CS261: A Second Course in Algorithms
Lecture #5: Minimum-Cost Bipartite Matching*

Tim Roughgarden†

January 19, 2016

1 Preliminaries

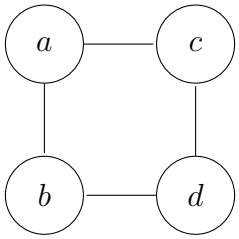

Figure 1: Example of bipartite graph. The edges $\{a, b\}$ and $\{c, d\}$ constitute a matching.

Last lecture introduced the maximum-cardinality bipartite matching problem. Recall that a bipartite graph $G = (V \cup W, E)$ is one whose vertices are split into two sets such that every edge has one endpoint in each set (no edges internal to V or W allowed). Recall that a matching is a subset $M \subseteq E$ of edges with no shared endpoints (e.g., Figure 1). Last lecture, we sketched a simple reduction from this problem to the maximum flow problem. Moreover, we deduced from this reduction and the max-flow/min-cut theorem a famous optimality condition for bipartite matchings. A special case is Hall's theorem, which states that a bipartite graph with $|V| \leq |W|$ has a perfect matching if and only if for every subset $S \subseteq V$ of the left-hand side, the number $|N(S)|$ of S on the right-hand side is at least $|S|$. See Problem Set #2 for quite good running time bounds for the problem.

But what if a bipartite graph has many perfect matchings? In applications, there are often reasons to prefer one over another. For example, when assigning jobs to works, perhaps there are many workers who can perform a particular job, but some of them are better at

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

it than others. The simplest way to model such preferences is attach a *cost* c_e to each edge $e \in E$ of the input bipartite graph $G = (V \cup W, E)$.

We also make three assumptions. These are for convenience, and are not crucial for any of our results.

1. The sets V and W have the same size, call it n. This assumption is easily enforced by adding "dummy vertices" (with no incident edges) to the smaller side.

2. The graph G has at least one perfect matching. This is easily enforced by adding "dummy edges" that have a very high cost (e.g., one such edge from the ith vertex of V to the ith vertex of W, for each i).

3. Edge costs are nonnegative. This can be enforced in the obvious way: if the most negative edge cost is $-M$, just add M to the cost of every edge. This adds the same number (nM) to every perfect matching, and thus does not change the problem.

The goal in the minimum-cost perfect bipartite matching problem is to compute the perfect matching M that minimizes $\sum_{e \in M} c_e$. The feasible solutions to the problem are the perfect matchings of G. An equivalent problem is the maximum-weight perfect bipartite matching problem (just multiply all weights by -1 to transform them into costs).

When every edge has the same cost and we only care about cardinality, the problem reduces to the maximum flow problem (Lecture #4). With general costs, there does not seem to be a natural reduction to the maximum flow problem. It's true that edges in a flow network come with attached numbers (their capacities), but there is a type mismatch: edge capacities affect the set of feasible solutions but not their objective function values, while edge costs do the opposite. Thus, the minimum-cost perfect bipartite matching problem seems like a new problem, for which we have to design an algorithm from scratch.

We'll follow the same kind of disciplined approach that served us so well in the maximum flow problem. First, we identify optimality conditions, which tell us when a given perfect matching is in fact minimum-cost. This step is structural, not algorithmic, and is analogous to our result in Lecture #2 that a flow is maximum if and only if there is no s-t path in the residual network. Then, we design an algorithm that can only terminate with the feasibility and optimality conditions satisfied. For maximum flow, we had one algorithmic paradigm that maintained feasibility and worked toward the optimality conditions (augmenting path algorithms), and a second paradigm that maintain the optimality conditions and worked toward feasibility (push-relabel). Here, we follow the second approach. We'll identify invariants that imply the optimality condition, and design an algorithm that keeps them satisfied at all times and works toward a feasible solution (i.e., a perfect matching).

