\right)^k \\
&= 2^{-n}(1 + \tfrac{1}{3})^n \\
&= \left(\frac{2}{3}\right)^n,
\end{aligned}$$

where the penultimate equality follows from a slick application of the binomial formula.[4]

Thus, taking $T = \Theta((\frac{3}{2})^n \log n)$, the random search algorithm is correct with high probability.

3.4 Analysis (Version 3)

For the final analysis, we tweak the version of Schöning's algorithm above slightly, replacing "repeat n times" in the inner loop by "repeat $3n$ times." This only increases the running time by a constant factor.

Our two previous analyses only considered the cases where the random search algorithm made a beeline for the reference satisfying assignment $\mathbf{a}^*$, never making an incorrect choice of which variable to flip. There are also other cases where the algorithm will succeed. For example, if the algorithm chooses a bad variable once (increasing $\text{dist}(\mathbf{a}, \mathbf{a}^*)$ by 1), but then a good variable $k + 1$ times, then after these $k + 2$ iterations $\mathbf{a}$ is the same as the satisfying assignment $\mathbf{a}^*$ (unless the algorithm stopped early due to finding a different satisfying assignment).

[4] I.e., the formula $(a + b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$.

For the analysis, we'll focus on the specific case where, in the first $3k$ inner loop iterations, the algorithm chooses a bad variable k times and a good variable $2k$ times. This idea leads to

$$\mathbf{Pr}[\text{success}] \geq \sum_{k=0}^{n} 2^{-n} \binom{n}{k} \binom{3k}{k} \left(\frac{1}{3}\right)^{2k} \left(\frac{2}{3}\right)^{k}, \tag{1}$$

since the probability that the random local search algorithm chooses a good variable $2k$ times in the first $3k$ inner loop iterations is at least $\binom{3k}{k}(\frac{1}{3})^{2k}(\frac{2}{3})^{k}$.

This inequality is pretty messy, with no less than two binomial coefficients complicating each summand. We'll be able to handle the $\binom{n}{k}$ terms using the same slick binomial expansion trick from the previous analysis, but the $\binom{3k}{k}$ terms are more annoying. To deal with them, recall *Stirling's approximation* for the factorial function:

$$n! = \Theta\left(\sqrt{n}\left(\frac{n}{e}\right)^{n}\right).$$

(The hidden constant is $\sqrt{2\pi}$, but we won't need to worry about that.) Thus, in the grand scheme of things, $n!$ is not all that much smaller than n^n.

We can use Stirling's approximation to simplify $\binom{3k}{k}$:

$$\begin{aligned}
\binom{3k}{k} &= \frac{(3k!)}{(2k)!k!} \\
&= \Theta\left(\frac{\sqrt{3k}}{\sqrt{k}\sqrt{2k}} \cdot \frac{(\frac{3k}{e})^{3k}}{(\frac{k}{e})^{k}(\frac{2k}{e})^{2k}}\right) \\
&= \Theta\left(\frac{1}{\sqrt{k}} \cdot \frac{3^{3k}}{2^{2k}}\right).
\end{aligned}$$

Thus,

$$\underbrace{\binom{3k}{k}}_{=\Theta(3^{3k}/2^{2k}\sqrt{k})} \left(\frac{1}{3}\right)^{2k} \left(\frac{2}{3}\right)^{k} = \Theta\left(\frac{2^{-k}}{\sqrt{k}}\right).$$

Substituting back into (1), we find that for some constant $c > 0$ (hidden in the Θ notation),

$$\begin{aligned}
\mathbf{Pr}[\text{success}] &\geq \sum_{k=0}^{n} 2^{-n} \binom{n}{k} \binom{3k}{k} \left(\frac{1}{3}\right)^{2k} \left(\frac{2}{3}\right)^{k} \\
&\geq c2^{-n} \sum_{k=0}^{n} \binom{n}{k} \frac{2^{-k}}{\sqrt{k}} \\
&\geq \frac{c}{\sqrt{n}} 2^{-n} \sum_{k=0}^{n} \binom{n}{k} 2^{-k} \\
&= \frac{c}{\sqrt{n}} 2^{-n} \left(1 + \frac{1}{2}\right)^{n} \\
&= \frac{c}{\sqrt{n}} \left(\frac{3}{4}\right)^{n}.
\end{aligned}$$

We conclude that with $T = \Theta\left(\left(\frac{4}{3}\right)^{n} \sqrt{n} \log n\right)$, the algorithm is correct with high probability.

This running time of $\approx (\frac{4}{3})^n$ has been improved somewhat since 1999, but this is still quite close to the state of the art, and it is an impressive improvement over the $\approx 2^n$ running time require by brute-force search. Can we do even better? This is an open question. The *exponential time hypothesis (ETH)* asserts that every correct algorithm for 3SAT has worst-case running time at least c^n for some constant $c > 1$. (For example, this rules out a "quasi-polynomial-time" algorithm, with running time $n^{\text{polylog}(n)}$.) The ETH is certainly a stronger assumption than $P \neq NP$, but most experts believe that it is true.

The random search idea can be extended from 3SAT to k-SAT for all constant values of k. For every constant k, the result is an algorithm that runs in time $O(c^n)$ for a constant $c < 2$. However, the constant c tends to 2 as k tends to infinity. The *strong exponential time hypothesis (SETH)* asserts that this is necessary — that there is no algorithm for the general SAT problem (with k arbitrary) that runs in worst-case running time $O(c^n)$ for some constant $c < 2$ (independent of k). Expert opinion is mixed on whether or not SETH holds. If it *does* hold, then there are interesting consequences for lots of different problems, ranging from the prospects of fixed-parameter tractable algorithms for NP-hard problems (Section 1) to lower bounds for classic algorithmic problems like computing the edit distance between two strings.

CS261: A Second Course in Algorithms
Lecture #2: Augmenting Path Algorithms for Maximum Flow*

Tim Roughgarden†

January 7, 2016

1 Recap

Figure 1: (a) original edge capacity and flow and (b) resultant edges in residual network.

Recall where we left off last lecture. We're considering a directed graph $G = (V, E)$ with a source s, sink t, and an integer capacity u_e for each edge $e \in E$. A flow is a nonnegative vector $\{f_e\}_{e \in E}$ that satisfies capacity constraints ($f_e \leq u_e$ for all e) and conservation constraints (flow in = flow out except at s and t).

Recall that given a flow f in a graph G, the corresponding residual network has two edges for each edge e of G, a forward edge with residual capacity $u_e - f_e$ and a reverse edge with residual capacity f_e that allows us to "undo" previously routed flow. See also Figure 1.[1]

The Ford-Fulkerson algorithm repeatedly finds an s-t path P in the current residual graph G_f, and augments along p as much as possible subject to the capacity constraints of

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

[1]We usually implicitly assume that all edges with zero residual capacity are omitted from the residual network.

the residual network.[2] We argued that the algorithm eventually terminates with a feasible flow. But is it a maximum flow? More generally, a major course theme is to understand

How do we know when we're done?

For example, could the maximum flow value in the network in Figure 2 really just be 3?

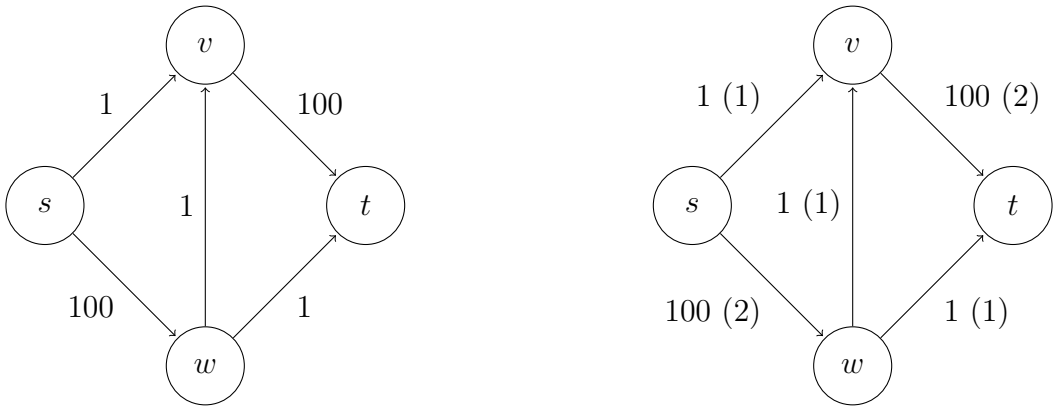

Figure 2: (a) A given network and (b) the alleged maximum flow of value 3.

2 Around the Maximum-Flow/Minimum-Cut Theorem

We ended last lecture with a claim that if there is no s-t path (with positive residual capacity on every edge) in the residual graph G_f, then f is a maximum flow in G. It's convenient to prove a stronger statement, from which we can also derive the famous maximum-flow/minimum cut theorem.

2.1 (s,t)-Cuts

To state the stronger result, we need an important definition, of objects that are "dual" to flows in a sense we'll make precise later.

Definition 2.1 (s-t Cut) An (s,t)-*cut* of a graph $G = (V,E)$ is a partition of V into sets A, B with $s \in A$ and $t \in B$.

Sometimes we'll simply say "cut" instead of "(s,t)-cut."

Figure 3 depicts a good (if cartoonish) way to think about an (s,t)-cut of a graph. Such a cut buckets the edges of the graph into four categories: those with both endpoints in A, those with both endpoints in B, those sticking out of A (with tail in A and head in B), and those sticking into A (with head in A and tail in B.

[2]To be precise, the algorithm finds an s-t path in G_f such that every edge has strictly positive residual capacity. Unless otherwise noted, in this lecture by "G_f" we mean the edges with positive residual capacity.

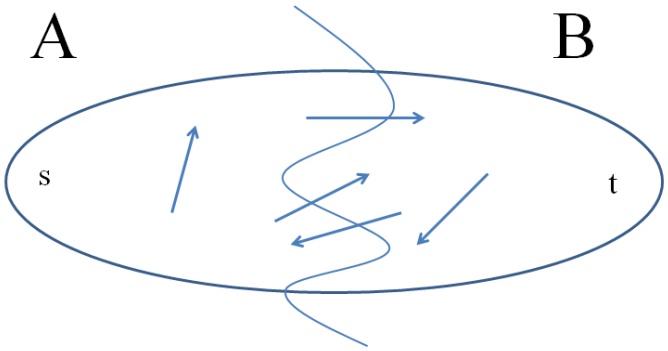

Figure 3: cartoonish visualization of cuts. The squiggly line splits the vertices into two sets A and B and edges in the graph into 4 categories.

The *capacity* of an (s,t)-cut (A,B) is defined as

$$\sum_{e \in \delta^+(A)} u_e.$$

where $\delta^+(A)$ denotes the set of edges sticking out of A. (Similarly, we later use $\delta^-(A)$ to denote the set of edges sticking into A.)

Note that edges sticking in to the source-side of an (s,t)-cut to do not contribute to its capacity. For example, in Figure 2, the cut $\{s,w\}$, $\{v,t\}$ has capacity 3 (with three outgoing edges, each with capacity 1). Different cuts have different capacities. For example, the cut $\{s\}$, $\{v,w,t\}$ in Figure 2 has capacity 101. A *minimum cut* is one with the smallest capacity.

2.2 Optimality Conditions for the Maximum Flow Problem

We next prove the following basic result.

Theorem 2.2 (Optimality Conditions for Max Flow) *Let f be a flow in a graph G. The following are equivalent:*[3]

(1) f is a maximum flow of G;

(2) there is an (s,t)-cut (A,B) such that the value of f equals the capacity of (A,B);

(3) there is no s-t path (with positive residual capacity) in the residual network G_f.

Theorem 2.2 asserts that any one of the three statements implies the other two. The special case that (3) implies (1) recovers the claim from the end of last lecture.

[3]Meaning, either all three statements hold, or none of the three statements hold.

Corollary 2.3 *If f is a flow in G such that the residual network G_f has no s-t path, then the f is a maximum flow.*

Recall that Corollary 2.3 implies the correctness of the Ford-Fulkerson algorithm, and more generally of any algorithm that terminates with a flow and a residual network with no s-t path.

Proof of Theorem 2.2: We prove a cycle of implications: (2) implies (1), (1) implies (3), and (3) implies (2). It follows that any one of the statements implies the other two.

Step 1: (2) implies (1): We claim that, for every flow f and every (s,t)-cut (A,B),

$$\text{value of } f \leq \text{capacity of } (A,B).$$

This claim implies that all flow values are at most all cut values; for a cartoon of this, see Figure 4. The claim implies that there no "x" strictly to the right of the "o".

Figure 4: cartoon illustrating that no flow value (x) is greater than a cut value (o).

To see why the claim yields the desired implication, suppose that (2) holds. This corresponds to an "x" and "o" that are co-located in Figure 4. By the claim, no "x"s can appear to the right of this point. Thus no flow has larger value than f, as desired.

We now prove the claim. If it seems intuitively obvious, then great, your intuition is spot-on. For completeness, we provide a brief algebraic proof.

Fix f and (A,B). By definition,

$$\text{value of } f = \underbrace{\sum_{e \in \delta^+(s)} f_e}_{\text{flow out of } s} = \sum_{e \in \delta^+(s)} f_e - \underbrace{\sum_{e \in \delta^-(s)} f_e}_{\text{vacuous sum}}; \tag{1}$$

the second equation is stated for convenience, and follows from our standing assumption that s has no incoming vertices. Recall that conservation constraints state that

$$\underbrace{\sum_{e \in \delta^+(v)} f_e}_{\text{flow out of } v} - \underbrace{\sum_{e \in \delta^-(v)} f_e}_{\text{flow into of } v} = 0 \tag{2}$$

for every $v \neq s,t$. Adding the equations (2) corresponding to all of the vertices of $A \setminus \{s\}$ to equation (1) gives

$$\text{value of } f = \sum_{v \in A} \left(\sum_{e \in \delta^+(v)} f_e - \sum_{e \in \delta^-(v)} f_e \right). \tag{3}$$

Next we want to think about the expression in (3) from an edge-centric, rather than vertex-centric, perspective. How much does an edge e contribute to (3)? The answer depends on which of the four buckets e falls into (Figure 3). If both of e's endpoints are in B, then e is not involved in the sum (3) at all. If $e = (v, w)$ with both endpoints in A, then it contributes f_e once (in the subexpression $\sum_{e \in \delta^+(v)} f_e$) and $-f_e$ once (in the subexpression $-\sum_{e \in \delta^-(w)} f_e$). Thus edges inside A contribute net zero to (3). Similarly, an edge e sticking out of A contributes f_e, while an edge sticking into A contributes $-f_e$. Summarizing, we have

$$\text{value of } f = \sum_{e \in \delta^+(A)} f_e - \sum_{e \in \delta^-(A)} f_e.$$

This equation states that the net flow (flow forward minus flow backward) across every cut is exactly the same, namely the value of the flow f.

Finally, using the capacity constraints and the fact that all flows values are nonnegative, we have

$$\begin{aligned} \text{value of } f &= \sum_{e \in \delta^+(A)} \underbrace{f_e}_{\leq u_e} - \sum_{e \in \delta^-(A)} \underbrace{f_e}_{\geq 0} \\ &\leq \sum_{e \in \delta^+(A)} u_e \qquad (4) \\ &= \text{capacity of } (A, B), \qquad (5) \end{aligned}$$

which completes the proof of the first implication.

Step 2: (1) implies (3): This step is easy. We prove the contrapositive. Suppose f is a flow such that G_f has an s-t path P with positive residual capacity. As in the Ford-Fulkerson algorithm, we augment along P to produce a new flow f' with strictly larger value. This shows that f is not a maximum flow.

Step 3: (3) implies (2): The final step is short and sweet. The trick is to define

$$A = \{v \in V \,:\, \text{there is an } s \rightsquigarrow v \text{ path in } G_f\}.$$

Conceptually, start your favorite graph search subroutine (e.g., BFS or DFS) from s until you get stuck; A is the set of vertices you get stuck at. (We're running this graph search only in our minds, for the purposes of the proof, and not in any actual algorithm.)

Note that $(A, V - A)$ is an (s, t)-cut. Certainly $s \in A$, so s can reach itself in G_f. By assumption, G_f has no s-t path, so $t \notin A$. This cut must look like the cartoon in Figure 5, with no edges (with positive residual capacity) sticking out of A. The reason is that if there *were* such an edge sticking out of A, then our graph search would not have gotten stuck at A, and A would be a bigger set.

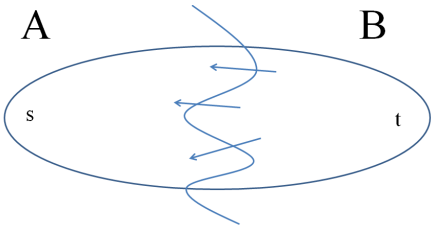

Figure 5: Cartoon of the cut. Note that edges crossing the cut only go from B to A.

Let's translate the picture in Figure 5, which concerns the residual network G_f, back to the flow f in the original network G.

1. Every edge sticking out of A in G (i.e., in $\delta^+(A)$) is saturated (meaning $f_u = u_e$). For if $f_e < u_e$ for $e \in \delta^+(A)$, then the residual network G_f would contain a forward version of e (with positive residual capacity) which would be an edge sticking out of A in G_f (contradicting Figure 5).

2. Every edge sticking into in A in G (i.e., in $\delta^-(A)$) is zeroed out ($f_u = 0$). For if $f_e < u_e$ for $e \in \delta^+(A)$, then the residual network G_f would contain a forward version of e (with positive residual capacity) which would be an edge sticking out of A in G_f (contradicting Figure 5).

These two points imply that the inequality (4) holds with equality, with

$$\text{value of } f = \text{capacity of } (A, V - A).$$

This completes the proof. ■

We can immediately derive some interesting corollaries of Theorem 2.2. First is the famous *Max-Flow/Min-Cut Theorem*.[4]

Corollary 2.4 (Max-Flow/Min-Cut Theorem) *In every network,*

$$\textit{maximum value of a flow} = \textit{minimum capacity of an } (s,t)\textit{-cut.}$$

Proof: The first part of the proof of Theorem 2.2 implies that the maximum value of a flow cannot exceed the minimum capacity of an (s,t)-cut. The third part of the proof implies that there cannot be a gap between the maximum flow value and the minimum cut capacity. ■

Next is an algorithmic consequence: the minimum cut problem reduces to the maximum flow problem.

Corollary 2.5 *Given a maximum flow, and minimum cut can be computed in linear time.*

[4]This is the theorem that, long ago, seduced your instructor into a career in algorithms.

Proof: Use BFS or DFS to compute, in linear time, the set A from the third part of the proof of Theorem 2.2. The proof shows that $(A, V - A)$ is a minimum cut. ■

In practice, minimum cuts are typically computed using a maximum flow algorithm and this reduction.

2.3 Backstory

Ford and Fulkerson published in the max-flow/min-cut theorem in 1955, while they were working at the RAND Corporation (a military think tank created after World War II). Note that this was in the depths of the Cold War between the (then) Soviet Union and the United States. Ford and Fulkerson got the problem from Air Force researcher Theodore Harris and retired Army general Frank Ross. Harris and Ross has been given, by the CIA, a map of the rail network connecting the Soviet Union to Eastern Bloc countries like Poland, Czechoslovakia, and Eastern Germany. Harris and Ross formed a graph, with vertices corresponding to administrative districts and edge capacities corresponding to the rail capacity between two districts. Using heuristics, Harris and Ross computed both a maximum flow and minimum cut of the graph, noting that they had equal value. They were rather more interested in the minimum cut problem (i.e., blowing up the least amount of train tracks to sever connectivity) than the maximum flow problem! Ford and Fulkerson proved more generally that in *every* network, the maximum flow value equals that minimum cut capacity. See [**?**] for further details.

3 The Edmonds-Karp Algorithm: Shortest Augmenting Paths

3.1 The Algorithm

With a solid understanding of when and why maximum flow algorithms are correct, we now focus on optimizing the running time. Exercise Set #1 asks to show that the Ford-Fulkerson algorithm is not a polynomial-time algorithm. It is a "pseudopolynomial-time" algorithm, meaning that it runs in polynomial time provided all edge capacities are polynomially bounded integers. With big edge capacities, however, the algorithm can require a very large number of iterations to complete. The problem is that the algorithm can keep choosing a "bad path" over and over again. (Recall that when the current residual network has multiple s-t paths, the Ford-Fulkerson algorithm chooses arbitrarily.) This motivates choosing augmenting paths more intelligently. The *Edmonds-Karp algorithm* is the same as the Ford-Fulkerson algorithm, except that it always chooses a *shortest* augmenting path of the residual graph (i.e., with the fewest number of hops). Upon hearing "shortest paths" you may immediately think of Dijkstra's algorithm, but this is overkill here — breadth-first search already computes (in linear time) a path with the fewest number of hops.

Edmonds-Karp Algorithm

initialize $f_e = 0$ for all $e \in E$
repeat
  compute an s-t path P (with positive residual capacity) in the current residual graph G_f with the fewest number of edges
  `// takes` $O(|E|)$ `time using BFS`
  if no such path **then**
    halt with current flow $\{f_e\}_{e \in E}$
  else
    let $\Delta = \min_{e \in P}$ (e's residual capacity in G_f)
    `// augment the flow` f `using the path` P
    for all edges e of G whose corresponding forward edge is in P **do**
      increase f_e by Δ
    for all edges e of G whose corresponding reverse edge is in P **do**
      decrease f_e by Δ

3.2 The Analysis

As a specialization of the Ford-Fulkerson algorithm, the Edmonds-Karp algorithm inherits its correctness. What about the running time?

Theorem 3.1 (Running Time of Edmonds-Karp [?]) *The Edmonds-Karp algorithm runs in $O(m^2 n)$ time.*[5]

Recall that m typically varies between $\approx n$ (the sparse case) and $\approx n^2$ (the dense case), so the running time in Theorem 3.1 is between n^3 and n^5. This is quite slow, but at least the running time is polynomial, no matter how big the edge capacities are. See below and Problem Set #1 for some faster algorithms.[6] Why study Edmonds-Karp, when we're just going to learn faster algorithms later? Because it provides a gentle introduction to some fundamental ideas in the analysis of maximum flow algorithms.

Lemma 3.2 (EK Progress Lemma) *Fix a network G. For a flow f, let $d(f)$ denote the number of hops in a shortest s-t path (with positive residual capacity) in G_f, or $+\infty$ if no such paths exist.*

(a) $d(f)$ never decreases during the execution of the Edmonds-Karp algorithm.

(b) $d(f)$ increases at least once per m iterations.

[5] In this course, m always denotes the number $|E|$ of edges, and n the number $|V|$ of vertices.

[6] Many different methods yield running times in the $O(mn)$ range, and state-of-the-art algorithm are still a bit faster. It's an open question whether or not there is a near-linear maximum flow algorithm.

Since $d(f) \in \{0, 1, 2, \ldots, n-2, n-1, +\infty\}$, once $d(f) \geq n$ we know that $d(f) = +\infty$ and s and t are disconnected in G_f.[7] Thus, Lemma 3.2 implies that the Edmonds-Karp algorithm terminates after at most mn iterations. Since each iteration just involves a breadth-first-search computation, we get the running time of $O(m^2n)$ promised in Theorem 3.1.

For the analysis, imagine running breadth-first search (BFS) in G_f starting from the source s. Recall that BFS discovers vertices in "layers," with s in the 0th layer, and layer $i+1$ consisting of those vertices not in a previous layer and reachable in one hop from a vertex in the ith layer. We can then classify the edges of G_f as *forward* (meaning going from layer i to layer $i+1$, for some i), *sideways* (meaning both endpoints are in the same layer), and *backwards* (traveling from a layer i to some layer j with $j < i$). By the definition of breadth-first search, no forward edge of G_f can shortcut over a layer; every forward edge goes only to the next layer.

We define L_f, with the L standing for "layered," as the subgraph of G_f consisting only of the forward edges (Figure 6). (Vertices in layers after the one containing t are irrelevant, so they can be discarded if desired.)

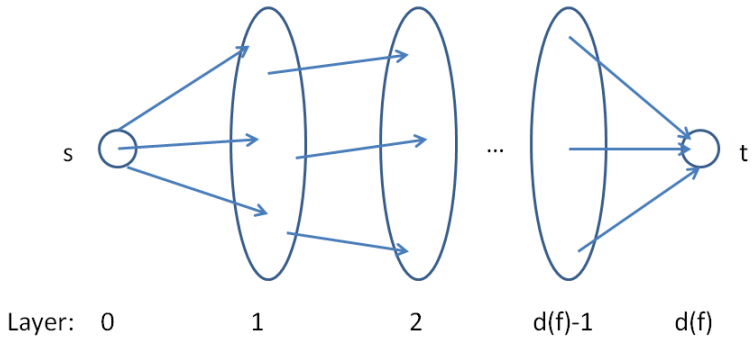

Figure 6: Layered subgraph L_f

Why bother defining L_f? Because it is a succinct encoding of all of the shortest s-t paths of G_f — the paths on which the Edmonds-Karp algorithm might augment. Formally, every s-t in L_f comprises only forward edges of the BFS and hence has exactly $d(f)$ hops, the minimum possible. Conversely, an s-t path that is in G_f but not L_f must contain at least one detour (a sideways or backward edge) and hence requires at least $d(f)+1$ hops to get to t.

[7] Any path with n or more edges has a repeated vertex, and deleted the corresponding cycle yields a path with the same endpoints and fewer hops.

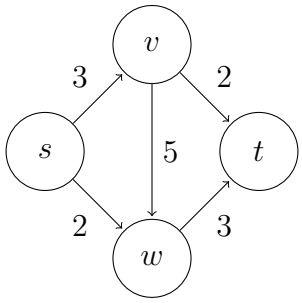

Figure 7: Example from first lecture. Initially, 0th layer is $\{s\}$, 1st layer is $\{v, w\}$, 2nd layer is $\{t\}$.

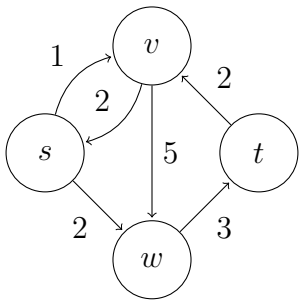

Figure 8: Residual graph after sending flow on $s \to v \to t$. 0th layer is $\{s\}$, 1st layer is $\{v, w\}$, 2nd layer is $\{t\}$.

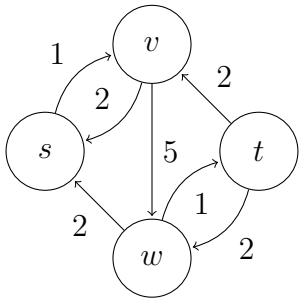

Figure 9: Residual graph after sending additional flow on $s \to w \to t$. 0th layer is $\{s\}$, 1st layer is $\{v\}$, 2nd layer is $\{w\}$, 3rd layer is $\{t\}$.

For example, let's return to our first example in Lecture #1, shown in Figure 7. Let's watch how $d(f)$ changes as we simulate the algorithm. Since we begin with the zero flow, initially the residual graph G_f is the original graph G. The 0th layer is s, the first layer is $\{v, w\}$, and the second layer is t. Thus $d(f) = 2$ initially. There are two shortest paths,

$s \to v \to t$ and $s \to w \to t$. Suppose the Edmonds-Karp algorithm chooses to augment on the upper path, sending two units of flow. The new residual graph is shown in Figure 8. The layers remain the same: $\{s\}$, $\{v, w\}$, and $\{t\}$, with $d(f)$ still equal to 2. There is only one shortest path, $s \to w \to t$. The Edmonds-Karp algorithm sends two units along this flow, resulting in the new residual graph in Figure 9. Now, no two-hop paths remain: the first layer contains only v, with w in second layer and t in the third layer. Thus, $d(f)$ has jumped from 2 to 3. The unique shortest path is $s \to v \to w \to t$, and after the Edmonds-Karp algorithm pushes one unit of flow on this path it terminates with a maximum flow.

Proof of Lemma 3.2: We start with part (a) of the lemma. Note that the only thing we're worried about is that an augmentation somehow introduces a new, shortest path that shortcuts over some layers of L_f (as defined above).

Suppose the Edmonds-Karp algorithm augments the current flow f by routing flow on the path P. Because P is a shortest s-t path in G_f, it is also a path in the layered graph L_f. The only new edges created by augmenting on P are edges that go in the reverse direction of P. These are all backward edges, so any s-t of G_f that uses such an edge has at least $d(f) + 2$ hops. Thus, no new shorter paths are formed in G_f after the augmentation.

Now consider a run of t iterations of the Edmonds-Karp algorithm in which the value of $d(f) = c$ stays constant. We need to show that $t \leq m$. Before the first of these iterations, we save a copy of the current layered network: let F denote the edges of L_f at this time, and $V_0 = \{s\}, V_1, V_2, \ldots, V_c$ the vertices if the various layers.[8]

Consider the first of these t iterations. As in the proof of part (a), the only new edges introduced go from some V_i to V_{i-1}. By assumption, after the augmentation, there is still an s-t path in the new residual graph with only c hops. Since no edge of of such a path can shortcut over one of the layers $V_0, V_1, \ldots, V_c$, it must consist only of edges in F. Inductively, every one of these t iterations augments on a path consisting solely of edges in F. Each such iteration zeroes out at least one edge $e = (v, w)$ of F (the one with minimum residual capacity), at which point edge e drops out of the current residual graph. The only way e can reappear in the residual graph is if there is an augmentation in the reverse direction (the direction (w, v)). But since (w, v) goes backward (from some V_i to V_{i-1}) and all of the t iterations route flow only on edges of F (from some V_i to to V_{i+1}), this can never happen. Since F contains at most m edges, there can only be m iterations before $d(f)$ increases (or the algorithm terminates). ■

4 Dinic's Algorithm: Blocking Flows

The next algorithm bears a strong resemblance to the Edmonds-Karp algorithm, though it was developed independently and contemporaneously by Dinic. Unlike the Edmonds-Karp algorithm, Dinic's algorithm enjoys a modularity that lends itself to optimized algorithms with faster running times.

[8]The residual and layered networks change during these iterations, but F and $V_0, \ldots, V_c$ always refer to networks before the first of these iterations.

Dinic's Algorithm

initialize $f_e = 0$ for all $e \in E$
while there is an s-t path in the current residual network G_f **do**
  construct the layered network L_f from G_f using breadth-first search,
   as in the proof of Lemma 3.2
  `// takes` $O(|E|)$ `time`
  compute a blocking flow g (Definition 4.1) in L_f
  `// augment the flow` f `using the flow` g
  for all edges (v, w) of G for which the corresponding forward edge
   of G_f carries flow ($g_{vw} > 0$) **do**
    increase f_e by g_e
  for all edges (v, w) of G for which the corresponding reverse edge
   of G_f carries flow ($g_{wv} > 0$) **do**
    decrease f_e by g_e

Dinic's algorithm can only terminate with a residual network with no s-t path, that is, with a maximum flow (by Corollary 2.3). While in the Edmonds-Karp algorithm we only formed the layered network L_f in the analysis (in the proof of Lemma 3.2), Dinic's algorithm explicitly constructs this network in each iteration.

A blocking flow is, intuitively, a bunch of shortest augmenting paths that get processed as a batch. Somewhat more formally, blocking flows are precisely the possible outputs of the naive greedy algorithm discussed at the beginning of Lecture #1. Completely formally:

Definition 4.1 (Blocking Flow) A *blocking flow* g in a network G is a feasible flow such that, for every s-t path P of G, some edge e is saturated by g (i.e.,. $f_e = u_e$).

That is, a blocking flow zeroes out an edge of every s-t path.

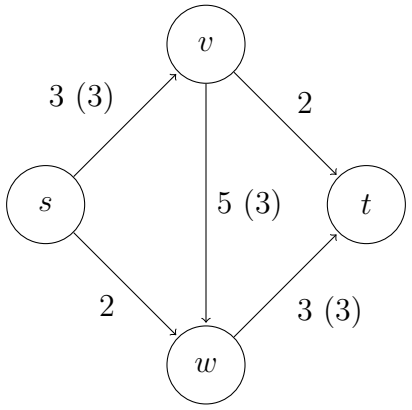

Figure 10: Example of blocking flow. This is not a maximum flow.

Recall from Lecture #1 that a blocking flow need not be a maximum flow; the blocking flow in Figure 10 has value 3, while the maximum flow value is 5. While the blocking flow in Figure 10 uses only one path, generally a blocking flow uses many paths. Indeed, every flow that is maximum (equivalently, no s-t paths in the residual network) is also a blocking flow (equivalently, no s-t paths in the residual network comprising only forward edges).

The running time analysis of Dinic's algorithm is anchored by the following progress lemma.

Lemma 4.2 (Dinic Progress Lemma) *Fix a network G. For a flow f, let $d(f)$ denote the number of hops in a shortest s-t path (with positive residual capacity) in G_f, or $+\infty$ if no such paths exist (or $+\infty$ if no such paths exist). If h is obtained from f be augmenting f by a blocking flow g in G_f, then $d(h) > d(f)$.*

That is, every iteration of Dinic's algorithm strictly increases the s-t distance in the current residual graph.

We leave the proof of Lemma 4.2 as Exercise #5; the proof uses the same ideas as that of Lemma 3.2. For an example, observe that after augmenting our running example by the blocking flow in Figure 10, we obtain the residual network in Figure 11. We had $d(f) = 2$ initially, and $d(f) = 3$ after the augmentation.

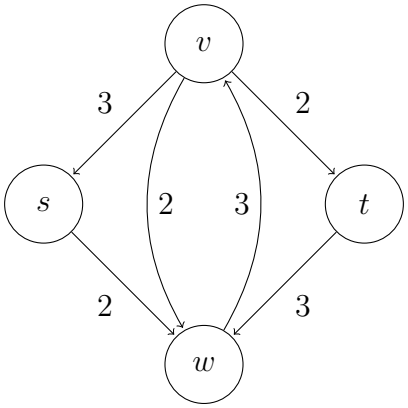

Figure 11: Residual network of blocking flow in Figure 10. $d(f) = 3$ in this residual graph.

Since $d(f)$ can only go up to $n - 1$ before becoming infinite (i.e., disconnecting s and t in G_f), Lemma 4.2 immediately implies that Dinic's algorithm terminates after at most n iterations. In this sense, the maximum flow problem reduces to n instances of the blocking flow problem (in layered networks). The running time of Dinic's algorithm is $O(n \cdot BF)$, where BF denotes the running time required to compute a blocking flow in a layered network.

The Edmonds-Karp algorithm and its proof effectively shows how to compute a blocking flow in $O(m^2)$ time, by repeatedly sending as much flow as possible on a single path of L_f with positive residual capacity. On Problem Set #1 you'll seen an algorithm, based on depth-first search, that computes a blocking flow in time $O(mn)$. With this subroutine, Dinic's

algorithm runs in $O(n^2m)$ time, improving over the Edmonds-Karp algorithm. (Remember, it's always a win to replace an m with an n.)

Using fancy data structures, it's known how to compute a maximum flow in near-linear time (with just one extra logarithmic factor), yielding a maximum flow algorithm with running time close to $O(mn)$. This running time is no longer so embarrassing, and resembles time bounds that you saw in CS161, for example for the Bellman-Ford shortest-path algorithm and for various all-pairs shortest paths algorithms.

5 Looking Ahead

Thus far, we focused on "augmenting path" maximum flow algorithms. Properly implemented, such algorithms are reasonably practical. Our motivation here is pedagogical: these algorithms remain the best way to develop your initial intuition about the maximum flow problem.

Next lecture introduces a different paradigm for computing maximum flows, known as the "push-relabel" framework. Such algorithms are reasonably simple, but somewhat less intuitive than augmenting path algorithms. Properly implemented, they are blazingly fast and are often the method of choice for solving the maximum flow problem in practice.

CS261: A Second Course in Algorithms
Lecture #20: The Maximum Cut Problem and Semidefinite Programming*

Tim Roughgarden†

March 10, 2016

1 Introduction

Now that you're finishing CS261, you're well equipped to comprehend a lot of advanced material on algorithms. This lecture illustrates this point by teaching you about a cool and famous approximation algorithm.

In the *maximum cut* problem, the input is an undirected graph $G = (V, E)$ with a nonnegative weight $w_e \geq 0$ for each edge $e \in E$. The goal is to compute a *cut* — a partition of the vertex set into sets A and B — that maximizes the total weight of the cut edges (the edges with one endpoint in each of A and B).

Now, if it were the *minimum* cut problem, we'd know what to do — that problem reduces to the maximum flow problem (Exercise Set #2). It's tempting to think that we can reduce the maximum cut problem to the minimum cut problem just by negating the weights of all of the edges. Such a reduction would yield a minimum cut problem with negative weights (or capacities). But if you look back at our polynomial-time algorithms for computing minimum cuts, you'll notice that we assumed nonnegative edge capacities, and that our proofs depended on this assumption. Indeed, it's not hard to prove that the maximum cut problem is NP-hard. So, let's talk about polynomial-time approximation algorithms.

It's easy to come up with a $\frac{1}{2}$-approximation algorithm for the maximum cut problem. Almost anything works — a greedy algorithm, local search, picking a random cut, linear programming rounding, and so on. But frustratingly, none of these techniques seemed capable of proving an approximation factor better than $\frac{1}{2}$. This made it remarkable when, in 1994, Goemans and Williamson showed how a new technique, "semidefinite programming rounding," could be used to blow away all previous approximation algorithms for the maximum cut problem.

