Original notes by Michael Tan.

11 Matchings

Read: Chapter 9 [21]. Chapter 10 [19].

In this lecture we will examine the problem of finding a maximum matching in bipartite graphs.

Given a graph $G = (V, E)$, a **matching** M is a subset of the edges such that no two edges in M share an endpoint. The problem is similar to finding an independent set of edges. In the maximum matching problem we wish to maximize $|M|$.

With respect to a given matching, a **matched edge** is an edge included in the matching. A **free edge** is an edge which does not appear in the matching. Likewise, a **matched vertex** is a vertex which is an endpoint of a matched edge. A **free vertex** is a vertex that is not the endpoint of any matched edge.

A **bipartite** graph $G = (U, V, E)$ has $E \subseteq U \times V$. For bipartite graphs, we can think of the matching problem in the following terms. Given a list of boys and girls, and a list of all marriage compatible pairs (a pair is a boy and a girl), a matching is some subset of the compatibility list in which each boy or girl gets at most one partner. In these terms, $E = \{$ all marriage compatible pairs $\}$, $U = \{$ the boys$\}$, $V = \{$ the girls$\}$, and $M = \{$ some potential pairing preventing polygamy$\}$.

A **perfect matching** is one in which all vertices are matched. In bipartite graphs, we must have $|V| = |U|$ in order for a perfect matching to possibly exist. When a bipartite graph has a perfect matching in it, the following theorem holds:

11.1 Hall's Theorem

Theorem 11.1 (Hall's Theorem) *Given a bipartite graph $G = (U, V, E)$ where $|U| = |V|$, $\forall S \subseteq U, |N(S)| \geq |S|$ (where N(S) is the set of vertices which are neighbors of S) iff G has a perfect matching.*

Proof:

($\leftarrow$) In a perfect matching, all elements of S will have at least a total of $|S|$ neighbors since every element will have a partner. ($\rightarrow$) We give this proof after the presentation of the algorithm, for the sake of clarity. □

For non-bipartite graphs, this theorem does not work. Tutte proved the following theorem (you can read the Graph Theory book by Bondy and Murty for a proof) for establishing the conditions for the existence of a perfect matching in a non-bipartite graph.

Theorem 11.2 (Tutte's Theorem) *A graph G has a perfect matching if and only if $\forall S \subseteq V, o(G - S) \leq |S|$. ($o(G - S)$ is the* **number** *of connected components in the graph $G - S$ that have an odd number of vertices.)*

Before proceeding with the algorithm, we need to define more terms.

An **alternating path** is a path whose edges alternate between matched and unmatched edges.

An **augmenting path** is an alternating path which starts and ends with unmatched vertex (and thus starts and ends with a free edge).

The matching algorithm will attempt to increase the size of a matching by finding an augmenting path. By inverting the edges of the path (matched becomes unmatched and vice versa), we increase the size of a matching by exactly one.

If we have a matching M and an augmenting path P (with respect to M), then $M \oplus P = (M - P) \cup (P - M)$ is a matching of size $|M| + 1$.

11.2 Berge's Theorem

Theorem 11.3 (Berge's Theorem) *M is a maximum matching iff there are no augmenting paths with respect to M.*

Proof:
($\rightarrow$) Trivial. ($\leftarrow$) Let us prove the contrapositive. Assume M is not a maximum matching. Then there exists some maximum matching M' and $|M'| > |M|$. Consider $M \oplus M'$. All of the following will be true of the graph $M \oplus M'$:

1. The highest degree of any node is two.
2. The graph is a collection of cycles and paths.
3. The cycles must be of even length, with half the edges from M and half from M'.
4. Given these first three facts (and since $|M'| > |M|$), there must be some path with more M' edges than M edges.

This fourth fact describes an augmenting path (with respect to M). This path begins and ends with M' edges, which implies that the path begins and ends with free nodes (i.e., free in M). □

Armed with this theorem, we can outline a primitive algorithm to solve the maximum matching problem. It should be noted that Edmonds (1965) was the first one to show how to find an augmenting path in an arbitrary graph (with respect to the current matching) in polynomial time. This is a landmark paper that also defined the notion of polynomial time computability.
Simple Matching Algorithm [Edmonds]:

Step 1: Start with $M = \emptyset$.

Step 2: Search for an augmenting path.

Step 3: Increase M by 1 (using the augmenting path).