2 Optimality Conditions

How do we know if a given perfect matching has the minimum-possible cost? Optimality conditions are different for different problems, but for the problems studied in CS261 they are

all quite natural in hindsight. We first need an analog of a residual network. This requires some definitions (see also Figure 2).

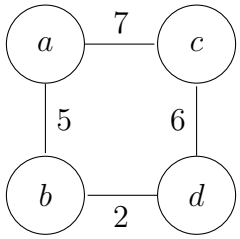

Figure 2: If our matching contains $\{a, b\}$ and $\{c, d\}$, then $a \to b \to d \to c \to a$ is both an M-alternating cycle and a negative cycle.

Definition 2.1 (Negative Cycle) Let M be a matching in the bipartite graph $G = (V \cup W, E)$.

(a) A cycle C of G is *M-alternating* if every other edge of C belongs to M (Figure 2).[1]

(b) An M-alternating cycle is *negative* if the edges in the matching have higher cost than those outside the matching:

$$\sum_{e \cup C \cap M} c_e > \sum_{e \in C \setminus M} c_e.$$

Otherwise, it is *nonnegative*.

C is the augmenting path…

One interesting thing about alternating cycles is that by "toggling" the edges of C with respect to M — that is, removing the edges of $C \cap M$ and plugging in the edges of $C \setminus M$ — yields a new matching M' that matches exactly the same set of vertices. (Vertices outside of C are clearly unaffected; vertices inside C remain matched to precisely one other vertex of C, just a different one than before.)

Suppose M is a perfect matching, and we toggle the edges of an M-alternating cycle to get another (perfect) matching M'. Dropping the edges from $C \cap M$ saves us a cost of $\sum_{e \cup C \cap M} c_e$, while adding the edges of $C \setminus M$ cost us $\sum_{e \in C \setminus M} c_e$. Then M' has smaller cost than M if and only if C is a negative cycle.

The point of a negative cycle is that it offers a quick and convincing proof that a perfect matching is not minimum-cost (since toggling the edges of the cycle yields a cheaper matching). But what about the converse? If a perfect matching is not minimum-cost, are we guaranteed such a short and convincing proof of this fact? Or are there "obstacles" to optimality beyond the obvious ones of negative cycles?

[1] Since G is bipartite, C is necessarily an even cycle. One certainly can't have more than every other edge of C contained in the matching M.

Theorem 2.2 (Optimality Conditions for Min-Cost Bipartite Matching) *A perfect matching of a bipartite graph has minimum-cost if and only if there is no negative M-alternating cycle.*

Proof: We have already argued the "only if" direction. For the harder "if" direction, suppose that M is a perfect matching and that there is no negative M-alternating cycle. Let M' be any other perfect matching; we want to show that the cost of M' is at least that of M. Consider $M \oplus M'$, meaning the symmetric difference of M, M' (if you want to think of them as sets) or their XOR (if you want to think of them as 0/1 vectors). See Figure 3 for two examples.

