These notes analyze algorithms for optimization problems involving matchings in 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 *bipartite graph* is a graph whose vertex set is partitioned into two disjoint sets L, R such that each edge has one endpoint in L and the other endpoint in R. When we write a bipartite graph G as an ordered triple $G = (L, R, E)$, it means that L and R are the two vertex sets (called the *left set* and *right set*, respectively) and E is the edge set.

Definition 2. A *matching* in an undirected graph is a set of edges such that no vertex belongs to more than element of the set.

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, $G = (L, R, E)$, is given in its adjacency list representation, and that the bipartition of G—that is, the partition of the vertex set into L and R—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 = ∅.
2: repeat
3:     Find an augmenting path P with respect to M.
4:     M ← M ⊕ 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 $G = (L, R, E)$ is a bipartite graph and M is a matching, the graph G_M is the directed graph formed from G by orienting each edge from L to R if it does not belong to M, and from R to L 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 $L \cap F$ to $R \cap F$ in G_M.*

Proof. If P is a directed path from $L \cap F$ to $R \cap F$ in G_M then P starts and ends at free vertices, and its edges alternate between those that are directed from L to R (which are in the complement of M) and those that are directed from R to L (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 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 L and the other endpoint in R. 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 G_M from $L \cap F$ to $R \cap F$. □

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 G_M and running a graph search algorithm such as BFS or DFS to search for a path from $L \cap F$ to $R \cap F$. Building G_M takes $O(m + n)$ time, where m is the number of edges in G, as does searching 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 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.

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

If (x, y) is any edge of G_M then $d(y) \leq d(x)+1$. Edges of 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 G_M from $L \cap F$ 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 $G_{M'}$, the orientation of every edge in Q is reversed and the orientation of every edge in R is preserved. Therefore, $G_{M'}$ 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. $\ell = \min\{d(v) \mid v \in R \cap F\}$. Let P be any path in $G_{M'}$ from $L \cap F'$ to $R \cap F'$. 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 L to v; if no such path exists $d(v) = \infty$. Recall also that an *advancing* edge in 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 $v \in R \cap F$), marks every vertex as unexplored, and repeatedly finds augmenting paths using the following procedure. Start at an unexplored vertex v in $R \cap F$ 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 $L \cap F$ 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 $L \cap F$ 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 $L \cap F$ to $R \cap F$ that is vertex-disjoint from the selected ones; thus, the selected set forms a blocking set of augmenting paths as desired.

2 Non-bipartite matching

When the graph G is not bipartite, Lemma 2 is still valid: a matching has maximum cardinality if and only if it has no augmenting path. Hence, as before, the problem of finding a maximum matching reduces to the problem of finding an augmenting path with respect to a given matching, or else certifying that there is none. However, whereas in the bipartite case the problem of finding an augmenting path reduced to searching for a path in the directed graph G_M, in the non-bipartite case there is no correspondingly simple reduction.

To see why, it's useful to consider what goes wrong with the naïve idea of searching for an augmenting path using "breadth-first search over the set of alternating paths". Here's one way of making this information idea precise. For a graph G and matching M, define $H(G, M)$ to be a directed graph with the same set of vertices as G, and with a directed edge (u, v) for every pair of vertices such that G contains a path made up of two edges (u, u') and (u', v) such that $(u, u') \notin M$ and $(u', v) \in M$. Note that if G contains an M-augmenting path made up of $2k + 1$ edges, then the first $2k$ of those edges correspond to a k-edge path in $H(G, M)$ that starts in F, the set of free vertices, and ends in $\Gamma(F)$, the set of vertices that are adjacent to a free vertex.

If the converse were true, i.e. if finding a path in $H(G, M)$ from F to $\Gamma(F)$ were equivalent to finding an M-augmenting path in G, then we could design a maximum non-bipartite matching algorithm along exactly the same lines as in the bipartite case. Instead, there is a second alternative represented by the diagram in Figure 1: a simple path in $H(G, M)$ from F to $\Gamma(F)$ might correspond to a *non-simple* alternating walk from F to $N(F)$ in G, i.e. an alternating walk that repeats some vertices.

Definition 6. If G is a graph and M is a matching in G, a *flower* with respect to M is an M-alternating walk $u_0, u_1, \ldots, u_s$ such that:

1. u_0 is a free vertex with respect to M;

2. the vertices $u_0, \ldots, u_{s-1}$ are distinct, whereas $u_s = u_r$ for some number $r < s$

3. r is even and s is odd.

The *stem* of the flower is the path $u_0, u_1, \ldots, u_r$. (Note that it is possible that $r = 0$, in which case the stem is an empty path.) The *blossom* of the flower is the cycle $u_r, \ldots, u_s$.

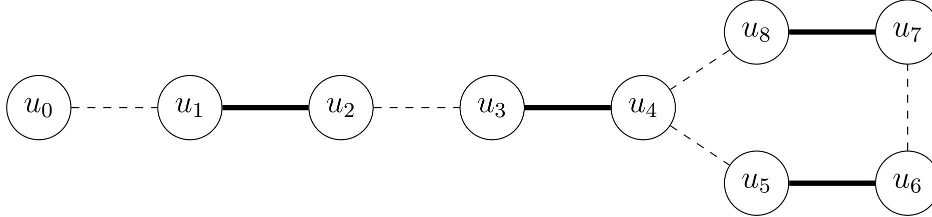

Figure 1: A flower

Lemma 8. *If the graph $H(G, M)$ contains a path P from F to $\Gamma(F)$, then G contains either an M-augmenting path or a flower.*

Proof. Suppose that $P = v_0, v_1, \ldots, v_k$ is a simple path in $H(G, M)$ from F to $\Gamma(F)$. For $i = 1, 2, \ldots, k$, the edge (v_{i-1}, v_i) in $H(G, M)$ corresponds to a sequence of two edges in G, the first lying outside M and the second belonging to M. Denote these two edges by $(u_{2i-2}, u_{2i-1}) \notin M$ and $(u_{2i-1}, u_{2i}) \in M$. Since $v_k = u_{2k}$ belongs to $\Gamma(F)$, we may choose a free vertex u_{2k+1} adjacent to u_{2k}. Now consider the alternating walk $u_0, u_1, \ldots, u_{2k+1}$ in G. If all of its vertices are distinct, then it is an M-augmenting path. Otherwise, let u_s be the earliest instance of a repeated vertex in the alternating walk, and let u_r denote the earlier occurrence of this same vertex.

