CSCE 411-502 Design and Analysis of Algorithms

Spring 2025

Instructor: Dr. Jianer Chen
Office: PETR 428
Phone: (979) 845-4259
Email: `chen@cse.tamu.edu`
Office Hours: MW 1:30 pm–3:00 pm

Senior Grader: William Kang
Phone: 979) 575-9987
Email: `rkdvlfah1018@tamu.edu`
Questions: via phone and email and by appointments

Assignment #6 Solutions

1. It is clear that if a graph G is not connected then G has no spanning tree. (Slightly) modify Kruskal's algorithm so that on an input graph G, the algorithm either reports that G is not connected, or produces a minimum spanning tree for G. Your algorithm should *not* call a subroutine that tests the connectivity of G. Do not forget to give the complexity analysis of your algorithm.

Solution. A critical observation is that a tree of n vertices has exactly $n-1$ edges. Thus, in the algorithm of Kruskal, if we we count the number of edges added to the output T, we can easily tell if T is a tree or not. In case T is a tree, T is a minimum spanning tree. On the other hand, if T is not a tree, then the graph G is not connected. Note that an edge is added to T only when Union is called.

The modified Kruskal's algorithm is given as follows, where we use N_v and N_e to record the number of vertices and the number of edges in the graph G, respectively.

Modified-Kruskal(G)
1. sort the edges of G in nodecreasing order in edge weights: $e_1, e_2, \ldots, e_m$;
2. $T = \emptyset$; $N_v = 0$; $N_e = 0$;
3. for (each vertex v)
   MakeSet(v); $N_v = N_v + 1$;
4. for ($i = 1$; $i \leq m$; i++)
   let $e_i = [v_i, w_i]$;
   $r_v = \text{Find}(v_i)$; $r_w = \text{Find}(w_i)$;
   if ($r_v \neq r_w$)
      Union(r_v, r_w); $T = T \cup \{e_i\}$; $N_e = N_e + 1$;
5. if ($N_e \neq N_v - 1$)
   return ('G is not connected');
   else return (T).

Since the algorithm is just Kruskal's algorithm with minor changes, which does not change the asymptotic complexity of the orginal Kruskal's algorithm, we conclude that the Modified-Kruskal's algorithm runs in time $O(m \log n)$.

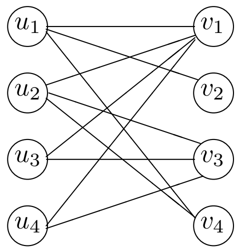
2. A matching M is *maximal* if you cannot add edges to M to make a larger matching (note that it is different from a *maximum* matching). Consider the bipartite graph $G = (U \cup V, E)$, where

$$U = \{u_1, u_2, u_3, u_4\}, \qquad V = \{v_1, v_2, v_3, v_4\},$$

$$E = \{[u_1, v_1], [u_1, v_2], [u_1, v_4], [u_2, v_1], [u_2, v_3], [u_2, v_4], [u_3, v_1], [u_3, v_3], [u_4, v_1], [u_4, v_3]\}.$$

(a) Draw the graph G.
(b) Does G have a (non-maximum) maximal matching of size 1?, size 2? size 3?
(c) For each "yes" answer to the previous part, give a maximal matching of the given size, and show an augmenting path with respect to that matching.

Solution.
(a) The graph G is drawn as follows.



(b) The graph G has no size-1 maximal matching. To see this, note that a maximal matching M_1 of size 1 means a single edge e in M_1 such that every other edge in G shares a common end with e. This is not hard to verify that no such an edge e exists in the graph G.

On the other hand, the graph G has size-2 maximal matchings and size-3 maximal matchings, as given in (c), where the existence of augmenting paths shows that the matchings are maximal but not maximum.

(c) The edge set $M_2 = \{[u_1, v_1], [u_2, v_3]\}$ makes a size-2 maximal matching. The following is an augmenting path with respect to M_2:

$$P = (v_2, u_1, v_1, u_2, v_3, u_4).$$

The edge set $M_3 = \{[u_1, v_1], [u_2, v_4], [u_3, v_3]\}$ makes a size-3 maximal matching. The following is an augmenting path with respect to M_3:

$$P = (u_4, v_3, u_3, v_1, u_1, v_2).$$

3. Consider the following JOB COMPLETION problem:

> Given n workers and m jobs, such that each worker has a list of jobs that he can do. Decide if there is an assignment such that every job gets assigned to a worker who can do the job, assuming that each worker is given at most one job.

Design an efficient algorithm for the problem. Analyze your algorithm.

Solution. This is a typical problem to which graph matching algorithms can be applied. For an instance as described in the problem statement, we build a bipartite graph G of $n+m$ vertices, in which n vertices represent the n workers and the other m vertices represent the m jobs. Add an edge between a worker and a job if the job is in the list of the worker (i.e., if the worker can do the job). Now a matching in G is a way to pair the workers and the jobs such that (1) if a worker is paired with a job, then the worker can do the job; (2) no two workers are assigned to the same job; and (3) no worker is assigned to more than one job. Thus, the problem is actually asking if there is a matching in the graph G in which all job vertices get matched.

The following algorithm implements this idea.

Job-Completion(W, J)
Input: a set W of n workers and a set J of m jobs, each worker has a list of jobs in J;
Output: decide if there is way to assign the jobs to the worker, with at most one job per worker, such that every job gets assigned to a distinct worker

1. construct a bipartite graph $G = (W \cup J, E)$, such that if a job $j \in J$ is in the list of a worker $w \in W$, then there is an edge in E between j and w;
2. call the maximum matching algorithm on the bipartite graph G to construct a maximum matching M in G;
3. if ($|M| = m$)
   return (M) \\this is a job assignment that makes all jobs get assigned
   else return ("no assginemt can make all jobs assigned.").

Define the *length* of the job list for a worker to be the number of jobs in the list. Let L be the sum of lengths of job lists for all workers. It is not hard to see that the length of an instance of the problem is equal to $n+m+L$, i.e., the instance consists of n workers, m jobs, and the job lists for all workers. By the construction, the graph G has $n+m$ vertices and L edges, and can be constructed from the input in time $O(n+m+L)$ in step 1 of the algorithm. As we studied in class, the maximum matching algorithm runs in time $O(|V| \cdot |E|)$ on a bipartite graph of $|V|$ vertices and $|E|$ edges. Now for the graph $G = (W \cup J, E)$, we have $|V| = |W \cup J| = n+m$, and $|E| = L$. Therefore, step 2 of the algorithm runs in time $O((n+m)L)$. In conlusion, the above algorithm Job-Completion solves the given problem in time $O((n+m)L)$, where n is the number of workers, m is the number of jobs, and L is the sum of the lengths of the job lists for the workers.