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

2 A Semidefinite Programming Relaxation for the Maximum Cut Problem

2.1 A Quadratic Programming Formulation

To motivate a novel relaxation for the maximum cut problem, we first reformulate the problem exactly via a quadratic program. (So solving this program is also NP-hard.) The idea is to have one decision variable y_i for each vertex $i \in V$, indicating which side of the cut the vertex is on. It's convenient to restrict y_i to lie in $\{-1, +1\}$, as opposed to $\{0, 1\}$. There's no need for any other constraints. In the objective function, we want an edge (i, j) of the input graph $G = (V, E)$ to contribute w_{ij} whenever i, j are on different sides of the cut, and 0 if they are on the same side of the cut. Note that $y_i y_j = +1$ if i, j are on the same side of the cut and $y_i y_j = -1$ otherwise. Thus, we can formulate the maximum cut objective function exactly as

$$\max \sum_{(i,j) \in E} w_{ij} \cdot \frac{1}{2} (1 - y_i y_j).$$

Note that the contribution of edge (i, j) to the objective function is w_{ij} if i and j are on different sides of the cut and 0 otherwise, as desired. There is a one-to-one and objective-function-preserving correspondence between cuts of the input graph and feasible solutions to this quadratic program.

This quadratic programming formulation has two features that make it a non-linear program: the integer constraints $y_i \in \{\pm 1\}$ for every $i \in V$, and the quadratic terms $y_i y_j$ in the objective function.

2.2 A Vector Relaxation

Here's an inspired idea for a relaxation: rather than requiring each y_i to be either -1 or +1, we only ask that each decision variable is a *unit vector in* $\mathbb{R}^n$, where $n = |V|$ denotes the number of vertices. We henceforth use x_i to denote the (vector-valued) decision variable corresponding to the vertex $i \in V$. We can think of the values +1 and -1 as the special cases of the unit vectors $(1, 0, 0, \ldots, 0)$ and $(-1, 0, 0, \ldots, 0)$. There is an obvious question of what we mean by the quadratic term $y_i \dot{y}_j$ when we switch to decision variables that are n-vectors; the most natural answer is to replace the scalar product $y_i \cdot y_j$ by the inner product $\langle x_i, x_j \rangle$. We then have the following "vector programming relaxation" of the maximum cut problem:

$$\max \frac{1}{2} \sum_{(i,j) \in E} w_{ij} (1 - \langle x_i, x_j \rangle)$$

subject to

$$\|x_i\|_2^2 = 1 \qquad \text{for every } i \in V.$$

It may seem obscure to write $\|x_i\|_2^2 = 1$ rather than $\|x_i\|_2 = 1$ (which is equivalent); the reason for this will become clear later in the lecture. Since every cut of the input graph G

maps to a feasible solution of this relaxation with the same objective function value, and the vector program only maximizes over more stuff, we have

$$\text{vector } OPT \geq OPT.$$

Geometrically, this relaxation maps all the vertices of the input graph G to the unit sphere in $\mathbb{R}^n$, while attempting to map the endpoints of each edge to points that are as close to antipodal as possible (to get $\langle x_i, x_j \rangle$ as close to -1 as possible).

2.3 Disguised Convexity

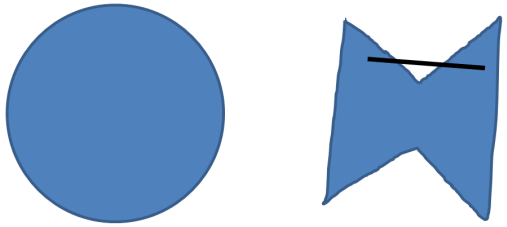

Figure 1: (a) a circle is convex, but (b) is not convex;the chord shown is not contained entirely in the set.

It turns out that the relaxation above can be solved to optimality in polynomial time.[1] You might well find this counterintuitive, given that the inner products in the objective function seem hopelessly quadratic. The moral reason for computational tractability is *convexity*. Indeed, a good rule of thumb very generally is to equate computational tractability with convexity. A mathematical program can be convex in two senses. The first sense is the same as that we discussed back in Lecture #9 — a subset of $\mathbb{R}^n$ is convex if it contains all of its chords. (See Figure 1.) Recall that the feasible region of a linear program is always convex in this sense. The second sense is that the objective function can be a convex function. (A linear function is a special case of a convex function.) We won't need this second type of convexity in this lecture, but it's extremely useful in other contexts, especially in machine learning.

OK... but where's the convexity in the vector relaxation above? After all, if you take the average of two points on the unit sphere, you don't get another point on the unit sphere.

We next expose the disguised convexity. A natural idea to remove the quadratic (inner product) character of the vector program above is to linearize it, meaning to introduce a new decision variable p_{ij} for each $i, j \in V$, with the intention that p_{ij} will take on the value $\langle x_i, x_j \rangle$. But without further constraints, this will lead to a relaxation of the relaxation —

[1] Strictly speaking, since the optimal solution might be irrational, we only solve it up to arbitrarily small error.

nothing is enforcing the p_{ij}'s to actually be of the form $\langle x_i, x_j \rangle$ for some collection $x_1, \ldots, x_n$ of n-vectors, and the p_{ij}'s could form an arbitrary matrix instead. So how can we enforce the intended semantics?

This is where elementary linear algebra comes to the rescue. We'll use some facts that you've almost surely seen in a previous course, and also have almost surely forgotten. That's OK — if you spend 20-30 minutes with your favorite linear algebra textbook (or Wikipedia), you'll remember why all of these relevant facts are true (none are difficult).

First, let's observe that a $V \times V$ matrix $P = \{p_{ij}\}$ is of the form $p_{ij} = \langle x_i, x_j \rangle$ for some vectors $x_1, \ldots, x_n$ (for every $i, j \in V$) if and only if we can write

$$P = X^T X \tag{1}$$

for some matrix $X \in \mathbb{R}^{V \times V}$. Recalling the definition of matrix multiplication, the (i, j) entry of $X^T X$ is the inner product of the ith row of X^T and the jth column of X, or equivalently the inner product of the ith and jth columns of X. Thus, for matrices P of the desired form, the columns of the matrix X provide the n-vectors whose inner products define all of the entries of P.

Matrices that are "squares" in the sense of (1) are extremely well understood, and they are called (symmetric) *positive semidefinite (psd)* matrices. There are many characterizations of symmetric psd matrices, and none are particularly hard to prove. For example, a symmetric matrix is psd if and only if all of its eigenvalues are nonnegative. (Recall that a symmetric matrix has a full set of real-valued eigenvalues.) The characterization that exposes the latent convexity in the vector program above is that a symmetric matrix P is psd if and only if

$$\underbrace{z^T P z}_{\text{"quadratic form"}} \geq 0 \tag{2}$$

for every vector $z \in \mathbb{R}^n$. Note that the forward direction is easy to see (if P can be written $P = X^T X$ then $z^T P z = (Xz)^T (Xz) = \|Xz\|_2^2 \geq 0$); the (contrapositive of the) reverse direction follows easily from the eigenvalue characterization already mentioned.

For a fixed vector $z \in \mathbb{R}^n$, the inequality (2) reads

$$\sum_{i,j \in V} p_{ij} z_i z_j \geq 0,$$

which is *linear* in the p_{ij}'s (for fixed z_i's). And remember that the p_{ij}'s are our decision variables!

2.4 A Semidefinite Relaxation

Summarizing the discussion so far, we've argued that the vector relaxation in Section 2.2 is equivalent to the linear program

$$\max \frac{1}{2} \sum_{(i,j) \in E} w_{ij} \left(1 - p_{ij}\right)$$

subject to

$$\sum_{i,j \in V} p_{ij} z_i z_j \geq 0 \qquad \text{for every } z \in \mathbb{R}^n \tag{3}$$

$$p_{ij} = p_{ji} \qquad \text{for every } i, j \in V \tag{4}$$

$$p_{ii} = 1 \qquad \text{for every } i \in V. \tag{5}$$

The constraints (3) and (4) enforce the p.s.d. and symmetry constraints on the p_{ij}'s. Their presence makes this program a *semidefinite program (SDP)*. The final constraints (5) correspond to the constraints that $\|x_i\|_2^2 = 1$ for every $i \in V$ — that the matrix formed by the p_{ij}'s not only has the form $X^T X$, but has this form for a matrix X whose columns are unit vectors.

2.5 Solving SDPs Efficiently

The good news about the SDP above is that every constraint is linear in the p_{ij}'s, so we're in the familiar realm of linear programming. The obvious issue is that the linear program has an infinite number of constraints of the form (3) — one for each real-valued vector $z \in \mathbb{R}^n$. So there's no hope of even writing this SDP down. But wait, didn't we discuss an algorithm for linear programming that can solve linear programs efficiently even when there are too many constraints to write down?

The first way around the infinite number of constraints is to use the ellipsoid method (Lecture #10) to solve the SDP. Recall that the ellipsoid method runs in time polynomial in the number of variables (n^2 variables in our case), provided that there is a polynomial-time *separation oracle* for the constraints. The responsibility of a separation oracle is, given an allegedly feasible solution, to either verify feasibility or else produce a violated constraint. For the SDP above, the constraints (4) and (5) can be checked directly. The constraints (3) can be checked by computing the eigenvalues and eigenvectors of the matrix formed by the p_{ij}'s.[2] As mentioned earlier, the constraints (3) are equivalent to this matrix having only nonnegative eigenvalues. Moreover, if the p_{ij}'s are not feasible and there is a negative eigenvalue, then the corresponding eigenvector serves as a vector z such that the constraint (3) is violated.[3] This separation oracle allows us to solve SDPs using the ellipsoid method.

The second solution is to use "interior-point methods," which were also mentioned briefly at the end of Lecture #10. State-of-the-art interior-point algorithms can solve SDPs both in theory (meaning in polynomial time) and in practice, meaning for medium-sized problems. SDPs are definitely harder in practice than linear programs, though — modern solvers have trouble going beyond thousands of variables and constraints, which is a couple orders of magnitude smaller than the linear programs that are routinely solved by commercial solvers.

[2] There are standard and polynomial-time matrix algorithms for this task; see any textbook on numerical analysis.

[3] If z is an eigenvector of a symmetric matrix P with eigenvalue λ, then $z^T P z = z^T(\lambda z) = \lambda \cdot \|z\|_2^2$, which is negative if and only if λ is negative.

A third option for many SDPs is to use an extension of the multiplicative weights algorithm (Lecture #11) to quickly compute an approximately optimal solution. This is similar in spirit to but somewhat more complicated than the application to approximate maximum flows discussed in Lecture #12.[4]

Henceforth, we'll just take it on faith that our SDP relaxation can be solved in polynomial time. But the question remains: what do we do with the solution to the relaxation?

3 Randomized Hyperplane Rounding

The SDP relaxation above of the maximum cut problem was already known in the 1980s. But only in 1994 did Goemans and Williamson figure out how to round its solution to a near-optimal cut. First, it's natural to round the solution of the vector programming relaxation (Section 2.2) rather than the equivalent SDP relaxation (Section 2.4), since the former ascribes one object (a vector) to each vertex $i \in V$, while the latter uses one scalar for each pair of vertices.[5] Thus, we "just" need to round each vector to a binary value, while approximately preserving the objective function value.

The first key idea is to use randomized rounding, as first discussed in Lecture #18. The second key idea is that a simple way to round a vector to a binary value is to look at which side of some hyperplane it lies on (cf., the machine learning examples in Lectures #7 and #12). See Figure 2. Combining these two ideas, we arrive at randomized hyperplane rounding.

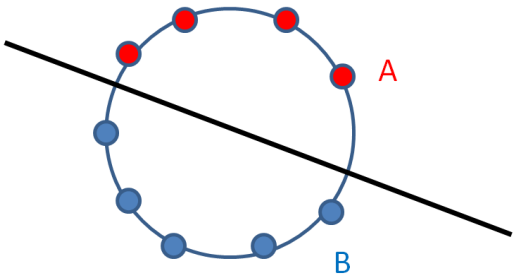

Figure 2: Randomized hyperplane rounding: points with positive dot product in set A, points with negative dot product in set B.

[4]Strictly speaking, the first two solutions also only compute an approximately optimal solution. This is necessary, because the optimal solution to an SDP (with all integer coefficients) might be irrational. (This can't happen with a linear program.) For a given approximation ϵ, the running time of the ellipsoid method and interior-point methods depend on $\log \frac{1}{\epsilon}$, while that of multiplicative weights depends inverse polynomially on $\frac{1}{\epsilon}$.

[5]After solving the SDP relaxation to get the matrix P of the p_{ij}'s, another standard matrix algorithm ("Cholesky decomposition") can be used to efficiently recover the matrix X in the equation $P = X^T X$ and hence the vectors (which are the columns of X).

Randomized Hyperplane Rounding

given: one vector x_i for each $i \in V$
choose a random unit vector $r \in \mathbb{R}^n$
set $A = \{i \in V : \langle x_i, r\rangle \geq 0\}$
set $B = \{i \in V : \langle x_i, r\rangle < 0\}$
return the cut (A, B)

Thus, vertices are partitioned according to which side of the hyperplane with normal vector r they lie on. You may be wondering how to choose a random unit vector in $\mathbb{R}^n$ in an algorithm. One simple way is: sample n independent standard Gaussian random variables (with mean 0 and variance 1) $g_1, \ldots, g_n$, and normalize to get a unit vector:

$$r = \frac{(g_1, \ldots, g_n)}{\|(g_1, \ldots, g_n)\|}.$$

(Or, note that the computed cut doesn't change if we don't bother to normalize.) The main property we need of the distribution of r is spherical symmetry — that all vectors at a given distance from the origin are equally likely.

We have the following remarkable theorem.

Theorem 3.1 *The expected weight of the cut produced by randomized hyperplane rounding is at least .878 times the maximum possible.*

The theorem follows easily from the following lemma.

Lemma 3.2 *For every edge $(i, j) \in E$ of the input graph,*

$$\mathbf{Pr}[(i,j) \text{ is cut}] \geq .878 \cdot \underbrace{\left[\frac{1}{2}(1 - \langle x_i, x_j\rangle)\right]}_{\text{contribution to SDP}}.$$

Proof of Theorem 3.1: We can derive

$$\begin{aligned}
\mathbf{E}[\text{weight of } (A,B)] &= \sum_{(i,j)\in E} w_{ij} \cdot \mathbf{Pr}[(i,j) \text{ is cut}] \\
&\geq .878 \cdot \sum_{(i,j)\in E} \left[\frac{1}{2}(1 - \langle x_i, x_j\rangle)\right] \\
&\geq .878 \cdot OPT,
\end{aligned}$$

where the equation follows from linearity of expectation (using one indicator random variable per edge), the first inequality from Lemma 3.2, and the second inequality from the fact that the x_i's are an optimal solution to vector programming relaxation of the maximum cut problem. ■

We conclude by proving the key lemma.

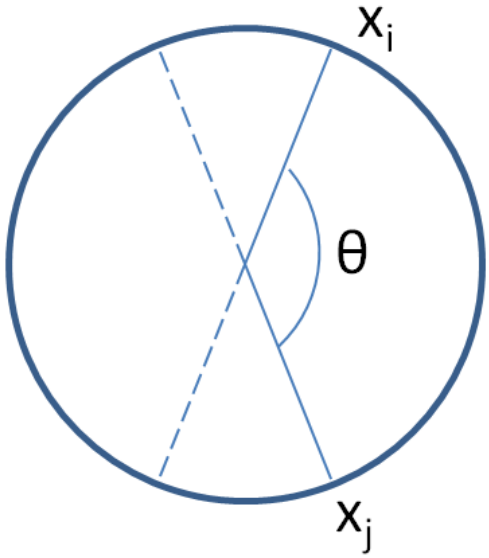

Figure 3: x_i and x_j are placed on different sides of the cut with probability θ/π.

Proof of Lemma 3.2: Fix an edge $(i, j) \in E$. Consider the two-dimensional subspace (through the origin) spanned by the vectors x_i and x_j. Since r was chosen from a spherically symmetric distribution, its projection onto this subspace is also spherically symmetric — it's equally likely to point in any direction. The vertices x_i and x_j are placed on different sides of the cut if and only if they are "split" by the projection of r. (Figure 3.) If we let θ denote the angle between x_i and x_j in this subspace, then 2θ out of the 2π radians of possible directions result in the edge (i, j) getting cut. So we know the cutting probability, as a function of θ:

$$\mathbf{Pr}[(i, j) \text{ is cut}] = \frac{\theta}{\pi}.$$

We still need to understand $\frac{1}{2}(1 - \langle x_i, x_j \rangle)$ as a function of θ. But remember from pre-calculus that $\langle x_i, x_j \rangle = \|x_i\|\|x_j\| \cos\theta$. And since x_i and x_j are both unit vectors (in the original space and also the subspace that they span), we have

$$\frac{1}{2}(1 - \langle x_i, x_j \rangle) = \tfrac{1}{2}(1 - \cos\theta).$$

The lemma thus boils down to verifying that

$$\frac{\theta}{\pi} \geq .878 \cdot \left[\tfrac{1}{2}(1 - \cos\theta)\right]$$

for all possible values of $\theta \in [0, \pi]$. This inequality is easily seen by plotting both sides, or if you're a stickler for rigor, by computations familiar from first-year calculus. ■

4 Going Beyond .878

For several lectures we were haunted by the number $1 - \frac{1}{e}$, which seemed like a pretty weird number. Even more bizarrely, it is provably the best-possible approximation guarantee for several natural problems, including online bipartite matching (Lecture #14) and, assuming $P \neq NP$, set coverage (Lecture #15).

Now the .878 in this lecture seems like a *really* weird number. But there is some evidence that it might be optimal! Specifically, in 2005 it was proved that, assuming that the "Unique Games Conjecture (UGC)" is true (and $P \neq NP$), there is no polynomial-time algorithm for the maximum cut problem with approximation factor larger than the one proved by Goemans and Williamson. The UGC (which is only from 2002) is somewhat technical to state precisely — it asserts that a certain constraint satisfaction problem is NP-hard. Unlike the $P \neq NP$ conjecture, which is widely believed, it is highly unclear whether the UGC is true or false. But it's amazing that *any* plausible complexity hypothesis implies the optimality of randomized hyperplane rounding for the maximum cut problem.

CS261: A Second Course in Algorithms
Lecture #3: The Push-Relabel Algorithm for Maximum Flow*

Tim Roughgarden†

January 12, 2016

1 Motivation

The maximum flow algorithms that we've studied so far are *augmenting path* algorithms, meaning that they maintain a flow and augment it each iteration to increase its value. In Lecture #1 we studied the Ford-Fulkerson algorithm, which augments along an arbitrary s-t path of the residual networks, and only runs in pseudopolynomial time. In Lecture #2 we studied the Edmonds-Karp specialization of the Ford-Fulkerson algorithm, where in each iteration a shortest s-t path in the residual network is chosen for augmentation. We proved a running time bound of $O(m^2n)$ for this algorithm (as always, $m = |E|$ and $n = |V|$). Lecture #2 and Problem Set #1 discuss Dinic's algorithm, where each iteration augments the current flow by a blocking flow in a layered subgraph of the residual network. In Problem Set #1 you will prove a running time bound of $O(n^2m)$ for this algorithm.

In the mid-1980s, a new approach to the maximum flow problem was developed. It is known as the "push-relabel" paradigm. To this day, push-relabel algorithms are often the method of choice in practice (even if they've never quite been the champion for the best worst-case asymptotic running time).

To motivate the push-relabel approach, consider the network in Figure 1, where k is a large number (like 100,000). Observe the maximum flow value is k. The Ford-Fulkerson and Edmonds-Karp algorithms run in $\Omega(k^2)$ time in this network. Moreover, much of the work feels wasted: each iteration, the long path of high-capacity edges has to be re-explored, even though it hasn't changed from the previous iteration. In this network, we'd rather route k units of flow from s to x (in $O(k)$ time), and then distribute this flow across the k paths from

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

x to t (in $O(k)$ time, linear-time overall). This is the idea behind push-relabel algorithms.[1] Of course, if there were strictly less than k paths from x to t, then not all of the k units of flow can be routed from x to t, and the remainder must be resent to the source. What is a principled way to organize such a procedure in an arbitrary network?

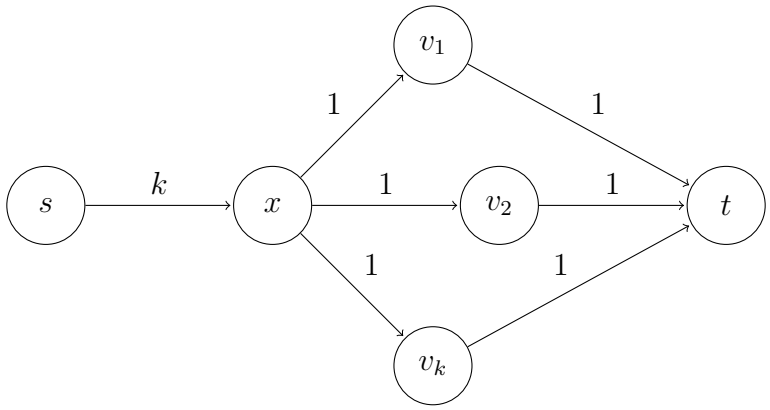

Figure 1: The edge $\{s, x\}$ has a large capacity k, and there are k paths from x to t via k different vertices v_i for $1 \leq i \leq k$ (3 are drawn for illustrative purposes). Both Ford-Fulkerson and Edmonds-Karp take $\Omega(k^2)$ time, but ideally we only need $O(k)$ time if we can somehow push k units of flow from s to x in one step.

2 Preliminaries

The first order of business is to relax the conservation constraints. For example, in Figure 1, if we've routed k units of flow to x but not yet distributed over the paths to t, then the vertex x has k units of flow incoming and zero units outgoing.

Definition 2.1 (Preflow) A *preflow* is a nonnegative vector $\{f_e\}_{e \in E}$ that satisfies two constraints:

Capacity constraints: $f_e \leq u_e$ for every edge $e \in E$;

Relaxed conservation constraints: for every vertex v other than s,

amount of flow entering $v \geq$ amount of flow exiting v.

The left-hand side is the sum of the f_e's over the edge incoming to v; likewise with the outgoing edges for the right-hand side.

[1] The push-relabel framework is not the unique way to address this issue. For example, fancy data structures ("dynamic trees" and their ilk) can be used to remember the work performed by previous searches and obtain faster running times.

The definition of a preflow is exactly the same as a flow (Lecture #1), except that the conservation constraints have been relaxed so that the amount of flow into a vertex is allowed to exceed the amount of flow out of the vertex.

We define the residual graph G_f with respect to a preflow f exactly as we did for the case of a flow f. That is, for an edge e that carries flow f_e and capacity u_e, G_f includes a forward version of e with residual capacity $u_e - f_e$ and a reverse version of e with residual capacity f_e. Edges with zero residual capacity are omitted from G_f.

Push-relabel algorithms work with preflows throughout their execution, but at the end of the day they need to terminate with an actual flow. This motivates a measure of the "degree of violation" of the conservation constraints.

Definition 2.2 (Excess) For a flow f and a vertex $v \neq s, t$ of a network, the *excess* $\alpha_f(v)$ is

$$\text{amount of flow entering } v - \text{amount of flow exiting } v.$$

For a preflow flow f, all excesses are nonnegative. A preflow is a flow if and only if the excess of every vertex $v \neq s, t$ is zero. Thus transforming a preflow to recover feasibility involves reducing and eventually eliminating all excesses.

3 The Push Subroutine

How do we augment a preflow? When we were restricting attention to flows only, our hands were tied — to maintain the conservation constraints, we only augmented along an s-t (or, for a blocking flow, a collection of such paths). With the relaxed conservation constraints, we have much more flexibility. All we need to is to augment a flow along a single edge at a time, routing flow from one of its endpoints to the other.

Push(v)

choose an outgoing edge (v, w) of v in G_f (if any)
`// push as much flow as possible`
let $\Delta = \min\{\alpha_f(v), \text{resid. cap. of } (v, w)\}$
push Δ units of flow along (v, w)

The point of the second step is to send as much flow as possible from v to w using the edge (v, w) of G_f, subject to the two constraints that define a preflow. There are two possible bottlenecks. One is the residual capacity of the edge (v, w) (as dictated by nonnegativity/capacity constraints); if this binds, then the push is called *saturating*. The other is the amount of excess at the vertex v (as dictated by the relaxed conservation constraints); if this binds, the push is *non-saturating*. In the final step, the preflow is updated as in our augmenting path algorithms: if (v, w) the forward version of edge $e = (v, w)$ in G, then f_e is increased by Δ; if (v, w) the reverse version of edge $e = (w, v)$ in G, then f_e is decreased by Δ. As always, the residual network is then updated accordingly. Note that after pushing flow from v to w, w has positive excess (if it didn't already).

4 Heights and Invariants

Just pushing flow around the residual network is not enough to obtain a correct maximum flow algorithm. One worry is illustrated by the graph in Figure 2 — after initially pushing one unit flow from s to v, how do we avoid just pushing the excess around the cycle $v \to w \to x \to y \to v$ forevermore. Obviously we want to push the excess to t when it gets to x, but how can we be systematic about it?

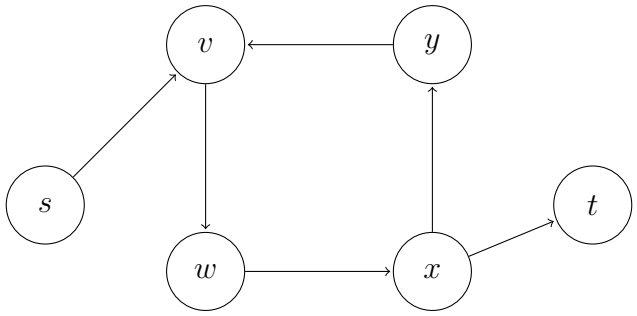

Figure 2: When we push flows in the above graph, how do we ensure that we do not push flows in the cycle $v \to w \to x \to y \to v$?

The next key idea will ensure termination of our algorithm, and will also implies correctness as termination. The idea is to maintain a *height* $h(v)$ for each vertex v of G. Heights will always be nonnegative integers. You are encouraged to visualize a network in 3D, with the height of a vertex giving it its z-coordinate, with edges going "uphill" and "downhill," or possibly staying flat. The plan for the algorithm is to always maintain three invariants (two trivial and one non-trivial):

Invariants

1. $h(s) = n$ at all times (where $n = |V|$);
2. $h(t) = 0$;
3. for every edge (v, w) of the current residual network (with positive residual capacity), $h(v) \leq h(w) + 1$.

Visually, the third invariant says that edges of the residual network are only to go downhill gradually (by one per hop). For example, if a vertex v has three outgoing edges (v, w_1), (v, w_2), and (v, w_3), with $h(w_1) = 3$, $h(w_2) = 4$, and $h(w_3) = 6$, then the third invariant requires that $h(v)$ be 4 or less (Figure 3). Note that edges are allowed to go uphill, stay flat, or go downhill (gradually).

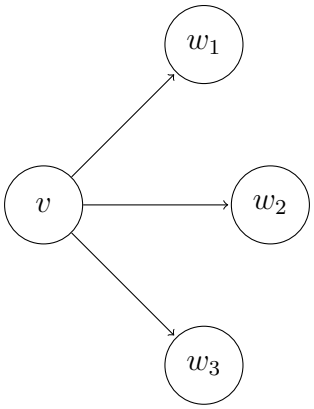

Figure 3: Given that $h(w_1) = 3, h(w_2) = 4, h(w_3) = 6$, it must be that $h(v) \leq 4$.

Where did these invariants come from? For one motivation, recall from Lecture #2 our optimality conditions for the maximum flow problem: a flow is maximum if and only if there is no s-t path (with positive residual capacity) in its residual graph. So clearly we want this property at termination. The new idea is to satisfy the optimality conditions *at all times*, and this is what the invariants guarantee. Indeed, since the invariants imply that s is at height n, t is at height 0, and each edge of the residual graph only goes downhill by at most 1, there can be no s-t path with at most $n-1$ edges (and hence no s-t path at all). It follows that if we find a preflow that is feasible (i.e., is actually a flow, with no excesses) and the invariants hold (for suitable heights), then the flow must be a maximum flow.

It is illuminating to compare and contrast the high-level strategies of augmenting path algorithms and of push-relabel algorithms.

Augmenting Path Strategy
Invariant: maintain a feasible flow.
Work toward: disconnecting s and t in the current residual network.

Push-Relabel Strategy
Invariant: maintain that s, t disconnected in the current residual network.
Work toward: feasibility (i.e., conservation constraints).

While there is a clear symmetry between the two approaches, most people find it less intuitive to relax feasibility and only restore it at the end of the algorithm. This is probably why the push-relabel framework only came along in the 1980s, while the augmenting path algorithms we studied date from the 1950s-1970s. The idea of relaxing feasibility is useful for many different problems.

In both cases, algorithm design is guided by an explicitly articulated strategy for guaranteeing correctness. The maximum flow problem, while polynomial-time solvable (as we know), is complex enough that solutions require significant discipline. Contrast this with, for example, the minimum spanning tree algorithms, where it's easy to come up with correct algorithms (like Kruskal or Prim) without any advance understanding of *why* they are correct.

5 The Algorithm

The high-level strategy of the algorithm is to maintain the three invariants above while trying to zero out any remaining excesses. Let's begin with the initialization. Since the invariants reference both a correct preflow and current vertex heights, we need to initialize both. Let's start with the heights. Clearly we set $h(s) = n$ and $h(t) = 0$. The first non-trivial decision is to set $h(v) = 0$ also for all $v \neq s, t$. Moving onto the initial preflow, the obvious idea is to start with the zero flow. *But this violates the third invariant:* edges going out of s would travel from height n to 0, while edges of the residual graph are supposed to only go downhill by 1. With the current choice of height function, no edges out of s can appear (with non-zero capacity) in the residual network. So the obvious fix is to initially saturate all such edges.

Initialization

set $h(s) = n$
set $h(v) = 0$ for all $v \neq s$
set $f_e = u_e$ for all edges e outgoing from s
set $f_e = 0$ for all other edges

All three invariants hold after the initialization (the only possible violation is the edges out of s, which don't appear in the initial residual network). Also, f is initialized to a preflow (with flow in $\geq$ flow out except at s).

Next, we restrict the Push operation from Section 3 so that it maintains the invariants. The restriction is that *flow is only allowed to be pushed downhill in the residual network.*

Push(v) [revised]

choose an outgoing edge (v, w) of v in G_f with $h(v) = h(w) + 1$ (if any)
`// push as much flow as possible`
let $\Delta = \min\{\alpha_f(v), \text{resid. cap. of } (v, w)\}$
push Δ units of flow along (v, w)

Here's the main loop of the push-relabel algorithm:

Main Loop

while there is a vertex $v \neq s, t$ with $\alpha_f(v) > 0$ **do**
    choose such a vertex v with the maximum height $h(v)$
    `// break ties arbitrarily`
    if there is an outgoing edge (v, w) of v in G_f with $h(v) = h(w) + 1$
    then
        Push(v)
    else
        increment $h(v)`// called a ‘‘relabel’’`

Every iteration, among all vertices that have positive excess, the algorithm processes the highest one. When such a vertex v is chosen, there may or may not be a downhill edge emanating from v (see Figure 4(a) vs. Figure 4(b)). Push(v) is only invoked if there is such an edge (in which case Push will push flow on it), otherwise the vertex is "relabeled," meaning its height is increased by one.

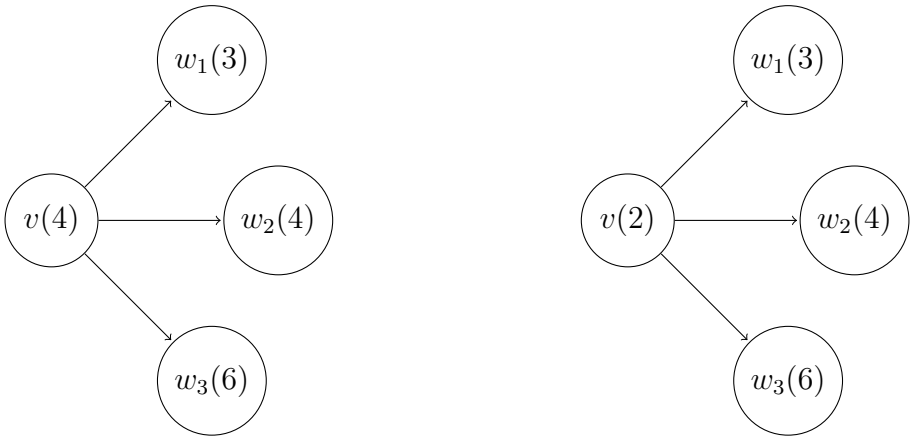

Figure 4: (a) $v \to w_1$ is downhill edge (4 to 3) (b) there are no downhill edges

Lemma 5.1 (Invariants Are Maintained) *The three invariants are maintained throughout the execution of the algorithm.*

Neither s not t ever get relabeled, so the first two invariants are always satisfied. For the third invariant, we consider separately a relabel (which changes the height function but not the preflow) and a push (which changes the preflow but not the height function). The only worry with a relabel at v is that, afterwards, some outgoing edge of v on the residual network goes downhill by more than one step. But the precondition for relabeling is that all outgoing edges are either flat or uphill, so this never happens. The only worry with a push from v to w is that it could introduce a new edge (w, v) to the residual network that might

go downhill by more than one step. But we only push flow downward, so a newly created reverse edge can only go upward.

The claim implies that if the push-relabel algorithm ever terminates, then it does so with a maximum flow. The invariants imply the maximum flow optimality conditions (no s-t path in the residual network), while the termination condition implies that the final preflow f is in fact a feasible flow.

6 Example

Before proceeding to the running time analysis, let's go through an example in detail to make sure that the algorithm makes sense. The initial network is shown in Figure 5(a). After the initialization (of both the height function and the preflow) we obtain the residual network in Figure 5(b). (Edges are labeled with their residual capacities, vertices with both their heights and their excesses.) [2]

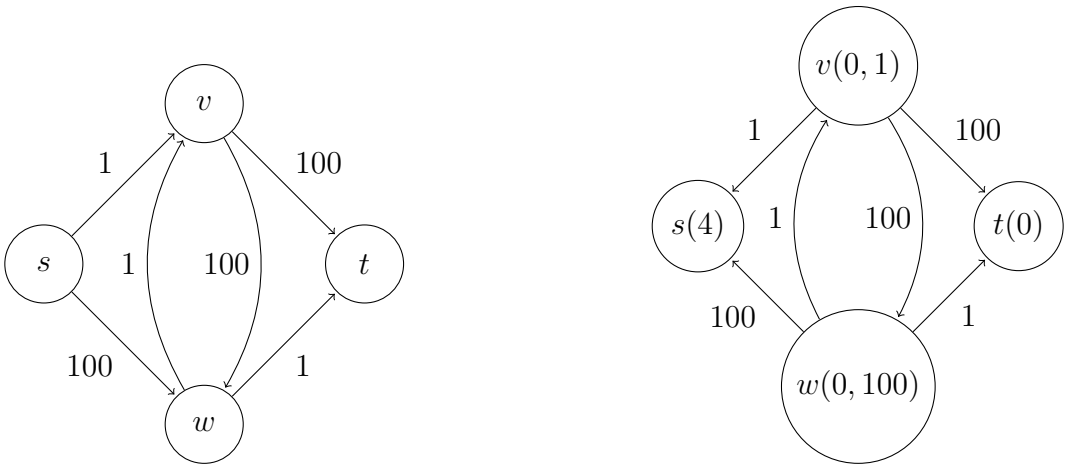

Figure 5: (a) Example network (b) Network after initialization. For v and w, the pair (a, b) denotes that the vertex has height a and excess b. Note that we ignore excess of s and t, so s and t both only have a single number denoting height.

In the first iteration of the main loop, there are two vertices with positive excess (v and w), both with height 0, and the algorithm can choose arbitrarily which one to process. Let's process v. Since v currently has height 0, it certainly doesn't have any outgoing edges in the residual network that go down. So, we relabel v, and its height increases to 1. In the second iteration of the algorithm, there is no choice about which vertex to process: v is now the unique highest label with excess, so it is chosen again. Now v *does* have downhill outgoing

[2] We looked at this network last lecture and determined that the maximum flow value is 3. So we should be skeptical of the 100 units of flow currently on edge (s, w); it will have to return home to roost at some point.

edges, namely (v, w) and (v, t). The algorithm is allowed to choose arbitrarily between such edges. You're probably rooting for the algorithm to push v's excess straight to t, but to keep things interesting let's assume that that the algorithm pushes it to w instead. This is a non-saturating push, and the excess at v drops to zero. The excess at w increases from 100 to 101. The new residual network is shown in Figure 6.

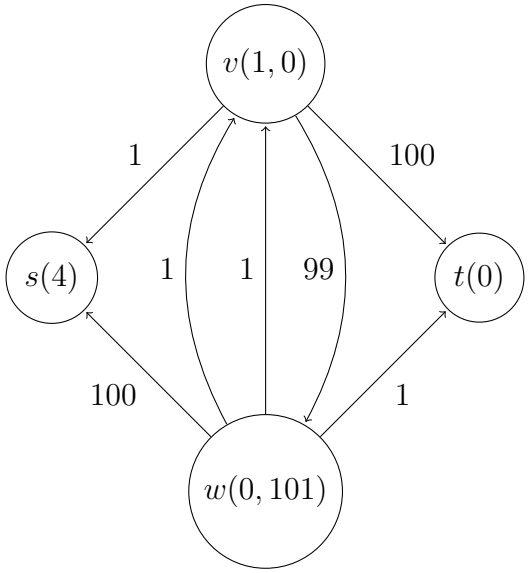

Figure 6: Residual network after non-saturating push from v to w.

In the next iteration, w is the only vertex with positive excess so it is chosen for processing. It has no outgoing downhill edges, so it get relabeled (so now $h(w) = 1$). Now w does have a downhill outgoing edge (w, t). The algorithm pushes one unit of flow on (w, t) — a saturating push —- the excess at w goes back down to 100. Next iteration, w still has excess but has no downhill edges in the new residual network, so it gets relabeled. With its new height of 2, in the next iteration the edges from w to v go downhill. After pushing two units of flow from w to v — one on the original (w, v) edge and one on the reverse edge corresponding to (v, w) — the excess at w drops to 98, and v now again has an excess (of 2). The new residual network is shown in Figure 7.