If s is even, then the edge (u_{s-1}, u_s) belongs to M. Note that this means s is not a free vertex, so $r > 0$. This means that either (u_{r-1}, u_r) or (u_r, u_{r+1}) belongs to M, hence u_{s-1} is equal to either u_{r-1} or u_{r+1}. This contradicts our choice of s unless $r + 1 = s - 1$, which is impossible because the case $(u_r, u_{r+1}) \in M$ only occurs when r is odd, in which case $r + 1 \neq s - 1$ because the left side is even and the right side is odd. Hence, the assumption that s is even leads to a contradiction.

If s and r are both odd, then $(u_r, u_{r+1}) \in M$ so u_s is not a free vertex. In particular this means that $s < 2k + 1$. The edge (u_s, u_{s+1}) belongs to M, which implies that $u_{s+1} = u_{r+1}$. However, since $s + 1$ and $r + 1$ are even, the vertices u_{s+1} and u_{r+1} both belong to P, contradicting the assumption that P is a simple path.

By process of elimination, we have deduced that s is odd and r is even, in which case the sequence $u_0, \ldots, u_s$ constitutes a flower. □

If G contains a flower with blossom B, our algorithm for finding an M-augmenting path in G will depend on an operation called *blossom shrinking* which forms a new graph G/B

with matching M/B, as follows. The vertices of B are replaced with a single vertex $\{v_b\}$. Edges having both endpoints in B are removed. For those having exactly one endpoint in B, that endpoint is changed to v_b and the other endpoint is preserved. Edges having no endpoints in B are unchanged. Note that this operation may produce a multigraph (i.e., there may be multiple edges between the same two vertices) in the case that there is a vertex having more than one neighbor in B. In the event that G/B contains multiple edges between the same two vertices, we can discard all but one of those edges without affecting the algorithm's correctness; however, in our analysis we prefer to treat G/B as a multigraph because it means that every edge of G/B has one unambiguous corresponding edge in G, which simplifies the analysis.

Let M/B denote the set of edges in G/B whose corresponding edge in G belongs to M. Note that M/B is a matching: for all vertices other than v_b it is clear that they belong to at most one edge in M/B, while for v_b this holds because if $u_r, u_{r+1}, \ldots, u_s$ denotes the list of vertices in B, in the order that they occur in the flower, then u_r (also known as u_s) is the only vertex in the blossom that potentially belongs to an edge of M whose other endpoint lies outside of M.

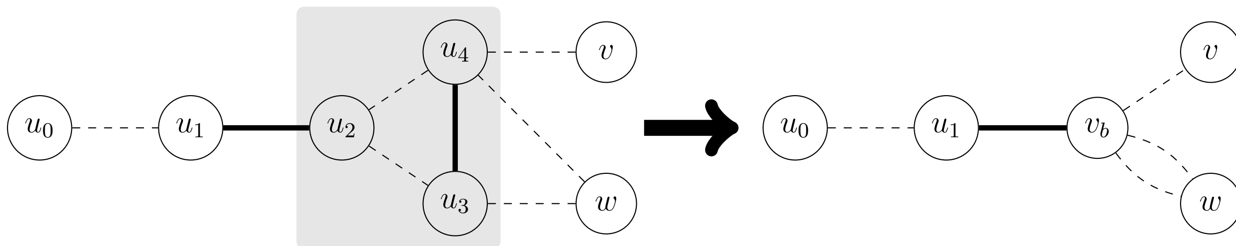

Figure 2: Shrinking a blossom

The following lemma on the relationship between augmenting paths in G and those in G/B accounts for the importance of the blossom shrinking operation.

Lemma 9. *If G is a graph, M is a matching, and B is the blossom of a flower with respect to M, then G/B contains an (M/B)-augmenting path if and only if G contains an M-augmenting path. Furthermore, any (M/B)-augmenting path in G/B can be modified into an M-augmenting path in G in time $|B|$.*

Proof. Denote the vertices of the flower containing B by $u_0, \ldots, u_s$, numbered as in Definition 6. If P is an (M/B)-augmenting path in G/B and P does not contain v_b, then it is already an M-augmenting path in G. Otherwise, P contains an edge (w, v_b) that does not belong to M/B. Let (w, u_t) be the corresponding edge of G, where $r \leq t < s$. An M-augmenting path in G can be constructed by replacing edge (w, v_b) with an M-alternating path from w to u_r in G whose first and last edges do not belong to M. If t is even, then replace edge (w, u_t) with path $w, u_t, u_{t-1}, \ldots, u_r$; if t is odd, then replace (w, u_t) with $w, u_t, u_{t+1}, \ldots, u_s$. Notice that the path segment that replaces (w, u_t) has fewer than $|B|$ edges, and the replacement can be done in $O(|B|)$ time if we use suitable data structures, e.g. representing the path P as a doubly linked list of edges. This justifies the running time bound in the last sentence of the lemma statement.

It remains for us to prove that if G contains an M-augmenting path, then G/B contains an (M/B)-augmenting path. One might expect this to be a simple matter of reversing the operation defined in the preceding paragraph, but in fact it's a little trickier. To see why, consider the augmenting path $\langle v, u_4, u_3, w\rangle$ in Figure 2. The corresponding path in G/B is $\langle v, v_b, w\rangle$ which is not an alternating path with respect to M/B.

