Cornell University, Fall 2016	CS 6820: Algorithms
Lecture notes: Matchings	24 Aug–2 Sep

These notes analyze algorithms for optimization problems involving matchings in bipartite graphs. Matching algorithms are not only useful in their own right (e.g., for matching clients to servers in a network, or buyers to sellers in a market) but also furnish a concrete starting point for learning many of the recurring themes in the theory of graph algorithms and algorithms in general. Examples of such themes are augmenting paths, linear programming relaxations, and primal-dual algorithm design.

1 Bipartite maximum matching

In this section we introduce the bipartite maximum matching problem, present a naïve algorithm with $O(mn)$ running time, and then present and analyze an algorithm due to Hopcroft and Karp that improves the running time to $O(m\sqrt{n})$.

1.1 Definitions

Definition 1. A *matching* in an undirected graph is a set of edges such that no vertex belongs to more than element of the set.

When we write a bipartite graph G as an ordered triple $G = (U, V, E)$, the notation means that U and V are disjoint sets constituting a partition of the vertices of G, and that every edge of G has one endpoint (the *left endpoint*) in U and the other endpoint (the *right endpoint*) in V.

Definition 2. The *bipartite maximum matching problem* is the problem of computing a matching of maximum cardinality in a bipartite graph.

We will assume that the input to the bipartite maximum matching problem, $G = (U, V, E)$, is given in its adjacency list representation, and that the bipartition of G—that is, the partition of the vertex set into U and V—is given as part of the input to the problem.

Exercise 1. Prove that if the bipartition is not given as part of the input, it can be constructed from the adjacency list representation of G in linear time.

(Here and elsewhere in the lecture notes for CS 6820, we will present exercises that may improve your understanding. You are encouraged to attempt to solve these exercises, but they are not homework problems and we will make no effort to check if you have solved them, much less grade your solutions.)

1.2 Alternating paths and cycles; augmenting paths

The following sequence of definitions builds up to the notion of an *augmenting path*, which plays a central role in the design of algorithms for the bipartite maximum matching problem.

Definition 3. If G is a graph and M is a matching in G, a vertex is called *matched* if it belongs to one of the edges in M, and *free* otherwise.

An *alternating component with respect to* M (also called an M*-alternating component*) is an edge set that forms a connected subgraph of G of maximum degree 2 (i.e., a path or cycle), in which every degree-2 vertex belongs to exactly one edge of M. An *augmenting path with respect to* M is an M-alternating component which is a path both of whose endpoints are free vertices.

In the following lemma, and throughout these notes, we use the notation $A \oplus B$ to denote the *symmetric difference* of two sets A and B, i.e. the set of all elements that belong to one of the sets but not the other.

Lemma 1. *If M is a matching and P is an augmenting path with respect to M, then $M \oplus P$ is a matching containing one more edge than M.*

Proof. P has an odd number of edges, and its edges alternate between belonging to M and its complement, starting and ending with the latter. Therefore, $M \oplus P$ has one more edge than M. To see that it is a matching, note that vertices in the complement of P have the same set of neighbors in M as in $M \oplus P$, and vertices in P have exactly one neighbor in $M \oplus P$. □

Lemma 2. *A matching M in a graph G is a maximum cardinality matching if and only if it has no augmenting path.*

Proof. We have seen in Lemma 1 that if M has an augmenting path, then it does not have maximum cardinality, so we need only prove the converse. Suppose that M^* is a matching of maximum cardinality and that $|M| < |M^*|$. The edge set $M \oplus M^*$ has maximum degree 2, and each vertex of degree 2 in $M \oplus M^*$ belongs to exactly one edge of M. Therefore each connected component of $M \oplus M^*$ is an M-alternating component. At least one such component must contain more edges of M^* than of M. It cannot be an alternating cycle or an even-length alternating path; these have an equal number of edges of M^* and M. It also cannot be an odd-length alternating path that starts and ends in M. Therefore it must be an odd-length alternating path that starts and ends in M^*. Since both endpoints of this path are free with respect to M, it is an M-augmenting path as desired. □

1.3 Bipartite maximum matching: Naïve algorithm

The foregoing discussion suggests the following general scheme for designing a bipartite maximum matching algorithm.

Algorithm 1 Naïve iterative scheme for computing a maximum matching

1: Initialize $M = \emptyset$.
2: **repeat**
3: Find an augmenting path P with respect to M.
4: $M \leftarrow M \oplus P$
5: **until** there is no augmenting with respect to M.

By Lemma 1, the invariant that M is a matching is preserved at the end of each loop iteration. Furthermore, each loop iteration increases the cardinality of M by 1, and the cardinality cannot exceed $n/2$, where n is the number of vertices of G. Therefore, the algorithm terminates after at most $n/2$ iterations. When it terminates, M is guaranteed to be a maximum matching by Lemma 2.

The algorithm is not yet fully specified because we have not indicated the procedure for finding an augmenting path with respect to M. When G is a bipartite graph, there is a simple linear-time procedure that we now describe.

Definition 4. If $G = (U, V, E)$ is a bipartite graph and M is a matching, the graph $D(G, M)$ is the directed graph formed from G by orienting each edge from U to V if it does not belong to M, and from V to U otherwise.

Lemma 3. *Suppose M is a matching in a bipartite graph G, and let F denote the set of free vertices. M-augmenting paths are in one-to-one correspondence with directed paths from $U \cap F$ to $V \cap F$ in $D(G, M)$.*

Proof. If P is a directed path from $U \cap F$ to $V \cap F$ in $D(G, M)$ then P starts and ends at free vertices, and its edges alternate between those that are directed from U to V (which are in the complement of M) and those that are directed from V to U (which are in M), so the undirected edge set corresponding to P is an augmenting path.

Conversely, if P is an augmenting path, then each vertex in the interior of P belongs to exactly one edge of M, so when we orient the edges of P as in $D(G, M)$ each vertex in the interior of P has exactly one incoming and one outgoing edge, i.e. P becomes a directed path. This path has an odd number of edges so it has one endpoint in U and the other endpoint in V. Both of these endpoints belong to F, by the definition of augmenting paths. Thus, the directed edge set corresponding to P is a path in $D(G, M)$ from $U \cap F$ to $V \cap F$. □

Lemma 3 implies that in each loop iteration of Algorithm 1, the step that requires finding an augmenting path (if one exists) can be implemented by building the auxiliary graph $D(G, M)$ and running a graph search algorithm such as BFS or DFS to search for a path from $U \cap F$ to $V \cap F$. Building $D(G, M)$ takes $O(m + n)$ time, where m is the number of edges in G, as does searching $D(G, M)$ using BFS or DFS. For convenience, assume $m \geq n/2$; otherwise G contains isolated vertices which may be eliminated in a preprocessing step requiring only $O(n)$ time. Then Algorithm 1 runs for at most $n/2$ iterations, each requiring $O(m)$ time, so its running time is $O(mn)$.

Remark 1. When G is not bipartite, our analysis of Algorithm 1 still proves that it finds a maximum matching after at most $n/2$ iterations. However, the task of finding an augmenting

path, if one exists, is much more subtle. The first polynomial-time algorithm for finding an augmenting path was discovered by Jack Edmonds in a 1965 paper entitled "Paths, Trees, and Flowers" that is one of the most influential papers in the history of combinatorial optimization. Edmonds' algorithm finds an augmenting path in $O(mn)$ time, leading to a running time of $O(mn^2)$ for finding a maximum matching in a non-bipartite graph. Faster algorithms have subsequently been discovered.

1.4 The Hopcroft-Karp algorithm

One potentially wasteful aspeect of the naïve algorithm for bipartite maximum matching is that it chooses one augmenting path in each iteration, even if it finds many augmenting paths in the process of searching the auxiliary graph $D(G, M)$. The Hopcroft-Karp algorithm improves the running time of the naïve algorithm by correcting this wasteful aspect; in each iteration it attempts to find many disjoint augmenting paths, and it uses all of them to increase the size of M.

The following definition specifies the type of structure that the algorithm searches for in each iteration.

Definition 5. If G is a graph and M is a maximum matching, a *blocking set of augmenting paths* with respect to M is a set $\{P_1, \ldots, P_k\}$ of augmenting paths such that:

1. the paths $P_1, \ldots, P_k$ are vertex disjoint;
2. they all have the same length, ℓ;
3. ℓ is the minimum length of an M-augmenting path;
4. every augmenting path of length ℓ has at least one vertex in common with $P_1 \cup \cdots \cup P_k$.

In other words, a blocking set of augmenting paths is a (setwise) maximal collection of vertex-disjoint minimum-length augmenting paths.

The following lemma generalizes Lemma 1 and its proof is a direct generalization of the proof of that lemma.

Lemma 4. *If M is a matching and $\{P_1, \ldots, P_k\}$ is any set of vertex-disjoint M-augmenting paths then $M \oplus P_1 \oplus P_2 \oplus \cdots \oplus P_k$ is a matching of cardinality $|M| + k$.*

Generalizing Lemma 2 we have the following.

Lemma 5. *Suppose G is a graph, M is a matching in G, and M^* is a maximum matching; let $k = |M^*| - |M|$. The edge set $M \oplus M^*$ contains at least k vertex-disjoint M-augmenting paths. Consequently, G has at least one M-augmenting path of length less than n/k, where n denotes the number of vertices of G.*

Proof. The edge set $M \oplus M^*$ has maximum degree 2, and each vertex of degree 2 in $M \oplus M^*$ belongs to exactly one edge of M. Therefore each connected component of $M \oplus M^*$ is an M-alternating component. Each M-alternating component which is *not* an augmenting path has at least as many edges in M as in M^*. Each M-augmenting path has exactly one fewer edge in M as in M^*. Therefore, at least k of the connected components of $M \oplus M^*$ must

be M-augmenting paths, and they are all vertex-disjoint. To prove the final sentence of the lemma, note that G has only n vertices, so it cannot have k disjoint subgraphs each with more than n/k vertices. □

These lemmas suggest the following method for finding a maximum matching in a graph, which constitutes the outer loop of the Hopcroft-Karp algorithm.

Algorithm 2 Hopcroft-Karp algorithm, outer loop

1: $M = \emptyset$
2: **repeat**
3: Let $\{P_1, \ldots, P_k\}$ be a blocking set of augmenting paths with respect to M.
4: $M \leftarrow M \oplus P_1 \oplus P_2 \oplus \cdots \oplus P_k$
5: **until** there is no augmenting path with respect to M

The key to the improved running-time guarantee is the following pair of lemmas which culminate in an improved bound on the number of outer-loop iterations.

Lemma 6. *The minimum length of an M-augmenting path strictly increases after each iteration of the Hopcroft-Karp outer loop in which a non-empty blocking set of augmenting paths is found.*

Proof. We will use the following notation.

$$\begin{aligned} M &= \text{matching at the start of one loop iteration} \\ P_1, \ldots, P_k &= \text{blocking set of augmenting paths found} \\ Q &= P_1 \cup \cdots \cup P_k \\ R &= E \setminus Q \\ M' &= M \oplus Q = \text{matching at the end of the iteration} \\ F &= \{\text{vertices that are free with respect to } M\} \\ F' &= \{\text{vertices that are free with respect to } M'\} \\ d(v) &= \text{length of shortest path in } D(G, M) \text{ from } U \cap F \text{ to } v \\ &\quad (\text{If no such path exists, } d(v) = \infty.) \end{aligned}$$

If (x, y) is any edge of $D(G, M)$ then $d(y) \leq d(x) + 1$. Edges of $D(G, M)$ that satisfy $d(y) = d(x) + 1$ will be called *advancing* edges, and all other edges will be called *retreating* edges. Note that a shortest path in $D(G, M)$ from $U \cap F$ to any vertex v must be formed entirely from advancing edges. In particular, Q is contained in the set of advancing edges.

In the edge set of $D(G, M')$, the orientation of every edge in Q is reversed and the orientation of every edge in R is preserved. Therefore, $D(G, M')$ has three types of directed edges (x, y):

1. reversed edges of Q, which satisfy $d(y) = d(x) - 1$;
2. advancing edges of R, which satisfy $d(y) = d(x) + 1$;
3. retreating edges of R, with satisfy $d(y) \leq d(x)$.

Note that in all three cases, the inequality $d(y) \leq d(x) + 1$ is satisfied.

Now let ℓ denote the minimum length of an augmenting path with respect to M, i.e. $\ell = \min\{d(v) \mid v \in V \cap F\}$. Let P be any path in $D(G, M')$ from $U \cap F'$ to $V \cap F'$. The lemma asserts that P has at least ℓ edges. The endpoints of P are free in M', hence also in M. As w ranges over the vertices of P, the value $d(w)$ increases from 0 to at least ℓ, and each edge of P increases the value of $d(w)$ by at most 1. Therefore P has at least ℓ edges, and the only way that it can have ℓ edges is if $d(y) = d(x) + 1$ for each edge (x, y) of P. We have seen that this implies that P is contained in the set of advancing edges of R, and in particular P is edge-disjoint from Q. It cannot be vertex-disjoint from Q because then $\{P_1, \ldots, P_k, P\}$ would be a set of $k+1$ vertex-disjoint minimum-length M-augmenting paths, violating our assumption that $\{P_1, \ldots, P_k\}$ is a blocking set. Therefore P has at least one vertex in common with $P_1, \ldots, P_k$, i.e. $P \cap Q \neq \emptyset$. The endpoints of P cannot belong to Q, because they are free in M' whereas every vertex in Q is matched in M'. Let w be a vertex in the interior of P which belongs to Q. The edge of M' containing w belongs to P, but it also belongs to Q. This violates our earlier conclusion that P is edge-disjoint from Q, yielding the desired contradiction. □

Lemma 7. *The Hopcroft-Karp algorithm terminates after fewer than $2\sqrt{n}$ iterations of its outer loop.*

Proof. After the first $\sqrt{n}$ iterations of the outer loop are complete, the minimum length of an M-augmenting path is greater than $\sqrt{n}$. This implies, by Lemma 5, that $|M^*| - |M| < \sqrt{n}$, where M^* denotes a maximum cardinality matching. Each remaining iteration strictly increases $|M|$, hence there are fewer than $\sqrt{n}$ iterations remaining. □

The inner loop of the Hopcroft-Karp algorithm must compute a blocking set of augmenting paths with respect to M. We now describe how to do this in linear time.

Recalling the distance labels $d(v)$ defined in the proof of Lemma 6; $d(v)$ is the length of the shortest alternating path from a free vertex in U to v; if no such path exists $d(v) = \infty$. Recall also that an *advancing* edge in $D(G, M)$ is an edge (x, y) such that $d(y) = d(x) + 1$, and that every minimum-length M-augmenting path is composed exclusively of advancing edges. The Hopcroft-Karp inner loop begins by performing a breadth-first search to compute the distance labels $d(v)$, along with the set A of advancing edges and a counter $c(v)$ for each vertex that counts the number of incoming advancing edges at v, i.e. advancing edges of the form (u, v) for some vertex u. It sets ℓ to be the minimum length of an M-augmenting path (equivalently, the minimum of $d(v)$ over all $v \in V \cap F$), marks every vertex as unexplored, and repeatedly finds augmenting paths using the following procedure. Start at an unexplored vertex v in $V \cap F$ such that $d(v) = \ell$, and trace backward along incoming edges in A until a vertex u with $d(u) = 0$ is reached. Add this path P to the blocking set and add its vertices to a "garbage collection" queue. While the garbage collection queue is non-empty, remove the vertex v at the head of the queue, mark it as explored, and delete its incident edges (both outgoing and incoming) from A. When deleting an outgoing edge (v, w), decrement the counter $c(w)$, and if $c(w)$ is now equal to 0, then add u to the garbage collection queue.

The inner loop performs only a constant number of operations per edge — traversing it during the BFS that creates the set A, traversing it while creating the blocking set of paths,

deleting it from A during garbage collection, and decrementing its tail's counter during garbage collection — and a constant number of operations per vertex: visiting it during the BFS that creates the set A, initializing $d(v)$ and $c(v)$, visiting it during the search for the blocking set of paths, marking it as explored, inserting it into the garbage collection queue, and removing it from that queue. Therefore, the entire inner loop runs in linear time.

By design, the algorithm discovers a set of minimum-length M-augmenting paths that are vertex disjoint, so we need only prove that this set is maximal. By induction on the number of augmenting paths the algorithm has discovered, the following invariants hold whenever the garbage collection queue is empty.

1. For every vertex v, $c(v)$ counts the number of advancing edges (u, v) that have not yet been deleted from A.

2. Whenever an edge e is deleted or a vertex v is placed into the garbage collection queue, any path made up of advancing edges that starts in $U \cap F$ and includes edge e or vertex v must have a vertex in common with the selected set of paths.

3. For every unmarked vertex v, $c(v) > 0$ and there exists a path in A from $U \cap F$ to v. (The existence of such a path follows by tracing backwards along edges of A from v to a vertex u such that $d(u) = 0$.)

The third invariant ensures that whenever the algorithm starts searching for an augmenting path at an unmarked free vertex, it is guaranteed to find such a path. The second invariant ensures that when there are no longer any unmarked free vertices v with $d(v) = \ell$, the set of advancing edges no longer contains a path from $U \cap F$ to $V \cap F$ that is vertex-disjoint from the selected ones; thus, the selected set forms a blocking set of augmenting paths as desired.

2 Bipartite min-cost perfect matching and its LP relaxation

In the bipartite minimum-cost perfect matching problem, we are given an undirected bipartite graph $G = (U, V, E)$ as before, together with a (non-negative, real-valued) cost c_e for each edge $e \in E$. Let $c(u, v) = c_e$ if $e = (u, v)$ is an edge of G, and $c(u, v) = \infty$ otherwise. As always, let n denote the number of vertices and m the number of edges of G.

A perfect matching M can be described by a matrix (x_{uv}) of 0's and 1's, where $x_{uv} = 1$ if and only if $(u, v) \in M$. The sum of the entries in each row and column of this matrix equals 1, since each vertex belongs to exactly one element of M. Conversely, for any matrix with $\{0, 1\}$-valued entries, if each row sum and column sum is equal to 1, then the corresponding set of edges is a perfect matching. Thus, the bipartite minimum-cost matching problem can be expressed as follows.

$$\begin{array}{lll} \min & \sum_{u,v} c(u,v) x_{uy} & \\ \text{s.t.} & \sum_v x_{uv} = 1 & \forall u \\ & \sum_u x_{uv} = 1 & \forall v \\ & x_{uv} \in \{0,1\} & \forall u, v \end{array}$$

This is a discrete optimization problem because of the constraint that $x_{uv} \in \{0, 1\}$. Although we already know how to solve this discrete optimization problem in polynomial time, many other such problems are not known to have any polynomial-time solution. It's often both interesting and useful to consider what happens when we relax the constraint $x_{uv} \in \{0, 1\}$ to $x_{uv} \geq 0$, allowing the variables to take any non-negative real value. This turns the problem into a continuous optimization problem, in fact a *linear program*.

$$\begin{array}{lll} \min & \sum_{u,v} c(u,v) x_{uv} & \\ \text{s.t.} & \sum_v x_{uv} = 1 & \forall u \\ & \sum_u x_{uv} = 1 & \forall v \\ & x_{uv} \geq 0 & \forall u, v \end{array}$$

How should we think about a matrix of values x_{uv} satisfying the constraints of this linear program? We've seen that if the values are integers, then it represents a perfect matching. A general solution of this constraint set can be regarded as a *fractional perfect matching*. What does a fractional perfect matching look like? An example is illustrated in Figure 1. Is it possible that this fractional perfect matching achieves a lower cost than any perfect

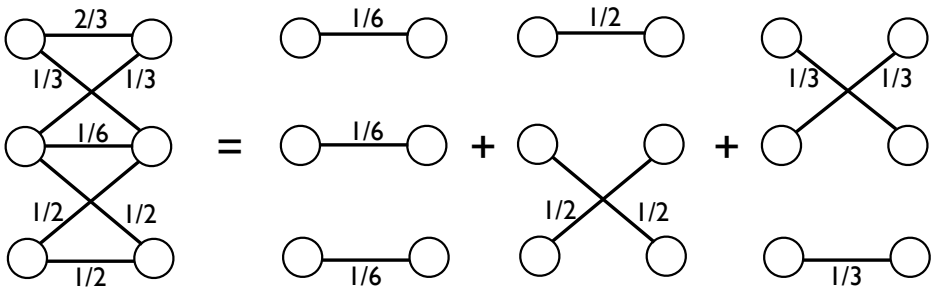

Figure 1: A fractional perfect matching.

matching? No, because it can be expressed as a convex combination of perfect matchings (again, see Figure 1) and consequently its cost is the weighted average of the costs of those perfect matchings. In particular, at least one of those perfect matchings costs no more than the fractional perfect matching illustrated on the left side of the figure. This state of affairs is not a coincidence. The *Birkhoff-von Neumann Theorem* asserts that every fractional perfect matching can be decomposed as a convex combination of perfect matchings. (Despite the eminence of its namesakes, the theorem is actually quite easy to prove. You should try finding a proof yourself, if you've never seen one.)

Now suppose we have an instance of bipartite minimum-cost perfect matching, and we want to prove a *lower bound* on the optimum: we want to prove that every fractional perfect matching has to cost at least a certain amount. How might we prove this? One way is to run a minimum-cost perfect matching algorithm, look at its output, and declare this to be a lower bound on the cost of any fractional perfect matching. (There exist polynomial-time algorithms for minimum-cost perfect matching, as we will see later in this lecture.) By the Birkhoff-von Neumann Theorem, this produces a valid lower bound, but it's not very satisfying. There's another, much more direct, way to prove lower bounds on the cost of

every fractional perfect matching, by directly combining constraints of the linear program. To illustrate this, consider the graph with edge costs as shown in Figure 2. Clearly, the

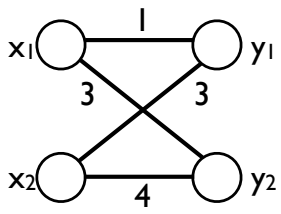

Figure 2: An instance of bipartite minimum cost perfect matching.

minimum cost perfect matching has cost 5. To prove that no fractional perfect matching can cost less than 5, we combine some constraints of the linear program as follows.

$$\begin{aligned} 2x_{11} + 2x_{21} &= 2 \\ -x_{11} - x_{12} &= -1 \\ 4x_{12} + 4x_{22} &= 4 \end{aligned}$$

Adding these constraints, we find that

$$x_{11} + 3x_{12} + 2x_{21} + 4x_{22} = 5 \quad (1)$$

$$x_{11} + 3x_{12} + 3x_{21} + 4x_{22} \geq 5 \quad (2)$$

Inequality (2) is derived from (1) because the only change we made on the left side was to increase the coefficient of x_{21} from 2 to 3, and we know that $x_{21} \geq 0$. The left side of (2) is the cost of the fractional perfect matching $\vec{m}$. We may conclude that the cost of every fractional perfect matching is at least 5.

What's the most general form of this technique? For every vertex $w \in U \cup V$, the linear program contains a "degree constraint" asserting that the degree of w in the fractional perfect matching is equal to 1. For each degree constraint, we multiply its left and right sides by some coefficient to obtain

$$\sum_v p_u x_{uv} = p_u$$

for some $u \in U$, or

$$\sum_u q_v x_{uv} = q_v$$

for some $v \in V$. Then we sum all of these equations, obtaining

$$\sum_{u,v} (p_u + q_v) x_{uv} = \sum_u p_u + \sum_v q_v. \quad (3)$$

If the inequality $p_u + q_v \leq c(u, v)$ holds for every $(u, v) \in U \times V$, then in the final step of the proof we (possibly) increase some of the coefficients on the left side of (3) to obtain

$$\sum_{u,v} c(u, v) x_{uv} \geq \sum_u p_u + \sum_v q_v,$$

thus obtaining a lower bound on the cost of every fractional perfect matching. This technique works whenever the coefficients p_u, q_v satisfy $p_u + q_v \leq c(x, y)$ for every edge (x, y), regardless of whether the values p_u, q_v are positive or negative. To obtain the strongest possible lower bound using this technique, we would set the coefficients p_u, q_v by solving the following linear program.

$$\begin{array}{ll} \max & \sum_u p_u + \sum_v q_v \\ \text{s.t.} & p_u + q_v \leq c(u, v) \quad \forall u, v \end{array}$$

This linear program is called the *dual* of the min-cost-fractional-matching linear program. We've seen that its optimum constitutes a lower bound on the optimum of the min-cost-fractional-matching LP. For any linear program, one can follow the same train of thought to develop a dual linear program. (There's also a formal way of specifying the procedure; it involves taking the transpose of the constraint matrix of the LP.) The dual of a minimization problem is a maximization problem, and its optimum constitutes a lower bound on the optimum of the minimization problem. This fact is called **weak duality**; as you've seen, weak duality is nothing more than an assertion that we can obtain valid inequalities by taking linear combinations of other valid inequalities, and that this sometimes allows us to bound the value of an LP solution from above or below. But actually, the optimum value of an LP is always *exactly equal* to the value of its dual LP! This fact is called **strong duality** (or sometimes simply "duality"), it is far from obvious, and it has important ramifications for algorithm design. In the special case of fractional perfect matching problems, strong duality says that the simple proof technique exemplified above is actually powerful enough to prove the *best possible* lower bound on the cost of fractional perfect matchings, for *every* instance of the bipartite min-cost perfect matching problem.

It turns out that there is a polynomial-time algorithm to solve linear programs. As you can imagine, this fact also has extremely important ramifications for algorithm design, but that's the topic of another lecture.

3 Primal-dual algorithm

In this section we will construct a fast algorithm for the bipartite minimum-cost perfect matching algorithm, exploiting insights gained from the preceding section. The basic plan of attack is as follows: we will design an algorithm that simultaneously computes two things: a minimum-cost perfect matching, and a dual solution (vector of p_u and q_v values) whose value (sum of p_u's and q_v's) equals the cost of the perfect matching. As the algorithm runs, it maintains the a dual solution $\vec{p}, \vec{q}$ and a matching M, and it preserves the following invariants:

1. Every edge (u, v) satisfies $p_u + q_v \leq c(u, v)$. If $p_u + q_v = c(u, v)$ we say that edge $e = (u, v)$ is *tight*.
2. The elements of M are a subset of the tight edges.
3. The cardinality of M increases by 1 in each phase of the algorithm, until it reaches n.

Assuming the algorithm can maintain these invariants until termination, its correctness will follow automatically. This is because the matching M at termination time will be a perfect matching satisfying

$$\sum_{(u,v)\in M} c(u,v) = \sum_{(u,v)\in M} p_u + q_v = \sum_{u\in U} p_u + \sum_{v\in V} q_v,$$

where the final equation holds because M is a perfect matching. The first invariant of the algorithm implies that $\vec{p}, \vec{q}$ is a feasible dual solution, hence the right side is a lower bound on the cost of any fractional perfect matching. The left side is the cost of the perfect matching M, hence M has the minimum cost of any fractional perfect matching.

So, how do we maintain the three invariants listed above while growing M to be a perfect matching? We initialize $M = \emptyset$ and $\vec{p} = \vec{q} = 0$. Note that the three invariants are trivially satisfied at initialization time. Now, as long as $|M| < n$, we want to find a way to either increase the value of the dual solution or enlarge M without violating any of the invariants. The easiest way to do this is to find an M-augmenting path P consisting of tight edges: in that case, we can update M to $M \oplus P$ without violating any invariants, and we reach the end of a phase. However, sometimes it's not possible to find an M-augmenting path consisting of tight edges: in that case, we must adjust some of the dual variables to make additional edges tight.

The process of adjusting dual variables is best described as follows. The easiest thing would be if we could find a vertex $u \in U$ that doesn't belong to any tight edges. Then we could raise p_u by some amount $\delta > 0$ until an edge containing u became tight. However, maybe every $u \in U$ belongs to a tight edge. In that case, we need to raise p_u by δ while lowering some other q_v by the same amount δ. This is best described in terms of a vertex set T which will have the property that if one endpoint of an edge $e \in M$ belongs to T, then both endpoints of e belong to T. Whenever T has this property, we can set

$$\delta = \min\{c(u,v) - p_u - q_v \,|\, u \in U \cap T, v \in V \setminus T\} \tag{4}$$

and adjust the dual variables by setting $p_u \leftarrow p_u + \delta, q_v \leftarrow q_v - \delta$ for all $u \in U \cap T, v \in V \cap T$. This preserves the feasibility of our dual solution $\vec{p}, \vec{q}$ (by the choice of δ) and it preserves the tightness of each edge $e \in M$ because every such edge has either both or neither of its endpoints in T.

Let F be the set of free vertices, i.e. those that don't belong to any element of M. T will be constructed by a sort of breadth-first search along tight edges, starting from the set $U \cap F$ of free vertices in U. We initialize $T = U \cap F$. Since $|M| < n$, T is nonempty. Define δ as in (4); if $\delta > 0$ then adjust dual variables as explained above. Call this a *dual adjustment step*. If $\delta = 0$ then there is at least one tight edge $e = (u, v)$ from $U \cap T$ to $V \setminus T$. If v is a free vertex, then we have discovered an augmenting path P consisting of tight edges (namely, P consists of a path in T that starts at a free vertex in U, walks to u, then crosses edge e to get to v) and we update M to $M \oplus P$ and finish the phase. Call this an *augmentation step*. Finally, if v is not a free vertex then we identify an edge $e = (u', v) \in M$ and we add both v and u' to T and call this a *T-growing step*. Notice that the left endpoint of an edge of M is always added to T at the same time as the right endpoint, which is why T never contains one endpoint of an edge of M unless it contains both.

A phase can contain at most n T-growing steps and at most one augmentation step. Also, there can never be two consecutive dual adjustment steps (since the value of δ drops to zero after the first such step) so the total number of steps in a phase is $O(n)$. Let's figure out the running time of one phase of the algorithm by breaking it down into its component parts.

1. There is only one augmentation step and it costs $O(n)$.
2. There are $O(n)$ T-growing steps and each costs $O(1)$.
3. There are $O(n)$ dual adjustment steps and each costs $O(n)$.
4. Finally, every step starts by computing the value δ using (4). Thus, the value of δ needs to be computed $O(n)$ times. Naïvely it costs $O(m)$ work each time we need to compute δ.

Thus, a naïve implementation of the primal-dual algorithm takes $O(mn^2)$.

However, we can do better using some clever book-keeping combined with efficient data structures. For a vertex $w \in T$, let $s(w)$ denote the number of the step in which w was added to T. Let δ_s denote the value of δ in step s of the phase, and let Δ_s denote the sum $\delta_1 + \cdots + \delta_s$. Let $p_{u,s}, q_{v,s}$ denote the values of the dual variables associated to vertices u, v at the end of step s. Note that

$$p_{u,s} = \begin{cases} p_{u,0} + \Delta_s - \Delta_{s(u)} & \text{if } u \in U \cap T \\ p_{u,0} & \text{if } u \in U \setminus T \end{cases} \tag{5}$$

$$q_{v,s} = \begin{cases} q_{v,0} - \Delta_s + \Delta_{s(v)} & \text{if } v \in V \cap T \\ q_{v,0} & \text{if } v \in V \setminus T \end{cases} \tag{6}$$

Consequently, if $e = (u, v)$ is any edge from $U \cap T$ to $V \setminus T$ at the end of step s, then

$$c(u, v) - p_{u,s} - q_{v,s} = c(u, v) - p_{u,0} - \Delta_s + \Delta_{s(u)} - q_{v,0}$$

The only term on the right side that depends on s is $-\Delta_s$, which is a global value that is common to all edges. Thus, choosing the edge that minimizes $c(u, v) - p_{u,s} - q_{v,s}$ is equivalent to choosing the edge that minimizes $c(u, v) - p_{u,0} + \Delta_{s(u)} - q_{v,0}$. Let us maintain a priority queue containing all the edges from $U \cap T$ to $V \setminus T$. An edge $e = (u, v)$ is inserted into this priority queue at the time its left endpoint u is inserted into T. The value associated to e in the priority queue is $c(u, v) - p_{u,0} + \Delta_{s(u)} - q_{v,0}$, and this value never changes as the phase proceeds. Whenever the algorithm needs to choose the edge that minimizes $c(u, v) - p_{v,s} - q_{u,s}$, it simply extracts the minimum element of this priority queue, repeating as necessary until it finds an edge whose right endpoint does not belong to T. The total amount of work expended on maintaining the priority queue throughout a phase is $O(m \log n)$.

Finally, our gimmick with the priority queue eliminates the need to actually update the values p_u, q_v during a dual adjustment step. These values are only needed for computing the value of δ_s, and for updating the dual solution at the end of the phase. However, if we

store the values $s(u), s(v)$ for all u, v as well as the values Δ_s for all s, then one can compute any specific value of $p_{u,s}$ or $q_{v,s}$ in constant time using (5)-(6). In particular, it takes $O(n)$ time to compute all the values p_u, q_v at the end of the phase, and it only takes $O(1)$ time to compute the value $\delta_s = c(u, v) - p_u - q_v$ once we have identified the edge $e = (u, v)$ using the priority queue. Thus, all the work to maintain the values p_u, q_v amounts to only $O(n)$ per phase.

In total, the amount of work in any phase is bounded by $O(m \log n)$ and consequently the algorithm's running time is $O(mn \log n)$.

Cornell University, Fall 2016	CS 6820: Algorithms
Lecture notes: Flows and Cuts	7 Sep–12 Sep

Network flows are a structure with many nice applications in algorithms and combinatorics. A famous result called the *max-flow min-cut theorem* exposes a tight relationship between network flows and graph cuts; the latter is also a fundamental topic in combinatorics and combinatorial optimization, with many important applications.

These notes introduce the topic of network flows, present and analyze some algorithms for computing a maximum flow, prove the max-flow min-cut theorem, and present some applications in combinatorics. There are also numerous applications of these topics elsewhere in computer science. For example, network flow has obvious applications to routing in communication networks. Algorithms for computing minimum cuts in graphs have important but less obvious applications in computer vision. Those applications (along with many other practical applications of maximum flows and minimum cuts) are beyond the scope of these notes.

1 Basic Definitions

We begin by defining flows in directed multigraphs. (A multigraph is a graph that is allowed to have parallel edges, i.e. two or more edges having the same endpoints.)

Definition 1. In a directed multigraph $G = (V, E)$, a *flow* with source s and sink t (where s and t are vertices of G) is an assignment of a non-negative value f_e to each edge e, called the "flow on e", such that for every $v \neq s, t$, the total flow on edges leaving v equals the total flow on edges entering v. This equation is called "flow conservation at v". The *value of the flow*, denoted by $|f|$, is the total amount of flow on edges leaving the source, s.

One can formulate a clean notation for re-expressing this definition using the *incidence matrix* of G, which is the matrix B with rows indexed by vertices, and columns indexed by edges, whose entries are defined as follows.

$$B_{we} = \begin{cases} 1 & \text{if } w \text{ is the head of } e, \text{ i.e. } e = (u, v) \text{ and } w = v \\ -1 & \text{if } w \text{ is the tail of } e, \text{ i.e. } e = (u, v) \text{ and } w = u \\ 0 & \text{otherwise} \end{cases}$$

For any vertex v let $\mathbf{1}_v$ denote the *indicator vector* of v, i.e. the column vector (with rows indexed by V) whose entries are defined as follows.

$$(\mathbf{1}_v)_w = \begin{cases} 1 & \text{if } v = w \\ 0 & \text{otherwise} \end{cases}$$

In this notation, if we interpret a flow f as a column vector whose rows are indexed by E, then a vector of non-negative numbers, f, is a flow from s to t if and only if $Bf = \lambda(\mathbf{1}_t - \mathbf{1}_s)$ for some scalar $\lambda \in \mathbb{R}$, in which case the value of f is given by $|f| = \lambda$.

A useful interpretation of flows is that "a flow is a weighted sum of source-sink paths and cycles".

Lemma 1. *For an edge set S let its* characteristic vector $\mathbf{1}_S$ *be the vector in $\mathbb{R}^E$ whose e^{th} entry equals 1 if $e \in S$, 0 if $e \notin S$. A vector $f \in \mathbb{R}^E$ is a flow from s to t if and only if f is equal to a weighted sum (with non-negative weights) of vectors $\mathbf{1}_S$ as S ranges over s-t paths, t-s paths, and directed cycles. The value of f is the combined weight of s-t paths minus the combined weight of t-s paths in any such weighted-sum decomposition.*

Proof. Let $\mathbb{R}_+$ denote the set of non-negative real numbers. When S is the edge set of (i) an s-t path, (ii) a t-s path, or (iii) a directed cycle, we have $\mathbf{1}_S \in \mathbb{R}_+$ and $B\mathbf{1}_S = \lambda_S(\mathbf{1}_t - \mathbf{1}_s)$ where λ_S equals 1 in case (i), -1 in case (ii), and 0 in case (iii), respectively. Taking weighted sums of these identities, this verifies that any non-negative weighted sum of source-sink paths and cycles is a flow with the stated value.

