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

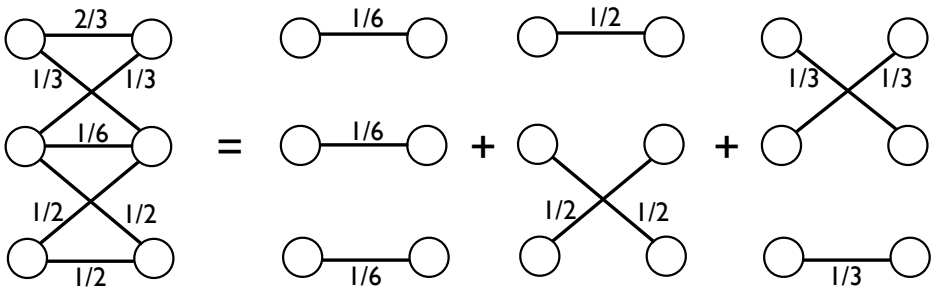

Figure 1: A fractional perfect matching.

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

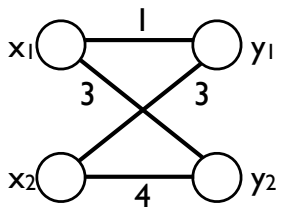

Figure 2: An instance of bipartite minimum cost perfect matching.

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 $O(mn \log n)$.