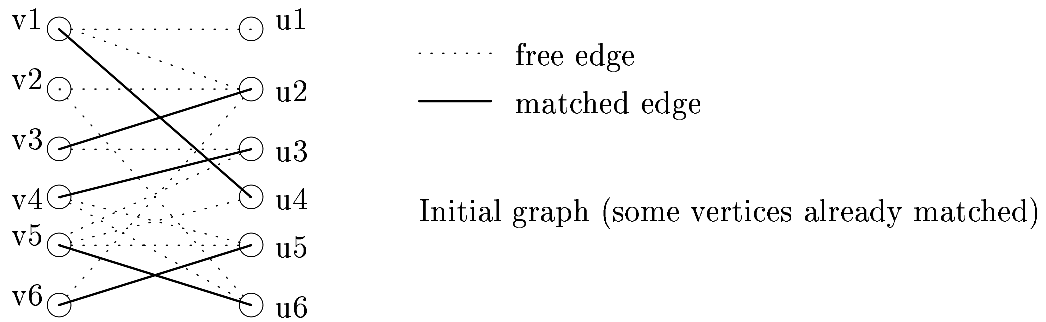
Step 4: Go to 2.

We will discuss the search for an augmenting path in bipartite graphs via a simple example (see Fig. 13). See [19] for a detailed description of the pseudo-code for this algorithm.

Theorem 11.4 (Hall's Theorem) *A bipartite graph $G = (U, V, E)$ has a perfect matching if and only if $\forall S \subseteq V, |S| \leq |N(S)|$.*

Proof:
To prove this theorem in one direction is trivial. If G has a perfect matching M, then for any subset S, $N(S)$ will always contain all the `mates` (in M) of vertices in S. Thus $|N(S)| \geq |S|$. The proof in the other direction can be done as follows. Assume that M is a maximum matching and is not perfect. Let u be a free vertex in U. Let Z be the set of vertices connected to u by alternating paths w.r.t M. Clearly u is the only free vertex in Z (else we would have an augmenting path). Let $S = Z \cap U$ and $T = Z \cap V$. Clearly the vertices in $S - \{u\}$ are matched with the vertices in T. Hence $|T| = |S| - 1$. In fact, we have $N(S) = T$ since every vertex in $N(S)$ is connected to u by an alternating path. This implies that $|N(S)| < |S|$. □

The upper bound on the number of iterations is $O(|V|)$ (the size of a matching). The time to find an augmenting path is $O(|E|)$ (use BFS). This gives us a total time of $O(|V||E|)$.
In the next lecture, we will learn the Hopcroft-Karp $O(\sqrt{|V|}|E|)$ algorithm for maximum matching on a bipartite graph. This algorithm was discovered in the early seventies. In 1981, Micali-Vazirani extended this algorithm to general graphs. This algorithm is quite complicated, but has the same running time as the Hopcroft-Karp method. It is also worth pointing out that both Micali and Vazirani were first year graduate students at the time.



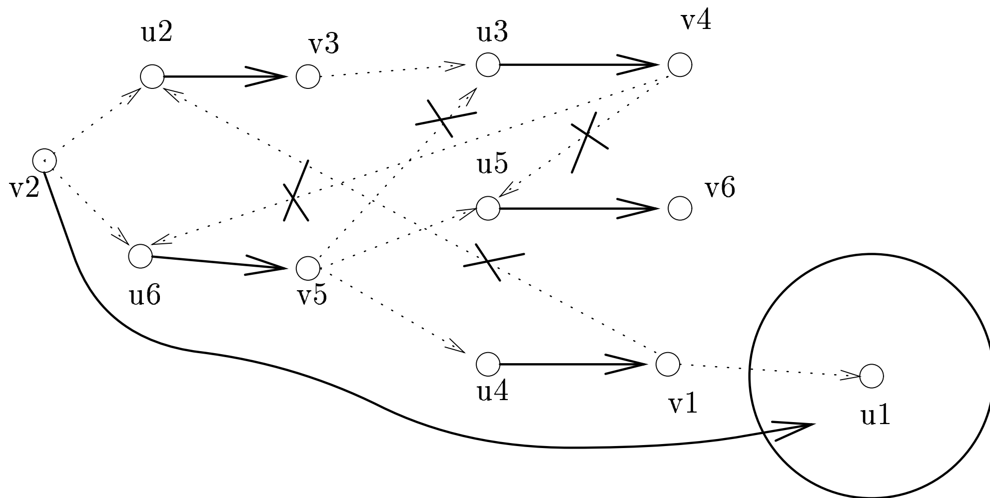


Figure 10: Sample execution of Simple Matching Algorithm.