Conversely, if f is a flow we must prove that it is a non-negative weighted sum of source-sink paths and cycles. Let $E_+(f) = \{e \mid f_e > 0\}$. The proof will be by induction on the number of edges in $E_+(f)$. When this number is zero, the lemma holds vacuously, so assume $|E_+(f)| > 0$. If $E_+(f)$ contains an s-t path, a t-s path, or a directed cycle, then let S denote the edge set of this path or cycle, and let $w = \min\{f_e \mid e \in S\}$. The vector $g = f - w\mathbf{1}_S$ is a flow of value $|g| = |f| - w\lambda_S$, and $|E_+(g)| < |E_+(f)|$, so by the induction hypothesis we can decompose g as a weighted sum of s-t paths, t-s paths, and cycles, and $|g|$ is the combined weight of s-t paths minus the combined weight of t-s paths. The induction step then follows because $f = g + w\mathbf{1}_S$.

To complete the proof we need to show that when $|E_+(f)| > 0$ there is an s-t path, a t-s path, or a directed cycle contained in $E_+(f)$. If $E_+(f)$ does not contain a directed cycle then $(V, E_+(f))$ is a directed acyclic graph with non-empty edge set. As such, it must have a source vertex, i.e. a vertex u_0 with at least one outgoing edge, but no incoming edges. Construct a path $P = u_0, u_1, \ldots, u_k$ starting from u_0 and choosing u_i, for $i > 1$, by following an edge $(u_{i-1}, u_i) \in E_+(f)$. Since $E_+(f)$ contains no cycles this greedy path construction process must terminate at a vertex with no outgoing edges. Flow conservation implies that every vertex other than s and t which belongs to an edge in $E_+(f)$ has both incoming and outgoing edges. Therefore, the endpoints of P are s and t (in some order) which completes the proof that $E_+(f)$ has either a path joining the source to the sink (in some order) or a directed cycle. □

Definition 2. A *flow network* is a directed multigraph $G = (V, E)$ together with a non-negative *capacity* $c(e)$ for each edge e. A *valid flow* in a flow network is a flow f in G that satisfies the *capacity constraints* $f_e \leq c(e)$ for all edges e. A *maximum flow* is a valid flow of maximum value.

Maximum flow turns out to be a versatile problem that encodes many other algorithmic problems. For example, the maximum bipartite matching in a graph $G = (U, V, E)$ can be encoded by a flow network with vertex set $U \cup V \cup \{s, t\}$ and with edge set $(\{s\} \times U) \cup E \cup (V \times \{t\})$, all edges having capacity 1. For each edge $(u, v) \in E$, the flow network contains a three-hop path $P_e = \langle s, u, v, t\rangle$, and for any matching M in G one can sum up the characteristic vectors of the paths P_e $(e \in M)$ to obtain a valid flow f such that $|f| = |M|$.

Conversely, any valid flow f satisfying $f_e \in \mathbb{Z}$ for all e is obtained from a matching M via this construction. As we will see shortly, in any flow network with integer edge capacities, there always exists an integer-valued maximum flow. Thus, the bipartite maximum matching problem reduces to maximum flow via the simple reduction given in this paragraph.

The similarity between maximum flow and bipartite maximum matching also extends to the algorithms for solving them. The most basic algorithms for solving maximum flow revolve around a graph called the *residual graph* which is analogous to the directed graph $D(G, M)$ that we defined when presenting algorithms for the bipartite maximum matching problem.

Definition 3. Let $G = (V, E, c)$ is a flow network and f a valid flow in G. Let $\bar{E}$ denote a set containing a directed edge $\bar{e}$ for every $e \in E$, whose endpoints are the same as the endpoints of e but in the opposite order. If $e \in E$ and $\bar{e} \in \bar{E}$, the *residual capacities* $c_f(e), c_f(\bar{e})$ are defined by

$$\begin{aligned} c_f(e) &= c(e) - f_e \\ c_f(\bar{e}) &= f_e. \end{aligned}$$

The *residual graph* G_f is the flow network $G_f = (V, E_f, c_f)$, where E_f is the set of all edges in $E \cup \bar{E}$ with positive residual capacity. An *augmenting path* is a path from s to t in G_f.

To any valid flow h in G_f one can associate the vector $\pi(h) \in \mathbb{R}^E$ defined by

$$\pi(h)_e = h_e - h_{\bar{e}}.$$

The vectors $\pi(h)$ encode all the ways of modifying f to another valid flow in G.

Lemma 2. *If f is a valid flow in G and h is a valid flow in the residual graph G_f then $f + \pi(h)$ is a valid flow in G. Conversely, every valid flow in G can be expressed as $f + \pi(h)$ for some valid flow h in G_f.*

Proof. The equation $Bh = B\pi(h)$ follows from the definition of $\pi(h)$, and implies that $\pi(h)$ satisfies the flow conservation equations, hence $f + \pi(h)$ does as well. The residual capacity constraints in G_f are designed precisely to guarantee that the value of $f + \pi(h)$ on each edge e lies between 0 and $c(e)$, hence $f + \pi(h)$ is a valid flow in G. Conversely, suppose $\tilde{f}$ is any valid flow in G. Using the notation x^+ to denote $\max\{x, 0\}$ for any real number x, we may define

$$\begin{aligned} h_e &= (\tilde{f}_e - f_e)^+ \quad \text{for all } e \in E \\ h_{\bar{e}} &= (f_e - \tilde{f}_e)^+ \quad \text{for all } \bar{e} \in \bar{E} \end{aligned}$$

and verify that h is a valid flow in G_f satisfying $\tilde{f} = f + \pi(h)$. □

Lemma 3. *If f is a valid flow in G, then f is a maximum flow if and only if G_f does not contain an augmenting path.*

Proof. If f is not a maximum flow then let f^* be any maximum flow and write $f^* = f + \pi(h)$. Since $|f^*| = |f| + |h| > |f|$, we must have $|h| > 0$. According to Lemma 1, the flow h decomposes as a weighted sum of vectors $\mathbf{1}_S$ where S ranges over s-t paths, t-s paths, and directed cycles in G_f, and at least one s-t path must have a positive coefficient in this decomposition because $|h| > 0$. In particular, this implies that G_f contains an s-t path, i.e. an augmenting path. Conversely, if G_f contains an augmenting path P, let $\delta(P)$ be the minimum residual capacity of an edge of P. The flow $f + \delta(P)\pi(\mathbf{1}_P)$ is a valid flow with value $|f| + \delta(P)$, so f is not a maximum flow. □

1.1 Comparison with Other Definitions

The Kleinberg-Tardos textbook assumes that s has no incoming edges and t has no outgoing edges. In these notes we do not impose any such assumption. Consequently G may contain a path from t to s, leading to the somewhat counter-intuitive convention that a flow on a path from t to s is considered to be an s-t flow of negative value. This convention is useful, for example, in Lemma 3 where it allows us to simply say that if f and $\tilde{f}$ are two s-t flows in G, then their difference $\tilde{f} - f$ can always be represented by an s-t flow in the residual graph G_f.

The Kozen textbook represents a flow using a skew-symmetric matrix, whose (u, v) entry represents the difference $f_{uv} - f_{vu}$, i.e. the net flow from u to v along edges that join the two vertices directly. This allows for a beautifully simple formulation of the flow conservation equations and of the lemma that the difference between any two flows is represented by a flow in the residual graph. However, when the graph contains a two-cycle comprising edges (u, v) and (v, u), the representation of a flow as a skew-symmetric matrix eliminates the distinction between sending zero flow on (u, v) and (v, u) and sending an equal (but non-zero) amount in both directions. Philosophically, I believe these should be treated as distinct flows so I have opted for a definition that enables such a distinction. The cost of making this choice is that some definitions become messier and more opaque, especially those involving the residual graph and the function π that maps flows in G_f to flows in G.

2 The Max-Flow Min-Cut Theorem

One important corollary of Lemma 3 is the *max-flow min-cut theorem*, which establishes a tight relationship between maximum flows and cuts separating the source from the sink. We first present some definitions involving cuts, and then we present and prove the theorem.

Definition 4 (s-t cut)**.** An *s-t cut* in a directed graph $G = (V, E)$ with vertices s and t is a partition of the vertex set V into two subsets S, T such that $s \in S$ and $t \in T$. An edge $e = (u, v)$ *crosses* the cut (S, T) if $u \in S$ and $v \in T$. (Note that edges from T to S do not cross the cut (S, T), under this definition.) The capacity of cut (S, T), denoted by $c(S, T)$, is the sum of the capacities of all edges that cross the cut.

Theorem 4 (Max-flow min-cut)**.** *For any flow network, the value of any maximum flow is equal to the capacity of any minimum s-t cut.*

Proof. Let (S, T) be any s-t cut, and let $\mathbf{1}_T$ be the vector in $\mathbb{R}^V$ whose v^{th} component is 1 if $v \in T$, 0 if $v \notin T$. The row vector $x = \mathbf{1}_T^\intercal B$ satisfies $x_e = 1$ if e goes from S to T, $x_e = -1$ if e goes from T to S, and $x_e = 0$ otherwise.

For a flow f and two disjoint vertex sets Q, R, let $f(Q, R)$ denote the sum of f_e over all edges e going from Q to R. We have

$$\mathbf{1}_T^\intercal B f = xf = f(S, T) - f(T, S) \leq c(S, T) \tag{1}$$

where the last inequality is justified because $f_e \leq c(e)$ for all e from S to T, and $f_e \geq 0$ for all e from T to S. Since f is a flow, we have

$$\mathbf{1}_T^\intercal B f = \mathbf{1}_T^\intercal (\mathbf{1}_t - \mathbf{1}_s)|f| = |f|. \tag{2}$$

Combining equations (1) and (2) yields the conclusion that the value of any flow is bounded above by the capacity of any s-t cut; in particular, the min-cut capacity is an upper bound on the maximum flow value.

To prove that this upper bound is tight, first note that in our derivation of the inequality $|f| \leq c(S, T)$, the only step that was an inequality (rather than an equation) was the inequality at the end of line (1). Reviewing our justification for that inequality, one can see that the two sides are equal if $f_e = c(e)$ for all edges e from S to T and $f_e = 0$ for all edges e from T to S. When f is a maximum flow, we can find a cut (S, T) that satisfies these properties by applying Lemma 3, which says that there is no s-t path in the residual graph G_f. Define S to be the set of all vertices reachable from s via a directed path in G_f, and T to be the complement of S; note that $s \in S$ and $t \in T$, so (S, T) is a valid cut. Since G_f contains no edges from S to T, it must be the case that for each edge e from S to T, the residual capacity $c_f(e)$ is zero (hence $f_e = c(e)$) and for each edge e from T to S, the residual capacity $c_f(\bar{e})$ is zero (hence $f_e = 0$). This confirms that S, T satisfies the conditions for the left and right sides of (1) to equal one another. □

3 Combinatorial Applications

In combinatorics, there are many examples of "min-max theorems" asserting that the minimum of `XXX` equals that maximum of `YYY`, where `XXX` and `YYY` are two different combinatorially-defined parameters related to some object such as a graph. Often these min-max theorems have two other salient properties.

1. It's straightforward to see that the maximum of `YYY` is no greater than the minimum of `XXX`, but the fact that they are equal is usually far from obvious, and in some cases quite surprising.

2. The theorem is accompanied by a polynomial-time algorithm to compute the minimum of `XXX` or the maximum of `YYY`.

Most often, these min-max relations can be derived as consequences of the max-flow min-cut theorem. (Which is, of course, one example of such a relation.) This also explains where the accompanying polynomial-time algorithm comes from.

There is a related phenomenon that applies to decision problems, where the question is whether or not an object has some property `P`, rather than a question about the maximum or minimum of some parameter. Once again, we find many theorems in combinatorics asserting that `P` holds if and only if `Q` holds, where:

1. It's straightforward to see that `Q` is necessary in order for `P` to hold, but the fact that `Q` is also sufficient is far from obvious.

2. The theorem is accompanied by a polynomial-time algorithm to decide whether property `P` holds.

Once again, these necessary and sufficient conditions can often be derived from the max-flow min-cut theorem

The main purpose of this section is to illustrate five examples of this phenomenon. Before getting to these applications, it's worth making a few other remarks.

1. The max-flow min-cut theorem is far from being the only source of such min-max relations. For example, many of the more sophisticated ones are derived from the Matroid Intersection Theorem, which is a topic that we will not be discussing this semester.

2. Another prolific source of min-max relations, namely LP Duality, has already been discussed informally this semester, and we will be coming to a proof later on. LP duality by itself yields statements about continuous optimization problems, but one can often derive consequences for discrete problems by applying additional special-purpose arguments tailored to the problem at hand.

3. The "applications" in these notes belong to mathematics (specifically, combinatorics) but there are many real-world applications of maximum flow algorithms. See Chapter 7 of Kleinberg & Tardos for applications to airline routing, image segmentation, determining which baseball teams are still capable of getting into the playoffs, and many more.

3.1 Preliminaries

The combinatorial applications of max-flow frequently rely on an easy observation about flow algorithms. The following theorem asserts that essentially everything we've said about network flow problems remains valid if some edges of the graph are allowed to have infinite capacity. Thus, in the following theorem, we define the term *flow network* to be a directed graph $G = (V, E)$ with source and sink vertices s, t and edge capacities $(c_e)_{e \in E}$ as before — including the stipulation that the vertex set V is finite — but we allow edge capacities $c(u, v)$ to be any non-negative real number *or infinity*. A flow is defined as before, except that when $c(u, v) = \infty$ it means that there is no capacity constraint for edge (u, v).

Theorem 5. *If G is a flow network containing an s-t path made up of infinite-capacity edges, then there is no upper bound on the maximum flow value. Otherwise, the maximum flow value*

and the minimum cut capacity are finite, and they are equal. Furthermore, any maximum flow algorithm that specializes the Ford-Fulkerson algorithm (e.g. Edmonds-Karp or Dinic) remains correct in the presence of infinite-capacity edges, and its worst-case running time remains the same.

Proof. If P is an s-t path made up of infinite capacity edges, then we can send an unbounded amount of flow from s to t by simply routing all of the flow along the edges of P. Otherwise, if S denotes the set of all vertices reachable from s by following a directed path made up of infinite-capacity edges, then by hypothesis $t \notin S$. So if we set $T = V \setminus S$, then (S, T) is an s-t cut and every edge from S to T has finite capacity. It follows that $c(S, T)$ is finite, and the maximum flow value is finite.

We now proceed by constructing a different flow problem $\hat{G}$ with the same directed graph structure finite edge capacities $\hat{c}_e$, and arguing that the outcome of running Ford-Fulkerson doesn't change when its input is modified from G to $\hat{G}$. The modified edge capacities in $\hat{G}$ are defined by

$$\hat{c}(u, v) = \begin{cases} c(u, v) & \text{if } c(u, v) < \infty \\ c(S, T) + 1 & \text{if } c(u, v) = \infty. \end{cases}$$

If (S', T') is any cut in $\hat{G}$ then either $\hat{c}(S', T') > \hat{c}(S, T) = c(S, T)$, or else $\hat{c}(S', T') = c(S', T')$; in particular, the latter case holds if (S', T') is a minimum cut in $\hat{G}$. To see this, observe that if $\hat{c}(S', T') \leq \hat{c}(S, T) = c(S, T)$, then for any $u \in S', v \in T'$, we have $\hat{c}(u, v) \leq c(S, T)$ and this in turn implies that $\hat{c}(u, v) = c(u, v)$ for all $u \in S', v \in T'$, and consequently $\hat{c}(S', T') = c(S', T')$.

Since $\hat{G}$ has finite edge capacities, we already know that any execution of the Ford-Fulkerson algorithm on input $\hat{G}$ will terminate with a flow f whose value is equal to the minimum cut capacity in $\hat{G}$. As we've seen, this is also equal to the minimum cut capacity in G itself, so the flow must be a maximum flow in G itself. Every execution of Ford-Fulkerson on $\hat{G}$ is also a valid execution on G and vice-versa, which substantiates the final claim about running times. □

3.2 Menger's Theorem

As a first application, we consider the problem of maximizing the number of disjoint paths between two vertices s, t in a graph. Menger's Theorem equates the maximum number of such paths with the minimum number of edges or vertices that must be deleted from G in order to separate s from t.

Definition 5. Let G be a graph, either directed or undirected, with distinguished vertices s, t. Two $s - t$ paths P, P' are *edge-disjoint* if there is no edge that belongs to both paths. They are *vertex-disjoint* if there is no vertex that belongs to both paths, other than s and t. (This notion is sometimes called *internally-disjoint*.)

Definition 6. Let G be a graph, either directed or undirected, with distinguished vertices s, t. An $s - t$ edge cut is a set of edges C such that every $s - t$ path contains an edge of C. An $s - t$ vertex cut is a set of vertices U, disjoint from $\{s, t\}$, such that every $s - t$ path contains a vertex of U.

Theorem 6 (Menger's Theorem)**.** *Let G be a (directed or undirected) graph and let s, t be two distinct vertices of G. The maximum number of edge-disjoint $s-t$ paths equals the minimum cardinality of an $s-t$ edge cut, and the maximum number of vertex-disjoint $s-t$ paths equals the minimum cardinality of an $s-t$ vertex cut. Furthermore the maximum number of disjoint paths can be computed in polynomial time.*

Proof. The theorem actually asserts four min-max relations, depending on whether we work with directed or undirected graphs and whether we work with edge-disjointness or vertex-disjointness. In all four cases, it is easy to see that the minimum cut constitutes an upper bound on the maximum number of disjoint paths, since each path must intersect the cut in a distinct edge/vertex. In all four cases, we will prove the reverse inequality using the max-flow min-cut theorem.

To prove the results about edge-disjoint paths, we simply make G into a flow network by defining $c(u, v) = 1$ for all directed edges $(u, v) \in E(G)$; if G is undirected then we simply set $c(u, v) = c(v, u) = 1$ for all $(u, v) \in E(G)$. The theorem now follows from two claims: **(A)** an integer $s-t$ flow of value k implies the existence of k edge-disjoint $s-t$ paths and vice versa; **(B)** a cut of capacity k implies the existence of an $s-t$ edge cut of cardinality k and vice-versa. To prove (A), we can decompose an integer flow f of value k into a set of edge-disjoint paths by finding one $s-t$ path consisting of edges (u, v) such that $f(u, v) = 1$, setting the flow on those edges to zero, and iterating on the remaining flow; the transformation from k disjoint paths to a flow of value k is even more straightforward. To prove (B), from an $s-t$ edge cut C of cardinality k we get an $s-t$ cut of capacity k by defining S to be all the vertices reachable from s without crossing C; the reverse transformation is even more straightforward.

To prove the results about vertex-disjoint paths, the transformation uses some small "gadgets". Every vertex v in G is transformed into a pair of vertices $v_{\text{in}}, v_{\text{out}}$, with $c(v_{\text{in}}, v_{\text{out}}) = 1$ and $c(v_{\text{out}}, v_{\text{in}}) = 0$. Every edge (u, v) in G is transformed into an edge from u_{out} to v_{in} with infinite capacity. In the undirected case we also create an edge of infinite capacity from v_{out} to u_{in}. Now we solve max-flow with source s_{out} and sink t_{in}. As before, we need to establish two claims: **(A)** an integer $s_{\text{out}}-t_{\text{in}}$ flow of value k implies the existence of k vertex-disjoint $s-t$ paths and vice versa; **(B)** a cut of capacity k implies the existence of an $s_{\text{out}}-t_{\text{in}}$ vertex cut of cardinality k and vice-versa. Claim (A) is established exactly as above. Claim (B) is established by first noticing that in any finite-capacity cut, the only edges crossing the cut must be of the form $(v_{\text{in}}, v_{\text{out}})$; the set of all such v then constitutes the $s-t$ vertex cut. □

3.3 The König-Egervary Theorem

Recall that a matching in a graph is a collection of edges such that each vertex belongs to at most one edge. A *vertex cover* of a graph is a vertex set A such that every edge has at least one endpoint in A. Clearly the cardinality of a maximum matching cannot be greater than the cardinality of a minimum vertex cover. (Every edge of the matching contains a distinct element of the vertex cover.) The König-Egervary Theorem asserts that in bipartite graphs, these two parameters are always equal.

Theorem 7 (König-Egervary)**.** *If G is a bipartite graph, the cardinality of a maximum matching in G equals the cardinality of a minimum vertex cover in G.*

Proof. The proof technique illustrates a very typical way of using network flow algorithms: we make a bipartite graph into a flow network by attaching a "super-source" to one side and a "super-sink" to the other side. Specifically, if G is our bipartite graph, with two vertex sets X, Y, and edge set E, then we define a flow network $\hat{G} = (X \cup Y \cup \{s, t\}, c, s, t)$ where the following edge capacities are nonzero, and all other edge capacities are zero:

$$\begin{aligned} c(s, x) &= 1 \quad \text{for all } x \in X \\ c(y, t) &= 1 \quad \text{for all } y \in Y \\ c(x, y) &= \infty \quad \text{for all } (x, y) \in E \end{aligned}$$

For any integer flow in this network, the amount of flow on any edge is either 0 or 1. The set of edges (x, y) such that $x \in X, y \in Y, f(x, y) = 1$ constitutes a matching in G whose cardinality is equal to $|f|$. Conversely, any matching in G gives rise to a flow in the obvious way. Thus the maximum flow value equals the maximum matching cardinality.

If (S, T) is any finite-capacity $s - t$ cut in this network, let $A = (X \cap T) \cup (Y \cap S)$. The set A is a vertex cover in G, since an edge $(x, y) \in E$ with no endpoint in A would imply that $x \in S, y \in T, c(x, y) = \infty$ contradicting the finiteness of $c(S, T)$. The capacity of the cut is equal to the number of edges from s to T plus the number of edges from S to t (no other edges from S to T exist, since they would have infinite capacity), and this sum is clearly equal to $|A|$. Conversely, a vertex cover A gives rise to an $s - t$ cut via the reverse transformation, and the cut capacity is $|A|$. □

3.4 Hall's Theorem

Theorem 8. *Let G be a bipartite graph with vertex sets X, Y and edge set E. Assume $|X| = |Y|$. For any $W \subseteq X$, let $\Gamma(W)$ denote the set of all $y \in Y$ such that $(w, y) \in E$ for at least one $w \in W$. In order for G to contain a perfect matching, it is necessary and sufficient that each $W \subseteq X$ satisfies $|\Gamma(W)| \geq |W|$.*

Proof. The stated condition is clearly necessary. To prove it is sufficient, assume that $|\Gamma(W)| \geq |W|$ for all W. Transform G into a flow network $\hat{G}$ as in the proof of the König-Egervary Theorem. If there is a integer flow of value $|X|$ in $\hat{G}$, then the edges (x, y) such that $x \in X, y \in Y, f(x, y) = 1$ constitute a perfect matching in G and we are done. Otherwise, there is a cut (S, T) of capacity $k < n$. We know that

$$|X \cap T| + |Y \cap S| = k < n = |X \cap T| + |X \cap S|$$

from which it follows that $|Y \cap S| < |X \cap S|$. Let $W = X \cap S$. The set $\Gamma(W)$ is contained in $Y \cap S$, as otherwise there would be an infinite-capacity edge crossing from S to T. Thus, $|\Gamma(W)| \leq |Y \cap S| < |W|$, and we verified that when a perfect matching *does not* exist, there is a set W violating Hall's criterion. □

3.5 Dilworth's Theorem

In a directed acyclic graph G, let us say that a pair of vertices v, w are *incomparable* if there is no path passing through both v and w, and define an *antichain* to be a set of pairwise incomparable vertices.

Theorem 9. *In any finite directed acyclic graph G, the maximum cardinality of an antichain equals the minimum number of paths required to cover the vertex set of G.*

The proof is much trickier than the others. Before presenting it, it is helpful to introduce a directed graph G^* called the *transitive closure* of G. This has same vertex set V, and its edge set E^* consists of all ordered pairs (v, w) such that $v \neq w$ and there exists a path in G from v to w. Some basic facts about the transitive closure are detailed in the following lemma.

Lemma 10. *If G is a directed acyclic graph, then its transitive closure G^* is also acyclic. A vertex set A constitutes an independent set in G^* (i.e. no edge in E^* has both endpoints in S) if and only if A is an antichain in G. A sequence of vertices $v_0, v_1, \ldots, v_k$ constitutes a path in G^* if and only if it is a subsequence of a path in G. For all k, G^* can be partitioned into k or fewer paths if and only if G can be covered by k or fewer paths.*

Proof. The equivalence of antichains in G and independent sets in G^* is a direct consequence of the definitions. If $v_0, \ldots, v_k$ is a directed walk in G^* — i.e., a sequence of vertices such that (v_{i-1}, v_i) is an edge for each $i = 1, \ldots, k$ — then there exist paths P_i from v_{i-1} to v_i in G, for each i. The concatenation of these paths is a directed walk in G, which must be a simple path (no repeated vertices) since G is acyclic. This establishes that $v_0, \ldots, v_k$ is a subsequence of a path in G, as claimed, and it also establishes that $v_0 \neq v_k$, hence G^* contains no directed cycles, as claimed. Finally, if G^* is partitioned into k paths then we may apply this construction to each of them, obtaining k paths that cover G. Conversely, given k paths $P_1, \ldots, P_k$ that cover G, then G^* can be partitioned into paths $P_1^*, \ldots, P_k^*$ where P_i^* is the subsequence of P_i consisting of all vertices that do not belong to the union of $P_1, \ldots, P_{i-1}$. □

Using these facts about the transitive closure, we may now prove Dilworth's Theorem.

Proof of Theorem 9. Define a flow network $\hat{G} = (W, c, s, t)$ as follows. The vertex set W contains two special vertices s, t as well as two vertices x_v, y_v for every vertex $v \in V(G)$. The following edge capacities are nonzero, and all other edge capacities are zero.

$$\begin{aligned} c(s, x_v) &= 1 \quad \text{for all } v \in V \\ c(x_v, y_w) &= \infty \quad \text{for all } (v, w) \in E^* \\ c(y_w, t) &= 1 \quad \text{for all } w \in V \end{aligned}$$

For any integer flow in the network, the amount of flow on any edge is either 0 or 1. Let F denote the set of edges $(v, w) \in E^*$ such that $f(x_v, y_w) = 1$. The capacity and flow conservation constraints enforce some degree constraints on F: every vertex of G^* has at

most one incoming edge and at most one outgoing edge in F. In other words, F is a union of disjoint paths and cycles. However, since G^* is acyclic, F is simply a union of disjoint paths in G^*. In fact, if a vertex doesn't belong to any edge in F, we will describe it as a path of length 0 and in this way we can regard F as a partition of the vertices of G^* into paths. Conversely, every partition of the vertices of G^* into paths translates into a flow in $\hat{G}$ in the obvious way: for every edge (v, w) belonging to one of the paths in the partition, send one unit of flow on each of the edges $(s, x_v), (x_v, y_w), (y_w, t)$.

The value of f equals the number of edges in F. Since F is a disjoint union of paths, and the number of vertices in a path always exceeds the number of edges by 1, we know that $n = |F| + p(F)$. Thus, if the maximum flow value in $\hat{G}$ equals k, then the minimum number of paths in a path-partition of G^* equals $n - k$, and Lemma 10 shows that this is also the minimum number of paths in a path-covering of G. By max-flow min-cut, we also know that the minimum cut capacity in $\hat{G}$ equals k, so to finish the proof, we must show that an $s - t$ cut of capacity k in $\hat{G}$ implies an antichain in G — or equivalently (again using Lemma 10) an independent set in G^* — of cardinality $n - k$.

Let S, T be an $s - t$ cut of capacity k in $\hat{G}$. Define a set of vertices A in G^* by specifying that $v \in A$ if $x_v \in S$ and $y_v in T$. If a vertex v does not belong to A then at least one of the edges (s, x_v) or (y_v, t) crosses from S to T, and hence there are at most k such vertices. Thus $|A| \geq n - k$. Furthermore, there is no edge in G^* between elements of A: if (v, w) were any such edge, then (v, w') would be an infinite-capacity edge of $\hat{G}$ crossing from S to T. Hence there is no path in G between any two elements of A, i.e. A is an antichain. □

4 The Ford-Fulkerson Algorithm

Lemma 3 constitutes the basis for the Ford-Fulkerson algorithm, which computes a maximum flow iteratively, by initializing $f = 0$ and repeatedly replacing f with $f + \delta(P)\pi(\mathbf{1}_P)$ where P is an augmenting path in G_f, and $\delta(P)$ is the minimum residual capacity of an edge in P. The algorithm terminates when G_f no longer contains an augmenting path, at which point Lemma 3 guarantees that f is a maximum flow.

Algorithm 1 FordFulkerson(G)

```
1: f ← 0; G_f ← G
2: while G_f contains an s-t path P do
3:     Let P be one such path.
4:     Let δ(P) = min{c_f(e) | e ∈ P}.
5:     f ← f + δ(P)π(1_P)              // Augment f using P.
6:     Update G_f.
7: end while
8: return f
```

Theorem 11. *In any flow network with integer edge capacities, any execution of the Ford-Fulkerson algorithm terminates and outputs an integer-valued maximum flow, f^*, after at most $|f^*|$ iterations of the main loop.*

Proof. At any time during the algorithm's execution, the residual capacities c_f are all integers; this can easily be seen by induction on the number of iterations of the main loop, the key observation being that the quantity $\delta(P)$ computed during each loop iteration must always be an integer.

It follows that $|f|$ increases by at least 1 during each loop iteration, so the algorithm terminates after at most $|f^*|$ loop iterations, where f^* denotes the output of the algorithm. Finally, Lemma 3 ensures that f^* must be a maximum flow because, by the algorithm's termination condition, its residual graph has no augmenting path. □

Each iteration of the Ford-Fulkerson main loop can be implemented in linear time, i.e. the time required to search for the augmenting path P in G_f (using BFS or DFS) and to construct the new residual graph after updating f. For the sake of simplicity, we will express the running time of each loop iteration as $O(m)$ rather than $O(m+n)$, which can be justified by making a standing assumption that each vertex of the graph is incident to at least one edge, hence $m \geq n/2$. (If the standing assumption is violated, isolated vertices can be removed using a trivial $O(n)$ preprocessing step, which adds $O(n)$ to the running time of every algorithm considered in these notes.) The benefit of the standing assumption is that it leads to simpler and more readable running time bounds for the maximum-flow algorithms we are analyzing. In integer-capacitated graphs, we have seen that the Ford-Fulkerson algorithm runs in at most $|f^*|$ linear-time iterations, where $|f^*|$ is the value of a maximum flow, hence the algorithm's running time is $O(m|f^*|)$.

5 The Edmonds-Karp and Dinitz Algorithms

The Ford-Fulkerson algorithm's running time is pseudopolynomial, but not polynomial. In other words, its running time is polynomial in the *magnitudes* of the numbers constituting the input (i.e., the edge capacities) but not polynomial in the *number of bits* needed to describe those numbers. To illustrate the difference, consider a flow network with vertex set $\{s, t, u, v\}$ and edge set $\{(s, u), (u, t), (s, v), (v, t), (u, v)\}$. The capacities of the edges are

$$c(s, u) = c(u, t) = c(s, v) = c(v, t) = 2^n, \quad c(u, v) = 1.$$

The maximum flow in this network sends 2^n units on each of the paths $\langle s, u, t\rangle$ and $\langle s, v, t\rangle$, and if the Ford-Fulkerson algorithm chooses these as its first two augmenting paths, it terminates after only two iterations. However, it could alternatively choose $\langle s, u, v, t\rangle$ as its first augmenting path, sending only one unit of flow on the path. This results in adding the edge (v, u) to the residual graph, at which point it becomes possible to send one unit of flow on the augmenting path $\langle s, u, v, t\rangle$. This process iterates 2^n times.

A more sophisticated example shows that in a flow network whose edge capacities are irrational numbers, the Ford-Fulkerson algorithm may run through its main loop an infinite number of times without terminating.

In this section we will present two maximum flow algorithms with *strongly polynomial* running times. This means that if we count each arithmetic operation is consuming only one unit of running time (regardless of the number of bits of precision of the numbers involved)

then the running time is bounded by a polynomial function of the number of vertices and edges of the network.

5.1 The Edmonds-Karp Algorithm

The Edmonds-Karp algorithm refines the Ford-Fulkerson algorithm by always choosing the augmenting path with the smallest number of edges.

Algorithm 2 EdmondsKarp(G)

```
1: f ← 0; G_f ← G
2: while G_f contains an s − t path P do
3:     Let P be an s − t path in G_f with the minimum number of edges.
4:     f ← f + δ(P)π(1_P)              // Augment f using P.
5:     Update G_f
6: end while
7: return f
```

To begin our analysis of the Edmonds-Karp algorithm, note that the s-t path in G_f with the minimum number of edges can be found in $O(m)$ time using breadth-first search. Once path P is discovered, it takes only $O(n)$ time to augment f using P and $O(n)$ time to update G_f, so we see that one iteration of the **while** loop in EdmondsKarp(G) requires only $O(m)$ time. However, we still need to figure out how many iterations of the **while** loop could take place, in the worst case.

To reason about the maximum number of **while** loop iterations, we will assign a distance label $d(v)$ to each vertex v, representing the length of the shortest path from s to v in G_f. We will show that $d(v)$ never decreases during an execution of EdmondsKarp(G). Recall that the same method of reasoning was instrumental in the running-time analysis of the Hopcroft-Karp algorithm.

Any edge (u, v) in G_f must satisfy $d(v) \leq d(u) + 1$, since a path of length $d(u) + 1$ can be formed by appending (u, v) to a shortest s-u path in G_f. Call the edge *advancing* if $d(v) = d(u) + 1$ and *retreating* if $d(v) \leq d(u)$. Any shortest augmenting path P in G_f is composed exclusively of advancing edges. Let G_f and $\tilde{G}_f$ denote the residual graph before and after augmenting f using P, respectively, and let $d(v), \tilde{d}(v)$ denote the distance labels of vertex v in the two residual graphs. Every edge (u, v) in $\tilde{G}_f$ is either an edge of G_f or the reverse of an edge of P; in both cases the inequality $d(v) \leq d(u) + 1$ is satisfied. Therefore, on any path in $\tilde{G}_f$ the value of d increases by at most one on each hop of the path, and consequently $\tilde{d}(v) \geq d(v)$ for every v. This proves that the distance labels never decrease, as claimed earlier.

When we choose augmenting path P in G_f, let us say that edge $e \in E(G_f)$ is a bottleneck edge for P if it has the minimum residual capacity of any edge of P. Notice that when $e = (u, v)$ is a bottleneck edge for P, then it is eliminated from G_f after augmenting f using P. Suppose that $d(u) = i$ and $d(v) = i + 1$ when this happens. In order for e to be added back into G_f later on, edge (v, u) must belong to a shortest augmenting path, implying

$d(u) = d(v) + 1 \geq i + 2$ at that time. Thus, the total number of times that e can occur as a bottleneck edge during the Edmonds-Karp algorithm is at most $n/2$. There are $2m$ edges that can potentially appear in the residual graph, and each of them serves as a bottleneck edge at most $n/2$ times, so there are at most mn bottleneck edges in total. In every iteration of the **while** loop the augmenting path has at least one bottleneck edge, so there are at most mn **while** loop iterations in total. Earlier, we saw that every iteration of the loop takes $O(m)$ time, so the running time of the Edmonds-Karp algorithm is $O(m^2n)$.

5.2 The Dinitz Algorithm

Similar to the way that the Hopcroft-Karp algorithm improves the running time for finding a maximum matching in a graph by finding a *maximal* set of shortest augmenting paths all at once, there is a maximum-flow algorithm due to Dinitz that improves the running time of the Edmonds-Karp algorithm by finding a so-called *blocking flow* in the residual graph.

Definition 7. If G is a flow network, f is a flow, and h is a flow in the residual graph G_f, then h is called a *blocking flow* if every shortest augmenting path in G_f contains at least one edge that is saturated by h, and every edge e with $h_e > 0$ belongs to a shortest augmenting path.

Algorithm 3 EdmondsKarp(G)

```
1: f ← 0; G_f ← G
2: while G_f contains an s − t path P do
3:     Let h be a blocking flow in G_f.
4:     f ← f + π(h)
5:     Update G_f
6: end while
7: return f
```

Later we will specify how to compute a blocking flow. For now, let us focus on bounding the number of iterations of the main loop. As in the analysis of the Edmonds-Karp algorithm, the distance $d(v)$ of any vertex v from the source s can never decrease during an execution of Dinitz's algorithm. Furthermore, the length of the shortest path from s to t in G_f must *strictly* increase after each loop iteration: the edges (u, v) which are added to G_f at the end of the loop iteration satisfy $d(v) \leq d(u)$ (where $d(\cdot)$ refers to the distance labels at the *start* of the iteration) so any s-t path of length $d(t)$ in the *new* residual graph would have to be composed exclusively of advancing edges which existed in the *old* residual graph. However, any such path must contain at least one edge which was saturated by the blocking flow, hence deleted from the residual graph. Therefore, each loop iteration strictly increases $d(t)$ and the number of loop iterations is bounded above by n.

The algorithm to compute a blocking flow explores the subgraph composed of advancing edges in a depth-first manner, repeatedly finding augmenting paths.