Instead, we first let $S = \{(u_{i-1}, u_i) \mid i = 1, \ldots, r\}$ denote the set of edges belonging to the stem of the flower, and we modify M to $M' = M \oplus S$. Note that $|M'| = |M|$ and u_r, u_{r+1}, u_s is a flower with respect to M' (having an empty stem). Hence, B is still a blossom with respect to M', and M'/B is still a matching in G/B, with the same number of edges as M/B. We will now apply the following chain of reasoning to deduce the existence of an (M/B)-augmenting path in G/B.

G has an M-augmenting path	$\Rightarrow M$ is not a maximum matching in G	*(Lemma 2)*
	$\Rightarrow M'$ is not a maximum matching in G	*($\lvert M\rvert = \lvert M'\rvert$)*
	$\Rightarrow G$ has an M'-augmenting path	*(Lemma 2)*
	$\Rightarrow G/B$ has an (M'/B)-augmenting path	*(proven below)*
	$\Rightarrow M'/B$ is not a maximum matching in G/B	*(Lemma 2)*
	$\Rightarrow M/B$ is not a maximum matching in G/B	*($\lvert M/B\rvert = \lvert M'/B\rvert$)*
	$\Rightarrow G/B$ has an (M/B)-augmenting path	*(Lemma 2)*

The only step that remains to be justified is that the existence of an M'-augmenting path in G implies the existence of an (M'/B)-augmenting path in G/B. Suppose that P is an M'-augmenting path in G. If P does not intersect B then it is already an augmenting path in G/B. Otherwise, since B contains only one free vertex (namely u_r), we know that at least one endpoint of P does not belong to B. Number the vertices of P as $v_0, v_1, \ldots, v_t$ with $v_0 \notin B$, and suppose that v_k is the lowest-numbered vertex of P that belongs to B. Then the path $\langle v_0, v_1, \ldots, v_{k-1}, v_b\rangle$ is an (M'/B)-augmenting path in G/B, as desired. □

Lemma 9 inspires the following algorithm for solving the maximum perfect matching problem in non-bipartite graphs.

Algorithm 3 Edmonds' non-bipartite matching algorithm

```
 1: Initialize M = ∅.
 2: repeat
 3:     P = Search(G, M)        // Return augmenting path, or empty set if none exists.
 4:     M ← M ⊕ P
 5: until P = ∅

 6: procedure Search(G, M)
 7:     Build the graph H(G, M).
 8:     Search for a path P̂ from F to Γ(F) in H(G, M).
 9:     if no path found then
10:         Return P = ∅
11:     end if
12:     Post-process P̂, as in the proof of Lemma 8, to extract an augmenting path or flower.
13:     if augmenting path P is found then
14:         Return P
15:     else
16:         Let B be the blossom of the flower.
17:         Let P′ = Search(G/B, M/B)
18:         if P′ = ∅ then
19:             Return P = ∅
20:         else
21:             Transform P′ to an M-augmenting path P as in the proof of Lemma 9.
22:         end if
23:     end if
24: end procedure
```

The algorithm's outer loop (Lines 2 and 5) iterates at most $n/2$ times. In each iteration, we make a sequence of recursive calls to the SEARCH procedure, which searches for an augmenting path. Each recursive call involves shrinking a blossom, which reduces the number of vertices in the graph by at least 2. Hence, we call SEARCH at most $n/2$ times within each iteration of the outer loop. To assess the amount of work done in each call to SEARCH (excluding work done in recursive sub-calls to the same procedure) we start by observing that the graph $H(G, M)$ has n vertices and at most $2m$ edges, since the number of outgoing edges from a vertex in $H(G, M)$ is bounded above by the degree of that vertex in G. Hence, building and searching the graph $H(G, M)$ takes $O(m)$ time. (Assuming, as always, that G has no isolated vertices so that $n = O(m)$.) The remaining steps of SEARCH also take $O(m)$ steps, if not fewer, as can be seen by reviewing the proofs of Lemma 8 and Lemma 9. Hence, the overall running time of Edmonds' algorithm is bounded above by $O(mn^2)$.

3 Bipartite min-cost perfect matching

In the bipartite minimum-cost perfect matching problem, we are given an undirected bipartite graph $G = (L, R, E)$ 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.

3.1 LP relaxation

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 3. Is it possible that this fractional perfect matching achieves a lower cost than any perfect matching? No, because it can be expressed as a convex combination of perfect matchings (again, see Figure 3) 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.)

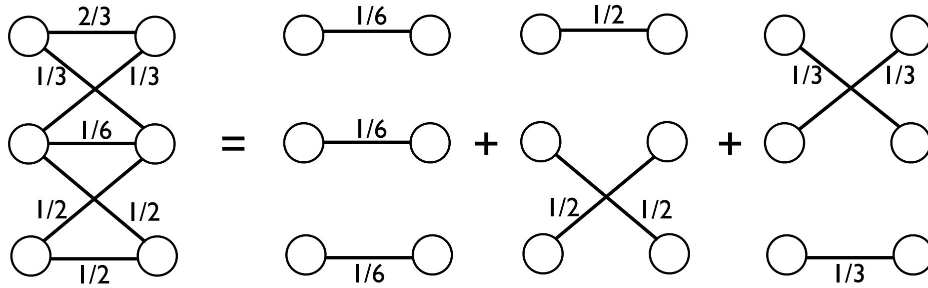

Figure 3: A fractional perfect matching.

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 4. Clearly, the

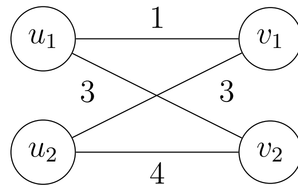

Figure 4: An instance of bipartite minimum cost perfect matching.

