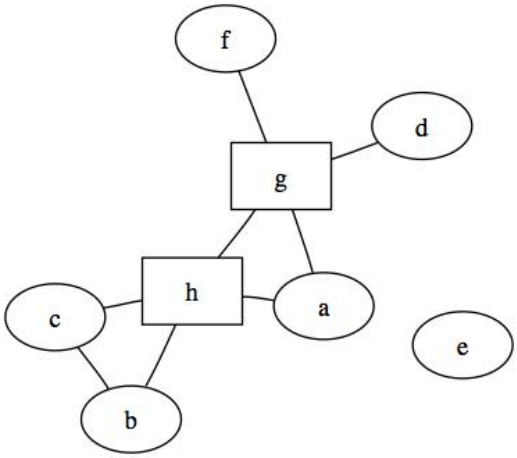
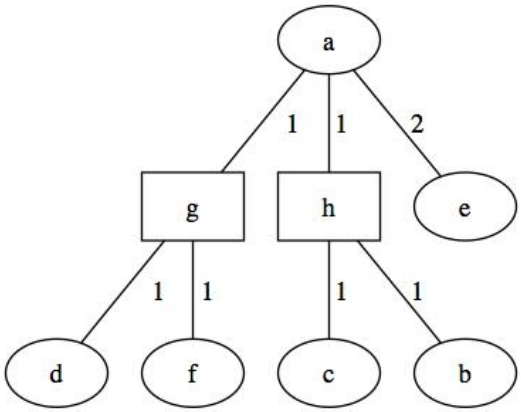
Combinatorial Optimization: Exact and Approximate Algorithms





Luca Trevisan

Stanford University

March 19, 2011

Foreword

These are minimally edited lecture notes from the class *CS261: Optimization and Algorithmic Paradigms* that I taught at Stanford in the Winter 2011 term. The following 18 lectures cover topics in approximation algorithms, exact optimization, and online algorithms.

I gratefully acknowledge the support of the National Science Foundation, under grant CCF 1017403. Any opinions, findings and conclusions or recommendations expressed in these notes are my own and do not necessarily reflect the views of the National Science Foundation.

Luca Trevisan, San Francisco, March 19, 2011.

Contents

Lecture 1

Introduction

In which we describe what this course is about and give a simple example of an approximation algorithm

1.1 Overview

In this course we study algorithms for combinatorial optimization problems. Those are the type of algorithms that arise in countless applications, from billion-dollar operations to everyday computing task; they are used by airline companies to schedule and price their flights, by large companies to decide what and where to stock in their warehouses, by delivery companies to decide the routes of their delivery trucks, by Netflix to decide which movies to recommend you, by a gps navigator to come up with driving directions and by word-processors to decide where to introduce blank spaces to justify (align on both sides) a paragraph.

In this course we will focus on general and powerful algorithmic techniques, and we will apply them, for the most part, to highly idealized model problems.

Some of the problems that we will study, along with several problems arising in practice, are NP-hard, and so it is unlikely that we can design exact efficient algorithms for them. For such problems, we will study algorithms that are worst-case efficient, but that output solutions that can be sub-optimal. We will be able, however, to prove worst-case bounds to the ratio between the cost of optimal solutions and the cost of the solutions provided by our algorithms. Sub-optimal algorithms with provable guarantees about the quality of their output solutions are called *approximation algorithms*.

The content of the course will be as follows:

- *Simple examples of approximation algorithms.* We will look at approximation algorithms for the Vertex Cover and Set Cover problems, for the Steiner Tree Problem and for the Traveling Salesman Problem. Those algorithms and their analyses will

be relatively simple, but they will introduce a number of key concepts, including the importance of getting upper bounds on the cost of an optimal solution.

- *Linear Programming.* A linear program is an optimization problem over the real numbers in which we want to optimize a linear function of a set of real variables subject to a system of linear inequalities about those variables. For example, the following is a linear program:

$$\begin{aligned} &\text{maximize } x_1 + x_2 + x_3 \\ &\text{Subject to :} \\ &2x_1 + x_2 \leq 2 \\ &x_2 + 2x_3 \leq 1 \end{aligned}$$

(A linear program is not a *program* as in *computer program*; here *programming* is used to mean *planning.*) An optimum solution to the above linear program is, for example, $x_1 = 1/2$, $x_2 = 1$, $x_3 = 0$, which has cost 1.5. One way to see that it is an optimal solution is to sum the two linear constraints, which tells us that in every admissible solution we have

$$2x_1 + 2x_2 + 2x_3 \leq 3$$

that is, $x_1 + x_2 + x_3 \leq 1.5$. The fact that we were able to verify the optimality of a solution by summing inequalities is a special case of the important theory of *duality* of linear programming.

A linear program is an optimization problem over real-valued variables, while this course is about *combinatorial* problems, that is problems with a finite number of discrete solutions. The reasons why we will study linear programming are that

1. Linear programs can be solved in polynomial time, and very efficiently in practice;
2. All the combinatorial problems that we will study can be written as linear programs, provided that one adds the additional requirement that the variables only take *integer value.*

This leads to two applications:

1. If we take the integral linear programming formulation of a problem, we remove the integrality requirement, we solve it efficient as a linear program over the real numbers, and we are lucky enough that the optimal solution happens to have integer values, then we have the optimal solution for our combinatorial problem. For some problems, it can be proved that, in fact, this will happen for every input.
2. If we take the integral linear programming formulation of a problem, we remove the integrality requirement, we solve it efficient as a linear program over the real numbers, we find a solution with fractional values, but then we are able to "round" the fractional values to integer ones without changing the cost of the

solution too much, then we have an efficient *approximation algorithm* for our problem.

- *Approximation Algorithms via Linear Programming.* We will give various examples in which approximation algorithms can be designed by "rounding" the fractional optima of linear programs.
- *Exact Algorithms for Flows and Matchings.* We will study some of the most elegant and useful optimization algorithms, those that find optimal solutions to "flow" and "matching" problems.
- *Linear Programming, Flows and Matchings.* We will show that flow and matching problems can be solved optimally via linear programming. Understanding why will make us give a second look at the theory of linear programming duality.
- *Online Algorithms.* An online algorithm is an algorithm that receives its input as a stream, and, at any given time, it has to make decisions only based on the partial amount of data seen so far. We will study two typical online settings: paging (and, in general, data transfer in hierarchical memories) and investing.

1.2 The Vertex Cover Problem

1.2.1 Definitions

Given an undirected graph $G = (V, E)$, a *vertex cover* is a subset of vertices $C \subseteq V$ such that for every edge $(u, v) \in E$ at least one of u or v is an element of C.

In the *minimum vertex cover* problem, we are given in input a graph and the goal is to find a vertex cover containing as few vertices as possible.

The minimum vertex cover problem is very related to the *maximum independent set* problem. In a graph $G = (V, E)$ an *independent set* is a subset $I \subseteq V$ of vertices such that there is no edge $(u, v) \in E$ having both endpoints u and v contained in I. In the maximum independent set problem the goal is to find a largest possible independent set.

It is easy to see that, in a graph $G = (V, E)$, a set $C \subseteq V$ is a vertex cover if and only if its complement $V - C$ is an independent set, and so, from the point of view of exact solutions, the two problems are equivalent: if C is an optimal vertex cover for the graph G then $V - C$ is an optimal independent set for G, and if I is an optimal independent set then $V - I$ is an optimal vertex cover.

From the point of view of approximation, however, the two problems are not equivalent. We are going to describe a linear time 2-approximate algorithm for minimum vertex cover, that is an algorithm that finds a vertex cover of size at most twice the optimal size. It is known, however, that no constant-factor, polynomial-time, approximation algorithms can exist for the independent set problem. To see why there is no contradiction (and how the notion of approximation is highly dependent on the cost function), suppose that we have a graph with n vertices in which the optimal vertex cover has size $.9 \cdot n$, and that our algorithm

finds a vertex cover of size $n-1$. Then the algorithm finds a solution that is only about 11% larger than the optimum, which is not bad. From the point of view of independent set size, however, we have a graph in which the optimum independent set has size $n/10$, and our algorithm only finds an independent set of size 1, which is terrible

1.2.2 The Algorithm

The algorithm is very simple, although not entirely natural:

- Input: graph $G = (V, E)$
- $C := \emptyset$
- while there is an edge $(u, v) \in E$ such that $u \notin C$ and $v \notin C$
  - $C := C \cup \{u, v\}$
- return C

We initialize our set to the empty set, and, while it fails to be a vertex cover because some edge is uncovered, we add *both* endpoints of the edge to the set. By the time we are finished with the *while* loop, C is such that for every edge $(u, v) \in E$, either $u \in C$ or $v \in C$ (or both), that is, C is a vertex cover.

To analyze the approximation, let *opt* be the number of vertices in a minimal vertex cover, then we observe that

- If $M \subseteq E$ is a *matching*, that is, a set of edges that have no endpoint in common, then we must have $opt \geq |M|$, because every edge in M must be covered using a distinct vertex.
- The set of edges that are considered inside the *while* loop form a matching, because if (u, v) and (u', v') are two edges considered in the *while* loop, and (u, v) is the one that is considered first, then the set C contains u and v when (u', v') is being considered, and hence u, v, u', v' are all distinct.
- If we let M denote the set of edges considered in the *while* cycle of the algorithm, and we let C_{out} be the set given in output by the algorithm, then we have

$$|C_{out}| = 2 \cdot |M| \leq 2 \cdot opt$$

As we said before, there is something a bit unnatural about the algorithm. Every time we find an edge (u, v) that violates the condition that C is a vertex cover, we add *both* vertices u and v to C, even though adding just one of them would suffice to cover the edge (u, v). Isn't it an overkill?

Consider the following alternative algorithm that adds only one vertex at a time:

- Input: graph $G = (V, E)$
- $C := \emptyset$
- while there is an edge $(u, v) \in E$ such that $u \notin C$ and $v \notin C$
  - $C := C \cup \{u\}$
- return C

This is a problem if our graph is a "star." Then the optimum is to pick the center, while the above algorithm might, in the worse case, pick all the vertices except the center.

Another alternative would be a greedy algorithm:

- Input: graph $G = (V, E)$
- $C := \emptyset$
- while C is not a vertex cover
  - let u be the vertex incident on the most uncovered edges
  - $C := C \cup \{u\}$
- return C

The above greedy algorithm also works rather poorly. For every n, we can construct an n vertex graph where the optimum is roughly $n/\ln n$, but the algorithm finds a solution of cost roughly $n - n/\ln n$, so that it does not achieve a constant-factor approximation of the optimum. We will return to this greedy approach and to these bad examples when we talk about the *minimum set cover* problem.

Lecture 2

Steiner Tree Approximation

In which we define the Steiner Tree problem, we show the equivalence of metric and general Steiner tree, and we give a 2-approximate algorithm for both problems.

2.1 Approximating the Metric Steiner Tree Problem

The Metric Steiner Tree Problem is defined as follows: the input is a set $X = R \cup S$ of points, where R is a set of *required* points and S is a set of *optional* points, and a symmetric distance function $d : X \times X \to \mathbb{R}_{\geq 0}$ that associates a non-negative distance $d(x, y) = d(y, x) \geq 0$ to every pair of points. We restrict the problem to the case in which d satisfies the *triangle inequality*, that is,

$$\forall x, y, z \in X. d(x, z) \leq d(x, y) + d(y, z)$$

In such a case, d is called a (semi-)metric, hence the name of the problem.

The goal is to find a tree $T = (V, E)$, where V is any set $R \subseteq V \subseteq X$ of points that includes all of the required points, and possibly some of the optional points, such that the *cost*

$$cost_d(T) := \sum_{(u,v) \in E} d(u, v)$$

of the tree is minimized.

This problem is very similar to the *minimum spanning tree* problem, which we know to have an exact algorithm that runs in polynomial (in fact, nearly linear) time. In the minimum spanning tree problem, we are given a weighted graph, which we can think of as a set of points together with a distance function (which might not satisfy the triangle inequality), and we want to find the tree of minimal total length that *spans all the vertices*. The difference is that in the minimum Steiner tree problem we only require to span a subset of

vertices, and other vertices are included only if they are beneficial to constructing a tree of lower total length.

We consider the following very simple approximation algorithm: *run a minimum spanning tree algorithm on the set of required vertices*, that is, find the best possible tree that uses none of the optional vertices.

We claim that this algorithm is 2-approximate, that is, it finds a solution whose cost is at most twice the optimal cost.

To do so, we prove the following.

Lemma 2.1 *Let $(X = R \cup S, d)$ be an instance of the metric Steiner tree problem, and $T = (V, E)$ be a Steiner tree with $R \subseteq V \subseteq X$.*

Then there is a tree $T' = (R, E')$ which spans the vertices in R and only the vertices in R such that

$$cost_d(T') \leq 2 \cdot cost_d(T)$$

In particular, applying the Lemma to the optimal Steiner tree we see that there is a spanning tree of R whose cost is at most twice the cost of the optimal Steiner tree. This also means that the minimal spanning tree of R also has cost at most twice the cost of the optimal Steiner tree.

PROOF: [Of Lemma 2.1] Consider a DFS traversal of T, that is, a sequence

$$x_0, x_1, x_2, \ldots, x_m = x_0$$

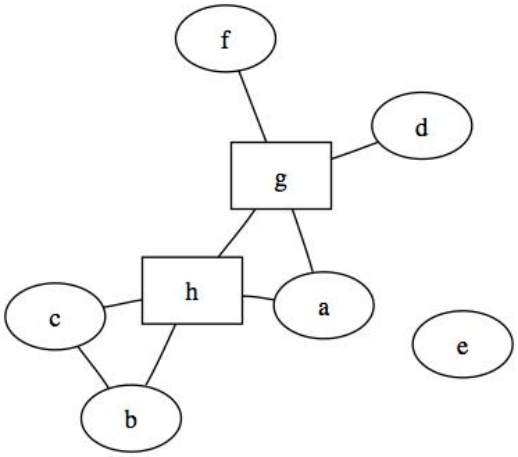
listing the vertices of T in the order in which they are considered during a DFS, including each time we return to a vertex at the end of each recursive call. The sequence describes a cycle over the elements of V whose total length $\sum_{i=0}^{m} d(x_i, x_{i+1})$ is precisely $2 \cdot cost_d(T)$, because the cycle uses each edge of the tree precisely twice.

Let now $y_0, y_1, \ldots, y_k$ be the sequence obtained from $x_0, \ldots, x_m$ by removing the vertices in S and keeping only the first occurrent of each vertex in R.

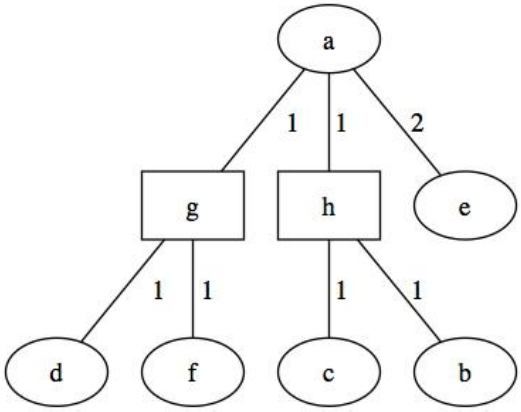
Then $y_0, \ldots, y_k$ is a path that includes all the vertices of R, and no other vertex, and its cost $\sum_{i=0}^{k} d(y_i, y_{i+1})$ is at most the cost of the cycle $x_0, x_1, x_2, \ldots, x_m$ (here we are using the triangle inequality), and so it is at most $2 \cdot cost_d(T)$.

But now note that $y_0, \ldots, y_k$, being a path, is also a tree, and so we can take T' to be tree (R, E') where E' is the edge set $\{(y_i, y_{i+1})\}_{i=0,\ldots,k}$. □

For example, if we have an instance in which $R = \{a, b, c, d, e, f\}$, $S = \{g, h\}$, and the distance function $d(\cdot, \cdot)$ assigns distance 1 to the points connected by an edge in the graph below, and distance 2 otherwise



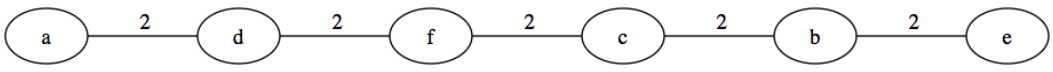
Then the following is a Steiner tree for our input whose cost is 8:



We use the argument in the proof of the Lemma to show that there is a Spanning tree of R of cost at most 16. (In fact we will do better.)

The order in which we visit vertices in a DFS of T is $a \to g \to d \to g \to f \to g \to a \to h \to c \to h \to b \to h \to a \to e \to a$. If we consider it as a loop that starts at a and goes back to a after touching all vertices, some vertices more than once, then the loop has cost 16, because it uses every edge exactly twice.

Now we note that if we take the DFS traversal, and we skip all the optional vertices and all the vertices previously visited, we obtain an order in which to visit all the required vertices, and no other vertex. In the example the order is $a \to d \to f \to c \to b \to e$.



Because this path was obtained by "shortcutting" a path of cost at most twice the cost of T, and because we have the triangle inequality, the path that we find has also cost at most

twice that of T. In our example, the cost is just 10. Since a path is, in particular, a tree, we have found a spanning tree of R whose cost is at most twice the cost of T.

The factor of 2 in the lemma cannot be improved, because there are instances of the Metric Steiner Tree problem in which the cost of the minimum spanning tree of R is, in fact, arbitrarily close to twice the cost of the minimum steiner tree.

Consider an instance in which $S = \{v_0\}$, $R = \{v_1, \ldots, v_n\}$, $d(v_0, v_i) = 1$ for $i = 1, \ldots, n$, and $d(v_i, v_j) = 2$ for all $1 \leq i < j \leq n$. That is, consider an instance in which the required points are all at distance two from each other, but they are all at distance one from the unique optional point. Then the minimum Steiner tree has v_0 as a root and the nodes $v_1, \ldots, v_n$ as leaves, and it has cost n, but the minimum spanning tree of R has cost $2n - 2$, because it is a tree with n nodes and $n - 1$ edges, and each edge is of cost 2.

2.2 Metric versus General Steiner Tree

The General Steiner Tree problem is like the Metric Steiner Tree problem, but we allow arbitrary distance functions.

In this case, it is not true any more that a minimum spanning tree of R gives a good approximation: consider the case in which $R = \{a, b\}$, $S = \{c\}$, $d(a, b) = 10^{100}$, $d(a, c) = 1$ and $d(b, c) = 1$. Then the minimum spanning tree of R has cost 10^{100} while the minimum Steiner tree has cost 2.

We can show, however, that our 2-approximation algorithm for Metric Steiner Tree can be turned, with some care, into a 2-approximation algorithm for General Steiner Tree.

Lemma 2.2 *For every $c \geq 1$, if there is a polynomial time c-approximate algorithm for Metric Steiner Tree, then there is a polynomial time c-approximate algorithm for General Steiner Tree.*

PROOF: Suppose that we have a polynomial-time c-approximate algorithm A for Metric Steiner Tree and that we are given in input an instance $(X = R \cup S, d)$ of General Steiner Tree. We show how to find, in polynomial time, a c-approximate solution for (X, d).

For every two points $x, y \in X$, let $d'(x, y)$ be the length of a shortest path from x to y in the weighted graph of vertex set X of weights $d(\cdot, \cdot)$. Note that $d'(\cdot, \cdot)$ is a distance function that satisfies the triangle inequality, because for every three points x, y, z it must be the case that the length of the shortest path from x to z cannot be any more than the length of the shortest path from x to y plus the length of the shortest path from y to z.

This means that (X, d') is an instance of Metric Steiner Tree, and we can apply algorithm A to it, and find a tree $T' = (V', E)$ of cost

$$cost_{d'}(T') \leq c \cdot opt(X, d')$$

Now notice that, for every pair of points, $d'(x, y) \leq d(x, y)$, and so if T^* is the optimal tree of our original input (X, d) we have

$$opt(X, d') \leq cost_{d'}(T^*) \leq cost_d(T^*) = opt(X, d)$$

So putting all together we have

$$cost_{d'}(T') \leq c \cdot opt(X, d)$$

Now, from T', construct a graph $G = (V, E)$ by replacing each edge (x, y) by the shortest path from x to y according to $d(\cdot)$. By our construction we have

$$cost_d(G) = \sum_{(x,y)\in E} d(x, y) \leq \sum_{(x,y)\in E'} d'(x, y) = cost_{d'}(T')$$

Note also that G is a connected graph.

The reason why we have an inequality instead of an equality is that certain edges of G might belong to more than one shortest path, so they are counted only once on the left-hand side.

Finally, take a minimum spanning tree T of G according to the weights $d(\cdot, \cdot)$. Now T is a valid Steiner tree, and we have

$$cost_d(T) \leq cost_d(G) \leq c \cdot opt(X, d)$$

□

Lecture 3

TSP and Eulerian Cycles

In which we prove the equivalence of three versions of the Traveling Salesman Problem, we provide a 2-approximate algorithm, we review the notion of Eulerian cycle, we think of the TSP as the problem of finding a minimum-cost connected Eulerian graph, and we revisit the 2-approximate algorithm from this perspective.

3.1 The Traveling Salesman Problem

In the Traveling Salesman Problem (abbreviated TSP) we are given a set of points X and a symmetric distance function $d : X \times X \to \mathbb{R}_{\geq 0}$. The goal is to find a cycle that reaches all points in X and whose total length is as short as possible.

For example, a shuttle driver that picks up seven people at SFO and needs to take them to their home and then go back to the airport faces a TSP instance in which X includes eight points (the seven home addresses and the airport), and the distance function is the driving time between two places. A DHL van driver who has to make a series of delivery and then go back to the warehouse has a similar problem. Indeed TSP is a basic model for several concrete problems, and it one of the most well studied problems in combinatorial optimization.

There are different versions of this problem depending on whether we require $d(\cdot, \cdot)$ to satisfy the triangle inequality or not, and whether we allow the loop to pass through the same point more than once.

1. *General TSP without repetitions* (General TSP-NR): we allow arbitrary symmetric distance functions, and we require the solution to be a cycle that contains every point exactly once;

2. *General TSP with repetitions* (General TSP-R): we allow arbitrary symmetric distance functions, and we allow all cycles as an admissible solution, even those that contain some point more than once;

3. *Metric TSP without repetitions* (Metric TSP-NR): we only allow inputs in which the distance function $d(\cdot,\cdot)$ satisfies the triangle inequality, that is

$$\forall x, y, z \in X.\ d(x,z) \leq d(x,y) + d(y,z)$$

and we only allow solutions in which each point is reached exactly once;

4. *Metric TSP with repetitions* (Metric TSP-R): we only allow inputs in which the distance function satisfies the triangle inequality, and we allow all cycles as admissible solutions, even those that contain some point more than once.

For all versions, it is NP-hard to find an optimal solution.

If we allow arbitrary distance functions, and we require the solution to be a cycle that reaches every point exactly once, then we have a problem for which no kind of efficient approximation is possible.

Fact 3.1 *If* $\mathbf{P} \neq \mathbf{NP}$ *then there is no polynomial-time constant-factor approximation algorithm for General TSP-NR.*

More generally, if there is a function $r : \mathbb{N} \to \mathbb{N}$ *such that* $r(n)$ *can be computable in time polynomial in* n *(for example,* $r(n) = 2^{100} \cdot 2^{n^2}$*), and a polynomial time algorithm that, on input an instance* (X, d) *of General TSP-NR with* n *points finds a solution of cost at most* $r(n)$ *times the optimum, then* $\mathbf{P} = \mathbf{NP}$*.*

The other three versions, however, are completely equivalent from the point of view of approximation and, as we will see, can be efficiently approximated reasonably well.

Lemma 3.2 *For every* $c \geq 1$*, there is a polynomial time* c*-approximate algorithm for Metric TSP-NR if and only if there is a polynomial time* c*-approximate approximation algorithm for Metric TSP-R. In particular:*

1. *If* (X, d) *is a Metric TSP input, then the cost of the optimum is the same whether or not we allow repetitions.*

2. *Every* c*-approximate algorithm for Metric TSP-NR is also a* c*-approximate algorithm for Metric TSP-R.*

3. *Every* c*-approximate algorithm for Metric TSP-R can be turned into a* c*-approximate algorithm for Metric TSP-NR after adding a linear-time post-processing step.*

PROOF: Let $opt_{TSP-R}(X, d)$ be the cost of an optimal solution for (X, d) among all solutions with or without repetitions, and $opt_{TSP-NR}(X, d)$ be the cost of an optimal solution for (X, d) among all solutions without repetitions. Then clearly in the former case we are minimizing over a larger set of possible solutions, and so

$$opt_{TSP-R}(X,d) \leq opt_{TSP-NR}(X,d)$$

Consider now any cycle, possibly with repetitions, $C = v_0 \to v_1 \to v_2 \to \cdots v_{m-1} \to v_m = v_0$. Create a new cycle C' from C by removing from C all the repeated occurrences of any vertex. (With the special case of v_0 which is repeated at the end.) For example, the cycle $C = a \to c \to b \to a \to d \to b \to a$ becomes $a \to c \to b \to d \to a$. Because of the triangle inequality, the total length of C' is at most the total length of C, and C' is a cycle with no repetitions. If we apply the above process by taking C to be the optimal solution of (X,d) allowing repetitions, we see that

$$opt_{TSP-R}(X,d) \geq opt_{TSP-NR}(X,d)$$

and so we have proved our first claim that $opt_{TSP-R}(X,d) = opt_{TSP-NR}(X,d)$.

Regarding the second claim, suppose that we have a c-approximate algorithm for Metric TSP-NR. Then, given an input (X,d) the algorithm finds a cycle C with no repetitions such that

$$cost_d(C) \leq c \cdot opt_{TSP-NR}(X,d)$$

but C is also an admissible solution for the problem Metric TSP-R, and

$$cost_d(C) \leq c \cdot opt_{TSP-NR}(X,d) = c \cdot opt_{TSP-R}(X,d)$$

and so our algorithm is also a c-approximate algorithm for opt_{TSP-NR}.

To prove the third claim, suppose we have a c-approximate algorithm for Metric TSP-R. Then, given an input (X,d) the algorithm finds a cycle C, possibly with repetitions, such that

$$cost_d(C) \leq c \cdot opt_{TSP-R}(X,d)$$

Now, convert C to a solution C' that has no repetitions and such that $cost_d(C') \leq cost_d(C)$ as described above, and output the solution C'. We have just described a c-approximate algorithm for Metric TSP-NR, because

$$cost_d(C') \leq cost_d(C) \leq c \cdot opt_{TSP-R}(X,d) = c \cdot opt_{TSP-NR}(X,d)$$

□

Lemma 3.3 *For every $c \geq 1$, there is a polynomial time c-approximate algorithm for Metric TSP-NR if and only if there is a polynomial time c-approximate approximation algorithm for General TSP-R. In particular:*

1. *Every c-approximate algorithm for General TSP-R is also a c-approximate algorithm for Metric TSP-R.*

2. *Every c-approximate algorithm for Metric TSP-R can be turned into a c-approximate algorithm for General TSP-R after adding polynomial-time pre-processing and post-processing steps.*

PROOF: The first claim just follows from the fact that General and Metric TSP-R are the same problem, except that in the general problem we allow a larger set of admissible inputs.

The proof of the second claim is similar to the proof that an approximation algorithm for Metric Steiner Tree can be converted to an approximation algorithm for General Steiner Tree.

Consider the following algorithm: on input an instance (X, d) of General TSP-R, we compute the new distance function $d'(\cdot, \cdot)$ such that $d'(x, y)$ is the length of a shortest path from x to y in a weighted graph that has vertex set X and weights $d(\cdot, \cdot)$. Note that $d'(\cdot, \cdot)$ is a distance function that satisfies the triangle inequality. We also compute the shortest path between any pair x, y.

We then pass the input (X, d') to our c-approximate algorithm for Metric TSP-R, and find a cycle C' such that

$$cost_{d'}(C') \leq c \cdot opt_{TSP-R}(X, d')$$

Note that, for every pair of points (x, y), we have $d'(x, y) \leq d(x, y)$ and so this implies that

$$opt_{TSP-R}(X, d') \leq opt_{TSP-R}(X, d)$$

Finally, we construct a cycle C by replacing each transition $x \to y$ in C' by the shortest path from x to y according to $d(\cdot, \cdot)$. Because of the definition of $d'(\cdot, \cdot)$, we have

$$cost_d(C) = cost_{d'}(C')$$

and, combining the inequalities we have proved so far,

$$cost_d(C) \leq c \cdot opt_{TSP-R}(X, d)$$

meaning that the algorithm that we have described is a c-approximate algorithm for General TSP-R. □

3.2 A 2-approximate Algorithm

When discussing approximation algorithms for the Minimum Steiner Tree problem in the last lecture, we proved (without stating it explicitly) the following result.

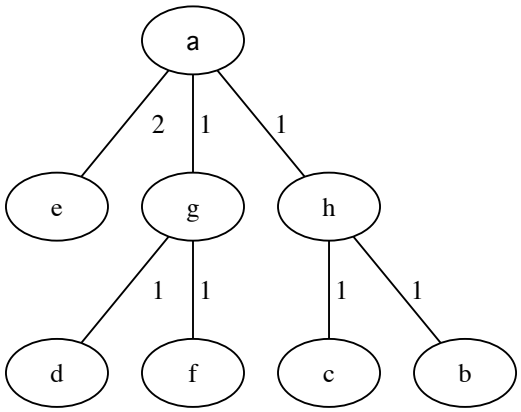
Lemma 3.4 *Let $T(X, E)$ be a tree over a set of vertices X, and $d : X \times X \to \mathbb{R}_{\geq 0}$ a symmetric distance function. Then there is a cycle $C = v_0 \to v_1 \to \cdots \to v_m = v_0$ that reaches every vertex at least once, and such that*

$$cost_d(C) = 2 \cdot cost_d(T)$$

where $cost_d(C) = \sum_{i=0,\ldots m-1} d(v_i, v_{i+1})$ and $cost_d(T) = \sum_{(x,y)\in E} d(x, y)$.

The proof is simply to consider a DFS visit of the tree, starting at the root; we define C to be the order in which vertices are visited, counting both the beginning of the recursive call in which they are reached, and also the time when we return at a vertex after the end of a recursive call originated at the vertex.

For example, revisiting an example from the last lecture, from the tree



We get the cycle $a \to e \to a \to g \to d \to g \to f \to g \to a \to h \to c \to b \to h \to a$, in which every point is visited at least once (indeed, a number of times equal to its degree in the tree) and every edge is traversed precisely twice.

Theorem 3.5 *There is a polynomial-time 2-approximate algorithm for General TSP-R. (And hence for Metric TSP-NR and Metric TSP-R.)*

PROOF: The algorithm is very simple: on input a (X, d) we find a minimum spanning tree T of the weighted graph with vertex set X and weights d, and then we find the cycle C of cost $2cost_d(T)$ as in Lemma 3.4.

It remains to prove that $opt_{TSP-R}(X, d) \geq opt_{MST}(X, d) = cost_d(T)$, which will then imply that we have found a solution whose cost is $\leq 2 \cdot opt_{TSP-R}(X, d)$, and that our algorithm is 2-approximate.

Let C^* be an optimal solution for (X, d), and consider the set of edges E^* which are used in the cycle, then (X, E^*) is a connected graph; take any spanning tree $T' = (X, E')$ of the graph (X, E^*), then

$$cost_d(T') \leq cost_d(C^*) = opt_{TSP-R}(X, d)$$

because T' uses a subset of the edges of C^*. On the other hand, T' is a spanning tree, and so

$$cost_d(T') \geq opt_{MST}(X, d)$$

So we have

$$opt_{TSP-R}(X, d) \geq opt_{MST}(X, d)$$

from which it follows that our algorithm achieves a 2-approximation.□

3.3 Eulerian Cycles

In this section we review the definition of Eulerian cycle. In the next section, we will use this notion to give a new view of the 2-approximate algorithm of the previous section, and we willnote that this new perspective suggests a potentially better algorithm, that we will analyze in the next lecture.