Algorithm 4 BLOCKINGFLOW(G_f)

$h \leftarrow 0$
Let G' be the subgraph composed of advancing edges in G_f.
Initialize $c'(e) = c_f(e)$ for each edge e in G'.
Initialize stack with $\langle s \rangle$.
repeat
    Let u be the top vertex on the stack.
    if $u = t$ **then**
        Let P be the path defined by the current stack. *// Now augment h using P.*
        Let $\delta(P) = \min\{c'(e) \mid e \in P\}$.
        $h \leftarrow h + \delta(P)\mathbf{1}_P$.
        $c'(e) \leftarrow c'(e) - \delta(P)$ for all $e \in P$.
        Delete edges with $c'(e) = 0$ from G'.
        Let (u, v) be the newly deleted edge that occurs earliest in P.
        Truncate the stack by popping all vertices above u.
    else if G' contains an edge (u, v) **then**
        Push v onto the stack.
    else
        Delete u and all of its incoming edges from G'.
        Pop u off of the stack.
    end if
until stack is empty
return h

The block of code that augments h using P is called at most m times (each time results in the deletion of at least one edge) and takes $O(n)$ steps each time, so it contributes $O(mn)$ to the running time of BLOCKINGFLOW(G_f). At most n vertices are pushed onto the stack before either a path is augmented or a vertex is deleted, so $O(mn)$ time is spent pushing vertices onto the stack. The total work done initializing G', as well as the total work done deleting vertices and their incoming edges, is bounded by $O(m)$. Thus, the total running time of BLOCKINGFLOW(G_f) is bounded by $O(mn)$, and the running time over Dinitz's algorithm overall is bounded by $O(mn^2)$.

A modification of Dinitz's algorithm using fancy data structures achieves running time $O(mn \log n)$. The preflow-push algorithm, presented in Section 7.4 of Kleinberg-Tardos, has a running time of $O(n^3)$. The fastest known strongly-polynomial algorithm, due to Orlin, has a running time of $O(mn)$. There are also weakly polynomial algorithms for maximum flow in integer-capacitated networks, i.e. algorithms whose running time is polynomial in the number of vertices and edges, and the logarithm of the largest edge capacity, U. The fastest such algorithm, due to Lee and Sidford, has a running time of $O(m\sqrt{n}\,\mathrm{poly}(\log n, \log U))$.

Analysis of Algorithms	Linear Programming Notes
CS 6820 Fall 2014	Lectures, October 3-20, 2014

1 Linear programming

The linear programming (LP) problem is the following optimization problem. We are given matrix A and vectors b and c and the problem is to find x that

$$\max(cx \text{ such that: } Ax \leq b\}.$$

Assume that A is an n by m matrix, b is an m vector, and c and x are n vectors so the multiplications, and inequalities above make sense. So x is a vector of n variables, and $Ax \leq b$ is a set of m inequalities. An example in two variables would be

$$\begin{aligned} max x_1 + x_2 & \\ 2x_1 + x_2 &\leq 3 \\ x_1 + 2x_2 &\leq 5 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned}$$

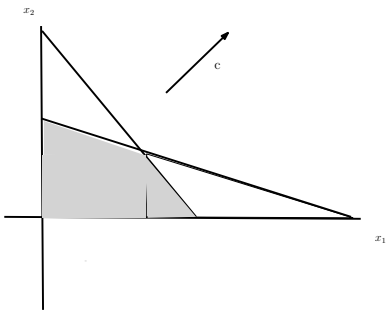

Figure 1: linear program in two dimensions.

An algorithm for linear programming takes A, b and c as input, and returns one of the following three answers:

- "no solutions exist", if there is no solution x such that $Ax \leq b$.
- "the maximum is unbounded", if for any γ there is a solution $Ax \leq b$ with $cx \geq \gamma$.
- return a vector x that satisfies $Ax \leq b$ and achieves the maximum of cx.

2 First example matching and fractional matching

As a first example of linear programming consider the matching problem. We are given a graph $G = (V, E)$. To think of matching this way, we associate a variable x_e with every edge $e \in E$. We would like to think of these variables taking values 0 or 1 with $x_e = 1$ indicating that edge e in the matching, and 0 when its not in the matching. To write the maximum matching problem as a set of inequalities we have

$$\begin{aligned} x_e &\in \{0,1\} \text{ for all } e \in E \\ \sum_{e \text{ adjacent to } v} x_e &\leq 1 \text{ for all } v \in V \\ max \sum_e x_e & \end{aligned}$$

Note that this is an *integer* linear program, as we require x_e to be 0 or 1, and not a fractional value between the two.

Lemma 1 *Integer solutions x to the above inequalities are in one-to-one correspondence to matchings $M = \{e : x_e = 1\}$, with the matching of maximum size corresponding to the optimal solution.*

To define the *fractional matching problem* we replace the constrain $x_e \in \{0, 1\}$ by $0 \leq x_e \leq 1$ for all edges. So the fractional matching problem is

$$\begin{aligned} 0 \leq x_e &\leq 1 \text{ for all } e \in E \\ \sum_{e \text{ adjacent to } v} x_e &\leq 1 \text{ for all } v \in V \\ max \sum_e x_e & \end{aligned}$$

What can we say about the maximum? One way to derive bounds on the sum is to add up all the $n = |V|$ inequalities for the nodes. We get

$$\sum_v \sum_{e \text{ adjacent to } v} x_e \leq n.$$

Each edge $e = (v, u)$ occurs twice on the left hand side, as e is on the list of edges for the sum at the node v and u, so the inequality is

$$2 \sum_e x_e \leq n,$$

i.e., the sum is at most $n/2$. Not only the maximum matching is limited to at most half the number of vertices, but also the maximum fractional matching.

Alternately, we can add up a subset of the inequalities. A subset $A \subset V$ is a *vertex cover* if A contains at least one of the ends at each edge e. Adding the inequalities for $v \in A$ we get

$$\sum_{v \in A} \sum_{e \text{ adjacent to } v} x_e \leq |A|.$$

Since A is a vertex cover, each edge variable x_e occurs at least once on the left hand side, and some occurs twice. However, we also have that $x_e \geq 0$, so we also have

$$\sum_e x_e \leq \sum_{v \in A} \sum_{e \text{ adjacent to } v} x_e \leq |A|.$$

for all vertex covers A. The minimum vertex cover problem is to find a vertex cover of minimum size. The above inequality is true for all vertex covers, and hence also for the minimum vertex cover.

Lemma 2 *The maximum $\sum_e x_e$ for a fractional matching is at most the minimum size $|A|$ of a vertex cover.*

We can further strengthen the inequality by considering *fractional vertex cover*: adding each inequality with a nonnegative multiplier y_v. A vector $y_v \geq 0$ for all $v \in V$ is a fractional vertex cover if for each edge $e = (v, u)$ we have $y_u + y_v \geq 1$. Note that a fractional vertex cover where $y_v \in \{0, 1\}$ is a regular vertex cover.

Lemma 3 *Integer solutions y to the above inequalities are in one-to-one correspondence to vertex covers $A = \{v : y_v = 1\}$, with the vertex cover of minimum size corresponding to the integer solution with minimum $\sum_v y_v$.*

Consider a fractional vertex cover y. Multiplying $\sum_{e \text{ adjacent to } v} x_e \leq 1$ by y_v we get

$$y_v \sum_{e \text{ adjacent to } v} x_e \leq y_v$$

and adding the inequalities for all nodes (and turning the sides around to help the write-up), we get

$$\begin{aligned}
\sum_{v \in V} y_v &\geq \sum_{v \in V} y_v \sum_{e \text{ adjacent to } v} x_e \\
&= \sum_{e=(u,v) \in E} x_e (y_v + y_u) \\
&\geq \sum_{e=(u,v) \in E} x_e
\end{aligned}$$

where the equation in the middle is just reordering the sum, and the inequality follows as y is a fractional vertex cover and x is nonnegative.

Summing up we get the following main theorem

Theorem 4

$$\max_{matching\ M} |M| \leq max_x \sum_e x_e \leq min_y \sum_v y_v \leq min_{A\ \ vertex\ cover} |A|$$

where the maximum in the middle is over fractional matchings x, and the minimum is over fractional vertex covers y.

Remark. Recall from a couple lectures ago we have seen as an application of max-flows and min-cuts that in bipartite graph the size of a maximum matching equals the minimum size of a vertex cover, so there is equation throughout the chain on inequalities above in bipartite graphs. This is not true in general graphs. Consider for example a triangle. The maximum matching is of size 1, the minimum vertex cover needs 2 nodes, and note that $x_e = 0.5$ for all e, and $y_v = 0.5$ for all v define fractional matching and fractional vertex covers with values 1.5. More generally, consider a complete graph on $n = 2k + 1$ nodes. The maximum matching is of size n, we can get a fractional matching of size $n/2$, by say using a triangle with $1/2$ on each edge, and a matching on the rest. Putting $y_v = 1/2$ in each node gives a fractional vertex cover of value $n/2$ also, while the minimum size of an integer vertex cover is $n - 1$.

3 Linear programs and their duals

Using the example of fractional matching, we derived upper bounds on the maximum, by adding up fractional copies of the inequalities (multiplying each by a nonnegative value y_v. Thinking about such bounds more generally leads to the concept of the dual of a linear program. Consider again linear programs in the general form

$$\max(cx \text{ such that: } Ax \leq b\}.$$

Let $a_i x \leq b_i$ denote the inequality in the ith row of this system. For any nonnegative $y_i \geq 0$ we can get the inequality $y_i(a_i x) \leq y_i b_i$, and adding up such inequalities for a vector $y \geq 0$ we get

$$\sum_i y_i(a_i x) \leq \sum_i y_i b_i$$

or using vector notation, we have $y(Ax) \leq yb$. If it happens to be the case, that $yA = c$, than the inequality we just derived is $cx \leq yb$, hence yb is an upper bound of the maximum our linear program seeks to find. The *dual linear program* is the best such upper bound possible. More formally, it is the program

$$\min(yb : y \geq 0 \text{ and } yA = c\}.$$

By the way we derived this program, for each y the value yb is an upper bound of our original linear program, which immediately gives us the following.

Theorem 5 (weak duality) *For any linear program defined by matrix A and vectors b and c we have*

$$\max(cx : Ax \leq b\} \leq \min(yb : y \geq 0 \text{ and } yA = c\}.$$

4 Fractional matchings, flows, and linear programs in nonnegative variables

Going back to fractional matching, the fractional matching problem had inequalities for all vertices, but also had constraints that require each variable $0 \leq x_e \leq 1$. Observe that the constraints $x_e \leq 1$ for an edge $e = (u, v)$ are redundant, as they follow from the inequalities that the variables associated with edges adjacent to, say the vertex v, need to sum to at most 1. However, the constraints $x_e \geq 0$ are important. It is useful to thing about what is the dual of a linear program with $x \geq 0$ constraints. To take the dual of this linear program with the method we have seen so far, we need to introduce nonnegative variables associates with both the $Ax \leq b$ constraints, as well as the $x \geq 0$ constraints (which we may want to write as $-x \leq 0$). Lets call this second set of variables s. Taking the dual we get the following:

$$\min(yb + s0 : y \geq 0, s \leq 0 \text{ and } yA - s = c\}.$$

Since the right hand side of the second set of constraints is 0, the s variables do not contribute to the objective function, so we can simply the dual linear program to be the following

$$\min(yb : y \geq 0 \text{ and } yA \leq c\}.$$

We get the

Theorem 6 (weak duality II) *For any linear program defined by matrix A and vectors b and c where the solution is required to be nonnegative, we have*

$$\max(cx : x \geq 0, Ax \leq b\} \leq \min(yb : y \geq 0 \text{ and } yA \geq c\}.$$

Notice that this applying this to fractional matching we see that fractional vertex cover is the dual linear program for fractional matching. When we write the fractional matching inequalities as a matrix $Ax \leq b$, we have a variable for each edge, and a constraint for each vertex. The matrix A therefore is $m = |E|$ by $n = |V|$. The matrix A has 0/1 entries. A row of A corresponding to vertex v has 1 in positions corresponding to edges e adjacent to v, and hence a column of A corresponding to an edge $e = (u, v)$ has 1 in the two positions associated with the two vertices u and v. So the dual inequality $yA \geq c$ becomes $y_u + y_v \geq 1$ for all edges $e = (u, v)$.

Corollary 7 *The dual linear program for fractional matching is the linear program for fractional vertex cover.*

Recall that in *bipartite* graphs we have seen that the maximum matching is the same size as the minimum size of a vertex cover. This implies that in bipartite graphs the maximum fractional matching is the same size as the minimum fractional vertex cover also. We also have seen that the integer matching and integer vertex cover is not the same size on a triangle, but we'll show below that the maximum size of a fractional matching is the same as the minimum size of a fractional vertex cover on all graphs. This will follow from the strong linear programming duality.

Next we consider the maximum flow problem. You may recall the formulation of maximum flow with variables on paths. Given a directed graph $G = (V, E)$ with nonnegative capacities $c_e \geq 0$ on the edges, and a source-sink pair $s, t \in V$, the flow problem is defined as a linear program with variables associated with all $s - t$ paths. Let $\mathcal{P}$ denote the set of paths in G from s to t. Now the problem is (using x as a variable name rather than f to make it more similar to our other linear programs):

$$\begin{aligned} x_P &\geq 0 \text{ for all } P \in \mathcal{P} \\ \sum_{P:e\in P} x_P &\leq c_e \text{ for all } e \in E \end{aligned}$$

$$max \sum_P x_P$$

The dual of this linear program has variables associated with the edges (the inequalities of the above system), and has a variable associated with each path $P \in \mathcal{P}$. The dual program then becomes the following.

$$\begin{aligned} y_e &\geq 0 \text{ for all } e \in E \\ \sum_{e\in P} y_e &\geq 1 \text{ for all } P \in \mathcal{P} \end{aligned}$$

$$min \sum_e c_e y_e$$

Notice that since the capacities c_e are nonnegative, an optimal solution will have $y_e \leq 1$ for all edges. Now consider an integer solution when y_e is 0 or 1 on all edges, and let $F = \{e : y_e = 1\}$ be the selected set of edges. The constraint on paths requires that all $s - t$ path must contain an edge in F, so F must contain an (s, t) cut, and by minimality, and optimal solution is then an (s, t)-cut, and its value is exactly the capacity of the cut.

Lemma 8 *Integer optimal solutions to the above dual linear program are in one-to-one correspondence with minimum capacity (s, t)-cuts in the graph.*

We know from linear programming duality that the maximum fractional flow has the same value as the minimum of the dual program. Note that in the case of flows, we have seen that the integer max flow is equal to the min cut value. Our observation that min-cuts are the integer solutions of the dual linear program shows that the dual linear program also has an integer dual solution.

Corollary 9 *The above dual of the max-flow problem is guaranteed to have an optimal solution with variables y integer, and hence the flow linear problem and its dual has the same optimal solution value.*

5 Strong duality of linear programs

We have seen that the primal and dual linear programs have equal values in the max-flow problem. While not all linear programs solve optimally in integer variables, we'll see that the primal and dual linear programs always have equal solution values. This is the main theorem of linear programming, called strong duality, i.e., that in inequality in the weak duality theorem is always equal.

Theorem 10 (strong duality) *For any linear program defined by matrix A and vectors b and c, if there is a solution x such that $Ax \leq b$ then we have*

$$\max(cx : Ax \leq b\} = \min(yb : y \geq 0 \text{ and } yA = c\}.$$

as well as

$$\max(cx : x \geq 0, Ax \leq b\} = \min(yb : y \geq 0 \text{ and } yA \geq c\}.$$

First observe the second statement follows from the first, by simply applying the first to linear programs with $x \geq 0$ as part of the constraint matrix. Second recall that by weak duality, we know that all solutions x and all solutions y have $cx \leq yb$. To prove the equality, all we have to do is to exhibit a pair of solutions x^* and y^* such that $cx^* = y^*b$. Once we do this we have that for al solutions x the value $cx \leq y^*b = cx^*$ so x^* is optimal, and similarly, for all solutions y we have that $yb \geq cx^* = yb^*$, so y^* is optimal.

We will not formally prove this theorem that such an x^* and y^* must exists, rather will present a"proof" based on physics principles, that we hope will give good intuition why the theorem is true without being too complex. We will think of the area $P = \{x : Ax \leq\}$ as a physical area enclosing say a small metal ball. The walls of the area are bounded by the inequalities $a_i x \leq b_i$, and x will denote the location of the ball. We will also imagine that there a strong magnet "at infinity" in direction of c that is puling the ball, however, the ball cannot leave the bounding area P.

1. if the ball keeps accelerating in c direction forever, the value of cx will go to infinity as the ball moves, so $\max cx = \infty$.

2. If the ball cannot accelerate forever, it must come to a stop. Let x denote the place it stops.

3. At this point the ball has a force c on it from the magnet, the walls of the bounding area must exert force that compensates the magnet's force. The wall $a_i x \leq b_i$ can exert force in a_i direction, so this force can be of the form $y_i a_i$ for a nonnegative number $y_i \geq 0$. The ball stopped, so the forces acting on it sum to 0, so we get $c - \sum_i y_i a_i = 0$.

4. Finally observe that only walls that touch the ball can exert any force on it, so if $a_i x <, b_i$ we must have $y_i = 0$.

We claim that the place x^* where the ball stops and the vector $y^* = (y_1^*, \ldots, y_m^*)$ form optimal primal and dual solutions.

Lemma 11 *The properties listed above that are derived from physical principles, imply that the place x^* where the ball stops and the vector $y^* = (y_1^*, \ldots, y_m^*)$ form optimal primal and dual solutions.*

Proof. First we note that x^* and y^* are feasible solutions. The ball is inside the region P, so $Ax^* \leq b$ by definition. We also have that $y^* \geq 0$ as the wall holds the ball inside by exerting force, but is not pulling the ball towards the wall, so $y_i^* \geq 0$ for all i, and we have seen that $c = \sum_i y_i^* a_i$ as the forces add up to 0.

Next we want to show that x^* is of maximum value cx and y^* has minimum value yb. Recall that all we have to do to show this is to argue that $cx^* = y * b$. To see this consider the chain of inequalities

$$cx^* = (y^*A)x^* \leq y^*(Ax^*) = \sum_i y_i^*(a_i x^*) \leq \sum_i y_i^* b_i,$$

that is true for all feasible solutions x^* and y^* (the first equality is by $c = y^*A$, the second by rearranging terms, and the inequality follows as its true term by term: $y_i^*(a_i x^*) \leq y_i^* b_i$ for all i, as $y_i \geq 0$ and $a_i x^* \leq b_i$. Now recall the last property, that force can be only exerted by walls that touch the ball x^*. This property implies that $y_i > 0$ only possible if $a_i x^* = b_i$, or in other words either $y_i^* = 0$ or $a_i x^* = b_i$ for all i. In either case $y_i^*(a_i x^*) = y_i^* b_i$, and so the last inequality above is actually equal. ■

Notice that the chain on inequalities is true for all feasible vectors x and feasible dual solutions y. The argument we just went through can be used to recognize a pair of optimal solutions.

Corollary 12 *For a solution x of $Ax \leq b$ and a vector $y \geq 0$ that satisfies $c = yA$, x and y are optimal solutions of the linear program and its dual if and only if for each i either $y_i = 0$ or $a_i x = b_i$. Or put it differently, x and y are optimal solutions of the linear program and its dual if and only if for all i we have $y_i(b_i - a_i x) = 0$.*

6 The ellipsoid method

Next we will roughly sketch one of the methods for solving linear programs, the ellipsoid method. This was the first polynomial time algorithm discovered for the problem. Its not the most efficient practically: practical algorithms either use the simplex method (which may be exponential in the worst case) or interior point methods. However, the ellipsoid method is based on a simple geometric idea, and it is the most powerful in the sense of being able to solve extensions of linear programs also in polynomial time.

The basis idea of the ellipsoid method is the following. Suppose we know that our feasible region $P = \{x : Ax \leq b\}$ is contained in a ball, say the ball $B = \{x : x^2 = \sum_i x_i^2 \leq 1\}$. If we also knew that say $x_1 \geq 0$ for all points in P, then P is contained in a half ball $B \cap \{x : x_1 \geq 0\}$, which is only half as big as B. Unfortunately, a half-ball is a much more complex object. The idea is to enclose the half ball in an ellipsoid E. The ellipsoid will still be smaller than B (tough by much less smaller than the half-ball), it will still contain the region of our interest P, and be again a simple shape, like a ball, so we will be able to recurse.

To develop the idea above, we need to find an ellipse E that encloses the half-ball as shows in the figure below. Our ellipse will be centered at $c = (\frac{1}{n+1}, 0, \ldots, 0)$. So the ball B

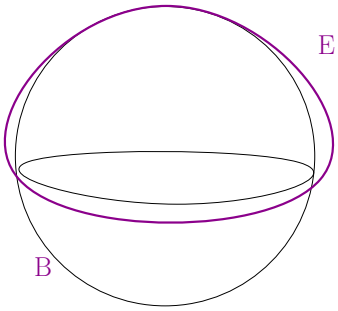

Figure 2: The ellipse E enclosing the top half of the ball B.

translated be centered at a point c would have definition $B(c) = \{x : \sum_i (x_i - c_i)^2 \leq 1\}$. The ellipse we are interested in is a bit squashed version of this ball defined as $E = \{x : \frac{n^2-1}{n^2} \sum_{i \leq 2} x_i^2 + (\frac{n+1}{n}(x_1 - \frac{1}{n+1}))^2 \leq 1\}$, where n is the dimension of the space $x \in R^n$.

Lemma 13 *The ellipse E contain the half ball $B \cap \{x : x_1 \geq 0\}$.*

Proof. First test two points $x = (1, 0 \ldots, 0)$. This point satisfies the ellipse inequality as

$$\frac{n^2-1}{n^2} \sum_{i \leq 2} x_i^2 + (\frac{n+1}{n}(x_1 - \frac{1}{n+1}))^2 = (\frac{n+1}{n}\frac{n}{n+1})^2 = 1.$$

Second, consider a point $x = (0, x_2, \ldots, x_n)$ on the "equator", i.e. where $\sum_{i \geq 0} x_i^2 = 1$. For such a point we get

$$\frac{n^2-1}{n^2} \sum_{i \leq 2} x_i^2 + (\frac{n+1}{n}(x_1 - \frac{1}{n+1}))^2 = \frac{n^2-1}{n^2} + (\frac{n+1}{n}\frac{1}{n+1})^2 = \frac{n^2-1}{n^2} + \frac{1}{n^2} = 1.$$

Finally, consider a general point x in the half ball, and let $\sum_{i \leq 2} x_i^2 = s^2$ for some value s. If $x_1 \leq \frac{1}{n+1}$, i.e., the point is below the level of the center, the argument we just used for

the equator works again. For the point is above the center, $\frac{1}{n+1} \leq x_1$ we use the fact that $x_1 \leq \sqrt{1-s^2}$. So we get

$$(x_1 - \frac{1}{n+1})^2 \leq (\sqrt{1-s^2} - \frac{1}{n+1})^2$$

In this case we get a rather complicated expression in s for our bound

$$\frac{n^2-1}{n^2}\sum_{i\leq 2} x_i^2 + (\frac{n+1}{n}(x_1 - \frac{1}{n+1}))^2 \leq \frac{n^2-1}{n^2}s^2 + (\frac{n+1}{n})^2(\sqrt{1-s^2} - \frac{1}{n+1})^2$$

Maybe the simplest way to show that this is at most 1 is to use calculus to show that the maximum of this expression on the interval $s \in [0,1]$ occurs at the two ends, and we have just seen that at $s = 0$ or 1 the value is 1. ■

Next we need to show that our ellipse E indeed has significantly smaller volume that the ball B, as we hope to claim to make progress by shrinking the volume. To do this, its useful to remember the expression for the volume of a ball with radius r in n dimension is $\gamma_n r^n$ for a constant γ that depends on the dimension. For example, in 2 dimension $\gamma_2 = \pi$, while in 3 dimension $\gamma_3 = \frac{4}{3}\pi$. We are thinking about ellipses of the form $E_0 = \{x : (\alpha_i x_i)^2 \leq 1\}$. A ball of radius r is expressed this way with $\alpha_i = 1/r$ for all i. More generally, the volume of the ellipse just defined is $V(E_0) = \gamma_n / \prod_i \alpha_i$. Using this expression we can get the ratio of the ellipse E containing the half-ball and the ball.

Lemma 14 *The ratio of the volumes of the ellipse E and the ball B is bounded by $V(E)/V(B) \leq e^{-1/4(n+1)}$*

Proof. The ball B has radius 1, so its volume is γ_n. In computing the ratio, γ_n cancels. In defining the ellipse E we used $\alpha_1 = \frac{n+1}{n}$, while α_i for $i \geq 2$ we have $\sqrt{\frac{n^2-1}{n^2}}$. So we get

$$V(E)/V(B) = \frac{n}{n+1}(\frac{n^2}{n^2-1})^{(n-1)/2}$$

To estimate this expression we can use that $1+x \approx e^x$ for small x, and use that $\frac{n}{n+1} = (1 - \frac{1}{n+1})$ and similarly $\frac{n^2}{n^2-1} = (1 + \frac{1}{n^2-1})$ so we get

$$V(E)/V(B) \approx e^{-1/(n+1)}(e^{1/(n^2-1)})^{(n-1)/2} = e^{-1/(n+1)+1/2(n+1)} = e^{-1/2(n+1)}$$

Being a bit more careful with the small error in the $1 + x \approx e^x$ approximation, decreases the bound a bit further, but we can get the bound claimed by the lemma. Unfortunately, the decrease is quite minimal. ■

Now we are ready to use the above geometry in an algorithm. To help the presentation, we will make a number of simplifying assumptions. As we make each assumption, we comment on how one may be able to solve problems without the assumption, but we will not elaborate in these further.

1. We will assume that we are looking for is known to be contained in a large ball $B_0 = \{x : x^2 \leq R^2\}$ for a parameter R. For example, if we know that the variables are $0 \leq x_i \leq 1$ than we can use $R = \sqrt{n}$. Similarly, if there are upper bounds on the variables, this implies an upper bound on R. Without any such bound, one would have to argue that if the maximum of the linear program is not infinite, it occurs in a bounded region.

2. To simplify the presentation, we will focus on just finding a feasible solution x satisfying $Ax \leq b$ without a maximization. One can incorporate an objective function of $\max cx$, for example, by adding a constraint that $cx \geq \gamma$ and binary searching on the maximum value of γ for which the system remains feasible.

3. Instead of looking for an exact solution of $Ax \leq b$, we will be satisfied with an approximate solution. Assume that we are given an error parameter $\epsilon > 0$. By dividing the inequalities in $Ax \leq b$ by appropriate constants, we can assume that each entry in the matrix is at most 1, i.e.,that $|a_{ij}| \leq 1$ for all i and j. With this assumption in place we will accept an x as a solution if for each constraint i it satisfies $a_i x \leq b_i + \epsilon$. However, the algorithm can conclude that no solution exists, assuming the original system $Ax \leq b$ has no solution.

   Notice that there are cases that it is not clear upfront what this approximate solution algorithm should output. If there is no solution to $Ax \leq b$, but there is a solution to the related system of $a_i x \leq b_i + \epsilon$ for all i the algorithm can find either answer. In most applications such an approximate solution is OK. To make the algorithm precise we would have to do two things. First find a small enough value $\epsilon > 0$ that guarantees that of $Ax \leq b$ has no solution, than $a_i x \leq b_i + \epsilon$ for all i also has no solution. Second, we need to show that for a small enough $\epsilon > 0$ a solution x of the approximate system $a_i x \leq b_i + \epsilon$ for all i can be rounded to become a solution of the original system.

The main idea of the algorithm is to start with the ball in assumption 1 that is known to contain all solutions. While we have not found a solution, we will have an ellipsoid E_i that contains all solutions. In each iteration, we test the center c^i of our ellipsoid E_i. If c^i is an (approximate) solution to our system, we return $x = c^i$ and we are done. If c^i is not a solution, than it must violate one of the constraints of the system $a_i c^i > b_i + \epsilon$. In this case, all solutions, even all approximate solutions, are in the half of the ellipsoid defined by the cut through the center of our ellipsoid $a_i x \leq a_i c^i$. We then define the next ellipsoid E_{i+1} to contain this half-ellipsoid $E_i \cap \{x : a_i x \leq a_i c^i\}$.

We defined an enclosing ellipsoid for a half-ball, with the ball centered at 0, and the half defined by one of the coordinate directions. However, now we need this for an ellipsoid with a different center, and a direction that may not be one of the coordinate directions. While working out the algebraic expression for this new ellipsoid is a bit complex, the geometric idea is simple via a geometric steps. To see how this translates to an algebraic expression, you may want to look at the notes by Santosh Vempala posted in the course web page.

- By translating the space we can assume that any given point c is the origin.

- For an ellipsoid E defined by the inequality $\sum_i(\alpha_i x_i) \leq 1$ we can stretch the coordinates, by using a new coordinate system with $y_i = \alpha_i x_i$, and the ellipsoid E becomes the unit ball in the new coordinate system.
- Finally, we want to take a half-space defined by an arbitrary vector $ax \geq 0$, rather than a coordinate direction. To do this, we can again change coordinate systems, by letting the unit vector in direction a become our first coordinate, and extending this to an orthogonal system of coordinates for the space.

Using these reductions allows us to take the ellipsoid defined in the beginning of this section for the unit ball as part of an iterative algorithm. From Lemma 14 we know that the volume in each iteration decreases by at least a bit. The last question we need to answer is how many iterations we need before we can conclude. In fact, we need to wonder about how the algorithm can conclude. It may find that the centers of one of the ellipsoids is a solution, and hence can terminate. But how does it terminate when there is no solution? The idea is to note that if $Ax \leq b$ has a solution, than the set of approximate solutions must have a volume δ. This is useful, as if we ever find that our ellipsoid E_i at some iteration has volume smatter than δ than we can conclude that $Ax \leq b$ cannot have a solution, and hence can terminate.

Lemma 15 *If the system of inequalities $\{x : Ax \leq b\}$ has a solution, and $|a_{ij}| \leq 1$ for all entries than the volume of the set $\{x : a_i x \leq b_i + \epsilon$ for all $i\}$ must be at least $\delta = (2\epsilon/n)^n$.*

Proof. Consider a solution x^* such that $Ax^* \leq b$. We define a small box around x^* as

$$B(x^*) = \{x : |x_i - x_i^*| \leq \epsilon/n \text{ for all } i\}$$

Observe that all points $x \in B(x^*)$ must satisfy the approximate inequalities, and the volume $V(B(x^*))$ is exactly δ proving the lemma. ■

Theorem 16 *Under the simplifying assumptions 1-3 made above, the ellipsoid algorithm solve the problem of finding a feasible solution to a system of inequalities in n dimension in $O(n^2 \log(Rn/\epsilon)$ iterations.*

Proof. By the 1st assumption we start out with a ball of volume $R^{n/2}$. By Lemma 14 each iteration decreases the volume of our ellipsoid by a factor of $e^{1/2(n+1)}$, so $2(n+1)$ iterations decrease the volume by a factor of e, i.e., by a constant factor. what is the range of volume that we need to decrease? we need to go from $R^{n/2}$ to $(2\epsilon/n)^n$, a range less than $(Rn/\epsilon)^n$. This happens after the volume decreases $\log((Rn/\epsilon)^n) = n\log(Rn/epsilon)$ times by a constant factor, so overall will take $O(n^2 \log(Rn/\epsilon)$ iterations as claimed. ■

7 Linear Programming and Randomized Rounding

As an application of linear programming, we will consider the following disjoint path problem. Given a directed graph $G = (V, E)$, capacities on the edges $c_e \geq 0$ for $e \in E$, and pairs of

nodes $s_i, t_i \in V$ for $i = 1, \ldots, k$, the problem is to find paths P_i from s_i to t_i that don't use any edge e more than c_e times. For example, when $c_e = 1$ for all e we are looking for disjoint paths. There may not be k such paths in G, so we will try to find as many as possible.

The first natural idea is to reduce the problem to max-flow, but unfortunately, this won't work out so well. To see why, note that the natural reduction would add a supersource s with edges to each s_i with capacity 1, and a supersink t with edges from each t_i to t with capacity 1, and than find a max-flow from s to t. We have seen that a flow can be decomposed into paths from s to t, with integer capacities, each flow will carry an integer amount of flow, and with the source and sink edges having capacity 1, each paths will have one unit of flow. What goes wrong is the pairing of the sources and sinks. There is nothing guaranteeing that the path starting at s_i ends at its pair t_i, rather than at some other terminal t_j. In fact, this approach cannot as finding disjoint paths (when $c_e = 1$ for all e) is an NP-complete problem. Here we will find a close to optimal solution when the capacities are high enough.

The high level idea is to take advantage that linear programs can be solved in polynomial time. The above paths problem can be formulated as an integer problem. We solve a fractional version, and then will want to use the fractional solution to get an integer solution. The most natural way to formulate the path problem is to have a variable x_P associated with every paths. Let $\mathcal{P}_i$ denote the set of paths in G from s_i to t_i. Then we write

$$\begin{aligned} x_P &\geq 0 \text{ for all } P \in \cup_i \mathcal{P}_i \\ \sum_{P:e\in P} x_P &\leq c_e \text{ for all } e \in E \\ \sum_{P\in\mathcal{P}_i} x_P &\leq 1 \text{ for all } i \\ max \sum_P x_P & \end{aligned}$$

The inequality for the edges enforces the capacities, the other set of inequalities is asking to select at most one total of the paths between any source and sink. While we didn't include the $x_P \leq 1$ constraint, this is implied by the inequality that is asking to select at most one total of the paths between any source and sink. So integer solutions will have x_P either 0 or 1, and we can think of the paths P with $x_P = 1$ as being selected.

Lemma 17 *Integer solutions to the above inequalities are in one-to-one correspondence to paths that satisfy the capacity constraints.*

However, unlike maximum flow, this linear program does **not** solve in integers, the optimal solution will result in fractional values. Before we start thinking about how to use the solution to this linear program, we need to worry if we can solve it at all. The problem is that the linear program as stated can have exponentially many variables, one for each paths. We will give a compact, polynomial size version by using the traditional flow formulation with flows on edge, but doing this separately for each edge e. We'll use variables $f_i(e)$ to denote the amount of flow from s_i to t_i on edges e, that is $f_i(e) = \sum_{P\in\mathcal{P}_i:e\in P} x_P$.

$$
\begin{aligned}
f_i(e) &\geq 0 \text{ for all } e \in E \text{ and all } i \\
\sum_i f_i(e) &\leq c_e \text{ for all } e \in E \\
\sum_{e \text{ enters } v} f_i(e) - \sum_{e \text{ leaves } v} f_i(e) &= 0 \text{ for all } i \text{ and all } v \in V,\ v \neq s_i, t_i \\
\sum_{e \text{enters } t_i} f_i(e) - \sum_{e \text{ leaves } t_i} f_i(e) &\leq 1 \text{ for all } i \\
\max \sum_i \sum_{e \text{ enters } t_i} f_i(e) - \sum_{e \text{ leaves } t_i} f_i(e) &
\end{aligned}
$$

We have a separate set of flow conservation constraints for each flow f_i, a bound of 1 on flow between any given source-sink paths, a joint capacity constraint founding the total flow $\sum_i f_i(e)$, and the goal is to maximize the total flow. The advantage of this linear program is its compact size. For a graph with n nodes and m edges, we have mk nonnegative variables, and $m + nk$ constraints. Integer solutions to this linear program also are solutions to the original path problem. Its not hard to see that in an integer solution the set of edges with $f_i(e) = 1$ for a given index i can form cycles and possibly a single path from s_i to t_i. Cycles do not contribute to the objective value, and path contribute 1. So ignoring the possible cycles in a solution, we get the following.

Lemma 18 *Integer solutions to the new inequalities correspondence to paths that satisfy the capacity constraints with the number of paths equal to the objective value.*

Further, we have seen that an edge-flow can be converted to be a path flow of equal value, so any (fractional) solution to the second linear program can be converted to a solution to the first linear program in polynomial time. Recall, the way we get around the exponential number of variables is that our solution will have most of them set to 0, and will define the solution by listing the non-zero values only.

Lemma 19 *The two linear programs have the same optimal value, and a solution to one can be converted to a solution to the other in polynomial time. The resulting solution of the path flow linear program will have only polynomially many paths with nonzero x_P values. Further, the linear programming optimum value is at least as high as the maximum number of paths that can be selected.*

Now suppose we have a solution to our path flow linear program. The next question is, how is this useful to find real paths. The idea is to use the variables x_P for all $P \in \mathcal{P}_i$ as probabilities. For each i independently, we want to select at most one path from s_i to t_i, selecting a given (s_i, t_i) path with probability x_P. This is possible as

$$\sum_{P \in \mathcal{P}_i} x_P \leq 1$$

Note that if we have a strict inequality here, with some probability we will not select any (s_i, t_i) path. There are some basic facts we can get about this selection method using linearity of expectation.