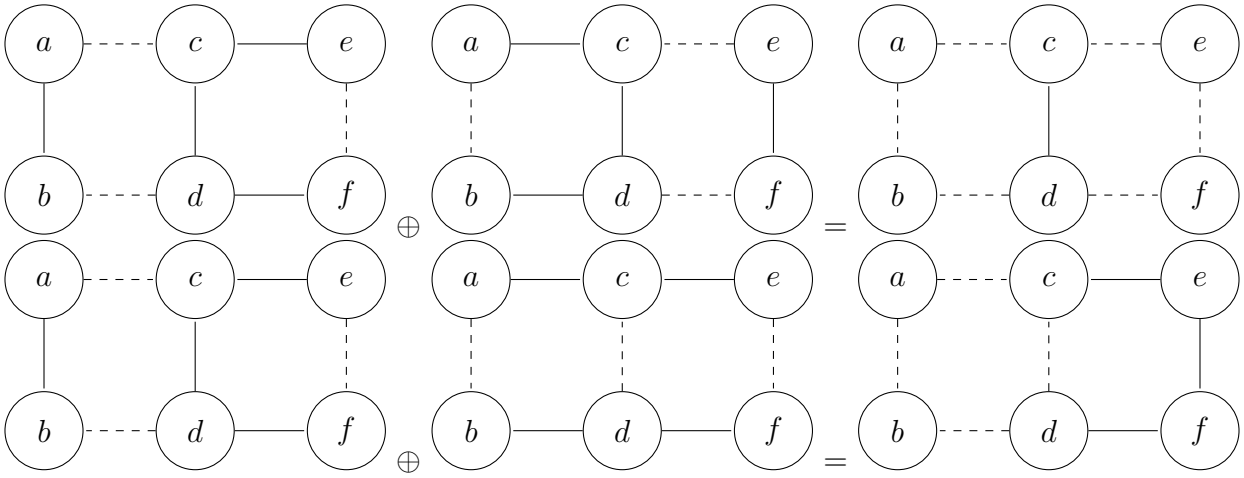

Figure 3: Two examples that show what happens when we XOR two matchings (the dashed edges).

In general, $M \oplus M'$ is a union of (vertex-)disjoint cycles. The reason is that, since every vertex has degree 1 in both M and M', every vertex of v has degree either 0 (if it is matched to the same vertex in both M and M') or 2 (otherwise). A graph with all degrees either 0 or 2 must be the union of disjoint cycles.

Since taking the symmetric difference/XOR with the same set two times in a row recovers the initial set, $(M \oplus M') \oplus M' = M$. Since $M \oplus M'$ is a disjoint union of cycles, taking the symmetric different/XOR with $M \oplus M'$ just means toggling the edges in each of its cycles (since they are disjoint, they don't interfere and the toggling can be done in parallel). Each of these cycles is M-alternating, and by assumption each is nonnegative. Thus toggling the edges of the cycles can only produce a more expensive perfect matching M'. Since M' was an arbitrary perfect matching, M must be a minimum-cost perfect matching. ■

3 Reduced Costs and Invariants

Now that we know when we're done, we work toward algorithms that terminate with the optimality conditions satisfied. Following the push-relabel approach (Lecture #3), we next identify invariants that will imply the optimality conditions at all times. Our algorithm will maintain these as it works toward a feasible solution (i.e., a perfect matching). Continuing the analogy with the push-relabel paradigm, we maintain a extra number p_v for each vertex $v \in V \cup W$, called a *price* (analogous to the "heights" in Lecture #3). Prices are allowed to be positive or negative. We use prices to force us to add edges to our current matching only in a disciplined way, somewhat analogous to how we only pushed flow "downhill" in Lecture #3.

Formally, for a price vector p (indexed by vertices), we define the *reduced cost* of an edge $e = (v, w)$ by

$$c_e^p = c_e - p_v - p_w. \tag{1}$$

Here are our invariants, which are respect to a current matching M and a current vector p of prices.

Invariants

1. Every edge of G has nonnegative reduced cost.
2. Every edge of M is *tight*, meaning it has zero reduced cost.

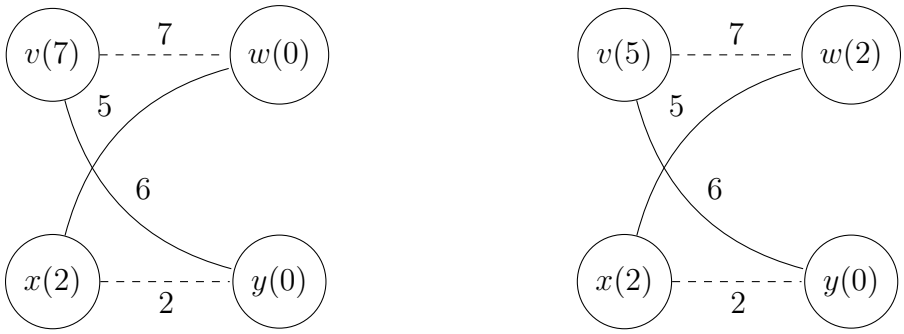

Figure 4: For the given (perfect) matching (dashed edges), (a) violates invariant 1, while (b) satisfies all invariants.