In this section, it will be convenient to work with *multi-graphs* instead of graphs. In an undirected multi-graph $G = (V, E)$, E is a multi-set of pairs of vertices, that is, the same pair (u, v) can appear more than once in E. Graphically, we can represent a multi-graph in which there are k edges between u and v by drawing k parallel edges between u and v. The degree of a vertex in a multigraph is the number of edges of the graph that have that vertex as an endpoint, counting multiplicities.

Definition 3.6 (Eulerian Cycle) *An Eulerian cycle in a multi-graph $G = (V, E)$ is a cycle $v_0 \to v_1 \to \cdots \to v_m = v_0$ such that the number of edges (u, v) in E is equal to the number of times (u, v) is used in the cycle.*

In a standard graph, a Eulerian cycle is a cycle that uses every edge of the graph exactly once.

Theorem 3.7 *A multi-graph $G = (V, E)$ has an Eulerian cycle if and only if every vertex has even degree and the vertices of positive degree are connected. Furthermore, there is a polynomial time algorithm that, on input a connected graph in which every vertex has even degree, outputs an Eulerian cycle.*

PROOF: If G is Eulerian, then the cycle gives a way to go from every vertex to every other vertex, except the vertices of zero degree. Furthermore, if a vertex v appears k times in the cycle, then there are $2k$ edges involving v in the cycle (because, each time v is reached,

there is an edge used to reach v and one to leave from v); since the cycle contains all the edges of the graph, it follows that v has degree $2k$, thus all vertices have even degree. This shows that if a graph contains an Eulerian cycle, then every vertex has even degree and all the vertices of non-zero degree are connected.

For the other direction, we will prove by induction a slightly stronger statement, that is we will prove that if G is a graph in which every vertex has even degree, then every connected component of G with more than one vertex has a Eulerian cycle. We will proceed by induction on the number of edges.

If there are zero edges, then every connected component has only one vertex and so there is nothing to prove. This is the base case of the induction.

If we have a graph $G = (V, E)$ with a non-empty set of edges and in which every vertex has even degree, then let $V_1, \ldots, V_m$ be the connected components of G that have at least two vertices. If $m \geq 2$, then every connected component has strictly fewer vertices than G, and so we can apply the inductive hypothesis and find Eulerian cycles in each of $V_1, \ldots, V_m$.

It remains to consider the case in which the set V' of vertices of positive degree of G are all in the same connected component. Let $G' = (V', E')$ be the restriction of G to the vertices of V'. Since every vertex of G' has degree ≥ 2, there must be a cycle in G'. This is because if a connected graph with n vertices has no cycles, then it is a tree, and so it has $n - 1$ edges; but in a graph in which there are n vertices and every vertex has degree ≥ 2, the number of edges is at least $\frac{1}{2} \cdot 2 \cdot n = n$. Let C be a simple cycle (that is, a cycle with no vertices repeated) in G', and let G'' be the graph obtained from G' by removing the edges of C. Since we have removed two edges from every vertex, we have that G'' is still a graph in which every vertex has even degree. Since G'' has fewer edges than G' we can apply the induction hypothesis, and find a Eulerian cycle in each non-trivial connected component (a connected component is trivial if it contains only an isolated vertex of degree zero) of G''. We can then patch together these Eulerian cycles with C as follows: we traverse C, starting from any vertex; the first time we reach one of the non-trivial connected components of G'', we stop traversing C, and we traverse the Eulerian cycle of the component, then continue on C, until we reach for the first time one of the non-trivial connected components of G'' that we haven't traversed yet, and so on. This describes a Eulerian path into all of G'

Finally, we note that this inductive argument can be converted into a recursive algorithm. The main computation is to find the connected components of a graph, which can be done in linear time, and to find a cycle in a given graph, which can also be done in linear time using a DFS. Hence the algorithm runs in polynomial time.□

3.4 Eulerian Cycles and TSP Approximation

Let (X, d) be an instance of General TSP-R. Suppose that $G = (X, E)$ is a connected multigraph with vertex set X that admits an Eulerian cycle C. Then the Eulerian cycle C is also an admissible solution for the TSP problem, and its cost is $\sum_{(u,v)\in E} d(u, v)$. Conversely, if we take any cycle which is a TSP-R solution for the input (X, d), and we let E be the multiset of edges used by the cycle (if the cycle uses the same edge more than once, we put

as many copies of the edge in E as the number of times it appears in the cycle), then we obtain a graph $G = (X, E)$ which is connected and which admits an Eulerian cycle.

In other words, we can think of the General TSP-R as the following problem: given a set of points X and a symmetric distance function $d(\cdot, \cdot)$, find the multi-set of edges E such that the graph $G = (V, E)$ is connected and Eulerian, and such that $\sum_{(u,v)\in E} d(u, v)$ is minimized.

The approach that we took in our 2-approximate algorithm was to start from a spanning tree, which is guaranteed to be connected, and then *take every edge of the spanning tree twice*, which guarantees that every vertex has even degree, and hence that an Eulerian cycle exists. The reader should verify that if we take a tree, double all the edges, and then apply to the resulting multigraph the algorithm of Theorem 3.7, we get the same cycle as the one obtained by following the order in which the vertices of the tree are traversed in a DFS, as in the proof of Lemma 3.4.

From this point of view, the 2-approximate algorithm seems rather wasteful: once we have a spanning tree, our goal is to add edges so that we obtain an Eulerian graph in which every vertex has even degree. Doubling every edge certainly works, but it is a rather "brute force" approach: for example if a vertex has degree 11 in the tree, we are going to add another 11 edges incident on that vertex, while we could have "fixed" the degree of that vertex by just adding one more edge. We will see next time that there is a way to implement this intuition and to improve the factor of 2 approximation.

Lecture 4

TSP and Set Cover

In which we describe a 1.5*-approximate algorithm for the Metric TSP, we introduce the Set Cover problem, observe that it can be seen as a more general version of the Vertex Cover problem, and we devise a logarithmic-factor approximation algorithm.*

4.1 Better Approximation of the Traveling Salesman Problem

In the last lecture we discussed equivalent formulations of the Traveling Salesman problem, and noted that Metric TSP-R can also be seen as the following problem: given a set of points X and a symmetric distance function $d : X \times X \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, find a multi-set of edges such that (X, E) is a connected multi-graph in which every vertex has even degree and such that $\sum_{(u,v)\in E} d(u, v)$ is minimized.

Our idea will be to construct E by starting from a minimum-cost spanning tree of X, and then adding edges so that every vertex becomes of even degree.

But how do we choose which edges to add to T?

Definition 4.1 (Perfect Matching) *Recall that graph (V, M) is a matching if no two edges in M have an endpoint in common, that is, if all vertices have degree zero or one. If (V, M) is a matching, we also call the edge set M a matching. A matching is a perfect matching if every vertex has degree one*

Note that a perfect matching can exist only if the number of vertices is even, in which case $|M| = |V|/2$.

Definition 4.2 (Min Cost Perfect Matching) *The Minimum Cost Perfect Matching Problem is defined as follows: an input of the problem is a an even-size set of vertices V and a non-negative symmetric weight function $w : V \times V \to \mathbb{R}_{\geq 0}$; the goal is to find a*

perfect matching (V, M) *such that the cost*

$$cost_w(M) := \sum_{(u,v)\in M} w(u,v)$$

of the matching is minimized.

We state, without proof, the following important result about perfect matchings.

Fact 4.3 *There is a polynomial-time algorithm that solves the Minimum Cost Perfect Matching Problem optimally.*

We will need the following observation.

Fact 4.4 *In every undirected graph, there is an even number of vertices having odd degree.*

PROOF: Let $G = (V, E)$ be any graph. For every vertex $v \in V$, let $deg(v)$ be the degree of v, and let O be the set of vertices whose degree is odd. We begin by noting that the sum of the degrees of all vertices is even, because it counts every edge twice:

$$\sum_{v\in V} deg(v) = 2 \cdot |E|$$

The sum of the degrees of the vertices in $V - O$ is also even, because it is a sum of even numbers. So we have that the sum of the degrees of the vertices in O is even, because it is a difference of two even numbers:

$$\sum_{v\in O} deg(v) = 2 \cdot |E| - \sum_{v\in V-O} deg(v) \equiv 0 \pmod 2$$

Now it follows from arithmetic modulo 2 that if we sum a collection of odd numbers and we obtain an even result, then it must be because we added an even number of terms. (Because the sum of an even number of odd terms is even.) So we have proved that $|O|$ is even. □

We are now ready to describe our improved polynomial-time approximation algorithm for General TSP-R.

- Input: instance (X, d) of Metric TSP-R
- Find a minimum cost spanning tree $T = (X, E)$ of X relative to the weight function $d(\cdot, \cdot)$
- Let O be the set of points that have odd degree in T
- Find a minimum cost perfect matching (O, M) over the points in O relative to the weight function $d(\cdot, \cdot)$

- Let E' be the multiset of edges obtained by taking the edges of E and the edges of M, with repetitions
- Find a Eulerian cycle C in the graph (X, E')
- Output C

We first note that the algorithm is correct, because (X, E') is a connected multigraph (because it contains the connected graph T) and it is such that all vertices have even degree, so it is possible to find an Eulerian cycle, and the Eulerian cycle is a feasible solution to General TSP-R.

The cost of the solution found by the algorithm is

$$\sum_{(u,v)\in E'} d(u,v) = cost_d(E) + cost_d(M)$$

We have already proved that, if $T = (X, E)$ is an optimal spanning tree, then $cost_d(E) \leq opt_{TSP-R}(X, d)$.

Lemma 4.5 below shows that $cost_d(M) \leq \frac{1}{2} opt_{TSP-R}(X, d)$, and so we have that the cost of the solution found by the algorithm is $\leq 1.5 \cdot opt_{TSP-R}(X, d)$, and so we have a polynomial time $\frac{3}{2}$-approximate algorithm for Metric TSP-R. (And also General TSP-R and Metric TSP-NR by the equivalence that we proved in the previous lecture.)

Lemma 4.5 *Let X be a set of points, $d(\cdot,\cdot)$ be a symmetric distance function that satisfies the triangle inequality, and $O \subseteq X$ be an-even size subset of points. Let M^* be a minimum cost perfect matching for O with respect to the weight function $d(\cdot,\cdot)$. Then*

$$cost_d(M^*) \leq \frac{1}{2} opt_{TSP-R}(X, d)$$

PROOF: Let C be a cycle which is an optimal solution for the Metric TSP-R instance (X, d). Consider the cycle C' which is obtained from C by skipping the elements of $X - O$, and also the elements of O which are repeated more than once, so that exactly once occurrence of every element of O is kept in C'. For example, if $X = \{a, b, c, d, e\}$, $O = \{b, c, d, e\}$ and C is the cycle $a \to c \to b \to d \to e \to a \to b \to a$ then we obtain C' by skipping the occurrences of a and the second occurrence of b, and we have the cycle $c \to b \to d \to e \to c$. Because of the triangle inequality, the operation of skipping a point (which means replacing the two edges $u \to v \to w$ with the single edge $u \to w$) can only make the cycle shorter, and so

$$cost_d(C') \leq cost_d(C) = opt_{TSP-R}(X, d)$$

Now, C' is a cycle with an even number of vertices and edges, so we can write $C' = v_1 \to v_2 \to \cdots \to v_{2k} \to v_1$, where $v_1, \ldots, v_{2k}$ is some ordering of the vertices and $k := |O|/2$. We note that we can partition the set of edges in C' into two perfect matchings: the perfect matching $\{(v_1, v_2), (v_3, v_4), \ldots, (v_{2k-1}, v_{2k})\}$ and the perfect matching

$\{(v_2, v_3), (v_4, v_5), \ldots, (v_{2k}, v_1)\}$. Since C' is made of the union of the edges of M_1 and M_2, we have

$$cost_d(C') = cost_d(M_1) + cost_d(M_2)$$

The perfect matching M^* is the minimum-cost perfect matching for O, and so $cost_d(M_1) \geq cost_d(M^*)$ and $cost_d(M_2) \geq cost_d(M^*)$, so we have

$$cost_d(C') \geq 2cost_d(M^*)$$

and hence

$$opt_{TSP-R}(X, d) \geq cost_d(C') \geq 2 \cdot cost_d(M^*)$$

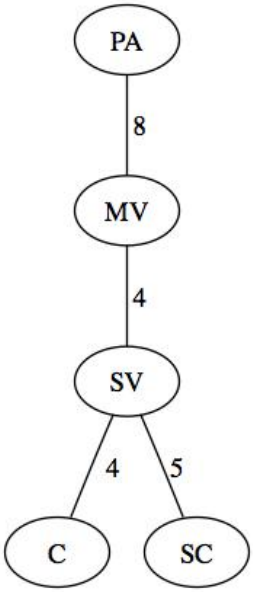
□

An important point is that the algorithm that we just analyzed, like every other approximation algorithm, is always able to provide, together with a feasible solution, a *certificate* that the optimum is greater than or equal to a certain lower bound. In the 2-approximate algorithm TSP algorithm from the previous lecture, the certificate is a minimum spanning tree, and we have that the TSP optimum is at least the cost of the minimum spanning tree. In the improved algorithm of today, the cost of minimum spanning tree gives a lower bound, and twice the cost of the minimum cost perfect matching over O gives another lower bound, and we can take the largest of the two.

Let us work out an example of the algorithm on a concrete instance, and see what kind of solution and what kind of lower bound we derive. Our set of points will be: Cupertino, Mountain View, Palo Alto, Santa Clara, and Sunnyvale. We have the following distances in miles, according to Google map:

	C	MV	PA	SC	SV
C	0	7	12	7	4
MV		0	8	9	4
PA			0	14	10
SC				0	5
SV					0

The reader can verify that the triangle inequality is satisfied. If we run a minimum spanning tree algorithm, we find the following tree of cost 21



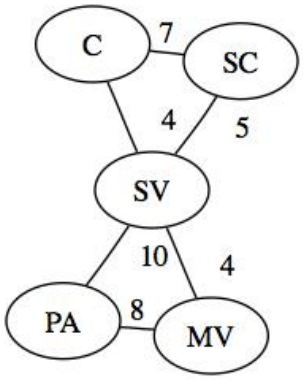
This tells us that the optimum is at least 21 miles.

If we employ the algorithm from the last lecture, we perform a DFS which gives us the cycle Palo Alto → Mountain View → Sunnyvale → Cupertino → Sunnyvale → Santa Clara → Sunnyvale → Mountain View → Palo Alto, which has a length of 42 miles. After skipping the places that have already been visited, we get the cycle Palo Alto → Mountain View → Sunnyvale → Cupertino → Santa Clara → Palo Alto, whose length is 37 miles.

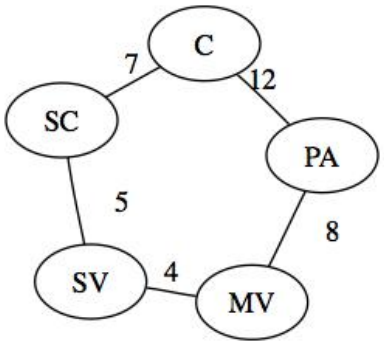
Today's algorithm, instead, looks for a minimum cost perfect matching of the points that have odd degree in the spanning tree, that is all the places except Mountain View. A minimum cost perfect matching (there are two optimal solutions) is $\{(PA, SV), (C, SC)\}$ whose cost is 17 miles, 10 for the connection between Palo Alto and Sunnyvale, and 7 for the one between Cupertino and Santa Clara.

This tells us that the TSP optimum must be at least 34, a stronger lower bound than the one coming from the minimum spanning tree.

When we add the edges of the perfect matching to the edges of the spanning tree we get the following graph, which is connected and is such that every vertex has even degree:



We can find an Eulerian cycle in the graph, and we find the cycle Palo Alto → Mountain View → Sunnyvale → Santa Clara → Cupertino → Sunnyvale → Palo Alto, whose length is 38 miles. After skipping Sunnyvale the second time, we have the cycle Palo Alto → Mountain View → Sunnyvale → Santa Clara → Cupertino → Palo Alto whose length is 36 miles.



In summary, yesterday's algorithm finds a solution of 37 miles, and a certificate that the optimum is at least 21. Today's algorithm finds a solution of 36 miles, and a certificate that the optimum is at least 34.

4.2 The Set Cover Problem

Definition 4.6 *The Minimum Set Cover problem is defined as follows: an input of the problem is a finite set X and a collection of subsets $S_1, \ldots, S_m$, where $S_i \subseteq X$ and $\bigcup_{i=1}^{m} S_i = X$.*

The goal of the problem is to find a smallest subcollection of sets whose union is X, that is we want to find $I \subseteq \{1, \ldots, m\}$ such that $\cup_{i \in I} S_i = X$ and $|I|$ is minimized.

For example, suppose that we want to assemble a team to work on a project, and each of the person that we can choose to be on the team has a certain set of skills; we want to find

the smallest group of people that, among themselves, have all the skills that we need. Say, concretely, that we want to form a team of programmers and that we want to make sure that, among the team members, there are programmers who can code in C, C++, Ruby, Python, and Java. The available people are Andrea, who knows C and C++, Ben, who knows C++ and Java, Lisa, who knows C++, Ruby and Python, and Mark who knows C and Java. Selecting the smallest team is the same as a Minimum Set Cove problem in which we have the instance

$$X = \{\mathrm{C}, \mathrm{C}++, \mathrm{Ruby}, \mathrm{Python}, \mathrm{Java}\}$$
$$S_1 = \{\mathrm{C}, \mathrm{C}++\}, S_2 = \{\mathrm{C}++, \mathrm{Java}\},$$
$$S_3 = \{\mathrm{C}++, \mathrm{Ruby}, \mathrm{Python}\}, S_4 = \{\mathrm{C}, \mathrm{Java}\}$$

In which the optimal solution is to pick S_3, S_4, that is Lisa and Mark.

Although this is an easy problem on very small instances, it is an NP-hard problem and so it is unlikely to be solvable exactly in polynomial time. In fact, there are bad news also about approximation.

Theorem 4.7 *Suppose that, for some constant $\epsilon > 0$, there is an algorithm that, on input an instance of Set Cover finds a solution whose cost is at most $(1 - \epsilon) \cdot \ln |X|$ times the optimum; then every problem in* **NP** *admits a randomized algorithm running in time $n^{O(\log \log n)}$, where n is the size of the input.*

If, for some constant c, there is a polynomial time c-approximate algorithm, then $\mathbf{P} = \mathbf{NP}$.

The possibility of nearly-polynomial time randomized algorithms is about as unlikely as $\mathbf{P} = \mathbf{NP}$, so the best that we can hope for is an algorithm providing a $\ln |X|$ factor approximation.

A simple greedy approximation provides such an approximation.

Consider the following greedy approach to finding a set cover:

- Input: A set X and a collection of sets $S_1, \ldots, S_m$
- $I := \emptyset$
- while there is an *uncovered* element, that is an $x \in X$ such that $\forall i \in I . x \notin S_i$
  - Let S_i be a set with the largest number of uncovered elements
  - $I := I \cup \{i\}$
- return I

To work out an example, suppose that our input is

$$X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}\}$$

$$S_1 = \{x_1, x_2, x_7, x_8\}$$
$$S_2 = \{x_1, x_3, x_4, x_8, x_{10}\}$$
$$S_3 = \{x_6, x_3, x_9, x_{10}\}$$
$$S_4 = \{x_1, x_5, x_7, x_8\}$$
$$S_5 = \{x_2, x_3, x_4, x_8, x_9\}$$

The algorithm will pick four sets as follows:

- At the first step, all the elements of X are uncovered, and the algorithm picks S_2, which is the set that covers the most elements (five);
- At the second step, there are five remaining uncovered elements, and the best that we can do is to cover two of them, for example picking S_1;
- At the third step there remain three uncovered elements, and again the best we can do is to cover two of them, by picking S_3;
- At the fourth step only x_5 remains uncovered, and we can cover it by picking S_4.

As with the other algorithms that we have analyzed, it is important to find ways to prove lower bounds to the optimum. Here we can make the following easy observations: at the beginning, we have 10 items to cover, and no set can cover more than 5 of them, so it is clear that we need at least two sets. At the second step, we see that there are five uncovered items, and that there is no set in our input that contains more than two of those uncovered items; this means that even the optimum solution must use at least $5/2$ sets to cover those five items, and so at least $5/2$ sets, that is at least 3 sets, to cover all the items.

In general, if we see that at some point there are k items left to cover, and that every set in our input contains at most t of those items, it follows that the optimum contains at least k/t sets. These simple observations are already sufficient to prove that the algorithm is $(\ln |X| + O(1))$-approximate.

We reason as follows. Let $X, S_1, \ldots, S_m$ be the input to the algorithm, and let $x_1, \ldots, x_n$ be an ordering of the elements of X in the order in which they are covered by the algorithm. Let c_i be the number of elements that become covered at the same time step in which x_i is covered. Let *opt* be the number of sets used by an optimal solution and *apx* be the number of sets used by the algorithm.

For every i, define

$$cost(x_i) := \frac{1}{c_i}$$

The intuition for this definition is that, at the step in which we covered x_i, we had to "pay" for one set in order to cover c_i elements that were previously uncovered. Thus, we can think of each element that we covered at that step as having cost us $\frac{1}{c_i}$ times the cost of a set. In particular, we have that the total number of sets used by the algorithm is the sum of the costs:

$$apx = \sum_{i=1}^{n} cost(x_i)$$

Now, consider the items $x_i, \ldots, x_n$ and let us reason about how the optimum solution manages to cover them. Every set in our input covers at most c_i of those $n - i + 1$ items, and it is possible, using the optimal solution, to cover all the items, including the items $x_i, \ldots, x_n$ with *opt* sets. So it must be the case that

$$opt \geq \frac{n - i + 1}{c_i} = (n - i + 1) \cdot cost(x_i)$$

from which we get

$$apx \leq opt \cdot \left(\sum_{i=1}^{n} \frac{1}{n - i + 1} \right)$$

The quantity

$$\sum_{i=1}^{n} \frac{1}{n - i + 1} = \sum_{i=1}^{n} \frac{1}{i}$$

is known to be at most $\ln n + O(1)$, and so we have

$$apx \leq (\ln n + O(1)) \cdot opt$$

It is easy to prove the weaker bound $\sum_{i=1}^{n} \frac{1}{n} \leq \lceil \log_2 n + 1 \rceil$, which suffices to prove that our algorithm is $O(\log n)$-approximate: just divide the sum into terms of the form $\sum_{i=2^k}^{2^{k+1}-1} \frac{1}{i}$, that is

$$1 + \left(\frac{1}{2} + \frac{1}{3} \right) + \left(\frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} \right) + \cdots$$

and notice that each term is at most 1 (because each term is itself the sum of 2^k terms, each $\leq 2^{-k}$) and that the whole sum contains at most $\lceil \log_2 n + 1 \rceil$ such terms.

4.3 Set Cover versus Vertex Cover

The Vertex Cover problem can be seen as the special case of Set Cover in which every item in X appears in precisely two sets.

If $G = (V, E)$ is an instance of Vertex Cover, construct the instance of Set Cover in which $X = E$, and in which we have one set S_v for every vertex v, defined so that S_v is the set of

all edges that have v as an endpoint. Then finding a subcollection of sets that covers all of X is precisely the same problem as finding a subset of vertices that cover all the edges.

The greedy algorithm for Set Cover that we have discussed, when applied to the instances obtained from Vertex Cover via the above transformation, is precisely the greedy algorithm for Vertex Cover: the algorithm starts from an empty set of vertices, and then, while there are uncovered edges, adds the vertex incident to the largest number of uncovered edges. By the above analysis, the greedy algorithm for Vertex Cover finds a solution that is no worse than $(\ln n + O(1))$ times the optimum, a fact that we mentioned without proof a couple of lectures ago.

Lecture 5

Linear Programming

In which we introduce linear programming.

5.1 Introduction to Linear Programming

A *linear program* is an optimization problem in which we have a collection of variables, which can take real values, and we want to find an assignment of values to the variables that satisfies a given collection of linear inequalities and that maximizes or minimizes a given linear function.

(The term *programming* in *linear programming*, is not used as in *computer programming*, but as in, e.g., *tv programming*, to mean *planning*.)

For example, the following is a linear program.

$$\begin{array}{ll} \text{maximize} & x_1 + x_2 \\ \text{subject to} & \\ & x_1 + 2x_2 \leq 1 \\ & 2x_1 + x_2 \leq 1 \\ & x_1 \geq 0 \\ & x_2 \geq 0 \end{array} \tag{5.1}$$

The linear function that we want to optimize ($x_1 + x_2$ in the above example) is called the *objective function*. A *feasible solution* is an assignment of values to the variables that satisfies the inequalities. The value that the objective function gives to an assignment is called the *cost* of the assignment. For example, $x_1 := \frac{1}{3}$ and $x_2 := \frac{1}{3}$ is a feasible solution, of cost $\frac{2}{3}$. Note that if x_1, x_2 are values that satisfy the inequalities, then, by summing the first two inequalities, we see that

$$3x_1 + 3x_2 \leq 2$$

that is,

$$x_1 + x_2 \leq \frac{2}{3}$$

and so no feasible solution has cost higher than $\frac{2}{3}$, so the solution $x_1 := \frac{1}{3}$, $x_2 := \frac{1}{3}$ is optimal. As we will see in the next lecture, this trick of summing inequalities to verify the optimality of a solution is part of the very general theory of *duality* of linear programming.

Linear programming is a rather different optimization problem from the ones we have studied so far. Optimization problems such as Vertex Cover, Set Cover, Steiner Tree and TSP are such that, for a given input, there is only a finite number of possible solutions, so it is always trivial to solve the problem in finite time. The number of solutions, however, is typically exponentially big in the size of the input and so, in order to be able to solve the problem on reasonably large inputs, we look for polynomial-time algorithms. In linear programming, however, each variable can take an infinite number of possible values, so it is not even clear that the problem is solvable in finite time.

As we will see, it is indeed possible to solve linear programming problems in finite time, and there are in fact, polynomial time algorithms and efficient algorithms that solve linear programs optimally.

There are at least two reasons why we are going to study linear programming in a course devoted to combinatorial optimization:

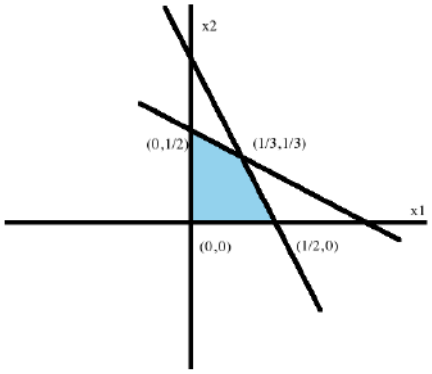
- Efficient linear programming solvers are often used as part of the toolkit to design exact or approximate algorithms for combinatorial problems.
- The powerful theory of *duality* of linear programming, that we will describe in the next lecture, is a very useful mathematical theory to reason about algorithms, including purely combinatorial algorithms for combinatorial problems that seemingly have no connection with continuous optimization.

5.2 A Geometric Interpretation

5.2.1 A 2-Dimensional Example

Consider again the linear program (5.1). Since it has two variables, we can think of any possible assignment of values to the variables as a point (x_1, x_2) in the plane. With this interpretation, every inequality, for example $x_1 + 2x_2 \leq 1$, divides the plane into two regions: the set of points (x_1, x_2) such that $x_1 + 2x_2 > 1$, which are definitely not feasible solutions, and the points such that $x_1 + 2x_2 \leq 1$, which satisfy the inequality and which could be feasible provided that they also satisfy the other inequalities. The line with equation $x_1 + 2x_2 = 1$ is the boundary between the two regions.

The set of feasible solutions to (5.1) is the set of points which satisfy all four inequalities, shown in blue below:

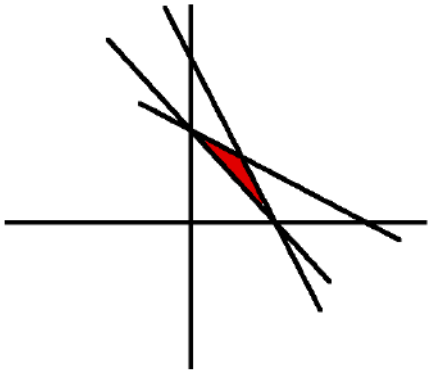


The feasible region is a polygon with four edges, one for each inequality. This is not entirely a coincidence: for each inequality, for example $x_1 + 2x_2 \leq 1$, we can look at the line which is the boundary between the region of points that satisfy the inequality and the region of points that do not, that is, the line $x_1 + 2x_1 = 1$ in this example. The points on the line that satisfy the other constraints form a segment (in the example, the segment of the line $x_1 + 2x_2 = 1$ such that $0 \leq x_1 \leq 1/3$), and that segment is one of the edges of the polygon of feasible solutions. Although it does not happen in our example, it could also be that if we take one of the inequalities, consider the line which is the boundary of the set of points that satisfy the inequality, and look at which points on the line are feasible for the linear program, we end up with the empty set (for example, suppose that in the above example we also had the inequality $x_1 + x_2 \geq -1$); in this case the inequality does not give rise to an edge of the polygon of feasible solutions. Another possibility is that the line intersects the feasible region only at one point (for example suppose we also had the inequality $x_1 + x_2 \geq 0$). Yet another possibility is that our polygon is unbounded, in which case one of its edges is not a segment but a half-line (for example, suppose we did not have the inequality $x_1 \geq 0$, then the half-line of points such that $x_2 = 0$ and $x_1 \leq 1$ would have been an edge).

To look for the best feasible solution, we can start from an arbitrary point, for example the vertex $(0, 0)$. We can then divide the plane into two regions: the set of points whose cost is greater than or equal to the cost of $(0, 0)$, that is the set of points such that $x_1 + x_2 \geq 0$, and the set of points of cost lower than the cost of $(0, 0)$, that is, the set of points such that $x_1 + x_2 < 0$. Clearly we only need to continue our search in the first region, although we see that actually the entire set of feasible points is in the first region, so the point $(0, 0)$ is actually the *worst* solution.

So we continue our search by trying another vertex, for example $(1/2, 0)$. Again we can divide the plane into the set of points of cost greater than or equal to the cost of $(1/2, 0)$, that is the points such that $x_1 + x_2 \geq 1/2$, and the set of points of cost lower than the cost of $(1, 0)$. We again want to restrict ourselves to the first region, and we see that we have

now narrowed down our search quite a bit: the feasible points in the first region are shown in red:



So we try another vertex in the red region, for example $\left(\frac{1}{3}, \frac{1}{3}\right)$, which has cost $\frac{2}{3}$. If we consider the region of points of cost greater than or equal to the cost of the point $(1/3, 1/3)$, that is, the region $x_1 + x_2 \geq 2/3$, we see that the point $(1/3, 1/3)$ is the only feasible point in the region, and so there is no other feasible point of higher cost, and we have found our optimal solution.

5.2.2 A 3-Dimensional Example

Consider now a linear program with three variables, for example

$$\begin{array}{ll} \text{maximize} & x_1 + 2x_2 - x_3 \\ \text{subject to} & \\ & x_1 + x_2 \leq 1 \\ & x_2 + x_3 \leq 1 \\ & x_1 \geq 0 \\ & x_2 \geq 0 \\ & x_3 \geq 0 \end{array} \tag{5.2}$$

In this case we can think of every assignment to the variables as a point (x_1, x_2, x_3) in three-dimensional space. Each constraint divides the space into two regions as before; for example the constraint $x_1 + x_2 \leq 1$ divides the space into the region of points (x_1, x_2, x_3) such that $x_1 + x_2 \leq 1$, which satisfy the equation, and points such that $x_1 + x_2 > 1$, which do not. The boundary between the regions is the *plane* of points such that $x_1 + x_2 = 1$. The region of points that satisfy an inequality is called a *half-space*.