Lemma 20 *The expected number of paths selected by our random selection is $\sum_P x_P$, and the expected number of paths using any edge e is at most c_e for each e.*

Proof. A path P contributes to the expected number of path by its probability of being selected, so the expected number of path overall, by linearity of expectation is $\sum_P x_P$ as claimed. The expected number of paths using an edge e is $\sum_{P:e\in P} x_P$, which is bounded by c_e by our capacity constraint. ■

We would like to show that the number of paths on any edge is below its capacity high probability. This simply won't be true if the expectation is an high as c_e. To help with this, we need to modify the algorithm to use $(1-\epsilon)x_P$ as the probability of selecting path P for all P. This decreases the expected number of paths selected by a factor of $(1-\epsilon)$, but allows us to bound the probability that the number of paths exceeds c_e on any edge.

Lemma 21 *If the expected number of paths on an edge e is at most $(1-\epsilon)c_e$, and $(1-\epsilon)c_e \geq 2\epsilon^{-2}\log m$ than the probability that there are more paths than c_e using edge e is at most $1/m^2$.*

Proof. We use Chernoff bound. Focusing on edge e let $X_i = 1$ of the s_i to t_i path selected uses edge e. By construction, the X_i variables are 0/1 and independent, and the number of paths using e is $X = \sum_i X_i$. Also note that $E(X) \leq (1-\epsilon)c_e$, which we can call $\mu = (1-\epsilon)c_e$, and $(1+\epsilon)\mu = (1-\epsilon^2)c_e \leq c_e$, so we get

$$Pr(X > c_e) \leq (e^{-0.5\epsilon^2})^{(1-\epsilon)c_e} \leq \frac{1}{m^2}$$

as claimed. ■

Theorem 22 *Assuming that all capacities $c_e \geq \epsilon^{-2}\log m$, then the above randomized rounding method, solving the linear program, using $(1-\epsilon')x_P$ for some $\epsilon' > 0$ as probabilities independently for each i, finds a solution to the path problem with high at high probability (say probability at east 3/4) using a number of path no less than a factor $1 - O(\epsilon)$ less than the optimal.*

Proof. We have seen by Lemma 19 that the expected number of paths using this method is at least $(1-\epsilon')$ times the optimal. For each edge, Lemma 21 shows that, we can get the probability of a single violated very low: $Pr(X > c_e) \leq \frac{1}{m^2}$. Let $X(e)$ denote the number of paths on edge e. Using Lemma 21 and the union bound for all edges, we get

$$Prob(\exists e : X_e > c_e) \leq \sum_e Prob(X_e > c_e) \leq m\frac{1}{m^2} = \frac{1}{m}.$$

Similarly, the probability that the total number of paths is below $(1-\epsilon')$ times its expectation can also be bounded by $\frac{1}{m^2}$ due to the lower bound version of the Chernoff bound. Using the union bound on these two events, the probability that the method has $(1-\epsilon')^2 \approx (1-2\epsilon')$ times the maximum possible number of path, and all edges have at most c_e paths is at least $1 - \frac{1}{m} - \frac{1}{m^2}$, which is high enough assuming the graph is not too small. ■

Cornell University, Fall 2016	CS 6820: Algorithms
Supplementary lecture notes on linear programming	26 Sep–28 Sep

1 The Simplex Method

We will present an algorithm to solve linear programs of the form

$$\begin{aligned} \text{maximize} \quad & c^\intercal x \\ \text{subject to} \quad & Ax \preceq b \\ & x \succeq 0 \end{aligned} \tag{1}$$

assuming that $b \succeq 0$, so that $x = 0$ is guaranteed to be a feasible solution. Let n denote the number of variables and let m denote the number of constraints.

A simple transformation modifies any such linear program into a form such that each variable is constrained to be non-negative, and all other linear constraints are expressed as *equations* rather than *inequalities*. The key is to introduce additioinal variables, called *slack variables* which account for the difference between and left and right sides of each inequality in the original linear program. In other words, linear program (1) is equivalent to

$$\begin{aligned} \text{maximize} \quad & c^\intercal x \\ \text{subject to} \quad & Ax + y = b \\ & x, y \succeq 0 \end{aligned} \tag{2}$$

where $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$.

The solution set of $\{Ax + y = b,\ x \succeq 0,\ y \succeq 0\}$ is a polytope in the $(n+m)$-dimensional vector space of ordered pairs $(x, y) \in \mathbb{R}^n \times \mathbb{R}^m$. The simplex algorithm is an iterative algorithm to solve linear programs of the form (2) by walking from vertex to vertex, along the edges of this polytope, until arriving at a vertex which maximizes the objective function $c^\intercal x$.

To illustrate the simplex method, for concreteness we will consider the following linear program.

$$\begin{aligned} \text{maximize} \quad & 2x_1 + 3x_2 \\ \text{subject to} \quad & x_1 + x_2 \leq 8 \\ & 2x_1 + x_2 \leq 12 \\ & x_1 + 2x_2 \leq 14 \\ & x_1, x_2 \geq 0 \end{aligned}$$

This LP has so few variables, and so few constraints, it is easy to solve it by brute-force enumeration of the vertices of the polytope, which in this case is a 2-dimensional polygon.

The vertices of the polygon are $\left[\begin{smallmatrix}0\\7\end{smallmatrix}\right]$, $\left[\begin{smallmatrix}2\\6\end{smallmatrix}\right]$, $\left[\begin{smallmatrix}4\\4\end{smallmatrix}\right]$, $\left[\begin{smallmatrix}6\\0\end{smallmatrix}\right]$, $\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]$. The objective function $2x_1 + 3x_2$ is maximized at the vertex $\left[\begin{smallmatrix}2\\6\end{smallmatrix}\right]$, where it attains the value 22. It is also easy to certify that this is the optimal value, given that the value is attained at $\left[\begin{smallmatrix}2\\6\end{smallmatrix}\right]$: simply add together the inequalities

$$\begin{aligned} x_1 + x_2 &\leq 8 \\ x_1 + 2x^2 &\leq 14 \end{aligned}$$

to obtain

$$2x_1 + 3x_2 \leq 22,$$

which ensures that no point in the feasible set attains an objective value greater than 22.

To solve the linear program using the simplex method, we first apply the generic transformation described earlier, to rewrite it in *equational form* as

$$\begin{aligned} \text{maximize} \quad & 2x_1 + 3x_2 \\ \text{subject to} \quad & x_1 + x_2 + y_1 = 8 \\ & 2x_1 + x_2 + y_2 = 12 \\ & x_1 + 2x_2 + y_3 = 14 \\ & x_1, x_2, y_1, y_2, y_3 \geq 0 \end{aligned}$$

From now on, we will be choosing a subset of two of the five variables (called the *basis*), setting them equal to zero, and using the linear equations to express the remaining three variables, as well as the objective function, as a function of the two variables in the basis. Initially the basis is $\{x_1, x_2\}$ and the linear program can be written in the form

$$\begin{aligned} \text{maximize} \quad & 2x_1 + 3x_2 \\ \text{subject to} \quad & y_1 = 8 - x_1 - x_2 \\ & y_2 = 12 - 2x_1 - x_2 \\ & y_3 = 14 - x_1 - 2x_2 \\ & x_1, x_2, y_1, y_2, y_3 \geq 0 \end{aligned}$$

which emphasizes that each of y_1, y_2, y_3 is determined as a function of x_1, x_2. Now, as long as the basis contains a variable which has a positive coefficient in the objective function, we select one such variable and greedily increasing its value until one of the non-negativity constraints becomes tight. At that point, one of the other variables attains the value zero: it enters the basis, and the variable whose value we increased leaves the basis. For example, we could choose to increase x_1 from 0 to 6, at which point $y_2 = 0$. Then the new basis becomes $\{y_2, x_2\}$. Rewriting the equation $y_2 = 12 - 2x_1 - x_2$ as

$$x_1 = 6 - \frac{1}{2}y_2 - \frac{1}{2}x_2, \tag{3}$$

we may substitute the right side of (3) in place of x_1 everywhere in the above linear program, arriving at the equivalent form

$$\begin{aligned}
&\text{maximize} && 12 - y_2 + 2x_2 \\
&\text{subject to} && y_1 = 2 + \tfrac{1}{2}y_2 - \tfrac{1}{2}x_2 \\
& && x_1 = 6 - \tfrac{1}{2}y_2 - x_2 \\
& && y_3 = 8 + \tfrac{1}{2}y_2 - \tfrac{3}{2}x_2 \\
& && x_1, x_2, y_1, y_2, y_3 \geq 0
\end{aligned}$$

At this point, x_2 still has a positive coefficient in the objective function, so we increase x_2 from 0 to 4, at which point $y_1 = 0$. Now x_2 leaves the basis, and the new basis is $\{y_1, y_2\}$. We use the equation $x_2 = 4 + y_2 - 2y_1$ to substitute a function of the basis variables in place of x_2 everywhere it appears, arriving at the new linear program

$$\begin{aligned}
&\text{maximize} && 20 - 4y_1 + y_2 \\
&\text{subject to} && x_2 = 4 + y_2 - 2y_1 \\
& && x_1 = 4 - y_2 + y_1 \\
& && y_3 = 2 - y_2 + 3y_1 \\
& && x_1, x_2, y_1, y_2, y_3 \geq 0
\end{aligned}$$

Now we increase y_2 from 0 to 2, at which point $y_3 = 0$ and the new basis is $\{y_1, y_3\}$. Substituting $y_2 = 2 - y_3 + 3y_1$ allows us to rewrite the linear program as

$$\begin{aligned}
&\text{maximize} && 22 - y_1 - y_3 \\
&\text{subject to} && x_2 = 6 + y_1 - y_3 \\
& && x_1 = 2 - 2y_1 + y_3 \\
& && y_2 = 2 + 3y_1 - y_3 \\
& && x_1, x_2, y_1, y_2, y_3 \geq 0
\end{aligned} \tag{4}$$

At this point, there is no variable with a positive coefficient in the objective function, and we stop.

It is trivial to verify that the solution defined by the current iteration—namely, $x_1 = 2$, $x_2 = 6$, $y_1 = 0$, $y_2 = 2$, $y_3 = 0$—is optimal. The reason is that we have managed to write the objective function in the form $22 - y_1 - y_3$. Since the coefficient on each of the variables y_1, y_3 is negative, and y_1 and y_3 are constrained to take non-negative values, the largest possible value of the objective function is achieved by setting both y_1 and y_3 to zero, as our solution does.

More generally, if the simplex method terminates, it means that we have found an equivalent representation of the original linear program (2) in a form where the objective function

attaches a non-positive coefficient to each of the basis variables. Since the basis variables are required to be non-negative, the objective function is maximized by setting all the basis variables to zero, which certifies that the solution at the end of the final iteration is optimal.

Note that, in our running example, the final objective function assigned coefficient -1 to both y_1 and y_3. This is closely related to the fact that the simple "certificate of optimality" described above (before we started running the simplex algorithm) we obtained by summing the first and third inequalities of the original linear program, each with a coefficient of 1. We will see in the following section that this is not a coincidence.

Before leaving this discussion of the simplex method, we must touch upon a subtle issue regarding the question of whether the algorithm always terminates. A basis is an n-element subset of $n + m$ variables, so there are at most $\binom{n+m}{n}$ bases; if we can ensure that the algorithm never returns to the same basis as in a previous iteration, then it must terminate. Note that each basis determines a unique point $(x, y) \in \mathbb{R}^{n+m}$—defined by setting the basis variables to zero and assigning to the remaining variables the unique values that satisfy the equation $Ax + b = y$—and as the algorithm proceeds from basis to basis, the objective function value at the corresponding points never decreases. If the objective function strictly increases when moving from basis B to basis B', then the algorithm is guaranteed never to return to basis B, since the objective function value is now strictly greater than its value at B, and it will never decrease. On the other hand, it is possible for the simplex algorithm to shift from one basis to a different basis with the same objective function value; this is called a *degenerate pivot*, and it happens when the set of variables whose value is 0 at the current solution is a strict superset of the basis.

There exist *pivot rules* (i.e., rules for selecting the next basis in the simplex algorithm) that are designed to avoid infinite loops of degenerate pivots. Perhaps the simplest such rule is *Bland's rule*, which always chooses to remove from the basis the lowest-numbered variable that has a positive coefficient in the objective function. (And, in case there is more than one variable that may move into the objective function to replace it, the rule also chooses the lowest-numbered such variable.) Although the rule is simple to define, proving that it avoids infinite loops is not easy, and we will omit the proof from these notes.

2 The Simplex Method and Strong Duality

An important consequence of the correctness and termination of the simplex algorithm is *linear programming duality*, which asserts that for every linear program with a maximization objective, there is a related linear program with a minimization objective whose optimum matches the optimum of the first LP.

Theorem 1. *Consider any pair of linear programs of the form*

$$\begin{array}{llcll} \text{maximize} & c^\intercal x & & \text{minimize} & b^\intercal \eta \\ \text{subject to} & Ax \preceq b & \textit{and} & \text{subject to} & A^\intercal \eta \succeq c \\ & x \succeq 0 & & & \eta \succeq 0 \end{array} \tag{5}$$

If the optimum of the first linear program is finite, then both linear programs have the same optimum value.

Proof. Before delving into the formal proof, the following intuition is useful. If a_i denotes the i^{th} row of the matrix A, then the relation $Ax \preceq b$ can equivalently be expressed by stating that $a_i^\intercal x \leq b_i$ for $j = 1, \ldots, m$. For any m-tuple of non-negative coefficients $\eta_1, \ldots, \eta_m$, we can form a weighted sum of these inequalities,

$$\sum_{j=1}^{m} \eta_i a_i^\intercal x \leq \sum_{j=1}^{m} \eta_i b_i, \tag{6}$$

obtaining an inequality implied by $Ax \preceq b$. Depending on the choice of weights $\eta_1, \ldots, \eta_m$, the inequality (6) may or may not imply an upper bound on the quantity $c^\intercal x$, for all $x \succeq 0$. The case in which (6) implies an upper bound on $c^\intercal x$ is when, for each variable x_j ($j = 1, \ldots, n$), the coefficient of x_j on the left side of (6) is greater than or equal to the coefficient of x_j in the expression $c^\intercal x$. In other words, the case in which (6) implies an upper bound on $c^\intercal x$ for all $x \succeq 0$ is when

$$\forall j \in \{1, \ldots, n\} \quad \sum_{i=1}^{m} \eta_i a_{ij} \geq c_j. \tag{7}$$

We can express (6) and (7) more succinctly by packaging the coefficients of the weighted sum into a vector, η. Then, inequality (6) can be rewritten as

$$\eta^\intercal A x \leq \eta^\intercal b, \tag{8}$$

and the criterion expressed by (7) can be rewritten as

$$\eta^\intercal A \succeq c^\intercal. \tag{9}$$

The reasoning surrounding inequalities (6) and (7) can now be summarized by saying that for any vector $\eta \in \mathbb{R}^m$ satisfying $\eta \succeq 0$ and $\eta^\intercal A \succeq c^\intercal$, we have

$$c^\intercal x \leq \eta^\intercal A x \leq \eta^\intercal b \tag{10}$$

for all $x \succeq 0$ satisfying $Ax \preceq b$. (In hindsight, proving inequality (10) is trivial using the properties of the vector ordering $\preceq$ and our assumptions about x and η.)

Applying (10), we may immediately conclude that the minimum of $\eta^\intercal b$ over all $\eta \succeq 0$ satisfying $\eta^\intercal A \succeq c^\intercal$, is greater than or equal to the maximum of $c^\intercal x$ over all $x \succeq 0$ satisfying $Ax \preceq b$. That is, the optimum of the first LP in (5) is less than or equal to the optimum of the second LP in (5), a relation known as *weak duality*.

To prove that the optima of the two linear programs are equal, as asserted by the theorem, we need to furnish vectors x, η satisfying the constraints of the first and second linear programs in (5), respectively, such that $c^\intercal x = b^\intercal \eta$. To do so, we will make use of the simplex algorithm and its termination condition. At the moment of termination, the objective function has been rewritten in a form that has no positive coefficient on any variable. In other words, the objective function is written in the form $v - \xi^\intercal x - \eta^\intercal y$ for some coefficient vectors $\xi \in \mathbb{R}^n$ and $\eta \in \mathbb{R}^m$ such that $\xi, \eta \succeq 0$.

An invariant of the simplex algorithm is that whenever it rewrites the objective function, it preserves the property that the objective function value matches $c^\intercal x$ for all pairs $(x, y) \in \mathbb{R}^n \times \mathbb{R}^m$ such that $Ax + y = b$. In other words, we have

$$\forall x \in \mathbb{R}^n \quad v - \xi^\intercal x - \eta^\intercal (b - Ax) = c^\intercal x. \tag{11}$$

Equating the constant terms on the left and right sides, we find that $v = \eta^\intercal b$. Equating the cofficient of x_j on the left and right sides for all j, we find that $\eta^\intercal A = \xi^\intercal + c^\intercal \succeq c^\intercal$. Thus, the vector η satisfies the constraints of the second LP in (5).

Now consider the vector (x, y) which the simplex algorithm outputs at termination. All the variables having a non-zero coefficient in the expression $-\xi^\intercal x - \eta^\intercal y$ belong to the algorithm's basis, and hence are set to zero in the solution (x, y). This means that

$$v = v - \xi^\intercal x - \eta^\intercal y = c^\intercal x$$

and hence, using the relation $v = \eta^\intercal b$ derived earlier, we have $c^\intercal x = b^\intercal \eta$ as desired. $\square$

Cornell University, Fall 2012	CS 6820: Algorithms
Lecture notes on approximation algorithms	October 10–15, 2010

1 Introduction: Approximation Algorithms

For many important optimization problems, there is no known polynomial-time algorithm to compute the exact optimum. In fact, when we discuss the topic of NP-completeness later in the semester, we'll see that a great many such problems are all equivalently hard to solve, in the sense that the existence of a polynomial-time algorithm for solving any one of them would imply polynomial-time algorithms for all the rest.

The study of approximation algorithms arose as a way to circumvent the apparent hardness of these problems by relaxing the algorithm designer's goal: instead of trying to compute an exactly optimal solution, we aim to compute a solution whose value is as close as possible to that of the optimal solution. However, in some situations it is desirable to run an approximation algorithm even when there exists a polynomial-time algorithm for computing an exactly optimal solution. For example, the approximation algorithm may have the benefit of faster running time, a lower space requirement, or it may lend itself more easily to a parallel or distributed implementation. These considerations become especially important when computing on "big data," where the input size is so astronomical that a running time which is a high-degree polynomial of the input size (or even quadratic, for that matter) cannot really be considered an efficient algorithm, at least on present-day hardware.

To make the definition of approximation algorithms precise, we say that an algorithm ALG for a maximization problem is an α-approximation algorithm (or that its approximation factor is α) if the following inequality holds for every input instance x:

$$\mathrm{ALG}(x) \leq \mathrm{OPT}(x) \leq \alpha \cdot \mathrm{ALG}(x).$$

Here $\mathrm{OPT}(x)$ denotes the value of the problem's objective function when evaluated on the optimal solution to input instance x, and $\mathrm{ALG}(x)$ denotes the algorithm's output when it runs on input instance x. Note that the definition only requires the algorithm to output a number (approximating the value of the optimal solution) and not the approximate solution itself. In most cases, it is possible to design the algorithm so that it also outputs a solution attaining the value $\mathrm{ALG}(x)$, but in these notes we adopt a definition of approximation algorithm that does not require the algorithm to do so.

Similarly, for a minimization problem, an α-approximation algorithm must satisfy

$$\mathrm{OPT}(x) \leq \mathrm{ALG}(x) \leq \alpha \cdot \mathrm{OPT}(x).$$

Note that in both cases the approximation factor α is a number greater than or equal to 1.

2 Approximation Algorithms Based on Linear Programming

Linear programming is an extremely versatile technique for designing approximation algorithms, because it is one of the most general and expressive problems that we know how to solve in polynomial time. In this section we'll discuss three applications of linear programming to the design and analysis of approximation algorithms.

2.1 LP Rounding Algorithm for Weighted Vertex Cover

In an undirected graph $G = (V, E)$, if $S \subseteq V$ is a set of vertices and e is an edge, we say that S *covers* e if at least one endpoint of e belongs to S. We say that S is a *vertex cover* if it covers every edge. In the weighted vertex cover problem, one is given an undirected graph $G = (V, E)$ and a weight $w_v \geq 0$ for each vertex v, and one must find a vertex cover of minimum combined weight.

We can express the weighted vertex cover problem as an integer program, by using *decision variables* x_v for all $v \in V$ that encode whether $v \in S$. For any set $S \subseteq V$ we can define a vector $\mathbf{x}$, with components indexed by vertices of G, by specifying that

$$x_v = \begin{cases} 1 & \text{if } v \in S \\ 0 & \text{otherwise.} \end{cases}$$

S is a vertex cover if and only if the constraint $x_u + x_v \geq 1$ is satisfied for every edge $e = (u, v)$. Conversely, if $\mathbf{x} \in \{0, 1\}^V$ satisfies $x_u + x_v \geq 1$ for every edge $e = (u, v)$ then the set $S = \{v \mid x_v = 1\}$ is a vertex cover. Thus, the weighted vertex cover problem can be expressed as the following integer program.

$$\begin{aligned} \min \quad & \textstyle\sum_{v \in V} w_v x_v & & \\ \text{s.t.} \quad & x_u + x_v \geq 1 & & \forall e = (u, v) \in E \\ & x_v \in \{0, 1\} & & \forall v \in V \end{aligned} \tag{1}$$

To design an approximation algorithm for weighted vertex cover, we will transform this integer program into a linear program by relaxing the constraint $x_v \in \{0, 1\}$ to allow the variables x_v to take fractional values.

$$\begin{aligned} \min \quad & \textstyle\sum_{v \in V} w_v x_v & & \\ \text{s.t.} \quad & x_u + x_v \geq 1 & & \forall e = (u, v) \in E \\ & x_v \geq 0 & & \forall v \in V \end{aligned} \tag{2}$$

It may seem more natural to replace the constraint $x_v \in \{0, 1\}$ with $x_v \in [0, 1]$ rather than $x_v \geq 0$, but the point is that an optimal solution of the linear program will never assign any of the variables x_v a value strictly greater than 1, because the value of any such variable could always be reduced to 1 without violating any constraints, and this would only improve the objective function $\sum_v w_v x_v$. Thus, writing the constraint as $x_v \geq 0$ rather than $x_v \in [0, 1]$ is without loss of generality.

It is instructive to present an example of a fractional solution of (2) that achieves a strictly lower weight than any integer solution. One such example is when G is a 3-cycle with vertices u, v, w, each having weight 1. Then the vector $\mathbf{x} = \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}\right)$ satisfies all of the constraints of (2) and the objective function evaluates to $\frac{3}{2}$ at $\mathbf{x}$. In contrast, the minimum weight of a vertex cover of the 3-cycle is 2.

We can solve the linear program (2) in polynomial time, but as we have just seen, the solution may be fractional. In that case, we need to figure out how we are going to post-process the fractional solution to obtain an actual vertex cover. In this case, the natural idea of rounding to the nearest integer works. Let $\mathbf{x}$ be an optimal solution of the linear program (2) and define

$$\tilde{x}_v = \begin{cases} 1 & \text{if } x_v \geq 1/2 \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

Now let $S = \{v \mid \tilde{x}_v = 1\}$. Note that S is a vertex cover because for every edge $e = (u, v)$ the constraint $x_u + x_v \geq 1$ implies that at least one of x_u, x_v is greater than or equal to $1/2$.

Finally, the analyze the approximation ratio of this algorithm, we observe that the rounding rule (3) has the property that for all v,

$$\tilde{x}_v \leq 2x_v.$$

Letting S denote the vertex cover chosen by our LP rounding algorithm, and letting OPT denote the optimum vertex cover, we have

$$\sum_{v \in S} w_v = \sum_{v \in V} w_v \tilde{x}_v \leq 2 \sum_{v \in V} w_v x_v \leq 2 \sum_{v \in \mathrm{OPT}} w_v,$$

where the final inequality holds because the fractional optimum of the linear program (2) must be less than or equal to the optimum of the integer program (1) because its feasible region is at least as big.

2.2 Primal-Dual Algorithm for Weighted Vertex Cover

The algorithm presented in the preceding section runs in polynomial time, and we have seen that it outputs a vertex cover whose weight is at most twice the weight of the optimum vertex cover, a fact that we express by saying that its *approximation factor* is 2.

However, the algorithm needs to solve a linear program and although this can be done in polynomial time, there are much faster ways to compute a vertex cover with approximation factor 2 without solving the linear program. One such algorithm, that we present in this section, is a *primal-dual approximation algorithm*, meaning that it makes choices guided by the linear program (2) and its dual but does not actually solve them to optimality.

Let us write the linear programming relaxation of weighted vertex cover once again, along with its dual.

$$\begin{aligned} \min \quad & \textstyle\sum_{v \in V} w_v x_v & & \\ \text{s.t.} \quad & x_u + x_v \geq 1 & \forall e = (u, v) \in E & \qquad (4) \\ & x_v \geq 0 & \forall v \in V & \end{aligned}$$

$$\begin{aligned} \max \quad & \textstyle\sum_{e \in E} y_e & & \\ \text{s.t.} \quad & \textstyle\sum_{e \in \delta(v)} y_e \leq w_v & \forall v \in V & \qquad (5) \\ & y_e \geq 0 & \forall e \in E & \end{aligned}$$

Here, the notation $\delta(v)$ denotes the set of all edges having v as an endpoint. One may interpret the dual LP variable y_e as *prices* associated to the edges, and one may interpret w_v as the *wealth* of vertex v. The dual constraint $\sum_{e \in \delta(v)} y_e \leq w_v$ asserts that v has enough wealth to pay for all of the edges incident to it. If edge prices satisfy all the constraints of (5) then *every* vertex has enough wealth to pay for its incident edges, and consequently every vertex set S has enough combined wealth to pay for all of the edges covered by S. In particular, if S is a vertex cover then the combined wealth of the vertices in S must be at least $\sum_{e \in E} y_e$, which is a manifestation of *weak duality*: the optimum value of the dual LP is a lower bound on the optimum value of the primal LP.

The dual LP insists that we maximize the combined price of all edges, subject to the constraint that each vertex has enough wealth to pay for all the edges it covers. Rather than exactly maximizing the combined price of all edges, we will set edge prices using a natural (but suboptimal) greedy heuristic: go through the edges in arbitrary order, increasing the price of each one as much as possible without violating the dual constraints. This results in the following algorithm.

Algorithm 1 Primal-dual algorithm for vertex cover

```
Initialize S = ∅, y_e = 0 ∀e ∈ E, s_v = 0 ∀v ∈ V.
for all e ∈ E do
    δ = min{w_u − s_u, w_v − s_v}
    y_e = y_e + δ
    s_u = s_u + δ
    s_v = s_v + δ
    if s_u = w_u then
        S = S ∪ {u}
    end if
    if s_v = w_v then
        S = S ∪ {v}
    end if
end for
return S
```

The variables s_v keep track of the sum $\sum_{e \in \delta(v)} y_e$ (i.e., the left-hand side of the dual constraint corresponding to vertex v) as it grows during the execution of the algorithm. The rule for updating S by inserting each vertex v such that $s_v = w_v$ is inspired by the principle of *complementary slackness* from the theory of linear programming duality: if x^* is an optimal solution of a primal linear program and y^* is an optimal solution of the dual, then for every i such that $x_i^* \neq 0$ the i^{th} dual constraint must be satisfied with equality by y^*; similarly, for every j such that $y_j^* \neq 0$, the j^{th} primal constraint is satisfied with equality by x^*. Thus, it is natural that our decisions of which vertices to include in our vertex cover (primal solution) should be guided by keeping track of which dual constraints are tight ($s_v = w_v$).

It is clear that each iteration of the main loop runs in constant time, so the algorithm runs in linear time. At the end of the loop processing edge $e = (u, v)$, at least one of the vertices u, v must belong to S. Therefore, S is a vertex cover. To conclude the analysis we need to prove that the approximation factor is 2. To do so, we note the following loop invariants — statements that hold at the beginning and end of each execution of the **for** loop, though not necessarily in the middle. Each of them is easily proven by induction on the number of iterations of the **for** loop.

1. **y** is a feasible vector for the dual linear program.
2. $s_v = \sum_{e \in \delta(v)} y_e$.
3. $S = \{v \mid s_v = w_v\}$.
4. $\sum_{v \in V} s_v = 2 \sum_{e \in E} y_e$.

Now the proof of the approximation factor is easy. Recalling that $\sum_{e \in E} y_e \leq \sum_{v \in \text{OPT}} w_v$ by weak duality, we find that

$$\sum_{v \in S} w_v = \sum_{v \in S} s_v \leq \sum_{v \in V} s_v = 2 \sum_{e \in E} y_e \leq 2 \sum_{v \in \text{OPT}} w_v.$$

2.3 Greedy Algorithm for Weighted Set Cover

Vertex cover is a special case of the *set cover* problem, in which there is a set U of n elements, and there are m subsets $S_1, \ldots, S_m \subseteq U$, with positive weights $w_1, \ldots, w_m$. The goal is to choose a subcollection of the m subsets (indexed by an index set $\mathscr{I} \subseteq \{1, \ldots, m\}$), such that $\bigcup_{i \in \mathscr{I}} S_i = U$, and to minimize the combined weight $\sum_{i \in \mathscr{I}} w_i$. We will analyze the following natural *greedy algorithm* that chooses sets according to a "minimum weight per new element covered" criterion. (The variable T in the pseudocode below keeps track of the set of elements that are not yet covered by $\bigcup_{i \in \mathscr{I}} S_i$.)

Algorithm 2 Greedy algorithm for set cover

Initialize $\mathscr{I} = \emptyset, T = U$.
while $T \neq \emptyset$ **do**
  $i = \arg\min_k \left\{ \frac{w_k}{|T \cap S_k|} \;\middle|\; 1 \leq k \leq m,\ T \cap S_k \neq \emptyset \right\}$.
  $\mathscr{I} = \mathscr{I} \cup \{i\}$.
  $T = T \setminus S_i$.
end while
return $\mathscr{I}$

It is clear that the algorithm runs in polynomial time and outputs a valid set cover. To analyze the approximation ratio, we will use the linear programming relaxation of set cover and its dual.

$$\begin{array}{lll} \min & \sum_{i=1}^{m} w_i x_i & \\ \text{s.t.} & \sum_{i: j \in S_i} x_i \geq 1 & \forall j \in U \\ & x_i \geq 0 & \forall i = 1, \ldots, m \end{array} \tag{6}$$

$$\begin{array}{lll} \max & \sum_{j \in U} y_j & \\ \text{s.t.} & \sum_{j \in S_i} y_j \leq w_i & \forall i = 1, \ldots, m \\ & y_j \geq 0 & \forall j \in U \end{array} \tag{7}$$

It will be helpful to rewrite the greedy set cover algorithm by adding some extra lines that do not influence the choice of which sets to place in $\mathscr{I}$, but merely compute extra data relevant to the analysis. Specifically, in the course of choosing sets to include in $\mathscr{I}$, we also compute a vector $\mathbf{z}$ indexed by elements of U. This is not a feasible solution of the dual LP, but at the end of the algorithm we scale it down to obtain another vector $\mathbf{y}$ that is feasible for the dual LP. The scale factor α will constitute an upper bound on the algorithm's approximation ratio. This is called the method of *dual fitting*.

Algorithm 3 Greedy algorithm for set cover

Initialize $\mathscr{I} = \emptyset$, $T = U$, $z_j = 0\ \forall j \in U$.
while $T \neq \emptyset$ **do**
  $i = \arg\min_k \left\{ \frac{w_k}{|T \cap S_k|} \;\middle|\; 1 \leq k \leq m,\ T \cap S_k \neq \emptyset \right\}$.
  $\mathscr{I} = \mathscr{I} \cup \{i\}$.
  for all $j \in T \cap S_i$ **do**
    $z_j = \frac{w_i}{|T \cap S_i|}$.
  end for
  $T = T \setminus S_i$.
end while
$\alpha = 1 + \ln(\max_{1 \leq i \leq m} |S_i|)$.
$\mathbf{y} = \frac{1}{\alpha}\mathbf{z}$.
return $\mathscr{I}$

The following three loop invariants are easily shown to hold at the beginning and end of each **while** loop iteration, by induction on the number of iterations.

1. $\sum_{j \in U} z_j = \sum_{i \in \mathscr{I}} w_i$.
2. For all $j \in U$, if the algorithm ever assigns a nonzero value to z_j then that value never changes afterward.

Below, in Lemma 1, we will show that the vector $\mathbf{y}$ is feasible for the dual LP (7). From this, it follows that the approximation ratio is bounded above by $\alpha = 1 + \ln(\max_{1 \leq i \leq m} |S_i|)$. To see this, observe that

$$\sum_{i \in \mathscr{I}} w_i = \sum_{j \in U} z_j = \alpha \sum_{j \in U} y_j \leq \alpha \sum_{i \in \mathrm{OPT}} w_i,$$

where the last line follows from weak duality.

Lemma 1. *The vector* $\mathbf{y}$ *computed in Algorithm* (3) *is feasible for the dual linear program* (7).

Proof. Clearly $y_j \geq 0$ for all j, so the only thing we really need to show is that $\sum_{j \in S_i} y_j \leq w_i$ for every set S_i. Let $p = |S_i|$ and denote the elements of S_i by $s_0, s_1, \ldots, s_{p-1}$, where the numbering corresponds to the order in which nonzero values were assigned to the variables z_j by Algorithm 3. Thus, a nonzero value was assigned to z_{s_0} before z_{s_1}, and so on. We know that

$$z_{s_0} \leq \frac{w_i}{p} \tag{8}$$

because at the time the value z_{s_0} was assigned, all of the elements of S_i still belonged to T. In that iteration of the while loop, the cost-effectiveness of S_i was judged to be w_i/p, the algorithm chose a set with the same or better cost-effectiveness, and all of the values z_j assigned during that iteration of the while loop were set equal to the cost-effectiveness of that set. Similarly, we know that for all $q < p$,

$$z_{s_q} \leq \frac{w_i}{p - q} \tag{9}$$

because at the time the value z_{s_q} was assigned, all of the elements $s_q, s_{q+1}, \ldots, s_{p-1}$ still belonged to T. In that iteration of the while loop, the cost-effectiveness of S_i was judged to be $w_i/(p-q)$ or smaller, the algorithm chose a set with the same or better cost-effectiveness, and all of the

values z_j assigned during that iteration of the while loop were set equal to the cost-effectiveness of that set.

Summing the bounds (9) for $q = 0, \ldots, p-1$, we find that

$$\sum_{j \in S_i} z_j \leq w_i \cdot \left(\frac{1}{p} + \frac{1}{p-1} + \cdots + \frac{1}{2} + 1 \right) < w_i \cdot \left(1 + \int_1^p \frac{dt}{t} \right) = w_i \cdot (1 + \ln p) \,.$$

The lemma follows upon dividing both sides by α. □

3 Randomized Approximation Algorithms

Randomized techniques give rise to some of the simplest and most elegant approximation algorithms. This section gives several examples.

3.1 A Randomized 2-Approximation for Max-Cut

In the max-cut problem, one is given an undirected graph $G = (V, E)$ and a positive weight w_e for each edge, and one must output a partition of V into two subsets A, B so as to maximize the combined weight of the edges having one endpoint in A and the other in B.

We will analyze the following extremely simple randomized algorithm: assign each vertex at random to A to B with equal probability, such that the random decisions for the different vertices are mutually independent. Let $E(A, B)$ denote the (random) set of edges with one endpoint in A and the other endpoint in B. The expected weight of our cut is

$$\mathbb{E}\left(\sum_{e \in E(A,B)} w_e \right) = \sum_{e \in E} w_e \cdot \Pr(e \in E(A, B)) = \frac{1}{2} \sum_{e \in E} w_e.$$

Since the combined weight of all edges in the graph is an obvious upper bound on the weight of any cut, this shows that the expected weight of the cut produced by our algorithm is at least half the weight of the maximum cut.

3.1.1 Derandomization using pairwise independent hashing

In analyzing the expected weight of the cut defined by our randomized algorithm, we never really used the full power of our assumption that the random decisions for the different vertices are mutually independent. The only property we needed was that for each pair of vertices u, v, the probability that u and v make different decisions is exactly $\frac{1}{2}$. It turns out that one can achieve this property using only $k = \lceil \log_2(n) \rceil$ independent random coin tosses, rather than n independent random coin tosses.

Let $\mathbb{F}_2$ denote the field $\{0, 1\}$ under the operations of addition and multiplication modulo 2. Assign to each vertex v a distinct vector $\mathbf{x}(v)$ in the vector space $\mathbb{F}_2^k$; our choice of $k = \lceil \log_2(n) \rceil$ ensures that the vector space contains enough elements to assign a distinct one to each vertex. Now let r be a uniformly random vector in $\mathbb{F}_2^k$, and partition the vertex set V into the subsets

$$A_{\mathbf{r}} = \{v \mid \mathbf{r} \cdot \mathbf{x}(v) = 0\}$$
$$B_{\mathbf{r}} = \{v \mid \mathbf{r} \cdot \mathbf{x}(v) = 1\}.$$

For any edge $e = (u, v)$, the probability that $e \in E(A_{\mathbf{r}}, B_{\mathbf{r}})$ is equal to the probability that $\mathbf{r} \cdot (\mathbf{x}(v) - \mathbf{x}(u))$ is nonzero. For any fixed nonzero vector $\mathbf{w} \in \mathbb{F}_2^k$, we have $\Pr(\mathbf{r} \cdot \mathbf{w} \neq 0) = \frac{1}{2}$ because the set of $\mathbf{r}$ satisfying $\mathbf{r} \cdot \mathbf{w} = 0$ is a linear subspace of $\mathbb{F}_2^k$ of dimension $k - 1$, hence exactly 2^{k-1} of the 2^k possible vectors r have zero dot product with $\mathbf{w}$ and the other 2^{k-1} of them have nonzero dot product with $\mathbf{w}$. Thus, if we sample $\mathbf{r} \in \mathbb{F}_2^k$ uniformly at random, the expected weight of the cut defined by $(A_{\mathbf{r}}, B_{\mathbf{r}})$ is at least half the weight of the maximum cut.