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 $w \in L \cup R$, 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

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

for some $u \in L$, or

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

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

$$\sum_{(u,v) \in L \times R} (y_u + y_v) x_{uv} = \sum_{u \in L} y_u + \sum_{v \in R} y_v. \tag{3}$$

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

$$\sum_{u,v} c(u, v) x_{uv} \geq \sum_{u \in L} y_u + \sum_{v \in R} y_v,$$

thus obtaining a lower bound on the cost of every fractional perfect matching. This technique works whenever the coefficients $(y_w)_{w \in L \cup R}$ satisfy $y_u + y_v \leq c(u, v)$ for every edge (u, v), regardless of whether the values y_u, y_v are positive or negative. To obtain the strongest possible lower bound using this technique, we would set the coefficients y_u, y_v by solving the following linear program.

$$\begin{array}{ll} \max & \sum_{w \in L \cup R} y_w \\ \text{s.t.} & y_u + y_v \leq c(u, v) \quad \forall u, v \end{array}$$

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.

3.2 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 y_u and y_v values) whose value (sum of y_u's and y_v's) equals the cost of the perfect matching. As the algorithm runs, it maintains a dual solution $\vec{y}$ and a matching M, and it preserves the following invariants:

1. Every edge (u, v) satisfies $y_u + y_v \leq c(u, v)$. If $y_u + y_v = c(u, v)$ 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

$$\sum_{(u,v)\in M} c(u, v) = \sum_{(u,v)\in M} y_u + y_v = \sum_{w\in L\cup R} y_w,$$

where the final equation holds because M is a perfect matching. The first invariant of the algorithm implies that $\vec{y}$ 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 $\vec{y} = 0$. 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 $u \in L$ that doesn't belong to any tight edges. Then we could raise y_u by some amount $\delta > 0$ until an edge containing u became tight. However, maybe every $u \in L$ belongs to a tight edge. In that case, we need to raise y_u by δ while lowering some other y_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

$$\delta = \min\{c(u,v) - y_u - y_v \mid u \in L \cap T, v \in R \setminus T\} \tag{4}$$

and adjust the dual variables by setting $y_u \leftarrow y_u + \delta, y_v \leftarrow y_v - \delta$ for all $u \in L \cap T, v \in R \cap T$. 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 $L \cap F$ of free vertices in L. We initialize $T = L \cap F$. 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 $L \cap T$ to $R \setminus T$. 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 L, 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 $y_{u,s}, y_{v,s}$ denote the values of the dual variables associated to vertices u, v

at the end of step s. Note that

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

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

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

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

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 $c(u, v)-y_{u,s}-y_{v,s}$ is equivalent to choosing the edge that minimizes $c(u, v) - y_{u,0} + \Delta_{s(u)} - y_{v,0}$. Let us maintain a priority queue containing all the edges from $L \cap T$ to $R \setminus T$. 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 $c(u, v) - y_{u,0} + \Delta_{s(u)} - y_{v,0}$, and this value never changes as the phase proceeds. Whenever the algorithm needs to choose the edge that minimizes $c(u, v)-y_{v,s}-y_{u,s}$, 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 y_u, y_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 $y_{u,s}$ or $y_{v,s}$ in constant time using (5)-(6). In particular, it takes $O(n)$ time to compute all the values y_u, y_v at the end of the phase, and it only takes $O(1)$ time to compute the value $\delta_s = c(u, v) - y_u - y_v$ once we have identified the edge $e = (u, v)$ using the priority queue. Thus, all the work to maintain the values y_u, y_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 $O(mn \log n)$.

4 Online matching

The study of *online algorithms* concerns problems in which information about the input is revealed over a sequence of time steps $t = 1, 2, \ldots$ and the algorithm must make decisions in each time step, without knowing what information will be revealed in future steps. In the online bipartite matching problem, there is a bipartite graph $G = (L, R, E)$ where L is known as the *offline side* and R is the *online side*. The contents of the set L are known to the algorithm at initialization time ($t = 0$), whereas the remaining information about G is revealed at times $t = 1, 2, \ldots, n = |R|$, by exposing one vertex of R at each time step. When vertex $j \in R$ arrives, all of its incident edges are revealed. The algorithm is then allowed

to take one of the following actions: select one of the edges (i, j) that was revealed in the current step; or do nothing. The set of selected edges is required to be a matching; thus, if vertex $i \in L$ belongs to a previously selected edge, then (i, j) may not be selected in the current time step. The algorithm's objective is to maximize the number of edges selected.

A variation of this problem is the *online bipartite fractional matching* problem, in which the input sequence is the same, but the algorithm's output at time j is a tuple of numbers $(x_{ij})_{i \in L}$ satisfying:

- $x_{ij} = 0$ when $(i, j) \notin E$.
- $\sum_{i \in L} x_{ij} \leq 1$.
- for all $i \in L$, $\sum_{j \in R} x_{ij} \leq 1$.

In other words, the matrix of values $(x_{ij})_{(i,j) \in L \times R}$ eventually computed by the algorithm must belong to the fractional matching polytope of G. Fractional matching is of interest as a problem in its own right, and also as a window into the design of randomized online bipartite matching algorithms. From any such randomized algorithm, one can define a corresponding deterministic online fractional bipartite matching algorithm, obtained by setting x_{ij} to be the unconditional probability that the online algorithm selects edge (i, j). (Note that this unconditional probability can be computed at the time when vertex j arrives — i.e., it does not depend on any information to be revealed in the future — which is the reason why the fractional matching algorithm is a valid online algorithm.) Note that there is no obvious way to invert this transformation; in other words, given a deterministic fractional online matching algorithm, there is no obvious way to obtain a randomized online matching algorithm whose expected behavior yields the designated fractional algorithm.

4.1 A lower bound

What should we hope to achieve in an online bipartite matching algorithm? If we are unreasonably optimistic, we might hope to design an algorithm that is guaranteed to output a maximum cardinality matching. The following example shows that this is hopeless. Suppose $L = \{i_1, i_2\}$ and $R = \{j_1, j_2\}$. Consider two possible input sequences. In both of them, vertex j_1 arrives at time $t = 1$ and reveals that it is connected to both i_1 and i_2. At time $t = 2$, vertex j_2 arrives and reveals that it has only one neighbor: in Input 1 this neighbor is i_1; in Input 2 it is i_2.

