Cornell University, Fall 2023	CS 6820: Algorithms
Lecture notes: Matchings	21 Aug – 1 Sep 2023

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 $M = \emptyset$.
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 $G = (L, R, E)$ 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 aspect 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.

In this section we will adopt the notations of the previous section: $G = (L, R, E)$ is a bipartite graph, M is a matching, F is the set of free vertices with respect to M, and G_M is the directed graph constructed from G and M by orienting edges of M from R to L and all other edges of G from L to R. We will refer to this directed graph henceforth as the *residual graph* of M in G.

When one performs breadth-first search starting from the vertex set $L \cap F$ in G_M, the search algorithm discovers, for each vertex $v \in L \cup R$, the length of the shortest directed path in G_M from $L \cap F$ to v. Denote this shortest path length by $d(v)$. (If there is no directed path in G_M from $L \cap F$ to v, then $d(v) = \infty$.)

Edges of G_M can be divided into two types: if $e = (u, v)$ satisfies $d(v) > d(u)$ we call e an *advancing edge*, otherwise we call it a *retreating edge*. Note that an advancing edge (u, v) must satisfy $d(v) = d(u) + 1$; the relation $d(v) > d(u) + 1$ is not possible because the shortest path from $L \cap F$ to v cannot be strictly longer than the path formed by appending edge (u, v) to the end of a shortest path from $L \cap F$ to u.

We are now ready to define the type of structure that the Hopcroft-Karp algorithm searches for in each of its iterations.

Definition 5. If G is a graph and M is a matching, a *blocking set of augmenting paths* with respect to M is a maximal set of vertex-disjoint advancing M-augmenting paths.

In the definition, the term "advancing M-augmenting path" means an M-augmenting path composed entirely of edges that are advancing in G_M. It is important that we define a blocking set to be a *maximal* set of such paths (meaning that the blocking set is not a proper subset of any other collection of vertex-disjoint advancing M-augmenting paths) rather than a *maximum* set of vertex-disjoint advancing M-augmenting paths (which would mean that there is no other collection of vertex-disjoint advancing M-augmenting paths containing strictly more paths than the blocking set). Weakening the definition from "maximum" to "maximal" allows us to design a linear-time algorithm to find a blocking set of augmenting paths with respect to any matching M, as we now explain.

1.4.1 Computing a blocking set of augmenting paths

The algorithm to compute a blocking set of augmenting paths consists of two phases: a breadth-first search of G_M to locate the set of advancing edges, and then a depth-first search on the graph H formed by the advancing edges. As advancing augmenting paths are discovered during the depth-first search they are placed into a set, B, which is a blocking set when the algorithm terminates.

Algorithm 2 Computing a blocking set of augmenting paths

Given: bipartite graph $G = (L, R, E)$, matching M
Form the set of free vertices, F, and the residual graph, G_M.
Perform breadth-first search in G_M starting from $L \cap F$ to find the advancing edges.
Initialize graph H with $V(H) = L \cup R$ and $E(H) = \{\text{advancing edges in } G_M\}$.
Initialize empty augmenting path set, B.
Initialize S to be an empty stack.
while $L \cap F \cap V(H)$ is non-empty **do**
    if S is empty **then**
        Push any element of $L \cap F \cap V(H)$ onto S.
    else
        Let u be the vertex on top of S.
        if $u \in R \cap F$ **then**
            Let P be the path formed by the vertices of S, ordered from bottom to top.
            Insert P into blocking set B.
            Delete the vertices of P and all of their incident edges from H.
            Reinitialize S to be an empty stack.
        else if $E(H)$ contains an edge (u, v) leaving u **then**
            Push v on top of S.
        else
            Delete u and all of its incident edges from H.
            Pop u from S.
        end if
    end if
end while
Output B.

Before proving the correctness of the algorithm, let's analyze its running time. Constructing G_M, performing breadth-first search on it, and constructing H, all take linear time, $O(m + n)$. Assuming as before that G has no isolated vertices, so that $n \leq 2m$, we can write this running time bound as $O(m)$. To bound the running time of the while loop in Algorithm 2, we make the following observations.

1. As the while-loop runs, it sometimes deletes vertices and edges from H. Once deleted, these vertices and edges are never re-inserted. Therefore, the total running time devoted to deleting them is $O(m + n) = O(m)$. Henceforward, we will ignore time spent deleting vertices and edges.

2. In a while-loop iteration that inserts a new path into B, the running time (ignoring time spent deleting edges) is proportional to the length of the path. The paths in B are all vertex-disjoint, so their combined length is at most n, hence the combined running time of these loop iterations is $O(n)$.

3. Every other while-loop iteration either pushes a vertex onto the stack or pops it from the top of the stack. Ignoring the edge deletions that occur when a vertex is popped,

these while-loop iterations have constant running time. The number of such iterations is $O(n)$ because each vertex is popped from the stack at most once, hence it is also pushed into the stack at most once.

Summing up all the components of the running time accounted above, the total running time of the blocking set computation is $O(m)$ as claimed.

The correctness of the algorithm is proven inductively. The inductive hypothesis is the conjunction of three assertions that hold at the start and end of each iteration of the main loop.

1. The contents of the stack S constitute a path in H starting from $L \cap F$.
2. The contents of the set B constitute vertex-disjoint advancing M-augmenting paths.
3. If P is a path in G_M composed of advancing edges and ending at a vertex in $R \cap F$, then either P is contained in H or P has a vertex in common with one of the paths in B.

At the start of the main loop, S and B are empty and H contains all of the advancing edges in G_M, so all three properties hold at that time. The first property is always satisfied because the algorithm only pushes elements of $L \cap F$ to the bottom of an empty stack, and it only pushes a vertex v onto a non-empty stack when H contains an edge from the vertex at the top of the stack into v. To verify that the second property is a loop invariant, observe that the paths in B must be vertex-disjoint because whenever a path is added to B its vertices are immediately and permanently deleted from H, so no path containing any of those vertices will later be added to B. Furthermore, a path is only added to B when it starts in $L \cap F$, ends in $R \cap F$, and is composed of edges of H. All the edges of of H are advancing edges in G_M, so the paths that are added to B are advancing paths in G_M, i.e. advancing M-augmenting paths. Finally, to see that the third property is a loop invariant, consider the loop iteration in which P ceases to be contained in H. Edges of H are only deleted when one of their endpoints is deleted, so when P ceases to be contained in H one of its vertices must be deleted. If this vertex belongs to a path being inserted into B, then the third invariant is preserved. The only other case in which the algorithm deletes a vertex from H is when the vertex is at the top of the stack, does not belong to $R \cap F$, and has no outgoing edges remaining in H. However, the final vertex of P belongs to $R \cap F$ and all other vertices of P have at least one outgoing edge remaining in H (recall that we are considering the last moment at which P is contained in H). Hence, this case cannot apply to P, so we have verified that the third invariant is preserved by each loop iteration.

From the second and third invariants, it is clear that the output of the algorithm is a blocking set of M-augmenting paths. The second invariant guarantees that the contents of B are a set of vertex-disjoint advancing M-augmenting paths, and the third invariant guarantees that this set is maximal because if there is another advancing M-augmenting path disjoint from those in B, then that path must be contained in H. In particular, the first vertex of the path must belong to $L \cap F \cap V(H)$, so the while loop would not have terminated.

1.4.2 The Hopcroft-Karp Algorithm and its Analysis

As indicated earlier, the Hopcroft-Karp Algorithm improves the naïve algorithm for bipartite maximum matching, by augmenting an entire blocking set of augmenting paths in each iteration instead of using just one augmenting path. The correctness of the algorithm follows, as before, from the fact that its termination condition guarantees that it outputs a matching that has no augmenting paths, and every such matching has maximum cardinality.

Algorithm 3 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

We have seen in Section 1.4.1 that each loop iteration takes $O(m)$ time, so the analysis of the running time boils down to proving an upper bound on the number of blocking-set computations the algorithm must perform. To prove the upper bound we will make use of two different measures of progress: the number of edges in the matching, and the length of the shortest M-augmenting path. In the following series of lemmas we will show that both of these parameters strictly increase in each iteration of the Hopcroft-Karp Algorithm.

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. □

Lemma 4 guarantees that the cardinality of the matching M strictly increases with each iteration of the Hopcroft-Karp Algorithm. Below, in Lemma 6 we establish the validity of

the other progress measure, namely the length of the shortest M-augmenting path. Finally, in Lemma 7 we combine the two progress measures to prove an upper bound on the total number of blocking set computations.

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.

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

In the edge set of $G_{M'}$, the orientation of every edge in Q is reversed and the orientation of every edge in $\overline{Q}$ 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 $\overline{Q}$, which satisfy $d(y) = d(x) + 1$;
3. retreating edges of $\overline{Q}$, 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 strictly more than 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. Above, when we enumerated the three types of edges in $G_{M'}$, we saw that the only type satisfying $d(y) = d(x) + 1$ are the advancing edges of $\overline{Q}$, so if the path P has length ℓ then it must be made up entirely of advancing edges in $\overline{Q}$. In particular this implies P is edge-disjoint from Q. The path P 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. Since Q is the set of augmenting paths that was used to transform M into M', every vertex of Q belongs to an edge of $Q \cap M'$. Let e be the edge in $Q \cap M'$ that contains w. Edge e must belong to P because any M'-augmenting path must contain the

edges of M' incident to all of its interior nodes. Hence, we have established that e belongs to the edge sets of both P and 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. □

Since each iteration of the Hopcroft-Karp algorithm takes time $O(m)$, and there are fewer than $2\sqrt{n}$ iterations, the total running time is bounded above by $O(m\sqrt{n})$.

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 $u_r, \ldots, u_s$.

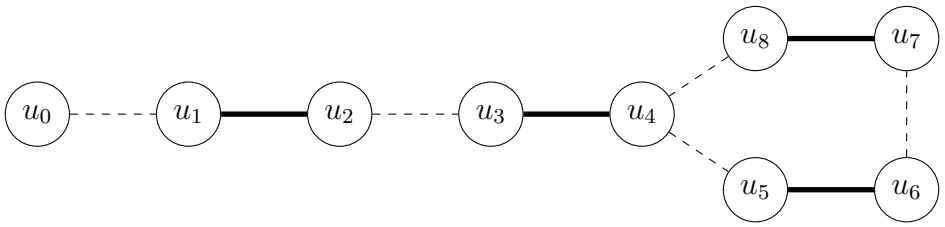

Figure 1: A flower

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.