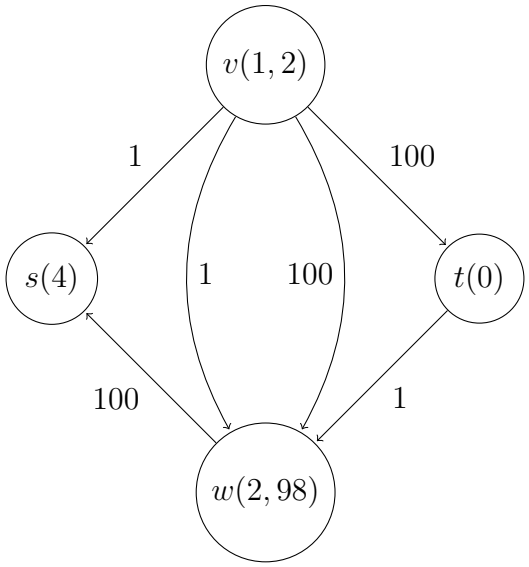

Figure 7: Residual network after non-saturating push from v to w.

Of the two vertices with excess, w is higher. It again has no downhill edges, however, so the algorithm relabels it three times in a row until it does. When its height reaches 5, the reverse edge (v, s) goes downhill, the algorithm pushes w's entire excess to s. Now v is the only vertex remaining with excess. Its edge (v, t) goes down hill, and after pushing two units of flow on it the algorithm halts with a maximum flow (with value 3).

7 The Analysis

7.1 Formal Statement and Discussion

Verifying that the push-relabel algorithm computes a maximum flow in one particular network is all fine and good, but it's not at all clear that it is correct (or even terminates) in general. Happily, the following theorem holds.[3]

Theorem 7.1 *The push-relabel algorithm terminates after $O(n^2)$ relabel operations and $O(n^3)$ push operations.*

The hidden constants in Theorem 7.1 are at most 2. Properly implemented, the push-relabel algorithm has running time $O(n^3)$; we leave the details to Exercise Set #2. The one point that requires some thought is to maintain suitable data structures so that a highest vertex with excess can be identified in $O(1)$ time.[4] In practice, the algorithm tends to run in sub-quadratic time.

[3] A sharper analysis yields the better bound of $O(n^2\sqrt{m})$; see Problem Set #1. Believe it or now, the worst-case running time of the algorithm is in fact $\Omega(n^2\sqrt{m})$.

[4] Or rather, $O(1)$ "amortized" time, meaning in total time $O(n^3)$ over all of the $O(n^3)$ iterations.

The proof of Theorem 7.1 is more indirect then our running time analyses of augmenting path algorithms. In the latter algorithms, there are clear progress measures that we can use (like the difference between the current and maximum flow values, or the distance between s and t in the current residual network). For push-relabel, we require less intuitive progress measures.

7.2 Bounding the Relabels

The analysis begins with the following key lemma, proved at the end of the lecture.

Lemma 7.2 (Key Lemma) *If the vertex v has positive excess in the preflow f, then there is a path $v \rightsquigarrow s$ in the residual network G_f.*

The intuition behind the lemma is that, since the excess for to v somehow from v, it should be possible to "undo" this flow in the residual network.

For the rest of this section, we assume that Lemma 7.2 is true and use it to prove Theorem 7.1. The lemma has some immediate corollaries.

Corollary 7.3 (Height Bound) *In the push-relabel algorithm, every vertex always has height at most $2n$.*

Proof: A vertex v is only relabeled when it has excess. Lemma 7.2 implies that, at this point, there is a path from v to s in the current residual network G_f. There is therefore such a path with at most $n-1$ edges (more edges would create a cycle, which can be removed to obtain a shorter path). By the first invariant (Section 4), the height of s is always n. By the third invariant, edges of G_f can only go downhill by one step. So traversing the path from v to s decreases the height by at most $n-1$, and winds up at height n. Thus v has height $2n-1$ or less, and at most one more than this after it is relabeled for the final time. ■

The bound in Theorem 7.1 on the number of relabels follows immediately.

Corollary 7.4 (Relabel Bound) *The push-relabel algorithm performs $O(n^2)$ relabels.*

7.3 Bounding the Saturating Pushes

We now bound the number of pushes. We piggyback on Corollary 7.4 by using the number of relabels as a progress measure. We'll show that lots of pushes happen only when there are already lots of relabels, and then apply our upper bound on the number of relabels.

We handle the cases of saturating pushes (which saturate the edge) and non-saturating pushes (which exhaust a vertex's excess) separately.[5] For saturating pushes, think about a particular edge (v, w). What has to happen for this edge to suffer two saturating pushes in the same direction?

[5]To be concrete, in case of a tie let's call it a non-saturating push.

Lemma 7.5 (Saturating Pushes) *Between two saturating pushes on the same edge (v, w) in the same direction, each of v, w is relabeled at least twice.*

Since each vertex is relabeled $O(n)$ times (Corollary 7.3), each edge (v, w) can only suffer $O(n)$ saturating pushes. This yields a bound of $O(mn)$ on the number of saturating pushes. Since $m = O(n^2)$, this is even better than the bound of $O(n^3)$ that we're shooting for.[6]

Proof of Lemma 7.5: Suppose there is a saturating push on the edge (v, w). Since the push-relabel algorithm only pushes downhill, v is higher than w ($h(v) = h(w) + 1$). Because the push saturates (v, w), the edge drops out of the residual network. Clearly, a prerequisite for another saturating push on (v, w) is for (v, w) to reappear in the residual network. The only way this can happen is via a push in the opposite direction (on (w, v)). For this to occur, w must first reach a height larger than that of v (i.e., $h(w) > h(v)$), which requires w to be relabeled at least twice. After (v, w) has reappeared in the residual network (with $h(v) < h(w)$), no flow will be pushed on it until v is again higher than w. This requires at least two relabels to v. ■

7.4 Bounding the Non-Saturating Pushes

We now proceed to the non-saturating pushes. Note that nothing we've said so far relies on our greedy criterion for the vertex to process in each iteration (the highest vertex with excess). This feature of the algorithm plays an important role in this final step.

Lemma 7.6 (Non-Saturating Pushes) *Between any two relabel operations, there are at most n non-saturating pushes.*

Corollary 7.4 and Lemma 7.6 immediately imply a bound of $O(n^3)$ on the number of non-saturating pushes, which completes the proof of Theorem 7.1 (modulo the key lemma).

Proof of lemma 7.6: Think about the entire sequence of operations performed by the algorithm. "Zoom in" to an interval bracketed by two relabel operations (possibly of different vertices), with no relabels in between. Call such an interval a *phase* of the algorithm. See Figure 8.

[6] We're assuming that the input network has no parallel edges, between the same pair of vertices and in the same direction. This is effectively without loss of generality — multiple edges in the same direction can be replaced by a single one with capacity equal to the sum of the capacities of the parallel edges.

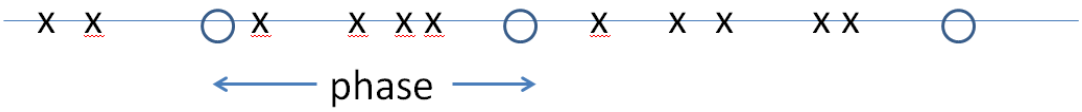

Figure 8: A timeline showing all operations ('O' represents relabels, 'X' represents non-saturating pushes). An interval between two relabels ('O's) is called a phase. There are $O(n^2)$ phases, and each phase contains at most n non-saturating pushes.

How does a non-saturating push at a vertex v make progress? By zeroing out the excess at v. Intuitively, we'd like to use the number of zero-excess vertices as a progress measure within a phase. But a non-saturating push can create a new excess elsewhere. To argue that this can't go on for ever, we use that *excess is only transferred from higher vertices to lower vertices*.

Formally, by the choice of v, as the highest vertex with excess, we have

$$h(v) \geq h(w) \qquad \text{for all vertices } w \text{ with excess} \tag{1}$$

at the time of a non-saturating push at v. Inequality (1) continues to hold as long as there is no relabel: pushes only send flow downhill, so can only transfer excess from higher vertices to lower vertices.

After the non-saturating push at v, its excess is zero. How can it become positive again in the future?[7] It would have to receive flow from a higher vertex (with excess). This cannot happen as long as (1) holds, and so can't happen until there's a relabel. We conclude that, within a phase, there cannot be two non-saturating pushes at the same vertex v. The lemma follows. ■

7.5 Analysis Recap

The proof of Theorem 7.1 has several cleverly arranged steps.

1. Each vertex can only be relabeled $O(n)$ times (Corollary 7.3 via Lemma 7.2), for a total of $O(n^2)$ relabels.

2. Each edge can only suffer $O(n)$ saturating pushes (only 1 between each time both endpoints are relabeled twice, by Lemma 7.5)), for a total of $O(mn)$ saturating pushes.

3. Each vertex can only suffer $O(n^2)$ non-saturating pushes (only 1 per phase, by Lemma 7.6), for a total of $O(n^3)$ such pushes.

[7] For example, recall what happened to the vertex v in the example in Section 6.

8 Proof of Key Lemma

We now prove Lemma 7.2, that there is a path from every vertex with excess back to the source s in the residual network. Recall the intuition: excess got to v from s somehow, and the reverse edges should form a breadcrumb trail back to s.

Proof of Lemma 7.2: Fix a preflow f.[8] Define

$$A = \{v \in V \,:\, \text{there is an } s \rightsquigarrow v \text{ path } P \text{ in } G \text{ with } f_e > 0 \text{ for all } e \in P\}.$$

Conceptually, run your favorite graph search algorithm, starting from s, in the subgraph of G consisting of the edges that carry positive flow. A is where you get stuck. (This is the second example we've seen of the "reachable vertices" proof trick; there are many more.)

Why define A? Note that for a vertex $v \in A$, there is a path of reverse edges (with positive residual capacity) from v to s in the residual network G_f. So we just have to prove that all vertices with excess have to be in A.

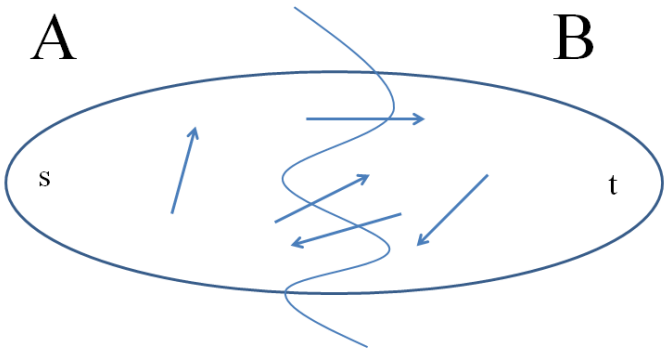

Figure 9: Visualization of a cut. Recall that we can partition edges into 4 categories:(i) edges with both endpoints in A; (ii) edges with both endpoints in B; (iii) edges sticking out of B; (iv) edges sticking into B.

Define $B = V - A$. Certainly s is in A, and hence not in B. (As we'll see, t isn't in B either.) We might have $B = \emptyset$ but this fine with us (we just want no vertices with excess in B).

The key trick is to consider the quantity

$$\sum_{v \in B} \underbrace{[\text{flow out of } v \text{ - flow in to } v]}_{\leq 0}. \tag{2}$$

[8] The argument bears some resemblance to the final step of the proof of the max-flow/min-cut theorem (Lecture #2) — the part where, given a residual network with no s-t path, we exhibited an s-t cut with value equal to that of the current flow.

Because f is a preflow (with flow in at least flow out except at s) and $s \notin B$, every term of (2) is non-positive. On the other hand, recall from Lecture #2 that we can write the sum in different way, focusing on edges rather than vertices. The partition of V into A and B buckets edges into four categories (Figure 9): (i) edges with both endpoints in A; (ii) edges with both endpoints in B; (iii) edges sticking out of B; (iv) edges sticking into B. Edges of type (i) are clearly irrelevant for (2) (the sum only concerns vertices of B). An edge $e = (v, w)$ of type (ii) contributes the value f_e once positively (as flow out of v) and once negatively (as flow into w), and these cancel out. By the same reasoning, edges of type (iii) and (iv) contribute once positively and once negatively, respectively. When the dust settles, we find that the quantity in (2) can also be written as

$$\sum_{e \in \delta^+(B)} \underbrace{f_e}_{\geq 0} - \sum_{e \in \delta^-(B)} \underbrace{f_e}_{=0}; \tag{3}$$

recall the notation $\delta^+(B)$ and $\delta^-(B)$ for the edges of G that stick out of and into B, respectively. Clearly each term is the first sum is nonnegative. Each term is the second sum must be zero: an edge $e \in \delta^-(B)$ sticks into A, so if $f_e > 0$ then the set A of vertices reachable by flow-carrying edges would not have gotten stuck as soon as it did.

The quantities (2) and (3) are equal, yet one is non-positive and the other non-negative. Thus, they must both be 0. Since every term in (2) is non-positive, every term is 0. This implies that conservation constraints (flow in = flow out) hold for all vertices of B. Thus all vertices with excess are in A. By the definition of A, there are paths of reverse edges in the residual network from these vertices to s, as desired. ■

CS261: A Second Course in Algorithms
Lecture #4: Applications of Maximum Flows and Minimum Cuts*

Tim Roughgarden†

January 14, 2016

1 From Algorithms to Applications

The first three lectures covered four maximum flow algorithms (Ford-Fulkerson, Edmonds-Karp, Dinic's blocking flow-based algorithm, and the Goldberg-Tarjan push-relabel algorithm). We could talk about maximum flow algorithms til the cows come home — there has been decades of intense work on the problem, including some interesting breakthroughs just in the last couple of years. But four algorithms is enough for a course like CS261; it's time to move on to applications of the algorithms, and then on to study other fundamental problems.

Let's remind ourselves why we studied these algorithms.

1. Often the best way to get a good understanding of a computational problem is to study algorithms for it. For example, the Ford-Fulkerson algorithm introduced the crucial concept of a residual network, and gave us an excellent initial feel for the maximum flow problem.

2. These algorithms are part of the canon, among the greatest hits of algorithms. So it's fun to know how they work.

3. Maximum flow problems really do come up in practice, so it good to how you might solve them quickly. The push-relabel algorithm is an excellent starting point for implementing fast maximum flow algorithms.

The above reasons assume that we care about the maximum flow problem. And why do we care? Because like all central algorithmic problems, it directly models several well-motivated

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

problems (traffic in transportation networks, oil in a distribution network, data packets in a communication network), and also a surprising number of problems are really just maximum flow in disguise. The lecture gives two examples, in computer vision and in graph matching, and the exercise and problem sets contain several more. Perhaps the most useful skill you can learn in CS261, for both practical and theoretical work, is how to recognize when the tools of the course apply. Hopefully, practice makes perfect.

2 The Minimum Cut Problem

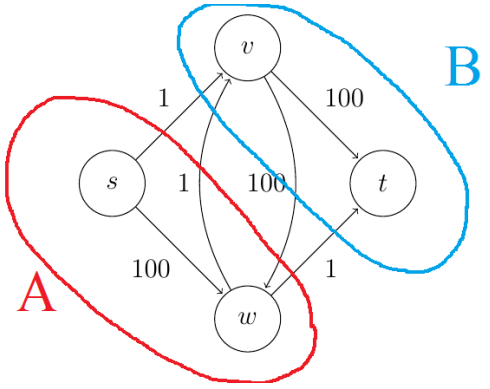

Figure 1: Example of an (s,t)-cut.

The minimum (s,t)-cut problem made a brief cameo in Lecture #2. It is the "dual" problem to maximum flow, in a sense we'll make precise in later lectures, and it is just as ubiquitous in applications. In the minimum (s,t)-cut problem, the input is the same as in the maximum flow problem (a directed graph, source and sink vertices, and edge capacities). The feasible solutions are the (s,t)-cuts, meaning the partitions of the vertex V into two sets A and B with $s \in A$ and $t \in B$ (Figure 1). The objective is to compute the s-t cut with the minimum capacity, meaning the total capacity on edges sticking out of the source-side of the cut (those sticking in don't count):

$$\text{capacity of } (A,B) = \sum_{e \in \delta^+(a)} u_e.$$

In Lecture #2 we noted a simple but extremely useful fact.

Corollary 2.1 *The minimum s-t cut problem reduces in linear time to the maximum flow problem.*

Recall the argument: given a maximum flow, just do breadth- or depth-first search from s in the residual graph (in linear time). We proved that if this search gets stuck at A, then $(A, V-A)$ is an (s,t)-cut with capacity equal to that of the flow; since no cut has capacity less than any flow, the cut $(A, V-A)$ must be a minimum cut.

While there are some algorithms for solving the minimum (s,t)-cut problem without going through maximum flows (especially for undirected graphs), in practice it is very common to solve it via this reduction. Next is an application of the problem to a basic image segmentation task.

3 Image Segmentation

3.1 The Problem

We consider the problem of classifying the pixels of an image as either foreground or background. We model the problem as follows. The input is an undirected graph $G = (V, E)$, where V is the set of pixels. The edges E designate pairs of pixels as neighbors. For example, a common input is a grid graph (Figure 2(a)), with an edge between two pixels that different by 1 in of the two coordinates. (Sometimes one also throws in the diagonals.) In any case, the solution we present works no matter than the graph G is.

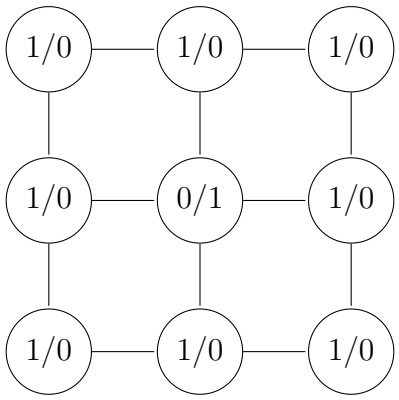

Figure 2: Example of a grid network. In each vertex, first value denotes a_v and second value denotes b_v.

The input also contains $2|V| + |E|$ parameter values. Each vertex v is annotated with two nonnegative numbers a_v and b_v, and each edge e has a nonnegative value p_e. We discuss the semantics of these shortly.

The feasible outputs are the partitions V into a foreground X and background Y; it's OK if X or Y is empty. We assess the quality of a solution by the objective function

$$\sum_{v \in X} a_v + \sum_{v \in Y} b_v - \sum_{e \in \delta(X)} p_e, \tag{1}$$

which we want to make as large as possible. ($\delta(X)$ denotes the edges cut by the partition (X, Y), with one endpoint on each side.)

We see that a vertex v earns a "prize" of a_v if it is included in X and b_v otherwise. In practice, these parameter values come from a prior as to whether a pixel v is more "likely" to be in the foreground (in which case a_v is big and b_v small) or in the background (leading to a big b_v and small a_v. It's not important for our purposes how this prior or these parameter are chosen, but it's easy to imagine examples. Perhaps a light blue pixel is typically part of the background (namely, the sky). Or perhaps one already knows a similar image that has already been segmented, like one taken earlier from the same position, and then declares that each pixel's region is likely to be the same as in the reference image.

If all we had were the a's and b's, the problem would be trivial — independently for each pixel, you would just assign it optimally to either X (if $a_v > b_v$) or Y (if $b_v > a_v$). The point of the neighboring relation E is that we also expect that images are mostly "smooth," with neighboring pixels much more likely to be in the same region than in different regions. The penalty p_e is incurred whenever the endpoints of e violate this prior belief. In machine learning terminology, the final objective (1) corresponds to a massaged version of the "maximum likelihood" objective function.

For example, suppose all p_e's are 0 in Figure 2(a). Then, the optimal solution assigns the entire boundary to the foreground and the middle pixel to the background. The objective function would be 9. If all the p_e's were 1, however, then this feasible solution would have value only 5 (because of the four cut edges). The optimal solution assigns all 9 pixels to the foreground, for a value of 8. The latter computation effectively recovers a corrupted pixel inside some homogeneous region.

3.2 Toward a Reduction

Theorem 3.1 *The image segmentation problem reduces, in linear time, to the minimum (s,t)-cut problem (and hence to the maximum flow problem).*

How would one ever suspect that such a reduction exists? The big clue is the form of the output of the image segmentation problem, as the partition of a vertex set into two pieces. This sure sounds like a cut problem. The coolest thing that could be true is that the problem reduces to a cut problem that we already know how to solve, like the minimum (s,t)-cut problem.

Digging deeper, there are several differences between image segmentation and (s,t)-cut that might give us pause (Table 1). For example, while both problems have one parameter per edge, the image segmentation problem has two parameters per vertex that seem to have no analog in the minimum (s,t)-cut problem. Happily, all of these issues can be addressed with the right reduction.

Minimum (s,t)-cut	Image segmentation
minimization objective	maximization objective
source s, sink t	no source, sink vertices
directed	undirected
no vertex parameters	a_v, b_v for each $v \in V$

Table 1: Differences between the image segmentation problem and the minimum (s,t)-cut problem.

3.3 Transforming the Objective Function

First, it's easy to convert the maximization objective function into a minimization one by multiplying through by -1:

$$\min_{(X,Y)} \sum_{e \in \delta(X)} p_e - \sum_{v \in X} a_v - \sum_{vinY} b_v.$$

Clearly, the optimal solution under this objective is the same as under the original objective.

It's hard not to be a little spooked by the negative numbers in this objective function (e.g., in max flow or min cut, edge capacities are always nonnegative). This is also easy to fix. We just shift the objective function by adding the constant value $\sum_{v \in V} a_v + \sum_{v \in V} b_v$ to every feasible solution. This gives the objective function

$$\min_{(X,Y)} \sum_{e \in \delta(X)} p_e + \sum_{v \in Y} a_v + \sum_{vinX} b_v. \tag{2}$$

Since we shifted all feasible solutions by the same amount, the optimal solution remains unchanged.

3.4 Transforming the Graph

We use tricks familiar from Exercise Set #1. Given the undirected graph $G = (V, E)$, we construct a directed graph $G' = (V', E')$ as follows:

- $V' = V \cup \{s, t\}$ (i.e., add a new source and sink)
- E' has two bidirected edges for each edge e in E (i.e., a directed edge in either direction). The capacity of both directed edges is defined to be p_e, the given penalty of edge e (Figure 3).

Figure 3: The (undirected) edges of G are bidirected in G'.

- E' also has an edge (s, v) for every pixel $v \in V$, with capacity $u_{sv} = a_v$.
- E' has an edge (v, t) for every pixel $v \in V$, with capacity $u_{vt} = b_v$.

See Figure 4 for a small example of the transformation.

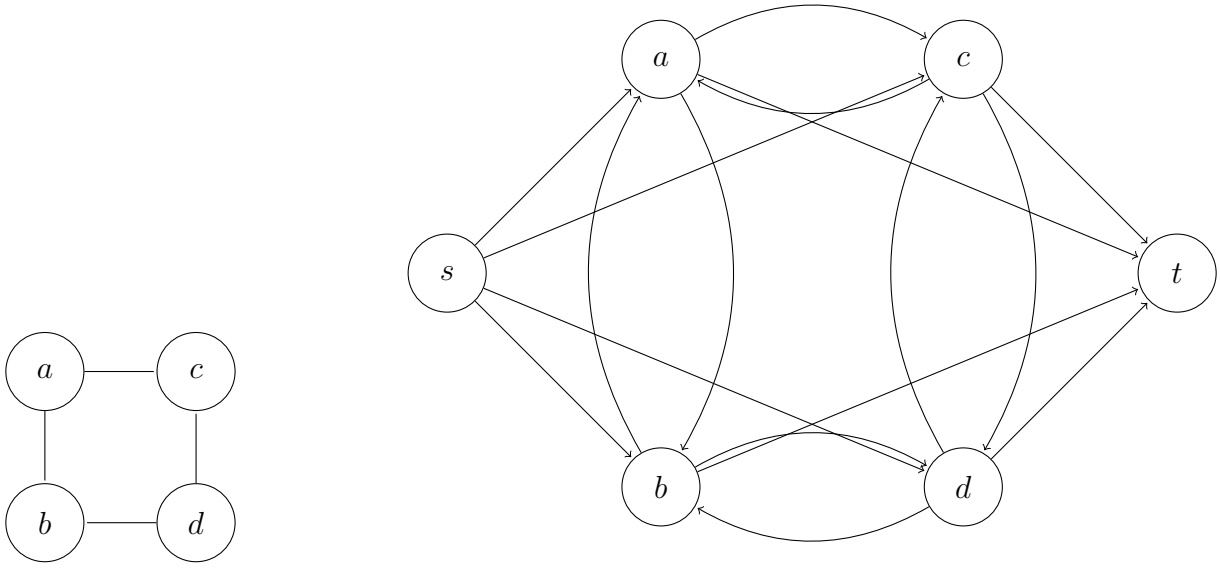

Figure 4: (a) initial network and (b) the transformation

3.5 Proof of Theorem 3.1

Consider an input $G = (V, E)$ to the image segmentation problem and directed graph $G' = (V', E')$ constructed by the reduction above. There is a natural bijection between partitions (X, Y) of V and (s, t)-cut (A, B) of G', with $A \leftrightarrow X \cup \{s\}$ and $B \leftrightarrow Y \cup \{t\}$. The key claim is that this correspondence preserves objective function value — that the capacity of every (s, t)-cut (A, B) of G' is precisely the objective function value (under (2))of the partition $(A \setminus \{s\}, B \setminus \{t\})$.

So fix an (s, t)-cut $(X \cup \{s\}, Y \cup \{t\})$ of G'. Here are the edges sticking out of $X \cup \{s\}$:

1. for every $v \in Y$, $\delta^+(X \cup \{s\})$ contains the edge (s, v), which has capacity a_v;

2. for every $v \in X$, $\delta^+(X \cup \{s\})$ contains the edge (v, t), which has capacity b_v;

3. for every edge $e \in \delta(X)$, $\delta^+(X \cup \{s\})$ contains exactly one of the two corresponding directed edges of G' (the other one goes backward), and it has capacity p_e.

These are precisely the edges of $\delta^+(X \cup \{s\})$. We compute the cut's capacity just be summing up, for a total of

$$\sum_{v \in Y} a_v + \sum_{v \in X} b_v + \sum_{e \in \delta(X)} p_e.$$

This is identical to the objective function value (2) of the partition (X, Y). We conclude that computing the optimal such partition reduces to computing a minimum (s, t)-cut of G'. The reduction can be implemented in linear time.

4 Bipartite Matching

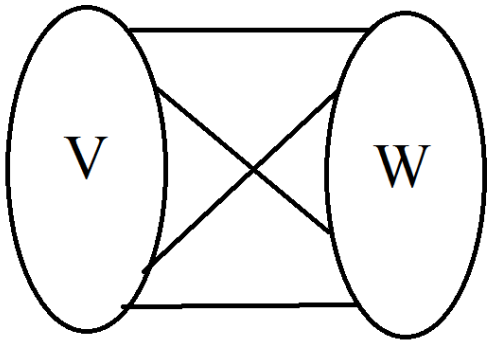

Figure 5: Visualization of bipartite graph. Edges exist only between the partitions V and W.

We next give a famous application of maximum flow. This application also serves as a segue between the first two major topics of the course, the maximum flow problem and graph matching problems.

In the bipartite matching problem, the input is an undirected bipartite graph $G = (V \cup W, E)$, with every edge of E having one endpoint in each of V and W. That is, no edges internal to V or W are allowed (Figure 5). The feasible solutions are the *matchings* of the graph, meaning subsets $S \subseteq E$ of edges that share no endpoints. The goal of the problem is to compute a matching with the maximum-possible cardinality. Said differently, the goal is to pair up as many vertices as possible (using edges of E).

For example, the square graph (Figure 6(a)) is bipartite, and the maximum-cardinality matching has size 2. It matches all of the vertices, which is obviously the best-case scenario. Such a matching is called *perfect*.

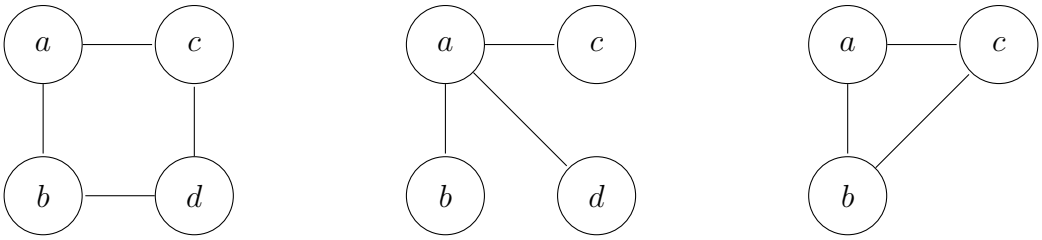

Figure 6: (a) square graph with perfect matching of 2. (b) star graph with maximum-cardinality matching of 1. (c) non-bipartite graph with maximum matching of 1.

Not all graphs have perfect matchings. For example, in the star graph (Figure 6(b)), which is also bipartite, no many how many vertices there are, the maximum-cardinality matching has size only 1.

It's also interesting to discuss the maximum-cardinality matching problem in general (non-bipartite) graphs (like Figure 6(c)), but this is a harder topic that we won't cover here. While one can of course consider the bipartite special case of any graph problem, in matching bipartite graphs play a particularly fundamental role. First, matching theory is nicer and matching algorithms are faster for bipartite graphs than for non-bipartite graphs. Second, a majority of the applications of already in the bipartite special case — assigning workers to jobs, courses to room/time slots, medical residents to hospitals, etc.

Claim: maximum-cardinality matching reduces in linear time to maximum flow.

Proof sketch: Given an undirected bipartite graph $(V \cup W, E)$, construct a directed graph G' as in Figure 7(b). We add a source and sink, so the new vertex set is $V' = V \cup W \cup \{s, t\}$. To obtain E' from E, we direct all edges of G from V to W and also add edges from s to every vertex of V and from every vertex of W to t. Edges incident to s or t have capacity 1, reflecting the constraints the each vertex of $V \cup W$ can only be matches to one other vertex. Each edge (v, w) directed from V to W can be given any capacity that is at least 1 (v can only receive one unit of flow, anyway); for simplicity, give all these edges infinite capacity.

You should check that there is a one-to-one correspondence between matchings of G and integer-valued flows in G', with edge (v, w) corresponding to one unit of flow on the path $s \to v \to w \to t$ in G' (Figure 7). This bijection preserves the objective function value. Thus, given an integral maximum flow in G', the edges from V to W that carry flow form a maximum matching.[1]

[1] All of the maximum flow algorithms that we've discussed return an integral maximum flow provided all the edge capacities are integers. The reason is that inductively, the current (pre)flow, and hence the residual capacities, and hence the augmentation amount, stay integral throughout these algorithms.

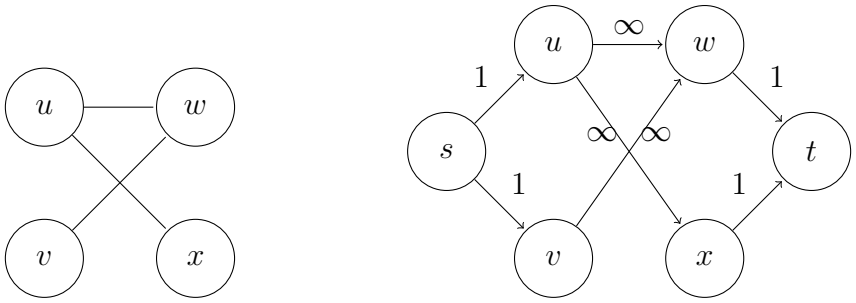

Figure 7: (a) original bipartite graph G and (b) the constructed directed graph G. There is one-to-one correspondence between matchings of G and integer-valued flows of G' e.g. (v, w) in G corresponds to one unit of flow on $s \to v \to w \to t$ in G'.

5 Hall's Theorem

In this final section we tie together a number of courses ongoing themes. We previously asked the question

> How do we know when we're done (i.e., optimal)?

for the maximum flow problem. Let's ask it again for the maximum-cardinality bipartite matching problem. Using the reduction in Section 4, we can translate the optimality conditions for the maximum flow problem (i.e., the max-flow/min-cut theorem) into a famous optimality condition for bipartite matchings.

Consider a bipartite graph $G = (V \cup W, E)$ with $|V| \leq |W|$, renaming V, W if necessary. Call a matching of G *perfect* if it matches every vertex in V; clearly, a perfect matching is a maximum matching. Let's first understand which bipartite graphs admit a perfect matching.

Some notation: for a subset $S \subseteq V$, let $N(S)$ denote the union of the neighborhoods of the vertices of S: $N(S) = \{w \in W : \exists v \in S \text{ s.t. } (v, w) \in E\}$. See Figure 8 for two examples of such neighbor sets.

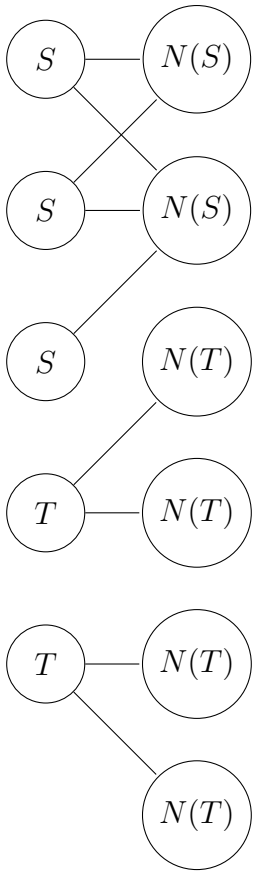

Figure 8: Two examples of vertex sets S and T and their respective neighbour sets $N(S)$ and $N(T)$.

Does the graph in Figure 8 have a perfect matching? A little thought shows that the answer is "no." The three vertices of S have only two distinct neighbors between them. Since each vertex can only be matched to one other vertex, there is no hope of matching more than two of the three vertices of S.

More generally, if a bipartite graph has a *constricting set* $S \subseteq V$, meaning one with $|N(S)| < |S|$, then it has no perfect matching. But what about the converse? If a bipartite graph admits no perfect matching, can you always find a short convincing argument of this fact, in the form of a constricting set? Or could there be obstructions to perfect matchings beyond just constricting sets? *Hall's Theorem* gives the beautiful answer that constricting sets are the only obstacles to perfect matchings.[2]

Theorem 5.1 (Hall's Theorem) *A bipartite graph $(V \cup W, E)$ with $|V| \leq |W|$ has a perfect matching if and only if, for every subset $S \subseteq V$, $|N(S)| \geq |S|$.*

[2]Hall's theorem actually predates the max-flow/min-cut theorem by 20 years.

Thus, it's not only easy to convince someone that a graph has a perfect matching (just exhibit a matching), it's also easy to convince someone that a graph does *not* have a perfect matching (just exhibit a constricting set).

Proof of Theorem 5.1: We already argued the easy "only if" direction. For the "if" direction, suppose that $|N(S)| \geq |S|$ for every $S \subseteq V$.

Claim: in the flow network G' that corresponds to G (Figure 7), every (s,t)-cut has capacity at least $|V|$.

To see why the claim implies the theorem, note that it implies that the minimum cut value in G' is at least $|V|$, so the maximum flow in G' is at least $|V|$ (by the max-flow/min-cut theorem), and an integral flow with value $|V|$ corresponds to a perfect matching of G.

Proof of claim: Fix an (s,t)-cut (A,B) of G'. Let $S = A \cap V$ denote the vertices of V that lie on the source side. Since $s \in A$, all (unit-capacity) edges from s to vertices of $V - A$ contribute to the capacity of (A,B). Recall that we gave the edges directed from V to W infinite capacity. Thus, if some vertex w of $N(S)$ fails to also be in A, then the cut (A,B) has infinite capacity (because of the edge from S to w) and there is nothing to prove. So suppose all of $N(S)$ belongs to A. Then all of the (unit-capacity) edges from vertices of $N(S)$ to t contribute to the capacity of (A,B). Summing up, we have

$$\begin{aligned} \text{capacity of } (A,B) &\geq \underbrace{(|V| - |S|)}_{\text{edges from } s \text{ to } V-S} + \underbrace{|N(S)|}_{\text{edges from } N(S) \text{ to } t} \\ &\geq |V|, \end{aligned} \tag{3}$$

where (3) follows from the assumption that $|N(S)| \geq |S|$ for every $S \subseteq V$. ■

On Exercise Set #2 you will extend this proof to show that, more generally, for every bipartite graph $(V \cup W, E)$ with $|V| \leq |W|$,

$$\text{size of maximum matching} = \min_{S \subseteq V} \left(|V| - (|S| - |N(S)|)\right).$$

Note that at least $|S| - |N(S)|$ vertices of S are unmatched in every matching.

CS261: A Second Course in Algorithms
Lecture #5: Minimum-Cost Bipartite Matching*

Tim Roughgarden†

January 19, 2016

1 Preliminaries

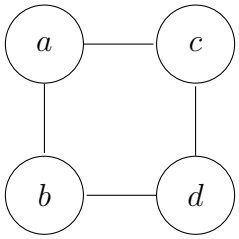

Figure 1: Example of bipartite graph. The edges $\{a, b\}$ and $\{c, d\}$ constitute a matching.

Last lecture introduced the maximum-cardinality bipartite matching problem. Recall that a bipartite graph $G = (V \cup W, E)$ is one whose vertices are split into two sets such that every edge has one endpoint in each set (no edges internal to V or W allowed). Recall that a matching is a subset $M \subseteq E$ of edges with no shared endpoints (e.g., Figure 1). Last lecture, we sketched a simple reduction from this problem to the maximum flow problem. Moreover, we deduced from this reduction and the max-flow/min-cut theorem a famous optimality condition for bipartite matchings. A special case is Hall's theorem, which states that a bipartite graph with $|V| \leq |W|$ has a perfect matching if and only if for every subset $S \subseteq V$ of the left-hand side, the number $|N(S)|$ of S on the right-hand side is at least $|S|$. See Problem Set #2 for quite good running time bounds for the problem.