Notice that the maximum matching has size 2 in both of these inputs: j_2 can be matched to its only neighbor, whereas j_1 can be matched to the remaining element of L. Also notice that in both cases, this is the *unique* matching of size 2. Therefore, an online algorithm that seeks to select the maximum matching faces an insurmountable predicament: at time $t = 1$ it must match j_1 to one of its neighbors, there is a unique choice that is consistent with picking the maximum matching, and there is no way to know which choice this is until time $t = 2$. Thus, for every deterministic online algorithm, we can find an input instance that causes the algorithm to select a matching of size at most 1, while the maximum matching has size 2.

One can place this impossibility result in the broader context of *competitive analysis of online algorithms*, which evaluates algorithms according to the following criterion.

Definition 7. An online algorithm for a maximization problem is c-competitive if there exists a constant b such that for all input sequences,

$$c \cdot \mathsf{ALG} + b \geq \mathsf{OPT},$$

where ALG and OPT denote the values of the algorithm's solution and the optimum one, respectively. It is *strictly c-competitive* if $b = 0$ in the above bound. A randomized algorithm is c-competitive (against an oblivious adversary) if the above holds with $\mathbb{E}[\mathsf{ALG}]$ in place of ALG.

Our analysis of the two four-vertex input sequences above implies that deterministic online matching algorithms cannot be strictly c-competitive for any $c < 2$. By considering inputs comprising an arbitrarily long sequence of disjoint copies of either Input 1 or Input 2, we can eliminate the word "strictly" and conclude that deterministic algorithms cannot be c-competitive for any $c < 2$.

Our above discussion of the relationship between randomized and fractional algorithms shows that a lower bound on the competitive ratio of deterministic fractional online algorithms implies the same lower bound on the competitive ratio of randomized online algorithms. In particular, the competitive ratio of fractional (and hence randomized) online matching algorithms can be bounded below by 4/3, by an easy analysis of the same set of input sequences that furnished the lower bound of 2 for deterministic algorithms.

4.2 The greedy algorithm

It turns out that the example presented in Section 4.1 is the worst possible for deterministic algorithms, from the standpoint of competitive analysis. There is a strictly 2-competitive deterministic online algorithm. In fact, a competitive ratio of 2 is achieved by the most naïve algorithm: the greedy algorithm that matches each new vertex j to an arbitrary unmatched neighbor, i, whenever an unmatched neighbor exists.

Exercise 2. Prove that the greedy algorithm for online bipartite matching is strictly 2-competitive.

4.3 Online fractional matching: the waterfilling algorithm

It turns out that online fractional matching algorithms can achieve competitive ratios significantly better than 2, as we will see in this section.

First, a useful bit of terminology: we will refer to the sum $\sum_{j \in R} x_{ij}$ as the *fractional degree* of vertex i in fractional matching x. For a vertex $j \in R$ the fractional degree is defined similarly.

Perhaps the most natural idea for online fractional matching is to have each vertex j balance load equally among its neighbors. In other words, if a new vertex j arrives and has

d neighbors, then for each neighbor i we set the value of x_{ij} to be $1/d$, unless that would violate the degree constraint of vertex i (the constraint that $\sum_j x_{ij} \leq 1$) in which case we merely increase x_{ij} as much as possible given the degree constraint.

However, this "stateless balancing" algorithm fails to be better than 2-competitive. To construct a counterexample, we take the example form Section 4.1 and blow up each vertex into n vertices, carefully modifying the edge set to cause the algorithm to make catastrophic decisions. The set L now has $2n$ vertices, which we will label as $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n$, and the set R has $2n$ vertices labeled $c_1, c_2, \ldots, c_n, d_1, d_2, \ldots, d_n$. Each vertex c_j has $n+1$ neighbors: it is connected to a_j and also to $b_1, b_2, \ldots, b_n$. Each vertex d_j has only one neighbor, namely b_j. The maximum matching in this graph has size $2n$: it matches (a_i, c_i) and (b_i, d_i) for $i = 1, \ldots, n$. If the vertices $c_1, \ldots, c_n, d_1, \ldots, d_n$ arrive in that order, the stateless balancing algorithm will first assign a value of $\frac{1}{n+1}$ to each edge incident to $c_1, \ldots, c_n$. Thus, when $d_1, \ldots, d_n$ start arriving, each of them has a unique neighbor and the fractional degree of that neighbor is already $\frac{n}{n+1}$, so d_j can contribute only $\frac{1}{n+1}$ additional units to the size of the fractional matching. Thus, when the algorithm is finished processing the entire graph, the total size of its fractional matching is $n + \frac{n}{n+1}$, only slightly more than half of the optimum.

What went wrong in this algorithm? The vertices $b_1, \ldots, b_n$ are more "highly demanded" than $a_1, \ldots, a_n$ and it was unwise for vertices $c_1, \ldots, c_n$ to use up almost all of the capacity of $b_1, \ldots, b_n$ while using almost none of $a_1, \ldots, a_n$. The first vertex, c_1, can be forgiven for making this mistake since all of its neighbors looked indistinguishable when it arrived. But later on, we should have known better: we had already seen that the capacities of $b_1, \ldots, b_n$ were being depleted and should have taken measures to conserve that capacity. In short, there was nothing evidently wrong with the load-balancing idea, but it was silly to do *stateless* load-balancing; instead, we should have kept track of the current state (the amount of load already placed on each vertex in L) and adjusted our load-balancing decisions to correct for imbalances in the current load vector.