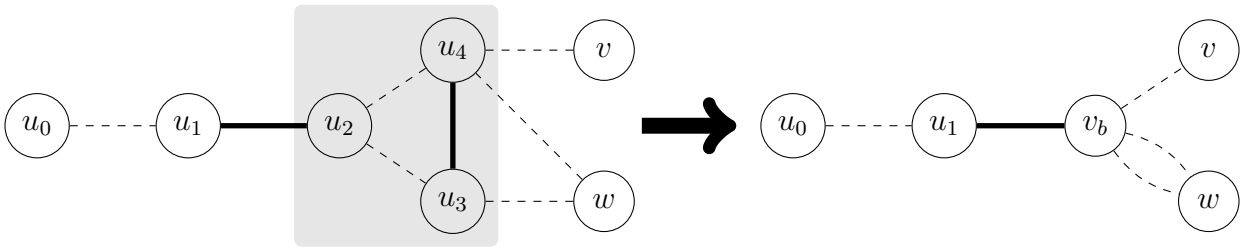

Figure 2: Shrinking a blossom

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.

$$
\begin{aligned}
G \text{ has an } M\text{-augmenting path} &\Rightarrow M \text{ is not a maximum matching in } G && \textit{(Lemma 2)}\\
&\Rightarrow M' \text{ is not a maximum matching in } G && \textit{($|M| = |M'|$)}\\
&\Rightarrow G \text{ has an } M'\text{-augmenting path} && \textit{(Lemma 2)}\\
&\Rightarrow G/B \text{ has an } (M'/B)\text{-augmenting path} && \textit{(proven below)}\\
&\Rightarrow M'/B \text{ is not a maximum matching in } G/B && \textit{(Lemma 2)}\\
&\Rightarrow M/B \text{ is not a maximum matching in } G/B && \textit{($|M/B| = |M'/B|$)}\\
&\Rightarrow G/B \text{ has an } (M/B)\text{-augmenting path} && \textit{(Lemma 2)}
\end{aligned}
$$

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 4 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 $e \in E$. Assume $|L| = |R|$. Let $c(u, v) = c_e$ if $e = (u, v)$ is an edge of G, and $c(u, v) = \infty$ otherwise. From now on, we will consider G to be a complete bipartite graph, with some edges having infinite cost. As always, let n denote the number of vertices and m the number of finite-cost edges of G.

When S is a set of edges we will write $c(S)$ to denote $\sum_{e \in S} c_e$. The bipartite min-cost perfect matching problem is to find a perfect matching M that minimizes $c(M)$.

3.1 Iterative min-cost augmenting paths

Noting the success of augmenting-path methods at solving the maximum-cardinality bipartite matching problem, it is logical to expect that they have a role to play in solving the minimum-cost perfect matching problem as well. To begin with, we seek to understand how augmenting paths affect the cost of a matching.

Lemma 10. *If M is a matching and S is an edge set such that $M \oplus S$ is also a matching, then*

$$c(M \oplus S) = c(M) + c(S \setminus M) - c(S \cap M) \tag{1}$$

Proof. The lemma follows immediately from the set-theoretic relation

$$M \oplus S = (M \setminus (S \cap M)) \cup (S \setminus M)$$

and the observation that the two sets on the right side are disjoint. □

In light of Lemma 10 we define the *incremental cost* of S relative to M as follows:

$$\Delta c(S; M) = c(S \setminus M) - c(S \cap M). \tag{2}$$

The following greedy algorithm starts with an empty matching and iteratively transforms it into a perfect matching using the augmenting path of least incremental cost.

Algorithm 5 Greedy algorithm for minimium cost bipartite perfect matching.

```
1: Initialize M = ∅.
2: while M is not a perfect matching do
3:     Find an M-augmenting path P that minimizes Δc(P; M).
4:     M ← M ⊕ P
5: end while
6: Output M.
```

Proving the correctness of this algorithm is surprisingly tricky. The proof is by induction on the number of loop iterations; the induction hypothesis is that the matching produced after k iterations has the minimum cost among all matchings of size k. The base case $k = 0$ is obvious, and the induction step is encapsulated in the following lemma.

Lemma 11. *If M_k is a minimum-cost matching of size k in G, and P is an M_k-augmenting path of minimum incremental cost, then $M_k \oplus P$ is a minimum-cost matching of size $k+1$.*

Proof. Let M_{k+1} denote a minimum-cost matching of size $k+1$. The inequality $c(M_k \oplus P) \geq c(M_{k+1})$ is obvious because $M_k \oplus P$ is a matching of size $k+1$. In the remainder of the proof we focus on proving the reverse inequality.

The symmetric difference $M_k \oplus M_{k+1}$ is made up of connected components, at least one of which is an M_k-augmenting path. Denote this path by Q and let R be the union of the remaining connected components of $M_k \oplus M_{k+1}$. Note that R contains an equal number of edges in M_k and in M_{k+1} (because it total, the number of M_{k+1} edges in the symmetric difference $M_k \oplus M_{k+1}$ exceeds the number of M_k edges by 1, and the path Q already accounts for this excess) and that every component of R is alternating with respect to both M_k and M_{k+1}. Therefore $R \oplus M_k$ is a k-edge matching and $R \oplus M_{k+1}$ is a $(k+1)$-edge matching. Since M_k and M_{k+1} are minimum-cost matchings for their respective cardinalities,

$$\begin{aligned} c(M_k) \leq c(R \oplus M_k) &= c(M_k) + \Delta c(R; M_k) \\ c(M_{k+1}) \leq c(R \oplus M_{k+1}) &= c(M_{k+1}) + \Delta c(R; M_{k+1}). \end{aligned}$$

Therefore, $\Delta c(R; M_k)$ and $\Delta c(R; M_{k+1})$ are both non-negative. On the other hand, they sum to zero because

$$\Delta c(R; M_k) + \Delta c(R; M_{k+1}) = [c(R \cap M_{k+1}) - c(R \cap M_k)] + [c(R \cap M_k) - c(R \cap M_{k+1}] = 0.$$

Hence, $\Delta c(R; M_k)$ and $\Delta c(R; M_{k+1})$ are both equal to zero.

Recalling that P is an M_k-augmenting path of minimum incremental cost, we have the inequality $\Delta c(P; M_k) \leq \Delta c(Q; M_k)$. Combining this with the equation $\Delta c(R; M_k) = 0$, we find that

$$\begin{aligned} c(M_k \oplus P) &= c(M_k) + \Delta c(P; M_k) \\ &\leq c(M_k) + \Delta c(Q; M_k) \\ &= c(M_k) + \Delta c(Q; M_k) + \Delta(R; M_k) \\ &= c(M_k) + \Delta c(Q \cup R; M_k) \\ &= c(M_k \oplus (Q \cup R)) \\ &= c(M_{k+1}) \end{aligned}$$

which confirms that $M_k \oplus P$ is a minimum cost matching of size $k+1$. □

To implement Algorithm 5, we need to specify how to compute the M-augmenting path P that minimizes $\Delta c(P; M)$. Recall that M-augmenting paths are in one-to-one correspondence with paths from $L \cap F$ to $R \cap F$ in the directed graph G_M. Assign costs to the edges of G_M by specifying that directed edge (u, v) has cost $c(u, v)$ if $(u, v) \notin M$ and (v, u) has cost $-c(u, v)$ if $(u, v) \in M$. According to this cost assignment, the cost of a path P in G_M is equal to $\Delta c(P; M)$, so we have reduced the problem of finding an augmenting path of minimum incremental cost to computing a minimum-cost path in G_M from $L \cap F$ to $R \cap F$. Since G_M has a mixture of positive and negative edge costs, the appropriate algorithm for finding

a minimum-cost path is the Bellman-Ford algorithm. In order for Bellman-Ford to succeed in solving the minimum-cost path problem, we require that G_M has no negative-cost cycles. Fortunately, we can prove that there are no negative-cost cycles in G_M. Indeed, if C is a cycle in G_M then C is an M-alternating cycle, hence $M \oplus C$ is a matching with the same cardinality as M. By the induction hypothesis,

$$c(M) \leq c(M \oplus C) = c(M) + \Delta c(C; M)$$

which verifies that $\Delta c(C; M) \geq 0$, i.e. the cost of cycle C is G_M is non-negative.

The running time of the Bellman-Ford algorithm is $O(mn)$, and Algorithm 5 requires $n/2$ iterations of its outer loop, with a call to Bellman-Ford in each iteration. Hence, the total running time is $O(mn^2)$. Next, we will present a modified implementation that improves the running time by finding a way to use Dijkstra's algorithm instead of Bellman-Ford.

To be able to use Dijkstra's algorithm, we need to modify the edge costs in G_M so that they become non-negative, yet searching for a minimum-cost path from $L \cap F$ to $R \cap F$ is still equivalent to searching for a minimum-incremental-cost M-augmenting path. The key idea will be to place values y_u on the vertices of G_M and redefine the edge costs to equal the following "reduced costs":

$$\begin{cases} c^y(u,v) = c(u,v) - y_u - y_v & \text{if } (u,v) \notin M \\ c^y(v,u) = y_u + y_v - c(u,v) & \text{if } (u,v) \in M. \end{cases} \tag{3}$$

To explain when this modification of the edge costs is effective, we introduce the following definition.

Definition 7. Let G be a bipartite graph with edge costs, and let M be a matching in G. An assignment of values y_u to the vertices of G is called *M-compatible* if it satisfies the following properties.

1. $y_u + y_v \leq c(u,v)$ for all edges (u,v).
2. $y_u + y_v = c(u,v)$ for all $(u,v) \in M$.
3. $y_u = \max_{w \in L}\{y_w\}$ for all $u \in L \cap F$.
4. $y_v = \max_{w \in R}\{y_w\}$ for all $v \in R \cap F$.

One useful property of this definition is that if there exists an M-compatible labeling $\boldsymbol{y}$, then it furnishes a very succinct proof that M has the minimum cost among all matchings of its cardinality.