But what if a bipartite graph has many perfect matchings? In applications, there are often reasons to prefer one over another. For example, when assigning jobs to works, perhaps there are many workers who can perform a particular job, but some of them are better at

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

it than others. The simplest way to model such preferences is attach a *cost* c_e to each edge $e \in E$ of the input bipartite graph $G = (V \cup W, E)$.

We also make three assumptions. These are for convenience, and are not crucial for any of our results.

1. The sets V and W have the same size, call it n. This assumption is easily enforced by adding "dummy vertices" (with no incident edges) to the smaller side.

2. The graph G has at least one perfect matching. This is easily enforced by adding "dummy edges" that have a very high cost (e.g., one such edge from the ith vertex of V to the ith vertex of W, for each i).

3. Edge costs are nonnegative. This can be enforced in the obvious way: if the most negative edge cost is $-M$, just add M to the cost of every edge. This adds the same number (nM) to every perfect matching, and thus does not change the problem.

The goal in the minimum-cost perfect bipartite matching problem is to compute the perfect matching M that minimizes $\sum_{e \in M} c_e$. The feasible solutions to the problem are the perfect matchings of G. An equivalent problem is the maximum-weight perfect bipartite matching problem (just multiply all weights by -1 to transform them into costs).

When every edge has the same cost and we only care about cardinality, the problem reduces to the maximum flow problem (Lecture #4). With general costs, there does not seem to be a natural reduction to the maximum flow problem. It's true that edges in a flow network come with attached numbers (their capacities), but there is a type mismatch: edge capacities affect the set of feasible solutions but not their objective function values, while edge costs do the opposite. Thus, the minimum-cost perfect bipartite matching problem seems like a new problem, for which we have to design an algorithm from scratch.

We'll follow the same kind of disciplined approach that served us so well in the maximum flow problem. First, we identify optimality conditions, which tell us when a given perfect matching is in fact minimum-cost. This step is structural, not algorithmic, and is analogous to our result in Lecture #2 that a flow is maximum if and only if there is no s-t path in the residual network. Then, we design an algorithm that can only terminate with the feasibility and optimality conditions satisfied. For maximum flow, we had one algorithmic paradigm that maintained feasibility and worked toward the optimality conditions (augmenting path algorithms), and a second paradigm that maintain the optimality conditions and worked toward feasibility (push-relabel). Here, we follow the second approach. We'll identify invariants that imply the optimality condition, and design an algorithm that keeps them satisfied at all times and works toward a feasible solution (i.e., a perfect matching).

2 Optimality Conditions

How do we know if a given perfect matching has the minimum-possible cost? Optimality conditions are different for different problems, but for the problems studied in CS261 they are

all quite natural in hindsight. We first need an analog of a residual network. This requires some definitions (see also Figure 2).

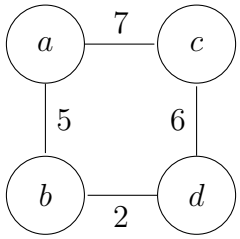

Figure 2: If our matching contains $\{a, b\}$ and $\{c, d\}$, then $a \to b \to d \to c \to a$ is both an M-alternating cycle and a negative cycle.

Definition 2.1 (Negative Cycle) Let M be a matching in the bipartite graph $G = (V \cup W, E)$.

(a) A cycle C of G is *M-alternating* if every other edge of C belongs to M (Figure 2).[1]

(b) An M-alternating cycle is *negative* if the edges in the matching have higher cost than those outside the matching:

$$\sum_{e \cup C \cap M} c_e > \sum_{e \in C \setminus M} c_e.$$

Otherwise, it is *nonnegative*.

One interesting thing about alternating cycles is that by "toggling" the edges of C with respect to M — that is, removing the edges of $C \cap M$ and plugging in the edges of $C \setminus M$ — yields a new matching M' that matches exactly the same set of vertices. (Vertices outside of C are clearly unaffected; vertices inside C remain matched to precisely one other vertex of C, just a different one than before.)

Suppose M is a perfect matching, and we toggle the edges of an M-alternating cycle to get another (perfect) matching M'. Dropping the edges from $C \cap M$ saves us a cost of $\sum_{e \cup C \cap M} c_e$, while adding the edges of $C \setminus M$ cost us $\sum_{e \in C \setminus M} c_e$. Then M' has smaller cost than M if and only if C is a negative cycle.

The point of a negative cycle is that it offers a quick and convincing proof that a perfect matching is not minimum-cost (since toggling the edges of the cycle yields a cheaper matching). But what about the converse? If a perfect matching is not minimum-cost, are we guaranteed such a short and convincing proof of this fact? Or are there "obstacles" to optimality beyond the obvious ones of negative cycles?

[1]Since G is bipartite, C is necessarily an even cycle. One certainly can't have more than every other edge of C contained in the matching M.

Theorem 2.2 (Optimality Conditions for Min-Cost Bipartite Matching) *A perfect matching of a bipartite graph has minimum-cost if and only if there is no negative M-alternating cycle.*

Proof: We have already argued the "only if" direction. For the harder "if" direction, suppose that M is a perfect matching and that there is no negative M-alternating cycle. Let M' be any other perfect matching; we want to show that the cost of M' is at least that of M. Consider $M \oplus M'$, meaning the symmetric difference of M, M' (if you want to think of them as sets) or their XOR (if you want to think of them as 0/1 vectors). See Figure 3 for two examples.

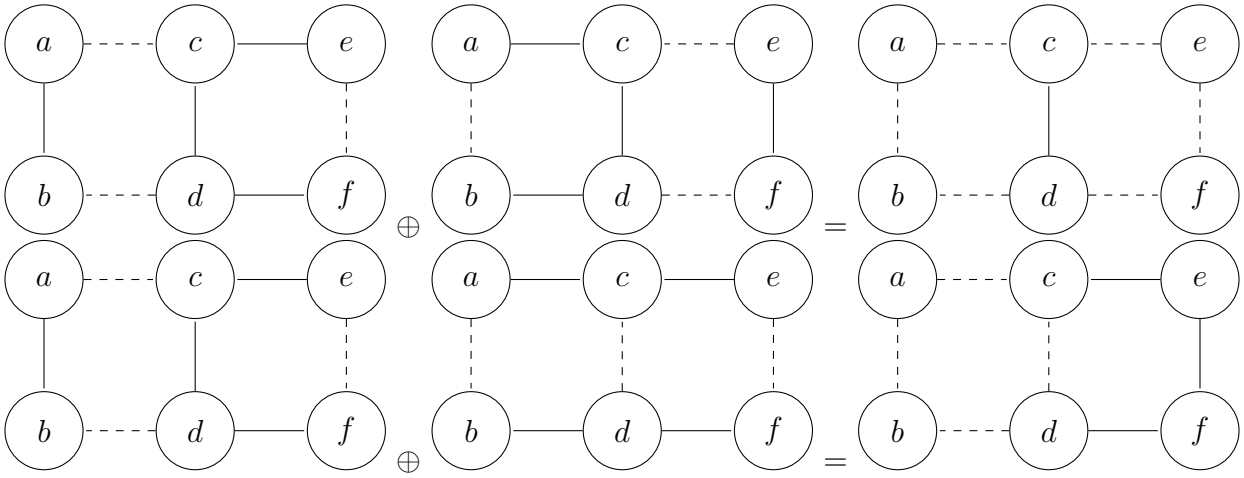

Figure 3: Two examples that show what happens when we XOR two matchings (the dashed edges).

In general, $M \oplus M'$ is a union of (vertex-)disjoint cycles. The reason is that, since every vertex has degree 1 in both M and M', every vertex of v has degree either 0 (if it is matched to the same vertex in both M and M') or 2 (otherwise). A graph with all degrees either 0 or 2 must be the union of disjoint cycles.

Since taking the symmetric difference/XOR with the same set two times in a row recovers the initial set, $(M \oplus M') \oplus M' = M$. Since $M \oplus M'$ is a disjoint union of cycles, taking the symmetric different/XOR with $M \oplus M'$ just means toggling the edges in each of its cycles (since they are disjoint, they don't interfere and the toggling can be done in parallel). Each of these cycles is M-alternating, and by assumption each is nonnegative. Thus toggling the edges of the cycles can only produce a more expensive perfect matching M'. Since M' was an arbitrary perfect matching, M must be a minimum-cost perfect matching. ■

3 Reduced Costs and Invariants

Now that we know when we're done, we work toward algorithms that terminate with the optimality conditions satisfied. Following the push-relabel approach (Lecture #3), we next identify invariants that will imply the optimality conditions at all times. Our algorithm will maintain these as it works toward a feasible solution (i.e., a perfect matching). Continuing the analogy with the push-relabel paradigm, we maintain a extra number p_v for each vertex $v \in V \cup W$, called a *price* (analogous to the "heights" in Lecture #3). Prices are allowed to be positive or negative. We use prices to force us to add edges to our current matching only in a disciplined way, somewhat analogous to how we only pushed flow "downhill" in Lecture #3.

Formally, for a price vector p (indexed by vertices), we define the *reduced cost* of an edge $e = (v, w)$ by

$$c_e^p = c_e - p_v - p_w. \tag{1}$$

Here are our invariants, which are respect to a current matching M and a current vector p of prices.

Invariants

1. Every edge of G has nonnegative reduced cost.
2. Every edge of M is *tight*, meaning it has zero reduced cost.

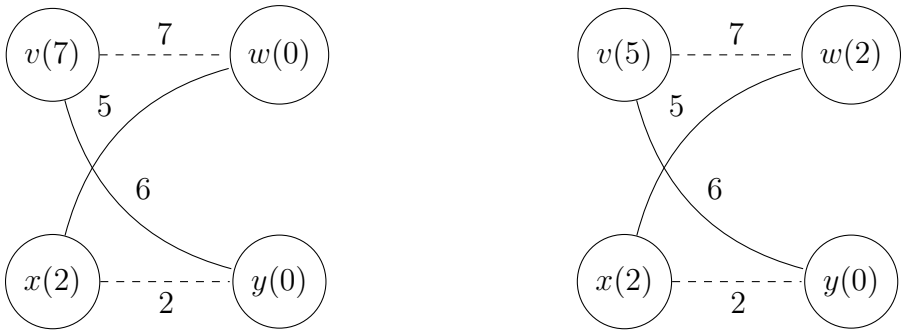

Figure 4: For the given (perfect) matching (dashed edges), (a) violates invariant 1, while (b) satisfies all invariants.

For example, consider the (perfect) matching in Figure 4. Is it possible to define prices so that the invariants hold? To satisfy the second invariant, we need to make the edges (v, w) and (x, y) tight. We could try setting the price of w and y to 0, which then dictates setting $p_v = 7$ and $p_x = 2$ (Figure 4(a)). This violates the first invariant, however, since the reduced cost of edge (v, y) is -1. We can satisfy both invariants by resetting $p_v = 5$ and $p_w = 2$; then both edges in the matching are tight and the other two edges have reduced cost 1 (Figure 4(b)).

The matching in Figure 4 is a min-cost perfect matching. This is no coincidence.

Lemma 3.1 (Invariants Imply Optimality Condition) *If M is a perfect matching and both invariants hold, then M is a minimum-cost perfect matching.*

Proof: Let M be a perfect matching such that both invariants hold. By our optimality condition (Theorem 2.2), we just need to check that there is no negative cycle. So consider any M-alternating cycle C (remember a negative cycle must be M-alternating, by definition). We want to show that the edges of C that are in M have cost at most that of the edges of C not in M. Adding and subtracting $\sum_{v \in C} p_v$ and using the fact that every vertex of C is the endpoint of exactly one edge of $C \cap M$ and of $C \setminus M$ (e.g., Figure 5), we can write

$$\sum_{e \in C \cap M} c_e = \sum_{e \in C \cap M} c_e^p + \sum_{v \in C} p_v \tag{2}$$

and

$$\sum_{e \in C \setminus M} c_e = \sum_{e \in C \setminus M} c_e^p + \sum_{v \in C} p_v. \tag{3}$$

(We are abusing notation and using C both to denote the vertices in the cycle and the edges in the cycle; hopefully the meaning is always clear from context.) Clearly, the third terms in (2) and (3) are the same. By the second invariant (edges of M are tight), the second term in (2) is 0. By the first invariant (all edges have nonnegative reduced cost), the second term in (3) is at least 0. We conclude that the left-hand side of (2) is at most that of (3), which proves that C is not a negative cycle. Since C was arbitrary M-alternating cycle, the proof is complete. ■

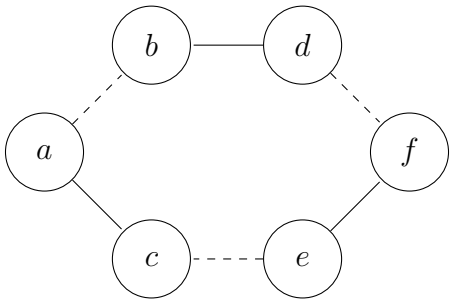

Figure 5: In the example M-alternating cycle and matching shown above, every vertex is an endpoint of exactly one edge in M and one edge not in M.

4 The Hungarian Algorithm

Lemma 3.1 reduces the problem of designing a correct algorithm for the minimum-cost perfect bipartite matching problem to that of designing an algorithm that maintains the two invariants and computes an arbitrary perfect matching. This section presents such an algorithm.

4.1 Backstory

The algorithm we present goes by various names, the two most common being the *Hungarian algorithm* and the *Kuhn-Munkres algorithm*. You might therefore find it weird that Kuhn and Munkres are American. Here's the story. In the early/mid-1950s, Kuhn really wanted an algorithm for solving the minimum-cost bipartite matching problem. So he was reading a graph theory book by Kőnig. This was actually the first graph theory book ever written — in the 1930s, and available in the U.S. only in 1950 (even then, only in German). Kuhn was intrigued by an offhand citation in the book, to a paper of Egerváry. Kuhn tracked down the paper, which was written in Hungarian. This was *way* before Google Translate, so he bought a big English-Hungarian dictionary and translated the whole thing. And indeed, Egerváry's paper had the key ideas necessary for a good algorithm. Kőnig and Egerváry were both Hungarian, so Kuhn called his algorithm the Hungarian algorithm. Kuhn only proved termination of his algorithm, and soon thereafter Munkres observed a polynomial time bound (basically the bound proved in this lecture). Hence, also called the Kuhn-Munkres algorithm.

In a (final?) twist to the story, in 2006 it was discovered that Jacobi, the famous mathematician (you've studied multiple concepts named after him in your math classes), came up with an equivalent algorithm in the 1840s! (Published only posthumously, in 1890.) Kuhn, then in his 80s, was a good sport about it, giving talks with the title "The Hungarian Algorithm and How Jacobi Beat Me By 100 Years."

4.2 The Algorithm: High-Level Structure

The Hungarian algorithm maintains both a matching M and prices p. The initialization is straightforward.

Initialization

set $M = \emptyset$
set $p_v = 0$ for all $v \in V \cup W$

The second invariant holds vacuously. The first invariant holds because we are assuming that all edge costs (and hence initial reduced costs) are nonnegative.

Informally (and way underspecified), the main loop works as follows. The terms "augment," "good path," and "good set" will be defined shortly.

Main Loop (High-Level)

while M is not a perfect matching **do**
    if there is a good path P **then**
        augment M by P
    else
        find a good set S; update prices accordingly

4.3 Good Paths

We now start filling in the details. Fix the current matching M and current prices p. Call a path P from v to w *good* if:

1. both endpoints v, w are unmatched in M, with $v \in V$ and $w \in W$ (hence P has odd length);
2. it alternates edges out of M with edges in M (since v, w are unmatched, the first and last edges are not in M);
3. every edge of P is tight (i.e., has zero reduced cost and hence eligible to be included in the current matching).

Figure 6 depicts a simple example of a good path.

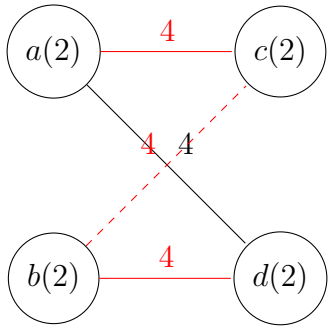

Figure 6: Dashed edges denote edges in the matching and red edges denote a good path.

The reason we care about good paths is that such a path allows us to increase the cardinality of M without breaking either invariant. Specifically, consider replacing M by $M' = M \oplus P$. This can be thought of as toggling which edges of P are in the current matching. By definition, a good path is M-alternating, with first and last hops not in M; thus, $|P \cap M| = |P \setminus M| - 1$, and the size of M' is one more than M. (E.g., if P is a 9-hop path, this toggling removes 4 edges from M but that adds in 5 other edges.) No reduced costs have changed, so certainly the first invariant still holds. All edges of P are tight be definition, so the second invariant also continues to hold.

Augmentation Step

given a good path P, replace M by $M \oplus P$

Finding a good path is definitely progress — after n such augmentations, the current matching M must be perfect and (since the invariants hold) we're done. How can we efficiently find such a path? And what do we do if there's no such path?

To efficiently search for such a path, let's just follow our nose. It turns out that breadth-first search (BFS), with a twist to enforce M-alternation, is all we need.

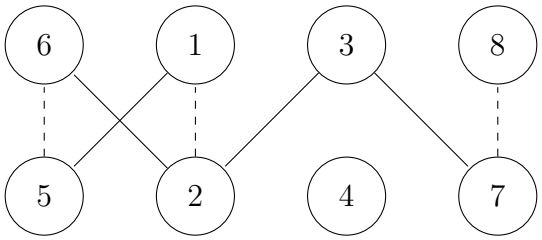

Figure 7: Dashed edges are the edges in the matching. Only tight edges are shown.

The algorithm will be clear from an example. Consider the graph in Figure 7; only the tight edges are shown. Note that the graph does not contain a good path (if it did, we could use it to augment the current matching to obtain a perfect matching, but vertex #4 is isolated so there is no perfect matching.).[2] So we know in advance that our search will fail. But it's useful to see what happens when it fails.

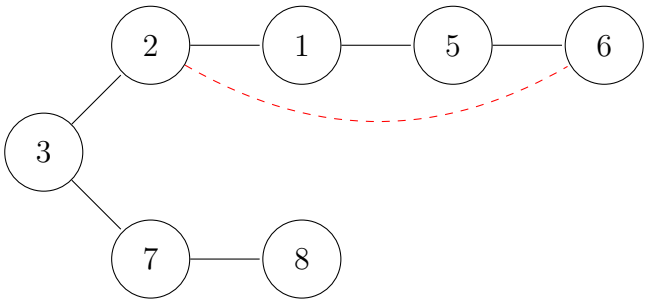

Figure 8: BFS spanning tree if we start BFS travel from node 3. Note that the edge $\{2, 6\}$ is not used.

We start a graph search from an unmatched vertex of V (the first such vertex, say); see also Figure 8. In the example, this is vertex #3. Layer 0 of our search tree is $\{3\}$. We obtain layer 1 from layer 0 by BFS; thus, layer 1 is $\{2, 7\}$. Note that if either 2 or 7 is unmatched, then we have found a (one-hop) good path and we can stop the search. Both 2 and 7 are already matched in the example, however. Here is the twist to BFS: at the next layer 2 we put only the vertices to which 2 and 7 are matched, namely 1 and 8. Conspicuous in its absence is vertex #6; in regular BFS it would be included in layer 2, but here we omit it because it is not matched to a vertex of layer 1. The reason for this twist is that we want every path in our search tree to be M-alternating (since good paths need to be M-alternating).

[2]Remember we assume only that G contains a perfect matching; the subgraph of tight edges at any given time will generally not contain a perfect matching.

We then switch back to BFS. At vertex #8 we're stuck (we've already seen its only neighbor, #7). At vertex #1, we've already seen its neighbor 2 but have not yet seen vertex #5, so the third layer is $\{5\}$. Note that if 5 were unmatched, we would have found a good path, from 5 back to the root 3. (All edges in the tree are tight by definition; the path is alternating and of odd length, joining two unmatched vertices of V and W.) But 5 is already matched to 6, so layer 4 of the search tree is $\{6\}$. We've already seen both of 6's neighbors before, so at this point we're stuck and the search terminates.

In general, here is the search procedure for finding a good path (given a current matching M and prices p).

Searching for a Good Path

level 0 = the first unmatched vertex r of V
while not stuck and no other unmatched vertex found **do**
    if next level i is odd **then**
        define level i from level $i-1$ via BFS
        `// i.e., neighbors of level` $i-1$ `not already seen`
    else if next level i is even **then**
        define level i as the vertices matched in M to vertices at level $i-1$
if found another unmatched vertex w **then**
    return the search tree path between the root r and w
else
    return "stuck"

To understand this subroutine, consider an edge $(v, w) \in M$, and suppose that v is reached first, at level i. Importantly, it is not possible that w is also reached at level i. This is where we use the assumption that G is bipartite: if v, w are reached in the same level, then pasting together the paths from r to v and from r to w (which have the same length) with the edge (v, w) exhibits an odd cycle, contradicting bipartiteness. Second, we claim that i must be odd (cf., Figure 8). The reason is just that, by construction, every vertex at an even level (other than 0) is the second endpoint reached of some matched edge (and hence cannot be the endpoint of any other matched edge). We conclude that:

(*) if either endpoint of an edge of M is reached in the search tree, then both endpoints are reached, and they appear at consecutive levels $i, i+1$ with i odd.

Suppose the search tree reaches an unmatched vertex w other than the root r. Since every vertex at an even level (after 0) is matched to a vertex at the previous level, w must be at an odd level (and hence in W). By construction, every edge of the search tree is tight, and every path in the tree is M-alternating. Thus the r-w path in the search tree is a good path, allowing us to increase the size of M by 1.

4.4 Good Sets

Suppose the search gets stuck, as in our example. How do we make progress, and in what sense? In this case, we keep the matching the same but update the prices.

Define $S \subseteq V$ as the vertices at even levels. Define $N(S) \subseteq W$ as the neighbors of S via tight edges, i.e.,

$$N(S) = \{w : \exists v \in S \text{ with } (v, w) \text{ tight}\}. \tag{4}$$

We claim that $N(S)$ is precisely the vertices that appear in the odd levels of the search tree. In proof, first note that every vertex at an odd level is (by construction/BFS) adjacent via a tight edge to a vertex at the previous (even) level. For the converse, every vertex $w \in N(S)$ must be reached in the search, because (by basic properties of graph search) the search can only stuck if there are no unexplored edges out of any even vertex.

The set S is a *good set*, in that is satisfies:

1. S contains an unmatched vertex;
2. every vertex of $N(S)$ is matched in M to a vertex of S (since the search failed, every vertex in an odd level is matched to some vertex at the next (even) level).

See also Figure 9.

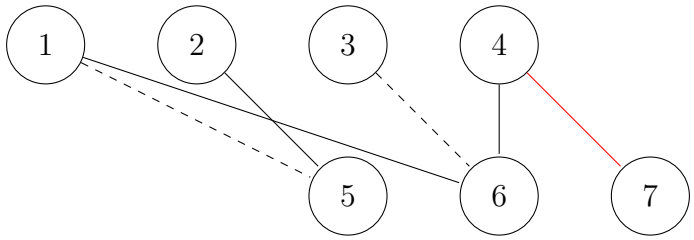

Figure 9: $S = \{1, 2, 3, 4\}$ is example of good set, with $N(S) = \{5, 6\}$. Only black edges are tight edges (i.e. $(4, 7)$ is not tight). The matching edges are dashed.

Having found such a good set S, the Hungarian algorithm updates prices as follows.

Price Update Step

given a good set S, with neighbors via tight edges $N(S)$
for all $v \in S$ **do**
    increase p_v by Δ
for all $w \in N(S)$ **do**
    decrease p_v by Δ
`// Δ is as large as possible, subject to invariants`

Prices in S (on the left-hand side) are increased, while prices in $N(S)$ (on the right-hand side) are decreased by the same amount. How does this affect the reduced cost of each edge of G (Figure 9)?

1. for an edge (v, w) with $v \notin S$ and $w \notin N(S)$, the prices of v, w are unchanged so c^p_{vw} is unchanged;

2. for an edge (v, w) with $v \in S$ and $w \in N(S)$, the sum of the prices of v, w is unchanged (one increased by Δ, the other decreased by Δ) so c^p_{vw} is unchanged;

3. for an edge (v, w) with $v \notin S$ and $w \in N(S)$, p_v stays the same while p_w goes down by Δ, so c^p_{vw} goes up by Δ;

4. for an edge (v, w) with $v \in S$ and $w \notin N(S)$, p_w stays the same while p_v goes up by Δ, so c^p_{vw} goes down by Δ.

So what happens with the invariants? Recalling (*) from Section 4.3, we see that edges of M are in either the first or second category. Thus they stay tight, and the second invariant remains satisfied. The first invariant is endangered by edges in the fourth category, whose reduced costs are dropping with Δ.[3] By the definition of $N(S)$, edges in this category are not tight. So we increase Δ to the largest-possible value subject to the first invariant — the first point at which the reduced cost of some edge in the fourth category is zeroed out.[4]

Every price update makes progress, in the sense that it strictly increases the size of search tree. To see this, suppose a price update causes the edge (v, w) to become tight (with $v \in S$, $w \notin N(S)$). What happens in the next iteration, when we search from the same vertex r for a good path? All edges in the previous search tree fall in the second category, and hence are again tight in the next iteration. Thus, the search procedure will regrow exactly the same search tree as before, will again reach the vertex v, and now will also explore along the newly tight edge (v, w), which adds the additional vertex $w \in W$ to the tree. This can only happen n times in a row before finding a good path, since there are only n vertices in W.

[3] Edges in third category might go from tight to non-tight, but these edges are not in M (every vertex of $N(S)$ is matched to a vertex of S) and so no invariant is violated.

[4] A detail: how do we know that such an edge exists? If not, then all neighbors of S in G (via tight edges or not) belong to $N(S)$. The two properties of good sets imply that $|N(S)| < |S|$. But this violates Hall's condition for perfect matchings (Lecture #4), contradicting our standing assumption that G has at least one perfect matching.

4.5 The Hungarian Algorithm (All in One Place)

The Hungarian Algorithm

```
set M = ∅
set p_v = 0 for all v ∈ V ∪ W
while M is not a perfect matching do
    level 0 of search tree T = the first unmatched vertex r of V
    while not stuck and no other unmatched vertex found do
        if next level i is odd then
            define level i of T from level i − 1 via BFS
            // i.e., neighbors of level i − 1 not already seen
        else if next level i is even then
            define level i of T as the vertices matched in M to vertices
             at level i − 1
    if T contains an unmatched vertex w ∈ W then
        let P denote the r-w path in T
        replace M by M ⊕ P
    else
        let S denote the vertices of T in even levels
        let N(S) denote the vertices of T in odd levels
        for all v ∈ S do
            increase p_v by Δ
        for all w ∈ N(S) do
            decrease p_v by Δ
        // Δ is as large as possible, subject to invariants
return M
```

4.6 Running Time

Since M can only contain n edges, there can only be n iterations that find a good path. Since the search tree can only contain n vertices of W, there can only be n prices updates between iterations that find good paths. Computing the search tree (and hence P or S and $N(S)$) and Δ (if necessary) can be done in $O(m)$ time. This gives a running time bound of $O(mn^2)$. See Problem Set #2 for an implementation with running time $O(mn \log n)$.

4.7 Example

We reinforce the algorithm via an example. Consider the graph in Figure 10.

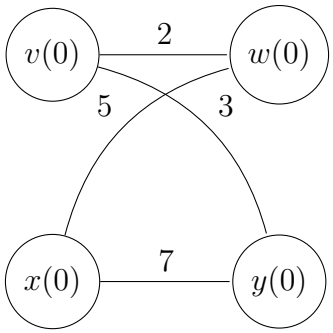

Figure 10: Example graph. Initially, all prices are 0.

We initialize all prices to 0 and the current matching to the empty set. Initially, there are no tight edges, so there is certainly no good path. The search for such a path gets stuck where it starts, at vertex v. So $S = \{v\}$ and $N(S) = \emptyset$. We execute a price update step, raising the price of v to 2, at which point the edge (v, w) becomes tight. Next iteration, the search starts at v, explores the tight edge (v, w), and encounters vertex w, which is unmatched. Thus this edge is added to the current matching. Next iteration, a new search starts from the only remaining unmatched vertex on the left (x). It has no tight incident edges, so the search gets stuck immediately, with $S = \{x\}$ and $N(S) = \emptyset$. We thus do a price update step, with $\Delta = 5$, at which point the edge (x, w) becomes newly tight. Note that the edges (v, y) and (x, y) have reduced costs 1 and 2, respectively, so neither is tight. Next iteration, the search from x explores the incident tight edge (x, w). If w were unmatched, we could stop the search and add the edge (x, w). But w is already matched, to v, so w and v are placed at levels 1 and 2 of the search tree. v has no tight incident edges other than to w, so the search gets stuck here, with $S = \{x, v\}$ and $N(S) = \{w\}$. So we do another a price update step, increasing the price of x and v by Δ and decreasing the price of w by Δ. With $\Delta = 1$, the reduced cost of edge (v, y) gets zeroed out. The final iteration discovers the good path $x \to w \to v \to y$. Augmenting on this path yields the minimum-cost perfect matching $\{(v, y), (x, w)\}$.

CS261: A Second Course in Algorithms
Lecture #6: Generalizations of Maximum Flow and Bipartite Matching*

Tim Roughgarden†

January 21, 2016

1 Fundamental Problems in Combinatorial Optimization

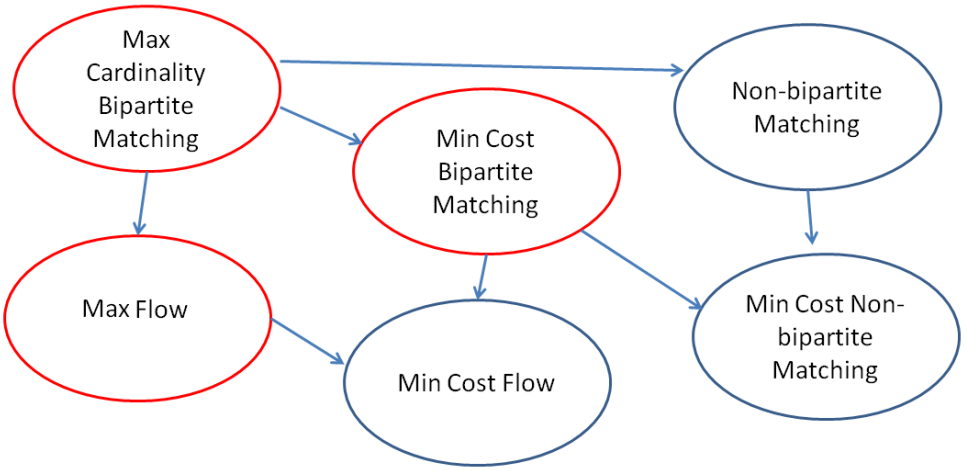

Figure 1: Web of six fundamental problems in combinatorial optimization. The ones covered thus far are in red. Each arrow points from a problem to a generalization of that problem.

We started the course by studying the maximum flow problem and the closely related s-t cut problem. We observed (Lecture #4) that the maximum-cardinality bipartite matching

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

problem can be viewed as a special case of the maximum flow problem (Figure 1). We then generalized the former problem to include edge costs, which seemed to give a problem incomparable to maximum flow.

The inquisitive student might be wondering the following:

1. Is there a natural common generalization of the maximum flow and minimum-cost bipartite matching problems?

2. What's up with graph matching in non-bipartite graphs?

The answer to the first question is "yes," and it's a problem known as minimum-cost flow (Figure 1). For the second question, there is a nice theory of graph matchings in non-bipartite graphs, both for the maximum-cardinality and minimum-cost cases, although the theory is more difficult and the algorithms are slower than in the bipartite case. This lecture introduces the three new problems in Figure 1 and some essential facts you should know about them. The six problems in Figure 1, along with the minimum spanning tree and shortest path problems that you already know well from CS161, arguably form the complete list of the most fundamental problems in *combinatorial optimization*, the study of efficiently optimizing over large collections of discrete structures.

The main take-ways from this lecture's high-level discussion are:

1. You should know about the existence of the minimum-cost flow and non-bipartite matching problems. They do come up in applications, if somewhat less frequently than the problems studied in the first five lectures.

2. There are reasonably efficient algorithms for all of these problems, if a bit slower than the state-of-the-art algorithms for the problems discussed previously. We won't discuss running times in any detail, but think of roughly $O(mn)$ or $O(n^3)$ as a typical time bound of a smart algorithm for these problems.

3. The algorithms and analysis for these problems follow exactly the same principles that you've been studying in previous lectures. They use optimality conditions, various progress measures, well-chosen invariants, and so on. So you're well-positioned to study deeply these problems and algorithms for them, in another course or on your own. Indeed, if CS261 were a semester-long course, we would cover this material in detail over the next 4-5 lectures. (Alas, it will be time to move on to linear programming.)

2 The Minimum Cost Flow Problem

An instance of the *minimum-cost flow problem* consists of the following ingredients:

- a directed graph $G = (V, E)$;
- a source $s \in V$ and sink $t \in V$;

- a target flow value d;
- a nonnegative capacity u_e for each edge $e \in E$;
- a real-valued cost c_e for each edge $e \in E$.

The goal is to compute a flow f with value d — that is, pushing d units of flow from s to t, subject to the usual conservation and capacity constraints — that minimizes the overall cost

$$\sum_{e \in E} c_e f_e. \tag{1}$$

Note that, for each edge e, we think of c_e as a "per-flow unit" cost, so with f_e units of flow the contribution of edge e to the overall cost is $c_e f_e$.[1]

There are two differences with the maximum flow problem. The important one is that now every edge has a cost. (In maximum flow, one can think of all the costs being 0.) The second difference, which is artificial, is that we specified a specific amount of flow d to send. There are multiple other equivalent formulations of the minimum-cost flow problem. For example, one can ask for the maximum flow with the minimum cost. Alternatively, instead of having a source s and sink t, one can ask for a "circulation" — meaning a flow that satisfies conservation constraints at every vertex of V — with the minimum cost (in the sense of (1)).[2]

Impressively, the minimum-cost flow problem captures three different problems that you've studied as special cases.

1. **Shortest paths.** Suppose you are given a "black box" that quickly does minimum-cost flow computations, and you want to compute the shortest path between some s and some t is a directed graph with edge costs. The black box is expecting a flow value d and edge capacities u_e (in addition to G, s, t, and the edge costs); we just set $d = 1$ and $u_e = 1$ (say) for every edge e. An integral minimum-cost flow in this network will be a shortest path from s to t (why?).

2. **Maximum flow.** Given an instances of the maximum flow problem, we need to define d and edge costs before feeding the input into our minimum-cost flow black box. The edge costs should presumably be set to 0. Then, to compute the maximum flow value, we can just use binary search to find the largest value of d for which the black box returns a feasible solution.

3. **Minimum-cost perfect bipartite matching.** The reduction here is the same as that from maximum-cardinality bipartite matching to maximum flow (Lecture #4) — the edge costs just carry over. The value d should be set to n, the number of vertices on each side of the bipartite graph (why?).

[1] If there is no flow of value d, then an algorithm should report this fact. Note this is easy to check with a single maximum flow computation.

[2] Of course if all edge costs are nonnegative, then the all-zero solution is optimal. But with negative cycles, this is a nontrivial problem.

Problem Set #2 explores various aspects of minimum-cost flows. Like the other problems we've studied, there are nice optimality conditions for minimum-cost flows. First, one extends the notion of a residual network to networks with costs — the only twist is that if an edge (w, v) of the residual network is the reverse edge corresponding to $(v, w) \in E$, then the cost of c_{wv} should be set to $-c_{vw}$. (Which makes sense given that reverse edges correspond to "undo" operations.) Then, a flow with value d is minimum-cost if and only if the corresponding residual network has no negative cycle. This then suggests a simple "cycle-canceling" algorithm, analogous to the Ford-Fulkerson algorithm. Polynomial-time algorithms can be designed using the same ideas we used for maximum flow in Lectures #2 and #3 and Problem Set #1 (blocking flows, push-relabel, scaling, etc.). There are algorithms along these lines with running time roughly $O(mn)$ that are also quite fast in practice. (Theoretically, it is also known how do a bit better.) In general, you should be happy if a problem that you care about reduces to the minimum-cost flow problem.

3 Non-Bipartite Matching

3.1 Maximum-Cardinality Non-Bipartite Matching

In the general (non-bipartite) matching problem, the input is an undirected graph $G = (V, E)$, not necessarily bipartite. The goal to compute a matching (as before, a subset $M \subseteq E$ with no shared endpoints) with the largest cardinality. Recall that the simplest non-bipartite graphs are odd cycles (Figure 2).

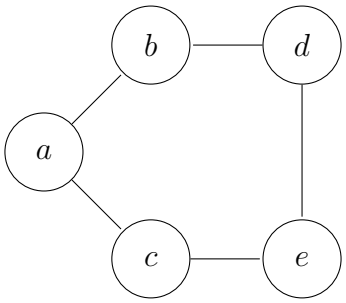

Figure 2: Example of non-bipartite graph: odd cycle.

A priori, it is far from obvious that the general graph matching problem is solvable in polynomial time (as opposed to being NP-hard). It appears to be significantly more difficult than the special case of bipartite matching. For example, there does not seem to be a natural reduction from non-bipartite matching to the maximum flow problem. Once again, we need to develop from scratch algorithms and strategies for correctness,

The non-bipartite matching problem admits some remarkable optimality conditions. For motivation, what is the maximum size of a matching in the graph in Figure 3? There are 16

vertices, so clearly a matching has at most 8 edges. It's easy to exhibit a matching of size 6 (Figure 3), but can we do better?