The set of feasible points is a *polyhedron* (plural: polyhedra). A polyhedron is bounded by *faces*, which are themselves polygons. For example, a cube has six faces, and each face is

a square. In general, if we take an inequality, for example $x_3 \geq 0$, and consider the plane $x_3 = 0$ which is the boundary of the half-space of points that satisfy the inequality, and we consider the set of feasible points in that plane, the resulting polygon (if it's not the empty set) is one of the faces of our polyhedron. For example, the set of feasible points in the place $x_3 = 0$ is the triangle given by the inequalities $x_1 \geq 0$, $x_2 \geq 0$, $x_1 + x_2 \leq 1$, with vertices $(0, 0, 0)$, $(0, 1, 0)$ and $(1, 0, 0)$. So we see that a 2-dimensional *face* is obtained by taking our inequalities and changing one of them to equality, provided that the resulting feasible region is two-dimensional; a 1-dimensional *edge* is obtained by changing two inequalities to equality, again provided that the resulting constraints define a 1-dimensional region; and a vertex is obtained by changing three inequalities to equality, provided that the resulting point is feasible for the other inequalities.

As before, we can start from a feasible point, for example the vertex $(0, 0, 0)$, of cost zero, obtained by changing the last three inequalities to equality. We need to check if there are feasible points, other than $(0, 0, 0)$, such that $x_1 + 2x_2 - x_3 \geq 0$. That is, we are interested in the set of points such that

$$\begin{aligned} x_1 + x_2 &\leq 1 \\ x_2 + x_3 &\leq 1 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \\ x_3 &\geq 0 \\ x_1 + 2x_2 - x_3 &\geq 0 \end{aligned}$$

which is again a polyhedron, of which $(0, 0, 0)$ is a vertex. To find another vertex, if any, we try to start from the three inequalities that we changed to equality to find $(0, 0, 0)$, and remove one to see if we get an edge of non-zero length or just the point $(0, 0, 0)$ again.

For example, if we keep $x_1 = 0$, $x_2 = 0$, we see that only feasible value of x_3 is zero, so there is no edge; if we keep $x_1 = 0, x_3 = 0$, we have that the region $0 \leq x_2 \leq 1$ is feasible, and so it is an edge of the above polyhedron. The other vertex of that edge is $(0, 1, 0)$, which is the next solution that we shall consider. It is a solution of cost 2, so, in order to look for a better solution, we want to consider the polyhedron

$$\begin{aligned} x_1 + x_2 &\leq 1 \\ x_2 + x_3 &\leq 1 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \\ x_3 &\geq 0 \\ x_1 + 2x_2 - x_3 &\geq 2 \end{aligned}$$

In order to see if this polyhedron has any edge of non-zero length, we again keep only two of the three equations that defined our vertex $(0, 1, 0)$, that is only two of the equations $x_1 = 0$, $x_3 = 0$, $x_1 + x_2 = 1$. If we keep the first two, $(0, 1, 0)$ is the only feasible point. If we keep $x_3 = 0$ and $x_1 + x_2 = 1$, then $(0, 1, 0)$ is again the only feasible point. If we keep $x_1 = 0$ and $x + 1 + x_2 = 1$, that is $x_1 = 0$ and $x_2 = 1$, we see again that the only feasible point is $(0, 1, 0)$. These are the only three edges that could have had $(0, 1, 0)$ as an endpoint,

and since $(0, 1, 0)$ is a vertex of the above polytope, we have to conclude that the polytope has no edge, and so it is made of the single point $(0, 1, 0)$.

This means that $(0, 1, 0)$ is the optimal solution of (5.2)

5.2.3 The General Case

In general, if we have a linear program with n variables $x_1, \ldots, x_n$, we can think of every assignment to the variables as an n-dimensional point $(x_1, \ldots, x_n)$ in $\mathbb{R}^n$.

Every inequality $a_1x_1 + \ldots a_nx_n \leq b$ divides the space $\mathbb{R}^n$ into the region that satisfies the inequality and the region that does not satisfy the inequality, with the *hyperplane* $a_1x_1 + \cdots a_nx_n = b$ being the boundary between the two regions. The two regions are called *half-spaces*.

The set of feasible points is an intersection of half-spaces and is called a *polytope*.

Generalizing the approach that we have used in the previous two examples, the following is the outline of an algorithm to find an optimal solution to a given maximization linear program:

1. Start from a vertex $(a_1, \ldots, a_n)$ in the feasible region, by changing n of the inequalities to equality in such a way that: (i) the resulting n equations are linearly independent, and (ii) the unique solution is feasible for the remaining inequalities;

   - If there is no such vertex, output "linear program is infeasible."

2. Consider the n possible edges of the polytope of feasible solutions that have $(a_1, \ldots, a_n)$ as an endpoint. That is, for each of the n equations that identified $(a_1, \ldots, a_n)$, set back that equation to an inequality, and consider the set of solutions that are feasible for the other $n - 1$ equations and for the inequalities (this set of points can be a line, a half-line, a segment, or just the point $(a_1, \ldots, a_n)$).

   - If there is an edge that contains points of arbitrarily large cost, then output "optimum is unbounded"
   - Else, if there are edges that contain points of cost larger than $(a_1, \ldots, a_n)$, then let $(b_1, \ldots, b_n)$ be the second endpoint of one of such edges
     - $(a_1, \ldots, a_n) := (b_1, \ldots, b_n)$;
     - go to 2
   - Else, output "$(a_1, \ldots, a_n)$ is an optimal solution"

This is the outline of an algorithm called the *Simplex algorithm*. It is not a complete description because:

- We haven't discussed how to find the initial vertex. This is done by constructing a new polytope such that finding an optimal solution in the new polytope either gives

us a feasible vertex for the original linear program, or a "certificate" that the original problem is infeasible. We then apply the Simplex algorithm to the new polytope. Now, this looks like the beginning of an infinite loop, but the new polytope is constructed in such a way that it is easy to find an initial feasible vertex.

- We haven't discussed certain special cases; for example it is possible for a polytope to not have any vertex, for example if the number of inequalities is less than the number of variables. (In such a case we can either add inequalities or eliminate variables in a way that do not change the optimum but that creates vertices.) Also there can be cases in which a vertex of a polytope in $\mathbb{R}^n$ is the endpoint of more than n edges (consider a pyramid with a square base in $\mathbb{R}^3$: the top vertex has is an endpoint of four edges), while the algorithm, as described above, considers at most n edges for every vertex.

The simplex algorithm shows that a linear program can always be solved in finite time, and in fact in time that is at most exponential in the number of variables. This is because each iteration takes polynomial time and moves to a new vertex, and if there are m inequalities and n variables there can be at most $\binom{m}{n} \leq m^n$ vertices.

Unfortunately, for all the known variants of the simplex method (which differ in the way they choose the vertex to move to, when there is more than one possible choice) there are examples of linear programs on which the algorithm takes exponential time. In practice, however, the simplex algorithm is usually very fast, even on linear programs with tens or hundreds of thousands of variables and constraints.

5.2.4 Polynomial Time Algorithms for LInear Programming

Two (families of) polynomial time algorithms for linear programming are known.

One, called the ellipsoid algorithm, starts by finding an ellipsoid (the high-dimensional analog of an ellipse, a "squashed disk") that contains the optimal solution, and then, at every step, it constructs another ellipsoid whose volume is a smaller than the previous one, while still being guaranteed to contain the optimal solution. After several iterations, the algorithm identifies a tiny region that contains the optimal solution. It is known that if a linear program has a finite optimum, the values of the x_i in the optimal solution are rational numbers in which both the denominator and numerator have a polynomial number of digits in the size of the input (assuming all coefficients in the objective function and in the inequalities are also rational numbers and that we count the number of digits in their fractional representation when we compute the size of the input), and so if the final ellipsoid is small enough there is only one point with such rational coordinates in the ellipsoid.

The other algorithm, which is actually a family of algorithms, uses the *interior point* method, in which the algorithm computes a sequence of points in the interior of the polytope (in contrast to the simplex algorithm, which finds a sequence of vertices on the exterior of the polytope), where each point is obtained from the previous one by optimizing a properly chosen function that favors points of higher cost for the objective function, and disfavors points that are too close to the boundary of the polytope. Eventually, the algorithm finds

a point that is arbitrary close to the optimal vertex, and the actual optimal vertex can be found, like in the ellipsoid method, once it is the unique point with bounded rational coordinates that is close to the current point.

5.2.5 Summary

Here are the important points to remember about what we discussed so far:

- In a linear program we are given a set of variables $x_1, \ldots, x_n$ and we want to find an assignment to the variables that satisfies a given set of linear equalities, and that maximizes or minimizes a given linear *objective function* of the variables. An assignment that satisfies the inequalities is called a *feasible solution*;

- If we think of every possible assignment to the variables as a point in $\mathbb{R}^n$, then the set of feasible solutions forms a *polytope*;

- It is possible that there is no feasible solution to the given inequalities, in which case we call the linear program *infeasible*.

- If the linear program is feasible, it is possible that it is of maximization type and there are solutions of arbitrarily large cost, or that it is of minimization type and there are solutions of arbitrarily small cost. In this case we say that the linear program is *unbounded*. Sometimes we will say that the optimum of an unbounded maximization linear program is $+\infty$, and that the optimum of an unbounded minimization linear program is $-\infty$, even though this is not entirely correct because there is no feasible solution of cost $+\infty$ or $-\infty$, but rather a sequence of solutions such the limit of their cost is $+\infty$ or $-\infty$.

- If the linear program is feasible and not unbounded then it has a finite optimum, and we are interested in finding a feasible solution of optimum cost.

- The simplex algorithm, the ellipsoid algorithm, and the interior point algorithms are able, given a linear program, to determine if it is feasible or not, if feasible they can determine if it is bounded or unbounded, and if feasible and bounded they can find a solution of optimum cost. All three run in finite time; in the worst case, the simplex algorithm runs in exponential time, while the other algorithms run in time polynomial in the size of the input.

- When we refer to the "size of the input" we assume that all coefficients are rational numbers, and the size of the input is the total number of bits necessary to represent the coefficients of the objective function and of the inequalities as ratios of integers. For example, the rational number a/b requires $\log_2 a + \log_2 b + O(1)$ bits to be represented, if a and b have no common factor.

5.3 Standard Form for Linear Programs

We say that a maximization linear program with n variables is in *standard form* if for every variable x_i we have the inequality $x_i \geq 0$ and all other m inequalities are of $\leq$ type. A linear program in standard form can be written as

$$\begin{array}{ll} \text{maximize} & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \\ & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array} \tag{5.3}$$

Let us unpack the above notation.

The vector $\mathbf{c} = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \in \mathbb{R}^n$ is the column vector of coefficients of the objective function, $\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$ is the column vector of variables, $\mathbf{c}^T = (c_1, \ldots, c_n)$ is the transpose of $\mathbf{c}$, a row vector, and $\mathbf{c}^T\mathbf{x}$ is the matrix product of the $1 \times n$ "matrix" $\mathbf{c}^T$ times the $n \times 1$ "matrix" $\mathbf{x}$, which is the value

$$c_1x_1 + \cdots c_nx_n$$

that is, the objective function.

The matrix A is the $n \times m$ matrix of coefficients of the left-hand sides of the inequalities, and b is the m-dimensional vector of right-hand sides of the inequalities. When we write $\mathbf{a} \leq \mathbf{b}$, for two vectors $\mathbf{a} = \begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix}$ and $\mathbf{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$ we mean the m inequalities $a_1 \leq b_1$, $\ldots$, $a_m \leq b_m$, so the inequality $A\mathbf{x} \leq \mathbf{b}$ means the collection of inequalities

$$a_{1,1}x_1 + \cdots + a_{1,n}x_n \leq b_1$$

$$\cdots$$

$$a_{m,1}x_1 + \cdots + a_{m,n}x_n \leq b_m$$

Putting a linear program in standard form is a useful first step for linear programming algorithms, and it is also useful to develop the theory of *duality* as we will do in the next lecture.

It is easy to see that given an arbitrary linear program we can find an equivalent linear program in standard form.

- If we have a linear program in which a variable x is not required to be ≥ 0, we can introduce two new variables x' and x'', apply the substitution $x \leftarrow x' - x''$ in every inequality in which x appears, and add the inequalities $x' \geq 0$, $x'' \geq 0$. Any feasible solution for the new linear program is feasible for the original one, after assigning $x := x' - x''$, with the same cost; also, any solution that was feasible for the original program can be converted into a solution for the new linear program, with the same cost, by assigning $x' := x$, $x'' := 0$ if $x > 0$ and $x' := 0$, $x'' = -x$ if $x \leq 0$.
- If we have an inequality of $\geq$ type, other than the inequalities $x_i \geq 0$, we can change sign to the left-hand side and the right-hand side, and change the direction of the inequality.
- Some definitions of linear programming allow equations as constraints. If we have an equation $\mathbf{a}^T\mathbf{x} = b$, we rewrite it as the two inequalities $\mathbf{a}^T\mathbf{x} \leq b$ and $-\mathbf{a}^T\mathbf{x} \leq -b$.

The standard form for a minimization problem is

$$\begin{array}{ll} \text{minimize} & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \\ & A\mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array} \tag{5.4}$$

As we did for maximization problems, every minimization problem can be put into normal form by changing the sign of inequalities and doing the substitution $x \rightarrow x' - x''$ for every variable x that does not have a non-negativity constraint $x \geq 0$.

Lecture 6

Linear Programming Duality

In which we introduce the theory of duality in linear programming.

6.1 The Dual of a Linear Program

Suppose that we have the following linear program in maximization standard form:

$$\begin{array}{ll} \text{maximize} & x_1 + 2x_2 + x_3 + x_4 \\ \text{subject to} & \\ & x_1 + 2x_2 + x_3 \leq 2 \\ & x_2 + x_4 \leq 1 \\ & x_1 + 2x_3 \leq 1 \\ & x_1 \geq 0 \\ & x_2 \geq 0 \\ & x_3 \geq 0 \end{array} \tag{6.1}$$

and that an LP-solver has found for us the solution $x_1 := 1$, $x_2 := \frac{1}{2}$, $x_3 := 0$, $x_4 := \frac{1}{2}$ of cost 2.5. How can we convince ourselves, or another user, that the solution is indeed optimal, without having to trace the steps of the computation of the algorithm?

Observe that if we have two valid inequalities

$$a \leq b \text{ and } c \leq d$$

then we can deduce that the inequality

$$a + c \leq b + d$$

(derived by "summing the left hand sides and the right hand sides" of our original inequalities) is also true. In fact, we can also scale the inequalities by a positive multiplicative factor before adding them up, so for every non-negative values $y_1, y_2 \geq 0$ we also have

$$y_1 a + y_2 c \leq y_1 b + y_2 d$$

Going back to our linear program (6.1), we see that if we scale the first inequality by $\frac{1}{2}$, add the second inequality, and then add the third inequality scaled by $\frac{1}{2}$, we get that, for every (x_1, x_2, x_3, x_4) that is feasible for (6.1),

$$x_1 + 2x_2 + 1.5x_3 + x_4 \leq 2.5$$

And so, for every feasible (x_1, x_2, x_3, x_4), its cost is

$$x_1 + 2x_2 + x_3 + x_4 \leq x_1 + 2x_2 + 1.5x_3 + x_4 \leq 2.5$$

meaning that a solution of cost 2.5 is indeed optimal.

In general, how do we find a good choice of scaling factors for the inequalities, and what kind of upper bounds can we prove to the optimum?

Suppose that we have a maximization linear program in standard form.

$$\begin{array}{ll} \text{maximize} & c_1 x_1 + \ldots c_n x_n \\ \text{subject to} & \\ & a_{1,1} x_1 + \ldots + a_{1,n} x_n \leq b_1 \\ & \vdots \\ & a_{m,1} x_1 + \ldots + a_{m,n} x_n \leq b_m \\ & x_1 \geq 0 \\ & \vdots \\ & x_n \geq 0 \end{array} \tag{6.2}$$

For every choice of non-negative scaling factors $y_1, \ldots, y_m$, we can derive the inequality

$$\begin{gathered} y_1 \cdot (a_{1,1} x_1 + \ldots + a_{1,n} x_n) \\ + \cdots \\ + y_n \cdot (a_{m,1} x_1 + \ldots + a_{m,n} x_n) \\ \leq y_1 b_1 + \cdots y_m b_m \end{gathered}$$

which is true for every feasible solution $(x_1, \ldots, x_n)$ to the linear program (6.2). We can rewrite the inequality as

$$\begin{gathered} (a_{1,1} y_1 + \cdots a_{m,1} y_m) \cdot x_1 \\ + \cdots \\ + (a_{1,n} y_1 \cdots a_{m,n} y_m) \cdot x_n \end{gathered}$$

$$\leq y_1 b_1 + \cdots y_m b_m$$

So we get that a certain linear function of the x_i is always at most a certain value, for every feasible $(x_1, \ldots, x_n)$. The trick is now to choose the y_i so that the linear function of the x_i for which we get an upper bound is, in turn, an upper bound to the cost function of $(x_1, \ldots, x_n)$. We can achieve this if we choose the y_i such that

$$\begin{aligned} c_1 &\leq a_{1,1} y_1 + \cdots a_{m,1} y_m \\ &\vdots \\ c_n &\leq a_{1,n} y_1 \cdots a_{m,n} y_m \end{aligned} \tag{6.3}$$

Now we see that for every non-negative $(y_1, \ldots, y_m)$ that satisfies (6.3), and for every $(x_1, \ldots, x_n)$ that is feasible for (6.2),

$$\begin{aligned} & c_1 x_1 + \ldots c_n x_n \\ \leq\ & (a_{1,1} y_1 + \cdots a_{m,1} y_m) \cdot x_1 \\ & + \cdots \\ & + (a_{1,n} y_1 \cdots a_{m,n} y_m) \cdot x_n \\ \leq\ & y_1 b_1 + \cdots y_m b_m \end{aligned}$$

Clearly, we want to find the non-negative values $y_1, \ldots, y_m$ such that the above upper bound is as strong as possible, that is we want to

$$\begin{array}{ll} \text{minimize} & b_1 y_1 + \cdots b_m y_m \\ \text{subject to} & \\ & a_{1,1} y_1 + \ldots + a_{m,1} y_m \geq c_1 \\ & \vdots \\ & a_{n,1} y_1 + \ldots + a_{m,n} y_m \geq c_n \\ & y_1 \geq 0 \\ & \vdots \\ & y_m \geq 0 \end{array} \tag{6.4}$$

So we find out that if we want to find the scaling factors that give us the best possible upper bound to the optimum of a linear program in standard maximization form, we end up with a new linear program, in standard minimization form.

Definition 6.1 *If*

$$\begin{array}{ll} \text{maximize} & \mathbf{c}^T \mathbf{x} \\ \text{subject to} & \\ & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array} \tag{6.5}$$

is a linear program in maximization standard form, then its dual is the minimization linear program

$$\begin{array}{ll} \text{minimize} & \mathbf{b}^T\mathbf{y} \\ \text{subject to} & \\ & A^T\mathbf{y} \geq \mathbf{c} \\ & \mathbf{y} \geq \mathbf{0} \end{array} \tag{6.6}$$

So if we have a linear program in maximization linear form, which we are going to call the *primal* linear program, its dual is formed by having one variable for each constraint of the primal (not counting the non-negativity constraints of the primal variables), and having one constraint for each variable of the primal (plus the non-negative constraints of the dual variables); we change maximization to minimization, we switch the roles of the coefficients of the objective function and of the right-hand sides of the inequalities, and we take the transpose of the matrix of coefficients of the left-hand side of the inequalities.

The optimum of the dual is now an upper bound to the optimum of the primal.

How do we do the same thing but starting from a minimization linear program?

We can rewrite

$$\begin{array}{ll} \text{minimize} & \mathbf{c}^T\mathbf{y} \\ \text{subject to} & \\ & A\mathbf{y} \geq \mathbf{b} \\ & \mathbf{y} \geq \mathbf{0} \end{array}$$

in an equivalent way as

$$\begin{array}{ll} \text{maximize} & -\mathbf{c}^T\mathbf{y} \\ \text{subject to} & \\ & -A\mathbf{y} \leq -\mathbf{b} \\ & \mathbf{y} \geq \mathbf{0} \end{array}$$

If we compute the dual of the above program we get

$$\begin{array}{ll} \text{minimize} & -\mathbf{b}^T\mathbf{z} \\ \text{subject to} & \\ & -A^T\mathbf{z} \geq -\mathbf{c} \\ & \mathbf{z} \geq \mathbf{0} \end{array}$$

that is,

$$\begin{array}{ll} \text{maximize} & \mathbf{b}^T\mathbf{z} \\ \text{subject to} & \\ & A^T\mathbf{z} \leq \mathbf{c} \\ & \mathbf{y} \geq \mathbf{0} \end{array}$$

So we can form the dual of a linear program in minimization normal form in the same way in which we formed the dual in the maximization case:

- switch the type of optimization,
- introduce as many dual variables as the number of primal constraints (not counting the non-negativity constraints),
- define as many dual constraints (not counting the non-negativity constraints) as the number of primal variables.
- take the transpose of the matrix of coefficients of the left-hand side of the inequality,
- switch the roles of the vector of coefficients in the objective function and the vector of right-hand sides in the inequalities.

Note that:

Fact 6.2 *The dual of the dual of a linear program is the linear program itself.*

We have already proved the following:

Fact 6.3 *If the primal (in maximization standard form) and the dual (in minimization standard form) are both feasible, then*

$$opt(\text{primal}) \leq opt(\text{dual})$$

Which we can generalize a little

Theorem 6.4 (Weak Duality Theorem) *If LP_1 is a linear program in maximization standard form, LP_2 is a linear program in minimization standard form, and LP_1 and LP_2 are duals of each other then:*

- *If LP_1 is unbounded, then LP_2 is infeasible;*
- *If LP_2 is unbounded, then LP_1 is infeasible;*
- *If LP_1 and LP_2 are both feasible and bounded, then*

$$opt(LP_1) \leq opt(LP_2)$$

PROOF: We have proved the third statement already. Now observe that the third statement is also saying that if LP_1 and LP_2 are both feasible, then they have to both be bounded, because every feasible solution to LP_2 gives a finite upper bound to the optimum of LP_1 (which then cannot be $+\infty$) and every feasible solution to LP_1 gives a finite lower bound to the optimum of LP_2 (which then cannot be $-\infty$). □

What is surprising is that, for bounded and feasible linear programs, there is always a dual solution that certifies the exact value of the optimum.

Theorem 6.5 (Strong Duality) *If either LP_1 or LP_2 is feasible and bounded, then so is the other, and*

$$opt(LP_1) = opt(LP_2)$$

To summarize, the following cases can arise:

- If one of LP_1 or LP_2 is feasible and bounded, then so is the other;
- If one of LP_1 or LP_2 is unbounded, then the other is infeasible;
- If one of LP_1 or LP_2 is infeasible, then the other cannot be feasible and bounded, that is, the other is going to be either infeasible or unbounded. Either case can happen.

Lecture 7

Rounding Linear Programs

In which we show how to use linear programming to approximate the vertex cover problem.

7.1 Linear Programming Relaxations

An *integer linear program* (abbreviated ILP) is a linear program (abbreviated LP) with the additional constraints that the variables must take integer values. For example, the following is an ILP:

$$\begin{array}{ll} \text{maximize} & x_1 - x_2 + 2x_3 \\ \text{subject to} & \\ & x_1 - x_2 \leq 1 \\ & x_2 + x_3 \leq 2 \\ & x_1 \in \mathbb{N} \\ & x_2 \in \mathbb{N} \\ & x_3 \in \mathbb{N} \end{array} \tag{7.1}$$

Where $\mathbb{N} = \{0, 1, 2, \ldots\}$ is the set of natural numbers.

The advantage of ILPs is that they are a very expressive language to formulate optimization problems, and they can capture in a natural and direct way a large number of combinatorial optimization problems. The disadvantage of ILPs is that they are a very expressive language to formulate combinatorial optimization problems, and finding optimal solutions for ILPs is NP-hard.

If we are interested in designing a polynomial time algorithm (exact or approximate) for a combinatorial optimization problem, formulating the combinatorial optimization problem as an ILP is useful as a first step in the following methodology (the discussion assumes that we are working with a minimization problem):

- Formulate the combinatorial optimization problem as an ILP;

- Derive a LP from the ILP by removing the constraint that the variables have to take integer value. The resulting LP is called a "relaxation" of the original problem. Note that in the LP we are minimizing the same objective function over a larger set of solutions, so $opt(LP) \leq opt(ILP)$;
- Solve the LP optimally using an efficient algorithm for linear programming;
  - If the optimal LP solution has integer values, then it is a solution for the ILP of cost $opt(LP) \leq opt(ILP)$, and so we have found an optimal solution for the ILP and hence an optimal solution for our combinatorial optimization problem;
  - If the optimal LP solution x^* has fractional values, but we have a *rounding* procedure that transforms x^* into an integral solution x' such that $cost(x') \leq c \cdot cost(x^*)$ for some constant c, then we are able to find a solution to the ILP of cost $\leq c \cdot opt(LP) \leq c \cdot opt(ILP)$, and so we have a c-approximate algorithm for our combinatorial optimization problem.

In this lecture and in the next one we will see how to round fractional solutions of relaxations of the Vertex Cover and the Set Cover problem, and so we will be able to derive new approximation algorithms for Vertex Cover and Set Cover based on linear programming.

7.2 The Weighted Vertex Cover Problem

Recall that in the vertex cover problem we are given an undirected graph $G = (V, E)$ and we want to find a minimum-size set of vertices S that "touches" all the edges of the graph, that is, such that for every $(u, v) \in E$ at least one of u or v belongs to S.

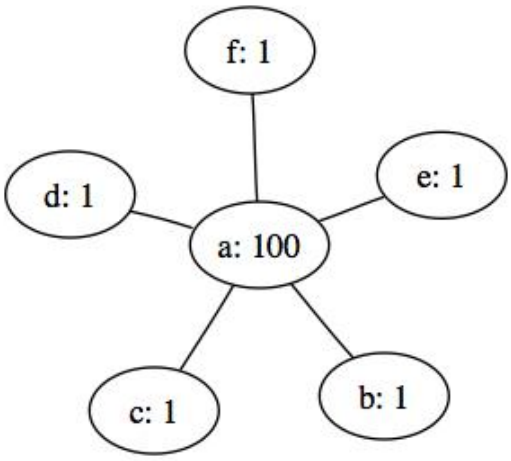
We described the following 2-approximate algorithm:

- Input: $G = (V, E)$
- $S := \emptyset$
- For each $(u, v) \in E$
  - if $u \notin S \wedge v \notin S$ then $S := S \cup \{u, v\}$
- return S

The algorithm finds a vertex cover by construction, and if the condition in the *if* step is satisfied k times, then $|S| = 2k$ and the graph contains a matching of size k, meaning that the vertex cover optimum is at least k and so $|S|$ is at most twice the optimum.

Consider now the *weighted* vertex cover problem. In this variation of the problem, the graph $G = (V, E)$ comes with *costs* on the vertices, that is, for every vertex v we have a non-negative cost $c(v)$, and now we are not looking any more for the vertex cover with the fewest vertices, but for the vertex cover S of minimum total cost $\sum_{v \in S} c(v)$. (The original problem corresponds to the case in which every vertex has cost 1.)

Our simple algorithm can perform very badly on weighted instances. For example consider the following graph:



Then the algorithm would start from the edge (a, b), and cover it by putting a, b into S. This would suffice to cover all edges, but would have cost 101, which is much worse than the optimal solution which consists in picking the vertices $\{b, c, d, e, f\}$, with a cost of 5.

Why does the approximation analysis fail in the weighted case? In the unweighted case, every edge which is considered by the algorithm must cost at least 1 to the optimum solution to cover (because those edges form a matching), and our algorithm invests a cost of 2 to cover that edge, so we get a factor of 2 approximation. In the weighted case, an edge in which one endpoint has cost 1 and one endpoint has cost 100 tells us that the optimum solution must spend at least 1 to cover that edge, but if we want to have both endpoints in the vertex cover we are going to spend 101 and, in general, we cannot hope for any bounded approximation guarantee.

We might think of a heuristic in which we modify our algorithm so that, when it considers an uncovered edge in which one endpoint is much more expensive than the other, we only put the cheaper endpoint in S. This heuristic, unfortunately, also fails completely: imagine a "star" graph like the one above, in which there is a central vertex of cost 100, which is connected to 10,000 other vertices, each of cost 1. Then the algorithm would consider all the 10,000 edges, and decide to cover each of them using the cheaper endpoint, finding a solution of cost 10,000 instead of the optimal solution of picking the center vertex, which has cost 100.

Indeed, it is rather tricky to approximate the weighted vertex cover problem via a combinatorial algorithm, although we will develop (helped by linear programming intuition) such an approximation algorithm by the end of the lecture.

Developing a 2-approximate algorithm for weighted vertex cover via a linear programming relaxation, however, is amazingly simple.

7.3 A Linear Programming Relaxation of Vertex Cover

Let us apply the methodology described in the first section. Given a graph $G = (V, E)$ and vertex costs $c(\cdot)$, we can formulate the minimum vertex cover problem for G as an ILP by using a variable x_v for each vertex v, taking the values 0 or 1, with the interpretation that $x_v = 0$ means that $v \notin S$, and $x_v = 1$ means that $v \in S$. The cost of the solution, which we want to minimize, is $\sum_{v \in V} x_v c(v)$, and we want $x_u + x_v \geq 1$ for each edge (u, v). This gives the ILP

$$\begin{array}{lll} \text{minimize} & \sum_{v \in V} c(v) x_v & \\ \text{subject to} & & \\ & x_u + x_v \geq 1 & \forall (u,v) \in E \\ & x_v \leq 1 & \forall v \in V \\ & x_v \in \mathbb{N} & \forall v \in V \end{array} \tag{7.2}$$

Next, we relax the ILP (7.2) to a linear program.

$$\begin{array}{lll} \text{minimize} & \sum_{v \in V} c(v) x_v & \\ \text{subject to} & & \\ & x_u + x_v \geq 1 & \forall (u,v) \in E \\ & x_v \leq 1 & \forall v \in V \\ & x_v \geq 0 & \forall v \in V \end{array} \tag{7.3}$$

Let us solve the linear program in polynomial time, and suppose that $\mathbf{x}^*$ is an optimal solution to the LP (7.3); how do we "round" it to a 0/1 solution, that is, to a vertex cover? Let's do it in the simplest possible way: round each value to the closest integer, that is, define $x'_v = 1$ if $x^*_v \geq \frac{1}{2}$, and $x'_v = 0$ if $x^*_v < \frac{1}{2}$. Now, find the set corresponding to the integral solution $\mathbf{x}'$, that is $S := \{v : x'_v = 1\}$ and output it. We have:

- *The set S is a valid vertex cover*, because for each edge (u, v) it is true that $x^*_u + x^*_v \geq 1$, and so at least one of x^*_u or x^*_v must be at least $1/2$, and so at least one of u or v belongs to S;
- *The cost of S is at most twice the optimum*, because the cost of S is

$$\begin{aligned} & \sum_{v \in S} c(v) \\ = & \sum_{v \in V} c(v) x'_v \\ \leq & \sum_{v \in V} c(v) \cdot 2 \cdot x^*_v \\ = & \, 2 \cdot opt(LP) \\ \leq & \, 2 \cdot opt(VC) \end{aligned}$$

And that's all there is to it! We now have a polynomial-time 2-approximate algorithm for weighted vertex cover.

7.4 The Dual of the LP Relaxation

The vertex cover approximation algorithm based on linear programming is very elegant and simple, but it requires the solution of a linear program. Our previous vertex cover approximation algorithm, instead, had a very fast linear-time implementation. Can we get a fast linear-time algorithm that works in the weighted case and achieves a factor of 2 approximation? We will see how to do it, and although the algorithm will be completely combinatorial, its *analysis* will use the LP relaxation of vertex cover.

How should we get started in thinking about a combinatorial approximation algorithm for weighted vertex cover?

We have made the following point a few times already, but it is good to stress it again: in order to have any hope to design a provably good approximation algorithm for a minimization problem, we need to have a good technique to prove lower bounds for the optimum. Otherwise, we will not be able to prove that the optimum is at least a constant fraction of the cost of the solution found by our algorithms.

In the unweighted vertex cover problem, we say that if a graph has a matching of size k, then the optimum vertex cover must contain at least k vertices, and that's our lower bound technique. We have already seen examples in which reasoning about matchings is not effective in proving lower bound to the optimum of weighted instances of vertex cover.