Lemma 12. *If G is a bipartite graph, M is a matching in G, and $\boldsymbol{y}$ is an M-compatible labeling of vertices, then M has the minimum cost among all matchings with the same number of edges as M.*

Proof. Let M' be another matching with the same number of edges as M. Let $W(M)$ and $W(M')$ denote the set of vertices that are matched in M and M', respectively. Then

$$\begin{aligned} c(M) &= \sum_{(u,v)\in M} c(u,v) = \sum_{(u,v)\in M} y_u + y_v = \sum_{w\in W(M)} y_w \\ c(M') &= \sum_{(u,v)\in M'} c(u,v) \geq \sum_{(u,v)\in M} y_u + y_v = \sum_{w\in W(M')} y_w \\ c(M') - c(M) &= \sum_{w\in W(M')} y_w \; - \sum_{w\in W(M)} y_w. \end{aligned}$$

If k denotes the number of edges in M, then the vertex sets $W(M)$ and $W(M')$ both contain k vertices on each side of the graph. Any vertex that belongs to $W(M')$ but not $W(M)$ is free in M. The M-compatibility condition ensures that $y_u \geq y_{u'}$ whenever u is free in M and u' belongs to the same side of the graph. Hence, $\sum_{w\in W(M')} y_w \geq \sum_{w\in W(M)} y_w$ and $c(M') - c(M) \geq 0$. □

The main algorithmic use of M-compatible labelings is that, as advertised, they allow us to use modified edge costs to more rapidly find an M-augmenting path of minimum incremental cost.

Lemma 13. *Suppose M is a matching and $\boldsymbol{y}$ is M-compatible. Then $c^y(e) \geq 0$ for each edge e in G_M, and the M-augmenting paths P of minimum incremental cost are precisely those that minimize $c^y(P)$.*

Proof. The relation $c^y(e) \geq 0$ follows from the definition of c^y and from properties 1 and 2 of a compatible labeling. If P is an M-augmenting path from $s \in L \cap F$ to $t \in R \cap F$ then

$$\begin{aligned} c^y(P) &= \sum_{(u,v)\in P\setminus M} c^y(u,v) \; + \sum_{(u,v)\in P\cap M} c^y(v,u) \\ &= \sum_{(u,v)\in P\setminus M} c(u,v) - y_u - y_v \; + \sum_{(u,v)\in P\cap M} y_u + y_v - c(u,v) \\ &= \Delta c(P;M) - (y_s + y_t) \end{aligned} \tag{4}$$

The second line follows from the definition of c^y. The third line follows because the y_u terms corresponding to internal vertices of P cancel: each such vertex belongs to exactly one edge of $P \setminus M$ and one edge of $P \cap M$. The fourth line follows from property 3 of a compatible labeling.

To conclude the proof of the lemma, observe that property 4 of a compatible labeling ensures that the value $y_s + y_t$ subtracted on the right side of (4) does not depend on the identity of the vertices s and t, only on the fact that $s \in L \cap F$ and $t \in R \cap F$. □

With Lemma 13 in hand, the strategy for modifying Algorithm 5 becomes clear: we need to compute a matching M and a vertex labeling $\boldsymbol{y}$ in each iteration, while maintaining the property that $\boldsymbol{y}$ is M-compatible. Then, the search for the M-augmenting path of minimum incremental cost can be carried out using Dijkstra's algorithm on the graph G_M with edge

Algorithm 6 Improved algorithm for minimium cost bipartite perfect matching.

Initialize $M = \emptyset$.
Initialize $y_u = 0$ for all $u \in V$.
while M is not a perfect matching **do**
    Construct the graph G_M and compute reduced costs c^y.
    Run Dijkstra's algorithm to compute, for every $w \in V(G_M)$, the minimum-reduced-cost path from $L \cap F$ to w. Denote the reduced cost of this path by d_w. If there is no path from $L \cap F$ to w in G_M, let $d_w = \infty$.
    Let P be a minimum-reduced-cost path from $L \cap F$ to $R \cap F$.
    $M \leftarrow M \oplus P$.
    For all $u \in L$, $y_u \leftarrow y_u + (c^y(P) - d_u)^+$.
    For all $v \in R$, $y_v \leftarrow y_v - (c^y(P) - d_v)^+$.
end while
Output M.

costs defined by c^y. The pseudocode for this search process is presented in Algorithm 6, where we adopt the notation that x^+ denotes $\max\{x, 0\}$ for any real number x.

Lemmas 12 and 13 already do most of the work of proving the correctness of this algorithm. The only remaining step is to show that y in M-compatible in each iteration of the algorithm.

Lemma 14. *At the start of every iteration of Algorithm 6 the labeling $\boldsymbol{y}$ is M-compatible.*

Proof. The proof is by induction on the number of iterations of the main loop. In the base case when $M = \emptyset$ and $y_u = 0$ for all u, property 1 of an M-compatible labeling is satisfied because the edge costs are non-negative, and the remaining properties are trivially satisfied. For the induction step, assume a loop iteration starts with matching M and with M-compatible labeling y. Let $M' = M \oplus P$ denote the new matching at the end of the loop iteration, and let y' denote the new labeling.

First, let us verify that $c(u, v) \geq y'_u + y'_v$ for all edges $e = (u, v)$, with equality when $e \in M'$.

Case 1: $e \in M$**.** For $e = (u, v) \in M$ we have $c(u, v) = y_u + y_v$ by the induction hypothesis. The unique edge leading into vertex u in G_M is (v, u) which satisfies $c^y(v, u) = 0$. Hence $d_u = d_v$ which implies $y'_u + y'_v = y_u + y_v = c(u, v)$.

Case 2: $e \in P \setminus M$**.** In this case the directed edge (u, v) in G_M belongs to the shortest path from $L \cap F$ to v. In fact the shortest such path is the initial segment of P ending with edge (u, v). Hence

$$d_v = d_u + c^y(u, v) = d_u + c(u, v) - y_u - y_v$$

which can be rearranged to yield

$$(y_u - d_u) + (y_v + d_v) = c(u, v).$$

Furthermore, the quantities $c^y(P) - d_u$ and $c^y(P) - d_v$ are non-negative, because $c^y(P) - d_u$ (respectively, $c^y(P) - d_v$) is the sum of $c^y(e)$ over all edges e in the subpath of P from u (respectively, v) to the right endpoint of P, and $c^y(e) \geq 0$ for all e. Hence,

$$y'_u + y'_v = y_u + (c^y(P) - d_u) + y_v - (c^y(P) - d_v) = (y_u - d_u) + (y_v + d_v) = c(u, v).$$

Case 3: $e \notin P \cup M$ In this case the directed edge (u, v) belongs to $E(G_M)$ so

$$d_v \leq d_u + c^y(u, v) \tag{5}$$

$$c^y(P) - d_u \leq c^y(P) - d_v + c^y(u, v) \tag{6}$$

$$(c^y(P) - d_u)^+ \leq (c^y(P) - d_v)^+ + c^y(u, v). \tag{7}$$

The last line follows from the line above by case analysis. If $c^y(P) - d_u < 0$ then the left side of Inequality (7) is zero and the right side is clearly non-negative. If $c^y(P) - d_u \geq 0$ then the left sides of (6) and (7) are equal to one another, and the right side of (7) is greater than or equal to the right side of (6). Now,

$$\begin{aligned} y'_u + y'_v &= y_u + (c^y(P) - d_u)^+ + y_v - (c^y(P) - d_v)^+ \\ &\leq y_u + (c^y(P) - d_v)^+ + c^y(u, v) + y_v - (c^y(P) - d_v)^+ \\ &= y_u + c^y(u, v) + y_v = c(u, v). \end{aligned}$$

In all three cases we conclude that y' satisfies property 1 of a compatible labeling. Since the edges of M' all belong to either M or $P \setminus M$, and we have seen that $y'_a + y'_b = c(a, b)$ in both of those cases, property 2 is also satisfied. Property 3 follows from the observation that for $u \in L$, the difference $y'_u - y_u = (c^y(P) - d_u)^+$ is maximized when $d_u = 0$, which holds for all $u \in L \cap F$. Finally, property 4 follows from the observation that for $v \in R$, the difference $y'_v - y_v = -(c^y(P) - d_v)^+$ is maximized when $d_v \geq c^y(P)$, which holds for all $v \in R \cap F$. □

3.2 LP relaxation

The aim of this section and the following one is to give the reader some insight into *where Algorithm 6 comes from* and *how it fits into broader themes in algorithm design.* When one first encounters Algorithm 6, at least two aspects may seem ad hoc and unmotivated: the definition of an M-compatible labeling, and the update rules for y_u and y_v at the end of each iteration of the while-loop. We will see that both of these features of the algorithm can be interpreted as stemming from the systematic analysis of a *linear programming (LP) relaxation of the min-cost bipartite matching problem.*

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

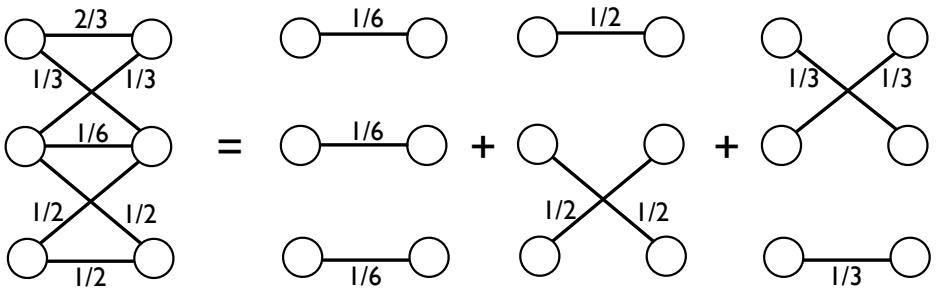

Figure 3: A fractional perfect matching.

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.)

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

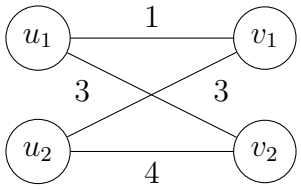

Figure 4: An instance of bipartite minimum cost perfect matching.

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

$$x_{11} + 3x_{12} + 2x_{21} + 4x_{22} = 5 \tag{8}$$

$$x_{11} + 3x_{12} + 3x_{21} + 4x_{22} \geq 5 \tag{9}$$

Inequality (9) is derived from (8) 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 (9) 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

$$\sum_{(u,v) \in L \times R} (y_u + y_v) x_{uv} = \sum_{u \in L} y_u + \sum_{v \in R} y_v. \tag{10}$$

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 (10) 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.