This bring us to the waterfilling algorithm. It keeps track of a "water level" for each $i \in L$ representing the current fractional degree $d(i) = \sum_j x_{ij}$, summing over all vertices $j \in R$ that have arrived in the past. When a new vertex j arrives, it allocates its one unit of fractional degree among its neighbors by finding the neighbors with the lowest water level and continuously raising their water level until either one unit of water has been poured into the graph, or the water level of all neighbors reaches 1, whichever comes first. In less metaphorical terms, the algorithm finds the unique number $\hat{\ell}(j)$ such that

$$\sum_{i \in N(j)} \max\{\hat{\ell}(j), d(i)\} = 1 + \sum_{i \in N(j)} d(i),$$

where $N(j)$ represents the set of all neighbors of j. It then sets

$$\begin{aligned} \ell(j) &= \min\{\hat{\ell}(j), 1\} \\ x_{ij} &= \max\{\ell(j), d(i)\} - d(i) \quad \forall (i,j) \in E \end{aligned}$$

and it updates $d(i)$ to $d(i) + x_{ij}$ for all i.

We will analyze the waterfilling algorithm using the primal-dual method. This means that we'll use the fractional matching LP

$$\begin{array}{lll} \max & \sum_{i,j} x_{ij} & \\ \text{s.t.} & \sum_j x_{ij} \le 1 & \forall i \\ & \sum_i x_{ij} \le 1 & \forall j \\ & x_{ij} \ge 0 & \forall i,j \end{array}$$

and its dual

$$\begin{array}{lll} \min & \sum_i \alpha_i + \sum_j \beta_j & \\ \text{s.t.} & \alpha_i + \beta_j \ge 1 & \forall (i,j) \in E \\ \alpha_i, \beta_j \ge 0 & \forall i,j & \end{array}$$

In particular, we define a dual solution $(\alpha_i)_{i \in L}$, $(\beta_j)_{j \in R}$ by specifying that

$$\alpha_i = g(d(i)) \quad \forall i \tag{7}$$

$$\beta_j = 1 - g(\ell(j)) \quad \forall j, \tag{8}$$

where

$$g(y) = \frac{e^y - 1}{e - 1}.$$

The choice of this specific function g will make more sense later in the analysis. The vital properties of g that are needed in the analysis are:

1. g is an increasing function.
2. $g(0) = 0$
3. $g(1) = 1$
4. $1 - g(t) + g'(t) = \frac{e}{e-1}$ for all t.

First, let's observe that the dual solution defined by (7)-(8) is feasible. This is because at the time we finish processing vertex j, the inequality $d(i) \ge \ell(j)$ is satisfied by all neighboring vertices i. Since the value $d(i)$ will not subsequently decrease, we also have $d(i) \ge \ell(j)$ at termination. Furthermore, since g is an increasing function, we have

$$\alpha_i + \beta_j = g(d(i)) + 1 - g(\ell(j)) \ge g(\ell(j)) + 1 - g(\ell(j)) = 1,$$

which verifies dual feasibility.

We claim that the fractional matching and the dual solution computed by our algorithm satisfy

$$\frac{e}{e-1} \sum_{(i,j) \in E} x_{ij} \ge \sum_{i \in L} \alpha_i + \sum_{j \in R} \beta_j. \tag{9}$$

By the weak duality, the sum on the right side is an upper bound on the size of any fractional matching in G, and therefore (9) implies that the waterfilling algorithm is $\left(\frac{e}{e-1}\right)$-competitive.

To prove (9), we compare β_j with a parameter β'_j defined as follows. For $t \in [0,1]$ let $n_j(t)$ denote the number of edges $(i,j) \in E$ such that the inequality $d(i) \le t$ held at the time when j arrived. Note that

$$\int_0^{\ell(j)} n_j(t)\, dt = 1$$

provided that $\ell(j) < 1$, because in that case vertex j contributed one unit of "water" and the integrand denotes the rate at which water was filling the system as we increased the water level ℓ from t to $t + dt$. Now, define

$$\beta'_j = \int_0^{\ell(j)} (1 - g(t)) \cdot n_j(t)\, dt.$$

The inequality $\beta'_j \geq \beta_j$ always holds: when $\ell(j) = 1$ this is because $\beta_j = 0$, and when $\ell(j) < 1$ it is because $1 - g(t)$ is a decreasing function of t and therefore

$$\int_0^{\ell(j)} (1 - g(t)) \cdot n_j(t)\, dt > (1 - g(\ell(j)) \cdot \int_0^{\ell(j)} n_j(t)\, dt = 1 - g(\ell(j)) = \beta_j.$$

Letting $d(i)$ denote the degree of a vertex $i \in L$ before the arrival of vertex j, the amount by which the dual objective increases when processing j is:

$$\begin{aligned}
\beta_j + \sum_{i \in N(j)} [g(\ell(j)) - g(d(i))] &= 1 - g(\ell(j)) + \sum_{i \in N(j)} \int_{d(i)}^{\ell(j)} g'(t)\, dt \\
&= 1 - g(\ell(j)) + \int_0^{\ell(j)} g'(t) \cdot n_j(t)\, dt \\
&\leq \int_0^{\ell(j)} [1 - g(t) + g'(t)] \cdot n_j(t)\, dt \\
&= \frac{e}{e-1} \int_0^{\ell(j)} n_j(t)\, dt \\
&= \frac{e}{e-1} \sum_{i \in N(j)} x_{ij},
\end{aligned}$$

hence the increase in the dual objective is at most $\frac{e}{e-1}$ times the increase in the primal objective. Since the primal and dual objectives both start out at zero, this means that the dual objective at termination is at most $\frac{e}{e-1}$ times the primal objective, certifying inequality (9) and completing the proof that the waterfilling algorithm is $\left(\frac{e}{e-1}\right)$-competitive.