How else can we prove lower bounds? Well, how did we establish a lower bound to the optimum in our LP-based 2-approximate algorithm? We used the fact that the optimum of the linear programming relaxation (7.3) is a lower bound to the minimum vertex cover optimum. The next idea is to observe that the cost of any feasible solution to the *dual* of (7.3) is a lower bound to the optimum of (7.3), by weak duality, and hence a lower bound to the vertex cover optimum as well.

Let us construct the dual of (7.3). Before starting, we note that if we remove the $x_v \leq 1$ constraints we are not changing the problem, because any solution in which some variables x_v are larger than 1 can be changed to a solution in which every x_v is at most one while decreasing the objective function, and without contradicting any constraint, so that an optimal solution cannot have any x_v larger than one. Our primal is thus the LP in standard form

$$\begin{array}{lll} \text{minimize} & \sum_{v \in V} c(v) x_v & \\ \text{subject to} & & \\ & x_u + x_v \geq 1 & \forall (u,v) \in E \\ & x_v \geq 0 & \forall v \in V \end{array} \tag{7.4}$$

Its dual has one variable $y_{(u,v)}$ for every edge (u, v), and it is

$$\begin{aligned} \text{maximize} \quad & \sum_{(u,v)\in E} y_{(u,v)} \\ \text{subject to} \quad & \\ & \sum_{u:(u,v)\in E} y_{(u,v)} \leq c(v) \quad \forall v \in V \\ & y_{(u,v)} \geq 0 \qquad\qquad\quad \forall (u,v) \in E \end{aligned} \tag{7.5}$$

That is, we want to assign a nonnegative "charge" $y_{(u,v)}$ to each edge, such that the total charge over all edges is as large as possible, but such that, for every vertex, the total charge of the edges incident on the vertex is at most the cost of the vertex. From weak duality and from the fact that (7.4) is a relaxation of vertex cover, we have that for any such system of charges, the sum of the charges is a lower bound to the cost of the minimum vertex cover in the weighted graph $G = (V, E)$ with weights $c(\cdot)$.

Example 7.1 (Matchings) *Suppose that we have an unweighted graph $G = (V, E)$, and that a set of edges $M \subseteq E$ is a matching. Then we can define $y_{(u,v)} := 1$ if $(u, v) \in M$ and $y_{(u,v)} := 0$ if $(u, v) \notin M$. This is a feasible solution for (7.5) of cost $|M|$.*

This means that any lower bound to the optimum in the unweighted case via matchings can also be reformulated as lower bounds via feasible solutions to (7.5). The latter approach, however, is much more powerful.

Example 7.2 *Consider the weighted star graph from Section 7.2. We can define $y_{(a,x)} = 1$ for each vertex $x = b, c, d, e, f$, and this is a feasible solution to (7.5). This proves that the vertex cover optimum is at least 5.*

7.5 Linear-Time 2-Approximation of Weighted Vertex Cover

Our algorithm will construct, in parallel, a valid vertex cover S, in the form of a valid integral solution $\mathbf{x}$ to the ILP formulation of vertex cover (7.2), and a feasible solution $\mathbf{y}$ to the dual (7.5) of the linear programming relaxation, such that the cost of $\mathbf{y}$ is at least half the cost S. Before starting, it is helpful to reformulate our old algorithms in this language

- Input: undirected, unweighted, graph $G = (V, E)$
- $\mathbf{x} = (0, \cdots, 0)$
- $\mathbf{y} = (0, \cdots, 0)$
- for each edge $(u, v) \in E$
  - if $x_u < 1$ and $x_v < 1$ then

    * $y_{(u,v)} := 1$
    * $x_u := 1$
    * $x_v := 1$

- $S := \{v : x_v = 1\}$
- return $S, \mathbf{y}$

Our goal is to modify the above algorithm so that it can deal with vertex weights, while maintaining the property that it finds an integral feasible $\mathbf{x}$ and a dual feasible $\mathbf{y}$ such that $\sum_{v \in V} c(v)x_v \leq 2 \cdot \sum_{(u,v) \in V} y_{u,v}$. The key property to maintain is that when we look at the edge (u, v), and we find it uncovered, what we are going to "spend" in order to cover it will be at most $2y_{u,v}$, where $y_{u,v}$ will be a charge that we assign to (u, v) without violating the constraints of (7.5).

We will get simpler formulas if we think in terms of a new set of variables p_v, which represent how much we are willing to "pay" in order to put v in the vertex cover; at the end, if $p_v = c_v$ then the vertex v is selected, and $x_v = 1$, and if $p_v < c_v$ then we are not going to use v in the vertex cover. Thus, in the integral solution, we will have $x_v = \lfloor p_v/c(v) \rfloor$, and so $c(v) \cdot x_v \leq p_v$ and so the total amount we are willing to pay, $\sum_v p_v$ is an upper bound to the cost of the integral solution $\sum_v c(v) \cdot x_v$.

Initially, we start from the all-zero dual solution $\mathbf{y} = \mathbf{0}$ and from no commitment to pay for any vertex, $\mathbf{p} = \mathbf{0}$. When we consider an edge (u, v), if $p_u = c(u)$ or $p_v = c(v)$, we have committed to pay for at least one of the endpoints of (u, v), and so the edge will be covered. If $p_u < c(u)$ and $p_v < c(v)$, we need to commit to pay for at least one of the endpoints of the edge. We need to pay an extra $c(u) - p_u$ to make sure u is in the vertex cover, or an extra $c(v) - p_v$ to make sure that v is. We will raise, and here is the main idea of the algorithm, *both* the values of p_u and p_v by the smallest of the two values. This will guarantee that we cover (u, v) by "fully funding" one of the endpoints, but it will also put some extra "funding" into the other vertex, which might be helpful later. We also set $y_{(u,v)}$ to $\min\{c(u) - p_u, c(v) - p_v\}$.

Here is the psedocode of the algorithm:

- Input: undirected, unweighted, graph $G = (V, E)$
- $\mathbf{p} = (0, \cdots, 0)$
- $\mathbf{y} = (0, \cdots, 0)$
- for each edge $(u, v) \in E$
    - if $p_u < c(u)$ and $p_v < c(v)$ then
        * $y_{(u,v)} := \min\{c(u) - p_u, c(v) - p_v\}$
        * $p_u := p_u + \min\{c(u) - p_u, c(v) - p_v\}$
        * $p_v := p_v + \min\{c(u) - p_u, c(v) - p_v\}$

- $S := \{v : p_v \geq c(v)\}$
- return $S, \mathbf{y}$

The algorithm outputs a correct vertex cover, because for each edge (u, v), the algorithm makes sure that at least one of $p_u = c(u)$ or $p_v = c(v)$ is true, and so at least one of u or v belongs to S at the end.

Clearly, we have

$$cost(S) = \sum_{v \in S} c(v) \leq \sum_{v \in V} p_v$$

Next, we claim that the vector $\mathbf{y}$ at the end of the algorithm is a feasible solution for the dual (7.5). To see this, note that, for every vertex v,

$$\sum_{u:(u,v) \in E} y_{(u,v)} = p_v$$

because initially all the $y_{(u,v)}$ and all the p_v are zero, and when we assign a value to a variable $y_{(u,v)}$ we also simultaneously raise p_u and p_v by the same amount. Also, we have that, for every vertex v

$$p_v \leq c(v)$$

by construction, and so the charges $\mathbf{y}$ satisfy all the constraints

$$\sum_{u:(u,v) \in E} y_{(u,v)} = c(v)$$

and define a feasible dual solution. We then have

$$\sum_{(u,v) \in E} y_{(u,v)} \leq opt_{VC}(G)$$

by weak duality. Finally, every time we give a value to a $y_{(u,v)}$ variable, we also raise the values of p_u and p_v by the same amount, and so the sum of our payment commitments is exactly twice the sum of the charges $y_{(u,v)}$

$$\sum_{v \in V} p_v = 2 \sum_{(u,v) \in E} y_{(u,v)}$$

Putting all together we have

$$cost(S) \leq 2 \cdot opt_{VC}(G)$$

and we have another 2-approximate algorithm for weighted vertex cover!

It was much more complicated than the simple rounding scheme applied to the linear programming optimum, but it was worth it because now we have a linear-time algorithm, and we have understood the problem quite a bit better.

Lecture 8

Randomized Rounding

In which we show how to round a linear programming relaxation in order to approximate the set cover problem, and we show how to reason about the dual of the relaxation to derive a simple combinatorial approximation algorithm for the weighted case.

Recall that in Set Cover we are given a finite set U and a collection $S_1, \ldots, S_n$ of subsets of U. We want to find the fewest sets whose union is U, that is, the smallest $I \subseteq \{1, \ldots, n\}$ such that $\bigcup_{i \in I} S_i = U$.

We described an algorithm whose approximation guarantee is $\ln |U| + O(1)$. The lower bound technique that we used in our analysis was to realize that if we have a subset $D \subseteq U$ of elements, such that every set S_i contains at most t elements of D, then $opt \geq |D|/t$.

Today we will describe a linear programming relaxation of set cover, and we will show a way to round it, in order to achieve an $O(\log |U|)$ approximation.

We will also show that the lower bounds to the optimum that we used in the analysis of the greedy algorithm can also be expressed as feasible solutions for the dual of our linear programming relaxation.

8.1 A Linear Programming Relaxation of Set Cover

We begin by formulating the set cover problem as an Integer Linear Programming problem. Given an input $(U, S_1, \ldots, S_n)$ of the set cover problem, we introduce a variable x_i for every set S_i, with the intended meaning that $x_i = 1$ when S_i is selected, and $x_i = 0$ otherwise. We can express the set cover problem as the following integer linear program:

$$\begin{array}{lll} \text{minimize} & \sum_{i=1}^{n} x_i & \\ \text{subject to} & & \\ & \sum_{i : v \in S_i} x_i \geq 1 & \forall v \in U \\ & x_i \leq 1 & \forall i \in \{1, \ldots, n\} \\ & x_i \in \mathbb{N} & \forall i \in \{1, \ldots, n\} \end{array} \tag{8.1}$$

From which we derive the linear programming relaxation

$$\begin{array}{lll} \text{minimize} & \sum_{i=1}^{n} x_i & \\ \text{subject to} & & \\ & \sum_{i:v\in S_i} x_i \geq 1 & \forall v \in U \\ & x_i \leq 1 & \forall i \in \{1,\ldots,n\} \\ & x_i \geq 0 & \forall i \in \{1,\ldots,n\} \end{array} \tag{8.2}$$

Remark 8.1 *In an earlier lecture, we noted that every instance G of the vertex cover problem can be translated into an instance $(U, S_1, \ldots, S_n)$ of the set cover problem in which every element of U belongs precisely to two sets. The reader should verify that the linear programming relaxation (8.2) of the resulting instance of set cover is identical to the linear programming relaxation of the vertex cover problem on the graph G.*

More generally, it is interesting to consider a *weighted* version of set cover, in which we are given the set U, the collection of sets $S_1, \ldots, S_n$, and also a *weight* w_i for every set. We want to find a sub-collection of *minimal total weight* whose union is U, that is, we want to find I such that $\bigcup_{i\in I} S_i = U$, and such that $\sum_{i\in I} w_i$ is minimized. The unweighted problem corresponds to the case in which all weights w_i equal 1.

The ILP and LP formulation of the unweighted problem can easily be generalized to the weighted case: just change the objective function from $\sum_i x_i$ to $\sum_i w_i x_i$.

$$\begin{array}{lll} \text{minimize} & \sum_{i=1}^{n} w_i x_i & \\ \text{subject to} & & \\ & \sum_{i:v\in S_i} x_i \geq 1 & \forall v \in U \\ & x_i \leq 1 & \forall i \in \{1,\ldots,n\} \\ & x_i \geq 0 & \forall i \in \{1,\ldots,n\} \end{array} \tag{8.3}$$

Suppose now that we solve the linear programming relaxation (8.3), and we find an optimal fractional solution $\mathbf{x}^*$ to the relaxation, that is, we are given a number x_i^* in the range $[0, 1]$ for every set S_i. Unlike the case of vertex cover, we cannot round the x_i^* to the nearest integer, because if an element u belongs to 100 sets, it could be that $x_i^* = 1/100$ for each of those sets, and we would be rounding all those numbers to zero, leaving the element u not covered. If we knew that every element u belongs to at most k sets, then we could round the numbers $\geq 1/k$ to 1, and the numbers $< 1/k$ to zero. This would give a feasible cover, and we could prove that we achieve a k-approximation. Unfortunately, k could be very large, even $n/2$, while we are trying to prove an approximation guarantee that is never worse than $O(\log n)$.

Maybe the next most natural approach after rounding the x_i^* to the nearest integer is to think of each x_i^* as a *probability*, and we can think of the solution $\mathbf{x}^*$ as describing a probability distribution over ways of choosing some of the subsets $S_1, \ldots, S_n$, in which we choose S_1 with probability x_1^*, S_2 with probability x_2^*, and so on.

Algorithm *RandomPick*

- Input: values $(x_1, \ldots, x_n)$ feasible for (8.3)
- $I := \emptyset$
- for $i = 1$ to n
  - with probability x_i, assign $I := I \cup \{i\}$, otherwise do nothing
- return I

Using this probabilistic process, the expected cost of the sets that we pick is $\sum_i w_i x_i^*$, which is the same as the cost of $\mathbf{x}^*$ in the linear programming problem, and is at most the optimum of the set cover problem. Unfortunately, the solution that we construct could have a high probability of missing some of the elements.

Indeed, consider the probabilistic process "from the perspective" of an element u. The element u belongs to some of the subsets, let's say for simplicity the first k sets $S_1, \ldots, S_k$. As long as we select at least one of $S_1, \ldots, S_k$, then we are going to cover u. We select S_i with probability x_i^* and we know that $x_1^* + \cdots x_k^* \geq 1$; what is the probability that u is covered? It is definitely not 1, as we can see by thinking of the case that u belongs to only two sets, and that each set has an associated x_i^* equal to $1/2$; in such a case u is covered with probability $3/4$. This is not too bad, but maybe there are $n/10$ elements like that, each having probability $1/4$ of remaining uncovered, so that we would expect $n/40$ uncovered elements on average. In some examples, $\Omega(n)$ elements would remain uncovered with probability $1 - 2^{-\Omega(n)}$. How do we deal with the uncovered elements?

First of all, let us see that every element has a reasonably good probability of being covered.

Lemma 8.2 *Consider a sequence of k independent experiments, in which the i-th experiment has probability p_i of being successful, and suppose that $p_1 + \cdots + p_k \geq 1$. Then there is a probability $\geq 1 - 1/e$ that at least one experiment is successful.*

PROOF: The probability that no experiment is successful is

$$(1 - p_1) \cdot \cdots \cdot (1 - p_k) \tag{8.4}$$

This is the kind of expression for which the following trick is useful: $1 - x$ is approximately e^{-x} for small values of x.

More precisely, using the Taylor expansion around 0 of e^x, we can see that, for $0 \leq x \leq 1$

$$e^{-x} = 1 - x + \frac{x^2}{2} - \frac{x^3}{3!} + \cdots$$

and so

$$1 - x \leq e^{-x} \tag{8.5}$$

because

$$e^{-x} - (1-x) = \left(\frac{x^2}{2} - \frac{x^3}{3!}\right) + \left(\frac{x^4}{4!} - \frac{x^5}{5!}\right) \cdots \geq 0$$

where the last inequality follows from the fact that we have an infinite sum of non-negative terms.

Combining (8.4) and (8.5) we have

$$(1-p_1)\cdot\cdots\cdot(1-p_k) \leq e^{-p_1-\ldots-p_k} \leq e^{-1}$$

□

Our randomized rounding process will be as follows: we repeat the procedure *RandomPick* until we have covered all the elements.

Algorithm *RandomizedRound*

1. Input: $x_1, \ldots, x_n$ feasible for (8.3)
2. $I := \emptyset$
3. while there are elements u such that $u \notin \bigcup_{i\in I} S_i$
   - for $i := 1$ to n
     - with probability x_i, assign $I := I \cup \{i\}$, otherwise do nothing
4. return I

How do we analyze such a procedure? We describe a very simple way to get a loose estimate on the quality of the cost of the solution found by the algorithm.

Fact 8.3 *There is a probability at most e^{-100} that the while loop is executed for more than $\ln|U| + 100$ times. In general, there is a probability at most e^{-k} that the while loop is executed form more than $\ln|U| + k$ times.*

Proof: The probability that we need to run the *while* loop for more than $\ln|U| + k$ times is the same as the probability that, if we run the body of the *while* loop (that is, the *RandomPick* procedure) for exactly $\ln|U| + k$ steps, we end up with some uncovered elements.

Consider an element $v \in U$. For each iteration of the *while* loop, there is a probability at most $1/e$ that v is not covered by the sets added to I in that iteration. The probability that v is not covered after $\ln|U| + k$ iterations is then at most

$$\left(\frac{1}{e}\right)^{\ln|U|+k} = \frac{1}{|U|}\cdot e^{-k}$$

The probability that, after $\ln |U| + k$ iterations, there is an element that is not covered, is at most the sum over all v of the probability that v is not covered, which is at most e^{-k}. □

Fact 8.4 *Fix any positive integer parameter t and any feasible solution $(x_1, \ldots, x_n)$ for (8.3). Then the expected size of I in Algorithm RandomizedRound on input $(x_1, \ldots, x_n)$ after t iterations (or at the end of the algorithm, if it ends in fewer than t iterations) is at most*

$$t \cdot \sum_{i=1}^{n} w_i x_i$$

PROOF: Let $A_1, \ldots, A_t$ be the total cost of the elements assigned to I by the algorithm during the first, second, ..., t-th iteration, respectively. Let W denote the total weight $\sum_{i \in I} w_i$ after t iterations (or at the end of the algorithm if it terminates in fewer than t iterations). Note that these are random variables determined by the random choices made by the algorithm. Clearly, we have, with probability 1,

$$W \leq A_1 + \cdots + A_t$$

where we do not have equality because certain elements might be assigned to I in more than one iteration.

If the algorithm stops before the j-th iteration, then $A_j = 0$, because there is no j-th iteration and so no assignment happens during it. Conditioned on the j-th iteration occurring, the expectation of A_j is

$$\mathbb{E}[A_j | j\text{th iteration happens }] = \sum_{i=1}^{n} w_i x_i$$

and so we have

$$\mathbb{E}[A_j] \leq \sum_{i=1}^{n} w_i x_i$$

□

Recall that Markov's inequality says that if X is a non-negative random variable (for example, the size of a set), then, for every $c > \mathbb{E}\, X$

$$\mathbb{P}[X \geq c] \leq \frac{\mathbb{E}\, X}{c}$$

For example, $\mathbb{P}[X \geq 2\, \mathbb{E}\, X] \leq 1/2$.

We can combine these observations to get a rather loose bound on the size of the set output by RandomizedRound

Fact 8.5 *Given an optimal solution $(x_1^*, \ldots, x_n^*)$ to (8.3), algorithm RandomizedRound outputs, with probability $\geq .45$, a feasible solution to the set cover problem that contains at most $(2 \ln |U| + 6) \cdot opt$ sets.*

PROOF: Let $t := \ln |U| + 3$. There is a probability at most $e^{-3} < .05$ that the algorithm runs the *while* loop for more than t steps. After the first t steps, the expected total weight of the solution I is at most $t \sum_i w_i x_i^*$, which is at most $t \cdot opt$. Thus, the probability that the solution I, after the first t steps, contains sets of total weight more than $2t \cdot opt$ is at most $1/2$. Taking the union of the two events, there is a probability at most .55 that either the algorithm runs for more than t iterations, or that it adds to its solution sets of total cost more than $2t \cdot opt$ in the first t steps.

Equivalently, there is a probability at least .45 that the algorithm halts within t iterations and that, within that time, it constructs a solution of total cost at most $2t \cdot opt$. □

8.2 The Dual of the Relaxation

In a past lecture, we analyzed a simple greedy algorithm for unweighted set cover, that repeatedly picks a set that covers the largest number of uncovered elements, until all elements are covered, and we proved that the algorithm uses at most $(\ln |U| + O(1)) \cdot opt$ sets.

As in all analyses of approximation algorithms, we had to come up with ways to prove lower bounds to the optimum. In the unweighted case, we noted that if we have a subset $D \subseteq U$ of elements (D for "difficult" to cover), and every S_i contains at most t elements of D, then $opt \geq |D|/t$. In the weighted case, neither this lower bound technique nor the approximation analysis works. It is possible to modify the algorithm so that, at every step, it picks the set that is most "cost-effective" at covering uncovered elements, and we can modify the analysis to take weights into accounts. This modification will be easier to figure out if we first think about the analysis of our previous algorithm in terms of the dual of the LP relaxation of set cover.

To form the dual of (8.3), we first note that the constraints $x_i \leq 1$ do not change the optimum, because a solution in which some x_is are bigger than 1 can be converted to a solution in which all variables are ≤ 1 while decreasing the objective function, and so no variable is larger than 1 in an optimal solution, even if we do not have the constraint $x_i \leq 1$.

If we work with this simplified version of the LP relaxation of set cover

$$\begin{array}{lll} \text{minimize} & \sum_{i=1}^{n} w_i x_i & \\ \text{subject to} & & \\ & \sum_{i:v \in S_i} x_i \geq 1 & \forall v \in U \\ & x_i \geq 0 & \forall i \in \{1, \ldots, n\} \end{array} \tag{8.6}$$

we see that its dual is an LP with one variable y_v for every element $v \in U$, and it is defined

as follows:

$$\begin{array}{lll} \text{maximize} & \sum_{v \in U} y_v & \\ \text{subject to} & & \\ & \sum_{v: v \in S_i} y_v \leq w_i & \forall i \in \{1, \ldots, n\} \\ & y_v \geq 0 & \forall v \in U \end{array} \tag{8.7}$$

That is, we assign a "charge" to every element, subject to the constraint that, for every set, the sum of the charges of the elements of the set are at most 1. The cost of the dual solution (and a lower bound to the optimum of the set cover problem) is the sum of the charges.

In the unweighted case, $w_i = 1$ for every i. Suppose that we are in the unweighted case, and that we notice a set $D \subseteq U$ of elements such that every S_i contains at most t elements of D. Then we can form the feasible dual solution

$$y_v := \begin{cases} \frac{1}{t} & \text{if } v \in D \\ 0 & \text{otherwise} \end{cases}$$

This is a feasible dual solution of cost $|D|/t$, and so it is a way to formulate our old lower bound argument in the language of dual solutions. The simplest extension of this example to the weighted setting is that: if we have a set D of elements such that for every set S_i of weight w_i we have that S_i contains at most $t \cdot w_i$ elements of D; then the assignment $y_v := 1/t$ for $v \in D$ and $y_v := 0$ for $v \notin D$ is a feasible dual solution of cost $|D|/t$, and so the optimum is at most $|D|/t$.

These observations are already enough to derive a version of the greedy algorithm for weighted instances.

Algorithm: SetCoverGreedy

- Input: set U, subsets $S_1, \ldots, S_n$, weights $w_1, \ldots, w_n$
- $I := \emptyset$
- while there are uncovered elements $v \in U$ such that $v \notin S_i$ for every $i \in I$
  - let D be the set of uncovered elements
  - for every set S_i, let $e_i := |D \cap S_i|/w_i$ be the *cost effectiveness* of S_i
  - let S_{i^*} be a set with the best cost-effectiveness
  - $I := I \cup \{i^*\}$
- return I

We adapt the analysis of the unweighted case.

Let $v_1, \ldots, v_m$ be an enumeration of the elements of U in the order in which they are covered by the algorithm. Let c_j be the cost-effectiveness of the set that was picked at the iteration in which v_j was covered for the first time. Then, in that iteration, there is a set D of at

least $m - j + 1$ elements, and every set S_i of weight w_i contains at most $w_i c_j$ elements of D. This means that setting $y_{v_1} = \cdots = y_{v_{j-1}} = 0$ and $y_{v_j} = \cdots = y_{v_m} = 1/c_j$ is a feasible dual solution of cost $(m - j + 1)/c_j$, and so

$$opt \geq (m - j + 1)\frac{1}{c_j}$$

If we denote by apx the cost of the solution found by our algorithm, we see that

$$apx = \sum_{j=1}^{m} \frac{1}{c_j}$$

because, at each iteration, if we pick a set S_{i^*} of weight w_{i^*} that covers t new elements, then each of those t elements has a parameter c_j equal to t/w_{i^*}, and so when we sum $1/c_j$ over all elements v_j that are covered for the first time at that iteration, we get exactly the weight w_{i^*}.

Rearranging the equations, we get again

$$apx \leq \sum_{j=1}^{m} \frac{1}{m - j + 1} \cdot opt \leq (\ln m + O(1)) \cdot opt$$

Lecture 9

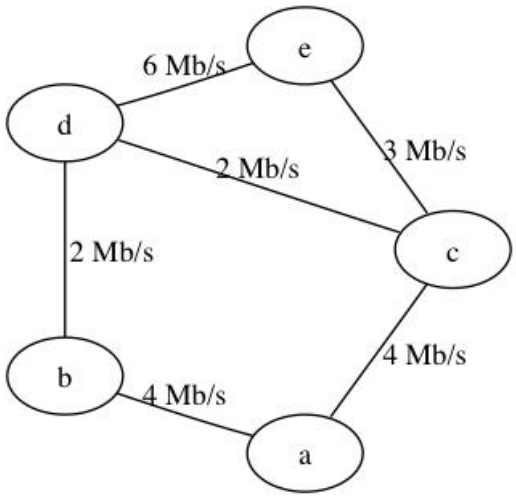
Max Flow

In which we introduce the maximum flow problem.

9.1 Flows in Networks

Today we start talking about the *Maximum Flow* problem. As a motivating example, suppose that we have a communication network, in which certain pairs of nodes are linked by connections; each connection has a limit to the rate at which data can be sent. Given two nodes on the network, what is the maximum rate at which one can send data to the other, assuming no other pair of nodes are attempting to communicate?

For example, consider the following network, and suppose that a needs to send data at a rate of $6Mb/s$ to e. Is this possible?



The answer is yes: a can split its stream of data, and send $4Mb/s$ to c and $2Mb/s$ to b. The node b relays the $2Mb/s$ stream of data that it receives from a to d, while node c splits

the $4Mb/s$ stream that it receives into two parts: it relays $3Mb/s$ of data to e, and send the remaining $1Mb/s$ to d. Overall, d receives $3Mb/s$ of data, which it relays to e.

Can a send more data, such as, say, $7Mb/s$ to e? The answer is no. Consider the two links (b, d) and (a, c) in the network, and suppose you removed them from the network. Then there would be no way reach e from a, and no communication would be possible. This means that all the data that a sends to e must pass through one of those links. Between the two of them, the two links (b, d) and (a, c) support at most the transmission of $6Mb/s$ of data, and so that is the maximum rate at which a can send data to e.

The networks that we will work with could model other settings in which "stuff" has to go from one place to another, subject to capacity constraints in links. For example the network could model a public transit system, or a highway system. In some cases, we will be interested in instances of the problem which are constructed to model other combinatorial problems.

In some such applications, it makes sense to consider networks in which the capacity of a link depends on the direction, so that the capacity of the link $u \to v$ could be different from the capacity of the link $v \to u$. Formally, an instance of the maximum flow problem is defined as follows.

Definition 9.1 *A network is a directed graph $G(V, E)$, in which*

- *a vertex $s \in V$ and a vertex $t \in V$ are specified as being the source node and the sink node, respectively.*
- *every directed edge $(u, v) \in E$ has a positive capacity $c(u, v) > 0$ associated to it. If both the edges (u, v) and (v, u) belong to E, we allow their capacities to be different.*

Sometimes we will write expressions that include capacities between pairs of vertices that are not connected by an edge. In that case the convention is that the capacity is zero.

A *flow* in a network is a specification of how to route "stuff" from s to t so that no link is used beyond its capacity, and so that every link, except the sender s and the receiver t, relays out "stuff" at exactly the same rate at which it receives from other vertices. In the motivating example of a communication network, if nodes where sending out less data than they receive, there would be data loss, and they cannot send out more data than they receive because they are simply forwarding data. Formally, we have the following definition.

Definition 9.2 *A flow in an network (G, s, t, c) is an assignment of a non-negative number $f(u, v)$ to every edge $(u, v) \in E$ such that*

- *For every edge $(u, v) \in E$, $f(u, v) \leq c(u, v)$;*
- *For every vertex $v \in V$, $\sum_{u \in V} f(u, v) = \sum_{w \in V} f(v, w)$*

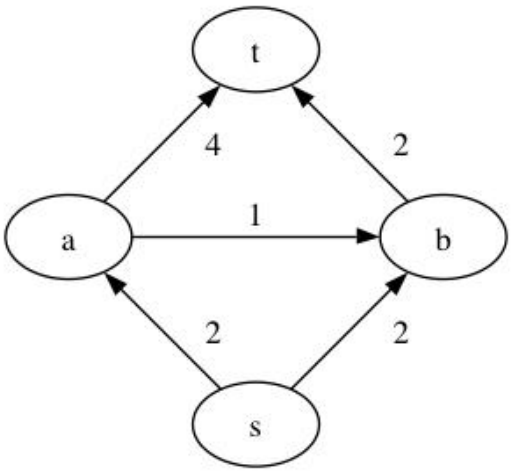
where we follow the convention that $f(u, v) = 0$ if $(u, v) \notin E$. (This convention is useful otherwise the second condition would have to be something like $\sum_{u:(u,v)\in E} f(u, v) = \sum_{w:(v,w)\in E} f(v, w)$

The *cost* of the flow (the throughput of the communication, in the communication network example) is

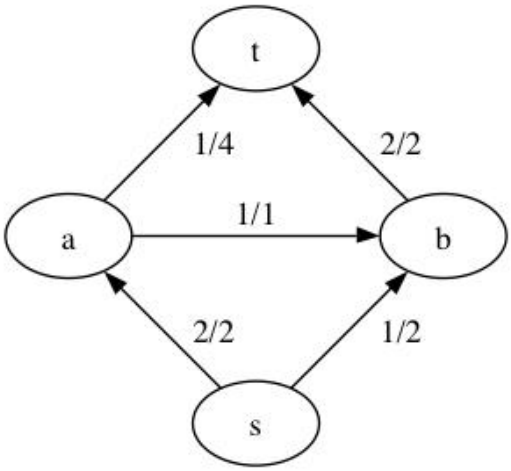
$$\sum_v f(s, v)$$

In the *maximum flow* problem, given a network we want to find a flow of maximum cost.

For example, here is an example of a network:



And the following is a flow in the network (a label x/y on an edge (u, v) means that the flow $f(u, v)$ is x and the capacity $c(u, v)$ is y).



Is the flow optimal? We are only sending 3 units of flow from s to t, while we see that we can send 2 units along the $s \to a \to t$ path, and another 2 units along the $s \to b \to t$ path,

for a total of 4, so the above solution is not optimal. (Is the solution we just discussed, of cost 4, optimal?)

In general, how do we reason about the optimality of a flow? This is an important question, because when we talked about approximation algorithms for minimization problems we noted that, in each case, we were able to reason about the approximation quality of an algorithm only because we had ways to prove lower bounds to the optimum. Here that we are dealing with a maximization problem, if we are going to reason about the quality of solutions provided by our algorithms, we need ways to prove upper bounds to the optimum.

When we considered the first example of the lecture, we noticed that if we look at any set of edges whose removal disconnects the receiver from the sender, then all the flow from the sender to the receiver must pass through those edges, and so their total capacity is an upper bound to the cost of any flow, including the optimal flow. This motivates the following definition.

Definition 9.3 (Cut) *A cut in a network $(G = (V, E), s, t, c)$, is partition $(A, V - A)$ of the set of vertices V into two sets, such that $s \in A$ and $t \in V - A$. We will usually identify a cut with the set A that contains s. The capacity of a cut is the quantity*

$$c(A) := \sum_{u \in A, v \notin A} c(u, v)$$

The motivation for the definition is the following: let A be a cut in a network, that is, a set of vertices that contains s and does not contain t. Consider the set of edges $\{(u, v) \in E : u \in A \wedge v \notin A\}$. If we remove those edges from the network, then it is not possible to go from any vertex in A to any vertex outside A and, in particular, it is not possible to go from s to t. This means that all the flow from s to t must pass through those edges, and so the total capacity of those edges (that, is $c(A)$) is an upper bound to the cost of any feasible flow.