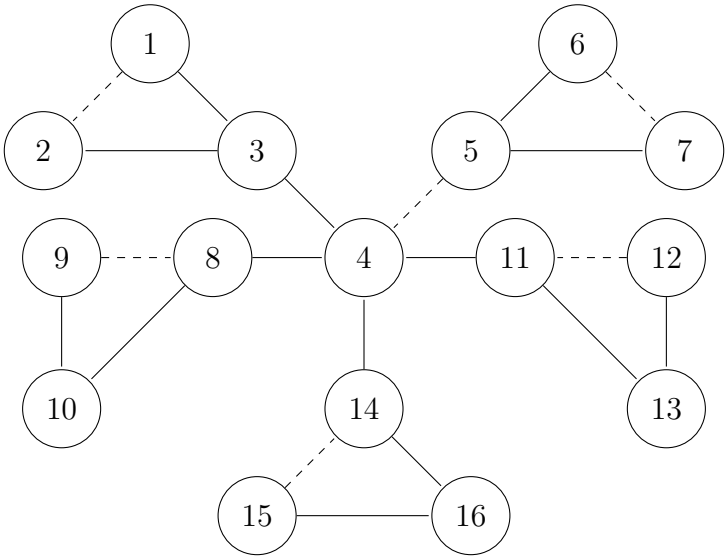

Figure 3: Example graph. A matching of size 6 is denoted by dashed edges.

Here's one way to argue that there is no better matching. In each of the 5 triangles, at most 2 of the 3 vertices can be matched to each other. This leaves at least five vertices, one from each triangle, that, if matched, can only be matched to the center vertex. The center vertex can only be matched to one of these five, so every matching leaves at least four vertices unmatched. This translates to matching at most 12 vertices, and hence containing at most 6 edges.

In general, we have the following.

Lemma 3.1 *In every graph $G = (V, E)$, the maximum cardinality of a matching is at most*

$$\min_{S \subseteq V} \frac{1}{2} \left[|V| - (\mathrm{oc}(S) - |S|) \right], \tag{2}$$

where $\mathrm{oc}(S)$ *denotes the number of odd-size connected components in the graph* $G \setminus S$.

Note that $G \setminus S$ consists of the pieces left over after ripping the vertices in S out of the graph G (Figure 4).

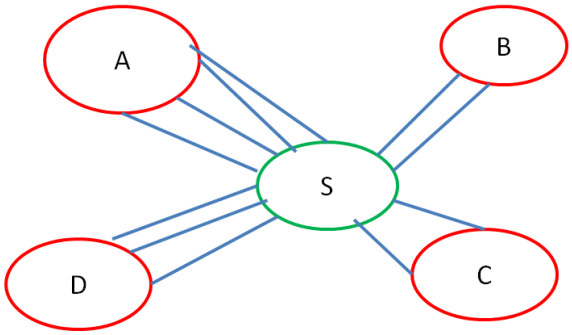

Figure 4: Suppose removing S results in 4 connected components, A, B, C and D. If 3 of them are odd-sized, then $oc(S) = 3$

For example, in the Figure 3, we effectively took S to be the center vertex, so $\text{oc}(S) = 5$ (since $G \setminus S$ is the five triangles) and (2) is $\frac{1}{2}(16-(5-1)) = 6$. The proof is a straightforward generalization of our earlier argument.

Proof of Lemma 3.1: Fix $S \subseteq V$. For every odd-size connected component C of $G \setminus S$, at least one vertex of C is not matched to some other vertex of C. These $\text{oc}(S)$ vertices can only be matched to vertices of S (if two vertices of C_1 and C_2 could be matched to each other, then C_1 and C_2 would not be separate connected components of $G \setminus S$). Thus, every matching leaves at least $\text{oc}(S) - |S|$ vertices unmatched, and hence matches at most $|V| - (\text{oc}(S) - |S|)$ vertices, and hence has at most $\frac{1}{2}(|V| - (\text{oc}(S) - |S|))$ edges. Ranging over all choices of $S \subseteq V$ yields the upper bound in (2). ■

Lemma 3.1 is an analog of the fact that a maximum flow is at most the value of a minimum s-t cut. We can think of (2) as the best upper bound that we can prove if we restrict ourselves to "obvious obstructions" to large matchings. Certainly, if we ever find a matching with size equal to (2), then no other matching could be bigger. But can there be a gap between the maximum size of a matching and the upper bound in (2)? Could there be obstructions to large matchings more subtle than the simple parity argument used to prove Lemma 3.1? One of the more beautiful theorems in combinatorics asserts that there can never be a gap.

Theorem 3.2 (Tutte-Berge Formula) *In Lemma 3.1, equality always holds:*

$$\textit{max matching size} = \min_{S \subseteq V} \frac{1}{2}\left[|V| - (\text{oc}(S) - |S|)\right].$$

The original proof of the Tutte-Berge formula is via induction, and does not seems to lead to an efficient algorithm.[3] In 1965, Edmonds gave the first polynomial-time algorithm for

[3]Tutte characterized the graphs with perfect matchings in the 1940s; in the 1950s, Berge extended this characterization to prove Theorem 3.2.

computing a maximum-cardinality matching.[4] Since the algorithm is guaranteed to produce a matching with cardinality equal to (2), Edmonds' algorithm provides an algorithmic proof of the Tutte-Berge formula.

A key challenge in non-bipartite matching is searching for a good path to use to increase the size of the current matching. Recall that in the Hungarian algorithm (Lecture #5), we used the bipartite assumption to argue that there's no way to encounter both endpoints of an edge in the current matching in the same level of the search tree. But this certainly *can* happen in non-bipartite graphs, even just in the triangle. Edmonds called these odd cycles "blossoms," and his algorithm is often called the "blossom algorithm." When a blossom is encountered, it's not clear how to proceed with the search. Edmonds' idea was to "shrink," meaning contract, a blossom when one is found. The blossom becomes a super-vertex in the new (smaller) graph, and the algorithm can continue. All blossoms are uncontracted in reverse order at the end of the algorithm.[5]

3.2 Minimum-Cost Non-Bipartite Matching

An algorithm designer is never satisfied, always wanting better and more general solutions to computational problems. So it's natural to consider the graph matching problem with both of the complications that we've studied so far: general (non-bipartite) graphs and edge costs.

The minimum-cost non-bipartite matching problem is again polynomial-time solvable, again first proved by Edmonds. From 30,000 feet, the idea to combine the blossom shrinking idea above (which handles non-bipartiteness) with the vertex prices we used in Lecture #5 for the Hungarian algorithm (which handle costs). This is not as easy as it sounds, however — it's not clear what prices should be given to super-vertices when they are created, and such super-vertices may need to be uncontracted mid-algorithm. With some care, however, this idea can be made to work and yields a polynomial-time algorithm.

While polynomial-time solvable, the minimum-cost matching problem is a relatively hard problem within the class P. State-of-the-art algorithms can handle graphs with 100s of vertices, but graphs with 1000s of vertices are already a challenge. From your other computer science courses, you know that in applications one often wants to handle graphs that are bigger than this by 1–6 orders of magnitude. This motivates the design of heuristics for matching that are very fast, even if not fully correct.[6]

For example, the following Kruskal-like greedy algorithm is a natural one to try. For convenience, we work with the equivalent maximum-weight version of the problem (each edge

[4] In this remarkable paper, titled "Paths, Trees, and Flowers," Edmonds defines the class of polynomial-time solvable problems and conjectures that the traveling salesman problem is not in the class (i.e., that $P \neq NP$). Keep in mind that NP-completeness wasn't defined (by Cook and Levin) until 1971.

[5] Your instructor covered this algorithm in last year's CS261, in honor of the algorithm's 50th anniversary. It takes two lectures, however, and has been cut this year in favor of other topics.

[6] In the last part of the course, we explore this idea in the context of approximation algorithms for NP-hard problems. It's worth remembering that for sufficiently large data sets, approximation is the most appropriate solution even for problems that are polynomial-time solvable.

has a weight w_e, the goal is to compute the matching with largest sum of weights).

Greedy Matching Algorithm

sort and rename the edges $E = \{1, 2, \ldots, m\}$ so that $w_1 \geq w_2 \geq \cdots w_m$
$M = \emptyset$
for $i = 1$ to m **do**
    if e_i shares no endpoint with edges in M **then**
        add e_i to M

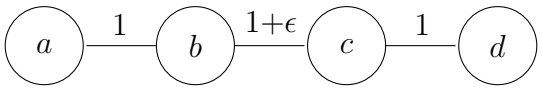

Figure 5: The greedy algorithm picks the edge (b, c), while the optimal matching consists of (a, b) and (c, d).

A simple example (Figure 5) shows that, at least for some graphs, the greedy algorithm can produce a matching with weight only 50% of the maximum possible. On Problem Set #2 you will prove that there are no worse examples — for every (non-bipartite) graph and edge weights, the matching output by the greedy algorithm has weight at least 50% of the maximum possible. Just over the past few years, new matching approximation algorithms have been developed, and it's now possible to get a $(1 - \epsilon)$-approximation in $O(m)$ time, for any constant $\epsilon > 0$ (the hidden constant in the "big-oh" depends on $\frac{1}{\epsilon}$) [**?**].

CS261: A Second Course in Algorithms
Lecture #7: Linear Programming: Introduction and Applications*

Tim Roughgarden†

January 26, 2016

1 Preamble

With this lecture we commence the second part of the course, on *linear programming*, with an emphasis on applications on duality theory.[1] We'll spend a fair amount of quality time with linear programs for two reasons.

First, linear programming is very useful algorithmically, both for proving theorems and for solving real-world problems.

> Linear programming is a remarkable sweet spot between power/generality and computational efficiency.

For example, all of the problems studied in previous lectures can be viewed as special cases of linear programming, and there are also zillions of other examples. Despite this generality, linear programs can be solved efficiently, both in theory (meaning in polynomial time) and in practice (with input sizes up into the millions).

Even when a computational problem that you care about does not reduce directly to solving a linear program, linear programming is an extremely helpful subroutine to have in your pocket. For example, in the fourth and last part of the course, we'll design approximation algorithms for NP-hard problems that use linear programming in the algorithm and/or analysis. In practice, probably most of the cycles spent on solving linear programs is in service of solving integer programs (which are generally NP-hard). State-of-the-art

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

[1]The term "programming" here is not meant in the same sense as computer programming (linear programming pre-dates modern computers). It's in the same spirit as "television programming," meaning assembling a scheduled of planned activities. (See also "dynamic programming".)

algorithms for the latter problem invoke a linear programming solver over and over again to make consistent progress.

Second, linear programming is conceptually useful —- understanding it, and especially LP duality, gives you the "right way" to think about a host of different problems in a simple and consistent way. For example, the optimality conditions we've studied in past lectures (like the max-flow/min-cut theorem and Hall's theorem) can be viewed as special cases of linear programming duality. LP duality is more or less the ultimate answer to the question "how do we know when we're done?" As such, it's extremely useful for proving that an algorithm is correct (or approximately correct).

We'll talk about both these aspects of linear programming at length.

2 How to Think About Linear Programming

2.1 Comparison to Systems of Linear Equations

Once upon a time, in some course you may have forgotten, you learned about linear systems of equations. Such a system consists of m linear equations in real-valued variables $x_1, \ldots, x_n$:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m. \end{aligned}$$

The a_{ij}'s and the b_i's are given; the goal is to check whether or not there are values for the x_j's such that all m constraints are satisfied. You learned at some point that this problem can be solved efficiently, for example by Gaussian elimination. By "solved" we mean that the algorithm returns a feasible solution, or correctly reports that no feasible solution exists.

Here's an issue, though: what about inequalities? For example, recall the maximum flow problem. There are conservation constraints, which are equations and hence OK. But the capacity constraints are fundamentally inequalities. (There is also the constraint that flow values should be nonnegative.) Inequalities are part of the problem description of many other problems that we'd like to solve. The point of linear programming is to solve systems of linear equations *and inequalities*. Moreover, when there are multiple feasible solutions, we would like to compute the "best" one.

2.2 Ingredients of a Linear Program

There is a convenient and flexible language for specifying linear programs, and we'll get lots of practice using it during this lecture. Sometimes it's easy to translate a computational problem into this language, sometimes it takes some tricks (we'll see examples of both).

To specify a linear program, you need to declare what's allowed and what you want.

Ingredients of a Linear Program

1. *Decision variables* $x_1, \ldots, x_n \in \mathbb{R}$.
2. *Linear constraints*, each of the form

$$\sum_{j=1}^{n} a_j x_j \quad (*) \quad b_i,$$

   where (*) could be $\leq$, $\geq$, or $=$.

3. A *linear objective function*, of the form

$$\max \sum_{j=1}^{n} c_j x_j$$

   or

$$\min \sum_{j=1}^{n} c_j x_j.$$

Several comments. First, the a_{ij}'s, b_i's, and c_j's are *constants*, meaning they are part of the input, numbers hard-wired into the linear program (like 5, -1, 10, etc.). The x_j's are free, and it is the job of a linear programming algorithm to figure out the best values for them. Second, when specifying constraints, there is no need to make use of both "$\leq$" and "$\geq$"inequalities — one can be transformed into the other just by multiplying all the coefficients by -1 (the a_{ij}'s and b_i's are allowed to be positive or negative). Similarly, equality constraints are superfluous, in that the constraint that a quantity equals b_i is equivalent to the pair of inequality constraints stating that the quantity is both at least b_i and at most b_i. Finally, there is also no difference between the "min" and "max" cases for the objective function — one is easily converted into the other just by multiplying all the c_j's by -1 (the c_j's are allowed to be positive or negative).

So what's not allowed in a linear program? Terms like x_j^2, $x_j x_k$, $\log(1 + x_j)$, etc. So whenever a decision variable appears in an expression, it is alone, possibly multiplied by a constant (and then summed with other such terms). While these linearity requirements may seem restrictive, we'll see that many real-world problems can be formulated as or well approximated by linear programs.

2.3 A Simple Example

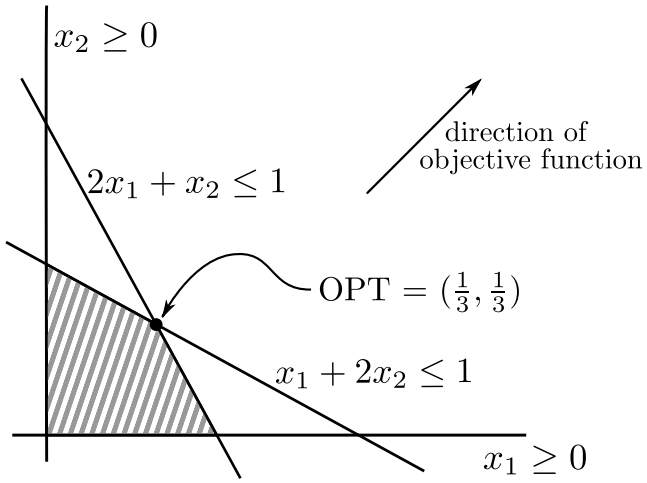

Figure 1: a toy example of linear program.

To make linear programs more concrete and develop your geometric intuition about them, let's look at a toy example. (Many "real" examples of linear programs are coming shortly.) Suppose there are two decision variables x_1 and x_2 — so we can visualize solutions as points (x_1, x_2) in the plane. See Figure 2.3. Let's consider the (linear) objective function of maximizing the sum of the decision variables:

$$\max x_1 + x_2.$$

We'll look at four (linear) constraints:

$$\begin{aligned} x_1 &\geq 0 \\ x_2 &\geq 0 \\ 2x_1 + x_2 &\leq 1 \\ x_1 + 2x_2 &\leq 1. \end{aligned}$$

The first two inequalities restrict feasible solutions to the non-negative quadrant of the plane. The second two inequalities further restrict feasible solutions to lie in the shaded region depicted in Figure 2.3. Geometrically, the objective function asks for the feasible point furthest in the direction of the coefficient vector $(1, 1)$ — the "most northeastern" feasible point. Put differently, the level sets of the objective function are parallel lines running northwest to southeast.[2] Eyeballing the feasible region, this point is $(\frac{1}{3}, \frac{1}{3})$, for an optimal objective function value of $\frac{2}{3}$. This is the "last point of intersection" between a level set of the objective function and the feasible region (as one sweeps from southwest to northeast).

[2]Recall that a *level set* of a function g has the form $\{\mathbf{x} : g(\mathbf{x}) = c\}$, for some constant c. That is, all points in a level set have equal objective function value.

2.4 Geometric Intuition

While it's always dangerous to extrapolate from two or three dimensions to an arbitrary number, the geometric intuition above remains valid for general linear programs, with an arbitrary number of dimensions (i.e., decision variables) and constraints. Even though we can't draw pictures when there are many dimensions, the relevant algebra carries over without any difficulties. Specifically:

1. A linear constraint in n dimensions corresponds to a halfspace in $\mathbb{R}^n$. Thus a feasible region is an intersection of halfspaces, the higher-dimensional analog of a polygon.[3]

2. The level sets of the objective function are parallel $(n-1)$-dimensional hyperplanes in $\mathbb{R}^n$, each orthogonal to the coefficient vector $\mathbf{c}$ of the objective function.

3. The optimal solution is the feasible point furthest in the direction of $\mathbf{c}$ (for a maximization problem) or $-\mathbf{c}$ (for a minimization problem). Equivalently, it is the last point of intersection (traveling in the direction $\mathbf{c}$ or $-\mathbf{c}$) of a level set of the objective function and the feasible region.

4. When there is a unique optimal solution, it is a vertex (i.e., "corner") of the feasible region.

There are a few edge cases which can occur but are not especially important in CS261.

1. There might be no feasible solutions at all. For example, if we add the constraint $x_1 + x_2 \geq 1$ to our toy example, then there are no longer any feasible solutions. Linear programming algorithms correctly detect when this case occurs.

2. The optimal objective function value is unbounded ($+\infty$ for a maximization problem, $-\infty$ for a minimization problem). Note a necessary but not sufficient condition for this case is that the feasible region is unbounded. For example, if we dropped the constraints $2x_1 + x_2 \leq 1$ and $x_1 + 2x_2 \leq 1$ from our toy example, then it would have unbounded objective function value. Again, linear programming algorithms correctly detect when this case occurs.

3. The optimal solution need not be unique, as a "side" of the feasible region might be parallel to the levels sets of the objective function. Whenever the feasible region is bounded, however, there always exists an optimal solution that is a vertex of the feasible region.[4]

[3] A finite intersection of halfspaces is also called a "polyhedron;" in the common special case where the feasible region is bounded, it is called a "polytope."

[4] There are some annoying edge cases for unbounded feasible regions, for example the linear program $\max(x_1 + x_2)$ subject to $x_1 + x_2 = 1$.

3 Some Applications of Linear Programming

Zillions of problems reduce to linear programming. It would take an entire course to cover even just its most famous applications. Some of these applications are conceptually a bit boring but still very important — as early as the 1940s, the military was using linear programming to figure out the most efficient way to ship supplies from factories to where they were needed.[5] Several central problems in computer science reduce to linear programming, and we describe some of these in detail in this section. Throughout, keep in mind that all of these linear programs can be solved efficiently, both in theory and in practice. We'll say more about algorithms for linear programming in a later lecture.

3.1 Maximum Flow

If we return to the definition of the maximum flow problem in Lecture #1, we see that it translates quite directly to a linear program.

1. *Decision variables:* what are we try to solve for? A flow, of course, Specifically, the amount f_e of flow on each edge e. So our variables are just $\{f_e\}_{e \in E}$.

2. *Constraints:* Recall we have conservation constraints and capacity constraints. We can write the former as

$$\underbrace{\sum_{e \in \delta^-(v)} f_e}_{\text{flow in}} - \underbrace{\sum_{e \in \delta^-(v)} f_e}_{\text{flow out}} = 0$$

   for every vertex $v \neq s, t$.[6] We can write the latter as

$$f_e \leq u_e$$

   for every edge $e \in E$. Since decision variables of linear programs are by default allowed to take on arbitrary real values (positive or negative), we also need to remember to add nonnegativity constraints:

$$f_e \geq 0$$

   for every edge $e \in E$. Observe that every one of these $2m + n - 2$ constraints (where $m = |E|$ and $n = |V|$) is linear — each decision variable f_e only appears by itself (with a coefficient of 1 or -1).

3. *Objective function:* We just copy the same one we used in Lecture #1:

$$\max \sum_{e \in \delta^+(s)} f_e.$$

   Note that this is again a linear function.

[5] Note this is well before computer science was field; for example, Stanford's Computer Science Department was founded only in 1965.

[6] Recall that δ^- and δ^+ denote the edges incoming to and outgoing from v, respectively.

3.2 Minimum-Cost Flow

In Lecture #6 we introduced the minimum-cost flow problem. Extending specialized algorithms for maximum flow to generalized algorithms takes non-trivial work (see Problem Set #2 for starters). If we're just using linear programming, however, the generalization is immediate.[7] The main change is in the objective function. As defined last lecture, it is simply

$$\min \sum_{e \in E} c_e f_e,$$

where c_e is the cost of edge e. Since the c_e's are fixed numbers (i.e., part of the input), this is a linear objective function.

For the version of the minimum-cost flow problem defined last lecture, we should also add the constraint

$$\sum_{e \in \delta^+(s)} f_e = d,$$

where d is the target flow value. (One can also add the analogous constraint for t, but this is already implied by the other constraints.)

To further highlight how flexible linear programs can be, suppose we want to impose a lower bound ℓ_e (other than 0) on the amount of flow on each edge e, in addition to the usual upper bound u_e. This is trivial to accommodate in our linear program — just replace "$f_e \geq 0$" by $f_e \geq \ell_e$.[8]

3.3 Fitting a Line

We now consider two less obvious applications of linear programming, to basic problems in machine learning. We first consider the problem of fitting a line to data points (i.e., linear regression), perhaps the simplest non-trivial machine learning problem.

Formally, the input consists of m data points $\mathbf{p}^1, \ldots, \mathbf{p}^m \in \mathbb{R}^d$, each with d real-valued "features" (i.e., coordinates).[9] For example, perhaps $d = 3$, and each data point corresponds to a 3rd-grader, listing the household income, number of owned books, and number of years of parental education. Also part of the input is a "label" $\ell_i \in \mathbb{R}$ for each point $\mathbf{p}^i$.[10] For example, ℓ_i could be the score earned by the 3rd-grader in question on a standardized test. We reiterate that the $\mathbf{p}^i$'s and ℓ_i's are fixed (part of the input), not decision variables.

[7] While linear programming is a reasonable way to solve the maximum flow and minimum-cost flow problems, especially if the goal is to have a "quick and dirty" solution, but the best specialized algorithms for these problems are generally faster.

[8] If you prefer to use flow algorithms, there is a simple reduction from this problem to the special case with $\ell_e = 0$ for all $e \in E$ (do you see it?).

[9] Feel free to take $d = 1$ throughout the rest of the lecture, which is already a practically relevant and computationally interesting case.

[10] This is a canonical "supervised learning" problem, meaning that the algorithm is provided with labeled data.

Informally, the goal is to expresses the ℓ_i as well as possible as a linear function of the $\mathbf{p}_i$'s. That is, the goal is to compute a linear function $h : \mathbb{R}^d \to \mathbb{R}$ such that $h(\mathbf{p}^i) \approx \ell_i$ for every data point i.

The two most common motivations for computing a "best-fit" linear function are prediction and data analysis. In the first scenario, one uses labeled data to identify a linear function h that, at least for these data points, does a good job of predicting the label ℓ_i from the feature values $\mathbf{p}^i$. The hope is that this linear function "generalizes," meaning that it also makes accurate predictions for other data points for which the label is not already known. There is a lot of beautiful and useful theory in statistics and machine learning about when one can and cannot expect a hypothesis to generalize, which you'll learn about if you take courses in those areas. In the second scenario, the goal is to understand the relationship between each feature of the data points and the labels, and also the relationships between the different features. As a simple example, it's clearly interesting to know when one of the d features is much more strongly correlated with the label ℓ^i than any of the others.

We now show that computing the best line, for one definition of "best," reduces to linear programming. Recall that every linear function $h : \mathbb{R}^d \to \mathbb{R}$ has the form

$$h(\mathbf{z}) = \sum_{j=1}^{d} a_j z_j + b$$

for some coefficients $a_1, \ldots, a_d$ and intercept b. (This is one of several equivalent definitions of a linear function.[11] So it's natural to take $a_1, \ldots, a_d, b$ as our decision variables.

What's our objective function? Clearly if the data points are colinear we want to compute the line that passes through all of them. But this will never happen, so we must compromise between how well we approximate different points.

For a given choice of $a_1, \ldots, a_d, b$, define the error on point i as

$$E_i(\mathbf{a}, b) = \left| \underbrace{\left(\sum_{j-1}^{d} a_j p_j^i - b \right)}_{\text{prediction}} - \underbrace{\ell^i}_{\text{"ground truth"}} \right|. \tag{1}$$

Geometrically, when $d = 1$, we can think of each $(\mathbf{p}^i, \ell^i)$ as a point in the plane and (1) is just the vertical distance between this point and the computed line.

In this lecture, we consider the objective function of minimizing the sum of errors:

$$\min_{\mathbf{a},b} \sum_{i=1}^{m} E_i(\mathbf{a}, b). \tag{2}$$

This is not the most common objective for linear regression; more standard is minimizing the squared error $\sum_{i=1}^{m} E_i^2(\mathbf{a}, b)$. While our motivation for choosing (2) is primarily pedagogical,

[11] Sometimes people use "linear function" to mean the special case where $b = 0$, and "affine function" for the case of arbitrary b.

this objective is reasonable and is sometimes used in practice. The advantage over squared error is that it is more robust to outliers. Squaring the error of an outlier makes it a squeakier wheel. That is, a stray point (e.g., a faulty sensor or data entry error) will influence the line chosen under (2) less that it would with the squared error objective (Figure 2).[12]

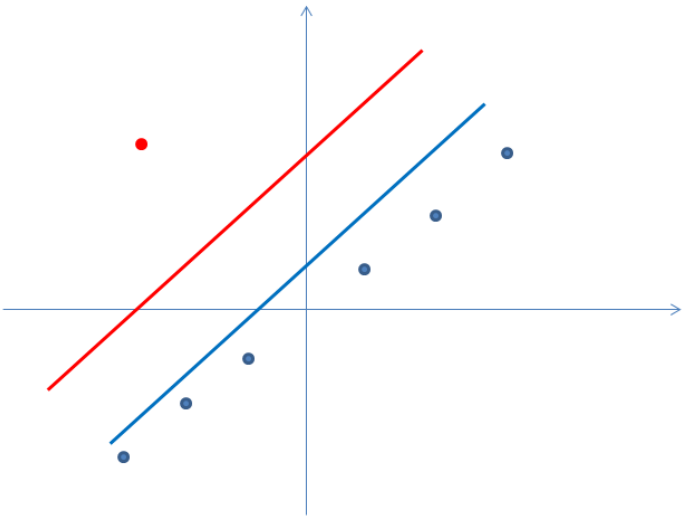

Figure 2: When there exists an outlier (red point), using the objective function defined in (2) causes the best-fit line not to "stray" as far away from the non-outliers (blue line) as when using the squared error objective (red line), because the squared error objective would penalize more greatly when the chosen line is far from the outlier.

Consider the problem of choosing $\mathbf{a}, b$ to minimize (2). (Since the a_j's and b can be anything, there are no constraints.) The problem: *this is not a linear program.* The source of nonlinearity is the absolute value sign $|\cdot|$ in (1). Happily, in this case and many others, absolute values can be made linear with a simple trick.

The trick is to introduce extra variables $e_1, \ldots, e_m$, one per data point. The intent is for e_i to take on the value $E_i(\mathbf{a}, b)$. Motivated by the identify $|x| = \max\{x, -x\}$, we add two constraints for each data point:

$$e_i \geq \left(\sum_{j=1}^{d} a_j p_j^i - b\right) - \ell^i \tag{3}$$

and

$$e_i \geq -\left[\left(\sum_{j=1}^{d} a_j p_j^i - b\right) - \ell^i\right]. \tag{4}$$

[12]Squared error can be minimized efficiently using an extension of linear programming known as *convex programming*. (For the present "ordinary least squares" version of the problem, it can even be solved analytically, in closed form.) We may discuss convex programming in a future lecture.

We change the objective function to

$$\min \sum_{i=1}^{m} e_i. \tag{5}$$

Note that optimizing (5) subject to all constraints of the form (3) and (4) is a linear program, with decision variables $e_1, \ldots, e_m, a_1, \ldots, a_d, b$.

The key point is: at an optimal solution to this linear program, it must be that $e_i = E_i(\mathbf{a}, b)$ for every data point i. Feasibility of the solution already implies that $e_i \geq E_i(\mathbf{a}, b)$ for every i. And if $e_i > E_i(\mathbf{a}, b)$ for some i, then we can decrease e_i slightly, so that (3) and (4) still hold, to obtain a superior feasible solution. We conclude that an optimal solution to this linear program represents the line minimizing the sum of errors (2).

3.4 Computing a Linear Classifier

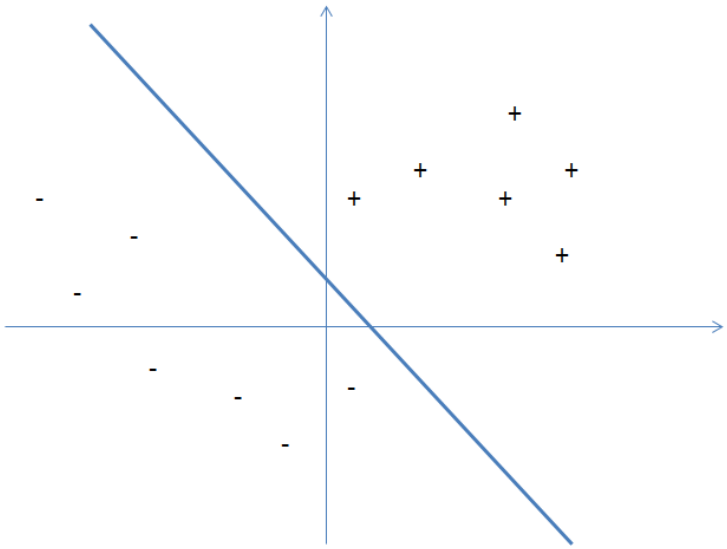

Figure 3: We want to find a linear function that separates the positive points (plus signs) from the negative points (minus signs)

Next we consider a second fundamental problem in machine learning, that of learning a linear classifier.[13] While in Section 3.3 we sought a real-valued function (from $\mathbb{R}^d$ to $\mathbb{R}$), here we're looking for a binary function (from $\mathbb{R}^d$ to $\{0, 1\}$). For example, data points could represent images, and we want to know which ones contain a cat and which ones don't.

Formally, the input consists of m "positive" data points $\mathbf{p}^1, \ldots, \mathbf{p}^m \in \mathbb{R}^d$ and m' "negative" data points $\mathbf{q}^1, \ldots, \mathbf{q}^{m'}$. In the terminology of the previous section, all of the labels

[13] Also called halfspaces, perceptrons, linear threshold functions, etc.

are "1" or "0," and we have partitioned the data accordingly. (So this is again a supervised learning problem.)

The goal is to compute a linear function $h(\mathbf{z}) = \sum_{j=1} a_j z_j + b$ (from $\mathbb{R}^d$ to $\mathbb{R}$) such that

$$h(\mathbf{p}^i) > 0 \tag{6}$$

for all positive points and

$$h(\mathbf{q}^i) < 0 \tag{7}$$

for all negative points. Geometrically, we are looking for a hyperplane in $\mathbb{R}^d$ such all positive points are on one side and all negative points on the other; the coefficients $\mathbf{a}$ specify the normal vector of the hyperplane and the intercept b specifies its shift. See Figure 3. Such a hyperplane can be used for predicting the labels of other, unlabeled points (check which side of the hyperplane it is on and predict that it is positive or negative, accordingly). If there is no such hyperplane, an algorithm should correctly report this fact.

This problem almost looks like a linear program by definition. The only issue is that the constraints (6) and (7) are strict inequalities, which are not allowed in linear programs. Again, the simple trick of adding an extra decision variable solves the problem. The new decision variable δ represents the "margin" by which the hyperplane satisfies (6) and (7). So we

$$\max \delta$$

subject to

$$\sum_{j=1}^{d} a_j p_j^i + b - \delta \geq 0 \qquad \text{for all positive points } \mathbf{p}^i$$

$$\sum_{j=1}^{d} a_j q_j^i + b + \delta \leq 0 \qquad \text{for all negative points } \mathbf{q}^i,$$

which is a linear program with decision variables $\delta, a_1, \ldots, a_d, b$. If the optimal solution to this linear program has strictly positive objective function value, then the values of the variables $a_1, \ldots, a_d, b$ define the desired separating hyperplane. If not, then there is no such hyperplane. We conclude that computing a linear classifier reduces to linear programming.

3.5 Extension: Minimizing Hinge Loss

There is an obvious issue with the problem setup in Section 3.4: what if the data set is not as nice as the picture in Figure 3, and there is no separating hyperplane? This is usually the case in practice, for example if the data is noisy (as it always is). Even if there's no perfect hyperplane, we'd still like to compute something that we can use to predict the labels of unlabeled points.

We outline two ways to extend the linear programming approach in Section 3.4 to handle non-separable data.[14] The first idea is to compute the hyperplane that minimizes some notion

[14] In practice, these two approaches are often combined.

of "classification error." After all, this is what we did in Section 3.3, where we computed the line minimizing the sum of the errors.

Probably the most natural plan would be to compute the hyperplane that puts the fewest number of points on the wrong side of the hyperplane — to minimize the number of inequalities of the form (6) or (7) that are violated. Unfortunately, this is an NP-hard problem, and one typically uses notions of error that are more computationally tractable. Here, we'll discuss the widely used notion of *hinge loss*.

Let's say that in a perfect world, we would like a linear function h such that

$$h(\mathbf{p}^i) \geq 1 \tag{8}$$

for all positive points $\mathbf{p}^i$ and

$$h(\mathbf{q}^i) \leq -1 \tag{9}$$

for all negative points $\mathbf{q}^i$; the "1" here is somewhat arbitrary, but we need to pick some constant for the purposes of normalization. The *hinge loss* incurred by a linear function h on a point is just the extent to which the corresponding inequality (8) or (9) fails to hold. For a positive point $\mathbf{p}^i$, this is $\max\{1 - h(\mathbf{p}^i), 0\}$; for a negative point $\mathbf{q}^i$, this is $\max\{1 + h(\mathbf{p}^i), 0\}$. Note that taking the maximum with zero ensures that we don't reward a linear function for classifying a point "extra-correctly." Geometrically, when $d = 1$, the hinge loss is the vertical distance that a data point would have to travel to be on the correct side of the hyperplane, with a "buffer" of 1 between the point and the hyperplane.

Computing the linear function that minimizes the total hinge loss can be formulated as a linear program. While hinge loss is not linear, it is just the maximum of two linear functions. So by introducing one extra variable and two extra constraints per data point, just like in Section 3.3, we obtain the linear program

$$\min \sum_{i=1}^{m} e_i$$

subject to:

$$\begin{aligned}
e_i &\geq 1 - \left(\sum_{j=1}^{d} a_j p_j^i + b \right) \qquad \text{for every positive point } \mathbf{p}^i \\
e_i &\geq 1 + \left(\sum_{j=1}^{d} a_j q_j^i + b \right) \qquad \text{for every negative point } \mathbf{q}^i \\
e_i &\geq 0 \qquad \text{for every point}
\end{aligned}$$

in the decision variables $e_1, \ldots, e_m, a_1, \ldots, a_d, b$.

3.6 Extension: Increasing the Dimension

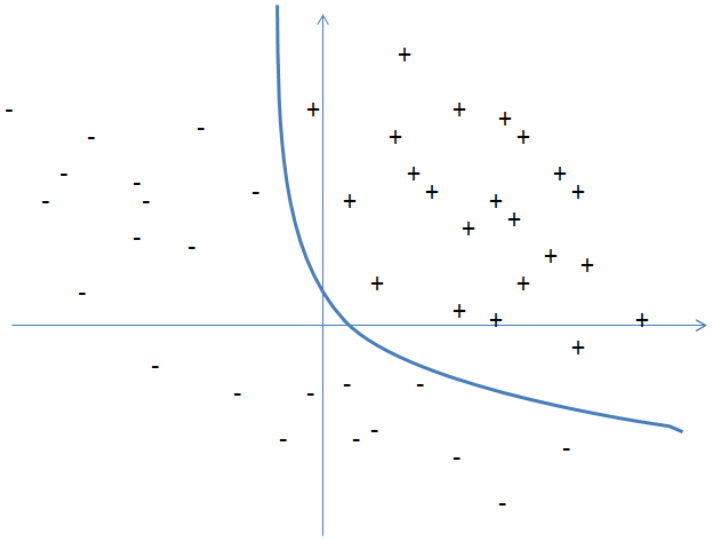

Figure 4: The points are not linearly separable, but they can be separated by a quadratic line.

A second approach to dealing with non-linearly-separable data is to use nonlinear boundaries. E.g., in Figure 4, the positive and negative points cannot be separated perfectly by any line, but they can be separated by a relatively simple boundary (e.g., of a quadratic function). But how we can allow nonlinear boundaries while retaining the computationally tractability of our previous solutions?