3.3 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 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

$$\delta = \min\{c(u,v) - y_u - y_v \,|\, u \in L \cap T, v \in R \setminus T\} \tag{11}$$

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 (11); 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 (11). 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

$$y_{u,s} = \begin{cases} y_{u,0} + \Delta_s - \Delta_{s(u)} & \text{if } u \in L \cap T \\ y_{u,0} & \text{if } u \in L \setminus T \end{cases} \tag{12}$$

$$y_{v,s} = \begin{cases} y_{v,0} - \Delta_s + \Delta_{s(v)} & \text{if } v \in R \cap T \\ y_{v,0} & \text{if } v \in R \setminus T \end{cases} \tag{13}$$

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 (12)-(13). 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 8. 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

$$\alpha_i = g(d(i)) \quad \forall i \tag{14}$$

$$\beta_j = 1 - g(\ell(j)) \quad \forall j, \tag{15}$$

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 (14)-(15) is feasible. This is because at the time we finish processing vertex j, the inequality $d(i) \geq \ell(j)$ is satisfied by all neighboring vertices i. Since the value $d(i)$ will not subsequently decrease, we also have $d(i) \geq \ell(j)$ at termination. Furthermore, since g is an increasing function, we have

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

which verifies dual feasibility.

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

$$\frac{e}{e-1} \sum_{(i,j) \in E} x_{ij} \geq \sum_{i \in L} \alpha_i + \sum_{j \in R} \beta_j. \tag{16}$$

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 (16) implies that the waterfilling algorithm is $\left(\frac{e}{e-1}\right)$-competitive.

To prove (16), 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 $d(i) \leq t$ 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 (16) 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 $i, k \in \{j, j+1, \ldots, n\}$, 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 $x_{\pi(j),j} = x_{\pi(j+1),j} = \cdots = x_{\pi(n),j}$ 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

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

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 7 below.

Algorithm 7 The RANKING algorithm.

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

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

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

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 $\alpha_i + \beta_j \ge 1$ 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 15 (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 16 (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 17. 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 15) 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 16), $\beta_j \geq \beta_j^c = \frac{e}{e-1} \cdot (1 - h(y^c))$ for all choices of Y_i. The lemma now follows from the integral equation listed above as property 3 of h. □

5 Algebraic Algorithms for Matching Problems

The close relationship between matrix determinants and perfect matchings has surprising and beautiful algorithmic applications, which we explore in this section.

5.1 Permanents, determinants, and graph matchings

For a bipartite graph $G = (L, R, E)$ with $|L| = |R| = n$, let $L = \{u_1, u_2, \ldots, u_n\}$ and $R = \{v_1, v_2, \ldots, v_n\}$ and define A to be the "bipartite adjacency matrix" whose entries are

$$A_{ij} = \begin{cases} 1 & \text{if } (u_i, v_j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

The number of perfect matchings in G is expressed by a multivariate polynomial in the entries of A called the *permanent*.

$$\operatorname{per}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} A_{i,\sigma(i)}.$$

Here the sum is over all permutations σ of the index set $\{1, 2, \ldots, n\}$; this set of permutations is denoted by S_n.

The sum defining $\operatorname{per}(A)$ has $n!$ terms, so it is not apparent how to evaluate it efficiently. In fact, the problem of computing the permanent of a matrix is known to be complete for the complexity class $\sharp\mathsf{P}$, which is at least as hard as NP and generally believed to be much harder. On the other hand, a very similar polynomial that is easy to evaluate is the determinant of a matrix,

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} A_{i,\sigma(i)}.$$

The only difference is the factor $\operatorname{sgn}(\sigma)$, which equals $+1$ if σ is the product of an even number of transpositions and -1 if σ is the product of an odd number of transpositions.

The determinant can be evaluated in polynomial time, for example using Gaussian elimination to perform a sequence of elementary row operations to put it in upper triangular form. This allows us to express an arbitrary matrix A in the form

$$A = E_1 E_2 \ldots E_m U$$

where U is an upper triangular matrix and each of $E_1, E_2, \ldots, E_m$ is an *elementary matrix*. (An elementary matrix is a matrix obtained from the identity matrix by either transposing two rows or modifying one entry. In the former case, its determinant is -1, in the latter case its determinant is the product of its diagonal entries.) Since the determinant of an elementary matrix is easy to evaluate, and since the determinant of an upper triangular matrix is the product of its diagonal entries, the relation $\det(A) = (\prod_{i=1}^{m} \det(E_i)) \cdot \det(U)$ allows us to efficiently evaluate $\det(A)$. A naïve implementation of this Gaussian elimination algorithm takes $O(n^3)$ time: the elementary matrices $E_1, \ldots, E_m$ are computed by performing a sequence of $O(n^2)$ elementary row operations on A, and each row operation entails performing $O(n)$ arithmetic operations. Later in this section we will discuss how to compute $\det(A)$ in $O(n^{\omega+o(1)})$ time, where $O^*(n^{\omega+o(1)})$ denotes the number of arithmetic operations required for multiplying two n-by-n matrices. (The exponent is written as $\omega+o(1)$ to account for the possibility that there is no algorithm whose running time is precisely $O(n^\omega)$, but instead there is an algorithm whose running time is $O(n^{\omega+\varepsilon})$ for every $\varepsilon > 0$.) Fast matrix multiplication remains an active area of research; at the time of writing these notes, the asymptotically fastest known algorithm, due to Alman and Vassilevska Williams, runs in time $O(n^{2.373})$.

One consequence of these observations is that, although evaluating the permanent of an integer matrix is thought to be computationally hard, it is easy to determine whether the permanent is even or odd. This is because $\operatorname{sgn}(\sigma) \equiv 1 \pmod 2$ for every permutation σ, so the permanent and determinant of an integer matrix are always congruent mod 2. Hence, perhaps surprisingly, there is an efficient algorithm to determine whether a bipartite graph has an even or odd number of perfect matchings.

5.2 Lovász's algorithm for perfect matching detection

It is rarely useful to know whether a graph has an even or odd number of perfect matchings, but it is often useful to know whether a graph has at least one perfect matching. There is a randomized algorithm, discovered in 1979 by Lovász, that reduces this problem to computing a matrix determinant. Since this algorithm runs in time $O(n^{2.373})$, it is asymptotically faster than the $O(m\sqrt{n})$ running time of the Hopcroft-Karp algorithm when the graph G is sufficiently dense, having $m \gg n^{1.873}$ edges.

Let B be a matrix defined similarly to the matrix A, but using formal variables x_{ij}, one for each edge (u_i, v_j) of the bipartite graph.

$$B_{ij} = \begin{cases} x_{ij} & \text{if } (u_i, v_i) \in E \\ 0 & \text{otherwise.} \end{cases}$$

The multivariate polynomial $\det(B)$ is non-zero if and only if G has a perfect matching. That is because the formula

$$\det(B) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} B_{ij}$$

expresses $\det(B)$ as a sum of $n!$ terms, each of which is either zero, if the set of pairs $M = \{(u_i, v_{\sigma(i)})\}$ does not constitute a perfect matching in G, or a monomial of degree n in

the formal variables $\{x_{ij}\}$, if M is a perfect matching in G. Furthermore, in case G has at least one perfect matching, distinct perfect matchings correspond to monomials with distinct sets of variables, so there is no cancellation in the above formula and the polynomial $\det(B)$ is non-zero.

Since $\det(B)$ is a multivariate polynomial that can potentially have as many as $n!$ distinct monomials, directly evaluating $\det(B)$ is computationally inefficient. Instead, Lovász's algorithm substitutes random numbers X_{ij} in place of the formal variables x_{ij} and evaluates the determinant of the resulting matrix. If the determinant is non-zero, then $\det(B)$ must be a non-zero polynomial. However, if the determinant evaluates to zero after substituting $\{X_{ij}\}$, it is still possible that $\det(B)$ is a non-zero polynomial and we just made an unlucky substitution that caused the polynomial to evaluate to zero. However, we can bound the probability of this unlucky event using a fact known as the *Schwartz-Zippel Lemma*.

Lemma 18 (Schwartz-Zippel)**.** *Let $P[x_1, \ldots, x_m]$ be a non-zero multivariate polynomial with coefficients in a field $\mathbb{F}$, and suppose that the exponent of any variable in any monomial of P is at most d. If S is a set of s elements of $\mathbb{F}$ and $X_1, \ldots, X_m$ are independent random variables, each uniformly distributed over S, then the probability that $P(X_1, \ldots, X_m) = 0$ is at most $\frac{md}{s}$.*

Proof. The proof is by induction on m. In the base case $m = 0$, P is a non-zero constant so the probability that it evaluates to zero is indeed equal to zero. For the induction step, let $c \leq d$ denote the maximum degree of the variable x_m in any monomial of P. We can express P as

$$P(x_1, \ldots, x_m) = \sum_{i=0}^{c} Q_i(x_1, \ldots, x_{m-1}) x_m^i,$$

where each Q_i is a polynomial in $x_1, \ldots, x_{m-1}$ and $Q_c \neq 0$. If $P(X_1, \ldots, X_m) = 0$ then one of the following two events must happen.

1. $Q_c(X_1, \ldots, X_{m-1}) = 0$, or
2. $Q_c(X_1, \ldots, X_{m-1}) \neq 0$ but $P(X_1, \ldots, X_m) = 0$.

By the induction hypothesis, the first of these two events has probability at most $(m-1)d/s$. For any fixed $X_1, \ldots, X_{m-1}$ such that $Q_c(X_1, \ldots, X_{m-1}) \neq 0$, the second event happens only when X_m is a root of the polynomial $R(x) = \sum_{i=0}^{c} Q_i(X_1, \ldots, X_{m-1}) x^i$. Since a polynomial of degree c has at most c roots in $\mathbb{F}$, the probability that X_m is equal to one of these roots is at most c/s, which is less than or equal to d/s. Summing the probabilities of the two events, we find that the probability of the event $P(X_1, \ldots, X_m) = 0$ is at most $(m-1)d/s + d/s = md/s$. □

Corollary 19. *Let $G = (L, R, E)$ be a bipartite graph with vertex sets $L = \{u_1, \ldots, u_n\}$ and $R = \{v_1, \ldots, v_n\}$. For any $\delta > 0$, let S be a set of at least m/δ distinct integers, where m is the number of edges of G. Consider the random matrix C formed by sampling independent random numbers X_{ij}, each uniformly distributed in S, and assembling them into the matrix*

$$C_{ij} = \begin{cases} X_{ij} & \text{if } (u_i, v_j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

If G has no perfect matching then $\det(C) = 0$*, and if G has a perfect matching then* $\det(C) \neq 0$ *with probability at least* $1 - \delta$*.*

5.3 Fast algorithms for matrix inversion

To finish analyzing Lovász's randomized algorithm for deciding whether a bipartite graph contains a perfect matching, we explain how to decide in $O(n^{\omega+o(1)})$ time whether the determinant of a matrix is zero. It turns out that the determinant of a matrix over an arbitrary ring can be computed in $O(n^{\omega+o(1)})$ ring operations. That algorithm is more complicated than necessary for the present application, which only requires testing whether a matrix defined over the real numbers has a non-zero determinant, so we will present a simpler algorithm that suffices for that application.