The vector space $\mathbb{F}_2^k$ has only $2^k = O(n)$ vectors in it, which suggests a deterministic alternative to our randomized algorithm. Instead of choosing $\mathbf{r}$ at random, we compute the weight of the cut $(A_{\mathbf{r}}, B_{\mathbf{r}})$ for every $r \in \mathbb{F}_2^k$ and take the one with maximum weight. This is at least as good as choosing r at random, so we get a deterministic 2-approximation algorithm at the cost of increasing the running time by a factor of $O(n)$.

3.1.2 Derandomization using conditional expectations

A different approach for converting randomization approximation algorithms into deterministic ones is the *method of conditional expectations*. In this technique, rather than making all of our random decisions simultaneously, we make them sequentially. Then, instead of making the decisions by choosing randomly between two alternatives, we evaluate both alternatives according to the conditional expectation of the objective function if we fix the decision (and all preceding ones) but make the remaining ones at random. Then we choose the alternative that optimizes this conditional expectation.

To apply this technique to the randomized max-cut algorithm, we imagine maintaining a partition of the vertex set into three sets A, B, C while the algorithm is running. Sets A, B are the two pieces of the partition we are constructing. Set C contains all the vertices that have not yet been assigned. Initially $C = V$ and $A = B = \emptyset$. When the algorithm terminates C will be empty. At an intermediate stage when we have constructed a partial partition (A, B) but C contains some unassigned vertices, we can imagine assigning each element of C randomly to A or B with equal probability, independently of the other elements of C. If we were to do this, the expected weight of the random cut produced by this procedure would be

$$w(A, B, C) = \sum_{e \in E(A,B)} w_e + \frac{1}{2} \sum_{e \in E(A,C)} w_e + \frac{1}{2} \sum_{e \in E(B,C)} w_e + \frac{1}{2} \sum_{e \in E(C,C)} w_e.$$

This suggests the following deterministic algorithm that considers vertices one by one, assigning them to either A or B using the function $w(A, B, C)$ to guide its decisions.

Algorithm 4 Derandomized max-cut algorithm using method of conditional expectations

```
Initialize A = B = ∅, C = V.
for all v ∈ V do
    Compute w(A + v, B, C − v) and w(A, B + v, C − v).
    if w(A + v, B, C − v) > w(A, B + v, C − v) then
        A = A + v
    else
        B = B + v
    end if
    C = C − v
end for
return A, B
```

The analysis of the algorithm is based on the simple observation that for every partition of V into three sets A, B, C and every $v \in C$, we have

$$\frac{1}{2}w(A+v, B, C-v) + \frac{1}{2}w(A, B+v, C-v) = w(A, B, C).$$

Consequently

$$\max\{w(A+v, B, C-v),\ w(A, B+v, C-v)\} \geq w(A, B, C)$$

so the value of $w(A, B, C)$ never decreases during the execution of the algorithm. Initially the value of $w(A, B, C)$ is equal to $\frac{1}{2}\sum_{e \in E} w_e$, whereas when the algorithm terminates the value of $w(A, B, C)$ is equal to $\sum_{e \in E(A,B)} w_e$. We have thus proven that the algorithm computes a partition (A, B) such that the weight of the cut is at least half the combined weight of all edges in the graph.

Before concluding our discussion of this algorithm, it's worth noting that the algorithm can be simplified by observing that

$$w(A+v, B, C-v) - w(A, B+v, C-v) = \frac{1}{2}\sum_{e \in E(B,v)} w_e - \frac{1}{2}\sum_{e \in E(A,v)} w_e.$$

The algorithm runs faster if we skip the step of actually computing $w(A+v, B, C-v)$ and jump straight to computing their difference. This also means that there's no need to explicitly keep track of the vertex set C.

Algorithm 5 Derandomized max-cut algorithm using method of conditional expectations

```
Initialize A = B = ∅.
for all v ∈ V do
    if Σ_{e∈E(B,v)} w_e − Σ_{e∈E(A,v)} w_e > 0 then
        A = A + v
    else
        B = B + v
    end if
end for
return A, B
```

This version of the algorithm runs in linear time: the amount of time spent on the loop iteration that processes vertex v is proportional to the length of the adjacency list of that vertex. It's also easy to prove that the algorithm has approximation factor 2 without resorting to any discussion of random variables and their conditional expectations. One simply observes that the property

$$\sum_{e \in E(A,B)} w_e \geq \sum_{e \in E(A,A)} w_e + \sum_{e \in E(B,B)} w_e$$

is a loop invariant of the algorithm. The fact that this property holds at termination implies that $\sum_{e \in E(A,B)} w_e \geq \frac{1}{2}\sum_{e \in E} w_e$ and hence the algorithm's approximation factor is 2.

3.1.3 Semidefinite programming and the Goemans-Williamson algorithm

So far, in our discussion of max-cut, we have made no mention of linear programming. It's worth considering for a moment the question of whether the natural linear programming relaxation of the max-cut problem can achieve an approximation factor better than 2. It's actually not so easy to write down the natural linear programming relaxation of max-cut. We can define decision variables $\{x_v \mid v \in V\}$, taking values in $[0,1]$, with the intended semantics that $x_v = 0$ if $v \in A$ and $x_v = 1$ if $v \in B$. The trouble is that the natural way to write the objective function is $\sum_{e \in E} |x_u - x_v|$, which is not a linear function because the absolute value function is non-linear. A workaround is to define a variable y_e for each edge e, with the intended semantics that $y_e = 1$ if e crosses the cut (A, B) and otherwise $y_e = 0$. This suggests the following linear programming relaxation of max-cut.

$$\begin{array}{lll} \max & \sum_{e \in E} y_e & \\ \text{s.t.} & y_e \leq x_u + x_v & \forall e = (u,v) \in E \\ & y_e \leq (1 - x_u) + (1 - x_v) & \forall e = (u,v) \in E \\ & 0 \leq x_v \leq 1 & \forall v \in V \end{array} \tag{10}$$

As noted earlier, for every partition of V into two sets A, B we can set $x_v = 0$ if $v \in A$, $x_v = 1$ if $v \in B$, and $y_e = 1$ if and only if e crosses the cut (A, B); this yields a valid integer solution of the linear program (10) such that the LP objective value equals the number of edges crossing the cut.

Unfortunately, for every graph the linear program (10) also has a fractional solution whose objective value equals m, the number of edges in the graph. Namely, setting $x_v = \frac{1}{2}$ for all v and $y_e = 1$ for all e satisfies the constraints of the linear program and achieves the objective value m. Since there exists graphs whose maximum cut contains barely more than half the edges (e.g., a complete graph on n vertices), solving this LP relaxation of max-cut only yields a 2-approximation, the same approximation factor achieved by the much simpler algorithms presented above.

For many years, it was not known whether *any* polynomial-time approximation algorithm for max-cut could achieve an approximation factor better than 2. Then in 1994, Michel Goemans and David Williamson discovered an algorithm with approximation factor roughly 1.14, based on *semidefinite programming* (SDP). Since then, SDP has found an increasing number of applications algorithm design, not only in approximation algorithms (where SDP has many other applications besides max-cut), but also in machine learning and high-dimensional statistics, coding theory, and other areas.

So what is semidefinite programming? An SDP is an optimization problem that seeks the maximum of a linear function over the set of symmetric positive semidefinite $n \times n$ matrices, subject to linear inequality constraints. See Lemma 2 below for the definition and some basic properties of symmetric positive semidefinite matrices. For now, we limit ourselves to the following remarks motivating why semidefinite programming is a powerful tool.

1. Semidefinite programming is solvable (to any desired precision) in polynomial time. The solution may contain irrational numbers, even if the input is made up exclusively of rational numbers, which is why the parenthetical remark about "to any desired precision" was necessary.

2. One of the main themes of algorithm design is, "When you have a discrete optimization problem that you don't know how to solve, formulate a related continuous optimization problem that you *do* know how to solve, and then try to figure out how to transform a solution of the continuous problem into a solution (either exact or approximate) or the original discrete problem. An obvious consequence of this theme is: *any time someone discovers a continuous optimization problem that can be solved by an efficient algorithm, that's a potential opportunity to design better algorithms for discrete optimization problems too.* This remark alone could justify the importance of SDP's in algorithm design.

3. Any linear program can be reformulated as a semidefinite program, by optimizing over the set of *diagonal* positive semidefinite matrices. Thus, SDP is at least as powerful as LP.

4. Often, one thinks of LP as relaxing a discrete optimization problem by allowing $\{0,1\}$-valued quantities to take continuous (scalar) values. In the same spirit, SDP can be thought of as further relaxing the problem by allowing scalar quantities to take (potentially high-dimensional) vector values.

To define semidefinite programming, we start with a lemma about real symmetric matrices. Any matrix satisfying the equivalent conditions listed in the lemma is called a *symmetric positive semidefinite (PSD)* matrix. The notation $A \succeq 0$ denotes the fact that A is PSD.

Lemma 2. *For a real symmetric matrix A, the following properties are equivalent.*

1. *Every eigenvalue of A is non-negative.*

2. *A can be expressed as a weighted sum of the form*

$$A = \sum_{i=1}^{m} c_i y_i y_i^{\mathsf{T}} \tag{11}$$

   where the coefficients c_i are non-negative.

3. *$A = XX^{\mathsf{T}}$ for some matrix X.*

4. *There exists a vector space containing vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$ such that A is the matrix of dot products of these vectors, i.e. for $1 \leq i, j \leq n$, the dot product $\mathbf{x}_i \cdot \mathbf{x}_j$ occurs in the i^{th} row and j^{th} column of A.*

5. *For every vector z, A satisfies the inequality $z^{\mathsf{T}} A z \geq 0$.*

Proof. To prove that the first property implies the second, we use the fact that every real symmetric matrix is orthogonally diagonalizable, i.e. that A can be written in the form $A = QDQ^{\mathsf{T}}$ where Q is an orthogonal matrix and D is a diagonal matrix whose entries are the eigenvalues of A. Defining c_i to be the i^{th} diagonal entry of D, and y_i to be the i^{th} column of Q, the equation $A = QDQ^{\mathsf{T}}$ expands as $A = \sum_{i=1}^{n} c_i y_i y_i^{\mathsf{T}}$, as desired.

It is elementary to prove that (2) $\rightarrow$ (3) $\rightarrow$ (5) $\rightarrow$ (1), as follows. If A is represented by a weighted sum as in (11), then $A = XX^{\mathsf{T}}$ where X is a matrix with k columns whose i^{th} column is $\sqrt{c_i} \cdot y_i$. If $A = XX^{\mathsf{T}}$, then for every z we have $z^{\mathsf{T}} A z = (z^{\mathsf{T}} X)(X^{\mathsf{T}} z) = \|X^{\mathsf{T}} z\|^2 \geq 0$. If the inequality $z^{\mathsf{T}} A z \geq 0$ is satisfied for every vector z, then in particular it is satisfied whenever z is an eigenvector of A with eigenvalue λ. This implies that $0 \leq z^{\mathsf{T}} A z = z^{\mathsf{T}}(\lambda z) = \lambda \|z\|^2$ and hence that $\lambda \geq 0$. Having proven that (1) $\rightarrow$ (2) in the preceding paragraph, we may now conclude that

all properties except possibly the fourth are equivalent. Finally, observe that the fourth property is simply a restatement of the third, adopting the notation $\mathbf{x}_i$ to denote the vector representing the i^{th} row of X. Thus, (3) and (4) are trivially equivalent. □

To begin relating the max-cut problem to a semidefinite program, it begins to reformulate max-cut as a problem about labeling vertices of a graph with $\{\pm 1\}$ rather than $\{0, 1\}$. Encoding cuts in that way, we can write max-cut as the following *quadratic* optimization problem.

$$\begin{array}{lll} \max & \frac{1}{2}\sum_{e=(u,v)\in E}(1 - x_u x_v) & \\ \text{s.t.} & x_v^2 = 1 & \forall v \in V \end{array} \tag{12}$$

The next step in transforming this into a semidefinite program is to treat the variables x_v as vectors rather than scalars. We'll change the notation to $\mathbf{x}_v$ to reflect this shift.

$$\begin{array}{lll} \max & \frac{1}{2}\sum_{e=(u,v)\in E}(1 - \mathbf{x}_u \cdot \mathbf{x}_v) & \\ \text{s.t.} & \|\mathbf{x}_v\|^2 = 1 & \forall v \in V \end{array} \tag{13}$$

For every $\{\pm 1\}$ solution of (12) there is a corresponding solution of (13) in which the vectors $\{\mathbf{x}_v \mid v \in V\}$ are all equal to $\pm\mathbf{w}$ for some fixed unit vector $\mathbf{w}$. On the other hand, there are also solutions of (13) that do not correspond to any $\{\pm 1\}$ solution of (12), so (13) is a relaxation of (12) and an optimal solution of (13) might be able to achieve a strictly higher objective value than the size of the maximum cut in the graph G.

Using Lemma 2 we know that (13) is equivalent to solving the following semidefinite program to optimize over PSD matrices $A = (a_{uv})$ whose entries are indexed by ordered pairs of vertices of G.

$$\begin{array}{lll} \max & \frac{1}{2}\sum_{e=(u,v)\in E} 1 - a_{uv} & \\ \text{s.t.} & a_{vv} = 1 & \forall v \in V \\ & A \succeq 0 & \end{array} \tag{14}$$

Furthermore, given the matrix A that solves (14), we can obtain, in polynomial time, a matrix X such that $A = XX^{\mathsf{T}}$. (This is tantamount to computing the eigenvalues and eigenvectors of A. It would be more accurate to say that we can compute X to arbitrary precision in polynomial time.) The rows of this X constitute the vectors $\mathbf{x}_v$ representing a solution of (13).

The question now arises: given the vectors $\{\mathbf{x}_v\}$, how do we output a cut in G such that the number of edges in the cut is approximately as large as $\frac{1}{2}\sum(1 - \mathbf{x}_u \cdot \mathbf{x}_v)$? The idea that works in this algorithms, and many other applications of semidefinite programming, is to choose a random hyperplane through the origin, partitioning $\mathbb{R}^n$ into two halfspaces. Then we partition the vertices of the graph according to which halfspace their corresponding vector belongs to. If (u, v) is an edge of G with corresponding vectors $\mathbf{x}_u, \mathbf{x}_v$, and if θ denotes the angle between those two vectors, then the probability that the random halfspace separates $\mathbf{x}_u$ from $\mathbf{x}_v$ is θ/π, whereas the contribution of edge (u, v) to the objective function of the SDP is $\frac{1}{2}(1-\mathbf{x}_u\cdot\mathbf{x}_v) = \frac{1}{2}(1-\cos\theta)$. It is an elementary exercise in calculus to prove that

$$\forall \theta \in [0, \pi]\ \frac{\theta}{\pi} \geq (0.878) \cdot \left[\tfrac{1}{2}(1 - \cos\theta)\right].$$

Therefore, the approximation ratio of the algorithm is at most $1/(0.878) \approx 1.14$.

3.2 A Randomized 2-Approximation for Vertex Cover

For the unweighted vertex cover problem (the special case of weighted vertex cover in which $w_v = 1$ for all v) the following incredibly simple algorithm is a randomized 2-approximation.

Algorithm 6 Randomized approximation algorithm for unweighted vertex cover

```
Initialize S = ∅.
for all e = (u, v) ∈ E do
    if neither u nor v belongs to S then
        Randomly choose u or v with equal probability.
        Add the chosen vertex into S.
    end if
end for
return S
```

Clearly, the algorithm runs in linear time and always outputs a vertex cover. To analyze its approximation ratio, as usual, we define an appropriate loop invariant. Let OPT denote any vertex cover of minimum cardinality. Let S_i denote the contents of the set S after completing the i^{th} iteration of the loop. We claim that for all i,

$$\mathbb{E}[|S_i \cap \text{OPT}|] \geq \mathbb{E}[|S_i \setminus \text{OPT}|]. \tag{15}$$

The proof is by induction on i. In a loop iteration in which $e = (u, v)$ is already covered by S_{i-1}, we have $S_i = S_{i-1}$ so (15) clearly holds. In a loop iteration in which $e = (u, v)$ is not yet covered, we know that at least one of u, v belongs to OPT. Thus, the left side of (15) has probability at least $1/2$ of increasing by 1, while the right side of (15) has probability at most $1/2$ of increasing by 1. This completes the proof of the induction step.

Consequently, letting S denote the random vertex cover generated by the algorithm, we have $\mathbb{E}[|S \cap \text{OPT}|] \geq \mathbb{E}[|S \setminus \text{OPT}|]$ from which it easily follows that $\mathbb{E}[|S|] \leq 2 \cdot |\text{OPT}|$.

The same algorithm design and analysis technique can be applied to weighted vertex cover. In that case, we choose a random endpoint of an uncovered edge (u, v) with probability inversely proportional to the weight of that endpoint.

Algorithm 7 Randomized approximation algorithm for weighted vertex cover

```
Initialize S = ∅.
for all e = (u, v) ∈ E do
    if neither u nor v belongs to S then
        Randomly choose u with probability w_v/(w_u+w_v) and v with probability w_u/(w_u+w_v).
        Add the chosen vertex into S.
    end if
end for
return S
```

The loop invariant is

$$\mathbb{E}\left[\sum_{v \in S_i \cap \text{OPT}} w_v\right] \geq \mathbb{E}\left[\sum_{v \in S_i \setminus \text{OPT}} w_v\right].$$

In a loop iteration when (u, v) is uncovered, the expected increase in the left side is at least $\frac{w_u w_v}{w_u + w_v}$ whereas the expected increase in the right side is at most $\frac{w_u w_v}{w_u + w_v}$.

4 Linear Programming with Randomized Rounding

Linear programming and randomization turn out to be a very powerful when used in combination. We will illustrate this by presenting an algorithm of Raghavan and Thompson for a problem of routing paths in a network to minimize congestion. The analysis of the algorithm depends on the *Chernoff bound*, a fact from probability theory that is one of the most useful tools for analyzing randomized algorithms.

4.1 The Chernoff bound

The Chernoff bound is a very useful theorem concerning the sum of a large number of independent random variables. Roughly speaking, it asserts that for any fixed $\beta > 1$, the probability of the sum exceeding its expected value by a factor greater than β tends to zero exponentially fast as the expected sum tends to infinity.

Theorem 3. *Let $X_1, \ldots, X_n$ be independent random variables taking values in $[0, 1]$, let X denote their sum, and let $\mu = \mathbb{E}[X]$. For every $\beta > 1$,*

$$\Pr(X \geq \beta\mu) < e^{-\mu[\beta \ln(\beta) - (\beta - 1)]}. \tag{16}$$

For every $\beta < 1$,

$$\Pr(X \leq \beta\mu) < e^{-\mu[\beta \ln(\beta) - (\beta - 1)]}. \tag{17}$$

Proof. The key idea in the proof is to make use of the moment-generating function of X, defined to be the following function of a real-valued parameter t:

$$M_X(t) = \mathbb{E}\left[e^{tX}\right].$$

From the independence of $X_1, \ldots, X_n$, we derive:

$$M_X(t) = \mathbb{E}\left[e^{tX_1} e^{tX_2} \cdots e^{tX_n}\right] = \prod_{i=1}^{n} \mathbb{E}\left[e^{tX_i}\right]. \tag{18}$$

To bound each term of the product, we reason as follows. Let Y_i be a $\{0, 1\}$-valued random variable whose distribution, conditional on the value of X_i, satisfies $\Pr(Y_i = 1 \mid X_i) = X_i$. Then for each $x \in [0, 1]$ we have

$$\mathbb{E}\left[e^{tY_i} \middle| X_i = x\right] = xe^t + (1 - x)e^0 \geq e^{tx} = \mathbb{E}\left[e^{tX_i} \middle| X_i = x\right],$$

where the inequality in the middle of the line uses the fact that e^{tx} is a convex function. Since this inequality holds for every value of x, we can integrate over x to remove the conditioning, obtaining

$$\mathbb{E}\left[e^{tY_i}\right] \geq \mathbb{E}\left[e^{tX_i}\right].$$

Letting μ_i denote $\mathbb{E}[X_i] = \Pr(Y_i = 1)$ we find that

$$\left[e^{tX_i}\right] \leq \left[e^{tY_i}\right] = \mu_i e^t + (1 - \mu_i) = 1 + \mu_i(e^t - 1) \leq \exp\left(\mu_i(e^t - 1)\right),$$

where $\exp(x)$ denotes e^x, and the last inequality holds because $1 + x \leq \exp(x)$ for all x. Now substituting this upper bound back into (18) we find that

$$\mathbb{E}\left[e^{tX}\right] \leq \prod_{i=1}^{n} \exp\left(\mu_i(e^t - 1)\right) = \exp\left(\mu(e^t - 1)\right).$$

The proof now splits into two parts depending on whether $\beta > 1$ or $\beta < 1$. In both cases we will be choosing $t = \ln \beta$ for a reason to be disclosed later. If $\beta > 1$ then $t = \ln \beta > 0$, hence $e^{tX} \geq e^{t\beta\mu}$ whenever $X \geq \beta\mu$. Since $e^{tX} > 0$ regardless, we have $\mathbb{E}\left[e^{tX}\right] \geq e^{t\beta\mu} \Pr(X \geq \beta\mu)$ and

$$\Pr(X \geq \beta\mu) \leq \exp\left(\mu(e^t - 1 - \beta t)\right). \tag{19}$$

If $\beta < 1$ then $t = \ln \beta < 0$, hence $e^{tX} \geq e^{t\beta\mu}$ whenever $X \leq \beta\mu$. Since $e^{tX} > 0$ regardless, we have $\mathbb{E}\left[e^{tX}\right] \geq e^{t\beta\mu} \Pr(X \leq \beta\mu)$ and

$$\Pr(X \leq \beta\mu) \leq \exp\left(\mu(e^t - 1 - \beta t)\right). \tag{20}$$

In both cases, our choice of t is designed to minimize the right-hand side of (19) or (20); elementary calculus reveals that the global minimum is attained when $t = \ln \beta$. Substituting this value of t into (19) and (20) completes the proof of the theorem. □

Corollary 4. *Suppose $X_1, \ldots, X_k$ are independent random variables taking values in $[0, 1]$, such that $\mathbb{E}[X_1 + \cdots + X_k] \leq 1$. Then for any $N > 2$ and any $b \geq \frac{3 \log N}{\log \log N}$, where* log *denotes the base-2 logarithm, we have*

$$\Pr(X_1 + \cdots + X_k \geq b) < \frac{1}{N}. \tag{21}$$

Proof. Let $\mu = \mathbb{E}[X_1 + \cdots + X_k]$ and $\beta = b/\mu$. Applying Theorem 3 we find that

$$\begin{aligned} \Pr(X_1 + \cdots + X_k \geq b) &\leq \exp\left(-\mu\beta \ln(\beta) + \mu\beta - \mu\right) \\ &= \exp\left(-b(\ln(\beta) - 1) - \mu\right) \leq e^{-b(\ln(\beta/e))}. \end{aligned} \tag{22}$$

Now, $\beta = b/\mu \geq b$, so

$$\frac{\beta}{e} \geq \frac{b}{e} \geq \frac{3 \log N}{e \log \log N}$$

and

$$\begin{aligned} b \ln\left(\frac{\beta}{e}\right) &\geq \left(\frac{3 \ln N}{\ln(\log N)}\right) \cdot \ln\left(\frac{3 \log N}{e \log \log N}\right) \\ &= 3 \ln(N) \cdot \left(1 - \frac{\ln(\log \log N) - \ln(3) + 1}{\ln(\log N)}\right) > \ln(N), \end{aligned} \tag{23}$$

where the last inequality holds because one can verify that $\ln(\log x) - \ln(3) + 1 < \frac{2}{3} \ln(x)$ for all $x > 1$ using basic calculus. Now, exponentiating both sides of (23) and combining with (22) we obtain the bound $\Pr(X_1 + \cdots + X_k \geq b) < 1/N$, as claimed. □

4.2 An approximation algorithm for congestion minimization

We will design an approximation algorithm for the following optimization problem. The input consists of a directed graph $G = (V, E)$ with positive integer edge capacities c_e, and a set of source-sink pairs (s_i, t_i), $i = 1, \ldots, k$, where each (s_i, t_i) is a pair of vertices such that G contains at least one path from s_i to t_i. The algorithm must output a list of paths $P_1, \ldots, P_k$ such that P_i is a path from s_i to t_i. The load on edge e, denoted by ℓ_e, is defined to be the number of paths P_i that traverse edge e. The congestion of edge e is the ratio ℓ_e/c_e, and the algorithm's objective is to minimize congestion, i.e. minimize the value of $\max_{e \in E} (\ell_e/c_e)$. This problem turns out to be NP-hard, although we will not prove that fact here.

The first step in designing our approximation algorithm is to come up with a linear programming relaxation. To do so, we define a decision variable $x_{i,e}$ for each $i = 1, \ldots, k$ and each $e \in E$, denoting whether or not e belongs to P_i, and we allow this variable to take fractional values. The resulting linear program can be written as follows, using $\delta^+(v)$ to denote the set of edges leaving v and $\delta^-(v)$ to denote the set of edges entering v.

$$\begin{array}{lll} \min & r & \\ \text{s.t.} & \sum_{e \in \delta^+(v)} x_{i,e} - \sum_{e \in \delta^-(v)} x_{i,e} = \begin{cases} 1 & \text{if } v = s_i \\ -1 & \text{if } v = t_i \\ 0 & \text{if } v \neq s_i, t_i \end{cases} & \forall i = 1, \ldots, k,\ v \in V \\ & \sum_{i=1}^{k} x_{i,e} \leq c_e \cdot r & \forall e \in E \\ & x_{i,e} \geq 0 & \forall i = 1, \ldots, k,\ e \in E \end{array} \tag{24}$$

When $(x_{i,e})$ is a $\{0,1\}$-valued vector obtained from a collection of paths $P_1, \ldots, P_k$ by setting $x_{i,e} = 1$ for all $e \in P_i$, the first constraint ensures that P_i is a path from s_i to t_i while the second one ensures that the congestion of each edge is bounded above by r.

Our approximation algorithm solves the linear program (24), does some postprocessing of the solution to obtain a probability distribution over paths for each terminal pair (s_i, t_i), and then outputs an independent random sample from each of these distributions. To describe the postprocessing step, it helps to observe that the first LP constraint says that for every $i \in \{1, \ldots, k\}$, the values $x_{i,e}$ define a network flow of value 1 from s_i to t_i. Define a flow to be *acyclic* if there is no directed cycle C with a positive amount of flow on each edge of C. The first step of the postprocessing is to make the flow $(x_{i,e})$ acyclic, for each i. If there is an index $i \in \{1, \ldots, k\}$ and a directed cycle C such that $x_{i,e} > 0$ for every edge $e \in C$, then we can let $\delta = \min\{x_{i,e} \mid e \in C\}$ and we can modify $x_{i,e}$ to $x_{i,e} - \delta$ for every $e \in C$. This modified solution still satisfies all of the LP constraints, and has strictly fewer variables $x_{i,e}$ taking nonzero values. After finitely many such modifications, we must arrive at a solution in which each of the flow $(x_{i,e})$, $1 \leq i \leq k$ is acyclic. Since this modified solution is also an optimal solution of the linear program, we may assume without loss of generality that in our original solution $(x_{i,e})$ the flow was acyclic for each i.

Next, for each $i \in \{1, \ldots, k\}$ we take the acyclic flow $(x_{i,e})$ and represent it as a probability distribution over paths from s_i to t_i, i.e. a set of ordered pairs (P, π_P) such that P is a path from s_i to t_i, π_P is a positive number interpreted as the probability of sampling P, and the sum of the probabilities π_P over all paths P is equal to 1. The distribution can be constructed using the following algorithm.

Algorithm 8 Postprocessing algorithm to construct path distribution

Given: Source s_i, sink t_i, acyclic flow $x_{i,e}$ of value 1 from s_i to t_i.
Initialize $\mathcal{D}_i = \emptyset$.
while there is a path P from s_i to t_i such that $x_{i,e} > 0$ for all $e \in P$ **do**
  $\pi_P = \min\{x_{i,e} \mid e \in P\}$
  $\mathcal{D}_i = \mathcal{D}_i \cup \{(P, \pi_P)\}$.
  for all $e \in P$ **do**
    $x_{i,e} = x_{i,e} - \pi_P$
  end for
end while
return $\mathcal{D}_i$

Each iteration of the **while** loop strictly reduces the number of edges with $x_{i,e} > 0$, hence the algorithm must terminate after selecting at most m paths. When it terminates, the flow $(x_{i,e})$ has value zero (as otherwise there would be a path from s_i to t_i with positive flow on each edge) and it is acyclic because $(x_{i,e})$ was initially acyclic and we never put a nonzero amount of flow on an edge whose flow was initially zero. The only acyclic flow of value zero is the zero flow, so when the algorithm terminates we must have $x_{i,e} = 0$ for all e.

Each time we selected a path P, we decreased the value of the flow by exactly π_P. The value was initially 1 and finally 0, so the sum of π_P over all paths P is exactly 1 as required. For any given edge e, the value $x_{i,e}$ decreased by exactly π_P each time we selected a path P containing e, hence the combined probability of all paths containing e is exactly $x_{i,e}$.

Performing the postprocessing algorithm 8 for each i, we obtain probability distributions $\mathcal{D}_1, \ldots, \mathcal{D}_k$ over paths from s_i to t_i, with the property that the probability of a random sample from $\mathcal{D}_i$ traversing edge e is equal to $x_{i,e}$. Now we draw one independent random sample from each of these k distributions and output the resulting k-tuple of paths, $P_1, \ldots, P_k$. We claim that with probability at least $1/2$, the parameter $\max_{e \in E} \{\ell_e / c_e\}$ is at most αr, where $\alpha = \frac{3 \log(2m)}{\log \log(2m)}$. This follows by a direct application of Corollary 4 of the Chernoff bound. For any given edge e, we can define independent random variables $X_1, \ldots, X_k$ by specifying that

$$X_i = \begin{cases} (c_e \cdot r)^{-1} & \text{if } e \in P_i \\ 0 & \text{otherwise.} \end{cases}$$

These are independent and the expectation of their sum is $\sum_{i=1}^{k} x_{i,e}/(c_e \cdot r)$, which is at most 1 because of the second LP constraint above. Applying Corollary 4 with $N = 2m$, we find that the probability of $X_1 + \cdots + X_k$ exceeding α is at most $1/(2m)$. Since $X_1 + \cdots X_k = \ell_e (c_e \cdot r)^{-1}$, this means that the probability of ℓ_e / c_e exceeding αr is at most $1/(2m)$. Summing the probabilities of these failure events for each of the m edges of the graph, we find that with probability at least $1/2$, none of the failure events occur and $\max_{e \in E} \{\ell_e / c_e\}$ is bounded above by αr. Now, r is a lower bound on the parameter $\max_{e \in E} \{\ell_e / c_e\}$ for *any* k-tuple of paths with the specified source-sink pairs, since any such k-tuple defines a valid LP solution and r is the optimum value of the LP. Consequently, our randomized algorithm achieves approximation factor α with probability at least $1/2$.

The multiplicative weights update method is a family of algorithms that have found many different applications in CS: algorithms for learning and prediction problems, fast algorithms for approximately solving certain linear programs, and hardness amplification in complexity theory, to name a few examples. It is a general and surprisingly powerful iterative method based on maintaining a vector of state variables and applying small multiplicative updates to the components of the vector to converge toward an optimal solution of some problem. These notes introduce the basic method and explore two applications: online prediction problems and packing/covering linear programs.

1 Investing and combining expert advice

In this section we analyze two inter-related problems. The first problem is an *investment problem* in which there are n stocks numbered $1, \ldots, n$, and an investor with an initial wealth $W(0) = 1$ must choose in each period how to split the current wealth among the securities. The price of each stock then increases by some factor between 1 and $1 + \varepsilon$ (a different factor for each stock, not known to the investor at the time of making her investment) and the wealth increases accordingly. The goal is to do nearly as well as buying and holding the single best-performing stock.

Let's introduce some notation for the investment problem. At time $t = 1, \ldots, T$, the investor chooses to partition her wealth into shares $x_1(t), \ldots, x_n(t)$. These shares must be non-negative (short-selling stocks is disallowed) and they must sum to 1 (the investor's money must be fully invested).

$$\sum_{i=1}^{n} x_i(t) = 1 \quad \forall t$$
$$x_i(t) \geq 0 \quad \forall t \; \forall i$$

We summarize these constraints by saying that the vector $\mathbf{x}(t) = (x_1(t), \ldots, x_n(t))$ belongs to the probability simplex $\mathbf{\Delta}(n)$. As stated earlier, the amount by which the price of stock i appreciates at time t is a number between 1 and $1 + \varepsilon$. Denote this number by $(1 + \varepsilon)^{r_i(t)}$.

If we let $W(t)$ denote the investor's wealth at the end of round t, then the wealth at the start of round t is $W(t-1)$ and the amount invested in stock i is $x_i(t)W(t-1)$. We thus have

$$W(t) = \sum_{i=1}^{n} (1 + \varepsilon)^{r_i(t)} x_i(t) W(t-1).$$

The prediction problem that we will study bears some superficial similarities to the investment problem. (And, as we will see, the similarity extends much deeper.) In this problem there is a gambler and n "experts". At time $t = 1, \ldots, n$, the gambler bets \$1 by dividing it among the experts. Once again, we will use $\mathbf{x}(t) \in \mathbf{\Delta}(n)$ to denote the vector representing how the gambler splits her bet at time t. Each expert generates a payoff at time t denoted by $r_i(t)$, and the gambler's payoff is the dot product $\mathbf{x}(t) \cdot \mathbf{r}(t)$. In other words, placing a bet of $x_i(t)$ on expert i yields a payoff of $x_i(t)r_i(t)$ in round t, and the gambler's total payoff is the sum of these

payoffs. The goal is to gain nearly as much payoff as the strategy that always bets on the single best-performing expert.

There are some clear relationships between the two problems, but also some clear differences, chiefly that payoffs accumulate multiplicatively in one problem, and additively in the other. Consequently, the relationship between the problems becomes clearer when we take the logarithm of the investor's wealth. For example, if the investor follows the strategy of buying and holding stock i, her wealth after time t would satisfy

$$W(t) = \prod_{i=1}^{t} (1+\varepsilon)^{r_i(t)}$$
$$\log_{1+\varepsilon} W(t) = \sum_{i=1}^{t} r_i(t) = r_i(1:t)$$

where the last equation should be interpreted as the definition of the notation $r_i(1:t)$. Similarly, if the investor follows the "uniform buy-and-hold strategy" of initially investing $1/n$ in each stock, and never performing any trades after that, then her investment in stock i after time t is given by $\frac{1}{n}(1+\varepsilon)^{r_i(1:t)}$, and her log-wealth after time t satisfies

$$\log_{1+\varepsilon} W(t) = \log_{1+\varepsilon} \left(\frac{1}{n} \sum_{i=1}^{n} (1+\varepsilon)^{r_i(1:t)} \right).$$

Letting i denote an arbitrary stock (e.g. the best-performing one), the wealth of the uniform buy-and-hold strategy satisfies

$$\log_{1+\varepsilon} W(t) > \log_{1+\varepsilon} \left(\frac{1}{n} (1+\varepsilon)^{r_i(1:t)} \right) = r_i(1:t) - \log_{1+\varepsilon}(n).$$

This already gives a useful bound on the additive difference in log-wealth between the uniform buy-and-hold strategy and the strategy that buys and holds the single best-performing stock.

An important relationship between the investment and prediction problems is expressed by the following calculation, which applies to an investor who distributes her wealth at time t using vector $\mathbf{x}(t)$. The log-wealth after time t then satisfies the following.

$$\begin{aligned}
\log_{1+\varepsilon} W(t) &= \log_{1+\varepsilon} \left(\sum_{i=1}^{n} (1+\varepsilon)^{r_i(t)} x_i(t) W(t-1) \right) \\
&= \log_{1+\varepsilon} W(t-1) + \log_{1+\varepsilon} \left(\sum_{i=1}^{n} (1+\varepsilon)^{r_i(t)} x_i(t) \right) \\
&\leq \log_{1+\varepsilon} W(t-1) + \log_{1+\varepsilon} \left(\sum_{i=1}^{n} (1+\varepsilon r_i(t)) x_i(t) \right) \\
&= \log_{1+\varepsilon} W(t-1) + \frac{\ln\left(1+\varepsilon \sum_{i=1}^{n} r_i(t) x_i(t)\right)}{\ln(1+\varepsilon)} \\
&\leq \log_{1+\varepsilon} W(t-1) + \frac{\varepsilon}{\ln(1+\varepsilon)} \mathbf{x}(t) \cdot \mathbf{r}(t).
\end{aligned}$$

Summing over $t = 1, \ldots, T$, we find that

$$\log_{1+\varepsilon} W(T) \leq \frac{\varepsilon}{\ln(1+\varepsilon)} \sum_{t=1}^{T} \mathbf{x}(t) \cdot \mathbf{r}(t),$$

which implies a relation between the log-wealth of an investor using strategy $\mathbf{x}(1), \ldots, \mathbf{x}(T)$ and the payoff of a gambler using the same strategy sequence in the prediction problem.