The key idea is to generate extra features (i.e., dimensions) for each data point. That is, for some dimension $d' \geq d$ and some function $\varphi : \mathbb{R}^d \to \mathbb{R}^{d'}$, we map each $\mathbf{p}^i$ to $\varphi(\mathbf{p}^i)$ and each $\mathbf{q}_i$ to $\varphi(\mathbf{q}^i)$. We'll then try to separate the images of these points in d'-dimensional space using a linear function.[15]

A concrete example of such a function φ is the map

$$(z_1, \ldots, z_d) \mapsto (z_1, \ldots, z_d, z_1^2, \ldots, z_d^2, z_1 z_2, z_1 z_3, \ldots, z_{d-1} z_d); \tag{10}$$

that is, each data point is expanded with all of the pairwise products of its features. This map is interesting even when $d = 1$:

$$z \mapsto (z, z^2). \tag{11}$$

Our goal is now to compute a linear function in the expanded space, meaning coefficients

[15] This is the basic idea behind "support vector machines;" see CS229 for much more on the topic.

$a_1, \ldots, a_{d'}$ and an intercept b, that separates the positive and negative points:

$$\sum_{i=1}^{d'} a_j \cdot \varphi(\mathbf{p}^i)_j + b > 0 \tag{12}$$

for all positive points and

$$\sum_{i=1}^{d'} a_j \cdot \varphi(\mathbf{q}^i)_j + b < 0 \tag{13}$$

for all negative points. Note that if the new feature set includes all of the original features, as in (10), then every hyperplane in the original d-dimensional space remains available in the expanded space (just set $a_{d+1}, a_{d+2}, \ldots, a_{d'} = 0$). But there are also many new options, and hence it is more likely that there is way to perfectly separate the (images under φ of the) data points. For example, even with $d = 1$ and the map (11), linear functions in the expanded space have the form $h(z) = a_1 z^2 + a_2 z + b$, which is a quadratic function in the original space.

We can think of the map φ as being applied in a preprocessing step. Then, the resulting problem of meeting all the constraints (12) and (13) is exactly the problem that we already solved in Section 3.4. The resulting linear program has decision variables $\delta, a_1, \ldots, a_{d'}, b$ ($d' + 2$ in all, up from $d + 2$ in the original space).[16]

[16]The magic of support vector machines is that, for many maps φ including (10) and (11), and for many methods of computing a separating hyperplane, the computation required scales only with the original dimension d, even if the expanded dimension d' is radically larger. This is known as the "kernel trick;" see CS229 for more details.

CS261: A Second Course in Algorithms
Lecture #8: Linear Programming Duality (Part 1)*

Tim Roughgarden†

January 28, 2016

1 Warm-Up

This lecture begins our discussion of linear programming duality, which is the really the heart and soul of CS261. It is the topic of this lecture, the next lecture, and (as will become clear) pretty much all of the succeeding lectures as well.

Recall from last lecture the ingredients of a linear program: decision variables, linear constraints (equalities or inequalities), and a linear objective function. Last lecture we saw that lots of interesting problems in combinatorial optimization and machine learning reduce to linear programming.

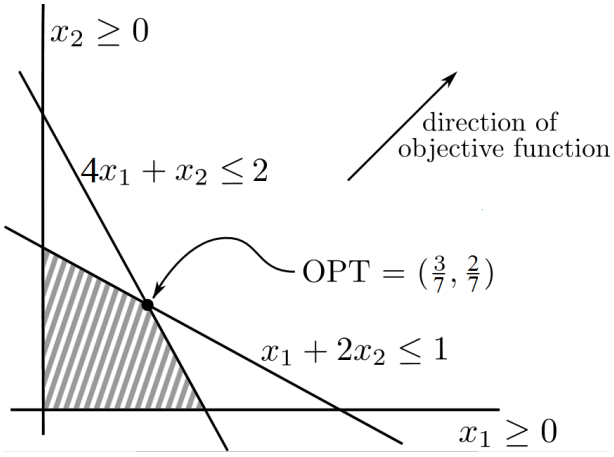

Figure 1: A toy example to illustrate duality.

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

To start getting a feel for linear programming duality, let's begin with a toy example. It is a minor variation on our toy example from last time. There are two decision variables x_1 and x_2 and we want to

$$\max x_1 + x_2 \tag{1}$$

subject to

$$4x_1 + x_2 \leq 2 \tag{2}$$
$$x_1 + 2x_2 \leq 1 \tag{3}$$
$$x_1 \geq 0 \tag{4}$$
$$x_2 \geq 0. \tag{5}$$

(Last lecture, the first constraint of our toy example read $2x_1 + x_2 \leq 1$; everything else is the same.)

Like last lecture, we can solve this LP just by eyeballing the feasible region (Figure 1) and searching for the "most northeastern" feasible point, which in this case is the vertex (i.e., "corner") at $(\frac{3}{7}, \frac{2}{7})$. Thus the optimal objective function value if $\frac{5}{7}$.

When we go beyond three dimensions (i.e., decision variables), it seems hopeless to solve linear programs by inspection. With a general linear program, even if we are handed on a silver platter an allegedly optimal solution, how do we know that it is really is optimal?

Let's try to answer this question at least in our toy example. What's an easy and convincing proof that the optimal objective function value of the linear program can't be too large? For starters, for any feasible point (x_1, x_2), we certainly have

$$\underbrace{x_1 + x_2}_{\text{objective}} \leq 4x_1 + x_2 \leq \underbrace{2}_{\text{upper bound}},$$

with the first inequality following from $x_1 \geq 0$ and the second from the first constraint. We can immediately conclude that the optimal value of the linear program is at most 2. But actually, it's obvious that we can do better by using the second constraint instead:

$$x_1 + x_2 \leq x_1 + 2x_2 \leq 1,$$

giving us a better (i.e., smaller) upper bound of 1. Can we do better? There's no reason we need to stop at using just one constraint at a time, and are free to blend two or more constraints. The best blending takes $\frac{1}{7}$ of the first constraint and $\frac{3}{7}$ of the second to give

$$x_1 + x_2 \leq \tfrac{1}{7}\underbrace{(4x_1 + x_2)}_{\leq 2 \text{ by } (2)} + \tfrac{3}{7}\underbrace{(x_1 + 2x_2)}_{\leq 1 \text{ by } (3)} \leq \frac{1}{7} \cdot 2 + \frac{3}{7} \cdot 1 = \frac{5}{7}. \tag{6}$$

(The first inequality actually holds with equality, but we don't need the stronger statement.) So this is a convincing proof that the optimal objective function value is at most $\frac{5}{7}$. Given the feasible point $(\frac{3}{7}, \frac{2}{7})$ that actually does realize this upper bound, we can conclude that $\frac{5}{7}$ really is the optimal value for the linear program.

Summarizing, for the linear program (1)–(5), there is a quick and convincing proof that the optimal solution has value at least $\frac{5}{7}$ (namely, the feasible point $(\frac{3}{7}, \frac{2}{7})$) and also such a proof that the optimal solution has value at most $\frac{5}{7}$ (given in (6)). This is the essence of linear programming duality.

2 The Dual Linear Program

We now generalize the ideas of the previous section. Consider an arbitrary linear program (call it (P)) of the form

$$\max \sum_{j=1}^{n} c_j x_j \tag{7}$$

subject to

$$\sum_{j=1}^{n} a_{1j} x_j \leq b_1 \tag{8}$$

$$\sum_{j=1}^{n} a_{2j} x_j \leq b_2 \tag{9}$$

$$\vdots \leq \vdots \tag{10}$$

$$\sum_{j=1}^{n} a_{mj} x_j \leq b_m \tag{11}$$

$$x_1, \ldots, x_n \geq 0. \tag{12}$$

This linear program has n nonnegative decision variables $x_1, \ldots, x_n$ and m constraints (not counting the nonnegativity constraints). The a_{ij}'s, b_i's, and c_j's are all part of the input (i.e., fixed constants).[1]

You may have forgotten your linear algebra, but it's worth paging the basics back in when learning linear programming duality. It's very convenient to write linear programs in matrix-vector notation. For example, the linear program above translates to the succinct description

$$\max \mathbf{c}^T \mathbf{x}$$

subject to

$$\mathbf{A}\mathbf{x} \leq \mathbf{b}$$
$$\mathbf{x} \geq 0,$$

[1]Remember that different types of linear programs are easily transformed to each other. A minimization objective can be turned into a maximization objective by multiplying all c_j's by -1. An equality constraint can be simulated by two inequality constraints. An inequality constraint can be flipped by multiplying by -1. Real-valued decision variables can be simulated by the difference of two nonnegative decision variables. An inequality constraint can be turned into an equality constraint by adding an extra "slack" variable.

where $\mathbf{c}$ and $\mathbf{x}$ are n-vectors, $\mathbf{b}$ is an m-vector, $\mathbf{A}$ is an $m \times n$ matrix (of the a_{ij}'s), and the inequalities are componentwise.

Remember our strategy for deriving upper bounds on the optimal objective function value of our toy example: take a nonnegative linear combination of the constraints that (componentwise) dominates the objective function. In general, for the above linear program with m constraints, we denote by $y_1, \ldots, y_m \geq 0$ the corresponding multipliers that we use. The goal of dominating the objective function translates to the conditions

$$\sum_{i=1}^{m} y_i a_{ij} \geq c_j \tag{13}$$

for each objective function coefficient (i.e. for $j = 1, 2, \ldots, m$). In matrix notation, we are interested in nonnegative m-vectors $\mathbf{y} \geq 0$ such that

$$\mathbf{A}^T \mathbf{y} \geq \mathbf{c};$$

note the sum in (13) is over the rows i of A, which corresponds to an inner product with the jth column of $\mathbf{A}$, or equivalently with the jth row of $\mathbf{A}^T$.

By design, every such choice of multipliers $y_1, \ldots, y_m$ implies an upper bound on the optimal objective function value of the linear program (7)–(12): for every feasible solution $(x_1, \ldots, x_n)$,

$$\underbrace{\sum_{j=1}^{n} c_j x_j}_{\mathbf{x}\text{'s obj fn}} \leq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} y_i a_{ij} \right) x_j \tag{14}$$

$$= \sum_{i=1}^{m} y_i \cdot \left(\sum_{j=1}^{n} a_{ij} x_j \right) \tag{15}$$

$$\leq \underbrace{\sum_{i=1}^{m} y_i b_i}_{\text{upper bound}}. \tag{16}$$

In this derivation, inequality (14) follows from the domination condition in (13) and the nonnegativity of $x_1, \ldots, x_n$; equation (15) follows from reversing the order of summation; and inequality (16) follows from the feasibility of $\mathbf{x}$ and the nonnegativity of $y_1, \ldots, y_m$.

Alternatively, the derivation may be more transparent in matrix-vector notation:

$$\mathbf{c}^T \mathbf{x} \leq (\mathbf{A}^T \mathbf{y})^T \mathbf{x} = \mathbf{y}^T (\mathbf{A}\mathbf{x}) \leq \mathbf{y}^t \mathbf{b}.$$

The upshot is that, whenever $\mathbf{y} \geq 0$ and (13) holds,

$$\text{OPT of (P)} \leq \sum_{i=1}^{m} b_i y_i.$$

In our toy example of Section 1, the first upper bound of 2 corresponds to taking $y_1 = 1$ and $y_2 = 0$. The second upper bound of 1 corresponds to $y_1 = 0$ and $y_2 = 1$ The final upper bound of $\frac{5}{7}$ corresponds to $y_1 = \frac{1}{7}$ and $y_2 = \frac{3}{7}$.

Our toy example illustrates that there can be many different ways of choosing the y_i's, and different choices lead to different upper bounds on the optimal value of the linear program (P). Obviously, the most interesting of these upper bounds is the tightest (i.e., smallest) one. So we really want to range over all possible $\mathbf{y}$'s and consider the minimum such upper bound.[2]

Here's the key point: *the tightest upper bound on OPT is itself the optimal solution to a linear program.* Namely:

$$\min \sum_{i=1}^{m} b_i y_i$$

subject to

$$\begin{aligned}
\sum_{i=1}^{m} a_{i1} y_i &\geq c_1 \\
\sum_{i=1}^{m} a_{i2} y_i &\geq c_2 \\
\vdots &\geq \vdots \\
\sum_{i=1}^{m} a_{in} y_i &\geq c_n \\
y_1, \ldots, y_m &\geq 0.
\end{aligned}$$

Or, in matrix-vector form:

$$\min \mathbf{b}^T \mathbf{y}$$

subject to

$$\begin{aligned}
\mathbf{A}^T \mathbf{y} &\geq \mathbf{c} \\
\mathbf{y} &\geq 0.
\end{aligned}$$

This linear program is called the *dual* to (P), and we sometimes denote it by (D).

For example, to derive the dual to our toy linear program, we just swap the objective and the right-hand side and take the transpose of the constraint matrix:

$$\min 2y_1 + y_2$$

[2]For an analogy, among all s-t cuts, each of which upper bounds the value of a maximum flow, the minimum cut is the most interesting one (Lecture #2). Similarly, in the Tutte-Berge formula (Lecture #5), we were interested in the tightest (i.e., minimum) upper bound of the form $|V| - (\text{oc}(S) - |S|)$, over all choices of the set S.

subject to

$$\begin{aligned} 4y_1 + y_2 &\geq 1 \\ y_1 + 2y_2 &\geq 1 \\ y_1, y_2 &\geq 0. \end{aligned}$$

The objective function values of the feasible solutions $(1, 0)$, $(0, 1)$, and $(\frac{1}{7}, \frac{3}{7})$ (of 2, 1, and $\frac{5}{7}$) correspond to our three upper bounds in Section 1.

The following important result follows from the definition of the dual and the derivation (14)–(16).

Theorem 2.1 (Weak Duality) *For every linear program of the form (P) and corresponding dual linear program (D),*

$$\textit{OPT value for (P)} \leq \textit{OPT value for (D)}. \tag{17}$$

(Since the derivation (14)–(15) applies to any pair of feasible solutions, it holds in particular for a pair of optimal solutions.) Next lecture we'll discuss *strong duality*, which asserts that (17) always holds with equality (as long as both (P) and (D) are feasible).

3 Duality Example #1: Max-Flow/Min-Cut Revisited

This section brings linear programming duality back down to earth by relating it to an old friend, the maximum flow problem. Last lecture we showed how this problem translates easily to a linear program. This lecture, for convenience, we will use a different linear programming formulation. The new linear program is much bigger but also simpler, so it is easier to take and interpret its dual.

3.1 The Primal

The idea is to work directly with path decompositions, rather than flows. So the decision variables have the form f_P, where P is an s-t path. Let $\mathcal{P}$ denote the set of all such paths. The benefit of working with paths is that there is no need to explicitly state the conservation constraints. We do still have the capacity (and nonnegativity) constraints, however.

$$\max \sum_{P \in \mathcal{P}} f_P \tag{18}$$

subject to

$$\underbrace{\sum_{P \in \mathcal{P} : e \in P} f_P}_{\text{total flow on } e} \leq u_e \qquad \text{for all } e \in E \tag{19}$$

$$f_P \geq 0 \qquad \text{for all } P \in \mathcal{P}. \tag{20}$$

Again, call this (P). The optimal value to this linear program is the same as that of the linear programming formulation of the maximum flow problem given last lecture. Every feasible solution to (18)–(20) can be transformed into one of equal value for last lecture's LP, just by setting f_e equal to the left-hand side of (19) for each e. For the reverse direction, one takes a path decomposition (Problem Set #1). See Exercise Set #4 for details.

3.2 The Dual

The linear program (18)–(20) conforms to the format covered in Section 2, so it has a well-defined dual. What is it? It's usually easier to take the dual in matrix-vector notation:

$$\max \mathbf{1}^T \mathbf{f}$$

subject to

$$\begin{aligned} \mathbf{A}\mathbf{f} &\leq \mathbf{u} \\ \mathbf{f} &\geq 0, \end{aligned}$$

where the vector $\mathbf{f}$ is indexed by the paths $\mathcal{P}$, $\mathbf{1}$ stands for the ($|\mathcal{P}|$-dimensional) all-ones vector, $\mathbf{u}$ is indexed by E, and $\mathbf{A}$ is a $\mathcal{P} \times E$ matrix. Then, the dual (D) has decision variables indexed by E (denoted $\{\ell_e\}_{e \in E}$ for reasons to become clear) and is

$$\min \mathbf{u}^T \ell$$

$$\begin{aligned} \mathbf{A}^T \ell &\geq \mathbf{1} \\ \ell &\geq 0. \end{aligned}$$

Typically, the hardest thing about understanding a dual is interpreting what the transpose operation on the constraint matrix ($\mathbf{A} \mapsto \mathbf{A}^T$) is doing. By definition, each row (corresponding to an edge e) of A has a 1 in the column corresponding to a path P if $e \in P$, and 0 otherwise. So an entry a_{eP} of $\mathbf{A}$ is 1 if $e \in P$ and 0 otherwise. In the column of $\mathbf{A}$ (and hence row of $\mathbf{A}^T$) corresponding to a path P, there is a 1 in each row corresponding an edge e of P (and zeroes in the other rows).

Now that we understand $\mathbf{A}^T$, we can unpack the dual and write it as

$$\min \sum_{e \in E} u_e \ell_e$$

subject to

$$\begin{aligned} \sum_{e \in P} \ell_e &\geq 1 \qquad \text{for all } P \in \mathcal{P} \\ \ell_e &\geq 0 \qquad \text{for all } e \in E. \end{aligned} \tag{21}$$

3.3 Interpretation of Dual

The duals of natural linear programs are often meaningful in their own right, and this one is a good example. A key observation is that every s-t cut corresponds to a feasible solution to this dual linear program. To see this, fix a cut (A, B), with $s \in A$ and $t \in B$, and set

$$\ell_e = \begin{cases} 1 & \text{if } e \in \delta^+(A) \\ 0 & \text{otherwise.} \end{cases}$$

(Recall that $\delta^+(A)$ denotes the edges sticking out of A, with tail in A and head in B; see Figure 2.) To verify the constraints (21) and hence feasibility for the dual linear program, note that every s-t path must cross the cut (A, B) as some point (since it starts in A and ends in B). Thus every s-t path has at least one edge e with $\ell_e = 1$, and (21) holds. The objective function value of this feasible solution is

$$\sum_{e \in E} u_e \ell_e = \sum_{e \in \delta^+(A)} u_e = \text{capacity of } (A, B),$$

where the second equality is by definition (recall Lecture #2).

s-t-cuts correspond to one type of feasible solution to this dual linear program, where every decision variable is set to either 0 or 1. Not all feasible solutions have this property: any assignment of nonnegative "lengths" ℓ_e to the edges of G satisfying (21) is feasible. Note that (21) is equivalent to the constraint that the shortest-path distance from s to t, with respect to the edge lengths $\{\ell_e\}_{e \in E}$, is at least 1.[3]

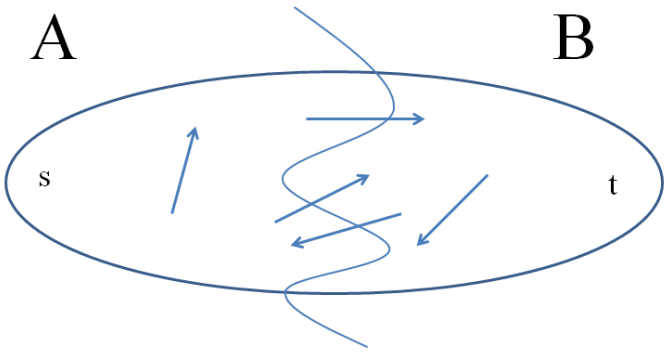

Figure 2: $\delta^+(A)$ denotes the two edges that point from A to B.

3.4 Relation to Max-Flow/Min-Cut

Summarizing, we have shown that

$$\text{max flow value} = \text{OPT of (P)} \leq \text{OPT of (D)} \leq \text{min cut value}. \tag{22}$$

[3]To give a simple example, in the graph $s \to v \to t$, one feasible solution assigns $\ell_{sv} = \ell_{vt} = \frac{1}{2}$. If the edge (s, v) and (v, t) have the same capacity, then this is also an optimal solution.

The first equation is just the statement the maximum flow problem can be formulated as the linear program (P). The first inequality is weak duality. The second inequality holds because the feasible region of (D) includes all (0-1 solutions corresponding to) s-t cuts; since it minimizes over a superset of the s-t cuts, the optimal value can only be less than that of the minimum cut.

In Lecture #2 we used the Ford-Fulkerson algorithm to prove the maximum flow/minimum cut theorem, stating that there is never a gap between the maximum flow and minimum cut values. So the first and last terms of (22) are equal, which means that both of the inequalities are actually equalities. The fact that

$$\text{OPT of (P)} = \text{OPT of (D)}$$

is interesting because it proves a natural special case of strong duality, for flow linear programs and their duals. The fact that

$$\text{OPT of (D)} = \text{min cut value}$$

is interesting because it implies that the linear program (D), despite allowing fractional solutions, always admits an optimal solution in which each decision variable is either 0 or 1.

3.5 Take-Aways

The example in this section illustrates three general points.

1. The duals of natural linear programs are often natural in their own right.
2. Strong duality. (We verified it in a special case, and will prove it in general next lecture.)
3. Some natural linear programs are guaranteed to have integral optimal solutions.

4 Recipe for Taking Duals

Section 2 defines the dual linear program for primal linear programs of a specific form (maximization objective, inequality constraints, and nonnegative decision variables). As we've mentioned, different types of linear programs are easily converted to each other. So one perfectly legitimate way to take the dual of an arbitrary linear program is to first convert it into the form in Section 2 and then apply that definition. But it's more convenient to be able to take the dual of any linear program directly, using a general recipe.

The high-level points of the recipe are familiar: dual variables correspond to primal constraints, dual constraints correspond to primal variables, maximization and minimization get exchanged, the objective function and right-hand side get exchanged, and the constraint matrix gets transposed. The details concern the different type of constraints (inequality vs. equality) and whether or not decision variables are nonnegative.

Here is the general recipe for maximization linear programs:

Primal	Dual
variables $x_1, \ldots, x_n$	n constraints
m constraints	variables $y_1, \ldots, y_m$
objective function $\mathbf{c}$	right-hand side $\mathbf{c}$
right-hand side $\mathbf{b}$	objective function $\mathbf{b}$
$\max \mathbf{c}^T\mathbf{x}$	$\min \mathbf{b}^T\mathbf{y}$
constraint matrix $\mathbf{A}$	constraint matrix $\mathbf{A}^T$
ith constraint is "$\leq$"	$y_i \geq 0$
ith constraint is "$\geq$"	$y_i \leq 0$
ith constraint is "$=$"	$y_i \in \mathbb{R}$
$x_j \geq 0$	jth constraint is "$\geq$"
$x_j \leq 0$	jth constraint is "$\leq$"
$x_j \in \mathbb{R}$	jth constraint is "$=$"

For minimization linear programs, we define the dual as the reverse operation (from the right column to the left). Thus, by definition, the dual of the dual is the original primal.

5 Weak Duality

The above recipe allows you to take duals in a mechanical way, without thinking about it. This can be very useful, but don't forget the true meaning of the dual (which holds in all cases): *feasible dual solutions correspond to bounds on the best-possible primal objective function value (derived from taking linear combinations of the constraints), and the optimal dual solution is the tightest-possible such bound.*

If you remember the meaning of duals, then it's clear that weak duality holds in all cases (essentially by definition).[4]

Theorem 5.1 (Weak Duality) *For every maximization linear program (P) and corresponding dual linear program (D),*

$$\textit{OPT value for (P)} \leq \textit{OPT value for (D)};$$

for every minimization linear program (P) and corresponding dual linear program (D),

$$\textit{OPT value for (P)} \geq \textit{OPT value for (D)}.$$

Weak duality can be visualized as in Figure 3. Strong duality also holds in all cases; see next lecture.

[4]Math classes often teach mathematical definitions as if they fell from the sky. This is not representative of how mathematics actually develops. Typically, definitions are reverse engineered so that you get the "right" theorems (like weak/strong duality).

Figure 3: visualization of weak duality. X represents feasible solutions for P while O represents feasible solutions for D.

Weak duality already has some very interesting corollaries.

Corollary 5.2 *Let (P),(D) be a primal-dual pair of linear programs.*

(a) If the optimal objective function value of (P) is unbounded, then (D) is infeasible.

(b) If the optimal objective function value of (D) is unbounded, then (P) is infeasible.

(c) If $\mathbf{x}, \mathbf{y}$ *are feasible for (P),(D) and* $\mathbf{c}^T\mathbf{x} = \mathbf{y}^T\mathbf{b}$*, then both* $\mathbf{x}$ *and* $\mathbf{y}$ *are both optimal.*

Parts (a) and (b) hold because any feasible solution to the dual of a linear program offers a bound on the best-possible objective function value of the primal (so if there is no such bound, then there is no such feasible solution). The hypothesis in (c) asserts that Figure 3 contains an "x" and an "o" that are superimposed. It is immediate that no other primal solution can be better, and that no other dual solution can be better. (For an analogy, in Lecture #2 we proved that capacity of every cut bounds from above the value of every flow, so if you ever find a flow and a cut with equal value, both must be optimal.)

CS261: A Second Course in Algorithms
Lecture #9: Linear Programming Duality (Part 2)*

Tim Roughgarden†

February 2, 2016

1 Recap

This is our third lecture on linear programming, and the second on linear programming duality. Let's page back in the relevant stuff from last lecture.

One type of linear program has the form

$$\max \sum_{j=1}^{n} c_j x_j$$

subject to

$$\begin{aligned}
\sum_{j=1}^{n} a_{1j} x_j &\leq b_1 \\
\sum_{j=1}^{n} a_{2j} x_j &\leq b_2 \\
\vdots &\leq \vdots \\
\sum_{j=1}^{n} a_{mj} x_j &\leq b_m \\
x_1, \ldots, x_n &\geq 0.
\end{aligned}$$

Call this linear program (P), for "primal." Alternatively, in matrix-vector notation it is

$$\max \mathbf{c}^T \mathbf{x}$$

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

subject to

$$\begin{aligned} \mathbf{A}\mathbf{x} &\leq \mathbf{b} \\ \mathbf{x} &\geq 0, \end{aligned}$$

where $\mathbf{c}$ and $\mathbf{x}$ are n-vectors, $\mathbf{b}$ is an m-vector, $\mathbf{A}$ is an $m \times n$ matrix (of the a_{ij}'s), and the inequalities are componentwise.

We then discussed a method for generating upper bounds on the maximum-possible objective function value of (P): take a nonnegative linear combination of the constraints so that the result dominates the objective $\mathbf{c}$, and you get an upper bound equal to the corresponding nonnegative linear combination of the right-hand side $\mathbf{b}$. A key point is that the tightest upper bound of this form is the solution to another linear program, known as the "dual." We gave a general recipe for taking duals: the dual has one variable per primal constraint and one constraint per primal variable; "max" and "min" get interchanged; the objective function and the right-hand side get interchanged; and the constraint matrix gets transposed. (There are some details about whether decision variables are nonnegative or not, and whether the constraints are equalities or inequalities; see the table last lecture.)

For example, the dual linear program for (P), call it (D), is

$$\min \mathbf{y}^T \mathbf{b}$$

subject to

$$\begin{aligned} \mathbf{A}^T \mathbf{y} &\geq \mathbf{c} \\ \mathbf{y} &\geq 0 \end{aligned}$$

in matrix-vector form. Or, if you prefer the expanded version,

$$\min \sum_{i=1}^{m} b_i y_i$$

subject to

$$\begin{aligned} \sum_{i=1}^{m} a_{i1} y_i &\geq c_1 \\ \sum_{i=1}^{m} a_{i2} y_i &\geq c_2 \\ \vdots &\geq \vdots \\ \sum_{i=1}^{m} a_{in} y_i &\geq c_n \\ y_1, \ldots, y_m &\geq 0. \end{aligned}$$

In all cases, the meaning of the dual is the tightest upper bound that can proved on the optimal primal objective function by taking suitable linear combinations of the primal constraints. With this understanding, we see that weak duality holds (for all forms of LPs), essentially by construction.

For example, for a primal-dual pair (P),(D) of the form above, for every pair $\mathbf{x}, \mathbf{y}$ of feasible solutions to (P),(D), we have

$$\underbrace{\sum_{j=1}^{n} c_j x_j}_{\mathbf{x}\text{'s obj fn}} \leq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} y_i a_{ij} \right) x_j \tag{1}$$

$$= \sum_{i=1}^{m} y_i \left(\sum_{j=1}^{n} a_{ij} x_j \right) \tag{2}$$

$$\leq \underbrace{\sum_{i=1}^{m} y_i b_i}_{\mathbf{y}\text{'s obj fn}} . \tag{3}$$

Or, in matrix-vector notion,

$$\mathbf{c}^T \mathbf{x} \leq (\mathbf{A}^T \mathbf{y})^T \mathbf{x} = \mathbf{y}^T (\mathbf{A}\mathbf{x}) \leq \mathbf{y}^t \mathbf{b}.$$

The first inequality uses that $\mathbf{x} \geq 0$ and $\mathbf{A}^t \mathbf{y} \geq \mathbf{c}$; the second that $\mathbf{y} \geq 0$ and $\mathbf{A}\mathbf{x} \leq \mathbf{b}$.

We concluded last lecture with the following sufficient condition for optimality.[1]

Corollary 1.1 *Let (P),(D) be a primal-dual pair of linear programs. If* $\mathbf{x}, \mathbf{y}$ *are feasible solutions to (P),(D), and* $\mathbf{c}^T \mathbf{x} = \mathbf{y}^T \mathbf{b}$*, then both* $\mathbf{x}$ *and* $\mathbf{y}$ *are both optimal.*

For the reason, recall Figure 1 — no "x" can be to the right of an "o", so if an "x" and "o" are superimposed it must be the rightmost "x" and the leftmost "o." For an analogy, whenever you find a flow and s-t cut with the same value, the flow must be maximum and the cut minimum.

Figure 1: Illustrative figure showing feasible solutions for the primal (x) and the dual (o).

[1] We also noted that weak duality implies that whenever the optimal objective function of (P) is unbounded the linear program (D) is infeasible, and vice versa.

2 Complementary Slackness Conditions

2.1 The Conditions

Next is a corollary of Corollary 1.1. It is another sufficient (and as we'll see later, necessary) condition for optimality.

Corollary 2.1 (Complementary Slackness Conditions) *Let (P),(D) be a primal-dual pair of linear programs. If* $\mathbf{x}, \mathbf{y}$ *are feasible solutions to (P),(D), and the following two conditions hold then both* $\mathbf{x}$ *and* $\mathbf{y}$ *are both optimal.*

(1) Whenever $x_j \neq 0$, $\mathbf{y}$ *satisfies the* j*th constraint of (D) with equality.*

(2) Whenever $y_i \neq 0$, $\mathbf{x}$ *satisfies the* i*th constraint of (P) with equality.*

The conditions assert that no decision variable and corresponding constraint are simultaneously "slack" (i.e., it forbids that the decision variable is not 0 and also the constraint is not tight).

Proof of Corollary 2.1: We prove the corollary for the case of primal and dual programs of the form (P) and (D) in Section 1; the other cases are all the same.

The first condition implies that

$$c_j x_j = \left(\sum_{i=1}^{m} y_i a_{ij} \right) x_j$$

for each $j = 1, \ldots, n$ (either $x_j = 0$ or $c_j = \sum_{i=1}^{m} y_i a_{ij}$). Hence, inequality (1) holds with equality. Similarly, the second condition implies that

$$y_i \left(\sum_{j=1}^{n} a_{ij} x_i \right) = y_i b_i$$

for each $i = 1, \ldots, m$. Hence inequality (3) also holds with equality. Thus $\mathbf{c}^T \mathbf{x} = \mathbf{y}^T \mathbf{b}$, and Corollary 1.1 implies that both $\mathbf{x}$ and $\mathbf{y}$ are optimal. ■

2.2 Physical Interpretation

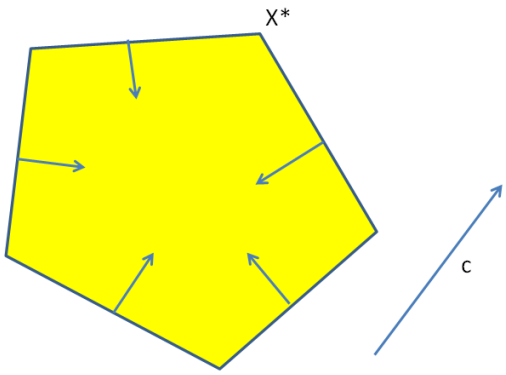

Figure 2: Physical interpretation of complementary slackness. The objective function pushes a particle in the direction c until it rests at x^*. Walls also exert a force on the particle, and complementary slackness asserts that only walls touching the particle exert a force, and sum of forces is equal to 0.

We offer the following informal physical metaphor for the complementary slackness conditions, which some students find helpful (Figure 2). For a linear program of the form (P) in Section 1, think of the objective function as exerting "force" in the direction $\mathbf{c}$. This pushes a particle in the direction $\mathbf{c}$ (within the feasible region) until it cannot move any further in this direction. When the particle comes to rest at position $\mathbf{x}^*$, the sum of the forces acting on it must sum to 0. What else exerts force on the particle? The "walls" of the feasible region, corresponding to the constraints. The direction of the force exerted by the ith constraint of the form $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$ is perpendicular to the wall, that is, $-\mathbf{a}_i$, where $\mathbf{a}_i$ is the ith row of the constraint matrix. We can interpret the corresponding dual variable y_i as the magnitude of the force exerted in this direction $-\mathbf{a}_i$. The assertion that the sum of the forces equals 0 corresponds to the equation $\mathbf{c} = \sum_{i=1}^{n} y_i\mathbf{a}_i$. The complementary slackness conditions assert that $y_i > 0$ only when $\mathbf{a}_i^T\mathbf{x} = b_i$ — that is, only the walls that the particle touches are allowed to exert force on it.

2.3 A General Algorithm Design Paradigm

So why are the complementary slackness conditions interesting? One reason is that they offer three principled strategies for designing algorithms for solving linear programs and their special cases. Consider the following three conditions.

A General Algorithm Design Paradigm

1. $\mathbf{x}$ is feasible for (P).

2. $\mathbf{y}$ is feasible for (D).

3. $\mathbf{x}, \mathbf{y}$ satisfy the complementary slackness conditions (Corollary 2.1).

Pick two of these three conditions to maintain at all times, and work toward achieving the third.

By Corollary 2.1, we know that achieving these three conditions simultaneously implies that both $\mathbf{x}$ and $\mathbf{y}$ are optimal. Each choice of a condition to relax offers a disciplined way of working toward optimality, and in many cases all three approaches can lead to good algorithms. Countless algorithms for linear programs and their special cases can be viewed as instantiations of this general paradigm. We next revisit an old friend, the Hungarian algorithm, which is a particularly transparent example of this design paradigm in action.

3 Example #2: The Hungarian Algorithm Revisited

3.1 Recap of Example #1

Recall that in Lecture #8 we reinterpreted the max-flow/min-cut theorem through the lens of LP duality (this was "Example #1"). We had a primal linear program formulation of the maximum flow problem. In the corresponding dual linear program, we observed that s-t cuts translate to 0-1 solutions to this dual, with the dual objective function value equal to the capacity of the cut. Using the max-flow/min-cut theorem, we concluded two interesting properties: first, we verified strong duality (i.e., no gap between the optimal primal and dual objective function values) for primal-dual pairs corresponding to flows and (fractional) cuts; second, we concluded that these dual linear programs are always guaranteed to possess an integral optimal solution (i.e., fractions don't help).

3.2 The Primal Linear Program

Back in Lecture #7 we claimed that all of the problems studied thus far are special cases of linear programs. For the maximum flow problem, this is easy to believe, because flows can be fractional. But for matchings? They are suppose to be integral, so how could they be modeled with a linear program? Example #1 provides the clue — sometimes, linear programs are guaranteed to have an optimal integral solution. As we'll see, this also turns out to be the case for bipartite matching.

Given a bipartite graph $G = (V \cup W, E)$ with a cost c_e for each edge, the relevant linear program (P-BM) is

$$\min \sum_{e \in E} c_e x_e$$

subject to

$$\sum_{e \in \delta(v)} x_e = 1 \qquad \text{for all } v \in V \cup W$$
$$x_e \geq 0 \qquad \text{for all } e \in E,$$

where $\delta(v)$ denotes the edges incident to v. The intended semantics is that each x_e is either equal to 1 (if e is in the chosen matching) or 0 (otherwise). Of course, the linear program is also free to use fractional values for the decision variables.[2]

In matrix-vector form, this linear program is

$$\min \mathbf{c}^T \mathbf{x}$$

subject to

$$\mathbf{A}\mathbf{x} = \mathbf{1}$$
$$\mathbf{x} \geq 0,$$

where $\mathbf{A}$ is the $(V \cup W) \times E$ matrix

$$\mathbf{A} = \left[a_{ve} = \begin{cases} 1 \text{ if } e \in \delta(v) \\ 0 \text{ otherwise} \end{cases} \right] \tag{4}$$

3.3 The Dual Linear Program

We now turn to the dual linear program. Note that (P-BM) differs from our usual form both by having a minimization objective and by having equality (rather than inequality) constraints. But our recipe for taking duals from Lecture #8 applies to all types of linear programs, including this one.

When taking a dual, usually the trickiest point is to understand the effect of the transpose operation (on the constraint matrix). In the constraint matrix $\mathbf{A}$ in (4), each row (indexed by $v \in V \cup W$) has a 1 in each column (indexed by $e \in E$) for which e is incident to v (and 0s in other columns). Thus, a column of $\mathbf{A}$ (and hence row of $\mathbf{A}^T$) corresponding to edge e has 1s in precisely the rows (indexed by v) such that e is incident to v — that is, in the two rows corresponding to e's endpoints.