Notes by Samir Khuller.

12 Hopcroft-Karp Matching Algorithm

The original paper is by Hopcroft and Karp [SIAM J. on Computing, 1973].

We present the Hopcroft-Karp matching algorithm that finds a maximum matching in bipartite graphs.

The main idea behind the Hopcroft-Karp algorithm is to augment along a *set* of shortest augmenting paths *simultaneously.* (In particular, if the shortest augmenting path has length k then in a single phase we obtain a *maximal* set S of vertex disjoint augmenting paths all of length k.) By the maximality property, we have that *any* augmenting path of length k will intersect a path in S. In the next phase, we will have the property that the augmenting paths found will be strictly longer (we will prove this formally later). We will implement each phase in linear time.

We first prove the following lemma.

Lemma 12.1 *Let M be a matching and P a shortest augmenting path w.r.t M. Let P' be a shortest augmenting path w.r.t $M \oplus P$ (symmetric difference of M and P). We have*

$$|P'| \geq |P| + 2|P \cap P'|.$$

$|P \cap P'|$ refers to the edges that are common to both the paths.

Proof:

Let $N = (M \oplus P) \oplus P'$. Thus $N \oplus M = P \oplus P'$. Consider $P \oplus P'$. This is a collection of cycles and paths (since it is the same as $M \oplus N$). The cycles are all of even length. The paths may be of odd or even length. The odd length paths are augmenting paths w.r.t M. Since the two matchings differ in cardinality by 2, there must be two odd length augmenting paths P_1 and P_2 w.r.t M. Both of these must be longer than P (since P was the shortest augmenting path w.r.t M).

$$|M \oplus N| = |P \oplus P'| = |P| + |P'| - 2|P \cap P'| \geq |P_1| + |P_2| \geq 2|P|.$$

Simplifying we get

$$|P'| \geq |P| + 2|P \cap P'|.$$

□

We still need to argue that after each phase, the shortest augmenting path is strictly longer than the shortest augmenting paths of the previous phase. (Since the augmenting paths always have an odd length, they must actually increase in length by two.)

Suppose that in some phase we augmented the current matching by a maximal set of vertex disjoint paths of length k (and k was the length of the shortest possible augmenting path). This yields a new matching M'. Let P' be an augmenting path with respect to the new matching. If P' is vertex disjoint from each path in the previous set of paths it must have length strictly more than k. If it shares a vertex with some path P in our chosen set of paths, then $P \cap P'$ contains at least one edge in M' since every vertex in P is matched in M'. (Hence P' cannot intersect P on a vertex and not share an edge with P.) By the previous lemma, P' exceeds the length of P by at least two.

Lemma 12.2 *A maximal set S of disjoint, minimum length augmenting paths can be found in $O(m)$ time.*

Proof:

Let $G = (U, V, E)$ be a bipartite graph and let M be a matching in G. We will grow a "Hungarian Tree" in G. (The tree really is not a tree but we will call it a tree all the same.) The procedure is similar to BFS and very similar to the search for an augmenting path that was done in the previous lecture. We start by putting the free vertices in U in level 0. Starting from even level $2k$, the vertices at level $2k + 1$ are obtained

by following free edges from vertices at level $2k$ that have not been put in any level as yet. Since the graph is bipartite the odd(even) levels contain only vertices from $V(U)$. From each odd level $2k+1$, we simple add the matched neighbours of the vertices to level $2k+2$. We repeat this process until we encounter a free vertex in an odd level (say t). We continue the search only to discover all free vertices in level t, and stop when we have found all such vertices. In this procedure clearly each edge is traversed at most once, and the time is $O(m)$.

We now have a second phase in which the maximal set of disjoint augmenting paths of length k is found. We use a technique called *topological erase*, called so because it is a little like topological sort. With *each* vertex x (except the ones at level 0), we associate an integer counter initially containing the number of edges entering x from the previous level (we also call this the indegree of the vertex). Starting at a free vertex v at the last level t, we trace a path back until arriving at a free vertex in level 0. The path is an augmenting path, and we include it in S.