6 - 7 - 0 = -1 <0

For example, consider the (perfect) matching in Figure 4. Is it possible to define prices so that the invariants hold? To satisfy the second invariant, we need to make the edges (v, w) and (x, y) tight. We could try setting the price of w and y to 0, which then dictates setting $p_v = 7$ and $p_x = 2$ (Figure 4(a)). This violates the first invariant, however, since the reduced cost of edge (v, y) is -1. We can satisfy both invariants by resetting $p_v = 5$ and $p_w = 2$; then both edges in the matching are tight and the other two edges have reduced cost 1 (Figure 4(b)).

The matching in Figure 4 is a min-cost perfect matching. This is no coincidence.

Lemma 3.1 (Invariants Imply Optimality Condition) *If M is a perfect matching and both invariants hold, then M is a minimum-cost perfect matching.*

Proof: Let M be a perfect matching such that both invariants hold. By our optimality condition (Theorem 2.2), we just need to check that there is no negative cycle. So consider any M-alternating cycle C (remember a negative cycle must be M-alternating, by definition). We want to show that the edges of C that are in M have cost at most that of the edges of C not in M. Adding and subtracting $\sum_{v \in C} p_v$ and using the fact that every vertex of C is the endpoint of exactly one edge of $C \cap M$ and of $C \setminus M$ (e.g., Figure 5), we can write

$$\sum_{e \in C \cap M} c_e = \sum_{e \in C \cap M} c_e^p + \sum_{v \in C} p_v \tag{2}$$

and

$$\sum_{e \in C \setminus M} c_e = \sum_{e \in C \setminus M} c_e^p + \sum_{v \in C} p_v. \tag{3}$$

(We are abusing notation and using C both to denote the vertices in the cycle and the edges in the cycle; hopefully the meaning is always clear from context.) Clearly, the third terms in (2) and (3) are the same. By the second invariant (edges of M are tight), the second term in (2) is 0. By the first invariant (all edges have nonnegative reduced cost), the second term in (3) is at least 0. We conclude that the left-hand side of (2) is at most that of (3), which proves that C is not a negative cycle. Since C was arbitrary M-alternating cycle, the proof is complete. ■

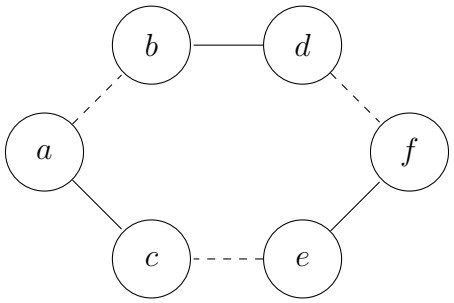

Figure 5: In the example M-alternating cycle and matching shown above, every vertex is an endpoint of exactly one edge in M and one edge not in M.

4 The Hungarian Algorithm

Lemma 3.1 reduces the problem of designing a correct algorithm for the minimum-cost perfect bipartite matching problem to that of designing an algorithm that maintains the two invariants and computes an arbitrary perfect matching. This section presents such an algorithm.

4.1 Backstory

The algorithm we present goes by various names, the two most common being the *Hungarian algorithm* and the *Kuhn-Munkres algorithm*. You might therefore find it weird that Kuhn and Munkres are American. Here's the story. In the early/mid-1950s, Kuhn really wanted an algorithm for solving the minimum-cost bipartite matching problem. So he was reading a graph theory book by Kőnig. This was actually the first graph theory book ever written — in the 1930s, and available in the U.S. only in 1950 (even then, only in German). Kuhn was intrigued by an offhand citation in the book, to a paper of Egerváry. Kuhn tracked down the paper, which was written in Hungarian. This was *way* before Google Translate, so he bought a big English-Hungarian dictionary and translated the whole thing. And indeed, Egerváry's paper had the key ideas necessary for a good algorithm. Kőnig and Egerváry were both Hungarian, so Kuhn called his algorithm the Hungarian algorithm. Kuhn only proved termination of his algorithm, and soon thereafter Munkres observed a polynomial time bound (basically the bound proved in this lecture). Hence, also called the Kuhn-Munkres algorithm.