Recall that the uniform buy-and-hold strategy was actually a pretty good strategy for the investor. This implies that the corresponding prediction strategy is pretty good for the gambler. In the gambling context (also known as the *predicting from expert advice* context) the strategy that corresponds to uniform-buy-and-hold is known as the **multiplicative weights algorithm** or **Hedge**. At time t it predicts the vector $\mathbf{x}(t)$ whose i^{th} component is given by

$$x_i(t) = \frac{(1+\varepsilon)^{r_i(1:t)}}{\sum_{j=1}^{n}(1+\varepsilon)^{r_j(1:t)}}.$$

We have seen that the payoff of the multiplicative weights algorithm satisfies

$$\begin{aligned}\sum_{t=1}^{T} \mathbf{x}(t)\cdot\mathbf{r}(t) &\geq \frac{\ln(1+\varepsilon)}{\varepsilon}\log_{1+\varepsilon} W(T) \\ &\geq \frac{\ln(1+\varepsilon)}{\varepsilon} r_i(1:t) - \frac{\ln(1+\varepsilon)}{\varepsilon}\log_{1+\varepsilon} n \\ &> (1-\varepsilon) r_i(1:t) - \frac{\ln n}{\varepsilon}.\end{aligned}$$

The last line used the identity $\frac{1}{x}\ln(1+x) > 1-x$ which is valid for any $x > 0$. (See the proof in Appendix **??**.)

The role of the parameter $\varepsilon > 0$ in the two problems deserves some discussion. In the investment problem, ε is a parameter of the model, and one can either treat it as an assumption about the way stock prices change in discrete time — never by a factor of more than $1+\varepsilon$ from one time period the next — or one can instead imagine that stock prices change continuously over time, and the parameter ε is determined by how rapidly the investor chooses to engage in trading. In the prediction problem, on the other hand, the model does not define ε and it is instead under the discretion of the algorithm designer. There is a tradeoff between choosing a small or a large value of ε, and the performance guarantee

$$\sum_{t=1}^{T} \mathbf{x}(t)\cdot\mathbf{r}(t) > (1-\varepsilon) r_i(1:t) - \frac{\ln n}{\varepsilon}$$

neatly summarizes the tradeoff. A smaller value of ε allows the gambler to achieve a better multiplicative approximation to the best expert, at the cost of a larger additive error term. In short, ε can be interpreted as a "learning rate" parameter: with a small ε (slow learning rate) the gambler pays a huge start-up cost in order to eventually achieve a very close multiplicative approximation to the optimum; with a large ε the eventual approximation is more crude, but the start-up cost is much cheaper.

2 Solving linear programs with multiplicative weights

This section presents an application of the multiplicative-weights method to solving packing and covering linear programs. When A is a non-negative matrix and p, b are non-negative vectors, the following pair of linear programs are called a *packing* and a *covering* linear program, respectively.

$$\begin{array}{lll} \max & p^\intercal y & \\ \text{s.t.} & Ay \preceq b & \\ & y \succeq 0 & \end{array} \qquad \begin{array}{lll} \min & b^\intercal x & \\ \text{s.t.} & A^\intercal x \succeq p & \\ & x \succeq 0 & \end{array}$$

Note that the covering problem is the dual of the packing problem and vice-versa. To develop intuitions about these linear programs it is useful to adopt the following metaphor. Think of the entries a_{ij} of matrix A as denoting the amount of raw material i needed to product one unit of product j. Think of b_i as the total supply of resource i available to a firm, and p_j as the unit price at which the firm can sell product j. If the vector y in the first LP is interpreted as the quantity of each product to be produced, then the vector Ay encodes the amount of each resource required to produce y, the constraint $Ay \preceq b$ says that the firm's production is limited by its resource budget, and the optimization criterion (maximize $p^\intercal y$) specifies that the firm's goal is to maximize revenue.

The dual LP also admits an interpretation within this metaphor. If we think of the vector x as designating a unit price for each raw material, then the constraint $A^\intercal x \succeq p$ expresses the property that for each product j, the cost of resources required to produce one unit of j exceeds the price at which it can be sold. Therefore, if a vector x is feasible for the dual LP, then the cost of obtaining the resource bundle b at prices x (namely, $b^\intercal x$) exceeds the revenue gained from selling any product bundle y that can be made from the resources in b (namely, $p^\intercal y$). This reflects weak duality, the assertion that the maximum of $p^\intercal y$ over primal-feasible vectors y is less than or equal to the minimum of $b^\intercal x$ over dual-feasible vectors x. Strong duality asserts that they are in fact equal; the algorithm we will develop supplies an algorithmic proof of this fact.

The multiplicative weights method for solving packing and covering linear programs was pioneered by Plotkin, Shmoys, and Tardos and independently by Grigoriadis and Khachiyan. The version we present here differs a bit from the Plotkin-Shmoys-Tardos exposition of the algorithm, in order to leverage the connection to the multiplicative weights method for online prediction, as well as to incorporate a "width reduction" technique introduced by Garg and Könemann. We will make the simplifying assuymption that $b = B \cdot \mathbf{1}$, for some scalar $B > 0$. We can always manipulate the linear program so that it satisfies this assumption, by simply changing the units in which resource consumption is measured. Also, after rescaling the units of resource consumption (by a common factor) we can assume that $0 \leq a_{ij} \leq 1$ for all i, j — possibly at the expense of changing the value of B.

The algorithm is as follows.

Algorithm 1 Multiplicative weights algorithm for packing/covering LP's

Given: parameters $\epsilon, \delta > 0$.
Initialize: $t \leftarrow 0, Y \leftarrow 0$. *// Y is a vector storing $\delta(y_1 + \cdots + y_t)$.*
while $AY \prec B\mathbf{1}$ **do**
  $t \leftarrow t + 1$.
  $\forall i = 1, \ldots, n \quad (x_t)_i \leftarrow \frac{(1+\varepsilon)^{(AY)_i/\delta}}{\sum_{j=1}^n (1+\varepsilon)^{(AY)_j/\delta}}$.
  $y_t \leftarrow \arg\min_{y \in \mathbf{\Delta}(n)} \left\{ \frac{x_t^\intercal A y}{p^\intercal y} \right\}$.
  $Y \leftarrow Y + \delta\, y_t$.
end while

The vector x_t is being set using the multiplicative weights algorithm with payoff sequence $r_t = Ay_t$. In the expression defining y_t, the ratio $\frac{x_t^\intercal Ay}{p^\intercal y}$ can be interpreted as the cost-benefit ratio of producing a product randomly sampled from the probability distribution y. The arg min of this ratio will therefore be a point-mass distribution concentrated on the single product with the smallest cost-benefit ratio, i.e. one can always choose the vector y_t to have only one non-zero entry.

To analyze the algorithm, we begin with the performance guarantee of the multiplicative weights prediction algorithm. Let T be the time when the algorithm terminates.

$$\sum_{t=1}^{T} x_t^\intercal Ay_t \geq (1-\varepsilon) \max_i \left\{ \sum_{t=1}^{T} (Ay_t)_i \right\} - \frac{\ln n}{\varepsilon} \geq (1-\varepsilon) \cdot \frac{B}{\delta} - \frac{\ln n}{\varepsilon}. \tag{1}$$

(The second inequality is justified by the stopping condition for the algorithm.)

Next we work on deriving an upper bound on the quantity on the left side of (1). The definition of y_t implies that for any other vector y,

$$\frac{x_t^\intercal Ay}{p^\intercal y} \geq \frac{x_t^\intercal Ay_t}{p^\intercal y_t}. \tag{2}$$

Setting y in this inequality equal to y_*, the optimum solution of the primal linear program, we find that

$$\frac{(x_t^\intercal Ay_*)(p^\intercal y_t)}{p^\intercal y_*} \geq x_t^\intercal Ay_t. \tag{3}$$

Let $\bar{x}$ denote the weighted average of the vectors $x_1, \ldots, x_T$, averaged with weights $\frac{p^\intercal y_t}{p^\intercal Y}$:

$$\bar{x} = \frac{\delta}{p^\intercal Y} \sum_{t=1}^{T} (p^\intercal y_t) x_t. \tag{4}$$

Summing (3) over $t = 1, \ldots, T$ and using the definition of $\bar{x}$, we obtain

$$\frac{1}{\delta} \cdot \frac{p^\intercal Y}{p^\intercal y_*} \cdot \bar{x}^\intercal Ay_* \geq \sum_{t=1}^{T} x_t^\intercal Ay_t. \tag{5}$$

Each of the vectors $x_1, \ldots, x_T$ satisfies $x_t^\intercal \mathbf{1} = 1$, so their weighted average $\bar{x}$ satisfies $\bar{x}^\intercal \mathbf{1} = 1$ as well. Using the inequality $Ay_* \preceq B\mathbf{1}$, which follows from primal feasibility of y_*, we now deduce that

$$B = \bar{x}^\intercal (B\mathbf{1}) \geq \bar{x}^\intercal Ay_*. \tag{6}$$

Combining (1), (5), (6) we obtain

$$\frac{p^\intercal Y}{p^\intercal y_*} \cdot \frac{B}{\delta} \geq (1-\varepsilon) \cdot \frac{B}{\delta} - \frac{\ln n}{\varepsilon} \tag{7}$$

$$\frac{p^\intercal Y}{p^\intercal y_*} \geq 1 - \varepsilon - \frac{\delta \ln n}{\varepsilon B}. \tag{8}$$

Thus, if we want to ensure that the algorithm computes a vector Y which is at least a $(1-2\varepsilon)$-approximation to the optimum of the linear program, it suffices to set $\delta = \frac{\varepsilon^2 B}{\ln n}$.

To bound the number of iterations of the algorithm's while loop, let γ be a parameter such that every column of A has an entry bounded below by γ. Then, in every iteration some entry of the vector AY increases by at least $\gamma\delta$. Since the algorithm stops as soon as some entry of AY exceeds B, the number of iterations is bounded above by $nB/(\gamma\delta)$. Substituting $\delta = \frac{\varepsilon^2 B}{\ln n}$, this means that the number of iterations is bounded by $(n \log n)/(\varepsilon^2 \gamma)$.

3 Multicommodity Flow

Now it's time to see how these ideas are applied in the context of a concrete optimization problem, multicommodity flow, which is a generalization of network flow featuring multiple source-sink pairs.

3.1 Problem definition

A multicommodity flow problem is specified by a graph (directed or undirected) G, a collection of k source-sink pairs $\{(s_i, t_i)\}_{i=1}^k$, and a non-negative capacity $c(e)$ for every edge $e = (u, v)$. A *multicommodity flow* is a k-tuple of flows $(f_1, \ldots, f_k)$ such that f_i is a flow from s_i to t_i, and the superposition of all k flows satisfies the edge capacity constraints in the sense that for every edge $e = (u, v)$ we have

$$\textit{[Undirected case]} \qquad c(e) \geq \sum_{i=1}^{k} |f_i(u, v)|$$

$$\textit{[Directed case]} \qquad c(e) \geq \sum_{i=1}^{k} \max\{0, f_i(u, v)\}$$

There are two different objectives that are commonly studied in multicommodity flow theory.

Maximum throughput: Maximize $\sum_{i=1}^{k} |f_i|$.

Maximum concurrent flow: Maximize $\min_{1 \leq i \leq k} |f_i|$.

3.2 The case of uniform edge capacities

It is fairly straightforward to apply the multiplicative-weights algorithm to solve multicommodity flow problems in graphs where all edges have identical capacity. (We will consider the general case, in which edges don't necessarily have identical capacity, in the next section of these notes.) Letting B denote the capacity of each edge, the multicommodity flow problem can be expressed by the following linear program with exponentially many variables y_P, where P ranges over all paths that join some source-sink pair (s_i, t_i).

$$\begin{array}{ll} \max & \sum_P y_P \\ \text{s.t.} & \sum_{P: e \in P} y_P \leq c(e) \quad \forall e \\ & y_P \geq 0 \quad \forall P \end{array} \tag{9}$$

This problem is a packing linear program. The objective function of the packing problem has coefficient vector $p = \mathbf{1}$, and the constraint matrix A has entries $a_{ij} = 1$ if edge e_i belongs to path P_j. Note that every column of A contains at least one entry equal to 1, so this problem has $\gamma = 1$. Thus, the multiplicative weights algorithm finds a $(1 - 2\varepsilon)$-approximation of the optimal solution in at most $m \log n / \varepsilon^2$ iterations, where m denotes the number of edges. (In previous sections we referred to the number of constraints in the packing LP as n rather than m, but it would be too confusing to use the letter n to denote the number of edges in a graph, which is always denoted by m. Accordingly, we have switched to using m in this section.)

In any iteration of the algorithm, we must solve the minimization problem $\arg\min\{(x_t^\intercal A y)/(\mathbf{1}^\intercal y) \mid y \in \mathbf{\Delta}(\text{paths})\}$, where $\mathbf{\Delta}(\text{paths})$ denotes the set of all probability distributions over paths that

join some (s_i, t_i) pair. Recalling that the minimum is always achieved at a distribution y that assigns probability 1 to one path and 0 to all others, and that the vector Ay in this case is a $\{0,1\}$-vector that identifies the edges of the path, we see that the expression $x_t^\intercal Ay$ can be interpreted as the combined cost of the edges in path y, when edge costs are given by the entries of the vector x_t. The expression $\mathbf{1}^\intercal y$ is simply equal to 1, so it can be ignored. Thus, the minimization problem that we must solve in one iteration of the algorithm is to find a minimum-cost path with respect to the edge costs given by x_t. This is easily done by running Dijkstra's algorithm to find the minimum cost (s_i, t_i) path for each $i = 1, \ldots, k$.

In summary, we have derived the following algorithm for approximately solving the maximum-throughput multicommodity flow problem in graphs whose edges all have identical capacity B. The algorithm reduces computing a $(1 - 2\varepsilon)$-approximate maximum multicommodity flow to solving $km \ln m/\varepsilon^2$ shortest-path problems. In the pseudocode, the variable z_e for an edge $e = e_i$ keeps track of the amount of flow we have sent on edge e, and $x_e = (1+\varepsilon)^{z_e/\delta}$ is a variable whose value in loop iteration t is proportional to (but not equal to) the i^{th} entry of the vector x_t in the above discussion. The algorithm's validity is unaffected by the fact that the vector $(x_e)_{e \in E}$ is a scalar multiple of the vector x_t in the above discussion, because the outcome of the min-cost path computation with respect to edge cost vector $\mathbf{x}$ is unaffected by rescaling the costs.

Algorithm 2 Max-throughput multicommodity flow algorithm, uniform-capacity case.

Given: Parameter $\varepsilon > 0$.
Initialize: $\delta = \varepsilon^2 B/(\ln m)$, $x = \mathbf{1}$, $f_1 = \cdots = f_k = 0$, $z = 0$.
while $z \prec B\mathbf{1}$ **do**
    for $i = 1, \ldots, k$ **do**
        $P_i \leftarrow$ minimum cost path from s_i to t_i, with respect to edge costs x_e.
    end for
    $i \leftarrow \arg\min_{1 \le j \le k}\{\text{cost}(P_j)\}$.
    Update flow f_i by sending δ units of flow on P_j.
    for all $e \in P_j$ **do**
        $x_e \leftarrow (1 + \varepsilon)x_e$.
        $z_e \leftarrow z_e + \delta$.
    end for
end while

Note that in this example, the fact that the packing linear program has exponentially many variables did not prevent us from designing an efficient algorithm to solve it. That is because, although the matrix A and vector Y in the multiplicative-weights algorithm have exponentially many entries, the algorithm never explicitly stores and manipulates them. This theme is quite common in applications of the multiplicative-weights method: the space requirement of the algorithm scales linearly with the number of *constraints* in the primal LP, but we can handle exponentially many variables in polynomial space and time, provided that we have a subroutine that efficiently solves the minimization problem $\arg\min\{(x_t^\intercal Ay)/(p^\intercal y)\}$.

3.3 General edge capacities

When edges have differing capacities, a small modification to the foregoing algorithm permits us to use it for computing an approximate maximum-throughput multicommodity flow.

The issue is that the multiplicative-weights algorithm we have presented in these notes requires a packing LP in which all of the constraints have the same number, B, appearing on their right-hand side. As a first step in dealing with this, we can rescale both sides of each constraint:

$$\sum_{P:e\in P} y_P \le c(e) \quad \Longleftrightarrow \sum_{P:e\in P} \frac{1}{c(e)} y_P \le 1.$$

The trouble with this rescaling is that now the constraint matrix entry a_{ij} is equal to $1/c(e_i)$ if edge e_i belongs to path P_j. Our algorithm requires $0 \le a_{ij} \le 1$ and this could be violated if some edges have capacity less than 1.

The simplest way to deal with this issue is to preprocess the graph, scaling all edge capacities by $1/c_{\min}$ where $c_{\min}$ denotes the minimum edge capacity, to obtain a graph whose edge capacities are bounded below by 1. Then we can solve for an approximate max-flow in the rescaled graph, and finally scale that flow down by $c_{\min}$ to obtain a flow that is feasible — and still approximately throughput-maximizing — in the original graph. To bound the number of iterations that this algorithm requires, we must determine the value of γ for the rescaled graph. The rescaled capacity of edge e is $c(e)/c_{\min}$, so the matrix entry a_{ij} is $c_{\min}/c(e_i)$ if edge e_i belongs to path P_j. Thus, the maximum entry in column j of the constraint matrix is $c_{\min}/c_{\min}(P_j)$, where $c_{\min}(P_j)$ denotes the minimum edge capacity in P_j. Thus $\gamma = \min_j\{c_{\min}/c_{\min}(P_j)\}$ and the number of iterations is

$$\frac{m \ln m}{\gamma \varepsilon^2} = \frac{m \ln m}{\varepsilon^2} \cdot \max_j \left\{ \frac{c_{\min}(P_j)}{c_{\min}} \right\}.$$

This could be a very large number of iterations, if the graph contains some very "fat" paths whose minimum-capacity edge has much more capacity than the globally minimum edge capacity.

Rather than rescaling all of the edge capacities in the graph by the same common factor, a smarter solution is to rescale the *flow* on path P_j by the factor $c_{\min}(P_j)$. More precisely, define the "P-saturating flow " to be the flow that sends $c_{\min}(P)$ units on every edge of P, and zero on all other edges. Our LP will have variables y_P for every path P that joins s_i to t_i for some $i = 1, \ldots, k$, and a primal-feasible solution will correspond to a multicommodity flow that is a weighted sum of P-saturating flows, scaled by the values y_P.

This leads to the following linear programming formulation of maximum-throughput multicommodity flow.

$$\begin{array}{ll} \max & \sum_P c_{\min}(P) y_P \\ \text{s.t.} & \sum_{P:e\in P} \frac{c_{\min}(P)}{c(e)} y_P \le 1 \quad \forall e \\ & y_P \ge 0 \quad \forall P \end{array} \tag{10}$$

The constraint matrix has entries $a_{ij} = \frac{c_{\min}(P_j)}{c(e_i)}$ if e_i belongs to P_j. By the definition of $c_{\min}(P_j)$, this implies that all entries are between 0 and 1, and that every column of A has at least one entry equal to 1. Thus the multiplicative weights method, applied to this LP formulation, yields a $(1-2\varepsilon)$-approximate solution after at most $\frac{m \ln m}{\varepsilon^2}$ iterations. To conclude the discussion of this algorithm, we should specify a procedure for solving the minimization problem $\arg\min\{(x_t^\intercal A y)/(p^\intercal y)\}$ in every iteration of the while loop. If y is the indicator vector for a path P, then $p^\intercal y = c_{\min}(P)$ while Ay is the vector whose i^{th} entry is $\frac{c_{\min}(P)}{c(e_i)}$ if e_i belongs to P, and 0

otherwise. Thus,

$$x_t^\intercal Ay = \sum_{e \in P} \frac{c_{\min}(P) x_{ti}}{c(e_i)}$$
$$\frac{x_t^\intercal Ay}{p^\intercal y} = \sum_{e \in P} \frac{x_{ti}}{c(e_i)}$$

so the minimization problem that must be solved in each loop iteration is merely finding a minimum-cost path with respect to the edge costs $\text{cost}(e_i) = \frac{x_{ti}}{c(e_i)}$.

Summarizing this discussion, we have the following algorithm which finds a $(1-2\varepsilon)$-approximate maximum-throughput multicommodity flow in general graphs using $\frac{m \ln m}{\varepsilon^2}$ loop iterations, each of which requires k minimum-cost path computations. In the pseudocode, the variable z_e for an edge $e = e_i$ keeps track of the *fraction* of e's capacity that has already been consumed by the flow sent in previous loop iterations. The variable $x_e = (1+\varepsilon)^{z_e/\delta}$ is a variable whose value in loop iteration t is proportional to (but not equal to) the i^{th} entry of the vector x_t in the above discussion. As in the preceding section, the algorithm's validity is unaffected by the fact that the vector $(x_e)_{e \in E}$ is a scalar multiple of the vector x_t in the above discussion, because the outcome of the min-cost path computation with respect to edge cost vector $\mathbf{x}$ is unaffected by rescaling the costs.

Algorithm 3 Max-throughput multicommodity flow algorithm, general case.

Given: Parameter $\varepsilon > 0$.
Initialize: $\delta = \varepsilon^2/(\ln m)$, $x = \mathbf{1}$, $f_1 = \cdots = f_k = 0$, $z = 0$.
while $z \prec \mathbf{1}$ **do**
    for $i = 1, \ldots, k$ **do**
        $P_i \leftarrow$ minimum cost path from s_i to t_i, with respect to edge costs $x_e/c(e)$.
    end for
    $i \leftarrow \arg\min_{1 \le j \le k} \{\text{cost}(P_j)\}$.
    Update flow f_i by sending $\delta c_{\min(P_j)}$ units of flow on P_j.
    for all $e \in P_j$ **do**
        $r \leftarrow \frac{c_{\min}(P_j)}{c(e)}$.
        $x_e \leftarrow (1+\varepsilon)^r x_e$.
        $z_e \leftarrow z_e + \delta\, r$.
    end for
end while

3.4 Maximum concurrent flow

The maximum concurrent flow problem can be solved using almost exactly the same technique. While the packing formulation of maximum-throughput multicommodity flow involves packing individual paths, each of which connects one source-sink pair, the natural packing formulation of maximum concurrent multicommodity flow involves packing k-tuples of paths, one for each source-sink pair. In the following LP, Q is an index that ranges over all such k-tuples. (As before, this means that there are exponentially many variables y_Q. Likewise, as before, this will not inhibit our ability to design an efficient algorithm for approximately solving the LP, because the algorithm need not explicitly represent all of the entries of the constraint matrix A or the vector Y.) The notation $n_Q(e)$ refers to the number of paths in the k-tuple Q that contain edge e;

thus, its value is always an integer between 0 and k. The notation $c_{\min}(Q)$ refers to the minimum capacity of an edge e such that $n_Q(e) > 0$.

$$\begin{array}{ll} \max & \sum_Q c_{\min}(Q) y_Q \\ \text{s.t.} & \sum_{Q: n_Q(e)>0} \frac{n_Q(e) c_{\min}(Q)}{k \cdot c(e)} y_Q \le 1 \quad \forall e \\ & y_Q \ge 0 \quad \forall Q \end{array} \tag{11}$$

For a path-tuple Q, the "Q-saturating flow" is a multicommodity flow that sends $c_{\min}(Q)/k$ units of flow on each of the k paths in Q. (The scaling by $1/k$ is necessary, to ensure that the Q-saturating flow doesn't exceed the capacity of any edge, even if the minimum-capacity edge of Q belongs to all k of the paths in Q.) A primal-feasible vector for the linear program 11 can be interpreted as a weighted sum of Q-saturating flows, weighted by y_Q. The coefficients in the capacity constraint for each e are justified by the observation that a Q-saturating flow sends a total of $n_Q(e)c_{\min}(Q)/k$ units of flow on edge e.

The value of γ for this linear program is $1/k$, so after at most $km\ln(m)/\varepsilon^2$ we obtain a $(1-2\varepsilon)$-approximation to the maximum concurrent multicommodity flow. The minimization problem $\arg\min\{(x_t^\intercal Ay)/(p^\intercal y)\}$ has the following interpretation: when y is the indicator vector of a path-tuple Q, then $p^\intercal y = c_{\min}(Q)$, while Ay is the vector whose i^{th} component is $\frac{n_Q(e_i)\, c_{\min}(Q)}{k\, c(e_i)}$. Thus, letting $Q_1, \ldots, Q_k$ denote the k paths that make up Q, we have

$$x_t^\intercal Ay = \sum_i \frac{x_{t,i}\, n_Q(e_i)\, c_{\min}(Q)}{k\, c(e_i)} = \frac{c_{\min}(Q)}{k} \sum_i \frac{x_{t,i}}{c(e_i)} \cdot n_Q(e_i) = \frac{c_{\min}(Q)}{k} \sum_{j=1}^{k} \sum_{e_i \in Q_j} \frac{x_{t,i}}{c(e_i)}$$

$$\frac{x_t^\intercal Ay}{p^\intercal y} = \frac{1}{k} \sum_{j=1}^{k} \sum_{e_i \in Q_j} \frac{x_{t,i}}{c(e_i)}.$$

Hence, the ratio $\frac{x_t^\intercal Ay}{p^\intercal y}$ is minimized by choosing Q to be the k-tuple consisting of the minimum-cost path from s_j to t_j, for each $j = 1, \ldots, k$, with respect to the edge costs $\frac{x_{t,i}}{c(e_i)}$.

Algorithm 4 Maximum concurrent multicommodity flow algorithm

Given: Parameter $\varepsilon > 0$.
Initialize: $\delta = \varepsilon^2/(\ln m)$, $x = \mathbf{1}$, $f_1 = \cdots = f_k = 0$, $z = 0$.
while $z \prec \mathbf{1}$ **do**
    for $i = 1, \ldots, k$ **do**
        $Q_i \leftarrow$ minimum cost path from s_i to t_i, with respect to edge costs $x_e/c(e)$.
        Update flow f_i by sending $\delta\, c_{\min(P_j)}/k$ units of flow on Q_i.
    end for
    for all edges e **do**
        $n_Q(e) \leftarrow$ the number of i such that $e \in Q_i$.
        $r \leftarrow \frac{c_{\min}(Q)\, n_Q(e)}{k\, c(e)}$.
        $x_e \leftarrow (1+\varepsilon)^r x_e$.
        $z_e \leftarrow z_e + \delta\, r$.
    end for
end while

4 The sparsest cut problem

Given the importance of the max-flow min-cut theorem in discrete mathematics and optimization, it is natural to wonder if there is an analogue of this theorem for multicommodity flows.

If one adopts the interpretation that "a minimum cut is an edge set whose capacity certifies an upper bound on the maximum flow," then the next question is: what upper bounds on throughput or concurrent multicommodity can be certified by an edge set?

Definition 1. Let G be a graph with edge capacities $c(e) \geq 0$ and source-sink pairs $\{(s_i, t_i)\}_{i=1}^{k}$. An edge set A is said to *separate* a source-sink pair (s_i, t_i) if every path from s_i to t_i contains an edge of A. A *cut* is an edge set that separates at least one source-sink pair. A *multicut* is an edge set that separates every source-sink pair. The *sparsity* of a cut is its capacity divided by the number of source-sink pairs it separates.

If G contains a multicut A of capacity c, then the throughput of any multicommodity flow cannot exceed c, since each unit of flow must consume at least one unit of capacity on one of the edges in A. A similar argument shows that if G contains a cut A with sparsity c, then the maximum concurrent flow rate cannot exceed c.

Unlike in the case of single-commodity flows, it is *not* the case that the maximum throughput is equal to the minimum capacity of a multicut, not is it the case that the maximum concurrent flow rate is equal to the sparsest cut value. In both cases, the relevant cut-defined quantity may exceed the flow-defined quantity, by only by a factor of $O(\log k)$ in undirected graphs. This bound is known to be tight up to constant factors. In directed graphs the situation is worse: the minimum multicut may exceed the maximum throughput by $\Theta(k)$ and this is again tight in terms of k, but the way this gap depends on n (the number of vertices) in the worst case remains an open question.

In this section we will present a randomized algorithm to construct a cut whose (expected) sparsity is within a $O(\log k)$ factor of the maximum concurrent flow rate, in undirected graphs. Thus, we will be giving an algorithmic proof of the $O(\log k)$-approximate max-flow min-cut theorem for concurrent multicommodity flows.

4.1 Fractional cuts

To start designing the algorithm, let us recall the sparsest cut LP and its dual. (In the following linear programs, the index Q ranges over k-tuples of paths joining each source to its sink.)

$$\begin{array}{lll} \max & \sum_Q y_Q & \\ \text{s.t.} & \forall e & \sum_Q n_Q(e) y_Q \leq c(e) \\ & \forall Q & y_Q \geq 0 \end{array} \qquad \begin{array}{lll} \min & \sum_e c(e) x_e & \\ \text{s.t.} & \forall Q & \sum_e n_Q(e) x_e \geq 1 \\ & \forall e & x_e \geq 0 \end{array}$$

If one interprets $x = (x_e)_{e \in E}$ as a vector of edge lengths, then the expression $\sum_e n_Q(e) x_e$ in the dual LP represents the sum of the lengths of all paths in Q. Thus, a feasible solution of the dual LP is an assignment of a length to each edge of G, such that the sum of shortest-path lengths between all source-sink pairs is at least 1. For example, if there is a cut A whose sparsity is C/p because it separates p source-sink pairs and has capacity C, then we obtain a dual-feasible vector by setting $x_e = 1/p$ if e belongs to A and $x_e = 0$ otherwise. For each of the p source-sink pairs separated by A, their distance in the graph with edge lengths defined by x is at least

$1/p$, and therefore the combined distance of all source-sink pairs is at least 1 as required by the dual feasibility condition. For this particular dual-feasible vector x, the dual objective function is $\sum_{e \in A} c(e)/p = C/p$, which matches the sparsity of A. Thus, we have confirmed that the optimum of the dual LP is a lower bound on the sparsest cut value. (Which we knew anyhow, because the optimum of the dual LP coincides with the maximum concurrent multicommodity flow rate, and we already knew that was a lower bound on the sparsest cut value.)

Owing to these considerations, a dual-feasible vector x is often called a *fractional cut* and $\sum_e c(e)x_e$ is called the sparsity of the fractional cut. Our randomized algorithm for the sparsest cut problem starts by computing an optimal (or approximately optimal) solution x to the dual LP — for example, using the multiplicative weights algorithm developed in the preceding section — and then "rounding" x to produce a cut whose sparsity exceeds the sparsity of x by a factor of at most $O(\log k)$, in expectation.

4.2 Dependent rounding

One natural idea for transforming a fractional cut into a genuine cut is to sample a random edge set by selecting each edge e independently with probability x_e. This turns out to be a terrible idea. For example, consider that case that $k = 1$ (a single-commodity flow problem) and G is the complete bipartite graph $K_{2,n}$; the two nodes on the left side of the bipartition are the source and sink, s and t. In this graph there is a fractional cut defined by setting $x_e = 1/2$ for every edge e. However, if we construct a random edge by sampling every edge independently with probability $1/2$, the probability of separating s from t is exponentially small in n.

Rather than independent randomized rounding, a better plan is to do some sort of *dependent rounding*. Once again, the case of single-commodity flows is a fertile source of intuition. Suppose x is a fractional cut for a single-commodity flow problem with source s and sink t. Using x, we will construct a random cut based on a sort of "breadth-first search" starting from s. For every vertex u let $d(s, u)$ denote the length of the shortest path from s to u when edge lengths are defined by x. Choose a uniformly random number $r \in [0, 1]$, and cut all edges (u, v) such that $d(s, u) \leq r < d(s, v)$. This random cut always separates s from t: on every path from s to t there is an earliest vertex whose distance from s exceeds r, and the edge leading into this vertex belongs to the cut. The expected capacity of the cut can be computed by linearity of expectation: for any edge $e = (u, v)$, the probability that the random cut contains e is $|d(s, u) - d(s, v)|$, which is bounded above by x_e. Hence the expected capacity of the random cut is bounded above by $\sum_e c(e)x_e$. We have thus shown that in the special case of single-commodity flow, for any fractional cut of capacity C, there is a simple randomized algorithm to compute a cut whose expected capacity is at most C.

The randomized sparsest cut algorithm that we will develop uses a similar dependent rounding scheme based on breadth-first search, but this time starting from a set of sources rather than just one source. The precise sampling procedure looks a little bit strange at first sight. Here it is:

1. Sample t uniformly at random from the set $\{0, 1, \ldots, \lfloor \log(2k) \rfloor\}$.

2. Sample a random set W by selecting each element of the set $\{s_1, t_1, s_2, t_2, \ldots, s_k, t_k\}$ independently with probability 2^{-t}.

3. Sample a uniformly random r in $[0, 1]$.

4. Cut all edges (u, v) such that $d(u, W) < r < d(v, W)$, where the expression $d(u, W)$ refers to the minimum of $d(u, w)$ over all $w \in W$.

Why does this work? We have to estimate two things: the expected capacity of the cut, and the expected number of source-sink pairs that it separates.

Expected capacity. Estimating the expected capacity is surprisingly easy. It closely parallels the argument in the single-commodity case. For an edge $e = (u, v)$, no matter what set W is chosen, we have

$$\Pr(d(u, W) < r < d(v, W)) = |d(u, W) - d(v, W)| \leq x_e$$

so the expected combined capacity of the edges in the cut, by linearity of expectation, is at most $\sum_e c(e)x_e$, the value of the fractional cut x. Recall that this is equal to the maximum concurrent flow rate, if x is an optimal solution to the dual of the maximum concurrent flow LP.

Expected number of separated pairs. For a source-sink pair (s_i, t_i), the probability that the cut separates s_i from t_i is

$$\int_0^1 [\Pr(d(s_i, W) < r < d(t_i, W)) + \Pr(d(t_i, W) < r < d(s_i, W))]\, dr$$

To prove a lower bound on this integral, we will show that for $0 < r < \frac{1}{2}d(s_i, t_i)$, the integrand is bounded below by $\Omega(1/\log(2k))$. This will imply that the integral is bounded below by $\Omega(1/\log(2k))d(s_i, t_i)$. Recall that dual-feasibility of x implies that $\sum_{i=1}^k d(s_i, t_i) = 1$. Thus, the expected number of source-sink pairs separated by our random cut is $\Omega(1/\log(2k))$.

For $0 < r < \frac{1}{2}d(s_i, t_i)$ let S and T denote the subsets of $\{s_1, t_1, \ldots, s_k, t_k\}$ consisting of all terminals within distance r of s_i and t_i, respectively. Note that S and T are non-empty (they contain s_i and t_i, respectively) and they are disjoint, because $r < \frac{1}{2}d(s_i, t_i)$. The event that $d(s_i, W) < r < d(t_i, W)$ is the same as the event that $S \cap W$ is nonempty but $T \cap W$ is not, and similarly for the event $d(t_i, W) < r < d(s_i, W)$. Hence the integrand $\Pr(d(s_i, W) < r < d(t_i, W)) + \Pr(d(t_i, W) < r < d(s_i, W))$ is equal to the probability that precisely one of the sets $S \cap W, T \cap W$ is non-empty. Note that whenever $|(S \cup T) \cap W| = 1$, it is always the case that precisely one of the sets $S \cap W, T \cap W$ is non-empty. Let $h = |S \cup T|$ There is a unique $t \in \{0, 1, \ldots, \lfloor \log(2k) \rfloor\}$ such that $2^t < h \leq 2^{t+1}$. Assuming this value of t is sampled in the first step of our sampling algorithm, the probability that W contains exactly one element of $S \cup T$ is precisely

$$h \cdot 2^{-t} \cdot (1 - 2^{-t})^{h-1} = \frac{h}{2^t} \cdot \left(1 + \frac{1}{2^t - 1}\right)^{-(h-1)} > e^{-(h-1)/(2^t-1)} \geq e^{-3}.$$

So the integrand is bounded below by $e^{-3} \cdot \frac{1}{\log(2k)}$ when $0 < r < \frac{1}{2}d(s_i, t_i)$, which completes the proof.