At all points of time we maintain the invariant that there is a path from each free node (in V) in the last level to some free node in level 0. This is clearly true initially, since each free node in the last level was discovered by doing a BFS from free nodes in level 0. When we find an augmenting path we place all vertices along this path on a deletion queue. As long as the deletion queue is non-empty, we remove a vertex from the queue, delete it together with its adjacent vertices in the Hungarian tree. Whenever an edge is deleted, the counter associated with its right endpoint is decremented if the right endpoint is not on the current path (since this node is losing an edge entering it from the left). If the counter becomes 0, the vertex is placed on the deletion queue (there can be no augmenting path in the Hungarian tree through this vertex, since all its incoming edges have been deleted). This maintains the invariant that when we grab paths coming backwards from the final level to level 0, then we automatically delete nodes that have no paths backwards to free nodes.

After the queue becomes empty, if there is still a free vertex v at level t, then there must be a path from v backwards through the Hungarian tree to a free vertex on the first level; so we can repeat this process. We continue as long as there exist free vertices at level t. The entire process takes linear time, since the amount of work is proportional to the number of edges deleted. □

Let's go through the following example (see Fig. 11) to make sure we understand what's going on. Start with the first free vertex v_6. We walk back to u_6 then to v_5 and to u_1 (at this point we had a choice of u_5 or u_1). We place all these nodes on the deletion queue, and start removing them and their incident edges. v_6 is the first to go, then u_6 then v_5 and then u_1. When we delete u_1 we delete the edge (u_1, v_1) and decrease the indegree of v_1 from 2 to 1. We also delete the edges (v_6, u_3) and (v_5, u_5) but these do not decrease the indegree of any node, since the right endpoint is already on the current path.

Start with the next free vertex v_3. We walk back to u_3 then to v_1 and to u_2. We place all these nodes on the deletion queue, and start removing them and their incident edges. v_3 is the first to go, then u_3 (this deletes edge (u_3, v_4) and decreases the indegree of v_4 from 2 to 1). Next we delete v_1 and u_2. When we delete u_2 we remove the edge (u_2, v_2) and this decreases the indegree of v_2 which goes to 0. Hence v_2 gets added to the deletion queue. Doing this makes the edge (v_2, u_4) get deleted, and drops the indegree of u_4 to 0. We then delete u_4 and the edge (u_4, v_4) is deleted and v_4 is removed. There are no more free nodes in the last level, so we stop. The key point is that it is not essential to find the maximum set of disjoint augmenting paths of length k, but only a maximal set.

Theorem 12.3 *The total running time of the above described algorithm is $O(\sqrt{n}m)$.*

Proof:

Each phase runs in $O(m)$ time. We now show that there are $O(\sqrt{n})$ phases. Consider running the algorithm for exactly $\sqrt{n}$ phases. Let the obtained matching be M. Each augmenting path from now on is of length at least $2\sqrt{n}+1$. (The paths are always odd in length and always increase by at least two after each phase.) Let M^* be the max matching. Consider the symmetric difference of M and M^*. This is a collection of cycles and paths, that contain the augmenting paths w.r.t M. Let $k = |M^*| - |M|$. Thus there are k augmenting paths w.r.t M that yield the matching M^*. Each path has length at least $2\sqrt{n}+1$, and they are disjoint. The total length of the paths is at most n (due to the disjointness). If l_i is the length of each augmenting path we have:

$$k(2\sqrt{n}+1) \leq \sum_{i=1}^{k} l_i \leq n.$$

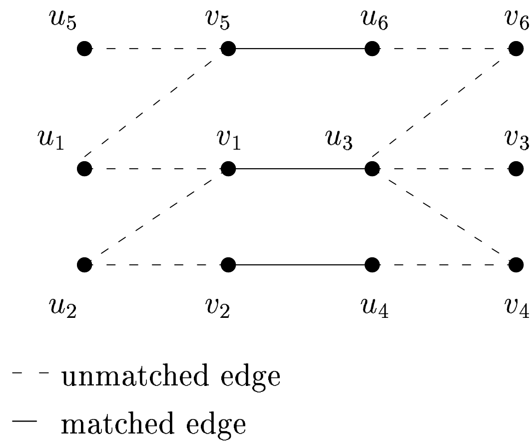


Figure 11: Sample execution of Topological Erase.