4.4 The waterfilling algorithm is optimal

It turns out that $\frac{e}{e-1}$ is precisely the best competitive ratio that can be achieved by an online fractional matching algorithm. To prove this, we consider an arbitrary fractional matching algorithm ALG and evaluate its performance on a random input sequence generated as follows. The graph G has vertex sets $L = R = [n] = \{1, \ldots, n\}$. We sample a uniformly random permutation π of the set $[n]$, and we define the edge set of the graph to be

$$E = \{(\pi(i), j) \mid i \geq j\}.$$

The elements of R arrive in the order $j = 1, 2, \ldots, n$.

Observe first that there is always a perfect matching in G, consisting of the edges $(\pi(j), j)$ for $j = 1, \ldots, n$. In fact, this is the unique perfect matching in G: one can easily show that

every perfect matching must contain the edge $(\pi(j), j)$ for all $j \in [n]$, by downward induction on j starting from $j = n$.

To place an upper bound on the expected size of the matching produced by ALG, we argue as follows. The expected value of $x_{\pi(i),j}$ is zero if $i < j$, and it is at most $\frac{1}{n+1-j}$ if $i \geq j$. To see this latter fact, note that for any two elements $i, k \in \{j, j+1, \dots, n\}$, we have $\mathbb{E}[x_{\pi(i),j}] = \mathbb{E}[x_{\pi(k),j}]$ by symmetry, since the subgraph of G consisting of all edges observed up until time j has an automorphism that exchanges i and k. Since $x_{\pi(j),j} = x_{\pi(j+1),j} = \dots = x_{\pi(n),j}$ and the sum of these numbers is at most 1, each of them is at most $\frac{1}{n+1-j}$.

Now, let $k = n - \lceil n/e \rceil$, and observe that $\sum_{j=1}^{k} \frac{1}{n+1-j}$ is between $1 - \frac{5}{n}$ and 1. This is proven by the integral test:

$$\sum_{j=1}^{k} \frac{1}{n+1-j} < \int_{n/e}^{n} \frac{dx}{x} = 1$$

while

$$\frac{5}{n} + \sum_{j=1}^{k} \frac{1}{n+1-j} > \frac{1}{n+5} + \frac{1}{n+5} + \dots + \frac{1}{n+1} + \sum_{j=1}^{k} \frac{1}{n+1-j} > \int_{(n+6)/e}^{n+6} \frac{dx}{x} = 1.$$

The expected size of the fractional matching produced by ALG is bounded above by:

$$\begin{aligned}
\sum_{i=1}^{n} \mathbb{E}\left[\sum_{j=1}^{i} x_{\pi(i),j}\right] &\leq \sum_{i=1}^{k}\sum_{j=1}^{i} \frac{1}{n+1-j} + \sum_{i=k+1}^{n} 1 \\
&< \sum_{i=1}^{k}\sum_{j=1}^{i} \frac{1}{n+1-j} + \sum_{i=k+1}^{n} \left[\frac{5}{n} + \sum_{j=1}^{k} \frac{1}{n+1-j}\right] \\
&< 5 + \sum_{j=1}^{k} \frac{(k+1-j) + (n-k)}{n+1-j} \\
&= 5 + k < 5 + \left(1 - \frac{1}{e}\right) n.
\end{aligned}$$

As the expected size of the maximum matching is n, and the expected size of the fractional matching produced by ALG is bounded above by $5 + \left(\frac{e-1}{e}\right) n$, we see that ALG cannot be c-competitive for any $c < \frac{e}{e-1}$.

4.5 Randomized online matching: The RANKING algorithm

(Most of this section is an excerpt from the paper "Randomized Primal-Dual Analysis of RANKING *for Online Bipartite Matching" by N. Devanur, K. Jain, and R. Kleinberg, 2012.)*

Given an online fractional matching algorithm, it is tempting to try constructing a randomized online matching algorithm whose probability of choosing edge (i, j) is equal to the value x_{ij} computed by the fractional matching algorithm. If such a transformation were

possible, it would yield a randomized online matching algorithm whose competitive ratio is exactly the same as that of the given fractional matching algorithm. Unfortunately, such a transformation is not possible in general. (For example, there is no randomized matching algorithm whose probability of selecting each edge (i, j) is exactly equal to the value assigned to that edge by the waterfilling algorithm. It is quite instructive to try proving this.)

However, there *is* a randomized online matching algorithm, known as RANKING, that achieves exactly the same competitive ratio as the waterfilling algorithm, namely $\frac{e}{e-1}$. Since the existence of a c-competitive randomized online matching algorithm implies the existence of a c-competitive online fractional matching algorithm, we can deduce that RANKING achieves the best possible competitive ratio for randomized online matching algorithms.

The RANKING algorithm is actually very intuitive: at initialization time, it samples a uniformly random total ordering of the vertices in L. Subsequently, as each vertex $j \in R$ arrives, if j has an unmatched neighbor in L then we choose the unmatched neighbor i that comes earliest in the random ordering, and we add (i, j) to the matching.

To analyze the RANKING algorithm, we begin with a reinterpretation of the algorithm in a way that is conducive to our analysis. Instead of picking a random total ordering of the vertices in L, each vertex in L picks a random number in $[0, 1]$ and a vertex $j \in R$, upon its arrival, is assigned to the unmatched neighbor who picked the lowest number. The algorithm is presented as Algorithm 4 below.

Algorithm 4 The RANKING algorithm.