In a (final?) twist to the story, in 2006 it was discovered that Jacobi, the famous mathematician (you've studied multiple concepts named after him in your math classes), came up with an equivalent algorithm in the 1840s! (Published only posthumously, in 1890.) Kuhn, then in his 80s, was a good sport about it, giving talks with the title "The Hungarian Algorithm and How Jacobi Beat Me By 100 Years."

4.2 The Algorithm: High-Level Structure

The Hungarian algorithm maintains both a matching M and prices p. The initialization is straightforward.

Initialization

set $M = \emptyset$
set $p_v = 0$ for all $v \in V \cup W$

The second invariant holds vacuously. The first invariant holds because we are assuming that all edge costs (and hence initial reduced costs) are nonnegative.

Informally (and way underspecified), the main loop works as follows. The terms "augment," "good path," and "good set" will be defined shortly.

Main Loop (High-Level)

while M is not a perfect matching **do**
    if there is a good path P **then**
        augment M by P
    else
        find a good set S; update prices accordingly

4.3 Good Paths

We now start filling in the details. Fix the current matching M and current prices p. Call a path P from v to w *good* if:

1. both endpoints v, w are unmatched in M, with $v \in V$ and $w \in W$ (hence P has odd length);
2. it alternates edges out of M with edges in M (since v, w are unmatched, the first and last edges are not in M);
3. every edge of P is tight (i.e., has zero reduced cost and hence eligible to be included in the current matching).

Figure 6 depicts a simple example of a good path.

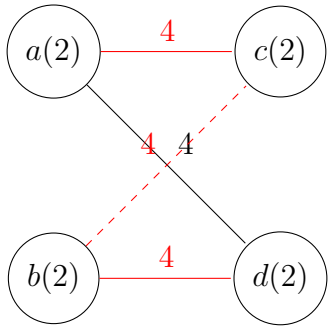

Figure 6: Dashed edges denote edges in the matching and red edges denote a good path.

The reason we care about good paths is that such a path allows us to increase the cardinality of M without breaking either invariant. Specifically, consider replacing M by $M' = M \oplus P$. This can be thought of as toggling which edges of P are in the current matching. By definition, a good path is M-alternating, with first and last hops not in M; thus, $|P \cap M| = |P \setminus M| - 1$, and the size of M' is one more than M. (E.g., if P is a 9-hop path, this toggling removes 4 edges from M but that adds in 5 other edges.) No reduced costs have changed, so certainly the first invariant still holds. All edges of P are tight be definition, so the second invariant also continues to hold.

Augmentation Step

given a good path P, replace M by $M \oplus P$

Finding a good path is definitely progress — after n such augmentations, the current matching M must be perfect and (since the invariants hold) we're done. How can we efficiently find such a path? And what do we do if there's no such path?

To efficiently search for such a path, let's just follow our nose. It turns out that breadth-first search (BFS), with a twist to enforce M-alternation, is all we need.

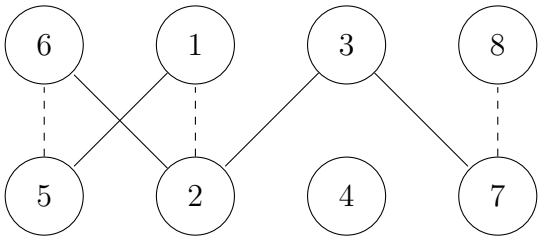

Figure 7: Dashed edges are the edges in the matching. Only tight edges are shown.

The algorithm will be clear from an example. Consider the graph in Figure 7; only the tight edges are shown. Note that the graph does not contain a good path (if it did, we could use it to augment the current matching to obtain a perfect matching, but vertex #4 is isolated so there is no perfect matching.).[2] So we know in advance that our search will fail. But it's useful to see what happens when it fails.