The starting point of our algorithm is the observation that for a matrix A over any field, the following properties are equivalent:

1. $\det(A) \neq 0$
2. A is invertible
3. $A^T A$ is invertible

Crucially, when A is a matrix over the real numbers, the matrix $C = A^T A$ is symmetric and positive semidefinite. It is invertible if and only if it is strictly positive definite. Therefore, to test whether $\det(A) = 0$ for an arbitrary matrix A, it suffices to be able to test whether a symmetric positive semidefinite matrix is invertible.

We will define a recursive algorithm AttemptInvert which, when applied to a symmetric positive semidefinite matrix, either computes the inverse matrix or determines that the matrix is not invertible. We will assume the matrix is $n \times n$ where n is a power of 2. If n is not a power of 2, then one should perform the following preprocessing step before running time algorithm. Let N be a power of 2 such that $n < N < 2n$, and let C' be the $N \times N$ matrix with the following block structure.

$$C' = \begin{pmatrix} C & 0 \\ 0 & \mathbb{1} \end{pmatrix}$$

Here, $\mathbb{1}$ denotes the identity matrix. The matrix C' is invertible if and only if C is, and the inverse of C' contains the inverse of C as its upper $n \times n$ block; hence, inverting C' implies that we can easily find the inverse of C.

Assuming n is a power of 2, we partition C into square blocks of size $(\frac{n}{2}) \times (\frac{n}{2})$, as follows.

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Now, we use the following identity, which is valid when C_{11} is invertible.

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} \mathbb{1} & 0 \\ C_{21}C_{11}^{-1} & \mathbb{1} \end{pmatrix} \begin{pmatrix} C_{11} & 0 \\ 0 & C_{22} - C_{21}C_{11}^{-1}C_{12} \end{pmatrix} \begin{pmatrix} \mathbb{1} & C_{11}^{-1}C_{12} \\ 0 & \mathbb{1} \end{pmatrix}. \tag{17}$$

The matrix $S = C_{22} - C_{21}C_{11}^{-1}C_{12}$ is called the *Schur complement* of $C_1 1$ in C. Inverting both sides of equation (17) we obtain

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}^{-1} = \begin{pmatrix} \mathbb{1} & -C_{11}^{-1}C_{12} \\ 0 & \mathbb{1} \end{pmatrix} \begin{pmatrix} C_{11}^{-1} & 0 \\ 0 & S^{-1} \end{pmatrix} \begin{pmatrix} \mathbb{1} & 0 \\ -C_{21}C_{11}^{-1} & \mathbb{1} \end{pmatrix} \tag{18}$$

This equation implies the following recursive algorithm. If $n = 1$, the matrix C is a scalar. The algorithm either outputs its inverse, or if $C = 0$ it outputs "not invertible." If $n > 1$ is a power of 2, we perform the following steps.

1. Run AttemptInvert(C_{11}) to obtain C_{11}^{-1}. If C_{11} is not invertible, output "not invertible".
2. Compute $C_{11}^{-1}C_{12}$ and its transpose, $C_{21}C_{11}^{-1}$.
3. Compute $S = C_{22} - C_{21}C_{11}^{-1}C_{12}$.
4. Run AttemptInvert(S) to obtain S^{-1}. If S is not invertible, output "not invertible."
5. Use equation (18) to compute C^{-1}.

The algorithm inverts a matrix of size n by recursively calling matrix inversion on two matrices of size $n/2$ and performing a constant number of matrix multiplication (and subtraction) operations on matrices of size n or $n/2$. Hence, its running time $T(n)$ satisfies the recurrence $T(n) = 2T(n/2) + O(n^{\omega+o(1)})$, whose solution is $T(n) = O(n^{\omega+o(1)})$.

Equation (18) justifies that when the algorithm outputs a matrix (as opposed to outputting the message "not invertible") then its output is indeed the inverse of C. To finish proving the correctness of the algorithm we must show, conversely, that when C is invertible the algorithm outputs the inverse of C, i.e. it will never mistakenly output "not invertible". The proof is by induction on n, and the key observation here is that C is assumed to be symmetric and positive semidefinite. If C is also invertible, then it is positive definite, meaning that $x^{\mathsf{T}}Cx > 0$ for every $x \neq 0$. This implies that C_{11} is invertible, as otherwise a non-zero vector $y \in \mathbb{R}^{n/2}$ that satisfies $C_{11}y = 0$ could be padded with zeroes to obtain a non-zero vector $x \in \mathbb{R}^n$ satisfying $x^{\mathsf{T}}Cx = 0$, contradicting the positive definiteness of C. Having established that C_{11} is invertible, we apply the induction hypothesis to deduce that the recursive call to AttemptInvert($C_1 1$) will succeed in inverting it. Now, by taking the determinant of both sides of equation (17), we see that $\det(C) = \det(C_{11})\det(S)$. Since $\det(C)$ and $\det(C_{11})$ are both non-zero, it follows that $\det(S) \neq 0$ as well. By another application of the induction hypothesis, the recursive call to AttemptInvert(S) will likewise succeed in inverting S. Consequently, the algorithm will succeed in inverting C, as desired.

This concludes the analysis of Lovász's algorithm for deciding whether a bipartite graph contains a perfect matching. Before moving on, however, we should mention that Mucha and Sankowski in 2004 discovered a randomized algorithm with running time $O(n^{\omega+o(1)})$ that not only decides whether a bipartite graph contains a perfect matching, it also finds the perfect matching (with probability close to 1) if one exists. Presenting their algorithm would be beyond the scope of these lecture notes, however the following lemma at least provides a useful initial step.

Lemma 20. *Let $G = (L, R, E)$ be a bipartite graph with vertex sets $L = \{u_1, \ldots, u_n\}$ and $R = \{v_1, \ldots, v_n\}$ and let B be the matrix given by*

$$B_{ij} = \begin{cases} x_{ij} & \text{if } (u_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

where the x_{ij} are formal variables. For any edge $(u_i, v_j) \in E$, the graph G has a perfect matching containing edge (u_i, v_j) if and only if B is invertible and the matrix entry $(B^{-1})_{ji}$ is a non-zero element of the field of rational functions of the variables $\{x_{ij}\}$.

Proof. We have seen that G has a perfect matching if and only if B is invertible. Now, let G_{-ij} denote the graph obtained from G by deleting u_i and v_j. For any edge $e = (u_i, v_j)$, the existence of a perfect matching in G containing edge e is equivalent to the existence of a perfect matching in G_{-ij}. If B_{-ij} denotes the matrix obtained from B by deleting row i and column j, then G_{-ij} has a perfect matching if and only if B_{-ij} is invertible, i.e. $\det(B_{-ij}) \neq 0$. Now, Cramer's rule tells us that

$$(B^{-1})_{ji} = (-1)^{i+j} \det(B_{-ij}) / \det(B).$$

Hence, the existence of a perfect matching in G containing edge e is equivalent to the condition that $(B^{-1})_{ji} \neq 0$. □

As before, we can substitute random values for the variables x_{ij} by sampling numbers uniformly at random from a sufficiently large set S, to form a random matrix C. Then an application of the Schwartz-Zippel Lemma will ensure that with high probability, the non-zero entries of the matrix C^{-1} correspond precisely to the set of edges that belong to a perfect matching. Since the matrix inverse can be computed in $O(n^{\omega+o(1)})$ time, this gives a $O(n^{\omega+o(1)})$-time randomized algorithm that not only decides whether G contains a perfect matching, but also finds an edge that belongs to a perfect matching in G in the event that a perfect matching exists. The naïve idea of iterating this algorithm n times to discover the edges of a perfect matching one by one would lead to an algorithm with running time $O(n^{\omega+1+o(1)})$, which is slower than Hopcroft-Karp even for dense graphs. Mucha and Sankowski instead found a way to interleave the process of finding edges of the matching with the steps of the divide-and-conquer matrix inversion algorithm, so that a perfect matching is computed simultaneously with executing a single instance of matrix inversion in $O(n^{\omega+o(1)})$ time.

5.4 Digression: detecting cliques using fast matrix multiplication

Detecting perfect matchings in bipartite graphs is not the only surprising application of fast matrix multiplication to the design of combinatorial algorithms. For example, consider the problem of deciding whether a given undirected graph contains a triangle. The naïve algorithm for this problem tests every set of $\binom{n}{3}$ vertices to determine whether they induce a triangle; this brute-force search algorithm runs in time $O(n^3)$. A faster algorithm to test for the presence of a triangle is based on the observation that G contains a triangle if and

only if there are two vertices u and v such that u and v are joined by an edge and they are also joined by a path of length 2. In other words, G contains a triangle if and only if its adjacency matrix A has a non-zero entry in a location where the squared adjacency matrix A^2 also has a non-zero entry. Since A^2 can be computed in $O(n^{\omega+o(1)})$ time, and it only takes $O(n^2)$ time to perform a brute-force search for a pair u, v such that the (u, v) entries of A and A^2 are both non-zero, this yields a triangle-detection algorithm running in time $O(n^{\omega+o(1)})$.

Similarly, fast matrix multiplication can be used to design an algorithm for detecting a k-clique in an undirected graph, with running time faster than the naïve $O(k^2 n^k)$ brute-force search. For simplicity assume k is divisible by 3. Let $N = \binom{n}{k/3} = O(n^{k/3})$, and consider the N-vertex graph H with one vertex v_S corresponding to each set of $k/3$ vertices of G. Place an edge (v_S, v_T) in H if the vertex set $S \cup T$ constitutes a $(2k/3)$-clique in G. The brute-force algorithm to compute the adjacency matrix of H takes time $O(k^2 N^2)$, since there are N^2 entries in the adjacency matrix and for each entry we need to test for the presence of $\binom{2k/3}{2} = O(k^2)$ edges in G to determine whether there is an edge between v_S and v_T in H. Now, observe that G contains a k-clique if and only if H contains a triangle. We have seen that there is an algorithm to decide whether H contains a triangle that runs in time

$$O(N^{\omega+o(1)}) = O(n^{(k/3)\cdot(\omega+o(1))}) = O(n^{(\omega/3+o(1))\cdot k}).$$

This algorithm, due to Nešetřil and Poljak (1985) is still the asymptotically fastest known algorithm for deciding if a graph contains a k-clique. Since $\omega < 2.373$, its running time is $O(n^{0.791\,k})$.

