Notes by Samir Khuller.

1 Assignment Problem

We spent some time discussing the stable marriage problem. I am not writing any notes for that since it was purely for your entertainment.

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 1.1 *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,

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

$\square$

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.

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. 1).

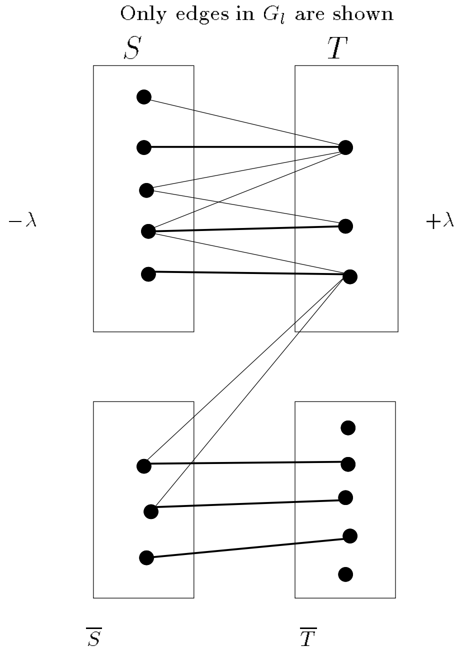

Figure 1: Sets S and T as maintained by the algorithm.

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.

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 $O(n^3)$.