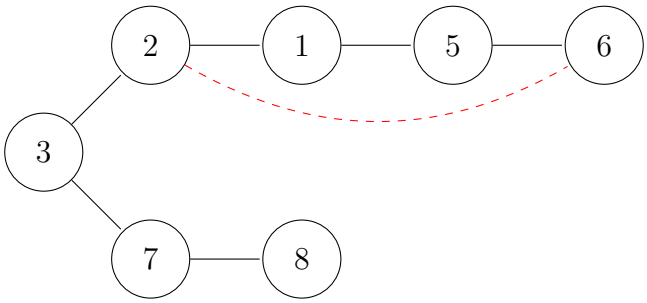

Figure 8: BFS spanning tree if we start BFS travel from node 3. Note that the edge $\{2, 6\}$ is not used.

We start a graph search from an unmatched vertex of V (the first such vertex, say); see also Figure 8. In the example, this is vertex #3. Layer 0 of our search tree is $\{3\}$. We obtain layer 1 from layer 0 by BFS; thus, layer 1 is $\{2, 7\}$. Note that if either 2 or 7 is unmatched, then we have found a (one-hop) good path and we can stop the search. Both 2 and 7 are already matched in the example, however. Here is the twist to BFS: at the next layer 2 we put only the vertices to which 2 and 7 are matched, namely 1 and 8. Conspicuous in its absence is vertex #6; in regular BFS it would be included in layer 2, but here we omit it because it is not matched to a vertex of layer 1. The reason for this twist is that we want every path in our search tree to be M-alternating (since good paths need to be M-alternating).

[2]Remember we assume only that G contains a perfect matching; the subgraph of tight edges at any given time will generally not contain a perfect matching.

We then switch back to BFS. At vertex #8 we're stuck (we've already seen its only neighbor, #7). At vertex #1, we've already seen its neighbor 2 but have not yet seen vertex #5, so the third layer is $\{5\}$. Note that if 5 were unmatched, we would have found a good path, from 5 back to the root 3. (All edges in the tree are tight by definition; the path is alternating and of odd length, joining two unmatched vertices of V and W.) But 5 is already matched to 6, so layer 4 of the search tree is $\{6\}$. We've already seen both of 6's neighbors before, so at this point we're stuck and the search terminates.

In general, here is the search procedure for finding a good path (given a current matching M and prices p).

Searching for a Good Path

```
level 0 = the first unmatched vertex r of V
while not stuck and no other unmatched vertex found do
    if next level i is odd then
        define level i from level i − 1 via BFS
        // i.e., neighbors of level i − 1 not already seen
    else if next level i is even then
        define level i as the vertices matched in M to vertices at
         level i − 1
if found another unmatched vertex w then
    return the search tree path between the root r and w
else
    return "stuck"
```

To understand this subroutine, consider an edge $(v, w) \in M$, and suppose that v is reached first, at level i. Importantly, it is not possible that w is also reached at level i. This is where we use the assumption that G is bipartite: if v, w are reached in the same level, then pasting together the paths from r to v and from r to w (which have the same length) with the edge (v, w) exhibits an odd cycle, contradicting bipartiteness. Second, we claim that i must be odd (cf., Figure 8). The reason is just that, by construction, every vertex at an even level (other than 0) is the second endpoint reached of some matched edge (and hence cannot be the endpoint of any other matched edge). We conclude that:

(*) if either endpoint of an edge of M is reached in the search tree, then both endpoints are reached, and they appear at consecutive levels $i, i+1$ with i odd.

Suppose the search tree reaches an unmatched vertex w other than the root r. Since every vertex at an even level (after 0) is matched to a vertex at the previous level, w must be at an odd level (and hence in W). By construction, every edge of the search tree is tight, and every path in the tree is M-alternating. Thus the r-w path in the search tree is a good path, allowing us to increase the size of M by 1.