Thus k is upper bounded by $\frac{n}{2\sqrt{n}+1} \leq \frac{n}{2\sqrt{n}} \leq \frac{\sqrt{n}}{2}$. In each phase we increase the size of the matching by at least one, so there are at most k more phases. This proves the required bound of $O(\sqrt{n})$. $\square$

CMSC 651 Advanced Algorithms	Lecture 4
Lecturer: Samir Khuller	Th. Feb 7. 2002

Notes by Samir Khuller.

13 Two Processor Scheduling

We will also talk about an application of matching, namely the problem of scheduling on two processors. The original paper is by Fujii, Kasami and Ninomiya [7].

There are two identical processors, and a collection of n jobs that need to be scheduled. Each job requires unit time. There is a precedence graph associated with the jobs (also called the DAG). If there is an edge from i to j then i must be finished before j is started by either processor. How should the jobs be scheduled on the two processors so that all the jobs are completed as quickly as possible. The jobs could represent courses that need to be taken (courses have prerequisites) and if we are taking at most two courses in each semester, the question really is: how quickly can we graduate?

This is a good time to note that even though the two processor scheduling problem can be solved in polynomial time, the three processor scheduling problem is not known to be solvable in polynomial time, or known to be NP-complete. In fact, the complexity of the k processor scheduling problem when k is *fixed* is not known.

From the acyclic graph G we can construct a *compatibility* graph G^* as follows. G^* has the same nodes as G, and there is an (undirected) edge (i, j) if there is no directed path from i to j, or from j to i in G. In other words, i and j are jobs that can be done together.

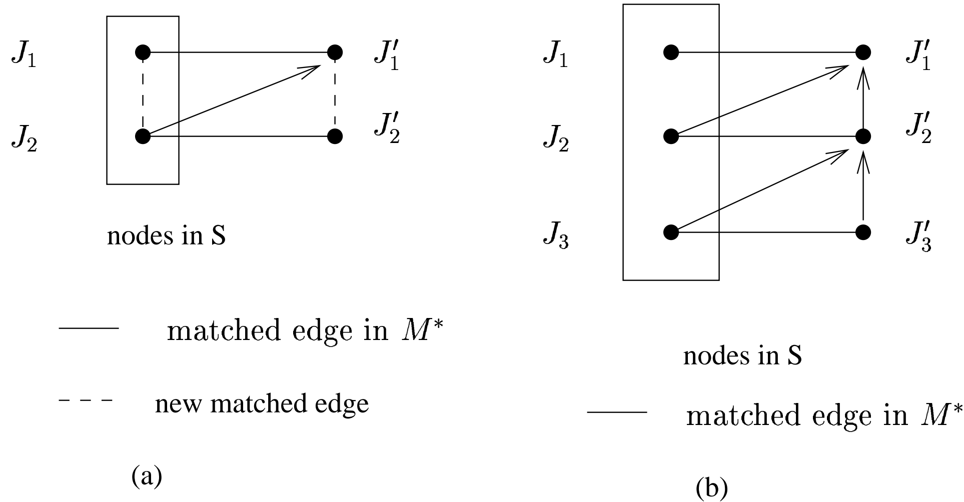


Figure 12: (a) Changing the schedule (b) Continuing the proof.

A maximum matching in G^* is the indicates the maximum number of pairs of jobs that can be processed simultaneously. Clearly, a solution to the scheduling problem can be used to obtain a matching in G^*. More interestingly, a solution to the matching problem can be used to obtain a solution to the scheduling problem!

Suppose we find a maximum matching in M in G^*. An unmatched vertex is executed on a single processor while the other processor is idle. If the maximum matching has size m^*, then $2m^*$ vertices are matched, and $n - 2m^*$ vertices are left unmatched. The time to finish all the jobs will be $m^* + n - 2m^* = n - m^*$. Hence a maximum matching will minimize the time required to schedule all the jobs.

We now argue that given a maximum matching M^*, we can extract a schedule of size $n - m^*$. The key idea is that each matched job is scheduled in a slot together with another job (which may be different from the job it was matched to). This ensures that we do not use more than $n - m^*$ slots.

Let S be the subset of vertices that have indegree 0. The following cases show how a schedule can be constructed from G and M^*. Basically, we have to make sure that for all jobs that are paired up in M^*, we pair them up in the schedule. The following rules can be applied repeatedly.

1. If there is an unmatched vertex in S, schedule it and remove it from G.
2. If there is a pair of jobs in S that are matched in M^*, then we schedule the pair and delete both the jobs from G.