1: **for all** $i \in L$ **do**
2:     Pick $Y_i \in [0, 1]$ uniformly at random
3: **end for**
4: **for all** $j \in R$ **do**
5:     When j arrives, let $N(j)$ denote the set of unmatched neighbors of j
6:     **if** $N(j) = \emptyset$ **then**
7:         j remains unmatched
8:     **else**
9:         Match j to $\arg\min\{Y_i : i \in N(j)\}$
10:     **end if**
11: **end for**

To analyze the algorithm, we note the standard LP relaxation for matching and its dual.

$$\begin{aligned} &\text{maximize} \sum_{(i,j) \in E} x_{ij} \text{ s.t.} \\ &\forall\, i \in V, \sum_{j:(i,j) \in E} x_{ij} \le 1. \\ &\forall\, (i,j) \in E, x_{ij} \ge 0. \end{aligned}$$

$$\begin{aligned} &\text{minimize} \sum_{i \in L} \alpha_i + \sum_{j \in R} \beta_j \text{ s.t.} \\ &\forall\, (i,j) \in E, \alpha_i + \beta_j \ge 1. \\ &\forall\, i, j, \alpha_i, \beta_j \ge 0. \end{aligned}$$

Our analysis constructs a dual solution which is also randomized. The dual variables we construct may not always be feasible; in other words, they may violate the constraint

$\alpha_i + \beta_j \geq 1$ for some edges (i, j). However, the *expected values* of the dual variables will constitute a feasible dual solution. The competitive ratio of $\frac{e}{e-1}$ will follow from the fact that the value of the dual solution is always $\frac{e}{e-1}$ times the size of the matching found, and that the expectation of the dual variables constitutes a feasible dual solution (whose value, of course, is also $\frac{e}{e-1}$ times the expected size of the matching found).

Our construction of the duals depends on a monotone non-decreasing function h that is closely related to the function g that came up in the analysis of the waterfilling algorithm in Section 4.3. The formula for h is $h(y) = e^{y-1}$ and its relevant properties are:

1. h is an increasing function;
2. $h(1) = 1$;
3. $\forall \theta \in [0, 1] \quad \int_0^\theta h(y)\, dy \; + 1 - h(\theta) = \frac{e-1}{e}$.

Note the similarity between the integral equation in property 3 of h and the differential equation in property 4 of the function g in Section 4.3; note also, however, that if we differentiate both sides of the integral equation defining h we certainly don't get the differential equation defining g.

Whenever i is matched to j, let

$$\alpha_i = \tfrac{e}{e-1} \cdot h(Y_i), \qquad \beta_j = \tfrac{e}{e-1} \cdot (1 - h(Y_i)).$$

For all unmatched i and j, set $\alpha_i = \beta_j = 0$. It will be useful to interpret the algorithm as follows: on matching i to j, we generate a value of 1 for the primal, which translates to a value of $\frac{e}{e-1}$ for the dual. Each unmatched vertex $i \in L$ that is a neighbor of j offers $\frac{e}{e-1} \cdot (1 - h(Y_i))$ of this value to j (to be assigned to β_j), while keeping the rest to itself (to be assigned to α_i). Then j is matched to the vertex that makes the highest offer.

Before we show that the expectation of the duals is feasible, we need certain properties of the algorithm specified by the following two lemmas. Let $(i, j) \in E$ be any edge in the graph. Consider an instance of the algorithm on $G \setminus \{i\}$, with the same choice of $Y_{i'}$ for all other $i' \in L$. Let y^c be the value of $Y_{i'}$ for the i' that is matched to j. Define y^c to be 1 if j is not matched. Let β_j^c be the value of β_j in this run, i.e. $\beta_j^c = \frac{e}{e-1} \cdot (1 - h(y^c))$.

Lemma 10 (Dominance Lemma)**.** *Given $Y_{i'}$ for all other $i' \in L$, i gets matched if $Y_i < y^c$.*

Proof. Suppose i is not matched when j arrives. This means that the run of the algorithm until then is identical to the run without i. From the definition of y^c, in the run without i, j is matched to i' such that $Y_{i'} = y^c$. Since $Y_i < y^c$, j is matched to i. □

Lemma 11 (Monotonicity Lemma)**.** *Given $Y_{i'}$ for all other $i' \in L$, for all choices of Y_i, $\beta_j \geq \beta_j^c$.*

Proof. Consider executing the algorithm on graphs G and $G \setminus \{i\}$ in parallel. At the start of every step of the two parallel executions, the unmatched vertices in L for the G execution constitute a superset of the unmatched vertices in L for the $G \setminus \{i\}$ execution. This statement is easily proven by induction: given that it holds at the start of one step, the only way it could be violated at the start of the next step is if the G execution chooses a vertex $i' \in L$ that is also unmatched, but is not chosen, in the $G \setminus \{i\}$ execution. Instead the $G \setminus \{i\}$

execution must choose some other vertex i'' such that $Y_{i''} < Y_{i'}$. By our induction hypothesis i'' was also unmatched in the G execution, contradicting the fact that the algorithm chose i' instead.

When node j arrives, its unmatched neighbors in the G execution form a superset of its unmatched neighbors in the $G \setminus \{i\}$ execution, so in the both executions j has an unmatched neighbor whose Y-value is y^c. If the algorithm instead chooses another neighbor of j, its Y-value can be at most y^c and hence, by the monotonicity of h, we have $\beta_j \geq \beta_j^c$. □

We now show that the above properties of h imply a competitive ratio of $\frac{e}{e-1}$ for RANKING.

Lemma 12. RANKING *is* $\frac{e}{e-1}$*-competitive.*

Proof. Whenever i is matched to j, $\alpha_i + \beta_j = \frac{e}{e-1}$. Therefore the ratio of the dual solution to the primal is always $\frac{e}{e-1}$. We show that the dual is feasible in expectation. In particular, we show that for all $(i, j) \in E$,

$$\mathbb{E}_{Y_i}[\alpha_i + \beta_j] \geq 1$$

for all choices of $Y_{i'}$ for all $i' \neq i \in L$. By the Dominance Lemma (Lemma 10) i is matched whenever $Y_i \leq y^c$. Hence

$$\mathbb{E}_{Y_i}[\alpha_i] \geq \frac{e}{e-1} \int_0^{y^c} h(y)\, dy.$$

By the Monotonicity Lemma (Lemma 11), $\beta_j \geq \beta_j^c = \frac{e}{e-1} \cdot (1 - h(y^c))$ for all choices of Y_i. The lemma now follows from the integral equation listed above as property 3 of h. □