5.5 Parallel algorithms for finding perfect matchings

The relationship between matchings and determinants can also be leveraged to design a randomized *parallel* algorithm for finding a perfect matching (if one exists) in a bipartite graph.

5.5.1 Overview of parallel complexity theory

Before we present the algorithm for finding a perfect matching, we must explain a bit about the theory of parallel algorithms. Just as sequential algorithms can be modeled as Turing machines or random access machines (RAMs), or as uniform[1] families of Boolean circuits, parallel algorithms can either be modeled as *parallel random access machines (PRAMs)* or as uniform families of Boolean circuits. The PRAM definitions are a bit complicated because one must specify the conventions for dealing with concurrency: a PRAM can be either CRCW (concurrent read, concurrent write), CREW (concurrent read, exclusive write), or EREW (exclusive read, exclusive write). The Boolean circuit definition is much more straightforward: a circuit is simply a directed acyclic graph whose vertices (called "gates") are associated with the operations INPUT, AND, OR, NOT. An INPUT gate has in-degree

[1]Here, the word "uniform" means that there is a Turing machine which, on input 1^n, writes a description of the Boolean circuit to be used for solving problem instances of input size n.

0 and a NOT gate has in-degree 1. The in-degree of an AND or OR gate is called its "fan-in". If the circuit has n input gates, it can be evaluated on n-bit inputs by labeling each input gate with the corresponding bit of the input string, labeling each edge with its tail's label, and labeling every non-input gate with the bit obtained by performing the associated Boolean operation on the bits that label its incoming edges. When using such a circuit to model a parallel algorithm, the assumption is that if a set of gates forms an *antichain* in the directed acyclic graph (meaning that no two of them are joined by a directed path) then they represent operations that can be executed in parallel (i.e., simultaneously) at any time after the labels of all of their incoming edges have been computed.

The two parameters that govern a parallel algorithm's complexity are its *work* — the total number of gates in the circuit — and its *running time*, which is defined to be the length of the longest path in the directed acyclic graph. Equivalently, the running time is t if the vertices can be organized into a sequence of t antichains such that for every edge (u, v), the antichain containing u occurs before the antichain containing v. These antichains represent the operations performed at the successive time steps of the parallel algorithm.

The complexity theory of parallel algorithms tends to focus on the question of which problems can be solved by an algorithm that performs $n^{O(1)}$ work in $\log^{O(1)}(n)$ time, where n is the input size. For a non-negative integer i, a problem belongs to NC^i if it there is a parallel algorithm that solves it in $n^{O(1)}$ work and $\log^i(n)$ time, in which each gate has fan-in at most 2. If one instead allows gates to have unbounded fan-in, one obtains the complexity class AC^i. It is evident that $\mathsf{NC}^i \subseteq \mathsf{AC}^i \subseteq \mathsf{NC}^{i+1}$; to prove the second containment, note that an AND or OR gate with fan-in $n^{O(1)}$ can be simulated by a depth $O(\log n)$ binary tree of AND or OR gates. A consequence of these containments is that $\bigcup_{i\in\mathbb{N}} NC^i = \bigcup_{i\in\mathbb{N}} AC^i$. The complexity class represented by this union is called NC and plays a role in the theory of parallel algorithms comparable to the role of the complexity class P in the theory of sequential algorithms. A major conjecture in parallel complexity theory is that $\mathsf{NC} \neq \mathsf{P}$.

5.5.2 Parallel algorithms for addition and multiplication

In this section we explain how to add and multiply integers and matrices in parallel in $O(\log n)$ time and $\text{poly}(n)$ work.

Integer addition. The challenge of designing fast parallel algorithms comes into view immediately when we consider how to add two n-bit integers in $O(\log n)$ time. The grade-school algorithm for addition requires "carrying" when the sum in a given place value exceeds one digit. The question of whether to carry a 1 from the digit sum in the n^{th} place value may depend on the carry digit from the $(n-1)^{\text{th}}$ place value, which in turn may depend on the carry digit from the $(n-2)^{\text{th}}$ place value, and so on, leading to a chain of n dependencies. Fortunately the computation can be reorganized so that the chains of dependencies are only logarithmic in length, at the cost of doing $O(n \log n)$ work rather than $O(n)$ work as in the standard grade-school algorithm.

Let us number digits from least significant to most significant, so that the operands a and b are represented by binary digit sequences $a_{n-1}, a_{n-2}, \ldots, a_0$ and $b_{n-1}, b_{n-2}, \ldots, b_0$. In other words, $a = \sum_{i=0}^{n-1} a_i 2^i$ and $b = \sum_{i=0}^{n-1} b_i 2^i$. Define c_i, the "carry digit in place value i", to be

1 if the grade-school algorithm for adding a and b carries a 1 from the sum in place value $i-1$ to the sum in place value i, and otherwise $c_i = 0$. Equivalently, and more precisely,

$$c_i = \left\lfloor \sum_{j=0}^{i-1} (a_j + b_j) 2^{j-i} \right\rfloor .$$

Our algorithm will compute symbols $c_{i,k} \in \{0, 1, *\}$ for $0 \le i < n$ and $0 \le k \le \lceil \log_2(n) \rceil$, with the following meaning. If $c_{i,k} = 0$ or $c_{i,k} = 1$ it means that the carry digit in place value i is assured of being 0 (or 1, respectively) based on the values of the digits $a_{i-1}, a_{i-2}, \ldots, a_{i-2^k}$ and $b_{i-1}, b_{i-2}, \ldots, b_{i-2^k}$. If $c_{i,k} = *$ it means that the carry digit in place value i depends on the carry digit in place value $i - 2^k$; in fact, it means those two carry digits must be equal to one another. Equivalently, and more precisely,

$$\begin{aligned}
c_{i,k} = 1 &\iff \sum_{j=i-2^k}^{i-1} (a_j + b_j) 2^{j-i} \ge 1 \\
c_{i,k} = 0 &\iff \sum_{j=i-2^k}^{i-1} (a_j + b_j) 2^{j-i} < 1 - 2^{-2^k} \\
c_{i,k} = * &\iff \sum_{j=i-2^k}^{i-1} (a_j + b_j) 2^{j-i} = 1 - 2^{-2^k}
\end{aligned}$$

where a_j and b_j are interpreted as being equal to zero when $j < 0$. These values $c_{i,k}$ obey the recurrence

$$c_{i,0} = \begin{cases} 0 & \text{if } a_j = b_j = 0 \\ 1 & \text{if } a_j = b_j = 1 \\ * & \text{if } a_j \ne b_j \end{cases} \tag{19}$$

$$c_{i,k} \begin{cases} 0 & \text{if } c_{i,k-1} = 0 \text{ or } i - 2^{k-1} < 0 \\ 1 & \text{if } c_{i,k-1} = 1 \\ c_{i-2^{k-1},k-1} & \text{if } c_{i,k-1} = * \text{ and } i - 2^{k-1} \ge 0 \end{cases} \tag{20}$$

and when $k = \lceil \log_2(n) \rceil$ the value of $c_{i,k}$ is equal to the carry digit c_i defined earlier. Relations (19) and (20) can be interpreted as a parallel algorithm for computing all of the values $c_{i,k}$ ($0 \le i < n$, $0 \le k \le \lceil \log_2(n) \rceil$) in $O(\log n)$ time and $O(n \log n)$ work. Once the value c_i is known for each i, the digits of the sum $a + b$ can be computed in parallel in constant time; the digit in place value j is the least significant bit of $a_j + b_j + c_j$.

The algorithm presented above adds n-bit non-negative integers in time $O(\log n)$ and work $O(n \log n)$. We leave to the reader the exercise of figuring out how to enhance the algorithm to allow for signed integers which may be negative.

Integer multiplication. The grade-school algorithm for multiplying two n-bit integers involves computing n "partial products", each n bits long, and then summing up the partial products. If the operands are $a = \sum_{i=0}^{n-1} a_i 2^i$ and $b = \sum_{j=0}^{n-1} b_j 2^j$, then the j^{th} partial product is $p_j = \sum_{i=0}^{n-1} a_i b_j 2^{i+j}$. The following two facts are clear.

1. The digits of all the partial products $p_0, p_1, \ldots, p_{n-1}$ can be computed in parallel in constant time and $O(n^2)$ work.

2. $a \cdot b = p_0 + p_1 + \cdots + p_{n-1}$.

To finish presenting the multiplication algorithm, we will show how to compute the sum of n integers, each represented by $2n$ or fewer binary digits, using a parallel algorithm that runs in $O(\log n)$ time. The key trick is the following lemma.

Lemma 21. *There is a parallel algorithm running in $O(1)$ time and $O(b)$ work that reduces the task of computing the sum of three b-bit binary integers to computing the sum of two $(b+1)$-bit binary integers.*

Proof. Let the three operands be $a = \sum_{i=0}^{n-1} a_i 2^i$, $b = \sum_{i=0}^{n-1} b_i 2^i$, and $c = \sum_{i=0}^{n-1} c_i 2^i$. Define

$$r = \sum_{k=0}^{\lfloor (n-1)/2 \rfloor} (a_{2k} + b_{2k} + c_{2k}) 2^{2k} \tag{21}$$

$$s = \sum_{k=0}^{\lfloor (n-2)/2 \rfloor} (a_{2k+1} + b_{2k+1} + c_{2k+1}) 2^{2k+1}. \tag{22}$$

Clearly $r + s = a + b + c = \sum_{i=0}^{n-1} (a_i + b_i + c_i) 2^i$. Since the sums $a_{2k} + b_{2k} + c_{2k}$ and $a_{2k+1} + b_{2k+1} + c_{2k+1}$ are in the range $\{0, 1, 2, 3\}$, when one computes the binary representation of r using equation (21) the digit r_i in place value i depends only on the digits a_{2k}, b_{2k}, c_{2k} where $k = \lfloor i/2 \rfloor$, and can be computed in constant time. Similarly when computing the binary representation of s using equation (22), the digit s_i in place value i depends only on the digits $a_{2k+1}, b_{2k+1}, c_{2k+1}$ where $k = \lfloor (i-1)/2 \rfloor$, and s_i can be computed in constant time as well. Hence, the digits of r and s can all be computed in parallel using $O(n)$ work and $O(1)$ time, as claimed. □

Applying the lemma recursively, we find that the time $T(n, b)$ and work $W(n, b)$ required to add n integers, each of length b binary digits, satisfy the recurrences

$$T(n, b) \leq T(\lceil 2n/3 \rceil, b + 1) + O(1)$$
$$W(n, b) \leq W(\lceil 2n/3 \rceil, b + 1) + O(nb)$$

with base cases $T(2, b) = O(\log b)$ and $W(2, b) = O(b \log b)$. Solving the recurrences, we find that $T(n, b) = O(\log n + \log b) = O(\log(nb))$ and $W(n, b) = O(nb + b \log b)$. In particular, summing up n partial products each composed of $2n$ or fewer binary digits requires time $O(\log n)$ and work $O(n^2)$.