If none of the above rules are applicable, then all jobs in S are matched; moreover each job in S is matched with a job not in S.

Let $J_1 \in S$ be matched to $J_1' \notin S$. Let J_2 be a job in S that has a path to J_1'. Let J_2' be the mate of J_2. If there is path from J_1' to J_2', then J_2 and J_2' could not be matched to each other in G^* (this cannot be an edge in the compatibility graph). If there is no path from J_2' to J_1' then we can *change* the matching to match (J_1, J_2) and (J_1', J_2') since (J_1', J_2') is an edge in G^* (see Fig. 12 (a)).

The only remaining case is when J_2' has a path to J_1'. Recall that J_2 also has a path to J_1'. We repeat the above argument considering J_2' in place of J_1'. This will yield a new pair (J_3, J_3') etc (see Fig. 12 (b)). Since the graph G has no cycles at each step we will generate a distinct pair of jobs; at some point of time this process will stop (since the graph is finite). At that time we will find a pair of jobs in S we can schedule together.

CMSC 651 Advanced Algorithms	Lecture 7
Lecturer: Samir Khuller	Tu. Feb 19, 2002

Notes by Samir Khuller.

14 Assignment Problem

Consider a complete bipartite graph, $G(X, Y, X \times Y)$, with weights $w(e_i)$ assigned to every edge. (One could think of this problem as modeling a situation where the set X represents workers, and the set Y represents jobs. The weight of an edge represents the "compatability" factor for a (worker,job) pair. We need to assign workers to jobs such that each worker is assigned to exactly one job.) The **Assignment Problem** is to find a matching with the greatest total weight, i.e., the maximum-weighted perfect matching (which is not necessarily unique). Since G is a complete bipartite graph we know that it has a perfect matching.

An algorithm which solves the Assignment Problem is due to Kuhn and Munkres. We assume that all the edge weights are non-negative,

$$w(x_i, y_j) \geq 0.$$

where

$$x_i \in X, y_j \in Y.$$

We define a *feasible vertex labeling* l as a mapping from the set of vertices in G to the real numbers, where

$$l(x_i) + l(y_j) \geq w(x_i, y_j).$$

(The real number $l(v)$ is called the label of the vertex v.) It is easy to compute a feasible vertex labeling as follows:

$$(\forall y_j \in Y) \quad [l(y_j) = 0].$$

and

$$l(x_i) = \max_j w(x_i, y_j).$$

We define the **Equality Subgraph**, G_l, to be the spanning subgraph of G which includes all vertices of G but only those edges (x_i, y_j) which have weights such that

$$w(x_i, y_j) = l(x_i) + l(y_j).$$

The connection between equality subgraphs and maximum-weighted matchings is provided by the following theorem:

Theorem 14.1 *If the Equality Subgraph, G_l, has a perfect matching, M^*, then M^* is a maximum-weighted matching in G.*

Proof:

Let M^* be a perfect matching in G_l. We have, by definition,

$$w(M^*) = \sum_{e \in M^*} w(e) = \sum_{v \in X \cup Y} l(v).$$

Let M be any perfect matching in G. Then

$$w(M) = \sum_{e \in M} w(e) \leq \sum_{v \in X \cup Y} l(v) = w(M^*).$$

Hence,

$$w(M) \leq w(M^*).$$

□

In fact note that the sum of the labels is an upper bound on the weight of the maximum weight perfect matching.

High-level Description:

The above theorem is the basis of an algorithm, due to Kuhn and Munkres, for finding a maximum-weighted matching in a complete bipartite graph. Starting with a feasible labeling, we compute the equality subgraph and then find a maximum matching in this subgraph (now we can ignore weights on edges). If the matching found is perfect, we are done. If the matching is not perfect, we add more edges to the equality subgraph by revising the vertex labels. We also ensure that edges from our current matching do not leave the equality subgraph. After adding edges to the equality subgraph, either the size of the matching goes up (we find an augmenting path), or we continue to grow the hungarian tree. In the former case, the phase terminates and we start a new phase (since the matching size has gone up). In the latter case, we grow the hungarian tree by adding new nodes to it, and clearly this cannot happen more than n times.

Let S = the set of free nodes in X. Grow hungarian trees from each node in S. Let T = all nodes in Y encountered in the search for an augmenting path from nodes in S. Add all nodes from X that are encountered in the search to S.