Even though what we just said is rather self-evident, let us give a rigorous proof, because this will help familiarize ourselves with the techniques used to prove rigorous results about flows. (Later, we will need to prove statements that are far from obvious.)

Lemma 9.4 *For every network (G, s, t, c), any flow f in the network, and any cut A,*

$$\sum_{v \in V} f(s, v) \leq \sum_{a \in A, b \notin A} c(a, b)$$

That is, the cost of the flow is at most the capacity of the cut.

We will derive the lemma from Fact 9.5 below.

If (G, s, t, c) is a network, f is a flow, and A is a cut, define the *net flow out of A* to be

$$f(A) := \sum_{a \in A, b \notin A} f(a, b) - \sum_{b \notin A, a \in A} f(b, a)$$

that is, the total flow from A to $V - A$ minus the total flow from $V - A$ to A. Then we have:

Fact 9.5 *For every network (G, s, t, c), every flow f, and every cut A, the net flow out of A is the same as the cost of the flow:*

$$f(A) = \sum_{v \in V} f(s, v)$$

Proof: Consider the expression

$$S := \sum_{v \in V} f(s, v) - \sum_{a \in A} \left(\sum_{v \in V} f(v, a) - \sum_{w \in V} f(a, w) \right) + \sum_{a \in A, b \notin A} f(a, b) - \sum_{b \in B, a \in A} f(b, a)$$

on the one hand, we have

$$S = 0$$

because for every edge (u, v) such that at least one of u, v is in A we see that $f(u, v)$ appears twice in the expression for S, once with a $+$ sign and once with a $-$ sign, so all terms cancel. On the other hand, we have

$$\sum_{a \in A} \left(\sum_{v \in V} f(v, a) - \sum_{w \in V} f(a, w) \right) = 0$$

because of the definition of flow, and so

$$\sum_{v \in V} f(s, v) = \sum_{a \in A, b \notin A} f(a, b) - \sum_{b \in B, a \in A} f(b, a) = f(A)$$

□

To prove Lemma 9.4, consider any network (G, s, t, c), any flow f and any cut A. We have

$$cost(f) = f(A) \leq \sum_{a \in A, b \notin A} f(u, v) \leq \sum_{a \in A, b \notin A} c(u, v) = c(A)$$

So we have proved Lemma 9.4, and we have a way to "certify" the optimality of a flow, if we are able to find a flow and a cut such that the cost of the flow is equal to the capacity of the cut.

Consider now the complementary question: how do we see if a given flow in a network can be improved? That is, what is a clear sign that a given flow is not optimal? If we see a path from s to t such that all the edges are used at less than full capacity, then it is clear that we can push extra flow along that path and that the solution is not optimal. Can there be other cases in which a given solution is not optimal? Indeed there can. Going back to the last example that we considered, we had a flow of cost 3, which was not optimal (because

a flow of cost 4 existed), but if we look at the three possible paths from s to t, that is, $s \to a \to t$, $s \to a \to b \to t$, and $s \to b \to t$, they each involve an edge used at full capacity.

However, let us reason in the following way: suppose that, in addition to the edges shown in the last picture, there were also an edge of capacity 1 from b to a. Then we would have had the path $s \to b \to a \to t$ in which every edge has one unit of residual capacity, and we could have pushed an extra unit of flow along that path. In the resulting solution, a sends one unit flow to b, and b sends one unit of flow to a, a situation that is perfectly equivalent to a and b not sending each other anything, so that the extra edge from b to a is not needed after all. In general, if we are sending $f(u, v)$ units of flow from u to v, then we are effectively increasing the capacity of the edge from v to u, because we can "simulate" the effect of sending flow from v to u by simply sending less flow from u to v. These observations motivate the following definition:

Definition 9.6 (Residual Network) *Let $N = (G, s, t, c)$ be a network, and f be a flow. Then the residual network of N with respect to f is a network in which the edge u, v has capacity*

$$c(u, v) - f(u, v) + f(v, u)$$

The idea is that, in the residual network, the capacity of an edge measures how much *more* flow can be pushed along that edge in addition to the flow f, without violating the capacity constraints. The edge (u, v) starts with capacity $c(u, v)$, and $f(u, v)$ units of that capacity are taken by the current flow; in addition, we have $f(v, u)$ additional units of "virtual capacity" that come from the fact that we can reduce the flow from v to u.

An *augmenting path* in a network is a path from s to t in which every edge has positive capacity in the residual network. For example, in our last picture, the path $s \to b \to a \to t$ is an augmenting path.

The *Ford-Fulkerson* algorithm is a simple greedy algorithm that starts from an empty flow and, as long as it can find an augmenting path, improves the current solution using the path.

Algorithm *Ford-Fulkerson*

- Input: network $(G = (V, E), s, t, c)$
- $\forall u, v. f(u, v) := 0$
- compute the capacities $c'(\cdot, \cdot)$ of the residual network
- while there is a path p from s to t such that all edges in the path have positive residual capacity
  - let $c'_{\min}$ be the smallest of the residual capacities of the edges of p
  - let f' be a flow that pushes $c'_{\min}$ units of flow along p, that is, $f'(u, v) = c'_{\min}$ if $(u, v) \in p$, and $f'(u, v) = 0$ otherwise
  - $f := f + f'$, that is, $\forall u, v. f(u, v) := f(u, v) + f'(u, v)$
  - for every pair of vertices such that $f(u, v)$ and $f(v, u)$ are both positive, let $f_{\min} := \min\{f(u, v), f(v, u)\}$ and let $f(u, v) := f(u, v) - f_{\min}$ and $f(v, u) := f(v, u) - f_{\min}$
  - recompute the capacities $c'(\cdot, \cdot)$ of the residual network according to the new flow
- return f

At every step, the algorithm increases the cost of the current solution by a positive amount $c'_{\min}$ and, the algorithm converges in finite time to a solution that cannot be improved via an augmenting path. Note the "clean-up" step after the flow is increased, which makes sure that flow pushed along a "virtual edge" in the residual network is realized by reducing the actual flow in the opposite direction. The following theorem shows that the Ford-Fulkerson algorithm is optimal, and it proves the important fact that whenever a cut is optimal, its optimality can always be proved by exhibiting a cut whose capacity is equal to the cost of the flow.

Theorem 9.7 (Max Flow-Min Cut) *The following three conditions are equivalent for a flow f in a network:*

1. *There is a cut whose capacity is equal to the cost of f*
2. *The flow f is optimal*
3. *There is no augmenting path for the flow f*

PROOF: We have already proved that (1) implies (2), and it is clear that (2) implies (3), so the point of the theorem is to prove that (3) implies (1).

Let f be a flow such that there is no augmenting path in the residual network. Take A to be the set of vertices reachable from s (including s) via edges that have positive capacity in the residual network. Then:

- $s \in A$ by definition
- $t \notin A$ otherwise we would have an augmenting path.

So A is a cut. Now observe that for every two vertices $a \in A$ and $b \notin A$, the capacity of the edge (a, b) in the residual network must be zero, otherwise we would be able to reach b from s in the residual network via positive-capacity edges, but $b \notin A$ means that no such path can exist.

Recall that the residual capacity of the edge (a, b) is defined as

$$c(a, b) - f(a, b) + f(b, a)$$

and that $f(a, b) \leq c(a, b)$ and that $f(b, a) \geq 0$, so that the only way for the residual capacity to be zero is to have

- $f(a, b) = c(a, b)$
- $f(b, a) = 0$

Now just observe that the cost of the flow is

$$cost(f) = f(A) = \sum_{a \in A, b \notin A} f(a, b) - \sum_{b \notin A, a \in A} f(b, a) = \sum_{a \in A, b \notin A} c(a, b) = c(A)$$

and so the capacity of the cut is indeed equal to the cost of the flow. □

Remark 9.8 *Suppose that we had defined the residual network as a network in which the capacity of the edge (u, v) is $c(u, v) - f(u, v)$, without the extra "virtual capacity" coming from the flow from v to u, and suppose that we defined an augmenting path to be a path from s to t in which each capacity in the residual network (according to this definition) is positive. Then we have seen before an example of a flow that has no augmenting path according to this definition, but that is not optimal. Where does the proof of the Max-Flow Min-Cut theorem break down if we use the $c(u, v) - f(u, v)$ definition of residual capacity?*

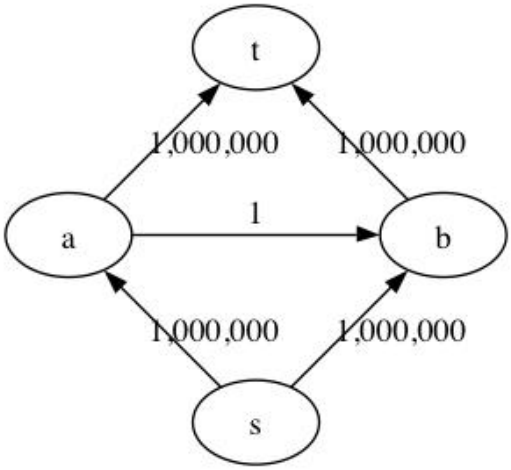
Lecture 10

The Fattest Path

In which we discuss the worst-case running of the Ford-Fulkerson algorithm, discuss plausible heuristics to choose an augmenting path in a good way, and begin analyzing the "fattest path" heuristic.

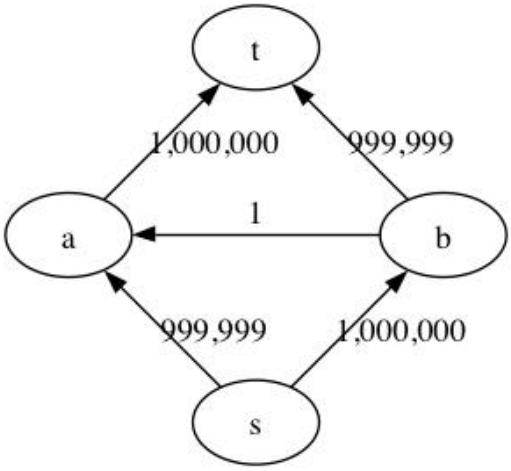
In the last lecture we proved the Max-Flow Min-Cut theorem in a way that also established the optimality of the Ford-Fulkerson algorithm: if we iteratively find an augmenting path in the residual network and push more flow along that path, as allowed by the capacity constraints, we will eventually find a flow for which no augmenting path exists, and we proved that such a flow must be optimal.

Each iteration of the algorithm takes linear time in the size of the network: the augmenting path can be found via a DFS of the residual network, for example. The problem is that, in certain cases, the algorithm might take a very long time to finish. Consider, for example, the following network.



Suppose that, at the first step, we pick the augmenting path $s \to a \to b \to t$. We can

only push one unit of flow along that path. After this first step, our residual network (not showing edges out of t and into s, which are never used in an augmenting path) is



Now it is possible that the algorithm picks the augmenting path $s \to b \to a \to t$ along which, again, only one unit of flow can be routed. We see that, indeed, it is possible for the algorithm to keep picking augmenting paths that involve a link between a and b, so that only one extra unit of flow is routed at each step.

The problem of very slow convergence times as in the above example can be avoided if, at each iteration, we choose more carefully which augmenting path to use. One reasonable heuristic is that it makes sense to pick the augmenting path along which the most flow can be routed in one step. If we had used such an heuristic in the above example, we would have found the optimum in two steps. Another, alternative, heuristic is to pick the shortest augmenting path, that is, the augmenting path that uses the fewest edges; this is reasonable because in this way we are going to use the capacity of fewer edges and keep more residual capacity for later iterations. The use of this heuristic would have also resulted in a two-iterations running time in the above example.

10.1 The "fattest" augmenting path heuristic

We begin by studying the first heuristic: that is we consider an implementation of the Ford-Fulkerson algorithm in which, at every iteration, we pick a *fattest* augmenting path in the residual network, where the fatness of a path in a capacitated network is the minimum capacity of the edges in the path. In the network of our previous example, the paths $s \to a \to t$ and $s \to b \to t$ have fatness $1,000,000$, while the path $s \to a \to b \to t$ has fatness 1.

How do we find a fattest augmenting path? We will show that it can be found with a simple modification of Dijkstra's algorithm for finding shortest paths.

10.1.1 Dijkstra's algorithm

Let us first quickly recall how Dijkstra's algorithm works. Suppose that we have a graph in which each edge (u, v) has a length $\ell(u, v)$ and, for two given vertices s, t, we want to find the path of minimal length from s to t, where the length of a path is the sum of the lengths of the edges in the path. The algorithm will solve, for free, the more general problem of computing the length of the shortest path from s to v for every vertex v. In the algorithm, the data structure that holds information about a vertex v has two fields: $v.dist$, which will eventually contain the length of the shortest path from s to v, and $v.pred$ which will contain the *predecessor* of v in a shortest path from s to v, that is, the identity of the vertex that comes immediately before v in such a path.

The distances are initialized to $+\infty$, except for $s.dist$ which is initialized to zero. The algorithm initially puts all vertices in a *priority queue* Q. Recall that a priority queue is a data structure that contains elements which have a numerical field called a *key* (in this case the key is the *dist* field), and that supports the operation of inserting an element in the queue, of finding and removing from the queue the element of minimal key value, and of reducing the key field of a given element.

The algorithm works as follows:

Algorithm *Dijkstra*

- Input: graph $G = (V, E)$, vertex $s \in V$, non-negative edge lengths $\ell(\cdot, \cdot)$
- for each $v \in V - \{s\}$, let $v.dist = \infty$
- $s.dist = 0$
- insert all vertices in a priority queue Q keyed by the *dist* field
- while Q is not empty
  - find and remove vertex u in Q whose field $u.dist$ is smallest among queue elements
  - for all vertices v such that $(u, v) \in E$
    - if $v.dist > u.dist + \ell(u, v)$ then
      - $v.dist := u.dist + \ell(u, v)$
      - update Q to reflect changed value of $v.dist$
      - $u.pred := v$

The running time is equal to whatever time it takes to execute $|V|$ *insert* operations, $|V|$ *remove-min* operations, and $|E|$ *reduce-key* operations in the priority queue. The simple implementation of a priority queue via a binary heap gives $O(\log |V|)$ running time for each operation, and a total running time of $O((|E| + |V|) \cdot \log |V|)$. A more elaborate data structure called a *Fibonacci heap* implements *insert* and *remove-min* in $O(\log |V|)$ time, and is such that k *decrease-key* operations always take at most $O(k)$ time overall, so that the total running time is $O(|V| \log |V| + |E|)$.

Regarding correctness, we can prove by induction that the algorithm maintains the following

invariant: at the beginning of each iteration of the *while* loop, the vertices x which are *not* in the queue are such that $x.dist$ is the correct value of the shortest path length from s to x and such a shortest path can be realized by combining a shortest path from s to $x.pred$ and then continuing with the edge $(x.pred, x)$. This is certainly true at the beginning, because the first vertex to be removed is s, which is at distance $s.dist = 0$ from itself, and if it is true at the end, when the queue is empty, it means that at the end of the algorithm all vertices get their correct values of $x.dist$ and $x.pred$. So we need to show that if the invariant is true at a certain step then it is true at the following step.

Basically, all we need to prove is that, at the beginning of each iteration, the vertex u that we remove from the queue has correct values of $u.dist$ and $u.pred$. If we call $x := u.pred$, then x is a vertex that was removed from the queue at an earlier iteration and so, by the inductive hypothesis, is such that $x.dist$ is the correct shortest path distance from s to x; if $x = u.pred$ we also have $u.dist = x.dist + \ell(u, v)$, which means that there is indeed a path of length $u.dist$ from s to u in which x is the predecessor of u. We need to prove that this path is a shortest path. So suppose toward a contradiction that there is a shorter path p of length $< u.dist$. The path p starts at s, which is outside the queue, and ends at u, which is in the queue, so at some point the path must have an edge (y, z) such that y is outside the queue and z is inside. This also means that when y was removed from the queue it had the correct value $y.dist$, and after we processed the neighbors of y we had $z.dist \leq y.dist + \ell(u, v)$. But this would mean that $z.dist$ is at most the length of the path p, while $u.dist$ is bigger than the length of the path p, which is impossible because u was chosen to be the element with the smallest $u.dist$ among elements of the queue.

10.1.2 Adaptation to find a fattest path

What would be the most straightforward adaptation of Dijkstra's algorithm to the problem of finding a fattest path? In the shortest path problem, the length of a path is the *sum* of the *lengths* of the edges of the path, and we want to find a path of *minimal* length; in the fattest path problem, the fatness of a path is the *minimum* of the *capacities* of the edges of the path, and we want to find a path of *maximal* fatness. So we just change sums to min, lengths to capacities, and minimization to maximization.

Algorithm *Dijkstra-F*

- Input: graph $G = (V, E)$, vertex $s \in V$, non-negative edge capacities $c(\cdot, \cdot)$
- for each $v \in V - \{s\}$, let $v.fat = 0$
- $s.dist = \infty$
- insert all vertices in a priority queue Q keyed by the *dist* field
- while Q is not empty
  - find and remove vertex u in Q whose field $u.fat$ is largest among queue elements
  - for all vertices v such that $(u, v) \in E$
    - if $v.fat < \min\{u.fat, c(u, v)\}$ then
      - $v.fat := \min\{u.fat, c(u, v)\}$
      - update Q to reflect changed value of $v.dist$
      - $u.pred := v$

The running time is the same and, quite amazingly, the proof of correctness is also essentially the same. (Try writing it up.)

Remark 10.1 *A useful feature of Dijkstra's algorithm (and other shortest path algorithms) is that it works to find "best" paths for a lot of different measures of "cost" for a path, besides length and fatness. Basically, the only requirements to implement the algorithm and prove correctness are:*

- *The cost of a path $u_1 \to u_2 \to \cdots u_t$ is no better than the cost of an initial segment $u_1 \to u_2 \to \cdots u_k$, $k < t$ of the path. That is, if we are trying to maximize the cost, we need the property that the cost of a path is at most the cost of any initial segment (e.g., the fatness of a path is at most the fatness of any initial segment, because in the former case we are taking the minimum over a larger set of capacities); if we are trying to minimize the cost, we need the property that the cost of a path is at least the cost of any initial segment.*
- *The cost of a path $u_1 \to u_2 \to \cdots \to u_{t-1} \to u_t$ can be determined by only knowing the cost of the path $u_1 \to u_2 \to \cdots \to u_{t-1}$ and the cost of the edge (u_{t-1}, u_t).*

10.1.3 Analysis of the fattest augmenting path heuristic

In the next lecture, we will prove the following result.

Theorem 10.2 *If $(G = (V, E), s, t, c)$ is a network in which the optimal flow has cost opt, then there is a path from s to t of fatness $\geq opt/m$.*

From the above theorem, we se that if we implement the Ford-Fulkerson algorithm with the fattest-path heuristic, then, after we have found t augmenting paths, we have a solution such that, in the residual network, the optimum flow has cost at most $opt \cdot \left(1 - \frac{1}{2|E|}\right)^t$.

To see why, call $flow_i$ the cost of the flow found by the algorithm after i iterations, and res_i the optimum of the residual network after i iterations of the algorithm. Clearly we have $res_i = opt - flow_i$.

The theorem tells us that at iteration $i+1$ we are going to find an augmenting path of fatness at least $res_i \cdot \frac{1}{2|E|}$. (Because of the "virtual capacities," the residual network could have as many as twice the number of edges of the original network, but no more.) This means that the cost of the flow at the end of the $(i+1)$-th iteration is going to be $flow_{i+1} \geq flow_i + res_i \cdot \frac{1}{2|E|}$, which means that the residual optimum is going to be

$$res_{i+1} = opt - flow_{i+1} \leq opt - \left(flow_i - res_i \cdot \frac{1}{2|E|}\right) = res_i \cdot \left(1 - \frac{1}{2|E|}\right)$$

We started with $flow_0 = 0$ and $res_0 = opt$, and so we must have $res_t \leq opt \cdot \left(1 - \frac{1}{2|E|}\right)^t$.

If the capacities are integers, then if the residual network has an optimum less than 1, its optimum must be zero. Recalling that $1 - x \leq e^{-x}$,

$$res_t \leq opt \left(1 - \frac{1}{2|E|}\right)^t \leq opte^{-t/2|E|} = e^{\ln opt - t/2|E|}$$

This means that if $t > 2|E| \ln opt$, then $res_t < 1$, which implies $res_t = 0$ and so it means that, within the first $1 + 2|E| \ln opt$ steps, the algorithm reaches a point in which the residual network has no augmenting path and it stops.

We said that, using the simple binary heap implementation of Dijkstra's algorithm, the running time of one iteration is $O((|V| + |E|) \cdot \log |V|)$, and so we have the following analysis.

Theorem 10.3 *The fattest-path implementation of the Ford-Fulkerson algorithm, given in input a network with integer capacities whose optimal flow has cost opt, runs in time at most*

$$O((|V| + |E|) \cdot |E| \cdot \log |V| \cdot \log opt)$$

To complete the above running time analysis, we need to prove Theorem 10.2, which we will do in the next lecture.

Lecture 11

Strongly Polynomial Time Algorithms

In which we prove that the Edmonds-Karp algorithm for maximum flow is a strongly polynomial time algorithm.

11.1 Flow Decomposition

In the last lecture, we proved that the Ford-Fulkerson algorithm runs in time

$$O(|E|^2 \log |E| \log opt)$$

if the capacities are integers and if we pick, at each step, the fattest path from s to t in the residual network. In the analysis, we skipped the proof of the following result.

Theorem 11.1 *If $(G = (V, E), s, t, c)$ is a network in which the cost of the maximum flow is opt, then there is a path from s to t in which every edge has capacity at least $opt/|E|$.*

We derive the theorem from the following result.

Lemma 11.2 (Flow Decomposition) *Let $(G = (V, E), s, t, c)$ be a network, and f be a flow in the network. Then there is a collection of feasible flows $f_1, \ldots, f_k$ and a collection of $s \to t$ paths $p_1, \ldots, p_k$ such that:*

1. *$k \leq |E|$;*
2. *the cost of f is equal to the sum of the costs of the flows f_i*
3. *the flow f_i sends positive flow only on the edges of p_i.*

Now we show how to prove Theorem 11.1 assuming that Lemma 11.2 is true.

We apply Lemma 11.2 to the maximum flow f of cost opt, and we find flows $f_1, \ldots, f_k$ and paths $p_1, \ldots, p_k$ as in the Lemma. From the first two properties, we get that there is an i_0 such that the cost of the flow f_{i_0} is at least $opt/|E|$. From the third property, we have that the $\geq opt/|E|$ units of flow of f_{i_0} are carried using only the path p_{i_0}, and so every edge of p_{i_0} must have capacity at least $opt/|E|$.

It remains to prove the Lemma.

PROOF: (Of Lemma 11.2) Now we see how to construct flows with the above three properties. We do so via the following procedure:

- $i := 1$
- $r := f$
- while $cost(r) > 0$
  - find a path from s to t using only edges (u, v) such that $r(u, v) > 0$, and call it p_i
  - let $f_{\min}$ be the minimum of $r(u, v)$ among the edges $(u, v) \in p_i$
  - let $f_i(u, v) := f_{\min}$ for each $(u, v) \in p_i$ and $f_i(u, v) := 0$ for the other edges
  - let $r(u, v) := r(u, v) - f_i(u, v)$ for each (u, v)
  - $i := i + 1$

The "residual" flow r is initialized to be equal to f, and so its cost is the same as the cost of f. At every step i, if the cost of r is still positive, we find a path p_i from s to t entirely made of edges with positive flow.

(Note that such a path must exist, because, if not, call A the set of nodes reachable from s using edges (u, v) that have $r(u, v) > 0$; then A contains s and it does not contain t, and so it is a cut and the net flow out of A is equal to cost of r; but there is no positive net flow out of A, because all the edges from vertices of A to vertices not in A must have $r(u, v) = 0$; this means that the cost of r must also be zero, which is a contradiction.)

We define the flow f_i by sending along p_i the smallest of the amounts of flow sent by r along the edges of p_i. Note that f_i is feasible, because for every edge we have $f_i(u, v) \leq r(u, v)$ and, by construction, we also have $r(u, v) \leq f(u, v)$, and f was a feasible flow and so $f(u, v) \leq c(u, v)$. We then decrease $r(u, v)$ by $f_i(u, v)$ on each edge. This is still a feasible flow, because we leave a non-negative flow on each edge and we can verify that we also maintain the conservation constraints. After the update, the cost of r decreases precisely by the same amount as the cost of f_i, so we maintain the invariant that, after i steps, we have

$$cost(f) = cost(r) + cost(f_1) + \cdots + cost(f_i)$$

It remains to observe that, after the update of r, at least one of the edges that had positive $r(u, v) > 0$ has now $r(u, v) = 0$. (This happens to the edge, or edges, that carry the

minimum amount of flow along p_i.) This means that, after i steps, the number of edges (u, v) such that $r(u, v) > 0$ is at most $|E| - i$ and that, in particular, the algorithm halts within at most $|E|$ iterations.

Call k the number of iterations after which the algorithm halts. When the algorithm halts, $cost(r) = 0$, and so we have

$$cost(f) = cost(f_1) + \cdots cost(f_k)$$

and so the flows and paths found by the algorithm satisfy all the requirements stated at the beginning. □

The running time $O(|E|^2 \log |E| \log opt)$ is not terrible, especially considering that it is a worst-case estimate and that often one has considerably faster convergence in practice. There is, however, an undesirable feature in our analysis: the running time depends on the actual values of the numbers that we get as input. An algorithm for a numerical problem is called *strongly polynomial* if, assuming unit-time operations on numerical quantities, the running time is at most a polynomial in the number of numerical quantities that we are given as input. In particular, a maximum flow algorithm is strongly polynomial if it runs in time polynomial in the number of edges in the network.

Today we describe the Edmonds-Karp algorithm, which is a simple variant of the Ford-Fulkerson algorithm (the variant is that, in each iteration, the $s \to t$ path in the residual network is found using a BFS). The Edmonds-Karp algorithm runs in strongly polynomial time $O(|V| \cdot |E|^2)$ in a simple implementation, and the worst-case running time can be improved to $O(|V|^2 \cdot |E|)$ with some adjustments.

We then begin to talk about an approach to solving the maximum flow problem which is rather different from the Fulkerson-Ford approach, and which is based on the "push-relabel" method. A simple implementation of the push-relabel method has running time $O(|V|^2 \cdot |E|)$, and a more sophisticated implementation has worst-case running time $O(|V|^3)$. We will only present the simpler algorithm.

11.2 The Edmonds-Karp Algorithm

The Edmonds-Karp algorithm is an implementation of the Ford-Fulkerson algorithm in which, at every step, we look for an $s \to t$ path in the residual network using BFS. This means that, if there are several possible such paths, we pick one with a minimal number of edges.

From now on, when we refer to a "shortest path" in a network, we mean a path that uses the fewest edges, and the "length" of a path is the number of edges. The "distance" of a vertex from s is the length of the shortest path from s to the vertex.

BFS can be implemented in $O(|V + E|) = O(|E|)$ time, and so to complete our analysis of the algorithm we need to find an upper bound to the number of possible iterations.

The following theorem says that, through the various iterations, the length of the shortest

path from s to t in the residual network can only increase, and it does increase at the rate of at least one extra edge in the shortest path for each $|E|$ iterations.

Theorem 11.3 *If, at a certain iteration, the length of a shortest path from s to t in the residual network is ℓ, then at every subsequent iteration it is $\geq \ell$. Furthermore, after at most $|E|$ iterations, the distance becomes $\geq \ell + 1$.*

Clearly, as long as there is a path from s to t, the distance from s to t is at most $|V| - 1$, and so Theorem 11.3 tells us that, after at most $|E| \cdot (|V| - 1)$ iterations, s and t must become disconnected in the residual network, at which point the algorithm terminates. Each iteration takes time $O(|E|)$, and so the total running time is $O(|V| \cdot |E|^2)$.

Let us now prove Theorem 11.3.

PROOF: Suppose that, after some number T of iterations, we have the residual network $R = (V, E'), s, t, c')$ and that, in the residual network, the length of the shortest path from s to t is ℓ. Construct a BFS tree starting at s, and call V_1, V_2, ..., V_k, ..., the vertices in the first, second, k-th, ..., layer of the tree, that is, the vertices whose distance from s is 1, 2, ..., and so on. Note that every edge (u, v) in the network is such that if $u \in V_i$ and $v \in V_j$ then $j \leq i + 1$, that is, nodes can go from higher-numbered layer to lower numbered layer, or stay in the same layer, or advance by at most one layer.

Let us call an edge (u, v) a *forward* edge if, for some i, $u \in V_i$ and $v \in V_{i+1}$. Then a shortest path from s to t must use a forward edge at each step and, equivalently, a path that uses a non-forward edge at some point cannot be a shortest path from s to t.

What happens at the next iteration $T + 1$? We pick one of the length-ℓ paths p from s to t and we push flow through it. In the next residual network, at least one of the edges in p disappears, because it has been saturated, and for each edge of p we see edges going in the opposite direction. Now it is still true that for every edge (u, v) of the residual network at the next step $T + 1$, if $u \in V_i$ and $v \in V_j$, then $j \leq i + 1$ (where $V_1, \ldots$ are the layers of the BFS tree of the network at iteration T), because all the edges that we have added actually go from higher-numbered layers to lower-numbered ones. This means that, at iteration $T + 1$ the distance of t from s is still at least ℓ, because $t \in V_\ell$ and, at every step on a path, we can advance at most by one layer.

(Note: we have proved that if the distance of t from s is ℓ at one iteration, then it is at least ℓ at the next iteration. By induction, this is enough to say that if will always be at least ℓ in all subsequent iterations.)

Furthermore, if there is a length-ℓ path from s to t in the residual network at iteration $T+1$, then the path must be using only edges which were already present in the residual network at iteration T and which were "forward edges" at iteration T. This also means that, in all the subsequent iterations in which the distance from s to t remains ℓ, it is so because there is a length-ℓ path made entirely of edges that were forward edges at iteration T. At each iteration, however, at least one of those edges is saturated and is absent from the residual network in subsequent steps, and so there can be at most $|E|$ iterations during which the distance from s to t stays ℓ. □

Lecture 12

The Push-Relabel Algorithm

In which we introduce the Push-Relabel algorithm, and prove that a basic implementation of this algorithm runs in time $O(|V|^2 \cdot |E|)$.

12.1 The Push-Relabel Approach

All maximum flow algorithms are based on the maximum flow – minimum cut theorem, which says that if there is no $s \to t$ path in the residual network then the flow is optimal. Our goal is thus to "simply" find a flow such that t is unreachable from s in the residual network.

In algorithms based on the Ford-Fulkerson approach, we keep at every step a feasible flow, and we stop when we reach a step in which there is no $s \to t$ path in the residual network.

In algorithms based on the *push-relabel* method, we take a somewhat complementary approach: at every step we have an assignment of flows to edges which is not a feasible flow (it violates the conservation constraints), which is called a *preflow*, but for which we can still define the notion of a residual network. The algorithm maintains the condition that, at every step, t is not reachable from s in the residual network. The algorithm stops when the preflow becomes a feasible flow.

The basic outline of the algorithm is the following: we begin by sending as much flow out of s as allowed by the capacities of the edges coming out of s, without worrying whether all that flow can actually reach t. Then, at each iteration, we consider nodes that have more incoming flow than outgoing flow (initially, the neighbors of s), and we route the excess flow to their neighbors, and so on. The idea is that if we attempted to send too much flow out of s in the first step, then the excess will eventually come back to s, while t will receive the maximum possible flow. To make such an idea work, we need to make sure that we do not keep sending the flow in circles, and that there is a sensible measure of "progress" that we can use to bound the running time of the algorithm.