4.4 Good Sets

Suppose the search gets stuck, as in our example. How do we make progress, and in what sense? In this case, we keep the matching the same but update the prices.

Define $S \subseteq V$ as the vertices at even levels. Define $N(S) \subseteq S$ as the neighbors of S via tight edges, i.e.,

$$N(S) = \{w : \exists v \in S \text{ with } (v, w) \text{ tight}\}. \tag{4}$$

We claim that $N(S)$ is precisely the vertices that appear in the odd levels of the search tree. In proof, first note that every vertex at an odd level is (by construction/BFS) adjacent via a tight edge to a vertex at the previous (even) level. For the converse, every vertex $w \in N(S)$ must be reached in the search, because (by basic properties of graph search) the search can only stuck if there are no unexplored edges out of any even vertex.

The set S is a *good set*, in that is satisfies:

1. S contains an unmatched vertex;
2. every vertex of $N(S)$ is matched in M to a vertex of S (since the search failed, every vertex in an odd level is matched to some vertex at the next (even) level).

See also Figure 9.

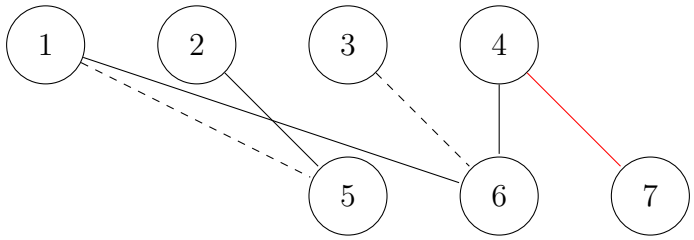

Figure 9: $S = \{1, 2, 3, 4\}$ is example of good set, with $N(S) = \{5, 6\}$. Only black edges are tight edges (i.e. $(4, 7)$ is not tight). The matching edges are dashed.

Having found such a good set S, the Hungarian algorithm updates prices as follows.

Price Update Step

given a good set S, with neighbors via tight edges $N(S)$
for all $v \in S$ **do**
    increase p_v by Δ
for all $w \in N(S)$ **do**
    decrease p_v by Δ
`// Δ is as large as possible, subject to invariants`

Prices in S (on the left-hand side) are increased, while prices in $N(S)$ (on the right-hand side) are decreased by the same amount. How does this affect the reduced cost of each edge of G (Figure 9)?

1. for an edge (v, w) with $v \notin S$ and $w \notin N(S)$, the prices of v, w are unchanged so c^p_{vw} is unchanged;

2. for an edge (v, w) with $v \in S$ and $w \in N(S)$, the sum of the prices of v, w is unchanged (one increased by Δ, the other decreased by Δ) so c^p_{vw} is unchanged;

3. for an edge (v, w) with $v \notin S$ and $w \in N(S)$, p_v stays the same while p_w goes down by Δ, so c^p_{vw} goes up by Δ;

4. for an edge (v, w) with $v \in S$ and $w \notin N(S)$, p_w stays the same while p_v goes up by Δ, so c^p_{vw} goes down by Δ.

So what happens with the invariants? Recalling (*) from Section 4.3, we see that edges of M are in either the first or second category. Thus they stay tight, and the second invariant remains satisfied. The first invariant is endangered by edges in the fourth category, whose reduced costs are dropping with Δ.[3] By the definition of $N(S)$, edges in this category are not tight. So we increase Δ to the largest-possible value subject to the first invariant — the first point at which the reduced cost of some edge in the fourth category is zeroed out.[4]

Every price update makes progress, in the sense that it strictly increases the size of search tree. To see this, suppose a price update causes the edge (v, w) to become tight (with $v \in S$, $w \notin N(S)$). What happens in the next iteration, when we search from the same vertex r for a good path? All edges in the previous search tree fall in the second category, and hence are again tight in the next iteration. Thus, the search procedure will regrow exactly the same search tree as before, will again reach the vertex v, and now will also explore along the newly tight edge (v, w), which adds the additional vertex $w \in W$ to the tree. This can only happen n times in a row before finding a good path, since there are only n vertices in W.