4.3 Rejection sampling

You may notice that we promised a sampling algorithm that produces a random cut whose expected sparsity is $O(\log k)$ times the maximum concurrent flow rate $\sum_e c(e)x_e$. Instead we have given a sampling algorithm that produces a random cut A such that

$$\frac{\mathbb{E}[\text{cap}(A)]}{\mathbb{E}[\text{sep}(A)]} \leq e^3 \log(2k) \sum_e c(e)x_e. \tag{12}$$

which is not quite the same thing. (Here, $\text{cap}(A)$ denotes the capacity of A and $\text{sep}(A)$ denotes the number of source-sink pairs that it separates.) To fix this problem, we rewrite (12) as follows, using the formula $\text{cap}(A) = \text{sep}(A) \cdot \text{sparsity}(A)$ along with the definition of the expected value of a random variable:

$$e^3 \log(2k) \sum_e c(e)x_e \geq \frac{\sum_A \Pr(A)\ \text{cap}(A)}{\sum_A \Pr(A)\ \text{sep}(A)} = \frac{\sum_A \Pr(A)\ \text{sep}(A) \cdot \text{sparsity}(A)}{\sum_A \Pr(A)\ \text{sep}(A)}.$$

So, if we adjust our sampling rule so that the probability of sampling a given cut A is scaled up by $\text{sep}(A)$ (and then renormalized so that probabilities sum up to 1) we get a random cut whose expected sparsity is at most $e^3 \log(2k) \sum_e c(e)x_e$, as desired. One way to adjust the probabilities in this way is to use *rejection sampling*, which leads to the following algorithm.

Algorithm 5 Rounding a fractional cut to a sparse cut.

Given: fractional cut x defining shortest-path distances $d(\cdot, \cdot)$.
repeat
    Sample t uniformly at random from the set $\{0, 1, \ldots, \lfloor \log(2k) \rfloor\}$.
    Sample a random set W by selecting each element of the set $\{s_1, t_1, s_2, t_2, \ldots, s_k, t_k\}$ independently with probability 2^{-t}.
    Sample a uniformly random r in $[0, 1]$.
    $A = \{(u, v) \mid d(u, W) < r < d(v, W)\}$.
    Sample a uniformly random $j \in \{1, \ldots, k\}$.
until $j \leq \text{sep}(A)$

Why does this work? Let $\Pr(A)$ denote the probability of sampling A under the previous algorithm. Imagine that we modified the algorithm to run a single iteration of the **repeat** loop and either output A if it passes the test $j \leq \text{sep}(A)$ at the end of the loop, or else the algorithm simply fails and outputs nothing. For any cut A, the probability that this modified algorithm outputs A would be $\Pr(A) \cdot \frac{\text{sep}(A)}{k}$. In other words, conditional on succeeding, the modified algorithm samples a cut from exactly the rescaled distribution that we wanted to sample from. By repeating the loop until it succeeds, we guarantee that the algorithm draws one sample from this conditional distribution.

Design and Analysis of Algorithms	Lecture Notes on Convex Optimization
CS 6820, Fall 2016	21 Nov – 2 Dec 2016

Let $\mathcal{D}$ be a convex subset of $\mathbb{R}^n$. A function $f : \mathcal{D} \to \mathbb{R}$ is convex if it satisfies

$$f(tx + (1-t)y) \;\leq\; tf(x) \;+\; (1-t)f(y)$$

for all $x, y \in \mathbb{R}^n$ and $0 \leq t \leq 1$. An equivalent (but not obviously equivalent) definition is that f is convex if and only if for every x in the relative interior of $\mathcal{D}$ there is a *lower bounding linear function* of the form

$$\ell_x(y) = f(x) + (\nabla_x f)^\intercal (y - x)$$

such that $f(y) \geq \ell_x(y)$ for all $y \in \mathcal{D}$. The vector $\nabla_x f \in \mathbb{R}^n$ is called a *subgradient* of f at x. It need not be unique, but it is unique almost everywhere, and it equals the gradient of f at points where the gradient is well-defined. The lower bounding linear function ℓ_x can be interpreted as the function whose graph constitutes the tangent hyperplane to the graph of f at the point $(x, f(x))$.

The *constrained convex optimization* problem is to find a point $x \in \mathcal{D}$ at which $f(x)$ is minimized (or approximately minimized). *Unconstrained convex optimization* is the case when $\mathcal{D} = \mathbb{R}^n$. The coordinates of the optimal point x need not, in general, be rational numbers, so it is unclear what it means to output an exactly optimal point x. Instead, we will focus on algorithms for ε-approximate convex optimization, meaning that the algorithm must output a point $\tilde{x}$ such that $f(\tilde{x}) \leq \varepsilon + \min_{x \in \mathcal{D}}\{f(x)\}$. We will assume that we are given an oracle for evaluating $f(x)$ and $\nabla_x f$ at any $x \in \mathcal{D}$, and we will express the running times of algorithms in terms of the number of calls to these oracles.

The following definition spells out some properties of convex functions that govern the efficiency of algorithms for minimizing them.

Definition 0.1. Let $f : \mathcal{D} \to \mathbb{R}$ be a convex function.

1. f is L-Lipschitz if
$$|f(y) - f(x)| \leq L \cdot \|y - x\| \qquad \forall x, y \in \mathcal{D}.$$
2. f is α-strongly convex if
$$f(y) \geq \ell_x(y) + \tfrac{1}{2}\alpha\|y - x\|^2 \qquad \forall x, y \in \mathcal{D}.$$
Equivalently, f is α-strongly convex if $f(x) - \frac{1}{2}\alpha\|x\|^2$ is a convex function of x.
3. f is β-smooth if
$$f(y) \leq \ell_x(y) + \tfrac{1}{2}\beta\|y - x\|^2 \qquad \forall x, y \in \mathcal{D}.$$
Equivalently, f is β-smooth if its gradient is β-Lipschitz, i.e. if
$$\|\nabla_x f - \nabla_y f\| \leq \beta\|x - y\| \qquad \forall x, y \in \mathcal{D}.$$
4. f has condition number κ if it is α-strongly convex and β-smooth where $\beta/\alpha \leq \kappa$.

The quintessential example of an α-strongly convex function is $f(x) = x^\intercal A x$ when A is a symmetric positive definite matrix whose eigenvalues are all greater than or equal to $\frac{1}{2}\alpha$. (*Exercise: prove that any such function is α-strongly convex.*) When $f(x) = x^\intercal A x$, the condition number of f is also equal to the condition number of A, i.e. the ratio between the maximum and minimum eigenvalues of A. In geometric terms, when κ is close to 1, it means that the level sets of f are nearly round, while if κ is large it means that the level sets of f may be quite elongated.

The quintessential example of a function that is convex, but is neither strongly convex nor linear, is $f(x) = (a^\intercal x)^+ = \max\{a^\intercal x, 0\}$, where a is any nonzero vector in $\mathbb{R}^n$. This function satisfies $\nabla_x f = 0$ when $a^\intercal x < 0$ and $\nabla_x f = a$ when $a^\intercal x > 0$.

1 Gradient descent for Lipschitz convex functions

If we make no assumption about f other than that it is L-Lipschitz, there is a simple but slow algorithm for unconstrained convex minimization that computes a sequence of points, each obtained from the preceding one by subtracting a fixed scalar multiple of the gradient.

Algorithm 1 Gradient descent with fixed step size

Parameters: Starting point $x_0 \in \mathbb{R}^n$, step size $\gamma > 0$, number of iterations $T \in \mathbb{N}$.

1: **for** $t = 0, \ldots, T-1$ **do**
2: $\quad x_{t+1} = x_t - \gamma \nabla_{x_t} f$
3: **end for**
4: Output $\tilde{x} = \arg\min\{f(x_0), \ldots, f(x_T)\}$.

Let x^* denote a point in $\mathbb{R}^n$ at which f is minimized. The analysis of the algorithm will show that if $\|x^* - x_0\| \leq D$ then gradient descent (Algorithm 3) with $\gamma = \varepsilon/L^2$ succeeds in $T = L^2D^2/\varepsilon^2$ iterations. The key parameter in the analysis is the squared distance $\|x_t - x^*\|^2$. The following lemma does most of the work, by showing that this parameter must decrease if $f(x_t)$ is sufficiently far from $f(x^*)$.

Lemma 1.1. $\|x_{t+1} - x^*\|^2 \leq \|x_t - x^*\|^2 - 2\gamma(f(x_t) - f(x^*)) + \gamma^2 L^2$.

Proof. Letting $x = x_t$ we have

$$\begin{aligned}
\|x_{t+1} - x^*\|^2 &= \|x - x^* - \gamma \nabla_x f\|^2 \\
&= \|x - x^*\|^2 - 2\gamma(\nabla_x f)^\intercal (x - x^*) + \gamma^2 \|\nabla_x f\|^2 \\
&= \|x - x^*\|^2 - 2\gamma\left[\ell_x(x) - \ell_x(x^*)\right] + \gamma^2 \|\nabla_x f\|^2 \\
&\leq \|x - x^*\|^2 - 2\gamma(f(x) - f(x^*)) + \gamma^2 \|\nabla_x f\|^2.
\end{aligned}$$

The proof concludes by observing that the L-Lipschitz property of f implies $\|\nabla_x f\| \leq L$. □

Let $\Phi(t) = \|x^t - x^*\|^2$. When $\gamma = \varepsilon/L^2$, the lemma implies that whenever $f(x_t) > f(x^*) + \varepsilon$, we have

$$\Phi(t) - \Phi(t+1) > 2\gamma\varepsilon - \gamma^2 L^2 = \varepsilon^2/L^2. \tag{1}$$

Since $\Phi(0) \leq D$ and $\Phi(t) \geq 0$ for all t, the equation (1) cannot be satisfied for all $0 \leq t \leq L^2D^2/\varepsilon^2$. Hence, if we run gradient descent for $T = L^2D^2/\varepsilon^2$ iterations, it succeeds in finding a point $\tilde{x}$ such that $f(\tilde{x}) \leq f(x^*) + \varepsilon$.

2 Gradient descent for smooth convex functions

If f is β-smooth, then we can obtain an improved convergence bound for gradient descent with step size $\gamma = 1/\beta$. The analysis of the algorithm will show that $O(1/\varepsilon)$ iterations suffice to find a point $\tilde{x}$ where $f(\tilde{x}) \leq f(x^*) + \varepsilon$, improving the $O(1/\varepsilon^2)$ iteration bound for convex functions that are Lipschitz but not necessarily smooth. The material in this section is drawn from Bubeck, *Convex Optimization: Algorithms and Complexity*, in Foundations and Trends in Machine Learning, Vol. 8 (2015).

A first observation which justifies the choice of step size $\gamma = 1/\beta$ is that with this step size, under the constraint that f is β-smooth, gradient descent is guaranteed to make progress.

Lemma 2.1. *If f is convex and β-smooth and $y = x - \frac{1}{\beta}(\nabla_x f)$ then*

$$f(y) \leq f(x) - \frac{1}{2\beta}\|\nabla_x f\|^2. \tag{2}$$

Proof. Using the definition of β-smoothness and the formula for y,

$$f(y) \leq f(x) + (\nabla_x f)^\intercal \left(-\frac{1}{\beta}\nabla_x f\right) + \frac{1}{2}\beta \left\|\frac{1}{\beta}(\nabla_x f)\right\|^2 = f(x) - \frac{1}{2\beta}\|\nabla_x f\|^2. \tag{3}$$

as claimed. □

The next lemma shows that if f is β-smooth and ∇_x and ∇_y are sufficiently different, the lower bound $f(y) \geq \ell_x(y)$ can be significantly strengthened.

Lemma 2.2. *If f is convex and β-smooth, then for any $x, y \in \mathbb{R}^n$ we have*

$$f(y) \geq \ell_x(y) + \frac{1}{2\beta}\|\nabla_x f - \nabla_y f\|^2. \tag{4}$$

Proof. Let $z = y - \frac{1}{\beta}(\nabla_y f - \nabla_x f)$. Then

$$f(z) \geq \ell_x(z) = f(x) + (\nabla_x f)^\intercal (z - x) \tag{5}$$

$$f(z) \leq \ell_y(z) + \frac{1}{2}\beta\|y - z\|^2 = f(y) + (\nabla_y f)^\intercal (z - y) + \frac{1}{2}\beta\|y - z\|^2 \tag{6}$$

and, combining (5) with (6), we have

$$f(y) \geq f(x) + (\nabla_x f)^\intercal (z - x) + (\nabla_y f)^\intercal (y - z) - \frac{1}{2}\beta\|y - z\|^2 \tag{7}$$

$$= f(x) + (\nabla_x f)^\intercal (y - x) + (\nabla_y f - \nabla_x f)^\intercal (y - z) - \frac{1}{2}\beta\|y - z\|^2 \tag{8}$$

$$= \ell_x(y) + \frac{1}{2\beta}\|\nabla_y f - \nabla_x f\|^2, \tag{9}$$

where the last line follows from the equation $y - z = \frac{1}{\beta}(\nabla_y f - \nabla_x f)$. □

Remark 2.3. Lemma 2.2 furnishes a proof that when f is convex and β-smooth, its gradient satisfies the β-Lipschitz inequality

$$\|\nabla_x f - \nabla_y f\| \leq \beta\|x - y\| \qquad \forall x, y \in \mathcal{D}, \tag{10}$$

as claimed in Definition 0.1. The converse, i.e. the fact that β-smoothness follows from property (10), is an easy consequence of the mean value theorem and is left as an exercise.

Lemma 2.2 implies the following corollary, which says that an iteration of gradient descent with step size $\gamma = 1/\beta$ cannot increase the distance from the optimum point, x^*, when the objective function is convex and β-smooth.

Lemma 2.4. *If $y = x - \frac{1}{\beta}(\nabla_x f)$ then $\|y - x^*\| \leq \|x - x^*\|$.*

Proof. Applying Lemma 2.2 twice and expanding the formulae for $\ell_x(y)$ and $\ell_y(x)$, we obtain

$$f(y) \geq f(x) + (\nabla_x)^\intercal (y - x) + \frac{1}{2\beta}\|\nabla_x f - \nabla_y f\|^2$$

$$f(x) \geq f(y) + (\nabla_y)^\intercal (x - y) + \frac{1}{2\beta}\|\nabla_x f - \nabla_y f\|^2$$

Summing and rearranging terms, we derive

$$(\nabla_x f - \nabla_y f)^\intercal (x - y) \geq \frac{1}{\beta} \|\nabla_x f - \nabla_y f\|^2. \tag{11}$$

Now expand the expression for the squared distance from y to x^*.

$$\begin{aligned}
\|y - x^*\|^2 &= \left\| x - x^* - \tfrac{1}{\beta}(\nabla_x f) \right\|^2 \\
&= \|x - x^*\|^2 - \frac{2}{\beta}(\nabla_x f)^\intercal (x - x^*) + \frac{1}{\beta^2}\|\nabla_x f\|^2 \\
&= \|x - x^*\|^2 - \frac{2}{\beta}(\nabla_x f - \nabla_{x^*} f)^\intercal (x - x^*) + \frac{1}{\beta^2}\|\nabla_x f\|^2 \\
&\leq \|x - x^*\|^2 - \frac{2}{\beta^2}\|\nabla_x f - \nabla_{x^*} f\|^2 + \frac{1}{\beta^2}\|\nabla_x f\|^2 \\
&= \|x - x^*\|^2 - \frac{1}{\beta^2}\|\nabla_x f\|^2,
\end{aligned}$$

which confirms that $\|y - x^*\| \leq \|x - x^*\|$. □

To analyze gradient descent using the preceding lemmas, define $\delta_t = f(x_t) - f(x^*)$. Lemma 2.1 implies

$$\delta_{t+1} \leq \delta_t - \frac{1}{2\beta}\|\nabla_{x_t} f\|^2. \tag{12}$$

Convexity of f also implies

$$\begin{aligned}
\delta_t &\leq (\nabla_{x_t} f)^\intercal (x_t - x^*) \\
&\leq \|\nabla_{x_t} f\| \cdot \|x_t - x^*\| \\
&\leq \|\nabla_{x_t} f\| \cdot D \\
\tfrac{\delta_t}{D} &\leq \|\nabla_{x_t} f\|
\end{aligned} \tag{13}$$

where $D \geq \|x_1 - x^*\|$, and the third line follows from Lemma 2.4. Combining (12) with (13) yields

$$\begin{aligned}
\delta_{t+1} &\leq \delta_t - \frac{\delta_t^2}{2\beta D^2} \\
\frac{1}{\delta_t} &\leq \frac{1}{\delta_{t+1}} - \frac{\delta_t}{\delta_{t+1}} \cdot \frac{1}{2\beta D^2} \\
\frac{\delta_t}{\delta_{t+1}} \cdot \frac{1}{2\beta D^2} &\leq \frac{1}{\delta_{t+1}} - \frac{1}{\delta_t} \\
\frac{1}{2\beta D^2} &\leq \frac{1}{\delta_{t+1}} - \frac{1}{\delta_t}
\end{aligned}$$

where the last line used the inequality $\delta_{t+1} \leq \delta_t$ (Lemma 2.1).

We may conclude that

$$\frac{1}{\delta_T} \geq \frac{1}{\delta_0} + \frac{T}{2\beta D^2} \geq \frac{T}{2\beta D^2}$$

from which it follows that $\delta_T \leq 2\beta D^2 / T$, hence $T = 2\beta D^2 \varepsilon^{-1}$ iterations suffice to ensure that $\delta_T \leq \varepsilon$, as claimed.

3 Gradient descent for smooth, strongly convex functions

The material in this section is drawn from Boyd and Vandenberghe, *Convex Optimization*, published by Cambridge University Press and available for free download (with the publisher's permission) at `http://www.stanford.edu/~boyd/cvxbook/`.

We will assume that f is α-strongly convex and β-smooth, with condition number $\kappa = \beta/\alpha$. Rather than assuming an upper bound on the *distance* between the starting point x_0 and x^*, as in the preceding section, we will merely assume that $f(x_0) - f(x^*) \leq B$ for some upper bound B.

We will analyze an algorithm which, in each iteration, moves in the direction of $-\nabla f(x)$ until it reaches the point on the ray $\{x - t\nabla f(x) \mid t \geq 0\}$ where the function f is (exactly or approximately) minimized. This one-dimensional minimization problem is called *line search* and can be efficiently accomplished by binary search on the parameter t. The advantage of gradient descent combined with line search is that it is able to take large steps when the value of f is far from its minimum, and we will see that this is a tremendous advantage in terms of the number of iterations.

Algorithm 2 Gradient descent with line search

1: **repeat**
2: $\quad \Delta x = -\nabla_x f$.
3: $\quad$ Choose $t \geq 0$ so as to minimize $f(x + t\Delta x)$.
4: $\quad x \leftarrow x + t\Delta x$.
5: **until** $\|\nabla_x f\| \leq 2\varepsilon\alpha$

To see why the stopping condition makes sense, observe that strong convexity implies

$$\begin{aligned} \ell_x(x^*) - f(x^*) &\leq -\tfrac{\alpha}{2}\|x - x^*\|^2 \\ f(x) - f(x^*) &\leq (\nabla_x f)^T(x - x^*) - \tfrac{\alpha}{2}\|x - x^*\|^2 \\ &\leq \max_{t\in\mathbb{R}} \left\{\|\nabla_x f\| t - \tfrac{\alpha}{2}t^2\right\} \\ &\leq \tfrac{\|\nabla_x f\|^2}{2\alpha}. \end{aligned} \tag{14}$$

The last line follows from basic calculus. The stopping condition $\|\nabla_x f\|^2 \leq 2\varepsilon\alpha$ thus ensures that $f(x) - f(x^*) \leq \varepsilon$ as desired.

To bound the number of iterations, we show that $f(x) - f(x^*)$ decreases by a prescribed multiplicative factor in each iteration. First observe that for any t,

$$\begin{aligned} f(x + t\Delta x) - \ell_x(x + t\Delta x) &\leq \frac{\beta}{2}\|t\Delta x\|^2 = \frac{\beta}{2}\|\nabla f(x)\|^2 t^2 \\ f(x + t\Delta x) - f(x^*) &\leq \ell_x(x + t\Delta x) - f(x^*) + \frac{\beta}{2}\|\nabla f(x)\|^2 t^2 \\ &= f(x) - f(x^*) + \nabla f(x)^\intercal(t\Delta x) + \frac{\beta}{2}\|\nabla f(x)\|^2 t^2 \\ &= f(x) - f(x^*) - \|\nabla f(x)\|^2 t + \frac{\beta}{2}\|\nabla f(x)\|^2 t^2 \end{aligned}$$

The right side can be made as small as $f(x) - f(x^*) - \frac{\|\nabla f(x)\|^2}{2\beta}$ by setting $t = \frac{\|\nabla f(x)\|}{\beta}$. Our algorithm sets t to minimize the left side, hence

$$f(x + t\Delta x) - f(x^*) \leq f(x) - f(x^*) - \frac{\|\nabla f(x)\|^2}{2\beta}. \tag{15}$$

Recalling from inequality (14) that $\|\nabla f(x)\|^2 \geq 2\alpha(f(x) - f(x^*))$, we see that inequality (15) implies

$$f(x + t\Delta x) - f(x^*) \leq f(x) - f(x^*) - \frac{\alpha}{\beta}[f(x) - f(x^*)] = \left(1 - \frac{1}{\kappa}\right)[f(x) - f(x^*)]. \tag{16}$$

This inequality shows that the difference $f(x) - f(x^*)$ shrinks by a factor of $1 - \frac{1}{\kappa}$, or better, in each iteration. Thus, after no more than $\log_{1-1/\kappa}(\varepsilon/B)$ iterations, we reach a point where $f(x) - f(x^*) \leq \varepsilon$, as was our goal. The expression $\log_{1-1/\kappa}(\varepsilon/B)$ is somewhat hard to parse, but we can bound it from above by a simpler expression, by using the inequality $\ln(1 - x) \leq -x$.

$$\log_{1-1/\kappa}(\varepsilon/B) = \frac{\ln(\varepsilon/B)}{\ln(1 - \alpha/\beta)} = \frac{\ln(B/\varepsilon)}{-\ln(1 - \alpha/\beta)} \leq \kappa \ln\left(\frac{B}{\varepsilon}\right).$$

The key things to notice about this upper bound are that it is logarithmic in $1/\varepsilon$—as opposed to the algorithm from the previous lecture whose number of iterations was quadratic in $1/\varepsilon$—and that the number of iterations depends linearly on the condition number. Thus, the method is very fast when the Hessian of the convex function is not too ill-conditioned; for example when κ is a constant the number of iterations is merely logarithmic in $1/\varepsilon$.

Another thing to point out is that our bound on the number of iterations has *no dependence on the dimension*, n. Thus, the method is suitable even for very high-dimensional problems, as long as the high dimensionality doesn't lead to an excessively large condition number.

4 Constrained convex optimization

In this section we analyze algorithms for constrained convex optimization, when $\mathcal{D} \subset \mathbb{R}^n$. This introduces a new issue that gradient-descent algorithms must deal with: if an iteration starts at a point x_t which is close to the boundary of $\mathcal{D}$, one step in the direction of the gradient might lead to a point y_t outside of $\mathcal{D}$, in which case the algorithm must somehow find its way back inside. The most obvious way of doing this is to move back to the closest point of $\mathcal{D}$. We will analyze this algorithm in §**??** below. The other way to avoid this problem is to avoid stepping outside $\mathcal{D}$ in the first place. This idea is put to use in the *conditional gradient descent* algorithm, also known as the Franke-Wolfe algorithm. We analyze this algorithm in §**??** below.

4.1 Projected gradient descent

Define the projection of a point $y \in \mathbb{R}^n$ onto a closed, convex set $\mathcal{D} \subseteq \mathbb{R}^n$ to be the point of $\mathcal{D}$ closest to y,

$$\Pi_{\mathcal{D}}(y) = \arg\min\{\|x - y\| \,:\, x \in \mathcal{D}\}.$$

Note that this point is always unique: if $x \neq x'$ both belong to $\mathcal{D}$, then their midpoint $\frac{1}{2}(x + x')$ is strictly closer to y than at least one of x, x'. A useful lemma is the following, which says that moving from y to $\Pi_{\mathcal{D}}(y)$ entails simultaneously moving closer to *every* point of $\mathcal{D}$.

Lemma 4.1. *For all $y \in \mathbb{R}^n$ and $x \in \mathcal{D}$,*

$$\|\Pi_{\mathcal{D}}(y) - x\| \leq \|y - x\|.$$

Proof. Let $z = \Pi_{\mathcal{D}}(y)$. We have

$$\|y - x\|^2 = \|(y - z) + (z - x)\|^2 = \|y - z\|^2 + 2(y - z)^\intercal(z - x) + \|z - x\|^2 \geq 2(y - z)^\intercal(z - x) + \|z - x\|^2.$$

The lemma asserts that $\|y-x\|^2 \geq \|z-x\|^2$, so it suffices to prove that $(y-z)^\intercal(z-x) \geq 0$. Assume to the contrary that $(y-z)^\intercal(x-z) > 0$. This means that the triangle formed by x, y, z has an acute angle at z. Consequently, the point on line segment xz nearest to y cannot be z. This contradicts the fact that the entire line segment is contained in $\mathcal{D}$, and z is the point of $\mathcal{D}$ nearest to y. □

We are now ready to define and analyze the project gradient descent algorithm. It is the same as the fixed-step-size gradient descent algorithm (Algorithm 3) with the sole modification that after taking a gradient step, it applies the operator $\Pi_{\mathcal{D}}$ to get back inside $\mathcal{D}$ if necessary.

Algorithm 3 Projected gradient descent

Parameters: Starting point $x_0 \in \mathcal{D}$, step size $\gamma > 0$, number of iterations $T \in \mathbb{N}$.

1: **for** $t = 0, \ldots, T-1$ **do**
2: $\quad x_{t+1} = \Pi_{\mathcal{D}}(x_t - \gamma \nabla_{x_t} f)$
3: **end for**
4: Output $\tilde{x} = \arg\min\{f(x_0), \ldots, f(x_T)\}$.

There are two issues with this approach.

1. Computing the operator $\Pi_{\mathcal{D}}$ may be a challenging problem in itself. It involves minimizing the convex quadratic function $\|x-y_t\|^2$ over $\mathcal{D}$. There are various reasons why this may be easier than minimizing f. For example, the function $\|x-y_t\|^2$ is smooth and strongly convex with condition number $\kappa = 1$, which is about as well-behaved as a convex objective function could possibly be. Also, the domain $\mathcal{D}$ might have a shape which permits the operation $\Pi_{\mathcal{D}}$ to be computed by a greedy algorithm or something even simpler. This happens, for example, when $\mathcal{D}$ is a sphere or a rectangular box. However, in many applications of projected gradient descent the step of computing $\Pi_{\mathcal{D}}$ is actually the most computationally burdensome step.
2. Even ignoring the cost of applying $\Pi_{\mathcal{D}}$, we need to be sure that it doesn't counteract the progress made in moving from x_t to $x_t - \gamma \nabla_{x_t} f$. Lemma 4.1 works in our favor here, as long as we are using $\|x_t - x^*\|$ as our measure of progress, as we did in §1.

For the sake of completeness, we include here the analysis of projected gradient descent, although it is a repeat of the analysis from §1 with one additional step inserted in which we apply Lemma 4.1 to assert that the projection step doesn't increase the distance from x^*.

Lemma 4.2. $\|x_{t+1} - x^*\|^2 \leq \|x_t - x^*\|^2 - 2\gamma(f(x_t) - f(x^*)) + \gamma^2 L^2$.

Proof. Letting $x = x_t$ we have

$$\begin{aligned}
\|x_{t+1} - x^*\|^2 &= \|\Pi_{\mathcal{D}}(x - \gamma \nabla_x f) - x^*\|^2 \\
&\leq \|(x - \gamma \nabla_x f) - x^*\|^2 \textit{(By Lemma 4.1)} \\
&= \|x - x^* - \gamma \nabla_x f\|^2 \\
&= \|x - x^*\|^2 - 2\gamma(\nabla_x f)^\intercal (x - x^*) + \gamma^2 \|\nabla_x f\|^2 \\
&= \|x - x^*\|^2 - 2\gamma\left[\ell_x(x) - \ell_x(x^*)\right] + \gamma^2 \|\nabla_x f\|^2 \\
&\leq \|x - x^*\|^2 - 2\gamma(f(x) - f(x^*)) + \gamma^2 \|\nabla_x f\|^2.
\end{aligned}$$

The proof concludes by observing that the L-Lipschitz property of f implies $\|\nabla_x f\| \leq L$. □

The rest of the analysis of projected gradient descent is exactly the same as th analysis of gradient descent in §1; it shows that when f is L-Lipschitz, if we set $\gamma = \varepsilon/L^2$ and run $T \geq L^2 D^2 \varepsilon^{-2}$ iterations, the algorithm finds a point $\tilde{x}$ such that $f(\tilde{x}) \leq f(x) + \varepsilon$.

4.2 Conditional gradient descent

In the conditional gradient descent algorithm, introduced by Franke and Wolfe in 1956, rather than taking a step in the direction of $-\nabla_x f$, we globally minimize the linear function $(\nabla_x f)^\intercal y$ over $\mathcal{D}$, then take a small step in the direction of the minimizer. This has a number of advantages relative to projected gradient descent.

1. Since we are taking a step from one point of $\mathcal{D}$ towards another point of $\mathcal{D}$, we never leave $\mathcal{D}$.
2. Minimizing a linear function over $\mathcal{D}$ is typically easier (computationally) than minimizing a quadratic function of $\mathcal{D}$, which is what we need to do when computing the operation $\Pi_{\mathcal{D}}$ in projected gradient descent.
3. When moving towards the global minimizer of $(\nabla_x f)^\intercal y$, rather than moving parallel to $-\nabla_x f$, we can take a longer step without hitting the boundary of $\mathcal{D}$. The longer steps will tend to reduce the number of iterations required for finding a near-optimal point.

Algorithm 4 Conditional gradient descent

Parameters: Starting point $x_0 \in \mathcal{D}$, step size sequence $\gamma_1, \gamma_2, \ldots$, number of iterations $T \in \mathbb{N}$.

1: **for** $t = 0, \ldots, T-1$ **do**
2: $\quad y_t = \arg\min_{y \in \mathcal{D}} \{(\nabla_{x_t} f)^\intercal y\}$
3: $\quad x_{t+1} = \gamma_t y_t + (1-\gamma_t) x_t$
4: **end for**
5: Output $\tilde{x} = \arg\min\{f(x_0), \ldots, f(x_T)\}$.

We will analyze the algorithm when f is β-smooth and $\gamma_t = \frac{2}{t+1}$ for all t.

Theorem 4.3. *If D is an upper bound on the distance between any two points of $\mathcal{D}$, and f is β-smooth, then the sequence of points computed by conditional gradient descent satisfies*

$$f(x_t) - f(x^*) \leq \frac{2\beta R^2}{t+1}$$

for all $t \geq 2$.

Proof. We have

$$\begin{aligned} f(x_{t+1}) - f(x_t) &\leq (\nabla_{x_t} f)^\intercal (x_{t+1} - x_t) + \tfrac{1}{2}\beta \|x_{t+1} - x_t\|^2 \beta\text{-smoothness} \\ &\leq \gamma_t (\nabla_{x_t} f)^\intercal (y_t - x_t) + \tfrac{1}{2}\beta\gamma_t^2 R^2 \, def'n \text{ of } x_{t+1} \\ &\leq \gamma_t (\nabla_{x_t} f)^\intercal (x^* - x_t) + \tfrac{1}{2}\beta\gamma_t^2 R^2 \, def'n \text{ of } y_t \\ &\leq \gamma_t (f(x^*) - f(x_t)) + \tfrac{1}{2}\beta\gamma_t^2 R^2 \, convexity. \end{aligned}$$

Letting $\delta_t = f(x_t) - f(x^*)$, we can rearrange this inequality to

$$\delta_{t+1} \leq (1-\gamma_t)\delta_t + \tfrac{1}{2}\beta\gamma_t^2 R^2.$$

When $t = 1$ we have $\gamma_t = 1$, hence this inequality specializes to $\delta_2 \leq \frac{1}{2}\beta R^2$. Solving the recurrence for $t > 1$ with initial condition $\delta_2 \leq \frac{\beta}{2} R^2$, we obtain $\delta_t \leq \frac{2\beta R^2}{t+1}$ as claimed. □

Cornell University, Fall 2012 **CS 6820: Algorithms**
Lecture notes on spectral methods in algorithm design

Studying the eigenvalues and eigenvectors of matrices has powerful consequences for at least three areas of algorithm design: graph partitioning, analysis of high-dimensional data, and analysis of Markov chains. Collectively, these techniques are known as *spectral methods* in algorithm design. These lecture notes present the fundamentals of spectral methods.

1 Review: symmetric matrices, their eigenvalues and eigenvectors

This section reviews some basic facts about real symmetric matrices. If $A = (a_{ij})$ is an $n - \times - n$ square symmetric matrix, then $\mathbb{R}^n$ has a basis consisting of eigenvectors of A, these vectors are mutually orthogonal, and all of the eigenvalues are real numbers. Furthermore, the eigenvectors and eigenvalues can be characterized as solutions of natural maximization or minimization problems involving *Rayleigh quotients*.

Definition 1.1. If x is a nonzero vector in $\mathbb{R}^n$ and A is an $n - \times - n$ matrix, then the Rayleigh quotient of x with respect to A is the ratio

$$RQ_A(x) = \frac{x^\mathsf{T} A x}{x^\mathsf{T} x}.$$

Definition 1.2. If A is an $n - \times - n$ matrix, then a linear subspace $V \subseteq \mathbb{R}^n$ is called an *invariant subspace* of A if it satisfies $Ax \in V$ for all $x \in V$.

Lemma 1.3. *If A is a real symmetric matrix and V is an invariant subspace of A, then there is some $x \in V$ such that $RQ_A(x) = \inf\{RQ_A(y) \mid y \in V\}$. Any $x \in V$ that minimizes $RQ_A(x)$ is an eigenvector of A, and the value $RQ_A(x)$ is the corresponding eigenvalue.*

Proof. If x is a vector and r is a nonzero scalar, then $RQ_A(x) = RQ_A(rx)$, hence every value attained in V by the function RQ_A is attained on the unit sphere $S(V) = \{x \in V \mid x^\mathsf{T} x = 1\}$. The function RQ_A is a continuous function on $S(V)$, and $S(V)$ is compact (closed and bounded) so by basic real analysis we know that RQ_A attains its minimum value at some unit vector $x \in S(V)$. Using the quotient rule we can compute the gradient

$$\nabla RQ_A(x) = \frac{2Ax - 2(x^\mathsf{T} A x)x}{(x^\mathsf{T} x)^2}. \tag{1}$$

At the vector $x \in S(V)$ where RQ_A attains its minimum value in V, this gradient vector must be orthogonal to V; otherwise, the value of RQ_A would decrease as we move away from x in the direction of any $y \in V$ that has negative dot product with $\nabla RQ_A(x)$. On the other hand, our assumption that V is an invariant subspace of A implies that the right side of (1) belongs to V. The only way that $\nabla RQ_A(x)$ could be orthogonal to V while also belonging to V is if it is the zero vector, hence $Ax = \lambda x$ where $\lambda = x^\mathsf{T} A x = RQ_A(x)$. □

Lemma 1.4. *If A is a real symmetric matrix and V is an invariant subspace of A, then $V^\perp = \{x \mid x^\mathsf{T} y = 0 \; \forall y \in V\}$ is also an invariant subspace of A.*

Proof. If V is an invariant subspace of A and $x \in V^\perp$, then for all $y \in V$ we have

$$(Ax)^\mathsf{T} y = x^\mathsf{T} A^\mathsf{T} y = x^\mathsf{T} A y = 0,$$

hence $Ax \in V^\perp$. □

Combining these two lemmas, we obtain a recipe for extracting all of the eigenvectors of A, with their eigenvalues arranged in increasing order.

Theorem 1.5. *Let A be an $n - \times - n$ real symmetric matrix and let us inductively define sequences*

$$\begin{aligned} x_1, \ldots, x_n &\in \mathbb{R}^n \\ \lambda_1, \ldots, \lambda_n &\in \mathbb{R} \\ \{0\} = V_0 \subseteq V_1 \subseteq \cdots \subseteq V_n &= \mathbb{R}^n \\ \mathbb{R}^n = W_0 \supseteq W_1 \supseteq \cdots \supseteq W_n &= \{0\} \end{aligned}$$

by specifying that

$$\begin{aligned} x_i &= \operatorname{argmin} \{RQ_A(x) \mid x \in W_{i-1}\} \\ \lambda_i &= RQ_A(x_i) \\ V_i &= \operatorname{span}(x_1, \ldots, x_i) \\ W_i &= V_i^\perp. \end{aligned}$$

Then $x_1, \ldots, x_n$ are mutually orthogonal eigenvectors of A, and $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$ are the corresponding eigenvalues.