A main idea in the algorithm is to associate to each vertex v a *height* $h[v]$, with the intuition that the flow wants to go downhill, and we will take the action of sending extra flow from

a vertex u to a vertex v only if $h[u] > h[v]$. This will help us avoid pushing flow around in circles, and it will help us define a measure of "progress" to bound the running time.

Here is the outline of the algorithm. Given an assignment of flows $f(u, v)$ to each edge (u, v), and a vertex v, the *excess flow* at v is the quantity

$$e_f(v) := \sum_u f(u, v) - \sum_w f(v, w)$$

that is, the difference between the flow getting into v and the flow getting out of v. If f is a feasible flow, then the excess flow is always zero, except at s and t.

- Input: network $(G = (V, E), s, t, c)$
- $h[s] := |V|$
- for each $v \in V - \{s\}$ do $h[v] := 0$
- for each $(s, v) \in E$ do $f(s, v) := c(s, v)$
- while f is not a feasible flow
  - let $c'(u, v) = c(u, v) + f(u, v) - f(v, u)$ be the capacities of the residual network
  - if there is a vertex $v \in V - \{s, t\}$ and a vertex $w \in V$ such that $e_f(v) > 0$, $h(v) > h(w)$, and $c'(v, w) > 0$ then
    - push $\min\{c'(v, w), e_f(v)\}$ units of flow on the edge (v, w)
  - else, let v be a vertex such that $e_f(v) > 0$, and set $h[v] := h[v] + 1$
- output f

As we said, the algorithm begins by pushing as much flow to the neighbors of s as allowed by the capacities of the edges coming out of s. This means that we get some vertices with positive excess flow, and some vertices with zero excess flow. Also, we do not violate the capacity constraints. These properties define the notion of a *preflow*.

Definition 12.1 (Preflow) *An assignment of a non-negative flow $f(u, v)$ to each edge (u, v) of a network $(G = (V, E), s, t, c)$ is a preflow if*

- *for each edge (u, v), $f(u, v) \leq c(u, v)$*
- *for each vertex $v \in V - \{t\}$,*

$$\sum_u f(u, v) - \sum_w f(v, w) \geq 0$$

We will always assume that, for every pair of vertices u, v, at most one of $f(u, v)$ and $f(v, u)$ is positive.

A preflow in which all excess flows $e_f(v)$ are zero for each $v \in V - \{s, t\}$ is a feasible flow.

The definition of *residual network* for a preflow is the same as for a flow; the capacity of an edge (u, v) in the residual network is

$$c(u, v) - f(u, v) + f(v, u)$$

If the edge (u, v) has capacity $\geq r$ in the residual network corresponding to a preflow f, and the vertex u has excess flow $\geq r$, then we can send r units of flow from u to v (by increasing $f(u, v)$ and/or reducing $f(v, u)$) and create another preflow. In the new preflow, the excess of u is r units less than before, and the excess flow of v is r units more than before.

Such a "shifting" of excess flow from one node to another is the basic operation of a push-relabel algorithm, and it is called a *push* operation. If we push an amount of flow along an edge equal to its capacity in the residual network, then we call it a *saturating push*, otherwise we call it a *nonsaturating push*. We execute a push operation provided that we find a pair of vertices such that we can push from a "higher" vertex to a "lower" vertex, according to the height function $h[\cdot]$.

If the above operation is not possible, we take a vertex with excess flow, and we increase its height. This operation is called a *relabel* operation.

The reader should try running this algorithm by hand on a few examples to get a sense of how it works.

12.2 Analysis of the Push-Relabel Algorithm

We begin by showing that no vertex can reach a height bigger than $2 \cdot |V| - 1$. This automatically puts an upper bound to the number of *relabel* operations that can be executed, and is an important starting point in analyzing the number of push operations.

Lemma 12.2 *At every step, if a vertex v has positive excess flow, then there is a path from v to s in the residual network.*

Proof: First, let us why this is "obvious:" in a preflow, vertices are allowed to "destroy" stuff, but not to "create" it, so if a vertex has positive excess flow, then in particular it has positive incoming flow, and the incoming stuff must be coming from s along a path made of edges with positive flow. To each such edge corresponds an edge in the opposite direction with positive capacity in the residual network, and so v is connected to s in the residual network.

The only part of the above argument that is not rigorous is when we say that if a vertex v has positive excess flow then there must be a path from s to v made entirely of edges with positive flow. Let A be the set of vertices which are reachable from s via such a path. Because of the preflow constraints on the vertices $v \notin A$, we have

$$\sum_{v \notin A} e_f(v) = \sum_{v \notin A} \left(\sum_u f(u, v) - \sum_w f(v, w) \right) \geq 0$$

but, in the second expression, all terms of the form $f(x, y)$ in which $x \notin A$ and $y \notin A$ cancel, because they appear once with a plus sign and once with a minus sign. The result of such cancellations is

$$\sum_{v \notin A} e_f(v) = \sum_{u \in A, v \notin A} f(u, v) - \sum_{v \notin A, w \in A} f(v, w) \leq 0$$

where the last inequality follows from the fact that $f(u, v)$ must be zero when $u \in A$ and $v \notin A$.

So we have that $e_f(v) = 0$ for every $v \notin A$, which means that every vertex v that has $e_f(v) > 0$ must be in A, and so it must be reachable from s via a path made of edges with positive flow.□

The connection between the previous lemma and the task of bounding the heights of vertices comes from the following observation.

Lemma 12.3 *At every step, if there is an edge (u, v) that has positive capacity $c'(u, v) > 0$ in the residual network, then*

$$h(u) \leq h(v) + 1$$

Before proving the lemma, let us understand what it is getting at. Our intuition for the heights, is that we want the flow to go "downhill," and in fact every time we do a push operation we do so from a higher vertex to a lower one. If the flow goes downhill, the edges with positive residual capacity go "uphill." This is not exactly true at each step, because of the relabeling operations, but the lemma is saying that edges with positive residual capacity cannot go downhill by more than one unit.

PROOF: We show that the property is an invariant preserved by the algorithm at each step.

At the beginning, the residual network contains: (i) the edges of the original networks between vertices other than s, and all such vertices have the same height 0, and (ii) edges from the neighbors of s to s, and such edges go uphill.

Now we show that the property is preserved at each step.

If we do a relabel step on a vertex v, the property remains true for all the edges (u, v) with positive capacity coming into v. About the edges (v, w) with positive capacity coming out of v, if we did a relabel step it was because we had $h(v) \leq h(w)$; after the relabel, we still have $h(v) \leq h(w) + 1$.

If we do a push step along an edge (u, v), we might introduce the reverse edge (v, u) in the residual network. The push step, however, happens only when $h(u) > h(v)$, and so the edge (v, u) satisfies the property. □

Fact 12.4 *At every step, if there is a path from u to v in the residual network, then*

$$h(u) \leq h(v) + |V| - 1$$

PROOF: If there is a path of length ℓ from u to v in the residual network, then, by applying ℓ times Lemma 12.2, we have $h(u) \leq h(v) + \ell$, and if there is a path from u to v there must be a path of length at most $|V| - 1$. □

We can now being to draw conclusions that are relevant to our analysis.

Fact 12.5 *At each step of the algorithm, there is no path from s to t in the residual network.*

Because, if there were such a path, we would have $h(s) \leq h(t) + |V| - 1$, but at the beginning we have $h(s) = |V|$ and $h(t) = 0$, and the heights of s and t never change.

This means that *if the algorithm terminates, then it outputs an optimal flow.* From now on, it remains to estimate the running time of the algorithm, which we do by finding upper bounds to the number of times the various operations can be executed.

Fact 12.6 *At each step of the algorithm, every vertex has height at most $2|V| - 1$.*

PROOF: Each time the height of a vertex v is increased, it is because it has positive excess flow. If a vertex v has positive excess flow, then there is a path from v to s in the residual network. If there is such a path, then $h(v) \leq h(s) + |V| - 1 \leq 2|V| - 1$. □

Fact 12.7 *The algorithm executes at most $(|V| - 2) \cdot (2 \cdot |V| - 1) \leq 2|V|^2$ relabel operations.*

PROOF: There are at most $|V| - 2$ vertices on which the relabel operation is admissible, and on each of them the algorithm executes the operation at most $2 \cdot |V| - 1$ times. □

We now estimate the number of *push* operations.

We call a push operation *saturating* if it uses the entire residual capacity of edge, making it disappear from the residual network. Otherwise, the push operation is *nonsaturating.*

Fact 12.8 *The algorithm executes at most $2|V| \cdot |E|$ saturating push operations.*

PROOF: Consider an edge (u, v). The first time there is a saturating push from u to v, it is because $h(u) > h(v)$. After the saturating push, the edge (u, v) disappears from the residual network, and so there cannot be any other saturating push from u to v (and, indeed, no push of any kind), until v sends back some flow to u with a push in the opposite direction. But for this to happen we must first have $h(v) > h(u)$, which requires at least two relabels of v. For the next saturating push from u to v, we must have again $h(u) > h(v)$, which requires two more relabels, at least. So, between two saturating pushes from u to v, at least four relabels must take place on u and v. Overall, u and v can relabeled at most $4|V|$ times, and so there can be at most $|V|$ saturating pushes.

There are $2 \cdot |E|$ edges that can appear in the residual network, and so in total we have at most $2 \cdot |V| \cdot |E|$ saturating pushes. □

The most interesting part of the analysis is how we analyze the number of non-saturating push operations.

At each step of the execution of the algorithm, we define the "energy" of the current preflow f as

$$\Phi(f) := \sum_{v:e_f(v)>0} h(v)$$

the sum of the heights of all vertices that have positive excess flow. The algorithm starts in a zero-energy state, but the energy becomes one after the first relabel operation. When the energy becomes zero again, it is because there are no nodes with excess flow, and the algorithm stops.

We have the following observations.

Fact 12.9 *Each relabel step increases the energy by exactly one unit.*

Fact 12.10 *Each saturating push increases the energy by at most* $2|V|$ *units.*

PROOF: A push step along an edge (u, v) does not change the height of any vertex, but it could give excess flow to vertex v, which possibly had zero excess flow before, so that the energy increases by $h(v) \leq 2|V|$ units. □

Fact 12.11 *Each nonsaturating push decreases the energy by at least one unit.*

PROOF: If we do a push on an edge (u, v), why would the push be nonsaturating? The only reason why we would not saturate the edge is that the excess flow of u is less than the residual capacity of (u, v), and so we can push the entire excess flow of u along (u, v) with residual capacity to spare. But this means that, after a nonsaturing push along (u, v), the excess flow of u becomes zero, and so $h(u)$ is not counted in the energy any more. It is possible that v had no excess flow before the push and now it does, which means that we need to add $h(v)$ in the energy, but we still have that the new energy is at most the old energy minus $h(u)$ plus $h(v)$ and, recalling that we do a push only if $h(v) < h(u)$, we have that the new energy is at most the old energy minus one. □

Fact 12.12 *The total number of nonsaturating pushes is at most* $2|V|^2 + 4|V|^2|E|$.

PROOF: If, at some step of the execution of the algorithm, the preflow is not yet a feasible flow, then the energy must by > 0. If, up to that point, the algorithm has executed r relabel operations, sp saturating push operations, and np nonsaturating push operations, then

$$0 < \Phi(f) \leq r + 2|V|sp - np$$

we now that $r \leq 2|V|^2$ and $sp \leq 2|V| \cdot |E|$, so the above expression implies

$$np < 2|V|^2 + 4|V|^2|E|$$

So if, at some point of the execution of the algorithm, we haven't reached the termination condition yet, this implies that we have executed fewer than $2|V|^2 + 4|V|^2|E|$ nonsaturating pushes.

Equivalently, when the algorithm terminates, it has executed at most $2|V|^2 + 4|V|^2|E|$ nonsaturating pushes. □

Overall, we have a total of at most $O(|V|^2 \cdot |E|)$ operations, and each can be implemented in $O(1)$ time, so the running time of the algorithm is $O(|V|^2 \cdot |E|)$.

12.3 Improved Running Time

We note that the algorithm is somewhat underspecified, in the sense that there could be more than one vertex out of which a push operation is allowed, and we are not saying how to pick one; if no push operation is possible, any of the vertices with positive excess can be picked for a relabel operation, and we are not specifying how to pick one. The analysis of the running time applies to any possible way to make such choices.

A reasonable heuristic is that if we have the choice of multiple vertices out of which to push flow, then we choose the vertex of biggest height. It can be proved (but we will not), this implementation of the algorithm executes at most $O(|V|^2\sqrt{|E|})$ nonsaturating pushes, and so the running time is $O(|V|^2\sqrt{|E|})$. A more complicated implementation, in which multiple pushes are done together, and the *dynamic tree* data structure is used to keep information about the current preflow, has running time $O(|V| \cdot |E| \cdot \log |V|)$. In terms of worst-case running time, this is the best known for strongly polynomial algorithms.

An algorithm of Goldberg and Rao has running time

$$O((\min\{|E| \cdot |V|^{2/3}\ ,\ |E|^{1.5}) \cdot (\log |V| \cdot \log opt))$$

In the interesting case in which $|E| = O(|V|)$, this is roughly $|V|^{1.5}$, compared the to the $|V|^2$ running time of the optimized push-relabel algorithm. This year, a new algorithm has been discovered that, in undirected networks, finds a flow of cost $\geq (1 - \epsilon) \cdot opt$ in time $O(|E|^{4/3} \cdot \epsilon^{-4} \cdot (\log |V|)^{O(1)})$.

There has been extensive experimental analysis of maximum flow algorithms. The fastest algorithms in practice are carefully tuned push-relabel implementations.

Lecture 13

Edge Connectivity

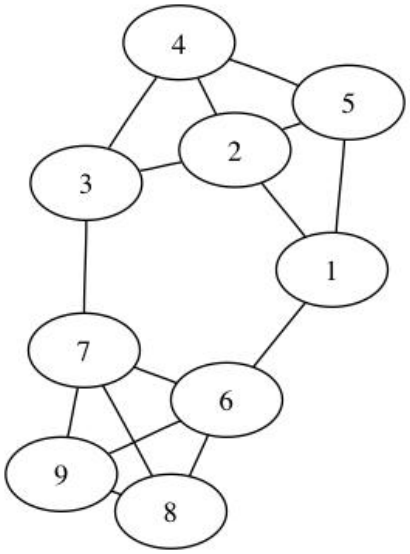
In which we describe a randomized algorithm for finding the minimum cut in an undirected graph.

13.1 Global Min-Cut and Edge-Connectivity

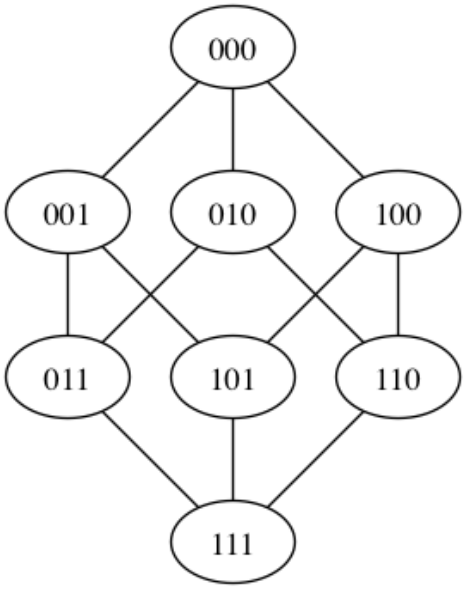
Definition 13.1 (Edge connectivity) *We say that an undirected graph is k-edge-connected if one needs to remove at least k edges in order to disconnect the graph. Equivalently, an undirected graph is k-edge-connected if the removal of any subset of $k-1$ edges leaves the graph connected.*

Note that the definition is given in such a way that if a graph is, for example 3-edge-connected, then it is also 2-edge-connected and 1-edge connected. Being 1-edge-connected is the same as being connected.

For example, the graph below is connected and 2-edge connected, but it is not 3-edge connected, because removing the two edges $(3, 7)$ and $(1, 6)$ disconnects the graph.



As another example, consider the 3-cube:



The 3-cube is clearly not 4-edge-connected, because we can disconnect any vertex by removing the 3 edges incident on it. It is clearly connected, and it is easy to see that it is 2-edge-connected; for example we can see that it has a Hamiltonian cycle (a simple cycle that goes through all vertices), and so the removal of any edge still leaves a path that goes trough every vertex. Indeed the 3-cube is 3-connected, but at this point it is not clear how to argue it without going through some complicated case analysis.

The *edge-connectivity* of a graph is the largest k for which the graph is k-edge-connected, that is, the minimum k such that it is possible to disconnect the graph by removing k edges.

In graphs that represent communication or transportation networks, the edge-connectivity is an important measure of reliability.

Definition 13.2 (Global Min-Cut) *The global min-cut problem is the following: given in input an undirected graph $G = (V, E)$, we want to find the subset $A \subseteq V$ such that $A \neq \emptyset$, $A \neq V$, and the number of edges with one endpoint in A and one endpoint in $V - A$ is minimized.*

We will refer to a subset $A \subseteq V$ such that $A \neq \emptyset$ and $A \neq V$ as a *cut* in the graph, and we will call the number of edges with one endpoint in A and one endpoint in $V - A$ the *cost* of the cut. We refer to the edges with one endpoint in A and one endpoint in $V - A$ as the edges that *cross* the cut.

We can see that the Global Min Cut problem and the edge-connectivity problems are in fact the same problem:

- if there is a cut A of cost k, then the graph becomes disconnected (in particular, no vertex in A is connected to any vertex in $V - A$) if we remove the k edges that cross the cut, and so the edge-connectivity is at most k. This means that the edge-connectivity of a graph is at most the cost of its minimum cut;

- if there is a set of k edges whose removal disconnects the graph, then let A be the set of vertices in one of the resulting connected components. Then A is a cut, and its cost is at most k. This means that the cost of the minimum cut is at most the edge-connectivity.

We will discuss two algorithms for finding the edge-connectivity of a graph. One is a simple reduction to the maximum flow problem, and runs in time $O(|E| \cdot |V|^2)$. The other is a surprising simple randomized algorithm based on *edge-contractions* – the surprising part is the fact that it correctly solves the problem, because it seems to hardly be doing any work. We will discuss a simple $O(|V|^3)$ implementation of the edge-contraction algorithm, which is already better than the reduction to maximum flow. A more refined analysis and implementation gives a running time $O(|V|^2 \cdot (\log |V|)^{O(1)})$.

13.1.1 Reduction to Maximum Flow

Consider the following algorithm:

- Input: undirected graph $G = (V, E)$
- let s be a vertex in V (the choice does not matter)
- define $c(u, v) = 1$ for every $(u, v) \in E$
- for each $t \in V - \{s\}$
  - solve the min cut problem in the network (G, s, t, c), and let A_t be the cut of minimum capacity
- output the cut A_t of minimum cost

The algorithm uses $|V| - 1$ minimum cut computations in networks, each of which can be solved by a maximum flow computation. Since each network can have a maximum flow of cost at most $|V| - 1$, and all capacities are integers, the Ford-Fulkerson algorithm finds each maximum flow in time $O(|E| \cdot opt) = O(|E| \cdot |V|)$ and so the overall running time is $O(|E| \cdot |V|^2)$.

To see that the algorithm finds the global min cut, let k be edge-connectivity of the graph, E^* be a set of k edges whose removal disconnects the graph, and let A^* be the connected component containing s in the disconnected graph resulting from the removal of the edges in E^*. So A^* is a global minimum cut of cost at most k (indeed, exactly k), and it contains s.

In at least one iteration, the algorithm constructs a network (G, s, t, c) in which $t \notin A^*$, which means that A^* is a valid cut, of capacity k, for the network, and so when the algorithm finds a minimum capacity cut in the network it must find a cut of capacity at most k (indeed, exactly k). This means that, for at least one t, the cut A_t is also an optimal global min-cut.

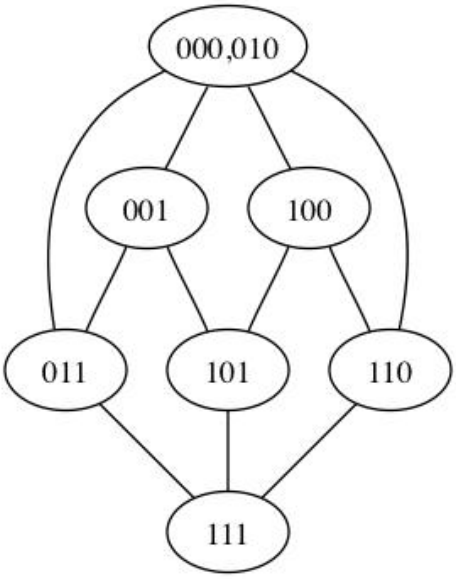
13.1.2 The Edge-Contraction Algorithm

Our next algorithm is due to David Karger, and it involves a rather surprising application of random choices.

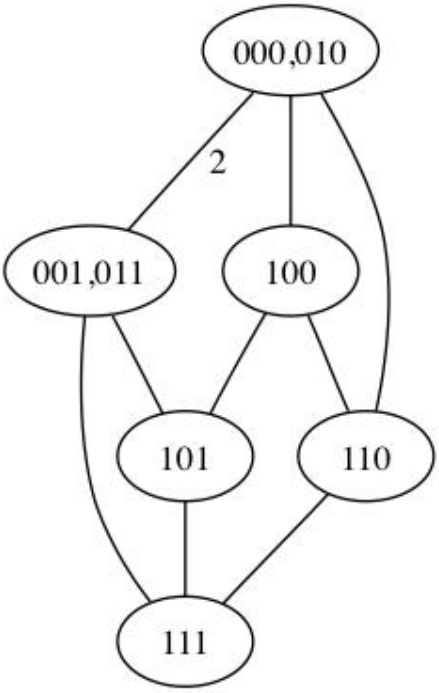
The algorithm uses the operation of *edge-contraction*, which is an operation defined over multi-graphs, that is graphs that can have multiple edges between a given pair of vertices or, equivalently, graphs whose edges have a positive integer weight.

If, in an undirected graph $G = (V, E)$ we *contract* an edge (u, v), the effect is that the edge (u, v) is deleted, and the vertices u and v are removed, and replaced by a new vertex, which we may call $[u, v]$; all other edges of the graph remain, and all the edges that were incident on u or v become incident on the new vertex $[u, v]$. If u had i edges connecting it to w, and v had j edges connecting it to w, then in the new graph there will be $i + j$ edges between w and $[u, v]$.

For example, if we contract the edge $(000, 010)$ in the 3-cube we have the following graph.



And if, in the resulting graph, we contract the edge $(001, 011)$, we have the following graph.



Note that, after the two contractions, we now have two edges between the "macro-vertices" $[000, 010]$ and $[001, 011]$.

The basic iteration of Karger's algorithm is the following:

- while there are ≥ 3 vertices in the graph

  - pick a random edge and contract it

- output the set A of vertices of the original graph that have been contracted into one of the two final macro-vertices.

One important point is that, in the randomized step, we sample uniformly at random among the edges of the multi-set of edges of the current *multi*-graph. So if there are 6 edges between the vertices (a, b) and 2 edges between the vertices (c, d), then a contraction of (a, b) is three times more likely than a contraction of (c, d).

The algorithm seems to pretty much pick a subset of the vertices at random. How can we hope to find an optimal cut with such a simple approach?

(In the analysis we will assume that the graph is connected. if the graph has two connected components, then the algorithm converges to the optimal min-cut of cost zero. If there are three or more connected components, the algorithm will discover them when it runs out of edges to sample, In the simplified pseudocode above we omitted the code to handle this exception.)

The first observation is that, if we fix for reference an optimal global min cut A^* of cost k, and if it so happens that there is never a step in which we contract one of the k edges that connect A^* with the rest of the graph then, at the last step, the two macro-vertices will indeed be A^* and $V - A^*$ and the algorithm will have correctly discovered an optimal solution.

But how likely is it that the k edges of the optimal solution are never chosen to be contracted at any iteration?

The key observation in the analysis is that if we are given in input a (multi-)graph whose edge-connectivity is k, then it must be the case that every vertex has degree $\geq k$, where the degree of a vertex in a graph or multigraph is the number of edges that have that vertex as an endpoint. This is because if we had a vertex of degree $\leq k - 1$ then we could disconnect the graph by removing all the edges incident on that vertex, and this would contradict the k-edge-connectivity of the graph.

But if every vertex has degree $\geq k$, then

$$|E| = \frac{1}{2} \sum_v \text{degree}(v) \geq \frac{k}{2} \cdot |V|$$

and, since each edge has probability $1/|E|$ of being sampled, the probability that, at the first step, we sample one of the k edges that cross the cut A^* is only

$$\frac{k}{|E|} \leq \frac{2}{|V|}$$

What about the second step, and the third step, and so on?

Suppose that we were lucky at the first step and that we did not select any of the k edges that cross A^*. Then, after the contraction of the first step we are left with a graph that

has $|V|-1$ vertices. The next observation is that this new graph *has still edge-connectivity* k because the cut defined by A^* is still well defined. If the edge-connectivity is still k, we can repeat the previous reasoning, and conclude that the probability that we select one of the k edges that cross A^* is at most

$$\frac{2}{|V|-1}$$

And now we see how to reason in general. If we did not select any of the k edges that cross A^* at any of the first step $t-1$ step, then the probability that we select one of those edges at step t is at most

$$\frac{2}{|V|-t+1}$$

So what is the probability that we never select any of those edges at any step, those ending up with the optimal solution A^*? If we write E_t to denote the event that "at step t, the algorithm samples an edge which does not cross A^*," then

$$\mathbb{P}[E_1 \wedge E_2 \wedge \cdots \wedge E_{n-2}]$$

$$\begin{aligned} &= \mathbb{P}[E_1] \cdot \mathbb{P}[E_2|E_1] \cdot \ldots \mathbb{P}[E_{n-2}|E_1 \wedge \cdots \wedge E_{n-3}] \\ &\geq \left(1-\frac{2}{|V|}\right) \cdot \left(1-\frac{2}{|V|-1}\right) \cdot \ldots \left(1-\frac{2}{3}\right) \end{aligned}$$

If we write $n := |V|$, the product in the last line is

$$\frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \cdot \ldots \frac{4}{6} \cdot \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3}$$

which simplifies to

$$\frac{2}{n \cdot (n-1)}$$

Now, suppose that we repeat the basic algorithm r times. Then the probability that it does not find a solution in any of the r attempts is at most

$$\left(1-\frac{2}{n \cdot (n-1)}\right)^r$$

So, for example, if we repeat the basic iteration $50n \cdot (n-1)$ times, then the probability that we do not find an optimal solution is at most

$$\left(1-\frac{2}{n \cdot (n-1)}\right)^{50n \cdot (n-1)} \leq e^{-100}$$

(where we used the fact that $1 - x \leq e^{-x}$), which is an extremely small probability.

One iteration of Karger's algorithm can be implemented in time $O(|V|)$, so overall we have an algorithm of running time $O(|V|^3)$ which has probability at least $1 - e^{-100}$ of finding an optimal solution.

Lecture 14

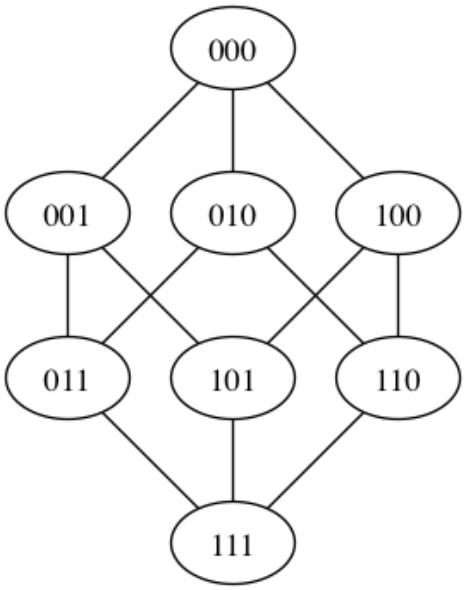
Algorithms in Bipartite Graphs

In which we show how to solve the maximum matching problem and the minimum vertex cover problem in bipartite graphs.

In this lecture we show applications of the theory of (and of algorithms for) the maximum flow problem to the design of algorithms for problems in bipartite graphs.

A bipartite graph is an undirected graph $G = (V, E)$ such that the set of vertices V can be partitioned into two subsets L and R such that every edge in E has one endpoint in L and one endpoint in R.

For example, the 3-cube is bipartite, as can be seen by putting in L all the vertices whose label has an even number of ones and in R all the vertices whose label has an odd number of ones.



There is a simple linear time algorithm that checks if a graph is bipartite and, if so, finds

a partition of V into sets L and R such that all edges go between L and R: run DFS and find a spanning forest, that is, a spanning tree of the graph in each connected component. Construct sets L and R in the following way. In each tree, put the root in L, and then put in R all the vertices that, in the tree, have odd distance from the root; put in L all the vertices that, in the tree, have even distance from the root. If the resulting partition is not valid, that is, if there is some edge both whose endpoints are in L or both whose endpoints are in R, then there is some tree in which two vertices u, v are connected by an edge, even though they are both at even distance or both at odd distance from the root r; in such a case, the cycle that goes from r to u along the tree, then follows the edge (u, v) and then goes from v to r along the three is an odd-length cycle, and it is easy to prove that in a bipartite graph there is no odd cycle. Hence the algorithm either returns a valid bipartition or a certificate that the graph is not bipartite.

Several optimization problems become simpler in bipartite graphs. The problem of finding a *maximum matching* in a graph is solvable in polynomial time in general graphs, but it has a very simple algorithm in bipartite graphs, that we shall see shortly. (The algorithm for general graphs is beautiful but rather complicated.) The algorithm is based on a reduction to the maximum flow problem. The reduction has other applications, because it makes the machinery of the max flow - min cut theorem applicable to reason about matchings. We are going to see a very simple proof of Hall's theorem, a classical result in graph theorem, which uses the max flow - min cut theorem.

As another application, we are going to show how to solve optimally the minimum vertex cover problem in bipartite graphs using a minimum cut computation, and the relation between flows and matchings. In general graphs, the minimum vertex cover problem is NP-complete.

The problem of finding a *maximum matching* in a graph, that is, a matching with the largest number of edges, often arises in assignment problems, in which tasks are assigned to agents, and almost always the underlying graph is bipartite, so it is of interest to have simpler and/or faster algorithms for maximum matchings for the special case in which the input graph is bipartite.

We will describe a way to r*reduce* the maximum matching problem in bipartite graphs to the maximum flow problem, that is, a way to show that a given bipartite graph can be transformed into a network such that, after finding a maximum flow in the network, we can easily reconstruct a maximum matching in the original graph.

14.1 Maximum Matching in Bipartite Graphs

Recall that, in an undirected graph $G = (V, E)$, a *matching* is a subset of edges $M \subseteq E$ that have no endpoint in common. In a bipartite graph with bipartition (L, R), the edges of the matching, like all other edges, have one endpoint in L and one endpoint in R.

Consider the following algorithm.

- Input: undirected bipartite graph $G = (V, E)$, partition of V into sets L, R
- Construct a network $(G' = (V', E'), s, t, c)$ as follows:
  - the vertex set is $V' := V \cup \{s, t\}$, where s and t are two new vertices;
  - E' contains a directed edge (s, u) for every $u \in L$, a directed edge (u, v) for every edge $(u, v) \in E$, where $u \in L$ and $v \in R$, and a directed edge (v, t) for every $v \in R$;
  - each edge has capacity 1;
- find a maximum flow $f(\cdot, \cdot)$ in the network, making sure that all flows $f(u, v)$ are either zero or one
- return $M := \{(u, v) \in E \text{ such that } f(u, v) = 1\}$

The running time of the algorithm is the time needed to solve the maximum flow on the network (G', s, t, c) plus an extra $O(|E|)$ amount of work to construct the network and to extract the solution from the flow. In the constructed network, the maximum flow is at most $|V|$, and so, using the Ford-Fulkerson algorithm, we have running time $O(|E| \cdot |V|)$. The fastest algorithm for maximum matching in bipartite graphs, which applies the push-relabel algorithm to the network, has running time $O(|V| \cdot \sqrt{|E|})$. It is also possible to solve the problem in time $O(MM(|V|))$, where $MM(n)$ is the time that it takes to multiply two $n \times n$ matrices. (This approach does not use flows.) Using the currently best known matrix multiplication algorithm, the running time is about $O(|V|^{2.37})$, which is better than $O(|V|\sqrt{|E|})$ in dense graphs. The algorithm based on push-relabel is always better in practice.