Applying our recipe for duals to (P-BM), initially in matrix-vector form for simplicity, yields

$$\max \mathbf{p}^T \mathbf{1}$$

subject to

$$\mathbf{A}^T \mathbf{p} \leq \mathbf{c}$$
$$\mathbf{p} \in \mathbb{R}^E.$$

[2] If you're tempted to also add in the constraints that $x_e \leq 1$ for every $e \in E$, note that these are already implied by the current constraints (why?).

We are using the notation p_v for the dual variable corresponding to a vertex $v \in V \cup W$, for reasons that will become clearly shortly. Note that these decision variables can be positive or negative, because of the equality constraints in (P-BM).

Unpacking this dual linear program, (D-BM), we get

$$\max \sum_{v \in V \cup W} p_v$$

subject to

$$\begin{aligned} p_v + p_w &\leq c_{vw} \qquad \text{for all } (v,w) \in E \\ p_v &\in \mathbb{R} \qquad \text{for all } v \in V \cup W. \end{aligned}$$

Here's the punchline: the "vertex prices" in the Hungarian algorithm (Lecture #5) correspond exactly to the decision variables of the dual (D-BM). Indeed, without thinking about this dual linear program, how would you ever think to maintain numbers attached to the *vertices* of a graph matching instance, when the problem definition seems to only concern the graph's edges?[3]

It gets better: rewrite the constraints of (D-BM) as

$$\underbrace{c_{vw} - p_v - p_w}_{\text{reduced cost}} \geq 0 \tag{5}$$

for every edge $(v,w) \in E$. The left-hand side of (5) is exactly our definition in the Hungarian algorithm of the "reduced cost" of an edge (with respect to prices $\mathbf{p}$). Thus the first invariant of the Hungarian algorithm, asserting that all edges have nonnegative reduced costs, is exactly the same as maintaining the dual feasibility of $\mathbf{p}$!

To seal the deal, let's check out the complementary slackness conditions for the primal-dual pair (P-BM),(D-BM). Because all constraints in (P-BM) are equations (not counting the nonnegativity constraints), the second condition is trivial. The first condition states that whenever $x_e > 0$, the corresponding constraint (5) should hold with equality — that is, edge e should have zero reduced cost. Thus the second invariant of the Hungarian algorithm (that edges in the current matching should be "tight") is just the complementary slackness condition!

We conclude that, in terms of the general algorithm design paradigm in Section 2.3, the Hungarian algorithm maintains the second two conditions ($\mathbf{p}$ is feasible for (D-BM) and complementary slackness conditions) at all times, and works toward the first condition (primal feasibility, i.e., a perfect matching). Algorithms of this type are called *primal-dual algorithms*, and the Hungarian algorithm is a canonical example.

[3] In Lecture #5 we motivated vertex prices via an analogy with the vertex labels maintained by the push-relabel maximum flow algorithm. But the latter is from the 1980s and the former from the 1950s, so that was a pretty ahistorical analogy. Linear programming (and duality) were only developed in the late 1940s, and so it was a new subject when Kuhn designed the Hungarian algorithm. But he was one of the first masters of the subject, and he put his expertise to good use.

3.4 Consequences

We know that

$$\text{OPT of (D-PM)} \leq \text{OPT of (P-PM)} \leq \text{min-cost perfect matching.} \tag{6}$$

The first inequality is just weak duality (for the case where the primal linear program has a minimization objective). The second inequality follows from the fact that every perfect matching corresponds to a feasible (0-1) solution of (P-BM); since the linear program minimizes over a superset of these solutions, it can only have a better (i.e., smaller) optimal objective function value.

In Lecture #5 we proved that the Hungarian algorithm always terminates with a perfect matching (provided there is at least one). The algorithm maintains a feasible dual and the complementary slackness conditions. As in the proof of Corollary 2.1, this implies that the cost of the constructed perfect matching equals the dual objective function value attained by the final prices. That is, both inequalities in (6) must hold with equality.

As in Example #1 (max flow/min cut), both of these equalities are interesting. The first equation verifies another special case of strong LP duality, for linear programs of the form (P-BM) and (D-BM). The second equation provides another example of a natural family of linear programs — those of the form (P-BM) — that are guaranteed to have 0-1 optimal solutions.[4]

4 Strong LP Duality

4.1 Formal Statement

Strong linear programming duality ("no gap") holds in general, not just for the special cases that we've seen thus far.

Theorem 4.1 (Strong LP Duality) *When a primal-dual pair (P),(D) of linear programs are both feasible,*

$$\textit{OPT for (P)} = \textit{OPT for (D)}.$$

Amazingly, our simple method of deriving bounds on the optimal objective function value of (P) through suitable linear combinations of the constraints is always guaranteed to produce the tightest-possible bound! Strong duality can be thought of as a generalization of the max-flow/min-cut theorem (Lecture #2) and Hall's theorem (Lecture #5), and as the ultimate answer to the question "how do we know when we're done?"[5]

[4] See also Exercise Set #4 for a direct proof of this.

[5] When at least one of (P),(D) is infeasible, there are three possibilities, all of which can occur. First, (P) might have unbounded objective function value, in which case (by weak duality) (D) is infeasible. It is also possible that (P) is infeasible while (D) has unbounded objective function value. Finally, sometimes both (P) and (D) are infeasible (an uninteresting case).

4.2 Consequent Optimality Conditions

Strong duality immediately implies that the sufficient conditions for optimality identified earlier (Corollaries 1.1 and 2.1) are also necessary conditions — that is, they are *optimality conditions* in the sense derived earlier for the maximum flow and minimum-cost perfect bipartite matching problems.

Corollary 4.2 (LP Optimality Conditions) *Let* $\mathbf{x}, \mathbf{y}$ *are feasible solutions to the primal-dual pair (P),(D) be a = primal-dual pair, then*

$\mathbf{x}, \mathbf{y}$ *are both optimal* *if and only if* $\mathbf{c}^T\mathbf{x} = \mathbf{y}^T\mathbf{b}$
if and only if *the complementary slackness conditions hold.*

The first if and only if follows from strong duality: since both (P),(D) are feasible by assumption, strong duality assures us of feasible solutions $\mathbf{x}^*, \mathbf{y}^*$ with $\mathbf{c}\mathbf{x}^* = (\mathbf{y}^*)^T\mathbf{b}$. If $\mathbf{x}, \mathbf{y}$ fail to satisfy this equality, then either $\mathbf{c}^T\mathbf{x}$ is worse than $\mathbf{c}^T\mathbf{x}^*$ or $\mathbf{y}^T\mathbf{b}$ is worse than $(\mathbf{y}^*)^T\mathbf{b}$ (or both). The second if and only if does not require strong duality; it follows from the proof of Corollary 2.1 (see also Exercise Set #4).

4.3 Proof Sketch: The Road Map

We conclude the lecture with a proof sketch of Theorem 4.1. Our proof sketch leaves some details to Problem Set #3, and also takes on faith one intuitive geometric fact. The goal of the proof sketch is to at least partially demystify strong LP duality, and convince you that it ultimately boils down to some simple geometric intuition.

Here's the plan:

$$\underbrace{\text{separating hyperplane}}_{\text{will assume}} \Rightarrow \underbrace{\text{Farkas's Lemma}}_{\text{will prove}} \rightarrow \underbrace{\text{strong LP duality}}_{\text{PSet \#3}}.$$

The "separating hyperplane theorem" is the intuitive geometric fact that we assume (Section 4.4). Section 4.5 derives from this fact Farkas's Lemma, a "feasibility version" of strong LP duality. Problem Set #3 asks you to reduce strong LP duality to Farkas's Lemma.

4.4 The Separating Hyperplane Theorem

In Lecture #7 we discussed separating hyperplanes, in the context of separating data points labeled "positive" from those labeled "negative." There, the point was to show that the computational problem of finding such a hyperplane reduces to linear programming. Here, we again discuss separating hyperplanes, with two differences: first, our goal is to separate a convex set from a point not in the set (rather than two different sets of points); second, the point here is to prove strong LP duality, not to give an algorithm for a computational problem.

We assume the following result.

Theorem 4.3 (Separating Hyperplane) *Let C be a closed and convex subset of $\mathbb{R}^n$, and $\mathbf{z}$ a point in $\mathbb{R}^n$ not in C. Then there is a* separating hyperplane*, meaning coefficients $\alpha \in \mathbb{R}^n$ and an intercept $\beta \in \mathbb{R}$ such that:*

(1)

$$\underbrace{\alpha^T \mathbf{x} \geq \beta \qquad \text{for all } \mathbf{x} \in C}_{\text{all of } C \text{ on one side of hyperplane}};$$

(2)

$$\underbrace{\alpha^T \mathbf{z} < \beta}_{\mathbf{z} \text{ on other side}} .$$

See also Figure 3. Note that the set C is not assumed to be bounded.

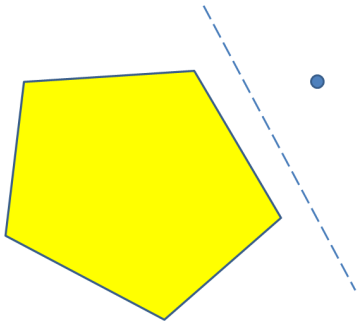

Figure 3: Illustration of separating hyperplane theorem.

If you've forgotten what "convex" or "closed" means, both are very intuitive. A convex set is "filled in," meaning it contains all of its chords. Formally, this translates to

$$\underbrace{\lambda \mathbf{x} + (1-\lambda)\mathbf{y}}_{\text{point on chord between } \mathbf{x}, \mathbf{y}} \in C$$

for all $\mathbf{x}, \mathbf{y} \in C$ and $\lambda \in [0, 1]$. See Figure 4 for an example (a filled-in polygon) and a non-example (an annulus).

A closed set is one that includes its boundary.[6] See Figure 5 for an example (the unit disc) and a non-example (the open unit disc).

[6] One formal definition is that whenever a sequence of points in C converges to a point $\mathbf{x}^*$, then $\mathbf{x}^*$ should also be in C.

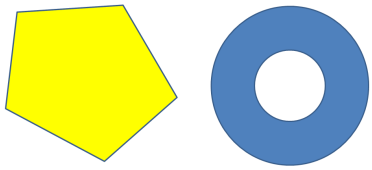

Figure 4: (a) a convex set (filled-in polygon) and (b) a non-convex set (annulus)

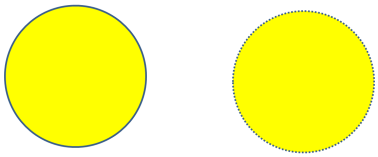

Figure 5: (a) a closed set (unit disc) and (b) non-closed set (open unit disc)

Hopefully Theorem 4.3 seems geometrically obvious, at least in two and three dimensions. It turns out that the math one would use to prove this formally extends without trouble to an arbitrary number of dimensions.[7] It also turns out that strong LP duality boils down to exactly this fact.

4.5 Farkas's Lemma

It's easy to convince someone whether or not a system of linear equations has a solution: just run Gaussian elimination and see whether or not it finds a solution (if there is a solution, Gaussian elimination will find one). For a system of linear *inequalities*, it's easy to convince someone that there is a solution — just exhibit it and let them verify all the constraints. But how would you convince someone that a system of linear inequalities has *no* solution? You can't very well enumerate the infinite number of possibilities and check that each doesn't work. *Farkas's Lemma* is a satisfying answer to this question, and can be thought of as the "feasibility version" of strong LP duality.

Theorem 4.4 (Farkas's Lemma) *Given a matrix* $\mathbf{A} \in \mathbb{R}^{m \times n}$ *and a right-hand side* $\mathbf{b} \in \mathbb{R}^m$*, exactly one of the following holds:*

(i) There exists $\mathbf{x} \in \mathbb{R}^n$ *such that* $\mathbf{x} \geq 0$ *and* $\mathbf{A}\mathbf{x} = \mathbf{b}$*;*

(ii) There exists $\mathbf{y} \in \mathbb{R}^m$ *such that* $\mathbf{y}^T\mathbf{A} \geq 0$ *and* $\mathbf{y}^T\mathbf{b} < 0$*.*

[7] If you know undergraduate analysis, then even the formal proof is not hard: let $\mathbf{y}$ be the nearest neighbor to $\mathbf{z}$ in C (such a point exists because C is closed), and take a hyperplane perpendicular to the line segment between $\mathbf{y}$ and $\mathbf{z}$, through the midpoint of this segment (cf., Figure 3). All of C lies on the same side of this hyperplane (opposite of $\mathbf{z}$) because C is convex and $\mathbf{y}$ is the nearest neighbor of $\mathbf{z}$ in C.

To connect the statement to the previous paragraph, think of $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{x} \geq 0$ as the linear system of inequalities that we care about, and solutions to (ii) as proofs that this system has no feasible solution.

Just like there are many variants of linear programs, there are many variants of Farkas's Lemma. Given Theorem 4.4, it is not hard to translate it to analogous statements for other linear systems of inequalities (e.g., with both inequality and nonnegativity constraints); see Problem Set #3.

Proof of Theorem 4.4: First, we have deliberately set up (i) and (ii) so that it's impossible for both to have a feasible solution. For if there were such an $\mathbf{x}$ and $\mathbf{y}$, we would have

$$\underbrace{(\mathbf{y}^T\mathbf{A})}_{\geq 0}\underbrace{\mathbf{x}}_{\geq 0} \geq 0$$

and yet

$$\mathbf{y}^T(\mathbf{Ax}) = \mathbf{y}^T\mathbf{b} < 0,$$

a contradiction. In this sense, solutions to (ii) are proofs of infeasibility of the the system (i) (and vice versa).

But why can't both (i) and (ii) be infeasible? We'll show that this can't happen by proving that, whenever (i) is infeasible, (ii) is feasible. Thus the "proofs of infeasibility" encoded by (ii) are all that we'll ever need — whenever the linear system (i) is infeasible, there is a proof of it of the prescribed type. There is a clear analogy between this interpretation of Farkas's Lemma and strong LP duality, which says that there is always a feasible dual solution proving the tightest-possible bound on the optimal objective function value of the primal.

Assume that (i) is infeasible. We need to somehow exhibit a solution to (ii), but where could it come from? The trick is to get it from the separating hyperplane theorem (Theorem 4.3) — the coefficients defining the hyperplane will turn out to be a solution to (ii). To apply this theorem, we need a closed convex set and a point not in the set.

Define

$$Q = \{\mathbf{d} : \exists \mathbf{x} \geq 0 \text{ s.t. } \mathbf{Ax} = \mathbf{d}\}.$$

Note that Q is a subset of $\mathbb{R}^m$. There are two different and equally useful ways to think about Q. First, for the given constraint matrix $\mathbf{A}$, Q is the set of all right-hand sides $\mathbf{d}$ that are feasible (in $\mathbf{x} \geq 0$) with this constraint matrix. Thus by assumption, $\mathbf{b} \notin Q$. Equivalently, considering all vectors of the form $\mathbf{Ax}$, with $\mathbf{x}$ ranging over all nonnegative vectors in $\mathbb{R}^n$, generates precisely the set of feasible right-hand sides. Thus Q equals the set of all nonnegative linear combinations of the columns of $\mathbf{A}$.[8] This definition makes it obvious that Q is convex (an average of two nonnegative linear combinations is just another nonnegative linear combination). Q is also closed (the limit of a convergent sequence of nonnegative linear combinations is just another nonnegative linear combination).

[8]Called the "cone generated by" the columns of $\mathbf{A}$.

Since Q is closed and convex and $\mathbf{b} \notin Q$, we can apply Theorem 4.3. In return, we are granted a coefficient vector $\alpha \in \mathbb{R}^m$ and an intercept $\beta \in \mathbb{R}$ such that

$$\alpha^T \mathbf{d} \geq \beta$$

for all $\mathbf{d} \in Q$ and

$$\alpha^T \mathbf{b} < \beta.$$

An exercise shows that, since Q is a cone, we can take $\beta = 0$ without loss of generality (see Exercise Set #5). Thus

$$\alpha^T \mathbf{d} \geq 0 \tag{7}$$

for all $\mathbf{d} \in Q$ while

$$\alpha^T \mathbf{b} < 0. \tag{8}$$

A solution $\mathbf{y}$ to (ii) satisfies $\mathbf{y}^T \mathbf{A} \geq 0$ and $\mathbf{y}^T \mathbf{b} < 0$. Suppose we just take $\mathbf{y} = \alpha$. Inequality (8) implies the second condition, so we just have to check that $\alpha^T \mathbf{A} \geq 0$. But what is $\alpha^T \mathbf{A}$? An n-vector, where the jth coordinate is inner product of α^T and the jth column $\mathbf{a}^j$ of $\mathbf{A}$. Since each $\mathbf{a}^j \in Q$ — the jth column is obviously one particular nonnegative linear combination of $\mathbf{A}$'s columns — inequality (7) implies that every coordinate of $\alpha^T \mathbf{A}$ is nonnegative. Thus α is a solution to (ii), as desired. ■

4.6 Epilogue

On Problem Set #3 you will use Theorem 4.4 to prove strong LP duality. The idea is simple: let $OPT_{(D)}$ denote the optimal value of the dual linear program, add a constraint to the primal stating that the (primal) objective function value must be equal to or better than $OPT_{(D)}$, and use Farkas's Lemma to prove that this augmented linear program is feasible.

In summary, strong LP duality is amazing and powerful, yet it ultimately boils down to the highly intuitive existence of a separating hyperplane between a closed convex set and a point not in the set.

CS261: Problem Set #1

Due by 11:59 PM on Tuesday, January 26, 2016

Instructions:

(1) Form a group of 1-3 students. You should turn in only one write-up for your entire group.

(2) Submission instructions: We are using Gradescope for the homework submissions. Go to www.gradescope.com to either login or create a new account. Use the course code 9B3BEM to register for CS261. Only one group member needs to submit the assignment. When submitting, please remember to add all group member names in Gradescope.

(3) Please type your solutions if possible and we encourage you to use the LaTeX template provided on the course home page.

(4) Write convincingly but not excessively.

(5) Some of these problems are difficult, so your group may not solve them all to completion. In this case, you can write up what you've got (subject to (3), above): partial proofs, lemmas, high-level ideas, counterexamples, and so on.

(6) Except where otherwise noted, you may refer to the course lecture notes *only*. You can also review any relevant materials from your undergraduate algorithms course.

(7) You can discuss the problems verbally at a high level with other groups. And of course, you are encouraged to contact the course staff (via Piazza or office hours) for additional help.

(8) If you discuss solution approaches with anyone outside of your group, you must list their names on the front page of your write-up.

(9) Refer to the course Web page for the late day policy.

Problem 1

This problem explores "path decompositions" of a flow. The input is a flow network (as usual, a directed graph $G = (V, E)$, a source s, a sink t, and a positive integral capacity u_e for each edge), as well as a flow f in G. As always with graphs, m denotes $|E|$ and n denotes $|V|$.

(a) A flow is *acyclic* if the subgraph of directed edges with positive flow contains no directed cycles. Prove that for every flow f, there is an acyclic flow with the same value of f. (In particular, this implies that some maximum flow is acyclic.)

(b) A *path flow* assigns positive values only to the edges of one simple directed path from s to t. Prove that every acyclic flow can be written as the sum of at most m path flows.

(c) Is the Ford-Fulkerson algorithm guaranteed to produce an acyclic maximum flow?

(d) A *cycle flow* assigns positive values only to the edges of one simple directed cycle. Prove that every flow can be written as the sum of at most m path and cycle flows.

(e) Can you compute the decomposition in (d) in $O(mn)$ time?

Problem 2

Consider a directed graph $G = (V, E)$ with source s and sink t for which each edge e has a positive integral capacity u_e. Recall from Lecture #2 that a *blocking flow* in such a network is a flow $\{f_e\}_{e \in E}$ with the property that, for every s-t path P of G, there is at least one edge of P such that $f_e = u_e$. For example, our first (broken) greedy algorithm from Lecture #1 terminates with a blocking flow (which, as we saw, is not necessarily a maximum flow).

Dinic's Algorithm

initialize $f_e = 0$ for all $e \in E$
while there is an s-t path in the current residual network G_f **do**
  construct the layered graph L_f, by computing the residual graph G_f and running breadth-first search (BFS) in G_f starting from s, stopping once the sink t is reached, and retaining only the forward edges[1]
  compute a blocking flow g in G_f
  `// augment the flow` f `using the flow` g
  for all edges (v, w) of G for which the corresponding forward edge of G_f carries flow ($g_{vw} > 0$) **do**
    increase f_e by g_e
  for all edges (v, w) of G for which the corresponding reverse edge of G_f carries flow ($g_{wv} > 0$) **do**
    decrease f_e by g_e

The termination condition implies that the algorithm can only halt with a maximum flow. Exercise Set #1 argues that every iteration of the main loop increases $d(f)$, the length (i.e., number of hops) of a shortest s-t path in G_f, and therefore the algorithm stops after at most n iterations. Its running time is therefore $O(n \cdot BF)$, where BF is the amount of time required to compute a blocking flow in the layered graph L_f. We know that $BF = O(m^2)$ — our first broken greedy algorithm already proves this — but we can do better.

Consider the following algorithm, inspired by depth-first search, for computing a blocking flow in L_f:

A Blocking Flow Algorithm

Initialize. Initialize the flow variables g_e to 0 for all $e \in E$. Initialize the path variable P as the empty path, from s to itself. Go to **Advance.**

Advance. Let v denote the current endpoint of the path P. If there is no edge out of v, go to **Retreat**. Otherwise, append one such edge (v, w) to the path P. If $w \neq t$ then go to **Advance**. If $w = t$ then go to **Augment**.

Retreat. Let v denote the current endpoint of the path P. If $v = s$ then halt. Otherwise, delete v and all of its incident edges from L_f. Remove from P its last edge. Go to **Advance**.

Augment. Let Δ denote the smallest residual capacity of an edge on the path P (which must be an s-t path). Increase g_e by Δ on all edges $e \in P$. Delete newly saturated edges from L_f, and let $e = (v, w)$ denote the first such edge on P. Retain only the subpath of P from s to v. Go to **Advance.**

And now the analysis:

(a) Prove that the running time of the algorithm, suitably implemented, is $O(mn)$. (As always, m denotes $|E|$ and n denotes $|V|$.)

[Hint: How many times can **Retreat** be called? How many times can **Augment** be called? How many times can **Advance** be called before a call to **Retreat** or **Augment**?]

[1] Recall that a forward edge in BFS goes from layer i to layer $(i + 1)$, for some i.

(b) Prove that the algorithm terminates with a blocking flow g in L_f.

[For example, you could argue by contradiction.]

(c) Suppose that every edge of L_f has capacity 1 (cf., Exercise #4). Prove that the algorithm above computes a blocking flow in linear (i.e., $O(m)$) time.

[Hint: can an edge (v, w) be chosen in two different calls to **Advance**?]

Problem 3

In this problem we'll analyze a different augmenting path-based algorithm for the maximum flow problem. Consider a flow network with integral edge capacities. Suppose we modify the Edmonds-Karp algorithm (Lecture #2) so that, instead of choosing a shortest augmenting path in the residual network G_f, it chooses an augmenting path on which it can push the most flow. (That is, it maximizes the minimum residual capacity of an edge in the path.) For example, in the network in Figure 1, this algorithm would push 3 units of flow on the path $s \to v \to w \to t$ in the first iteration. (And 2 units on $s \to w \to v \to t$ in the second iteration.)

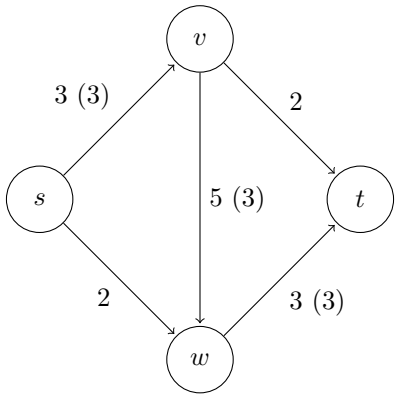

Figure 1: Problem 3. Edges are labeled with their capacities, with flow amounts in parentheses.

(a) Show how to modify Dijkstra's shortest-path algorithm, without affecting its asymptotic running time, so that it computes an s-t path with the maximum-possible minimum residual edge capacity.

(b) Suppose the current flow f has value F and the maximum flow value in G is F^*. Prove that there is an augmenting path in G_f such that every edge has residual capacity at least $(F^* - F)/m$, where $m = |E|$.

[Hint: if Δ is the maximum amount of flow that can be pushed on any s-t path of G_f, consider the set of vertices reachable from s along edges in G_f with residual capacity more than Δ. Relate the residual capacity of this (s, t)-cut to $F^* - F$.]

(c) Prove that this variant of the Edmonds-Karp algorithm terminates within $O(m \log F^*)$ iterations, where F^* is defined as in the previous problem.

[Hint: you might find the inequality $1 - x \leq e^{-x}$ for $x \in [0, 1]$ useful.]

(d) Assume that all edge capacities are integers in $\{1, 2, \ldots, U\}$. Give an upper bound on the running time of your algorithm as a function of $n = |V|$, m, and U. Is this bound polynomial in the input size?

Problem 4

In this problem we'll revisit the special case of *unit-capacity* networks, where every edge has capacity 1 (see also Exercise 4).

(a) Recall the notation $d(f)$ for the length (in hops) of a shortest s-t path in the residual network G_f. Suppose G is a unit-capacity network and f is a flow with value F. Prove that the maximum flow value is at most $F + \frac{m}{d(f)}$.

[Hint: use the layered graph L_f discussed in Problem 2 to identify an s-t cut of the residual graph that has small residual capacity. Then argue along the lines of Problem 3(b).]

(b) Explain how to compute a maximum flow in a unit-capacity network in $O(m^{3/2})$ time.

[Hints: use Dinic's algorithm and Problem 2(c). Also, in light of part (a) of this problem, consider the question: if you know that the value of the current flow f is only c less than the maximum flow value in G, then what's a crude upper bound on the number of additional blocking flows required before you're sure to terminate with a maximum flow?]

Problem 5

(Difficult.) This problem sharpens the analysis of the highest-label push-relabel algorithm (Lecture #3) to improve the running time bound from $O(n^3)$ to $O(n^2\sqrt{m})$.[2] (Replacing an n by a $\sqrt{m}$ is always a good thing.) Recall from the Lecture #3 analysis that it suffices to prove that the number of non-saturating pushes is $O(n^2\sqrt{m})$ (since there are only $O(n^2)$ relabels and $O(nm)$ saturating pushes, anyways).

For convenience, we augment the algorithm with some bookkeeping: each vertex v maintains at most one *successor*, which is a vertex w such that (v, w) has positive residual capacity and $h(v) = h(w)+1$ (i.e., (v, w) goes "downhill"). (If there is no such w, v's successor is NULL.) When a push is called on the vertex v, flow is pushed from v to its successor w. Successors are updated as needed after each saturating push or relabel.[3] For a preflow f and corresponding residual graph G_f, we denote by S_f the subgraph of G_f consisting of the edges $\{(v, w) \in G_f : w \text{ is } v\text{'s successor}\}$.

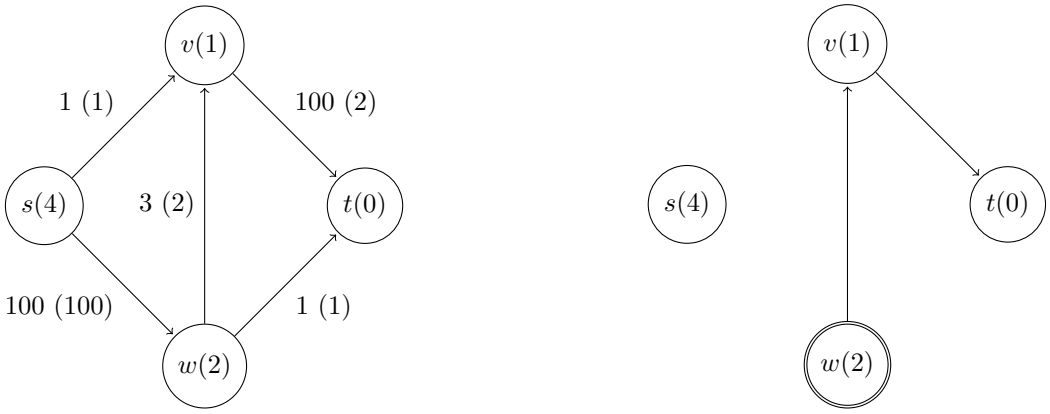

Figure 2: (a) Sample instance of running push-relabel algorithm. As usual, for edges, the flows values are in brackets. For vertices, the bracketed values denote the heights of vertices. (b) S_f for the given preflow in (a). Maximal vertices are denoted by two circles.

(a) Note that every vertex of S_f has out-degree 0 or 1. Prove that S_f is a directed forest, meaning a collection of disjoint directed trees (in each tree, all edges are directed inward toward the root).

(b) Define $D(v)$ as the number of descendants of v in its directed tree (including v itself). Equivalently, $D(v)$ is the number of vertices that can reach v by repeatedly following successor edges. (The $D(v)$'s can change each time the preflow, height function, or successor edges change.)

Prove that the push-relabel algorithm only pushes flow from v to w when $D(w) > D(v)$.

[2] Believe it or not, this is a tight upper bound — the algorithm requires $\Omega(n^2\sqrt{m})$ operations in the worst case.

[3] We leave it as an exercise to think about how to implement this to get an algorithm with overall running time $O(n^2\sqrt{m})$.

(c) Call a vertex with excess *maximal* if none of its descendants have excess. (Every highest vertex with excess is maximal — do you see why? — but the converse need not hold.) For such a vertex, define

$$\phi(v) = \max\{K - D(v) + 1, 0\},$$

where K is a parameter to be chosen in part (i). For the other vertices, define $\phi(v) = 0$. Define

$$\Phi = \sum_{v \in V} \phi(v).$$

Prove that a non-saturating push, from a highest vertex v with positive excess, cannot increase Φ. Moreover, such a push strictly decreases Φ if $D(v) \leq K$.

(d) Prove that changing a vertex's successor from NULL to a non-NULL value cannot increase Φ.

(e) Prove that each relabel increases Φ by at most K.

[Hint: before a relabel at v, v has out-degree 0 in S_f. After the re-label, it has in-degree 0. Can this create new maximal vertices? And how do the different $D(w)$'s change?]

(f) Prove that each saturating push increases Φ by at most K.

(g) A *phase* is a maximal sequence of operations such that the maximum height of a vertex with excess remains unchanged. (The set of such vertices can change.) Prove that there are $O(n^2)$ phases.

(h) Arguing as in Lecture #3 shows that each phase performs at most n non-saturating pushes (why?), but we want to beat the $O(n^3)$ bound. Suppose that a phase performs at least $\frac{2n}{K}$ non-saturating pushes. Show that at least half of these strictly decrease Φ.

[Hint: prove that if a phase does a non-saturating push at both v and w during a phase, then v and w share no descendants during the phase. How many such vertices can there be with more than K descendants?]

(i) Prove a bound of $O(\frac{n^3}{K} + nmK)$ on the total number of non-saturating pushes across all phases. Choose K so that the bound simplifies to $O(n^2\sqrt{m})$.

Problem 6

Suppose we are given an array $A[1..m][1..n]$ of non-negative real numbers. We want to round A to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A. (Assume that all row and column sums of A are integral.) For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

(a) Describe and analyze an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.

[Hint: don't solve the problem from scratch, use a reduction instead.]

(b) Prove that such a rounding is guaranteed to exist.

CS261: Problem Set #2

Due by 11:59 PM on Tuesday, February 9, 2016

Instructions:

(1) Form a group of 1-3 students. You should turn in only one write-up for your entire group.

(2) Submission instructions: We are using Gradescope for the homework submissions. Go to www.gradescope.com to either login or create a new account. Use the course code 9B3BEM to register for CS261. Only one group member needs to submit the assignment. When submitting, please remember to add all group member names in Gradescope.

(3) Please type your solutions if possible and we encourage you to use the LaTeX template provided on the course home page.

(4) Write convincingly but not excessively.

(5) Some of these problems are difficult, so your group may not solve them all to completion. In this case, you can write up what you've got (subject to (3), above): partial proofs, lemmas, high-level ideas, counterexamples, and so on.

(6) Except where otherwise noted, you may refer to the course lecture notes *only*. You can also review any relevant materials from your undergraduate algorithms course.

(7) You can discuss the problems verbally at a high level with other groups. And of course, you are encouraged to contact the course staff (via Piazza or office hours) for additional help.

(8) If you discuss solution approaches with anyone outside of your group, you must list their names on the front page of your write-up.

(9) Refer to the course Web page for the late day policy.

Problem 7

A *vertex cover* of an undirected graph (V, E) is a subset $S \subseteq V$ such that, for every edge $e \in E$, at least one of e's endpoints lies in S.[1]

(a) Prove that in every graph, the minimum size of a vertex cover is at least the size of a maximum matching.

(b) Give a non-bipartite graph in which the minimum size of a vertex cover is strictly bigger than the size of a maximum matching.

(c) Prove that the problem of computing a minimum-cardinality vertex cover can be solved in polynomial time in bipartite graphs.[2]

[Hint: reduction to maximum flow.]

(d) Prove that in every bipartite graph, the minimum size of a vertex cover equals the size of a maximum matching.

[1] Yes, the problem is confusingly named.

[2] In general graphs, the problem turns out to be NP-hard (you don't need to prove this).

Problem 8

This problem considers the special case of maximum flow instances where edges have integral capacities and also

(*) for every vertex v other than s and t, either (i) there is at most one edge entering v, and this edge (if it exists) has capacity 1; or (ii) there is at most one edge exiting v, and this edge (if it exists) has capacity 1.

Your tasks:

(a) Prove that the maximum flow problem can be solved in $O(m\sqrt{n})$ time in networks that satisfy (*). (As always, m is the number of edges and n is the number of vertices.)

[Hint: proceed as in Problem 4, but prove a stronger version of part (a) of that problem.]

(b) Prove that the maximum bipartite matching problem can be solved in $O(m\sqrt{n})$ time.

[Hint: examine the reduction in Lecture #4.]

Problem 9

This problem considers approximation algorithms for graph matching problems.

(a) For the maximum-cardinality matching problem in bipartite graphs, prove that for every constant $\epsilon > 0$, there is an $O(m)$-time algorithm that computes a matching with size at most ϵn less than the maximum possible (where n is the number of vertices). (The hidden constant in the big-oh notation can depend on $\frac{1}{\epsilon}$.)

[Hint: ideas from Problem 8(b) should be useful.]

(b) Now consider non-bipartite graphs where each edge e has a real-valued weight w_e. Recall the greedy algorithm from Lecture #6:

Greedy Matching Algorithm

sort and rename the edges $E = \{1, 2, \ldots, m\}$ so that $w_1 \geq w_2 \geq \cdots w_m$
$M = \emptyset$
for $i = 1$ to m **do**
  if $w_i > 0$ and e_i shares no endpoint with edges in M **then**
    add e_i to M

How fast can you implement this algorithm?

(c) Prove that the greedy algorithm always outputs a matching with total weight at least 50% times that of the maximum possible.

[Hint: if the greedy algorithm adds an edge e to M, how many edges in the optimal matching can this edge "block"? How do the weights of the blocked edges compare to that of e?]

Problem 10

This problem concerns running time optimizations to the Hungarian algorithm for computing minimum-cost perfect bipartite matchings (Lecture #5). Recall the $O(mn^2)$ running time analysis from lecture: there are at most n augmentation steps, at most n price update steps between two augmentation steps, and each iteration can be implemented in $O(m)$ time.

(a) By a *phase*, we mean a maximal sequence of price update iterations (between two augmentation iterations). The naive implementation in lecture regrows the search tree from scratch after each price update in a phase, spending $O(m)$ time on this for each of up to n iterations. Show how to reuse work from previous iterations so that the total amount of work done searching for good paths, in total over all iterations in the phase, is only $O(m)$.

[Hint: compare to Problem 2(a).]

(b) The other non-trivial work in a price update phase is computing the value of Δ (the magnitude of the update). This is easy to do in $O(m)$ time per iteration. Explain how to maintain a heap data structure so that the total time spent computing Δ over all iterations in the phase is only $O(m \log n)$. Be sure to explain what heap operations you perform while growing the search tree and when executing a price update.

[This yields an $O(mn \log n)$ time implementation of the Hungarian algorithm.]

Problem 11

In the *minimum-cost flow problem*, the input is a directed graph $G = (V, E)$, a source $s \in V$, a sink $t \in V$, a target flow value d, and a capacity $u_e \geq 0$ and cost $c_e \in \mathbb{R}$ for each edge $e \in E$. The goal is to compute a flow $\{f_e\}_{e \in E}$ sending d units from s to t with the minimum-possible cost $\sum_{e \in E} c_e f_e$. (If there is no such flow, the algorithm should correctly report this fact.)

Given a min-cost flow instance and a feasible flow f with value d, the corresponding *residual network* G_f is defined as follows. The vertex set remains V. For every edge $(v, w) \in E$ with $f_{vw} < u_{vw}$, there is an edge (v, w) in G_f with cost c_e and residual capacity $u_e - f_e$. For every edge $(v, w) \in E$ with $f_{vw} > 0$, there is a reverse edge (w, v) in G_f with the cost $-c_e$ and residual capacity f_e.

A *negative cycle* of G_f is a directed cycle C of G_f such that the sum of the edge costs in C is negative. (E.g., $v \to w \to x \to y \to v$, with $c_{vw} = 2$, $c_{wx} = -1$, $c_{xy} = 3$, and $c_{yv} = -5$.)

(a) Prove that if the residual network G_f of a flow f has a negative cycle, then f is not a minimum-cost flow.

(b) Prove that if the residual network G_f of a flow f has no negative cycles, then f is a minimum-cost flow.

[Hint: look to the proof of the minimum-cost bipartite matching optimality conditions (Lecture #5) for inspiration.]