Proof. The proof is by induction on i. The induction hypothesis is that $\{x_1, \ldots, x_i\}$ is a set of mutually orthogonal eigenvectors of A constituting a basis of V_i, and $\lambda_1 \leq \cdots \leq \lambda_i$ are the corresponding eigenvectors. Given this induction hypothesis, and the preceding lemmas, the proof almost writes itself. Each time we select a new x_i, it is guaranteed to be orthogonal to the preceding ones because $x_i \in W_{i-1} = V_{i-1}^\perp$. The linear subspace V_{i-1} is A-invariant because it is spanned by eigenvectors of A; by Lemma 1.4 its orthogonal complement W_{i-1} is also A-invariant and this implies, by Lemma 1.3 that x_i is an eigenvector of A and λ_i is its corresponding eigenvalue. Finally, $\lambda_i \geq \lambda_{i-1}$ because $\lambda_{i-1} = \min\{RQ_A(x) \mid x \in W_{i-2}\}$, while $\lambda_i = RQ_A(x_i) \in \{RQ_A(x) \mid x \in W_{i-2}\}$. □

An easy corollary of Theorem 1.5 is the *Courant-Fischer Theorem.*

Theorem 1.6 (Courant-Fischer)**.** *The eigenvalues $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$ of an $n - \times - n$ real symmetric matrix satisfy:*

$$\forall k \; \lambda_k = \min_{\dim(V)=k} \left(\max_{x \in V} RQ_A(x) \right) = \max_{\dim(W)=n-k+1} \left(\min_{x \in W} RQ_A(x) \right).$$

Proof. The vector space W_{k-1} constructed in the proof of Theorem 1.5 has dimension $n-k+1$, and by construction it satisfies $\min_{x \in W_{k-1}} RQ_A(x) = \lambda_k$. Therefore

$$\max_{\dim(W)=n-k+1} \left(\min_{x \in W} RQ_A(x) \right) \geq \lambda_k.$$

If $W \subseteq \mathbb{R}^n$ is any linear subspace of dimension $n-k+1$ then $W \cap V_k$ contains a nonzero vector x, because $\dim(W) + \dim(V_k) > n$. Since $V_k = \text{span}(x_1, \ldots, x_k)$ we can write $x = a_1 x_1 + \cdots + a_k x_k$. Rescaling $x_1, \ldots, x_k$ if necessary, we can assume that they are all unit vectors. Then, using the fact that $x_1, \ldots, x_k$ are mutually orthogonal eigenvectors of A, we obtain

$$RQ_A(x) = \frac{\lambda_1 a_1 + \cdots + \lambda_k a_k}{a_1 + \cdots + a_k} \leq \lambda_k.$$

Therefore $\max_{\dim(W)=n-k+1} (\min_{x \in W} RQ_A(x)) \leq \lambda_k$. Combining this with the inequality derived in the preceding paragraph, we obtain $\max_{\dim(W)=n-k+1} \min_{x \in W} RQ_A(x) = \lambda_k$. Replacing A with $-A$, and k with $n-k+1$, we obtain $\min_{\dim(V)=k} (\max_{x \in V} RQ_A(x)) = \lambda_k$. □

2 The Graph Laplacian

Two symmetric matrices play a vital role in the theory of graph partitioning. These are the Laplacian and normalized Laplacian matrix of a graph G.

Definition 2.1. If G is an undirected graph with non-negative edge weights $w(u, v) \geq 0$, the *weighted degree* of a vertex u, denoted by $d(u)$, is the sum of the weights of all edges incident to u. The Laplacian matrix of G is the matrix L_G with entries

$$(L_G)_{uv} = \begin{cases} d(u) & \text{if } u = v \\ -w(u,v) & \text{if } u \neq v \text{ and } (u,v) \in E \\ 0 & \text{if } u \neq v \text{ and } (u,v) \notin E. \end{cases}$$

If D_G is the diagonal matrix whose (u, u)-entry is $d(u)$, and if G has no vertex of weighted degree 0, then the normalized Laplacian matrix of G is

$$\overline{L}_G = D_G^{-1/2} L_G D_G^{-1/2}.$$

The eigenvalues of L_G and $\overline{L}_G$ will be denoted in these notes by $\lambda_1(G) \leq \cdots \leq \lambda_n(G)$ and $\nu_1(G) \leq \cdots \leq \nu_n(G)$. When the graph G is clear from context, we will simply write these as $\lambda_1, \ldots, \lambda_n$ or $\nu_1, \ldots, \nu_n$.

The "meaning" of the Laplacian matrix is best explained by the following observation.

Observation 2.2. The Laplacian matrix L_G is the unique symmetric matrix satisfying the following relation for all vectors $x \in \mathbb{R}^V$.

$$x^{\mathsf{T}} L_G x = \sum_{(u,v) \in E} w(u,v)(x_u - x_v)^2. \tag{2}$$

The following lemma follows easily from Observation 2.2.

Lemma 2.3. *The Laplacian matrix of a graph G is a positive semidefinite matrix. Its minimum eigenvalue is 0. The multiplicity of this eigenvalue equals the number of connected components of G.*

Proof. The right side of (2) is always non-negative, hence L_G is positive semidefinite. The right side is zero if and only if x is constant on each connected component of G (i.e., it satisfies $x_u = x_v$ whenever u, v belong to the same component), hence the multiplicity of the eigenvalue 0 equals the number of connected components of G. □

The normalized Laplacian matrix has a more obscure graph-theoretic meaning than the Laplacian, but its eigenvalues and eigenvectors are actually more tightly connected to the structure of G. Accordingly, we will focus on normalized Laplacian eigenvalues and eigenvectors in these notes. The cost of doing so is that the matrix $\overline{L}_G$ is a bit more cumbersome to work with. For example, when G is connected the 0-eigenspace of L_G is spanned by the all-ones vector $\mathbf{1}$ whereas the 0-eigenspace of $\overline{L}_G$ is spanned by the vector $\mathbf{d}^{1/2} = D_G^{1/2}\mathbf{1}$.

3 Conductance and expansion

We will relate the eigenvalue $\nu_2(G)$ to two graph parameters called the *conductance* and *expansion* of G. Both of them measure the value of the "sparsest" cut, with respect to subtly differing notions of sparsity. For any set of vertices S, define

$$d(S) = \sum_{u \in S} d(u)$$

and define the edge boundary

$$\partial S = \{e = (u, v) \mid \text{exactly one of } u, v \text{ belongs to } S\}.$$

The *conductance* of G is

$$\phi(G) = \min_{(S,\overline{S})} \left\{ d(V) \cdot \frac{w(\partial S)}{d(S)d(\overline{S})} \right\}$$

and the *expansion* of G is

$$h(G) = \min_{(S,\overline{S})} \left\{ \frac{w(\partial S)}{\min\{d(S), d(\overline{S})\}} \right\},$$

where the minimum in both cases is over all vertex sets $S \neq \emptyset, V$. Note that for any such S,

$$\frac{d(V)}{d(S)d(\overline{S})} = \frac{d(V)}{\min\{d(S), d(\overline{S})\} \cdot \max\{d(S), d(\overline{S})\}} = \frac{1}{\min\{d(S), d(\overline{S})\}} \cdot \frac{d(V)}{\max\{d(S), d(\overline{S})\}}.$$

The second factor on the right side is between 1 and 2, and it easily follows that

$$h(G) \leq \phi(G) \leq 2h(G).$$

Thus, each of the parameters $h(G), \phi(G)$ is a 2-approximation to the other one. Unfortunately, it is not known how to compute a $O(1)$-approximation to either of these parameters in polynomial time. In fact, assuming the Unique Games Conjecture, it is NP-hard to compute an $O(1)$-approximation to either of them.

4 Cheeger's Inequality: Lower Bound on Conductance

There is a sense, however, in which $\nu_2(G)$ constitutes an approximation to $\phi(G)$. To see why, let us begin with the following characterization of $\nu_2(G)$ that comes directly from Courant-Fischer.

$$\nu_2(G) = \min\left\{ \frac{x^\mathsf{T} \overline{L}_G x}{x^\mathsf{T} x} \middle| \, x \neq 0,\, x^\mathsf{T} D_G^{1/2} \mathbf{1} = 0 \right\} = \min\left\{ \frac{y^\mathsf{T} L_G y}{y^\mathsf{T} D_G y} \middle| \, y \neq 0,\, y^\mathsf{T} D_G \mathbf{1} = 0 \right\}.$$

The latter equality is obtained by setting $x = D_G^{1/2} y$.

The following lemma allows us to rewrite the Rayleigh quotient $\frac{y^\mathsf{T} L_G y}{y^\mathsf{T} D_G y}$ in a useful form, when $y^\mathsf{T} D_G \mathbf{1} = 0$.

Lemma 4.1. *For any vector y we have*

$$y^\mathsf{T} D_G y \geq \frac{1}{2d(V)} \sum_{u \neq v} d(u)d(v)(y(u) - y(v))^2,$$

with equality if and only if $y^\mathsf{T} D_G \mathbf{1} = 0$.

Proof.

$$\begin{aligned}
\frac{1}{2} \sum_{u \neq v} d(u)d(v)(y(u) - y(v))^2 &= \frac{1}{2} \sum_{u \neq v} d(u)d(v)[y(u)^2 + y(v)^2] - \sum_{u \neq v} d(u)d(v)y(u)y(v) \\
&= \sum_{u \neq v} d(u)d(v)y(u)^2 - \sum_{u \neq v} d(u)d(v)y(u)y(v) \\
&= \sum_{u,v} d(u)d(v)y(u)^2 - \sum_{u,v} d(u)d(v)y(u)y(v) \\
&= d(V) \sum_u d(u)y(u)^2 - \left(\sum_u d(u)y(u) \right)^2 \\
&= d(V) y^\mathsf{T} D_G y - \left(y^\mathsf{T} D_G \mathbf{1} \right)^2.
\end{aligned}$$

□

A corollary of the lemma is the formula

$$\nu_2(G) = \inf \left\{ d(V) \frac{\sum_{(u,v)\in E(G)} w(u,v)(y(u)-y(v))^2}{\sum_{u<v} d(u)d(v)(y(u)-y(v))^2} \,\middle|\, \text{denominator is nonzero} \right\}, \quad (3)$$

where the summation over $u < v$ in the denominator is meant to indicate that each unordered pair $\{u, v\}$ of distinct vertices contributes exactly one term to the sum. The corollary is obtained by noticing that the numerator and denominator on the right side are invariant under adding a scalar multiple of $\mathbf{1}$ to y, and hence one of the vectors attaining the infimum is orthogonal to $D_G\mathbf{1}$.

Let us evaluate the quotient on the right side of (3) when y is the characteristic vector of a cut $(S, \overline{S})$, defined by

$$y(u) = \begin{cases} 1 & \text{if } u \in S \\ 0 & \text{if } u \in \overline{S}. \end{cases}$$

In that case,

$$\sum_{(u,v)\in E(G)} w(u,v)(y(u)-y(v))^2 = \sum_{(u,v)\in \partial S} w(u,v) = w(\partial S)$$

while

$$\sum_{u<v} d(u)d(v)(y(u)-y(v))^2 = \sum_{u\in S}\sum_{v\in\overline{S}} d(u)d(v) = d(S)d(\overline{S}).$$

Hence,

$$\nu_2(G) \le d(V)\frac{w(\partial S)}{d(S)d(\overline{S})},$$

and taking the minimum over all $(S, \overline{S})$ we obtain

$$\nu_2(G) \le \phi(G).$$

5 Cheeger's Inequality: Upper Bound on Conductance

The inequality $\nu_2(G) \le \phi(G)$ is the easy half of Cheeger's Inequality; the more difficult half asserts that there is also an upper bound on $\phi(G)$ of the form

$$\phi(G) \le \sqrt{8\nu_2(G)}.$$

Owing to the inequality $\phi(G) \le 2h(G)$, it suffices to prove that

$$h(G) \le \sqrt{2\nu_2(G)}$$

and that is, in fact, the next thing we will prove.

For any vector y that is not a scalar multiple of $\mathbf{1}$, define

$$Q(y) = d(V)\frac{\sum_{(u,v)\in E(G)} w(u,v)(y(u)-y(v))^2}{\sum_{u<v} d(u)d(v)(y(u)-y(v))^2}.$$

Given any such y, we will find a cut $(S,\overline{S})$ such that $\frac{w(\partial S)}{\min\{d(S),d(\overline{S})\}} \leq \sqrt{2Q(y)}$; the upper bound $h(G) \leq \sqrt{2\nu_2(G)}$ follows immediately by choosing y to be a vector minimizing $Q(y)$. In fact, if we number the vertices of G as $v_1, v_2, \ldots, v_n$ such that $y_1 \leq y_2 \leq \cdots \leq y_n$, we will show that it suffices to take S to be one of the sets $\{y_1, \ldots, y_k\}$ for $1 \leq k < n$.

Note that $Q(y)$ is unchanged when we add a scalar multiple of $\mathbf{1}$ to y. Accordingly, we can assume without loss of generality that

$$\sum_{y_i<0} d(v_i) \leq \sum_{y_i\geq 0} d(v_i)$$
$$\sum_{y_i\leq 0} d(v_i) \geq \sum_{y_i>0} d(v_i)$$

For d-regular graphs, this essentially means that we're setting the median of the components of y to be zero. For irregular graphs, it essentially says that we're balancing the total degree of the vertices with positive $y(u)$ and those with negative $y(u)$.

Now here comes the most unmotivated part of the proof. Define a vector z by

$$z_i = \begin{cases} -y_i^2 & \text{if } y_i < 0 \\ y_i^2 & \text{if } y_i \geq 0. \end{cases}$$

Note also that $Q(y)$ is unchanged when we multiply y by a nonzero scalar. Accordingly, we can assume that $z_n - z_1 = 1$. Now choose a threshold value t uniformly at random from the interval $[z_1, z_n]$ and let

$$S = \{v_i \mid z_i < t\}.$$

We will prove that

$$\frac{\mathbb{E}[w(\partial S)]}{\mathbb{E}[\min\{d(S), d(\overline{S})\}]} \leq \sqrt{2Q(y)}$$

from which it follows that

$$\mathbb{E}[w(\partial S)] \leq \sqrt{2Q(y)} \cdot \mathbb{E}[\min\{d(S), d(\overline{S})\}]$$

and consequently that there is at least one S in the support of our distribution such that

$$w(\partial S) \leq \sqrt{2Q(y)} \cdot \min\{d(S), d(\overline{S})\}.$$

It is surprisingly easy to evaluate $\mathbb{E}[\min\{d(S), d(\overline{S})\}]$. Each vertex v_i contributes $d(v_i)$ to the expression inside the expectation operator when it belongs to the smaller side of the

cut, which happens if and only if t lands between 0 and z_i, an event with probability $|z_i|$. Consequently,

$$\mathbb{E}[\min\{d(S), d(\overline{S})\}] = \sum_u d(u)|z(u)| = \sum_u d(u)y(u)^2 = y^\mathsf{T} D_G y.$$

Meanwhile, to bound the numerator $\mathbb{E}[w(\partial S)]$, observe that an edge (u, v) contributes $w(u, v)$ to the numerator if and only if it is cut, an event having probability $|z(u) - z(v)|$. A bit of case analysis reveals that

$$\forall u, v \;\; |z(u) - z(v)| \leq |y(u) - y(v)| \cdot (|y(u)| + |y(v)|),$$

since the left and right sides are equal when $y(u), y(v)$ have the same sign, and otherwise the left side equals $y(u)^2 + y(v)^2$ while the right side equals $(|y(u)| + |y(v)|)^2$. Combining this estimate of the numerator with Cauchy-Schwartz, we find that

$$\begin{aligned}
\mathbb{E}[w(\partial S)] &\leq \sum_{(u,v)\in E(G)} w(u,v)|y(u) - y(v)|(|y(u)| + |y(v)|) \\
&\leq \left(\sum_{(u,v)\in E(G)} w(u,v)(y(u) - y(v))^2 \right)^{1/2} \left(\sum_{(u,v)\in E(G)} w(u,v)(|y(u)| + |y(v)|)^2 \right)^{1/2} \\
&\leq \left(\frac{Q(y)}{d(V)} \sum_{u<v} d(u)d(v)(y(u) - y(v))^2 \right)^{1/2} \left(\sum_{(u,v)\in E(G)} w(u,v)(2y(u)^2 + 2y(v)^2) \right)^{1/2} \\
&\leq \left(Q(y) y^\mathsf{T} D_G y\right)^{1/2} \left(2 \sum_u d(u)y(u)^2 \right)^{1/2} \\
&= (2Q(y))^{1/2} y^\mathsf{T} D_G y.
\end{aligned}$$

6 Laplacian eigenvalues and spectral partitioning

We've seen a connection between sparse cuts and eigenvectors of the *normalized* Laplacian matrix. However, in some contexts it is easier to work with eigenvalues and eigenvectors of the unnormalized Laplacian, L_G. One can use eigenvectors of L_G for spectral partitioning, provided one is willing to tolerate weaker bounds for graphs with unbalanced degree sequences. For example, if y is an eigenvector of L_G satisfying $L_G y = \lambda_2 y$ then we can express $Q(y)$ as follows:

$$Q(y) = d(V) \frac{y^\mathsf{T} L_G y}{\sum_{u<v} d(u)d(v)(y(u) - y(v))^2} = \frac{\lambda_2 \|y\|^2 d(V)}{\sum_{u<v} d(u)d(v)(y(u) - y(v))^2}.$$

To estimate the denominator, let $d_{\min}$ and d_{avg} denote the minimum and the average degree of G, respectively. We have

$$\begin{aligned}\sum_{u<v} d(u)d(v)(y(u)-y(v))^2 &= \frac{1}{2}\sum_{u\neq v} d(u)d(v)(y(u)-y(v))^2 \\ &\geq \frac{1}{2}d_{\min}^2 \sum_{u\neq v}(y(u)-y(v))^2 \\ &= nd_{\min}^2 \sum_u y(u)^2 = \frac{d(V)}{d_{\text{avg}}} d_{\min}^2 \|y\|^2.\end{aligned}$$

Hence

$$Q(y) \leq \frac{d_{\text{avg}}}{d(V)} \frac{\lambda_2 \|y\|^2 d(V)}{d_{\min}^2 \|y\|^2} = \left(\frac{d_{\text{avg}}}{d_{\min}^2}\right) \lambda_2.$$

7 Spectral sparsification of graphs

For a dense graph G with n vertices and $m \gg n$ edges, it is often desirable to compute a sparse approximation H, i.e. an edge-weighted graph with the same vertex set but with $O(n)$ or $O(n \log n)$ edges, such that

$$(1-\varepsilon)L_G \preceq L_H \preceq (1+\varepsilon)L_G. \tag{4}$$

Such a graph is called a *spectral sparsification* of G. It is useful because it preserves some of the essential features of G. For example, we have seen that for any vertex set S, if x denotes the vector

$$x_u = \begin{cases} 1 & \text{if } u \in S \\ 0 & \text{if } u \notin S \end{cases}$$

then the capacity of the cut $(S, \overline{S})$, with edge set ∂S, is given by

$$c(\partial S) = x^{\mathsf{T}} L_G x.$$

In light of equation (4) we know that a spectral sparsifier H satisfies

$$(1-\varepsilon)x^{\mathsf{T}} L_G x \leq x^{\mathsf{T}} L_H x \leq (1+\varepsilon)x^{\mathsf{T}} L_G x$$

hence a spectral sparsifier preserves the capacity of every cut in G, up to a factor of $1 \pm \varepsilon$.

Random sampling furnishes a simple method for computing a spectral sparsifier of G. We will be designing and analyzing an algorithm that samples edges $e_1, \ldots, e_k$ independently, each drawn from a probability distribution that will be denoted by $\{\pi(e) \mid e \in E\}$. Designing an appropriate sampling distribution will be the subtlest part of the algorithm, and we will defer discussion of how to choose π for now. The number of sampled edges, k, will turn out to be $O(n \log n)$, but for now we'll also leave k as a parameter of the algorithm whose precise value will be specified later.

For any edge $e = (u, v)$ let δ_e denote the vector whose components are defined by

$$(\delta_e)_w = \begin{cases} -1 & \text{if } w = u \\ 1 & \text{if } w = v \\ 0 & \text{otherwise.} \end{cases}$$

The vector δ_e is only well-defined up to sign. In other words, the undirected edge $e = (u, v)$ is equally well represented as $e = (v, u)$, but these two representations lead to the vectors δ_e and $-\delta_e$, respectively. The sign ambiguity will not matter, because we won't be dealing directly with the vector δ_e but instead with the rank-one matrix $\delta_e \delta_e^{\mathsf{T}}$. The equation $(-\delta_e)(-\delta_e^{\mathsf{T}}) = \delta_e \delta_e^{\mathsf{T}}$ assures that we get the same matrix no matter which choice we make for δ_e.

Recall that the Laplacian of a graph G with edge capacities $c(e)$ is given by the weighted sum

$$L_G = \sum_{e \in E} c(e) \delta_e \delta_e^{\mathsf{T}}. \tag{5}$$

Similarly, for our random graph H, if we choose a "rescaled capacity" for each edge e, and set the capacity of e in H to $\hat{c}(e)$ times the number of times e occurs in the multi-set $\{e_1, \ldots, e_k\}$ of randomly sampled edges, then the Laplacian of H will be given by

$$L_H = \sum_{i=1}^{k} \hat{c}(e_i) \delta_{e_i} \delta_{e_i}^{\mathsf{T}} \tag{6}$$

and its expected value will be

$$\mathbb{E}[L_H] = k \cdot \sum_{e \in E} \pi(e) \hat{c}(e) \delta_e \delta_e^{\mathsf{T}}.$$

To equate $\mathbb{E}[L_H]$ with L_G the simplest thing to do is to equate the coefficient of $\delta_e \delta_e^{\mathsf{T}}$ for each edge e, which necessitates setting

$$\begin{aligned} k \hat{c}(e) \pi(e) &= c(e) \\ \hat{c}(e) &= \tfrac{c(e)}{k \pi(e)}. \end{aligned}$$

Thus, the capacities $\hat{c}(e)$ of the sampled edges will be uniquely determined by the number of sampled edges, k, and the sampling distribution, π.

To analyze the quality of the spectral approximation achieved by the sampling algorithm, we need to estimate the extent to which the random matrix L_H may differ from its expectation, $\mathbb{E}[L_H]$. Since L_H is a sum of independent, identically distributed random matrices—namely, the summands on the right side of (6)—it is natural to use the Ahlswede-Winter Inequality. In our application of the inequality, the average of the k summands has expected value $\frac{1}{k}\mathbb{E}[L_H] = \frac{1}{k} L_G$. Thus, to apply the inequality, we need to find a constant $R \geq 1$ such that for each edge e,

$$\begin{aligned} \hat{c}(e) \delta_e \delta^{\mathsf{T}} &\preceq R \cdot \left(\tfrac{1}{k} L_G\right) \\ c(e) \delta_e \delta_e^{\mathsf{T}} &\preceq R \pi(e) \cdot L_G. \end{aligned} \tag{7}$$

The Ahlswede-Winter Inequality will then ensure that with probability at least $1-2n\,\exp(-\frac{\varepsilon^2 k}{4R})$, we have $(1-\varepsilon)L_G \preceq L_H \preceq (1+\varepsilon)L_G$, as desired. If we want this event to happen with probability at least $\frac{1}{2}$, we set $k = 4R\varepsilon^{-2}\ln(4n)$. Thus, the number of edges we need to sample when constructing H is linearly related to the constant R appearing on the right side of (7), and designing a good graph sparsification algorithms boils down to constructing a distribution $\{\pi(e)\}$ that allows R to be as small as possible.

As a naïve first attempt, we could take π to be the uniform distribution, $\pi(e) = \frac{1}{m}$ for all e. Then, noting that the formula (5) justifies the relation $c(e)\delta_e\delta_e^\mathsf{T} \preceq L_G$ for all e, we see that we just need to make R large enough that $R\pi(e) \geq 1$ for all e. Since we are using $\pi(e) = \frac{1}{m}$ this means setting $R = m$. Our naïve idea of setting π to be the uniform distribution has not worked out well: instead of sparsifying G, we have *increasing* the number of edges from m to $k = 4m\varepsilon^{-2}\ln(4n)$.

One might hope that the uniform sampling technique performs better than the above analysis would suggest. After all, our analysis made use of the relation $c(e)\delta_e\delta_e^\mathsf{T} \preceq L_G$, which typically has a large amount of "slack" because L_G is a sum of m positive semidefinite matrices, only one of which is $c(e)\delta_e\delta_e^\mathsf{T}$. However, on closer inspection, the whose idea of uniform edge sampling is doomed to require sampling $\Omega(m)$ edges in the worst case. To see why, consider the case that G is made up of two cliques K_0, K_1, each of size $n/2$, joined by a single edge, e. Let x denote the vector defined by setting $x_u = 1$ if $u \in K_0$, $x_u = 0$ if $u \in K_1$. If we fail to sample edge e when constructing the sparsifier, H, then $x^\mathsf{T}L_H x = 0$ whereas $x^\mathsf{T}L_G x > 0$, which rules out the possibility that H is a spectral sparsifier of G. Thus, if we sample $o(m)$ edges from the uniform distribution, with probability $1 - o(1)$ we will fail to obtain a spectral sparsifier. Our only hope is to use a non-uniform distribution over edges that assigns higher probability to edges, such as the edge e in the foregoing example, that are "spectrally irreplaceable", meaning that they must be included in any spectral sparsifier of G.

Since our goal is to minimize R, a more principled way of designing the distribution π consists of solving the following semidefinite program whose variables are R and $\{\pi(e) \mid e \in E\}$.

$$\begin{array}{lll} \text{minimize} & R & \\ \text{subject to} & c(e)\delta_e\delta_e^\mathsf{T} \preceq R\pi(e)\cdot L_G & \forall e \in E \\ & \sum_{e\in E}\pi(e) = 1 & \\ & \pi(e) \geq 0 & \forall e \in E \end{array} \tag{8}$$

The first constraint can be rewritten as

$$\forall e = (u,v) \in E \quad R \geq \frac{c(e)}{\pi(e)} \cdot \max\left\{ \frac{(\delta_e^\mathsf{T}x)^2}{x^\mathsf{T}L_G x} \,\middle|\, x \neq 0 \right\} \tag{9}$$

and it will be helpful to solve the maximization problem on the right side. Since δ_e is orthogonal to the nullspace of L_G, the quotient $\frac{c(e)(\delta_e^\mathsf{T}x)^2}{\pi(e)\,x^\mathsf{T}L_G x}$ is unchanged if we add any vector in the nullspace of L_G to x. For this reason, among the set of vectors x that attain the maximum

on the right side of (9) there is one that is orthogonal to the nullspace of L_G and we may assume henceforth that x is such a vector. In particular, this means $L_G^+ L_G x = L_G L_G^+ x = x$. Let $y = L_G^{1/2} x$ and . Then

$$\frac{(\delta_e^\mathsf{T} x)^2}{x^\mathsf{T} L_G x} = \frac{(\delta_e^\mathsf{T} (L_G^+)^{1/2} y)^2}{y^\mathsf{T} y}.$$

For any vector w, the maximum of $\frac{(w^\mathsf{T} y)^2}{y^\mathsf{T} y}$ over nonzero vectors y is attained when y is a unit vector in the direction of w, in which case $\frac{(w^\mathsf{T} y)^2}{y^\mathsf{T} y} = w^\mathsf{T} w$. Substituting $w = (L_G^+)^{1/2} \delta_e$ we find that

$$\max \left\{ \frac{(\delta_e^\mathsf{T} x)^2}{x^\mathsf{T} L_G x} \middle| x \neq 0 \right\} = \left((L_G^+)^{1/2} \delta_e \right)^\mathsf{T} (L_G^+)^{1/2} \delta_e = \delta_e^\mathsf{T} L_G^+ \delta_e.$$

Substituting this into (9) and multiplying both sides by $\pi(e)$ we find that

$$R\pi(e) \geq c(e) \delta_e^\mathsf{T} L_G^+ \delta_e.$$

Summing over e,

$$R = R \left(\sum_e \pi(e) \right) \geq \sum_e c(e) \delta_e^\mathsf{T} L_G^+ \delta_e \tag{10}$$

$$= \operatorname{tr} \left(\sum_e c(e) \delta_e \delta^\mathsf{T} L_G^+ \right) \tag{11}$$

$$= \operatorname{tr} \left(L_G L_G^+ \right) = n - c, \tag{12}$$

where c denotes the number of connected components of G, and the last inequality follows from the fact that $L_G L_G^+$ is the projection on $\mathbb{R}^n$ onto the nullspace of L_G.

Our objective of minimizing R will be served if we make the inequality in line (10) tight, which means setting $R = n - c$ and $\pi(e) = \frac{1}{n-c} c(e) \delta_e^\mathsf{T} L_G^+ \delta_e$ for each edge. This choice of R and $\{\pi(e)\}$ is the optimal solution of the semidefinite program (8) and leads to a spectral sparsifier H with $k < 4n\varepsilon^{-2} \ln(4n)$ edges.

Incidentally, the quantity $c(e) \delta_e^\mathsf{T} L_G^+ \delta_e$ is called the *effective resistance of edge e*. It can be interpreted as the resistance between the endpoints of edge e, if one were to build an electrical network in the shape of the graph G, with each edge e' represented by a resistor of resistance $c(e')$. This connection between electrical networks and graph sparsification is just one of many beautiful connections between electrical networks, spectral graph theory, graph algorithms, and random walks. For more on this topic, see Doyle and Snell's short book, "Random Walks and Electric Networks".

A Additional tools for working with symmetric matrices

This appendix contains some additional tools that are useful in the design and analysis of algorithms involving symmetric matrices.

A.1 Standard matrix functions

There is a standard way of extending any function that maps $\mathbb{R}$ to $\mathbb{R}$ into a function mapping $\mathrm{Sym}_n(\mathbb{R})$ to $\mathrm{Sym}_n(\mathbb{R})$. In this section we define these "standard matrices functions" and present some basic examples and properties.

Definition A.1. If $f : \mathbb{R} \to \mathbb{R}$ is any function, the matrix extension of f is the unique function from $\mathrm{Sym}_n(\mathbb{R})$ to $\mathrm{Sym}_n(\mathbb{R})$ satisfying $f(\mathrm{diag}(\lambda_1, \ldots, \lambda_n)) = \mathrm{diag}(f(\lambda_1), \ldots, f(\lambda_n))$ and $f(QDQ^{-1}) = Qf(D)Q^{-1}$ for every orthogonal matrix Q and diagonal matrix D.

The only subtlety in the definition of the matrix extension of f is that any given $A \in \mathrm{Sym}_n(\mathbb{R})$ can be written as QDQ^{-1} in more than one way, and one needs to verify that the definition of $f(A)$ does not depend on the choice of representation $A = QDQ^{-1}$. We leave this verification to the reader.

Some immediate consequences of the definition are the following.

1. If $A \in \mathrm{Sym}_n(\mathbb{R})$ has eigenvalues $\lambda_1, \ldots, \lambda_n$ and corresponding eigenvectors $x_1, \ldots, x_n$, then $f(A)$ has eigenvalues $f(\lambda_1), \ldots, f(\lambda_n)$ and corresponding eigenvectors $x_1, \ldots, x_n$.

2. If A is symmetric and Q is orthogonal, then $f(QAQ^{-1}) = Qf(A)Q^{-1}$.

3. If f and g are two functions from $\mathbb{R}$ to $\mathbb{R}$ and $f \circ g$, $f + g$, $f \cdot g$ are the functions defined by

$$(f \circ g)(\lambda) = f(g(\lambda)), \quad (f + g)(\lambda) = f(\lambda) + g(\lambda), \quad (f \cdot g)(\lambda) = f(\lambda)\, g(\lambda)$$

   then for all $a \in \mathrm{Sym}_n(\mathbb{R})$,

$$(f \circ g)(A) = f(g(A)), \quad (f + g)(A) = f(A) + g(A), \quad (f \cdot g)(A) = f(A)\, g(A).$$

The following are some useful properties and examples.

1. For any two functions f, g and any matrix $A \in \mathrm{Sym}_n(\mathbb{R})$, the matrices $f(A)$ and $g(A)$ commute with one another. This is due to the identity $f \cdot g = g \cdot f$.

2. If f is a polynomial function $f(\lambda) = \sum_{i=0}^{m} c_i \lambda^i$, then $f(A) = \sum_{i=0}^{n} c_i A^i$, where A^0 is interpreted as the identity matrix.

3. If f is represented by a power series $f(\lambda) = \sum_{i=0}^{\infty} c_i \lambda^i$ that converges on the open interval $(-R, R)$, then $f(A) = \sum_{i=0}^{\infty} c_i A^i$ for every matrix A whose eigenvalues are all contained in the interval $(-R, R)$.

4. An important special case of the preceding example is the matrix exponential function, defined by

$$e^A = \sum_{i=0}^{\infty} \frac{1}{i!} A^i.$$

5. If f is the function

$$f(\lambda) = \begin{cases} \lambda^{-1} & \text{if } \lambda \neq 0 \\ 0 & \text{if } \lambda = 0 \end{cases}$$

then $f(A)$ is denoted by A^+ and is called the Moore-Pensore pseudoinverse (or, simply, pseudoinverse) of A. When A is invertible, A^+ is the inverse of A. More generally, $A^+ A = A A^+$ is the matrix that represents the orthogonal projection of $\mathbb{R}^n$ onto the column space of A.

A.2 The Golden-Thompson Inequality

Theorem A.2 (Golden-Thompson Inequality)**.** *For any matrices $A, B \in \mathrm{Sym}_n(\mathbb{R})$,*

$$\mathrm{tr}\left(e^A e^B\right) \geq \mathrm{tr}\left(e^{A+B}\right).$$

A.3 The PSD ordering on symmetric matrices

Let $\mathrm{Sym}_n(\mathbb{R})$ denote the vector space of n-by-n symmetric matrices over $\mathbb{R}$. If $A - B$ is positive semidefinite we write $A \succeq B$ or $B \preceq A$. This relation is a partial order: it is reflexive (the zero matrix is PSD), antisymmetric (if a matrix and its negation are PSD, it is the zero matrix), and transitive (the sum of two PSD matrices is PSD).

A.4 The Ahlswede-Winter Inequality

The Ahlswede-Winter Inequality is a counterpart of the Chernoff bound, for sums of independent random symmetric PSD matrices rather than sums of independent random scalars.

Theorem A.3 (Ahlswede-Winter Inequality)**.** *Suppose $X_1, X_2, \ldots, X_k$ are mutually independent random, symmetric, positive semidefinite $n \times n$ matrices, and let $U = \mathbb{E}\left[\frac{1}{k}\sum_{i=1}^k X_i\right]$. If $R \geq 1$ is a scalar such that for all i, $X_i \preceq R \cdot U$ with probability 1, then for all $\varepsilon \in (0,1)$,*

$$\Pr\left[(1-\varepsilon)U \preceq \tfrac{1}{k}\sum_{i=1}^k X_i \preceq (1+\varepsilon)U\right] \;\geq\; 1 - 2n \cdot \exp\left(-\tfrac{\varepsilon^2 k}{4R}\right). \tag{13}$$

The proof is similar to the proof of the Chernoff bound. Letting X denote the random sum $\sum_{i=1}^k X_i$, the standard proof of the Chernoff bound uses the exponential generating function $\Phi(t) = \mathbb{E}[e^{tX}]$. Here, the expression e^{tX} is matrix-valued. To make Φ scalar-valued we instead use $\Phi(t) = \mathbb{E}[\mathrm{tr}(e^{tX})]$. This accounts for the extra factor of n in the failure probability on the right side of (13). (The trace of the identity matrix is n, not 1.) The main difficulty that arises in the proof of Ahlswede-Winter, relative to the proof of Chernoff, is that the exponentials of non-commuting matrices do not commute, so the identity $e^{tX} = \prod_{i=1}^n e^{tX_i}$ does not hold. However, the Golden-Thompson Inequality justifies the inequality

$$\mathrm{tr}(e^{tX}) \leq \mathrm{tr}\left(\prod_{i=1}^n e^{tX_i}\right)$$

which is good enough to complete the proof.

Proof. The proof begins by reducing to the case where U is the identity matrix. Since U is symmetric positive semidefinite, it can be written as QDQ^{-1}, where Q is orthogonal and D is a diagonal matrix with non-negative entries arranging in non-increasing order. Replacing each X_i with QX_iQ^{-1}, we can assume henceforth that $U = D$. If the nullspace of D has dimension $d > 0$, then the entries in the final d rows and columns of D are all equal to zero. The relation $X_i \preceq R \cdot D$ implies that any vector in the nullspace of D must also belong to the nullspace of X_i for each i. Thus, for each i, the entries in the final d rows and columns of X_i are all equal to zero. To prove a lower bound on the probability of the event $(1-\varepsilon)D \preceq \frac{1}{k}\sum_{i=1}^{k} X_i \preceq (1+\varepsilon)D$, it suffices to confine our attention to the square submatrices occupying the first $n-d$ rows and columns of each matrix involved.

Having thus reduced to the case that U is a non-singular diagonal matrix D, we may replace each X_i with $D^{-1/2}X_iD^{-1/2}$ to finally reduce to the case when $\mathbb{E}[\frac{1}{k}\sum_{i=1}^{k} X_i]$ is the identity matrix, $\mathbb{1}$.

Now let $X = \frac{1}{k}\sum_{i=1}^{k} X_i$ and consider the function

$$\Phi(t) = \mathbb{E}\left[\operatorname{tr}\left(e^{tX}\right)\right].$$

□