Remark 14.1 (Integral Flows) *It is important in the reduction that we find a flow in which all flows are either zero or one. In a network in which all capacities are zero or one, all the algorithms that we have seen in class will find an optimal solution in which all flows are either zero or one. More generally, on input a network with integer capacities, all the algorithms that we have seen in class will find a maximum flow in which all $f(u, v)$ are integers. It is important to keep in mind, however, that, even though in a network with zero/one capacities there always exists an optimal integral flow, there can also be optimal flows that are not integral.*

We want to show that the algorithm is correct that is that: (1) the algorithm outputs a matching and (2) that there cannot be any larger matching than the one found by the algorithm.

Claim 14.2 *The algorithm always outputs a matching, whose size is equal to the cost of the maximal flow of G'.*

PROOF: Consider the flow $f(\cdot, \cdot)$ found by the algorithm. For every vertex $u \in L$, the conservation constraint for u and the capacity constraint on the edge (s, u) imply:

$$\sum_{r:(u,r)\in E} f(u,r) = f(s,u) \leq 1$$

and so at most one of the edges of M can be incident on u.

Similarly, for every $v \in R$ we have

$$\sum_{\ell:(\ell,v)\in E} f(\ell,v) = f(v,t) \leq 1$$

and so at most one of the edges in M can be incident on v. □

Remark 14.3 *Note that the previous proof does not work if the flow is not integral*

Claim 14.4 *The size of the largest matching in G is at most the cost of the maximum flow in G'.*

PROOF: Let M^* be a largest matching in G. We can define a feasible flow in G' in the following way: for every edge $(u,v) \in M^*$, set $f(s,u) = f(u,v) = f(v,t) = 1$. Set all the other flows to zero. We have defined a feasible flow, because every flow is either zero or one, and it is one only on edges of G', so the capacity constraints are satisfied, and the conservation constraints are also satisfied, because for every vertex that is not matched in M^* there is zero incoming flow and zero outgoing flow, while for the matched vertices there is one unit of incoming flow and one unit of outgoing flow. The cost of the flow is the number of vertices in L that are matched, which is equal to $|M^*|$.

This means that there exists a feasible flow whose cost is equal to $|M^*|$, and so the cost of a maximum flow is greater than or equal to $|M^*|$. □

So we have established that our algorithm is correct and optimal.

14.2 Perfect Matchings in Bipartite Graphs

A *perfect* matching is a matching with $|V|/2$ edges. In a bipartite graph, a perfect matching can exist only if $|L| = |R|$, and we can think of it as defining a bijective mapping between L and R.

For a subset $A \subseteq L$, let us call $N(A) \subseteq R$ the *neighborhood* of A, that is, the set of vertices $\{r \in R : \exists a \in A.(a,r) \in E\}$ that are connected to vertices in A by an edge in E. Clearly, if there is a perfect matching in a bipartite graph $G = (V,E)$ with bipartition (L,R), then we must have $|A| \leq |N(A)|$, because the edges of the perfect matching match each vertex in A to a distinct vertex in $N(A)$, and this is impossible if $|N(A)| < |A|$.

A classical result in graph theory, Hall's Theorem, is that this is the only case in which a perfect matching does not exist.

Theorem 14.5 (Hall) *A bipartite graph $G = (V, E)$ with bipartition (L, R) such that $|L| = |R|$ has a perfect matching if and only if for every $A \subseteq L$ we have $|A| \leq |N(A)|$.*

The theorem precedes the theory of flows and cuts in network, and the original proof was constructive and a bit complicated. We can get a very simple non-constructive proof from the max flow - min cut theorem.

PROOF: We have already seen one direction of the theorem. It remains to prove that if $|A| \leq |N(A)|$ for every $A \subseteq L$, then G has a perfect matching.

Equivalently, we will prove that if G does not have a perfect matching, then there must be a set $A \subseteq V$ such that $|A| > |N(A)|$.

Let us construct the network (G', s, t, c) as in the algorithm above, an let us call $n = |L| = |R|$. If G does not have a perfect matching, then it means that the size of the maximum matching in G is $\leq n - 1$, and so the size of the maximum flow in G' is $\leq n - 1$, and so G' must have a cut of capacity $\leq n - 1$. Let us call the cut S.

Let us call $L_1 := S \cap L$ the left vertices in S, and $L_2 := L - S$ the remaining left vertices, and similarly $R_1 := S \cap R$ and $R_2 := R - S$.

In the network G', all edges have capacity one, so the capacity of the cut S is the number of edges that go from S to the complement of S, that is

$$capacity(S) = |L_2| + |R_1| + edges(L_1, R_2)$$

where $|L_2|$ is the number of edges from s to the complement of S, $|R_1|$ is the number of edges from S into t, and $edges(L_1, R_2)$ is the number of edges in E with one endpoint in L_1 and one endpoint in R_2.

This means that we have

$$n - 1 \geq |L_2| + |R_1| + edges(L_1, R_2)$$

and, recalling that $|L_1| = n - |L_2|$,

$$|L_1| \geq |R_1| + edges(L_1, R_2) + 1$$

We can also see that

$$|N(L_1)| \leq |R_1| + edges(L_1, R_2)$$

because the neighborhood of L_1 can at most include $edges(L_1, R_2)$ vertices in R_2. Overall, we have

$$|L_1| \geq N(L_1) + 1$$

and so we have found a set on the left that is bigger than its neighborhood. □

14.3 Vertex Cover in Bipartite Graphs

The work that we have done on matching in bipartite graphs also gives us a very simple polynomial time algorithm for vertex cover.

- Input: undirected bipartite graph $G = (V, E)$, partition of V into sets L, R
- Construct a network $(G' = (V', E'), s, t, c)$ as before
- Find a minimum-capacity cut S in the network
- Define $L_1 := L \cap S$, $L_2 := L - S$, $R_1 := R \cap S$, $R_2 := R - S$
- Let B be the set of vertices in R_2 that have neighbors in L_1
- $C := L_2 \cup R_1 \cup B$
- output C

We want to show that the algorithm outputs a vertex cover, and that the size of the output set C is indeed the size of the minimum vertex cover.

Claim 14.6 *The output C of the algorithm is a vertex cover*

Proof: The set C covers all edges that have an endpoint either in L_2 or R_1, because C includes all of L_2 and all or R_1. Regarding the remaining edges, that is, those that have endpoint in L_1 and the other endpoint in R_2, all such edges are covered by B. □

Claim 14.7 *There is no vertex cover of size smaller than $|C|$.*

Proof: Let k be the capacity of the cut. Then k is equal to

$$|L_2| + |R_1| + edges(L_1, R_2)$$

and so

$$k \geq |L_2| + |R_1| + |B| = |C|$$

but k is equal to the capacity of the minimum cut in G', which is equal to the cost of the maximum flow in G' which, by what we proved in the previous section, is equal to the size of the maximum matching in G. This means that G has a matching of size k, and so every vertex cover must have size $\geq k \geq |C|$. □

Lecture 15

The Linear Program of Max Flow

In which we look at the linear programming formulation of the maximum flow problem, construct its dual, and find a randomized-rounding proof of the max flow - min cut theorem.

In the first part of the course, we designed approximation algorithms "by hand," following our combinatorial intuition about the problems. Then we looked at linear programming relaxations of the problems we worked on, and we saw that approximation algorithms for those problems could also be derived by rounding a linear programming solution. We also saw that our algorithms could be interpreted as constructing, at the same time, an integral primal solution and a feasible solution for the dual problem.

Now that we have developed exact combinatorial algorithms for a few problems (maximum flow, minimum s-t cut, global min cut, maximum matching and minimum vertex cover in bipartite graphs), we are going to look at linear programming relaxations of those problems, and use them to gain a deeper understanding of the problems and of our algorithms.

We start with the maximum flow and the minimum cut problems.

15.1 The LP of Maximum Flow and Its Dual

Given a network $(G = (V, E), s, t, c)$, the problem of finding the maximum flow in the network can be formulated as a linear program by simply writing down the definition of feasible flow.

We have one variable $f(u,v)$ for every edge $(u,v) \in E$ of the network, and the problem is:

$$\begin{aligned} \text{maximize} \quad & \sum_{v:(s,v)\in E} f(s,v) \\ \text{subject to} \quad & \\ & \sum_{u:(u,v)\in E} f(u,v) = \sum_{w:(v,w)\in E} f(v,w) \quad \forall v \in V - \{s,t\} \\ & f(u,v) \le c(u,v) \qquad\qquad\qquad\qquad \forall (u,v) \in E \\ & f(u,v) \ge 0 \qquad\qquad\qquad\qquad\qquad \forall (u,v) \in E \end{aligned} \tag{15.1}$$

Now we want to construct the dual.

When constructing the dual of a linear program, it is often useful to rewrite it in a way that has a simpler structure, especially if it is possible to rewrite it in a way that has fewer constraints (which will correspond to fewer dual variables), even at the cost of introducing several new variables in the primal.

A very clean way of formulating the maximum flow problem is to think in terms of the *paths* along which we are going to send the flow, rather than in terms of how much flow is passing through a specific edge, and this point of view makes the conservation constraints unnecessary.

In the following formulation, we have one variable x_p for each of the possible simple paths from s to t (we denote by P the set of such paths), specifying how much of the flow from s to t is being routed along the path p:

$$\begin{aligned} \text{maximize} \quad & \sum_{p\in P} x_p \\ \text{subject to} \quad & \\ & \sum_{p\in P:(u,v)\in p} x_p \le c(u,v) \quad \forall (u,v) \in E \\ & x_p \ge 0 \qquad\qquad\qquad\qquad\qquad\quad \forall p \in P \end{aligned} \tag{15.2}$$

Note that, usually, a network has exponentially many possible paths from s to t, and so the linear program (15.2) has an exponential number of variables. This is ok because we are never going to write down (15.2) for a specific network and pass it to a linear programming solver; we are interested in (15.2) as a mathematical specification of the maximum flow problem. If we want to actually find a maximum flow via linear programming, we will use the equivalent formulation (15.1).

(There are several other cases in combinatorial optimization in which a problem has a easier-to-understand linear programming relaxation or formulation that is exponentially big, and one can prove that it is equivalent to another relaxation or formulation of polynomial size. One then proves theorems about the big linear program, and the theorems apply to the small linear program as well, because of the equivalence. Then the small linear program can be efficiently solved, and the theorems about the big linear program can be turned into efficient algorithms.)

Let us first confirm that indeed (15.1) and (15.2) are equivalent.

Fact 15.1 *If $f(\cdot,\cdot)$ is a feasible solution for (15.1), then there is a feasible solution for (15.2) of the same cost.*

PROOF: Note that this is exactly the Flow Decomposition Theorem that we proved in Lecture 11, in which it is stated as Lemma 2. □

Fact 15.2 *If $\{x_p\}_{p\in P}$ is a feasible solution for (15.2), then there is a feasible solution for (15.1) of the same cost.*

PROOF: Define

$$f(u,v) := \sum_{p\in P:(u,v)\in p} x_p$$

that is, let $f(u,v)$ the sum of the flows of all the paths that use the edge (u,v). Then $f(\cdot,\cdot)$ satisfies the capacity constraints and, regarding the conservation constraints, we have

$$\sum_{u:(u,v)\in E} f(u,v) = \sum_{p\in P:v\in p} x_p = \sum_{w:(v,w)\in E} f(u,v)$$

□

Let us now construct the dual of (15.2). We have one dual variable $y_{u,v}$ for every edge $(u,v)\in E$, and the linear program is:

$$\begin{aligned} \text{minimize} \quad & \sum_{(u,v)\in E} c(u,v)y_{u,v} \\ \text{subject to} \quad & \\ & \sum_{(u,v)\in p} y_{u,v} \geq 1 \qquad \forall p \in P \\ & y_{u,v} \geq 0 \qquad\qquad \forall (u,v)\in E \end{aligned} \tag{15.3}$$

The linear program (15.3) is assigning a weight to each edges, which we may think of as a "length," and the constraints are specifying that, along each possible path, s and t are at distance at least one. This means that dual variables are expressing a way of "separating" s from t and, after thinking about it for a moment, we see that (15.3) can be seen as a *linear programming relaxation of the minimum cut problem.*

Fact 15.3 *For every feasible cut A in the network (G,s,t,c), there is a feasible solution $\{y_{u,v}\}_{(u,v)\in E}$ to (15.3) whose cost is the same as the capacity of A.*

PROOF: Define $y_{u,v} = 1$ if $u \in A$ and $v \notin A$, and let $y_{u,v} = 0$ otherwise. Then

$$\sum_{u,v} c(u,v) y_{u,v} = \sum_{u \in A, v \notin A} c(u,v) = capacity(A)$$

□

This means that the optimum of (15.3) is smaller than or equal to the capacity of the minimum cut in the network. Now we are going to describe a randomized rounding method that shows that the optimum of (15.3) is actually equal to the capacity of the minimum cut. Since the optimum of (15.3) is equal to the optimum of (15.2) by the Strong Duality Theorem, and we have proved that the optimum of (15.3) is equal to the cost of the maximum flow of the network, Lemma 15.4 below will prove that the cost of the maximum flow in the network is equal to the capacity of the minimum flow, that is, it will be a different proof of the max flow - min cut theorem. It is actually a more difficult proof (because it uses the Strong Duality Theorem whose proof, which we have skipped, is not easy), but it is a genuinely different one, and a useful one to understand, because it gives an example of how to use randomized rounding to solve a problem optimally. (So far, we have only seen examples of the use of randomized rounding to design *approximate* algorithms.)

Lemma 15.4 *Given any feasible solution $\{y_{u,v}\}_{(u,v)\in E}$ to (15.3), it is possible to find a cut A such that*

$$capacity(A) \leq \sum_{u,v} c(u,v) y_{u,v}$$

PROOF: Interpret the $y_{u,v}$ as weights on the edges, and use Dijkstra's algorithm to find, for every vertex v, the distance $d(v)$ from s to v according to the weights $y_{u,v}$.

The constraints in (15.3) imply that $d(t) \geq 1$.

Pick a value T uniformly at random in the interval $[0, 1)$, and let A be the set

$$A := \{v : d(v) \leq T\}$$

Then, for every choice of T, A contains s and does not contain t, and so it is a feasible cut.

Using linearity of expectation, the average (over the choice of T) capacity of A can be written as

$$\mathop{\mathbb{E}}_{T \sim [0,1)} capacity(A) = \sum_{(u,v)\in E} c(u,v)\, \mathbb{P}[u \in A \wedge v \notin A]$$

and

$$\mathbb{P}[u \in A \wedge v \notin A] = \mathbb{P}[d(u) \leq T < d(v)] = d(v) - d(u)$$

Finally, we observe the "triangle inequality"

$$d(v) \leq d(u) + y_{u,v}$$

which says that the shortest path from s to v is at most the length of the shortest path from s to u plus the length of the edge (u, v).

Putting all together, we have

$$\mathop{\mathbb{E}}_{T\sim[0,1)} capacity(A) \leq \sum_{(u,v)\in E} c(u,v)y_{u,v}$$

and there clearly must exist a choice of T for which the capacity of A is at most the expected capacity.

About finding A efficiently, we can also note that, although there is an infinite number of choices for T, there are only at most $|V| - 1$ different cuts that can be generated. If we sort the vertices in increasing order of $d(v)$, and let them be $u_1, \ldots, u_{|V|}$ in this order, then we have $s = u_1$, and let k be such that $d(u_k) < 1$ but $d(u_{k+1}) \geq 1$. Then the only cuts which are generated in our probability distribution are the k cuts of the form

$$A_i := \{s = u_1, u_2, \ldots, u_i\}$$

for $i = 1, \ldots, k$, and one of them must have capacity $\leq \sum_{(u,v)\in E} y_{u,v}c(u,v)$. We can compute the capacity of each A_i and pick the A_i with the smallest capacity. □

Let us now see what the dual of (15.1) looks like. It will look somewhat more mysterious than (15.3), but now we know what to expect: because of the equivalence between (15.1) and (15.2), the dual of (15.1) will have to be a linear programming relaxation of the minimum cut problem, and it will have an exact randomized rounding procedure.

The dual of (15.1) has one variable for each vertex v (except s and t), which we shall call y_v, corresponding to the conservation constraints, and one variable for each edge, which we shall call $y_{u,v}$, corresponding to the capacity constraints.

$$\begin{aligned} \text{minimize} \quad & \sum_{(u,v)\in E} c(u,v)y_{u,v} \\ \text{subject to} \quad & \\ & y_v + y_{s,v} \geq 1 && \forall v : (s,v) \in E \\ & y_v - y_u + y_{u,v} \geq 0 && \forall (u,v) \in E, u \neq s, v \neq t \\ & -y_u + y_{u,t} \geq 0 && \forall u : (u,t) \in E \end{aligned} \tag{15.4}$$

Let us see that (15.4) is a linear programming relaxation of the minimum cut problem and that it admits an exact rounding algorithm.

Fact 15.5 *If A is a feasible cut in the network, then there is a feasible solution to (15.4) such that*

$$capacity(A) = \sum_{(u,v)\in E} c(u,v) y_{u,v}$$

PROOF: Define $y_v = 1$ if $v \in A$, and $y_v = 0$ if $v \notin A$. Define $y_{u,v} = 1$ if $u \in A$ and $v \notin A$, and $y_{u,v} = 0$ otherwise.

To see that it is a feasible solution, let us first consider the constraints of the first kind. They are always satisfied because if $v \in A$ then $y_v = 1$, and if $v \notin A$ then (s, v) crosses the cut and $y_{s,v} = 1$, so the left-hand-side is always at least one. We can similarly see that the constraints of the third type are satisfied.

Regarding the constraints of the second kind, we can do a case analysis and see that the constraint is valid if $y_u = 0$ (regardless of the value of the other variables), and it is also valid if $y_v = 1$ (regardless of the value of the other variables). The remaining case is $y_u = 1$ and $y_v = 0$, which is the case in which (u, v) crosses the cut and so $y_{u,v} = 1$. □

Fact 15.6 *Given a feasible solution of (15.4), we can find a feasible cut whose capacity is equal to the cost of the solution.*

PROOF: Pick uniformly at random T in $[0, 1]$, then define

$$A := \{s\} \cup \{v : y_v \geq T\}$$

This is always a cut, because, by construction, it contains s and it does not contain t. (Recall that there is no variable y_t because there is no conservation constraint for t.)

Then we have

$$\mathbb{E}\, capacity(A) = \sum_{u,v} c(u,v)\, \mathbb{P}[u \in A \wedge v \notin A]$$

It remains to argue that, for every edge (u, v), we have

$$\mathbb{P}[u \in A \wedge v \notin A] \leq y_{u,v}$$

For edges of the form (s, v), we have

$$\mathbb{P}[s \in A \wedge v \notin A] = \mathbb{P}[v \notin A] = \mathbb{P}[y_v < T \leq 1] = 1 - y_v \leq y_{s,v}$$

(Actually, the above formula applies if $0 \leq y_v < 1$. If $y_v \geq 1$, then the probability is zero and $y_{s,v} \geq 0$ and we are fine; if $y_v < 0$, then the probability is one, and $y_{s,v} \geq 1 - y_v > 1$, so we are again fine.)

For edges of the form (u, v) in which u and v are in $V - \{s, t\}$ we have

$$\mathbb{P}[u \in A \wedge v \notin A] = \mathbb{P}[y_v < T \leq y_u] = y_u - y_v \leq y_{u,v}$$

(Again, we have to look out for various exceptional cases, such as the case $y_v \geq y_u$, in which case the probability is zero and $y_{u,v} \geq 0$, and the case $y_v < 0$, and the case $y_u > 1$.)

For edges of the form (v, t), we have

$$\mathbb{P}[v \in A \wedge t \notin A] = \mathbb{P}[v \in A] = \mathbb{P}[y_v \geq T] = y_v \leq y_{v,t}$$

(Same disclaimers.) $\square$

Lecture 16

Multicommodity Flow

In which we define a multi-commodity flow problem, and we see that its dual is the relaxation of a useful graph partitioning problem. The relaxation can be rounded to yield an approximate graph partitioning algorithm.

16.1 Generalizations of the Maximum Flow Problem

An advantage of writing the maximum flow problem as a linear program, as we did in the past lecture, is that we can consider variations of the maximum flow problem in which we add extra constraints on the flow and, as long as the extra constraints are linear, we are guaranteed that we still have a polynomial time solvable problem. (Because we can still write the problem as a linear program, and we can solve linear programming in polynomial time.)

Certain variants of maximum flow are also easily reducible to the standard maximum flow problem, and so they are solvable using the combinatorial algorithms that we have discussed.

Example 16.1 (Vertex Capacities) *An interesting variant of the maximum flow problem is the one in which, in addition to having a capacity $c(u, v)$ for every edge, we also have a capacity $c(u)$ for every vertex, and a flow $f(\cdot, \cdot)$ is feasible only if, in addition to the conservation constraints and the edge capacity constraints, it also satisfies the vertex capacity constraints*

$$\sum_{u:(u,v)\in E} f(u, v) \leq c(v) \quad \forall v \in V$$

It is easy to see that the problem can be reduced to the standard maximum flow problem, by splitting every vertex v into two vertices v_{in} and v_{out}, adding one edge (v_{in}, v_{out}) of capacity $c(v)$, and then converting every edge (u, v) to an edge (u, v_{in}) and every edge (v, w) to an edge (v_{out}, w). It is easy to show that solving the (standard) maximum flow problem on the

new network is equivalent to solving the maximum flow with vertex capacity constraints in the original network.

Example 16.2 (Multiple Sources and Sinks and "Sum" Cost Function) *Several important variants of the maximum flow problems involve multiple source-sink pairs $(s_1, t_1), \ldots, (s_k, t_k)$, rather than just one source and one sink. Assuming that the "stuff" that the sources want to send to the sinks is of the same type, the problem is to find multiple feasible flows $f^1(\cdot, \cdot)$, $\ldots$, $f^k(\cdot, \cdot)$, where $f^i(\cdot, \cdot)$ is a feasible flow from the source s_i to the sink t_i, and such that the capacity constraints*

$$\sum_{i=1}^{k} f^i(u, v) \leq c(u, v) \quad \forall (u, v) \in E$$

are satisfied. Such a flow is called a "multi-commodity" flow.

How do we measure how "good" is a multicommodity flow? A simple measure is to consider the sum of the costs

$$\sum_{i=1}^{k} \sum_{v} f^i(s_i, v)$$

In such a case, we can do a reduction to the standard maximum flow problem by adding a "super-source" node s, connected with edges of infinite capacity to the sources s_i, and a "super-sink" node t, to which all sinks t_i are connected to, via infinite capacity edges. It is easy to see that the maximum flow from s to t is the same as the maximum sum of flows in a feasible multicommodity flow in the original network.

In many applications, looking at the sum of the costs of the various flows $f^i(\cdot, \cdot)$ is not a "fair" measure. For example, if the underlying network is a communication network, and $(s_1, t_1), (s_2, t_2)$ are pairs of nodes that need to communicate, a solution that provides $5Mb/s$ of bandwidth between s_1 and t_1 and no bandwidth between s_2 and t_2 is not a very good solution compared, for example, with a solution that provides $2Mb/s$ of bandwidth each between s_1 and t_1 and between s_2 and t_2. (Especially so from the point of view of s_2 and t_2.) There are various reasonable measures of the quality of a multicommodity flow which are more fair, for example we may be interested in maximizing the median flow, or the minimum flow. A rather general problem, which can be used to find multicommodity flows maximizing various cost measures is the following.

Definition 16.3 (Multicommodity Feasibility Problem) *Given in input a network $G = (V, E_G)$ with capacities $c(u, v)$ for each $(u, v) \in E_G$, and given a collection of (not necessarily disjoint) pairs (s_1, t_1), $\ldots$, (s_k, t_k), each having a demand $d(s_i, t_i)$, find a feasible multicommodity flow $f^1(\cdot, \cdot), \ldots, f^k(\cdot, \cdot)$ such that*

$$\sum_{v} f^i(s_i, v) \geq d(s_i, t_i) \quad \forall i = 1, \ldots, k$$

or determine that no such multicommodity flow exists.

A more general version, which is defined as an optimization problem, is as follows.

Definition 16.4 (Maximizing Fractional Demands) *Given in input a network $G = (V, E_G)$ with capacities $c(u, v)$ for each $(u, v) \in E_G$, and given a collection of (not necessarily disjoint) pairs (s_1, t_1), ..., (s_k, t_k), each having a demand $d(s_i, t_i)$, find a feasible multicommodity flow $f^1(\cdot, \cdot), \ldots, f^k(\cdot, \cdot)$ such that*

$$\sum_v f^i(s_i, v) \geq y \cdot d(s_i, t_i) \quad \forall i = 1, \ldots, k$$

and such that y is maximized.

Note that the vertices $s_1, \ldots, s_k, t_1, \ldots, t_k$ need not be distinct. For example, in the case of a communication network, we could have a broadcast problem in which a node s wants to send data to all other nodes, in which case the source-sink pairs are all of the form (s, v) for $v \in V - \{s\}$. It is useful to think of the pairs of vertices that require communication as defining a weighted graph, with the weights given by the demands. We will call $H = (V, E_H)$ the graph of demands. (In the broadcast example, H would be a star graph.)

The Fractional Multicommodity Flow Problem can be easily formulated as a linear program.

$$\begin{array}{lll}
\text{maximize} & y & \\
\text{subject to} & & \\
& \displaystyle\sum_u f^{s,t}(u, v) = \sum_w f^{s,t}(v, w) & \forall (s, t) \in E_H \forall v \in V - \{s, t\} \\
& \displaystyle\sum_{(s,t) \in E_H} f^{s,t}(u, v) \leq c(u, v) & \forall (u, v) \in E_G \\
& \displaystyle\sum_v f^{s,t}(s, v) \geq y \cdot d(s, t) & \forall (s, t) \in E_H \\
& f^{s,t}(u, v) \geq 0 & \forall (s, t) \in E_H, (u, v) \in E_G
\end{array} \tag{16.1}$$

As for the standard maximum flow problem, it is also possible to give a formulation that involves an exponential number of variables, but for which it is easier to derive the dual.

In the following formulation, $P_{s,t}$ is the set of all paths from s to t in G, and we have a variable x_p for each path in $P_{s,t}$, for each $(s, t) \in E_H$, corresponding to how many units of flow from s to t are routed through the path p.

$$\begin{array}{lll}
\text{maximize} & y & \\
\text{subject to} & & \\
& \displaystyle\sum_{p \in P_{s,t}} x_p \geq y \cdot d(s, t) & \forall (s, t) \in E_H \\
& \displaystyle\sum_{p:(u,v) \in p} x_p \leq c(u, v) & \forall (u, v) \in E_G \\
& x_p \geq 0 & \forall p \\
& y \geq 0 &
\end{array} \tag{16.2}$$

Note that if E_H contains only one edge (s,t), and $d(s,t) = 1$, then we have the standard maximum flow problem.

16.2 The Dual of the Fractional Multicommodity Flow Problem

The dual of (16.2) has one variable $w(s,t)$ for each $(s,t) \in E_H$, and a one variable $z(u,v)$ for each $(u,v) \in E_G$. It is as follows:

$$\begin{array}{ll} \text{minimize} & \sum_{u,v} z(u,v)c(u,v) \\ \text{subject to} & \\ & \displaystyle\sum_{(s,t)\in E_H} w(s,t)d(s,t) \geq 1 \\ & -w(s,t) + \displaystyle\sum_{(u,v)\in p} z(u,v) \geq 0 \quad \forall (s,t) \in E_H, p \in P_{s,t} \\ & w(s,t) \geq 0 \qquad\qquad \forall (s,t) \in E_H \\ & z(u,v) \geq 0 \qquad\qquad \forall (u,v) \in E_G \end{array} \tag{16.3}$$

Thinking a bit about (16.3) makes us realize that, in an optimal solution, without loss of generality $w(s,t)$ is be the shortest path from s to t in the graph weighted by the $z(u,v)$. Indeed, the constraints force $w(s,t)$ to be *at most* the length of the shortest $z(\cdot,\cdot)$-weighted path from s to t, and, if some $w(s,t)$ is strictly smaller than the length of the shortest path, we can make it equal to the length of the shortest path without sacrificing feasibility and without changing the cost of the solution. The other observation is that, in an optimal solution, we have $\sum w(s,t)d(s,t) = 1$, because, in a solution in which $\sum w(s,t)d(s,t) = c > 1$, we can divide all the $w(s,t)$ and all the $z(u,v)$ by c, and obtain a solution that is still feasible and has smaller cost. This means that the following linear program is equivalent to (16.3). We have a variable $\ell(x,y)$ for every pair of vertices in $E_G \cup E_H$:

$$\begin{array}{ll} \text{minimize} & \sum_{u,v} \ell(u,v)c(u,v) \\ \text{subject to} & \\ & \displaystyle\sum_{(s,t)\in E_H} \ell(s,t)d(s,t) = 1 \\ & \displaystyle\sum_{(u,v)\in p} \ell(u,v) \geq \ell(s,t) \qquad \forall (s,t) \in E_H, p \in P_{s,t} \\ & \ell(u,v) \geq 0 \qquad\qquad \forall (u,v) \in E_G \cup E_H \end{array} \tag{16.4}$$

16.3 The Sparsest Cut Problem

From now on, we restrict ourselves to the case in which the graphs E_G and E_H are undirected. In such a case, we have a variable $\ell(u,v)$ for each *unordered* pair u,v. The constraints $\sum_{(u,v)\in p} \ell(u,v) \geq \ell(s,t)$ can be equivalently restated as the *triangle inequalities*

$$\ell(u_1, u_3) \leq \ell(u_1, u_2) + \ell(u_2, u_3)$$

This means that we are requiring $\ell(u, v)$ to be non-negative, symmetric and to satisfy the triangle inequality, and so it is a *metric* over V. (Technically, it is a *semimetric* because we can have distinct vertices at distance zero, and $\ell(\cdot, \cdot)$ is not defined for all pairs, but only for pairs in $E_G \cup E_H$, although we could extend it to all pairs by computing all-pairs shortest paths based on the weights $\ell(x, y)$ for $(x, y) \in E_G \cup E_H$.)

These observations give us one more alternative formulation:

$$\min_{\ell(\cdot,\cdot) \text{ metric}} \frac{\sum_{(u,v)\in E_G} c(u, v) \cdot \ell(u, v)}{\sum_{(s,t)\in E_H} d(s, t) \cdot \ell(s, t)}$$

Now, finally, we can see that the above formulation is the linear programming relaxation of a *cut* problem.

If $A \subseteq V$ is a subset of vertices, we say that a pair (u, v) is *cut* by A if $u \in A$ and $v \notin A$, or vice versa.

Given an instance of the multicommodity flow problem, we say that a subset A of vertices is a *cut* if it cuts at least one of the pairs in E_H. The *sparsest cut* (also called quotient cut) problem is to find a cut A that minimizes the ratio

$$\frac{\displaystyle\sum_{(u,v)\in E_G \text{ cut by } A} c(u, v)}{\displaystyle\sum_{(s,t)\in E_H \text{ cut by } A} d(s, t)}$$

which is called the *sparsity* of the cut A.

Note that, if E_H contains only one pair (s, t), and $d(s, t) = 1$, then we have exactly the standard minimum cut problem.

Suppose that, in our multicommodity problem, there is a fractional flow of cost y. Then, for each pair (s, t) that is cut by A, the $yd(s, t)$ units of flow from s to t must pass through edges of E_G that are also cut by A. Overall, $\sum_{(s,t)\text{cut by } A} yd(s, t)$ units of flow must pass through those edges, whose overall capacity is at most $\sum_{(u,v)\text{cut by } A} c(u, v)$, so we must have

$$\sum_{(u,v)\text{cut by } A} c(u, v) \geq y \sum_{(s,t)\text{cut by } A} d(s, t)$$

which means that the sparsity of A must be at least y. This means that sparsity of every cut is at least the fractional cost of any flow. (This is not surprising because we derived the sparsest cut problem from the dual of the flow problem, but there is a very simple direct reason why the above bound holds.)