[3] Edges in third category might go from tight to non-tight, but these edges are not in M (every vertex of $N(S)$ is matched to a vertex of S) and so no invariant is violated.

[4] A detail: how do we know that such an edge exists? If not, then all neighbors of S in G (via tight edges or not) belong to $N(S)$. The two properties of good sets imply that $|N(S)| < |S|$. But this violates Hall's condition for perfect matchings (Lecture #4), contradicting our standing assumption that G has at least one perfect matching.

4.5 The Hungarian Algorithm (All in One Place)

The Hungarian Algorithm

```
set M = ∅
set p_v = 0 for all v ∈ V ∪ W
while M is not a perfect matching do
    level 0 of search tree T = the first unmatched vertex r of V
    while not stuck and no other unmatched vertex found do
        if next level i is odd then
            define level i of T from level i − 1 via BFS
            // i.e., neighbors of level i − 1 not already seen
        else if next level i is even then
            define level i of T as the vertices matched in M to vertices at
             level i − 1
    if T contains an unmatched vertex w ∈ W then
        let P denote the r-w path in T
        replace M by M ⊕ P
    else
        let S denote the vertices of T in even levels
        let N(S) denote the vertices of T in odd levels
        for all v ∈ S do
            increase p_v by Δ
        for all w ∈ N(S) do
            decrease p_v by Δ
        // Δ is as large as possible, subject to invariants
return M
```

4.6 Running Time

Since M can only contain n edges, there can only be n iterations that find a good path. Since the search tree can only contain n vertices of W, there can only be n prices updates between iterations that find good paths. Computing the search tree (and hence P or S and $N(S)$) and Δ (if necessary) can be done in $O(m)$ time. This gives a running time bound of $O(mn^2)$. See Problem Set #2 for an implementation with running time $O(mn \log n)$.

4.7 Example

We reinforce the algorithm via an example. Consider the graph in Figure 10.

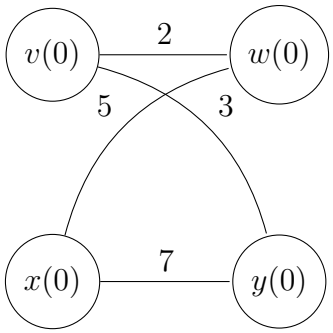

Figure 10: Example graph. Initially, all prices are 0.

We initialize all prices to 0 and the current matching to the empty set. Initially, there are no tight edges, so there is certainly no good path. The search for such a path gets stuck where it starts, at vertex v. So $S = \{v\}$ and $N(S) = \emptyset$. We execute a price update step, raising the price of v to 2, at which point the edge (v, w) becomes tight. Next iteration, the search starts at v, explores the tight edge (v, w), and encounters vertex w, which is unmatched. Thus this edge is added to the current matching. Next iteration, a new search starts from the only remaining unmatched vertex on the left (x). It has no tight incident edges, so the search gets stuck immediately, with $S = \{x\}$ and $N(S) = \emptyset$. We thus do a price update step, with $\Delta = 5$, at which point the edge (x, w) becomes newly tight. Note that the edges (v, y) and (x, y) have reduced costs 1 and 2, respectively, so neither is tight. Next iteration, the search from x explores the incident tight edge (x, w). If w were unmatched, we could stop the search and add the edge (x, w). But w is already matched, to v, so w and v are placed at levels 1 and 2 of the search tree. v has no tight incident edges other than to w, so the search gets stuck here, with $S = \{x, v\}$ and $N(S) = \{w\}$. So we do another a price update step, increasing the price of x and v by Δ and decreasing the price of w by Δ. With $\Delta = 1$, the reduced cost of edge (v, y) gets zeroed out. The final iteration discovers the good path $x \to w \to v \to y$. Augmenting on this path yields the minimum-cost perfect matching $\{(v, y), (x, w)\}$.