Some More Details:

We note the following about this algorithm:

$$\overline{S} = X - S.$$

$$\overline{T} = Y - T.$$

$$|S| > |T|.$$

There are no edges from S to $\overline{T}$, since this would imply that we did not grow the hungarian trees completely. As we grow the Hungarian Trees in G_l, we place alternate nodes in the search into S and T. To revise the labels we take the labels in S and start decreasing them uniformly (say by λ), and at the same time we increase the labels in T by λ. This ensures that the edges from S to T do not leave the equality subgraph (see Fig. 13).

As the labels in S are decreased, edges (in G) from S to $\overline{T}$ will potentially enter the Equality Subgraph, G_l. As we increase λ, at some point of time, an edge enters the equality subgraph. This is when we stop and update the hungarian tree. If the node from $\overline{T}$ added to G_l is matched to a node in $\overline{S}$, we move both these nodes to S and T, which yields a larger Hungarian Tree. If the node from $\overline{T}$ is free, we have found an augmenting path and the phase is complete. One phase consists of those steps taken between increases in the size of the matching. There are at most n phases, where n is the number of vertices in G (since in each phase the size of the matching increases by 1). Within each phase we increase the size of the hungarian tree at most n times. It is clear that in $O(n^2)$ time we can figure out which edge from S to $\overline{T}$ is the first one to enter the equality subgraph (we simply scan all the edges). This yields an $O(n^4)$ bound on the total running time. Let us first review the algorithm and then we will see how to implement it in $O(n^3)$ time.

The Kuhn-Munkres Algorithm (also called the Hungarian Method):

Step 1: Build an Equality Subgraph, G_l by initializing labels in any manner (this was discussed earlier).

Step 2: Find a maximum matching in G_l (not necessarily a perfect matching).

Step 3: If it is a perfect matching, according to the theorem above, we are done.

Step 4: Let S = the set of free nodes in X. Grow hungarian trees from each node in S. Let T = all nodes in Y encountered in the search for an augmenting path from nodes in S. Add all nodes from X that are encountered in the search to S.

Step 5: Revise the labeling, l, *adding edges to* G_l until an augmenting path is found, adding vertices to S and T as they are encountered in the search, as described above. Augment along this path and increase the size of the matching. Return to step 4.

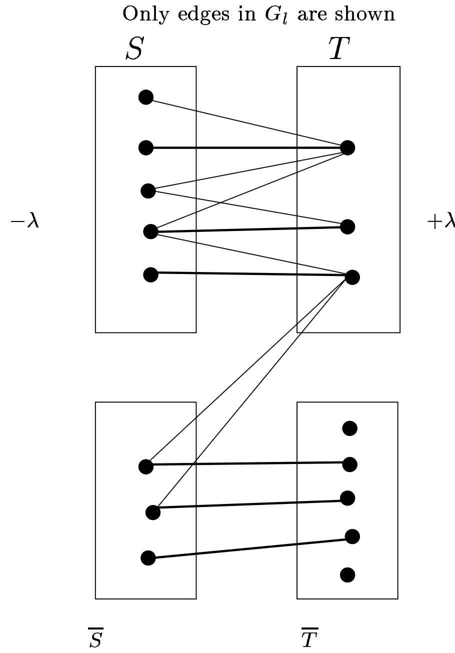


Figure 13: Sets S and T as maintained by the algorithm.

More Efficient Implementation:
We define the slack of an edge as follows:

$$slack(x, y) = l(x) + l(y) - w(x, y).$$

Then

$$\lambda = \min_{x \in S, y \in \overline{T}} slack(x, y)$$

Naively, the calculation of λ requires $O(n^2)$ time. For every vertex in $\overline{T}$, we keep track of the edge with the smallest slack, i.e.,

$$slack[y_j] = \min_{x_i \in S} slack(x_i, y_j)$$

The computation of $slack[y_j]$ requires $O(n^2)$ time at the start of a phase. As the phase progresses, it is easy to update all the *slack* values in $O(n)$ time since all of them change by the same amount (the labels of the vertices in S are going down uniformly). Whenever a node u is moved from $\overline{S}$ to S we must recompute the slacks of the nodes in $\overline{T}$, requiring $O(n)$ time. But a node can be moved from $\overline{S}$ to S at most n times.

Thus each phase can be implemented in $O(n^2)$ time. Since there are n phases, this gives us a running time of $O(n^3)$.