Now it would be very nice if we had an exact rounding algorithm to find the optimum of the sparsest cut problem.

For a given graph $G = (V, E_G)$ with capacities $c(u, v)$, if we define E_H to be a clique and $d(s, t) = 1$ for all s, t, then solving the sparsest cut problem on G and H becomes the problem of finding a set A that minimizes

$$\frac{\sum_{(u,v)\in E_G \text{ cut by } A} c(u, v)}{|A| \cdot |V - A|}$$

and optimizing such a cost function tends to favor finding sets A that are large and that have few edges coming out. This is useful in a number of contexts. In clustering problems, if the capacities represent similarity, a sparsest cut algorithm will pick out sets of vertices that are mostly similar to each other, but dissimilar to the other vertices, that is, a cluster. Very effective image segmentation algorithms are based on applying sparsest cut approximation algorithms (but not the one we are describing in these notes, which is too slow) to graphs in which there is a vertex for every pixel, and edges connect nearby pixels with a capacity corresponding to how likely the pixels are to belong to same object in the image.

Unfortunately, the sparsest cut problem is NP-hard. Rounding (16.4), however, it is possible to achieve a $O(\log |E_H|)$-factor approximation.

We very briefly describe what the approximation algorithm looks like.

First, we need the following result:

Lemma 16.5 *For every input G, H, c, d, and every feasible solution $\ell(\cdot, \cdot)$ of (16.4), it is possible to find in polynomial time a subset S of vertices, such that if we define*

$$g_S(v) := \min_{a\in S} \ell(a, v)$$

then we have

$$\sum_{(s,t)\in E_H} \ell(s, t)d(s, t) \le O(\log |E_H|) \sum_{(s,t)\in E_H} |g_S(s) - g_S(t)|d(s, t)$$

The proof is rather complicated, and we will skip it.

Then we have the following fact:

Lemma 16.6 *For every input G, H, c, d, every feasible solution $\ell(\cdot, \cdot)$ of (16.4), and every subset S of vertices, if we define $g_S(v) := \min_{a\in S} \ell(a, v)$, we have*

$$\sum_{(u,v)\in E_G} \ell(u, v)c(u, v) \ge \sum_{(u,v)\in E_G} |g_S(u) - g_S(v)|c(u, v)$$

PROOF: It is enough to show that we have, for every u, v,

$$\ell(u, v) \ge |g_S(u) - g_S(v)|$$

Let a be the vertex such that $\ell(a,u) = g_S(u)$ and b be the vertex such that $\ell(b,v) = g_S(v)$. (They need not be different.) Then, from the triangle inequality, we get

$$\ell(u,v) \geq \ell(u,b) - \ell(b,v) \geq \ell(u,a) - \ell(b,v) = g_S(u) - g_S(v)$$

and

$$\ell(u,v) \geq \ell(v,a) - \ell(a,u) \geq \ell(v,b) - \ell(u,a) = g_S(v) - g_S(a)$$

and so

$$\ell(u,v) \geq |g_S(u) - g_S(v)|$$

□

Lemma 16.7 *For every input G, H, c, d, and every function $g : V \to \mathbb{R}$, we can find in polynomial time a cut A such that*

$$\frac{\sum_{(u,v)\in E_G \text{ cut by } A} c(u,v)}{\sum_{(s,t)\in E_H \text{ cut by } A} d(s,t)} \leq \frac{\sum_{(u,v)\in E_G} |g(u) - g(v)| c(u,v)}{\sum_{(s,t)\in E_H} |g(s) - g(t)| d(s,t)}$$

PROOF: We sort the vertices in ascending value of g, so that we have an ordering $u_1, \ldots, u_n$ of the vertices such that

$$g(u_1) \leq g(u_2) \leq \cdots \leq g(u_n)$$

We are going to consider all the cuts of the form $A := \{u_1, \ldots, u_k\}$, and we will show that at least one of them has sparsity at most

$$r := \frac{\sum_{(u,v)\in E_G} |g(u) - g(v)| c(u,v)}{\sum_{(s,t)\in E_H} |g(s) - g(t)| d(s,t)}$$

Since r does not change if we scale $g(\cdot)$ by a multiplicative constant, we will assume without loss of generality that $g(u_n) - g(u_1) = 1$.

Let us pick a threshold T uniformly at random in the interval $[g(u_1), g(u_n)]$, and define the set $A := \{u : g(u) \leq T\}$. Now note that, for every pair of vertices x, y, the probability that (x,y) is cut by A is precisely $|g(x) - g(y)|$, and so

$$\mathbb{E} \sum_{(u,v)\in E_G \text{ cut by } A} c(u,v) = \sum_{(u,v)\in E_G} |g(u) - g(v)| c(u,v)$$

and

$$\mathbb{E} \sum_{(s,t)\in E_H \text{ cut by } A} d(s,t) = \sum_{(s,t)\in E_H} |g(s) - g(t)| d(s,t)$$

so that

$$\mathbb{E} \sum_{(u,v)\in E_G \text{ cut by } A} c(u,v) - r \sum_{(s,t)\in E_H \text{ cut by } A} d(s,t) = 0$$

and so there must exist an A in our sample space (which consists of sets of the form $\{u_1, \ldots, u_k\}$) such that

$$\sum_{(u,v)\in E_G \text{ cut by } A} c(u,v) - r \sum_{(s,t)\in E_H \text{ cut by } A} d(s,t) \geq 0$$

that is,

$$\frac{\displaystyle\sum_{(u,v)\in E_G \text{ cut by } A} c(u,v)}{\displaystyle\sum_{(s,t)\in E_H \text{ cut by } A} d(s,t)} \leq r$$

□

This is enough to have an $O(\log |E_H|)$-approximate algorithm for the sparsest cut problem.

On input the graphs G, H, the capacities $c(\cdot,\cdot)$ and the demands $d(\cdot,\cdot)$, we solve the linear program (16.4), and find an optimal solution $\ell(\cdot,\cdot)$ of cost *optlp*. Then we use Lemma 16.5 to find a set S such that, if we define $g_S(v) := \min_{a\in S} \ell(a,v)$, we have (using also Lemma 16.6)

$$\frac{\sum_{(u,v)\in E_G} |g(u) - g(v)| c(u,v)}{\sum_{(s,t)\in E_H} |g(s) - g(t)| d(s,t)} \leq \mathit{optlp} \cdot O(\log |E_H|)$$

Finally, we use the algorithm of Lemma 16.7 to find a cut A whose sparsity is at most $\mathit{optlp} \cdot O(\log |E_H|)$, which is at most $O(\log |E_H|)$ times the sparsity of the optimal cut. This proves the main result of this lecture.

Theorem 16.8 *There is a polynomial time $O(\log |E_H|)$-approximate algorithm for the sparsest cut problem.*

Lecture 17

Online Algorithms

In which we introduce online algorithms and discuss the buy-vs-rent problem, the secretary problem, and caching.

In this lecture and the next we will look at various examples of algorithms that operate under partial information. The input to these algorithms is provided as a "stream," and, at each point in time, the algorithms need to make certain decisions, based on the part of the input that they have seen so far, but without knowing the rest of the input. If we knew that the input was coming from a simple distribution, then we could "learn" the distribution based on an initial segment of the input, and then proceed based on a probabilistic prediction of what the rest of the input is going to be like. In our analysis, instead, we will mostly take a worst-case point of view in which, at any point in time, the unknown part of the input could be anything. Interestingly, however, algorithms that are motivated by "learn and predict" heuristics often work well also from the point of view of worst-case analysis.

17.1 Online Algorithms and Competitive Analysis

We will look at online algorithms for optimization problems, and we will study them from the point of view of *competitive analysis*. The *competitive ratio* of an online algorithm for an optimization problem is simply the approximation ratio achieved by the algorithm, that is, the worst-case ratio between the cost of the solution found by the algorithm and the cost of an optimal solution.

Let us consider a concrete example: we decide to go skiing in Tahoe for the first time. Buying the equipment costs about $500 and renting it for a weekend costs $50. Should we buy or rent? Clearly it depends on how many more times we are going to go skiing in the future. If we will go skiing a total of 11 times or more, then it is better to buy, and to do it now. If we will go 9 times or fewer, then it is better to rent, and if we go 10 times it does not matter. What is an "online algorithm" in this case? Each time we want to go skiing, unless we have bought equipment a previous time, we have to decide whether we are going to buy or rent. After we buy, there is no more decision to make; at any time, the

only "input" for the algorithm is the fact that this is the k-th time we are going skiing, and that we have been renting so far; the algorithm decides whether to buy or rent based on k. For deterministic algorithms, an algorithm is completely described by the time t at which it decides that it is time to buy.

What are the competitive ratios of the possible choices of t? If $t = 1$, that is if we buy before the first time we go skiing, then the competitive ratio is 10, because we always spend \$500, and if it so happens that we never go skiing again after the first time, then the optimum is \$50. If $t = 2$, then the competitive ratio is 5.5, because if we go skiing twice then we rent the first time and buy the second, spending a total of \$550, but the optimum is \$100. In general, for every $t \leq 10$, the competitive ratio is

$$\frac{500 + 50(t-1)}{50t} = 1 + \frac{9}{t}$$

If $t \geq 10$, then the competitive ratio is

$$\frac{500 + 50(t-1)}{500} = .9 + \frac{t}{10}$$

So the best choice of t is $t = 10$, which gives the competitive ratio 1.9.

The general rule for buy-versus-rent problems is to keep renting until what we have spent renting equals the cost of buying. After that, we buy.

(The "predicting" perspective is that if we have gone skiing 10 times already, it makes sense to expect that we will keep going at least 10 more times in the future, which justifies buying the equipment. We are doing worst-case analysis, and so it might instead be that we stop going skiing right after we buy the equipment. But since we have already gone 10 times, the prediction that we are going to go a total of at least 20 times is correct within a factor of two.)

17.2 The Secretary Problem

Suppose we have joined an online dating site, and that there are n people that we are rather interested in. We would like to end up dating the best one. (We are assuming that people are comparable, and that there is a consistent way, after meeting two people, to decide which one is better.) We could go out with all of them, one after the other, and then pick the best, but our traditional values are such that if we are dating someone, we are not going to go out with anybody else unless we first break up. Under the rather presumptuous assumption that everybody wants to date us, and that the only issue is who we are going to choose, how can we maximize the probability of ending up dating the best person? We are going to pick a random order of the n people, and go out, one after the other, with the first n/e people. In these first n/e dates, we just waste other people's time: no matter how the dates go, we tell them that it's not them, it's us, that we need some space and so on, and we move on to the next. The purpose of this first "phase" is to calibrate our expectations. After these n/e dates, we continue to go out on more dates following the random order, but as soon as we found someone who is *better than everybody we have seen so far*, that's

the one we are going to pick. We will show that this strategy picks the best person with probability about$1/e$, which is about 37%.

How does one prove such a statement? Suppose that our strategy is to reject the people we meet in the first t dates, and then from date $t+1$ on we pick the first person that is better than all the others so far. The above algorithm corresponds to the choice $t = n/e$.

Let us identify our n suitors with the integers $1, \ldots, n$, with the meaning that 1 is the best, 2 is the second best, and so on. After we randomly permute the order of the people, we have a random permutation π of the integers $1, \ldots, n$. The process described above corresponds to finding the minimum of $\pi[1], \ldots, \pi[t]$, where $t = n/e$, and then finding the first $j \geq t+1$ such that $\pi[j]$ is smaller than the minimum of $\pi[1], \ldots, \pi[t]$. We want to compute the probability that $\pi[j] = 1$. We can write this probability as

$$\sum_{j=t+1}^{n} \mathbb{P}[\pi[j] = 1 \text{ and we pick the person of the } j\text{-th date }]$$

Now, suppose that, for some $j > t$, $\pi[j] = 1$. When does it happen that we do *not* end up with the best person? We fail to get the best person if, between the $(t+1)$-th date and the $(j-1)$-th date we meet someone who is better than the people met in the first t dates, and so we pick that person instead of the best person. For this to happen, the minimum of $\pi[1], \ldots, \pi[j-1]$ has to occur in locations between $t+1$ and $j-1$. Equivalently, we *do* pick the best person if the best among the first $j-1$ people happen to be one of the first t people.

We can rewrite the probability of picking the best person as

$$\sum_{j=t+1}^{n} \mathbb{P}[\pi[j] = 1 \text{ and min of } \pi[1], \ldots, \pi[j-1] \text{ is in } \pi[1], \ldots, \pi[t]]$$

$$= \sum_{j=t+1}^{n} \frac{1}{n} \cdot \frac{t}{j-1}$$

To see that the above equation is right, $\mathbb{P}[\pi[j] = 1] = 1/n$ because, in a random permutation, 1 is equally likely to be the output of any of n possible inputs. Conditioned on $\pi[j] = 1$, the minimum of $\pi[1], \ldots, \pi[j-1]$ is equally likely to occur in any of the $j-1$ places, and so there is a probability $t/(j-1)$ that it occurs in one of the first t locations. (Some readers may find this claim suspicious; it can be confirmed by explicitly counting how many permutations are such that $\pi[j] = 1$ and $\pi[i] = \min\{\pi[1], \ldots, \pi[j-1]\}$, and to verify that for each $j > t$ and each $i \leq t$ the number of these permutations is exactly $(n-1)!/(j-1)$.)

So the probability of picking the best person is

$$\frac{t}{n} \sum_{j=t+1}^{n} \frac{1}{j-1} = \frac{t}{n} \left(\sum_{j=1}^{n-1} \frac{1}{j} - \sum_{j=1}^{t} \frac{i}{j} \right) \approx \frac{t}{n} \cdot (\ln n - \ln t) = \frac{t}{n} \ln \frac{n}{t}$$

And the last expression is optimized by $t = n/e$, in which case the expression is $1/e$.

Note that this problem was not in the "competitive analysis" framework, that is, we were not trying to find an approximate solution, but rather to find the optimal solution with high probability.

Note also that, with probability $1/e$, the algorithm causes us to pick nobody, because the best person is one of the first n/e with probability $1/e$, and when this happens we set our standards so high that we are ending up alone.

Suppose instead that we always want to end up with someone, and that we want to optimize the "rank" of the person we pick, that is, the place in which it fits in our ranking from 1 to n. If we apply the above algorithm, with the modification that we pick the last person if we have gotten that far, then with probability $1/e$ we pick the last person which, on average, has rank $n/2$, so the average rank of the person we pick is $\Omega(n)$. (This is not a rigorous argument, but it is close to the argument that establishes rigorously that the average is $\Omega(n)$.)

In general, any algorithm that is based on rejecting the first t people, and then picking the first subsequent one which is better than the first t, or the last one if we have gotten that far, picks a person of average rank $\Omega(\sqrt{n})$.

Quite surprisingly, there is an algorithm that picks a person of average rank $O(1)$, and which is then competitive for the optimization problem of minimizing the rank. The algorithm is rather complicated, and it is based on first computing a series of timesteps $t_0 \leq t_1 \leq \ldots \leq t_k \leq \ldots$ according to a rather complicated formula, and then proceed as follows: we reject the first t_0 people, then if we find someone in the first t_1 dates which is better than all the previous people, we pick that person. Otherwise, between the $(t_1 + 1)$-th and the t_2-th date, we are willing to pick someone if that person is either the best or the second best of those seen so far. Between the $(t_2 + 1)$-th and t_3-th date, we become willing to pick anybody who is at least the *third-best* person seen so far, and so on. Basically, as time goes on, we become increasingly desperate, and we reduce our expectations accordingly.

17.3 Paging and Caching

The next problem that we study arises in any system that has hierarchical memory, that is, that has a larger but slower storage device and a faster but smaller one that can be used as cache. Consider for example the virtual memory paged on a disk and the real memory, or the content of a hard disk and the cache on the controller, or the RAM in a computer and the level-2 cache on the processor, or the level-2 and the level-1 cache, and so on.

All these applications can be modeled in the following way: there is a cache which is an array with k entries. Each entry contains a copy of an entry of a larger memory device, together with a pointer to the location of that entry. When we want to access a location of the larger device (a *request*), we first look up in the cache whether we have the content of that entry stored there. If so, we have a *hit*, and the access takes negligible time. Otherwise, we have a *miss*, and we need to fetch the entry from the slower large device. In negligible extra time, we can also copy the entry in the cache for later use. If the cache is already full,

however, we need to first delete one of the current cache entries in order to make room for the new one. Which one should we delete?

Here we have an online problem in which the data is the sequence of requests, the decisions of the algorithm are the entries to delete from the cache when it is full and there is a miss, and the cost function that we want to minimize is the number of misses. (Which determine the only non-negligible computational time.)

A reasonably good competitive algorithm is to remove the entry for which *the longest time has passed since the last request*. This is the Least Recently Used heuristic, or LRU.

Theorem 17.1 *Suppose that, for a certain sequence of requests, the optimal sequence of choices for a size-h cache causes m misses. Then, for the same sequence of requests, LRU for a size-k cache causes at most*

$$\frac{k}{k-h+1} \cdot m$$

misses.

This means that, for a size-k cache, LRU is k-competitive against an algorithm that knows the future and makes optimal choices. More interestingly, it says that if LRU caused m misses on a size-k cache on a certain sequence of requests, then, even an optimal algorithm that knew the future, would have caused at least $m/2$ misses using a size $k/2$ cache.

PROOF: Suppose the large memory device has size N, and so a sequence of requests is a sequence $a_1, \ldots, a_n$ of integers in the range $\{1, \ldots, N\}$. Let us divide the sequence into "phases" in the following way. Let t be the time at which we see the $(k+1)$-th new request. Then the first phase is $a_1, \ldots, a_{t-1}$. Next, consider the sequence $a_t, \ldots, a_n$, and recursively divide it into phases. For example, if $k = 3$ and we have the sequence of requests

$$35, 3, 12, 3, 3, 3, 12, 3, 21, 12, 35, 12, 4, 6, 3, 1, 12, 4, 12, 3$$

then the division into phases is

$$(35, 3, 12, 3, 3), (12, 3, 3, 12, 3, 21, 12), (35, 12, 4), (6, 3, 1), (12, 4, 12, 3)$$

In each phase, LRU causes k misses.

Consider now an arbitrary other algorithm, operating with a size-h cache, and consider its behavior over a sequence of phases which is like the above one, but with the first item in each phase moved to the previous phase

$$(35, 3, 12, 3, 3, 12), (3, 3, 12, 3, 21, 12, 35), (12, 4, 6), (3, 1, 12), (4, 12, 3)$$

In the first phase, we have $k + 1$ distinct values, and so we definitely have at least $k + 1$ misses starting with an empty cache, no matter what the algorithm does. At the beginning

of each subsequent phase, we know that the algorithm has in the cache the last request of the previous phase, and then we do not know what is in the remaining $h-1$ entries. We know, however, that we are going to see k distinct requests which are different from the last request, and so at least $k-h+1$ of them must be out of the cache and must cause a miss. So even an optimal algorithm causes at least $k-h+1$ misses per phase, compared with the k misses per phase of LRU, hence the competitive ratio.

(Note: we are glossing over the issue of what happens in the last phase, if the last phase has less than k distinct requests, in which case it could happen that the optimal algorithm has zero misses and LRU has a positive number of misses. In that case, we use the "surplus" that we have in the analysis about the first phase, in which the optimum algorithm and LRU have both k misses.) □

It can be proved that, if we knew the sequence of requests, then the optimal algorithm is to take out of the cache the element *whose next request is further in future*. The LRU algorithm is motivated by the heuristic that the element that has not been used for the longest time is likely to also not be needed for the longest time. It is remarkable, however, that such a heuristic works well even in a worst-case analysis.

Lecture 18

Using Expert Advice

In which we show how to use expert advice, and introduce the powerful "multiplicative weight" algorithm.

We study the following online problem. We have n "experts" that, at each time step $t = 1, \ldots, T$, suggest a strategy about what to do at that time (for example, they might be advising on what technology to use, on what investments to make, they might make predictions on whether something is going to happen, thus requiring certain actions, and so on). Based on the quality of the advice that the experts offered in the past, we decide which advice to follow, or with what fraction of our investment to follow which strategy. Subsequently, we find out which loss or gain was associated to each strategy, and, in particular, what loss or gain we personally incurred with the strategy or mix of strategies that we picked, and we move to step $t + 1$.

We want to come up with an algorithm to use the expert advice such that, at the end, that is, at time T, we are about as well off as if we had known in advance which expert was the one that gave the best advice, and we had always followed the strategy suggested by that expert at each step. Note that we make no probabilistic assumption, and our analysis will be a worst-case analysis over all possible sequences of events.

The "multiplicative update" algorithm provides a very good solution to this problem, and the analysis of this algorithm is a model for the several other applications of this algorithm, in rather different contexts.

18.1 A Simplified Setting

We begin with the following simplified setting: at each time step, we have to make a prediction about an event that has two possible outcomes, and we can use the advice of n "experts," which make predictions about the outcome at each step. Without knowing anything about the reliability of the experts, and without making any probabilistic assumption on the outcomes, we want to come up with a strategy that will lead us to make not much more mistakes than the "offline optimal" strategy of picking the expert which makes the

fewest mistakes, and then always following the prediction of that optimal expert.

The algorithm works as follows: at each step t, it assigns a *weight* w_i^t to each expert i, which measures the confidence that the algorithm has in the validity of the prediction of the expert. Initially, $w_i^0 = 1$ for all experts i. Then the algorithm makes the prediction that is backed by the set of experts with largest total weight. For example, if the experts, and us, are trying to predict whether the following day it will rain or not, we will look at the sum of the weights of the experts that say it will rain, and the sum of the weights of the experts that say it will not, and then we agree with whichever prediction has the largest sum of weights. After the outcome is revealed, we divide by 2 the weight of the experts that were wrong, and leave the weight of the experts that were correct unchanged.

We now formalize the above algorithm in pseudocode. We use $\{a, b\}$ to denote the two possible outcomes of the event that we are required to predict at each step.

- for each $i \in \{1, \dots, n\}$ do $w_i^1 := 1$
- for each time $t \in \{1, \dots, T\}$
  - let $w^t := \sum_i w_i^t$
  - if the sum of w_i^t over all the experts i that predict a is $\geq w^t/2$, then predict a
  - else predict b
  - *wait until the outcome is revealed*
  - for each $i \in \{1, \dots, n\}$
    - if i was wrong then $w_i^{t+1} := w_i^t/2$

To analyze the algorithm, let m_i^t be the indicator variable that expert i was wrong at time t, that is, $m_i^t = 1$ if the expert i was wrong at time i and $m_i^t = 0$ otherwise. (Here m stands for "mistake.") Let $m_i = \sum_{t=1}^T m_i^t$ be the total number of mistakes made by expert i. Let m_A^t be the indicator variable that our algorithm makes a mistake at time t, and $m_A := \sum_{t=1}^T m_A^t$ be the total number of mistakes made by our algorithm.

We make the following two observations:

1. If the algorithm makes a mistake at time t, then the total weight of the experts that are mistaken at time t is $\geq w^t/2$, and, at the following step, the weight of those experts is divided by two, and this means that, if we make a mistake at time t then

$$w^{t+1} \leq \frac{3}{4} w^t$$

Because the initial total weight is $w^1 = n$, we have that, at the end,

$$w^{T+1} \leq \left(\frac{3}{4}\right)^{m_A} \cdot n$$

2. For each expert i, the final weight is $w_i^{T+1} = 2^{-m_i}$, and, clearly,

$$\frac{1}{2^{m_i}} = w_i^{T+1} \leq w^{T+1}$$

Together, the two previous observations mean that, for every expert i,

$$\frac{1}{2^{m_i}} \leq \left(\frac{3}{4}\right)^{m_A} \cdot n$$

which means that, for every expert i,

$$m_A \leq O(m_i + \log n)$$

That is, the number of mistakes made by the algorithm is at most a constant times the number of mistakes of the best expert, plus an extra $O(\log n)$ mistakes.

We will now discuss an algorithm that improves the above result in two ways. We will show that, for every ϵ, the improved algorithm we can make the number of mistakes be at most $(1+\epsilon)m_i + O\left(\frac{1}{\epsilon}\log n\right)$ for every ϵ, which can be seen to be optimal for small n, and the improved algorithm will be able to handle a more general problem, in which the experts are suggesting arbitrary strategies, and the outcome of each strategy can be an arbitrary gain or loss.

18.2 The General Result

We now consider the following model. At each time step t, each expert i suggests a certain strategy. We choose to follow the advice of expert i with probability p_i^t, or, equivalently, we allocate a p_i^t fraction of our resources in the way expert i advised. Then we observe the outcome of the strategies suggested by the experts, and of our own strategy. We call m_i^t the *loss* incurred by following the advice of expert i. The loss can be negative, in which case it is a *gain*, and we normalize losses and gains so that $m_i^t \in [-1, 1]$ for every i and every t. Our own loss for the time step t will then be $\sum_i p_i^t m_i^t$. At the end, we would like to say that our own sum of losses is not much higher than the sum of losses of the best expert.

As before, our algorithm maintains a *weight* for each expert, corresponding to our confidence in the expert. The weights are initialized to 1. When an expert causes a loss, we reduce his weight, and when an expert causes a gain, we increase his weight. To express the weight updated in a single instruction, we have $w_i^{t+1} := (1 - \epsilon m_i^t) \cdot w_i^t$, where $0 < \epsilon < 1/2$ is a parameter of our choice. Our probabilities p_i^t are chosen proportionally to weights w_i^t.

- for each $i \in \{1, \ldots, n\}$ do $w_i^1 := 1$
- for each time $t \in \{1, \ldots, T\}$
  - let $w^t := \sum_i w_i^t$
  - let $p_i^t := w_i^t / w^t$
  - for each i, follow the strategy of expert i with probability p_i^t
  - *wait until the outcome is revealed*
  - let m_i^t be the loss of the strategy of expert i
  - for each $i \in \{1, \ldots, n\}$
    - $w_i^{t+1} := (1 - \epsilon \cdot m_i^t) \cdot w_i^t$

To analyze the algorithm, we will need the following technical result.

Fact 18.1 *For every $\epsilon \in [-1/2, 1/2]$,*

$$e^{\epsilon - \epsilon^2} \leq 1 + \epsilon \leq e^{\epsilon}$$

PROOF: We will use the Taylor expansion

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \cdots$$

1. *The upper bound.* The Taylor expansion above can be seen as $e^x = 1 + x + \sum_{t=1}^{\infty} x^{2t} \cdot \left(\frac{1}{(2t)!} + \frac{x}{(2t+1)!} \right)$, that is, e^x equals $1 + x$ plus a sum of terms that are all non-negative when $x \geq -1$. Thus, in particular, we have $1 + \epsilon \leq e^{\epsilon}$ for $\epsilon \in [-1/2, 1/2]$.

2. *The lower bound for positive ϵ.* We can also see that, for $x \in [0, 1]$, we have

   $$e^x \leq 1 + x + x^2$$

   and so, for $\epsilon \in [0, 1]$ we have

   $$e^{\epsilon - \epsilon^2} \leq 1 + \epsilon - \epsilon^2 + \epsilon^2 - 2\epsilon^3 + \epsilon^4 \leq 1 + \epsilon$$

3. *The lower bound for negative ϵ.* Finally, for $x \in [-1, 0]$ we have

   $$e^x = 1 + x + \frac{x^2}{2} + \sum_{t=1}^{\infty} x^{2t+1} \left(\frac{1}{(2t+1)!} + \frac{x}{(2t+2)!} \right) \leq 1 + x + \frac{x^2}{2}$$

   and so, for $\epsilon \in [-1/2, 0]$ we have

   $$e^{\epsilon - \epsilon^2} \leq 1 + \epsilon - \epsilon^2 + \frac{1}{2}\epsilon^2 - \epsilon^3 + \frac{1}{4}\epsilon^4 \leq 1 + \epsilon$$

□

Now the analysis proceeds very similarly to the analysis in the previous section. We let

$$m_A^t := \sum_i p_i^t m_i^t$$

be the loss of the algorithm at time t, and $m_A := \sum_{t=1}^T m_A^t$ the total loss at the end. We denote by $m_i := \sum_{t=1}^T m_i^t$ the total loss of expert i.
If we look at the total weight at time $t+1$, it is

$$w^{t+1} = \sum_i w_i^{t+1} = \sum_i (1 - \epsilon m_i^t) \cdot w_i^t$$

and we can rewrite it as

$$w^{t+1} = w^t - \sum_i \epsilon m_i^t w_i^t = w^t - w^t \epsilon \cdot \sum_i m_i^t p_i^t = w^t \cdot (1 - \epsilon m_A^t)$$

Recalling that, initially, $w^1 = n$, we have that the total weight at the end is

$$w^{T+1} = n \cdot \prod_{t=1}^T (1 - \epsilon m_A^t)$$

For each expert i, the weight of that expert at the end is

$$w_i^{T+1} = \prod_{t=1}^T (1 - \epsilon m_i^t)$$

and, as before, we note that for every expert i we have

$$w_i^{T+1} \leq w^{T+1}$$

Putting everything together, for every expert i we have

$$\prod_{t=1}^T (1 - \epsilon m_i^t) \leq n \cdot \prod_{t=1}^T (1 - \epsilon m_A^t)$$

Now it is just a matter of taking logarithms and of using the inequality that we proved before.

$$\ln \prod_{t=1}^T (1 - \epsilon m_A^t) = \sum_{i=1}^T \ln 1 - \epsilon m_A^t \leq - \sum_{i=1}^T \epsilon m_A^t = -\epsilon m_A$$

$$\ln \prod_{t=1}^{T}(1-\epsilon m_i^t) = \sum_{t=1}^{T} \ln 1 - \epsilon m_i^T \geq \sum_{t=1}^{T} -\epsilon m_i^t - \epsilon^2 (m_i^t)^2$$

and, overall,

$$m_A \leq +m_i + \epsilon \sum_{i=1}^{T} |m_i^t| + \frac{\ln n}{\epsilon} \tag{18.1}$$

In the model of the previous section, at every step the loss of each expert is either 0 or 1, and so the above expression simplifies to

$$m_A \leq (1+\epsilon) m_i + \frac{\ln n}{\epsilon}$$

which shows that we can get arbitrarily close to the best expert.

In every case, (18.1) simplifies to

$$m_A \leq m_i + \epsilon T + \frac{\ln n}{\epsilon}$$

and, if we choose $\epsilon = \sqrt{\ln n / T}$, we have

$$m_A \leq m_i + 2\sqrt{T \ln n}$$

which means that we come close to the optimum up to a small *additive* error.

To see that this is essentially the best that we can hope for, consider a playing a fair roulette game as follows: for T times, we either bet \$1 on red or \$1 on black. If we win we win \$1, and if we lose we lose \$1; we win and lose with probability 1/2 each at each step. Clearly, for every betting strategy, our expected win at the end is 0. We can think of the problem as there being two experts: the *red* expert always advises to bet red, and the *black* expert always advises to bet black. For each run of the game, the strategy of always following the best expert has a non-negative gain and, on average, following the best expert has a gain of $\Omega(\sqrt{T})$, because there is $\Omega(1)$ probability that the best expert has a gain of $\Omega(\sqrt{T})$. This means that we cannot hope to always achieve at least the gain of the best expert minus $o(\sqrt{T})$, even in a setting with 2 experts.

18.3 Applications

The general expert setting is very similar to a model of investments in which the experts correspond to stocks (or other investment vehicles) and the outcomes correspond to the variation in value of the stocks. The difference is that in our model we "invest" one unit of money at each step regardless of what happened in previous steps, while in investment

strategies we compound our gains (and losses). If we look at the logarithm of the value of our investment, however, it is modeled correctly by the experts setting.

The multiplicative update algorithm that we described in the previous section arises in several other contexts, with a similar, or even identical, analysis. For example, it arises in the context of *boosting* in machine learning, and it leads to efficient approximate algorithms for certain special cases of linear programming.