(c) Give a polynomial-time algorithm that, given a residual network G_f, either returns a negative cycle or correctly reports that no negative cycle exists.

[Hint: feel free to use an algorithm from CS161. Be clear about which properties of the algorithm you're using.]

(d) Assume that all edge costs and capacities are integers with magnitude at most M. Give an algorithm that is guaranteed to terminate with a minimum-cost flow and has running time polynomial in $n = |V|$, $m = |E|$, and M.[3]

[Hint: what would the analog of Ford-Fulkerson be?]

Problem 12

The goal of this problem is to revisit two problems you studied in CS161 — the minimum spanning tree and shortest path problems — and to prove the optimality of Kruskal's and Dijkstra's algorithms via the complementary slackness conditions of judiciously chosen linear programs.

[3] Thus this algorithm is only "pseudo-polynomial." A polynomial algorithm would run in time polynomial in n, m, and $\log M$. Such algorithms can be derived for the minimum-cost flow problem using additional ideas.

(a) For convenience, we consider the maximum spanning tree problem (equivalent to the minimum spanning tree problem, after multiplying everything by -1). Consider a connected undirected graph $G = (V, E)$ in which each edge e has a weight w_e.

For a subset $F \subseteq E$, let $\kappa(F)$ denote the number of connected components in the subgraph (V, F). Prove that the spanning trees of G are in an objective function-preserving one-to-one correspondence with the 0-1 feasible solutions of the following linear program (with decision variables $\{x_e\}_{e \in E}$):

$$\max \sum_{e \in E} w_e x_e$$

subject to

$$\sum_{e \in F} x_e \leq |V| - \kappa(F) \qquad \text{for all } F \subseteq E$$

$$\sum_{e \in E} x_e = |V| - 1$$

$$x_e \geq 0 \qquad \text{for all } e \in E.$$

(While this linear program has a huge number of constraints, we are using it purely for the analysis of Kruskal's algorithm.)

(b) What is the dual of this linear program?

(c) What are the complementary slackness conditions?

(d) Recall that Kruskal's algorithm, adapted to the current maximization setting, works as follows: do a single pass over the edges from the highest weight to lowest weight (breaking ties arbitrarily), adding an edge to the solution-so-far if and only if it creates no cycle with previously chosen edges. Prove that the corresponding solution to the linear program in (a) is in fact an optimal solution to that linear program, by exhibiting a feasible solution to the dual program in (b) such that the complementary slackness conditions hold.[4]

[Hint: for the dual variables of the form y_F, it is enough to use only those that correspond to subsets $F \subseteq E$ that comprise the i edges with the largest weights (for some i).]

(e) Now consider the problem of computing a shortest path from s to t in a directed graph $G = (V, E)$ with a nonnegative cost c_e on each edge $e \in E$. Prove that every simple s-t path of G corresponds to a 0-1 feasible solution of the following linear program with the same objective function value:[5]

$$\min \sum_{e \in E} c_e x_e$$

subject to

$$\sum_{e \in \delta^+(S)} x_e \geq 1 \qquad \text{for all } S \subseteq V \text{ with } s \in S,\ t \notin S$$

$$x_e \geq 0 \qquad \text{for all } e \in E.$$

(Again, this huge linear program is for analysis only.)

(f) What is the dual of this linear program?

(g) What are the complementary slackness conditions?

[4] You can assume without proof that Kruskal's algorithm outputs a feasible solution (i.e., a spanning tree), and focus on proving its optimality.

[5] Recall that $\delta^+(S)$ denotes the edges sticking out of S.

(h) Let P denote the s-t path returned by Dijkstra's algorithm. Prove that the solution to the linear program in (e) corresponding to P is in fact an optimal solution to that linear program, by exhibiting a feasible solution to the dual program in (f) such that the complementary slackness conditions hold.

[Hint: it is enough to use only dual variables of the form y_S for subsets $S \subseteq V$ that comprise the first i vertices processed by Dijkstra's algorithm (for some i).]

CS261: Problem Set #3

Due by 11:59 PM on Tuesday, February 23, 2016

Instructions:

(1) Form a group of 1-3 students. You should turn in only one write-up for your entire group.

(2) Submission instructions: We are using Gradescope for the homework submissions. Go to www.gradescope.com to either login or create a new account. Use the course code 9B3BEM to register for CS261. Only one group member needs to submit the assignment. When submitting, please remember to add all group member names in Gradescope.

(3) Please type your solutions if possible and we encourage you to use the LaTeX template provided on the course home page.

(4) Write convincingly but not excessively.

(5) Some of these problems are difficult, so your group may not solve them all to completion. In this case, you can write up what you've got (subject to (3), above): partial proofs, lemmas, high-level ideas, counterexamples, and so on.

(6) Except where otherwise noted, you may refer to the course lecture notes *only*. You can also review any relevant materials from your undergraduate algorithms course.

(7) You can discuss the problems verbally at a high level with other groups. And of course, you are encouraged to contact the course staff (via Piazza or office hours) for additional help.

(8) If you discuss solution approaches with anyone outside of your group, you must list their names on the front page of your write-up.

(9) Refer to the course Web page for the late day policy.

Problem 13

This problem fills in some gaps in our proof sketch of strong linear programming duality.

(a) For this part, assume the version of Farkas's Lemma stated in Lecture #9, that given $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^m$, exactly one of the following statements holds: (i) there is an $\mathbf{x} \in \mathbb{R}^n$ such that $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{x} \geq 0$; (ii) there is a $\mathbf{y} \in \mathbb{R}^m$ such that $\mathbf{y}^T\mathbf{A} \geq 0$ and $\mathbf{y}^T\mathbf{b} < 0$.

Deduce from this a second version of Farkas's Lemma, stating that for $\mathbf{A}$ and $\mathbf{b}$ as above, exactly one of the following statements holds: (iii) there is an $\mathbf{x} \in \mathbb{R}^n$ such that $\mathbf{Ax} \leq \mathbf{b}$; (iv) there is a $\mathbf{y} \in \mathbb{R}^m$ such that $\mathbf{y} \geq 0$, $\mathbf{y}^T\mathbf{A} = 0$, and $\mathbf{y}^T\mathbf{b} < 0$.

[Hint: note the similarity between (i) and (iv). Also note that if (iv) has a solution, then it has a solution with $\mathbf{y}^T\mathbf{b} = -1$.]

(b) Use the second version of Farkas's Lemma to prove the following version of strong LP duality: if the linear programs

$$\max \mathbf{c}^T\mathbf{x}$$

subject to

$$\mathbf{Ax} \leq \mathbf{b}$$

with $\mathbf{x}$ unrestricted, and

$$\min \mathbf{b}^T\mathbf{y}$$

subject to

$$\mathbf{A}^T\mathbf{y} = \mathbf{c}, \mathbf{y} \geq 0$$

are both feasible, then they have equal optimal objective function values.

[Hint: weak duality is easy to prove directly. For strong duality, let γ^* denote the optimal objective function value of the dual linear program. Add the constraint $\mathbf{c}^T\mathbf{x} \geq \gamma^*$ to the primal linear program and use Farkas's Lemma to show that the feasible region is non-empty.]

Problem 14

Recall the *multicommodity flow* problem from Exercise 17. Recall the input consists of a directed graph $G = (V, E)$, k "commodities" or source-sink pairs $(s_1, t_1), \ldots, (s_k, t_k)$, and a positive capacity u_e for each edge.

Consider also the *multicut* problem, where the input is the same as in the multicommodity flow problem, and feasible solutions are subsets $F \subseteq E$ of edges such that, for every commodity (s_i, t_i), there is no s_i-t_i path in $G = (V, E \setminus F)$. (Assume that s_i and t_i are distinct for each i.) The *value* of a multicut F is just the total capacity $\sum_{e \in F} u_e$.

(a) Formulate the multicommodity flow problem as a linear program with one decision variable for each path P that travels from a source s_i to the corresponding sink t_i. Aside from nonnegativity constraints, there should be only be m constraints (one per edge).

[Note: this is a different linear programming formulation than the one asked for in Exercise 21.]

(b) Take the dual of the linear program in (a). Prove that every optimal 0-1 solution of this dual — i.e., among all feasible solutions that assign each decision variable the value 0 or 1, one of minimum objective function value — is the characteristic vector of a minimum-value multicut.

(c) Show by example that the optimal solution to this dual linear program can have objective function value strictly smaller than that of every 0-1 feasible solution. In light of your example, explain a sense in which there is no max-flow/min-cut theorem for multicommodity flows and multicuts.

Problem 15

This problem gives a linear-time (!) randomized algorithm for solving linear programs that have a large number m of constraints and a small number n of decision variables. (The constant in the linear-time guarantee $O(m)$ will depend exponentially on n.)

Consider a linear program of the form

$$\max \mathbf{c}^T\mathbf{x}$$

subject to

$$\mathbf{A}\mathbf{x} \leq \mathbf{b}.$$

For simplicity, assume that the linear program is feasible with a bounded feasible region, and let M be large enough that $|x_j| < M$ for every coordinate of every feasible solution. Assume also that the linear program is "non-degenerate," in the sense that no feasible point satisfies more than n constraints with equality. For example, in the plane (two decision variables), this just means that there does not exist three different constraints (i.e., halfplanes) whose boundaries meet at a common point. Finally, assume that the linear program has a unique optimal solution.[1]

Let $C = \{1, 2, \ldots, m\}$ denote the set of constraints of the linear program. Let B denote additional constraints asserting that $-M \leq x_j \leq M$ for every j. The high-level idea of the algorithm is: (i) drop a

[1] All of these simplifying assumptions can be removed without affecting the asymptotic running time; we leave the details to the interested reader.

random constraint and recursively compute the optimal solution $\mathbf{x}^*$ of the smaller linear program; (ii) if $\mathbf{x}^*$ is feasible for the original linear program, return it; (iii) else, if $\mathbf{x}^*$ violates the constraint $\mathbf{a}_i^T\mathbf{x} \leq b_i$, then change this inequality to an equality and recursively solve the resulting linear program.

More precisely, consider the following recursive algorithm with two arguments. The first argument C_1 is a subset of inequality constraints that must be satisfied (initially, equal to C). The second argument is a subset C_2 of constraints that must be satisfied with equality (initially, $\emptyset$). The responsibility of a recursive call is to return a point maximizing $\mathbf{c}^T\mathbf{x}$ over all points that satisfy all the constraints of $C_1 \cup B$ (as inequalities) and also those of C_2 (as equations).

Linear-Time Linear Programming

Input: two disjoint subsets $C_1, C_2 \subseteq C$ of constraints

Base case #1: if $|C_2| = n$, return the unique point that satisfies every constraint of C_2 with equality

Base case #2: if $|C_1| + |C_2| = n$, return the point that maximizes $\mathbf{c}^T\mathbf{x}$ subject to $\mathbf{a}_i^T\mathbf{x} \leq b_i$ for every $i \in C_1$, $\mathbf{a}_i^T\mathbf{x} = b_i$ for every $i \in C_2$, and the constraints in B

Recursive step:
choose $i \in C_1$ uniformly at random
recurse with the sets $C_1 \setminus \{i\}$ and C_2 to obtain a point $\mathbf{x}^*$
if $\mathbf{a}_i^T\mathbf{x}^* \leq b_i$ **then**
    return $\mathbf{x}^*$
else
    recurse with the sets $C_1 \setminus \{i\}$ and $C_2 \cup \{i\}$, and return the result

(a) Prove that this algorithm terminates with the optimal solution $\mathbf{x}^*$ of the original linear program.

[Hint: be sure to explain why, in the "else" case, it's OK to recurse with the ith constraint set to an equation.]

(b) Let $T(m, s)$ denote the expected number of recursive calls made by the algorithm to solve an instance with $|C_1| = m$ and $|C_2| = s$ (with the number n of variables fixed). Prove that T satisfies the following recurrence:

$$T(m,s) = \begin{cases} 1 & \text{if } s = n \text{ or } m + s = n \\ T(m-1,s) + \frac{n-s}{m} \cdot T(m-1,s+1) & \text{otherwise.} \end{cases}$$

[Hint: you should use the non-degeneracy assumption in this part.]

(c) Prove that $T(m, 0) \leq n! \cdot m$.

[Hint: it might be easiest to make the variable substitution $\delta = n - s$ and proceed by simultaneous induction on m and δ.]

(d) Conclude that, for every fixed constant n, the algorithm above can be implemented so that the expected running time is $O(m)$ (where the hidden constant can depend arbitrarily on n).

Problem 16

This problem considers a variant of the online decision-making problem. There are n "experts," where n is a power of 2.

Combining Expert Advice

At each time step $t = 1, 2, \ldots, T$:

- each expert offers a prediction of the realization of a binary event (e.g., whether a stock will go up or down)
- a decision-maker picks a probability distribution p^t over the possible realizations 0 and 1 of the event
- the actual realization $r^t \in \{0, 1\}$ of the event is revealed
- a 0 or 1 is chosen according to the distribution p^t, and a *mistake* occurs whenever it is different from r^t

You are promised that there is at least one omniscient expert who makes a correct prediction at every time step.

(a) Prove that the minimum worst-case number of mistakes that a deterministic algorithm can make is precisely $\log_2 n$.

(b) Prove that the minimum worst-case expected number of mistakes that a randomized algorithm can make is precisely $\frac{1}{2} \log_2 n$.

Problem 17

In Lecture #11 we saw that the follow-the-leader (FTL) algorithm, and more generally every deterministic algorithm, can have regret that grows linearly with T. This problem outlines a randomized variant of FTL, the *follow-the-perturbed-leader (FTPL)* algorithm, with worst-case regret comparable to that of the multiplicative weights algorithm. In the description of FTPL, we define each probability distribution p^t over actions implicitly through a randomized subroutine.

Follow-the-Perturbed-Leader (FTPL) Algorithm

for each action $a \in A$ **do**

- independently sample a geometric random variable with parameter η,[2] denoted by X_a

for each time step $t = 1, 2, \ldots, T$ **do**

- choose the action a that maximizes the perturbed cumulative reward $X_a + \sum_{u=1}^{t-1} r^u(a)$ so far

For convenience, assume that, at every time step t, there is no pair of actions whose (unperturbed) cumulative rewards-so-far differ by an integer.

(a) Prove that, at each time step $t = 1, 2, \ldots, T$, with probability at least $1 - \eta$, the largest perturbed cumulative reward of an action prior to t is more than 1 larger than the second-largest such perturbed reward.

[Hint: Sample the X_a's gradually by flipping coins only as needed, pausing once the action a^* with largest perturbed cumulative reward is identified. Resuming, only X_{a^*} is not yet fully determined. What can you say if the next coin flip comes up "tails?"]

[2] Equivalently, when repeatedly flipping a coin that comes up "heads" with probability η, count the number of flips up to and including the first "heads."

(b) As a thought experiment, consider the (unimplementable) algorithm that, at each time step t, picks the action that maximizes the perturbed cumulative reward $X_a + \sum_{u=1}^{t} r^u(a)$ over $a \in A$, *taking into account the current reward vector*. Prove that the regret of this algorithm is at most $\max_{a \in A} X_a$.

[Hint: Consider first the special case where $X_a = 0$ for all a. Iteratively transform the action sequence that always selects the best action in hindsight to the sequence chosen by the proposed algorithm. Work backward from time T, showing that the reward only increases with each step of the transformation.]

(c) Prove that $\mathbf{E}[\max_{a \in A} X_a] \leq b\eta^{-1} \ln n$, where n is the number of actions and $b > 0$ is a constant independent of η and n.

[Hint: use the definition of a geometric random variable and remind yourself about "the union bound."]

(d) Prove that, for a suitable choice of η, the worst-case expected regret of the FTPL algorithm is at most $b\sqrt{T \ln n}$, where $b > 0$ is a constant independent of n and T.

Problem 18

In this problem we'll show that there is no online algorithm for the online bipartite matching problem with competitive ratio better than $1 - \frac{1}{e} \approx 63.2\%$.

Consider the following probability distribution over online bipartite matching instances. There are n left-hand side vertices L, which are known up front. Let π be an ordering of L, chosen uniformly at random. The n vertices of the right-hand side R arrive one by one, with the ith vertex of R connected to the last $n - i + 1$ vertices of L (according to the random ordering π).

(a) Explain why $OPT = n$ for every such instance.

(b) Consider an arbitrary deterministic online algorithm $\mathbf{A}$. Prove that for every $i \in \{1, 2, \ldots, n\}$, the probability (over the choice of π) that $\mathbf{A}$ matches the ith vertex of L (according to π) is at most

$$\min\left\{\sum_{j=1}^{i} \frac{1}{n-j+1}, 1\right\}.$$

[Hint: for example, in the first iteration, assume that $\mathbf{A}$ matches the first vertex of R to the vertex $v \in L$. Note that $\mathbf{A}$ must make this decision without knowing π. What can you say if v does not happen to be the first vertex of π?]

(c) Prove that for every deterministic online algorithm $\mathbf{A}$, the expected (over π) size of the matching produced by $\mathbf{A}$ is at most

$$\sum_{i=1}^{n} \min\left\{\sum_{j=1}^{i} \frac{1}{n-j+1}, 1\right\}, \tag{1}$$

and prove that (1) approaches $n(1 - \frac{1}{e})$ as $n \to \infty$.

[Hint: for the second part, recall that $\sum_{j=1}^{d} \frac{1}{j} \approx \ln d$ (up to an additive constant less than 1). For what value of i is the inner sum roughly equal to 1?]

(d) Extend (c) to randomized online algorithms $\mathbf{A}$, where the expectation is now over both π and the internal coin flips of $\mathbf{A}$.

[Hint: use the fact that a randomized online algorithm is a probability distribution over deterministic online algorithms (as flipping all of $\mathbf{A}$'s coins in advance yields a deterministic algorithm).]

(e) Prove that for every $\epsilon > 0$ and (possibly randomized) online bipartite matching algorithm $\mathbf{A}$, there exists an input such that the expected (over $\mathbf{A}$'s coin flips) size of $\mathbf{A}$'s output is no more than $1 - \frac{1}{e} + \epsilon$ times that of an optimal solution.

CS261: Problem Set #4

Due by 11:59 PM on Tuesday, March 8, 2016

Instructions:

(1) Form a group of 1-3 students. You should turn in only one write-up for your entire group.

(2) Submission instructions: We are using Gradescope for the homework submissions. Go to www.gradescope.com to either login or create a new account. Use the course code 9B3BEM to register for CS261. Only one group member needs to submit the assignment. When submitting, please remember to add all group member names in Gradescope.

(3) Please type your solutions if possible and we encourage you to use the LaTeX template provided on the course home page.

(4) Write convincingly but not excessively.

(5) Some of these problems are difficult, so your group may not solve them all to completion. In this case, you can write up what you've got (subject to (3), above): partial proofs, lemmas, high-level ideas, counterexamples, and so on.

(6) Except where otherwise noted, you may refer to the course lecture notes *only*. You can also review any relevant materials from your undergraduate algorithms course.

(7) You can discuss the problems verbally at a high level with other groups. And of course, you are encouraged to contact the course staff (via Piazza or office hours) for additional help.

(8) If you discuss solution approaches with anyone outside of your group, you must list their names on the front page of your write-up.

(9) Refer to the course Web page for the late day policy.

Problem 19

This problem considers randomized algorithms for the online (integral) bipartite matching problem (as in Lecture #14).

(a) Consider the following algorithm: when a new vertex $w \in R$ arrives, among the unmatched neighbors of w (if any), choose one uniformly at random to match to w.

Prove that the competitive ratio of this algorithm is strictly smaller than $1 - \frac{1}{e}$.

(b) The remaining parts consider the following algorithm: before any vertices of R arrive, independently pick a number y_v uniformly at random from $[0, 1]$ for each vertex $v \in L$. Then, when a new vertex $w \in R$ arrives, match w to its unmatched neighbor with the smallest y-value (or to no one if all its neighbors are already matched).

For the analysis, when v and w are matched, define $q_v = g(y_v)$ and $q_w = 1 - g(y_v)$, where $g(y) = e^{y-1}$ is the same function used in Lecture #14.

Prove that with probability 1, at the end of the algorithm, $\sum_{v \in L \cup R} q_v$ equals the size of the computed matching.

(c) Fix an edge (v, w) in the final graph. Condition on the choice of y_x for every vertex $x \in L \cup R \setminus \{v\}$ other than v; q_v remains random. As a thought experiment, suppose we re-run the online algorithm from scratch with v deleted (the rest of the input and the y-values stay the same), and let $t \in L$ denote the vertex to which w is matched (if any).

Prove that the conditional expectation of q_v (given q_x for all $x \in L \cup R \setminus \{v\}$) is at least $\int_0^{y_t} g(z)dz$. (If t does not exist, interpret y_t as 1.)

[Hint: prove that v is matched (in the online algorithm with the original input, not in the thought experiment) whenever $y_v < y_t$. Conditioned on this event, what is the distribution of y_v?]

(d) Prove that, conditioned on q_x for all $x \in L \cup R \setminus \{v\}$, $q_w \geq 1 - g(y_t)$.

[Hint: prove that w is always matched (in the online algorithm with the original input) to a vertex with y-value at most y_t.]

(e) Prove that the randomized algorithm in (b) is $(1 - \frac{1}{e})$-competitive, meaning that for every input, the expected value of the computed matching (over the algorithm's coin flips) is at least $1 - \frac{1}{e}$ times the size of a maximum matching.

[Hint: use the expectation of the q-values to define a feasible dual solution.]

Problem 20

A set function $f : 2^U \to \mathbb{R}_+$ is *monotone* if $f(S) \leq f(T)$ whenever $S \subseteq T \subseteq U$. Such a function is *submodular* if it has diminishing returns: whenever $S \subseteq T \subseteq U$ and $i \notin T$, then

$$f(T \cup \{i\}) - f(T) \leq f(S \cup \{i\}) - f(S). \tag{1}$$

We consider the problem of, given a function f and a budget k, computing[1]

$$\max_{S \subseteq U : |S| = k} f(S). \tag{2}$$

(a) Prove that set coverage problem (Lecture #15) is a special case of this problem.

(b) Let $G = (V, E)$ be a directed graph and $p \in [0, 1]$ a parameter. Recall the cascade model from Lecture #15:

- Initially the vertices in some set S are "active," all other vertices are "inactive." Every edge is initially "undetermined."
- While there is an active vertex v and an undetermined edge (v, w):
  - with probability p, edge (v, w) is marked "active," otherwise it is marked "inactive;"
  - if (v, w) is active and w is inactive, then mark w as active.

Let $f(S)$ denote the expected number of active vertices at the conclusion of the cascade, given that the vertices of S are active at the beginning. (The expectation is over the coin flips made for the edges.) Prove that f is monotone and submodular.

[Hint: prove that the condition (1) is preserved under convex combinations.]

(c) Let f be a monotone submodular function. Define the greedy algorithm in the obvious way — at each of k iterations, add to S the element that increases f the most. Suppose at some iteration the current greedy solution is S and it decides to add i to S. Prove that

$$f(S \cup \{i\}) - f(S) \geq \frac{1}{k} \left(OPT - f(S) \right),$$

where OPT is the optimal value in (2).

[Hint: If you added every element in the optimal solution to S, where would you end up? Then use submodularity.]

[1] Don't worry about how f is represented in the input. We assume that it is possible to compute $f(S)$ from S in a reasonable amount of time.

(d) Prove that for every monotone submodular function f, the greedy algorithm is a $(1-\frac{1}{e})$-approximation algorithm.

Problem 21

This problem considers the "$\{1,2\}$" special case of the asymmetric traveling salesman problem (ATSP). The input is a complete directed graph $G=(V,E)$, with all $n(n-1)$ directed edges present, where each edge e has a cost c_e that is either 1 or 2. Note that the triangle inequality holds in every such graph.

(a) Explain why the $\{1,2\}$ special case of ATSP is NP-hard.

(b) Explain why it's trivial to obtain a polynomial-time 2-approximation algorithm for the $\{1,2\}$ special case of ATSP.

(c) This part considers a useful relaxation of the ATSP problem. A *cycle cover* of a directed graph $G=(V,E)$ is a collection $C_1,\ldots,C_k$ of simple directed cycles, each with at least two edges, such that every vertex of G belongs to exactly one of the cycles. (A traveling salesman tour is the special case where $k=1$.) Prove that given a directed graph with edge costs, a cycle cover with minimum total cost can be computed in polynomial time.

[Hint: bipartite matching.]

(d) Using (c) as a subroutine, give a $\frac{3}{2}$-approximation algorithm for the $\{1,2\}$ special case of the ATSP problem.

Problem 22

This problem gives an application of randomized linear programming rounding in approximation algorithms. In the *uniform labeling problem*, we are given an undirected graph $G=(V,E)$, costs $c_e \geq 0$ for all edges $e \in E$, and a set L of labels that can be assigned to the vertices of V. There is a non-negative cost $c_v^i \geq 0$ for assigning label $i \in L$ to vertex $v \in V$, and the edge cost c_e is incurred if and only if e's endpoints are given distinct labels. The goal of the problem is to assign each vertex a label so as to minimize the total cost.[2]

(a) Prove that the following is a linear programming relaxation of the problem:

$$\min \frac{1}{2}\sum_{e\in E} c_e \sum_{i\in L} z_e^i + \sum_{v\in V}\sum_{i\in L} c_v^i x_v^i$$

subject to

$$\sum_{i\in L} x_v^i = 1 \qquad \text{for all } v \in V$$

$$z_e^i \geq x_u^i - x_v^i \qquad \text{for all } e=(u,v)\in E \text{ and } i \in L$$

$$z_e^i \geq x_v^i - x_u^i \qquad \text{for all } e=(u,v)\in E \text{ and } i \in L$$

$$z_e^i \geq 0 \qquad \text{for all } e \in E \text{ and } i \in L$$

$$x_v^i \geq 0 \qquad \text{for all } v \in V \text{ and } i \in L.$$

Specifically, prove that for every feasible solution to the uniform labeling problem, there is a corresponding 0-1 feasible solution to this linear program that has the same objective function value.

[2] The motivation for the problem comes from image segmentation, generalizing the foreground-background segmentation problem discussed in Lecture #4.

(b) Consider now the following algorithm. First, the algorithm solves the linear programming relaxation above. The algorithm then proceeds in phases. In each phase, it picks a label $i \in L$ uniformly at random, and independently a number $\alpha \in [0, 1]$ uniformly at random. For each vertex $v \in V$ that has not yet been assigned a label, if $\alpha \leq x_v^i$, then we assign v the label i (otherwise it remains unassigned).

To begin the analysis of this randomized rounding algorithm, consider the start of a phase and suppose that the vertex $v \in V$ has not yet been assigned a label. Prove that (i) the probability that v is assigned the label i in the current phase is exactly $x_v^i/|L|$; and (ii) the probability that it is assigned some label in the current phase is exactly $1/|L|$.

(c) Prove that the algorithm assigns the label $i \in L$ to the vertex $v \in V$ with probability exactly x_v^i.

(d) We say that an edge e is *separated by a phase* if both endpoints were not assigned prior to the phase, and exactly one of the endpoints is assigned a label in this phase. Prove that, conditioned on neither endpoint being assigned yet, the probability that an edge e is separated by a given phase is at most $\frac{1}{|L|} \sum_{i \in L} z_e^i$.

(e) Prove that, for every edge e, the probability that the algorithm assigns different labels to e's endpoints is at most $\sum_{i \in L} z_e^i$.

[Hint: it might help to identify a sufficient condition for an edge $e = (u, v)$ to *not* be separated, and to relate the probability of this to the quantity $\sum_{i \in L} \min\{x_u^i, x_v^i\}$.]

(f) Prove that the expected cost of the solution returned by the algorithm is at most twice the cost of an optimal solution.

Problem 23

This problem explores *local search* as a technique for designing good approximation algorithms.

(a) In the *Max k-Cut* problem, the input is an undirected graph $G = (V, E)$ and a nonnegative weight w_e for each edge, and the goal is to partition V into at most k sets such that the sum of the weights of the cut edges — edges with endpoints in different sets of the partition — is as large as possible. The obvious local search algorithm for the problem is:

1. Initialize $(S_1, \ldots, S_k)$ to an arbitrary partition of V.
2. While there exists an improving move:

   [An *improving move* is a vertex $v \in S_i$ and a set S_j such that moving v from S_i to S_j strictly increases the objective function.]

   (a) Choose an arbitrary improving move and execute it — move the vertex v from S_i to S_j.

Since each iteration increases the objective function value, this algorithm cannot cycle and eventually terminates, at a "local maximum."

Prove that this local search algorithm is guaranteed to terminate at a solution with objective function value at least $\frac{k-1}{k}$ times the maximum possible.

[Hint: prove the statement first for $k = 2$; your argument should generalize easily. Also, you might find it easier to prove the stronger statement that the algorithm's final partition has objective function value at least $\frac{k-1}{k}$ times the sum of all the edge weights.]

(b) Recall the uniform metric labeling problem from Problem 22. We now give an equally good approximation algorithm based on local search.

Our local search algorithm uses the following local move. Given a current assignment of labels to vertices in V, it picks some label $i \in L$ and considers the minimum-cost *i-expansion* of the label i; that is, it considers the minimum-cost assignment of labels to vertices in V in which each vertex either keeps its current label or is relabeled with label i (note that all vertices currently with label i do not change their label). If the cost of the labeling from the i-expansion is cheaper than the current labeling, then

we switch to the labeling from the i-expansion. We continue until we find a locally optimal solution; that is, an assignment of labels to vertices such that every i-expansion can only increase the cost of the current assignment.

Give a polynomial-time algorithm that computes an improving i-expansion, or correctly decides that no such improving move exists.

[Hint: recall Lecture #4.]

(c) Prove that the local search algorithm in (b) is guaranteed to terminate at an assignment with cost at most twice the minimum possible.

[Hint: the optimal solution suggests some local moves. By assumption, these are not improving. What do these inequalities imply about the overall cost of the local minimum?]

Problem 24

This problem considers a natural clustering problem, where it's relatively easy to obtain a good approximation algorithm and a matching hardness of approximation bound.

The input to the *metric k-center* problem is the same as that in the metric TSP problem — a complete undirected graph $G = (V, E)$ where each edge e has a nonnegative cost c_e, and the edge costs satisfy the triangle inequality ($c_{uv} + c_{vw} \geq c_{uw}$ for all $u, v, w \in V$). Also given is a parameter k. Feasible solutions correspond to choices of k *centers*, meaning subsets $S \subseteq V$ of size k. The objective function is to minimize the furthest distance from a point to its nearest center:

$$\min_{S \subseteq V\,:\,|S|=k} \max_{v \in V} \min_{s \in S} c_{sv}. \tag{3}$$

We'll also refer to the well-known NP-complete *Dominating Set* problem, where given an undirected graph G and a parameter k, the goal is to decide whether or not G has a dominating set of size at most k.[3]

(a) (No need to hand in.) Let OPT denote the optimal objective function value (3). Observe that OPT equals the cost c_e of some edge, which immediately narrows down its possible values to a set of $\binom{n}{2}$ different possibilities (where $n = |V|$).

(b) Given an instance G to the metric k-center problem, let G_D denote the graph with vertices V and with an edge (u, v) if and only if the edge cost c_{uv} in G is at most $2D$. Prove that if we can efficiently compute a dominating set of size at most k in G_D, then we can efficiently compute a solution to the k-center instance that has objective function value at most $2D$.

(c) Prove that the following greedy algorithm computes a dominating set in G_{OPT} with size at most k:

- $S = \emptyset$
- While S is not a dominating set in G_{OPT}:
  - Let v be a vertex that is not in S and has no neighbor in S — there must be one, by the definition of a dominating set — and add v so S.

[Hint: the optimal k-center solution partitions the vertex set V into k "clusters," where the ith group consists of those vertices for which the ith center is the closest center. Argue that the algorithm above never picks two different vertices from the same cluster.]

(d) Put (a)–(c) together to obtain a 2-approximation algorithm for the metric k-center problem. (The running time of your algorithm should be polynomial in both n and k.)

(e) Using a reduction from the Dominating Set problem, prove that for every $\epsilon > 0$, there is no $(2 - \epsilon)$-approximation algorithm for the metric k-center problem, unless $P = NP$.

[Hint: look to our reduction to TSP (Lecture #16) for inspiration.]

[3] A *dominating set* is a subset $S \subseteq V$ of vertices such that every vertex $v \in V$ either belongs to S or has a neighbor in S.

CS261: A Second Course in Algorithms
The Top 10 List*

Tim Roughgarden†

March 10, 2016

If you've kept up with this class, then you've learned a tremendous amount of material. You know now more about algorithms than most people who don't have a PhD in the field, and are well prepared to tackle more advanced courses in theoretical computer science. To recall how far you've traveled, let's wrap up with a course top 10 list.

1. *The max-flow/min-cut theorem*, and the corresponding polynomial-time algorithms for computing them (augmenting paths, push-relabel, etc.). This is the theorem that seduced your instructor into a career in algorithms. Who knew that objects as seemingly complex and practically useful as flows and cuts could be so beautifully characterized?

   This theorem also introduced the running question of "how do we know when we're done?" We proved that a maximum flow algorithm is done (i.e., can correctly terminate with the current flow) when the residual graph contains no s-t path or, equivalently, when the current flow saturates some s-t cut.

2. *Bipartite matching,* including the Hungarian algorithm for the minimum-cost perfect bipartite matching problem. In this algorithm, we convinced ourselves we were done by exhibiting a suitable dual solution (which at the time we called "vertex prices") certifying optimality.

3. *Linear programming is in P.* We didn't have time to go into the details of any linear programming algorithms, but just knowing this fact as a "black box" is already extremely powerful. On the theoretical side, there are polynomial-time algorithms for solving linear programs — even those whose constraints are specified implicitly through a polynomial-time separation oracle — and countless theorems rely on this fact. In practice, commercial linear program solvers routinely solve problem instances with millions of variables and constraints and are a crucial tool in many real-world applications.

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

4. *Linear programming duality.* For linear programming problems, there's a generic way to know when you're done. Whatever the optimal solution of the linear program is, strong LP duality guarantees that there's a dual solution that proves its optimality. While powerful and perhaps surprising, the proof of strong duality boils down to the highly intuitive statement that, given a closed convex set and a point not in the set, there's a hyperplane with the set on one side and the point on the other.

5. *Online algorithms.* It's easy to think of real-world situations where decisions need to be made before all of the relevant information is available. In online algorithms, the input arrives "online" in pieces, and an irrevocable decision must be made at each time step. For some problems, there are online algorithms with good (close to 1) competitive ratios — algorithms that compute a solution with objective function value close to that of the optimal solution. Such algorithms perform almost as well as if the entire input was known in advance. For example, in online bipartite matching, we achieved a competitive ratio of $1 - \frac{1}{e} \approx 63\%$ (which is the best possible).

6. *The multiplicative weights algorithm.* This simple online algorithm, in the spirit of "reinforcement learning," achieves per-time-step regret approaching 0 as the time horizon T approaches infinity. That is, the algorithm does almost as well as the best fixed action in hindsight. This result is interesting in its own right as a strategy for making decisions over time. It also has some surprising applications, such as a proof of the minimax theorem for zero-sum games (if both players randomize optimally, then it doesn't matter who goes first) and fast approximation algorithms for several problems (maximum flow, multicommodity flow, etc.).

7. *The Traveling Salesman Problem (TSP).* The TSP is a famous NP-hard problem with a long history, and several of the most notorious open problems in approximation algorithms concern different variants of the TSP. For the metric TSP, you now know the state-of-the-art — Christofides's $\frac{3}{2}$-approximation algorithm, which is nearly 40 years old. Most researchers believe that better approximation algorithms exist. (You also know close to the state-of-the-art for asymmetric TSP, where again it seems that better approximation algorithms should exist.)

8. *Linear programming and approximation algorithms.* Linear programs are useful not only for solving problems exactly in polynomial time, but also in the design and analysis of polynomial-time approximation algorithms for NP-hard optimization problems. In some cases, linear programming is used only in the analysis of an algorithm, and not explicitly in the algorithm itself. A good example is our analysis of the greedy set cover algorithm, where we used a feasible dual solution as a lower bound on the cost of an optimal set cover. In other applications, such as vertex cover and low-congestion routing, the approximation algorithm first explicitly solves an LP relaxation of the problem, and then "rounds" the resulting fractional solution into a near-optimal integral solution. Finally, some algorithms, like our primal-dual algorithm for vertex

cover, use linear programs to guide their decisions, without ever explicitly or exactly solving the linear programs.

9. *Five essential tools for the analysis of randomized algorithms.* And in particular, the Chernoff bounds, which prove sharp concentration around the expected value for random variables that are sums of bounded independent random variables. Chernoff bounds are used *all the time*. We saw an application in randomized rounding, leading to a $O(\log n / \log \log n)$-approximation algorithm for low-congestion routing.

   We also reviewed four easy-to-prove tools that you've probably seen before: linearity of expectation (which is trivial but super-useful), Markov's inequality (which is good for constant-probability bounds), Chebyshev's inequality (good for random variables with small variance), and the union bound (which is good for avoiding lots of low-probability events simultaneously).

10. *Beating brute-force search.* NP-hardness is not a death sentence — it just means that you need to make some compromises. In approximation algorithms, one insists on a polynomial running time and compromises on correctness (i.e., on exact optimality). But one can also insist on correctness, resigning oneself to an exponential running time (but still as fast as possible). We saw three examples of NP-hard problems that admit exact algorithms that are significantly faster than brute-force search: the unweighted vertex cover problem (an example of a "fixed-parameter tractable" algorithm, with running time of the form $\text{poly}(n) + f(k)$ rather than $O(n^k)$); TSP (where dynamic programming reduces the running time from roughly $O(n!)$ to roughly $O(2^n)$); and 3SAT (where random search reduces the running time from roughly $O(2^n)$ to roughly $O((4/3)^n)$).