Once again, the extension to signed integer operands is left as an exercise for the reader.

Matrix multiplication. Given two $n \times n$ matrices, each of whose entries is an integer represented using b bits, multiplying them entails computing the inner product of each row of the first matrix with each column of the second matrix. These inner products are

computed in parallel. To compute an inner product, first perform n multiplication operations in parallel (this takes $O(\log b)$ time and $O(b^2)$ work for each multiplication, since the operands have b bits), then sum up the results using the parallel algorithm for summing a list of n numbers presented above; this takes $O(\log(nb))$ time and $O(nb + b \log b)$ work. Hence, the product of two matrices with b-bit integer entries can be computed in $O(\log(nb))$ time and $O(n^3 b + n^2 b^2 + n^2 b \log b)$ work.

5.5.3 Algebraic branching programs and parallel algorithms

We are working our way up to presenting a parallel algorithm for evaluating the determinant of a matrix. In this section we introduce a class of computations parameterized by edge-labeled directed acyclic graphs. The computation encoded by such a graph is called an *algebraic branching program*, and our aim in this section is to define algebraic branching programs and explain how to evaluate them in NC^2 by reducing them to iterated matrix multiplication.

Definition 9. If R is a ring and $x_1, \ldots, x_m$ are formal variables, an *algebraic branching program over R* (in the variables $x_1, \ldots, x_m$) is a directed acyclic graph with two distinguished nodes s (source) and t (sink), each of whose edges e is labeled with an affine form (i.e., a degree-1 polynomial) in the variables $x_1, \ldots, x_m$. If Π is an algebraic branching program in the variables $x_1, \ldots, x_m$, with edge labels $\Pi(e)$, the polynomial f^Π is the multivariate polynomial

$$f^\Pi(x_1, \ldots, x_m) = \sum_{P \in \mathcal{P}(s,t)} \prod_{e \in P} \Pi(e).$$

Here, the sum is over the set $\mathcal{P}(s,t)$ of all paths from s to t in the directed acyclic graph.

The *size* of an algebraic branching program is the number of vertices in the directed acyclic graph. Its *depth* is the maximum number of edges in a path from s to t.

An algebraic branching program Π thereby functions as a succinct encoding of a polynomial f^Π that may have exponentially many monomials, since the cardinality of the path set $\mathcal{P}(s,t)$ is typically exponentially larger than the vertex and edge sets of the graph encoded by Π.

A convenient observation about algebraic branching programs is that they can be efficiently evaluated by dynamic programming. If $(r_1, \ldots, r_m)$ is an m-tuple of elements of R, then the value $f^\Pi(r_1, \ldots, r_m)$ can be efficiently evaluated as follows. First, number the vertices of the branching program as $v_0, v_1, \ldots, v_{n-1}$ such that $s = v_0$, $t = v_{n-1}$, and every edge (v_i, v_j) satisfies $i < j$. (Such a numbering exists because the graph is acyclic.) Then let $\Pi\langle i \rangle$ denote the algebraic branching program obtained by taking the induced subgraph on vertex set $\{v_0, \ldots, v_i\}$ and setting the source and sink to be v_0 and v_i, respectively. The dynamic program evaluates $f^\Pi(r_1, \ldots, r_m)$ by computing the values $f^{\Pi\langle i \rangle}(r_1, \ldots, r_m)$ for $i = 0, 1, \ldots, n-1$ in sequence. The identity

$$f^{\Pi\langle j \rangle} = \sum_{(v_i, v_j) \in E} \ell(v_i, v_j) \cdot f^{\Pi\langle i \rangle},$$

reflects the fact that every path from s to v_j is obtained by appending an edge (v_i, v_j) to a path from s to v_i. This identity together with the initialization $f^{\Pi\langle 0\rangle} = 1$ specifies how the dynamic program evaluates $f^{\Pi} = f^{\Pi\langle n-1\rangle}$.

Another algorithm to evaluate algebraic branching programs uses iterated matrix multiplication. This is based on the observation that the (i, j) entry of a matrix product $M_1 M_2 \dots M_d$ is the sum, over all sequences of indices $i = i_0, i_1, \dots, i_d = j$, of the product $(M_1)_{i_0,i_1} \cdot (M_2)_{i_1,i_2} \cdot \dots \cdot (M_d)_{i_{d-1},i_d}$. Interpreting the sequence of indices as specifying the vertices of a path in a graph, it is natural to encode an algebraic branching program Π using a matrix M^{Π} whose entries are degree-1 polynomials in $R[x_1, \dots, x_m]$. The entries of M^{Π} are defined as follows:

$$(M^{\Pi})_{i,j} = \begin{cases} \ell(v_i, v_j) & \text{if } (v_i, v_j) \in E \\ 1 & \text{if } i = j = n \\ 0 & \text{otherwise.} \end{cases}$$

If d denotes the depth of the branching program, then paths in the set $\mathcal{P}(s, t)$ are in one-to-one correspondence with sequences of indices $i = i_0, i_1, \dots, i_d = j$ such that the product $(M^{\Pi})_{i_0,i_1} \cdot (M^{\Pi})_{i_1,i_2} \cdot \dots \cdot (M^{\Pi})_{i_{d-1},i_d}$ is non-zero. (The correspondence maps a path P to the index sequence obtained by taking the indices of the vertices in P and padding the end of sequence with copies of index n, as necessary, until its length is $d+1$.) Thus, the polynomial f^{Π} is equal to the $(1, n)$ entry of $(M^{\Pi})^d$. This gives a recipe for evaluating an algebraic branching program of size n, on b-bit integer inputs $r_1, \dots, r_m$, using a parallel algorithm that runs in time $O(\log d \cdot \log(nb))$ and work $\mathrm{poly}(n, b)$. The parallel algorithm computes the matrix M obtained by substituting $r_1, \dots, r_m$ for $x_1, \dots, x_m$ in matrix M^{Π}. Then it computes $M^2, M^4, M^8, \dots, M^{2^k}$, where $k = \lceil \log_2(d) \rceil$, by repeated squaring. Finally, it computes M^d by multiplying together a subset of the matrices in the sequence $M, M^2, \dots, M^{2^k}$ (corresponding to the binary digits of d) and outputs the $(1, n)$ entry of M^d. We have seen that each of the matrix multiplications in this algorithm takes $O(\log(nb))$ time and $\mathrm{poly}(n, b)$ work, hence the entire algorithm

5.5.4 An algebraic branching program for the determinant of a matrix

Here, we present a beautiful combinatorial construction due to Mahajan and Vinay that represents the determinant of an $n \times n$ matrix as a polynomial f^{Π}, where Π is an algebraic branching program of size $O(n^3)$.

A basic observation underlying the construction is that permutations of the set $[n] = \{1, 2, \dots, n\}$ can be given a graph-theoretic interpretation in which permutations are in one-to-one correspondence with "cycle covers" of K_n, the bidirected clique on n vertices with self-loops. In other words, K_n is the directed graph with vertex set $[n]$ and edge set $[n] \times [n]$. Define a *cycle cover* of K_n to be a subgraph consisting of vertex-disjoint directed cycles whose vertex sets constitute a partition of $[n]$. Equivalently, a cycle cover of K_n is a subgraph in which every vertex has in-degree 1 and out-degree 1. For every permutation $\sigma \in S_n$, the edge set $\{(i, \sigma(i)) \mid 1 \le i \le n\}$ constitutes a cycle cover of K_n. Conversely, if H is a cycle cover of K_n, there is a corresponding permutation $\sigma \in S_n$ such that for every $i \in [n]$, the

edge $(i, \sigma(i))$ is the unique edge in H leaving vertex i. Thus, we have defined a bijection between permutations in S_n and cycle covers of K_n.

Since the determinant of a matrix is a sum of monomials indexed by permutations, it can be interpreted as a sum over cycle covers of K_n. We now elaborate on this interpretation. Since the sign of a cyclic permutation on k elements is $(-1)^{k-1}$, a cycle cover H composed of $c(H)$ cycles represents a permutation whose sign is $(-1)^{n-c(H)}$. Thus, letting $\mathcal{C}_n$ denote the set of cycle covers of K_n, the determinant of a matrix $X = (x_{ij})_{i,j=1}^n$ is expressed by the following multivariate polynomial of its entries.

$$\det(X) = \sum_{H \in \mathcal{C}_n} (-1)^{n-c(H)} \prod_{(i,j) \in E(H)} x_{ij}. \tag{23}$$

This sum resembles an algebraic branching program, with two exceptions. First, and most importantly, the sum is over cycle covers rather than paths. Second, each term of the sum has an extra factor $(-1)^{n-c(H)}$ that does not seem to be associated to any specific edge of set of edges in H.

To compute the determinant using an algebraic branching program, it would thus be convenient if we could devise some directed acyclic graph G, with source s and sink t, such that s-t paths in G are in one-to-one correspondence with cycle covers of K_n. However, this is probably not possible using a graph of polynomial size. (If it were possible, the same directed graph could be used to compute the permanent of a matrix in polynomial time.) Instead, Mahajan and Vinay found a directed acyclic graph whose s-t paths correspond to combinatorial objects called *clow sequences*. A cycle cover is a special case of a clow sequence, and the clow sequences which are not cycle covers cancel each other out when one sums up their contributions to the algebraic branching program. The sign factor $(-1)^{n-c(H)}$ is vital in order for this cancellation to occur.

Definition 10. A *clow* (short for "closed walk") in K_n is a finite sequence of vertices $i_0, i_1, \ldots, i_\ell$ such that $i_0 = i_\ell$ and $i_k > i_0$ for $0 < k < \ell$. The vertex i_0 is called the *head* of the clow. The number ℓ is called the *length* of the clow. If the edges of K_n are labeled with elements of a ring R, the *weight* of the clow is the product of the labels of its edges.

A *clow sequence* in K_n is a sequence of clows whose heads are numbered in strictly increasing order, and whose lengths sum up to n. The *sign* of a clow sequence is $(-1)^{n-k}$ where k is the number of clows in the sequence. The *weight* of the clow sequence is calculated by multiplying its sign by the product of the weights of its constituent clows.

To compute the sum of the weights of all clow sequences, we seek an algebraic branching program that represents this sum. The vertices of the branching program, apart from the source s and sink t, will be organized into n layers, with edges linking consecutive layers. A clow sequence will be mapped to a path in this layout by removing the second occurrence of the head in each clow (so that a clow of length ℓ is truncated to a sequence of exactly ℓ vertices) and then concatenating the truncated clows into a sequence of n vertices, one in each layer. To ensure that all paths in the layered graph correspond to valid clow sequences, the vertices in each layer will be identified by ordered pairs (i, h), where index i denotes the

vertex currently being visited and index h denotes the head of the clow containing i. The edge set of the graph are then organized to ensure that the heads of the clows in the sequence must be numbered in increasing order.

In more detail, the vertices in layer k will be denoted by $v[i, h, k]$, where $(i, h, k) \in [n]^3$ and $h \leq i$. In layer $k = 1$, we include only vertices of the form $v[h, h, 1]$, since the first vertex of a clow is its head. There are edges of weight 1 from s to each of these vertices. In layer $k > 1$, a vertex $v[j, h, k]$ with $j > h$ has incoming edges from $v[i, h, k-1]$ for every i, and these edges have weight $-x_{ij}$. A vertex $v[h, h, k]$ has incoming edges from $v[i, g, k-1]$ for every i and every $g < h$, and these edges have weight x_{ig}. Finally, each vertex $v[i, h, n]$ has an edge to t of weight x_{ih}.

When a clow sequence is translated into a path as described above, a clow of length ℓ with head h translates to a sequence of ℓ edges in the path, namely all the edges whose tail is of the form $v[i, h, k]$. The product of the weights of those edges is $(-1)^{\ell-1}$ times the weight of the clow. Hence, the product of the weights of all edges in the path is equal to the weight of the clow sequence.

To sum up, the algebraic branching program Π described above has size $O(n^3)$, depth $n+1$, and f^Π is equal to the sum of the weights of all clow sequences in K_n. The cycle covers of K_n correspond to the subset of clow sequences in which each vertex occurs exactly once (provided that the head of a clow is considered to occur once, not twice, in that clow). The sum of the weights of the clow sequences that correspond to cycle covers is the determinant polynomial, $\det(X)$. To finish showing that $f^\Pi = \det(X)$, we will show how to group all the remaining clow sequences into pairs whose weights sum to zero.

If $C_1, C_2, \ldots, C_k$ is a clow sequence that does not correspond to a cycle cover, then there must be at least one vertex that occurs more than once among all the clows in the sequence. Let i be the largest index such that there is a vertex occurring more than once among the clows $C_i, C_{i+1}, \ldots, C_k$. Going through the vertices of C_i in sequence, let us stop at the earliest vertex j in the sequence that satisfies one of the following two conditions.

1. vertex j completes a simple cycle in C_i;
2. vertex j belongs to one of the higher-numbered clows, $C_{i+1}, \ldots, C_k$.

Note that j cannot satisfy both conditions simultaneously: if it did, then it would also occur earlier in C_i (at the start of the simple cycle which finishes at the current occurrence of j) and its earlier occurrence in C_i would also satisfy the second condition. Hence, exactly one of the two conditions is satisfied. If j satisfies the first condition, then we pair the clow sequence $C_1, \ldots, C_k$ with a modified clow sequence obtained as follows. Let C denote the subsequence of C_i beginning and ending with the first and second occurrences of j. Modify C_i to C_i' by extracting C and replacing it with a single occurrence of j. Take the simple cycle C and insert it into the clow sequence as a new clow (whose head is the lowest-numbered vertex of the simple cycle). In this way, the clow sequence $C_1, \ldots, C_k$ is modified to a new clow sequence with the same multiset of edges but one additional clow. If j satisfies the second condition, we pair it with a clow sequence constructed by the reverse operation: we take the clow that contains j and is numbered higher than i (there must be a unique such clow, since $C_{i+1}, \ldots, C_k$ are presumed to be vertex-disjoint) and we merge it into C_i by replacing the

first occurrence of j in C_i with a simple cycle that starts and ends at j and contains all of the edges of the higher-numbered clow that is being merged into C_i. In this way, we pair all clow sequences that are not cycle covers, in such a way that the two members of each pair have the same multiset of edges, but their weights are oppositely signed and hence their net contribution to the total weight is zero.

We have shown that the $n \times n$ determinant polynomial $\det(X)$ can be represented as an algebraic branching program of size $O(n^3)$ and depth $n + 1$. Consequently there is a parallel algorithm to compute the determinant that runs in time $O(\log n \cdot \log(nb))$. Since the input size is nb, this running time is (up to constant factors) bounded by the square of the logarithm of the input size, so the determinant is in NC^2.

5.5.5 A randomized parallel algorithm to find a perfect matching

Combining the NC^2 algorithm for the determinant with Lovász's idea of substituting random numbers for the variables $\{x_{ij}\}$, we immediately obtain a randomized NC^2 algorithm to test whether a bipartite graph has a perfect matching. However, we can go further and use one additional trick to actually compute the edge set of a perfect matching, if one exists, in randomized NC^2. This algorithm is due to Mulmuley, Vazirani, and Vazirani.

The algorithm first assigns random weights to the edges of G using independent uniformly-distributed integers in the range $\{0, 1, \ldots, b-1\}$ where $b = O(\text{poly}(n))$. A key lemma known as the *Isolation Lemma*, which we prove below, shows that with probability at least $1 - \frac{m}{b}$, there is a unique minimum-weight matching. The second key idea in the algorithm is to use matrix determinants to identify the edge set of the minimum-weight matching, when it is unique.

Lemma 22. *Suppose $\mathcal{F}$ is a family of subsets of $\{1, 2, \ldots, m\}$ and $w_1, \ldots, w_m$ are mutually independent random variables taking values in the real numbers. Define the weight of each set in $\mathcal{F}$ to be the sum of the weights of its elements. Let δ be the maximum, over all $i \in [m]$ and $x \in \mathbb{R}$, of the probability that $w_i = x$. With probability at least $1 - m\delta$ the minimum weight set in $\mathcal{F}$ is unique.*

Proof. Let us say that an element $i \in \{1, 2, \ldots, m\}$ is *confused* if the minimum weight of a set in $\mathcal{F}$ containing i equals the minimum weight of a set in $\mathcal{F}$ not containing i. The minimum weight set is non-unique if and only if there exists at least one confused element, so the probability that the minimum weight set is non-unique is bounded above by the expected number of confused elements.

To bound the probability that element i is confused, we condition on the weights of all other elements besides i, and we show that the conditional probability that i is confused can never exceed δ. Indeed, once we have conditioned on the weights of all other elements, the minimum weight of a set in $\mathcal{F}$ not containing i is determined; call this weight W_0. Furthermore, the set of elements of a minimum-weight set set in $\mathcal{F}$ containing i is also determined; call this weight $W_1 + w_i$, where W_1 represents the combined weight of all elements other than i in this minimum-weight set. Then element i is confused if and only if $w_i = W_0 - W_1$, an event with probability at most δ.

Summing up over all elements $i \in [m]$, we find that the expected number of confused elements is at most $m\delta$, so the probability that there are no confused elements (and that the minimum-weight set is unique) is at least $1 - m\delta$. □

Now, we show how to reduce the problem of finding a unique minimum-weight perfect matching (if one exists) to computing $m+1$ matrix determinants in parallel. Sample weights $w(i,j)$ independently and uniformly at random from $\{0, 1, \ldots, b-1\}$ for each edge (u_i, v_j) of G. Consider the matrix A defined by setting $A_{ij} = 2^{w(i,j)}$ if $(u_i, v_j) \in E$ and $A_{ij} = 0$ otherwise. The determinant of A is a sum, over all perfect matchings M, of $\pm 2^{w(M)}$, where $w(M)$ denotes the sum of the weights of all edges in M. If M^* is the unique minimum-weight perfect matching and its weight is W, then the contribution of M^* to the determinant is congruent to 2^W modulo 2^{W+1}, and the contribution of every other perfect matching is divisible by 2^{W+1}.

Now, let $e = (u_i, v_j)$ be any edge of G, and let A_{-ij} denote the matrix obtained from A by deleting row i and column j. The determinant of A_{-ij} is the sum, over all perfect matchings M in G that contain e, of $\pm 2^{w(M)-w(i,j)}$. Hence, $2^{w(i,j)} \cdot \det(A_{-ij})$ is congruent to 2^W modulo 2^{W+1} if e belongs to M^*, and otherwise $2^{w(i,j)} \cdot \det(A_{-ij})$ is divisible by 2^{W+1}.

It is now clear how to design a parallel algorithm to find the edge set of a minimum-weight perfect matching, when there is a unique such matching. In parallel for each of the $m+1$ matrices in the set $\{A\} \cup \{A_{ij} \mid (u_i, v_j) \in E\}$, the algorithm computes the determinant of the matrix, and then it computes the largest power of 2 dividing the determinant. It outputs the set of all edges (u_i, v_j) such that the largest power of 2 dividing $\det(A_{-ij})$ equals the largest power of 2 dividing $\det(A)$. It is an easy exercise to design a parallel algorithm that computes the largest power of 2 dividing a d-bit binary number in $O(\log d)$ time. Given the way we defined A, the determinant of each matrix our algorithm handles can be expressed in d binary digits where $d \leq O(n \log n + nb)$, so $\log d = O(\log(nb))$.

We have presented a randomized parallel algorithm for finding a perfect matching in a bipartite graph that runs in time $O(\log(n)\log(nb))$ and succeeds with probability $1 - m/b$. Letting $b = m^{c+1}$ for any constant c, we obtain an algorithm with running time $O((c+1)\log^2(n))$ and success probability $1 - 1/m^c$.