21-228 Homework 9

Due December 4, 2001, at 5 PM

For this HW, all graphs should be simple (i.e. no loops or parallel edges). The final page contains the algorithms you will need for this assignment. Also, read section 4.3 in Bogart – especially the secton on breadth-first and depth-first search.

Problems 1 and 2 use the following matrix. The matrix is a representation of a road network – there are five cities, and the entry in the ith row and jth column of the matrix is the cost of building a road from city i to city j. Note that since the matrix is symmetric, about the main diagonal. we can represent this data in an undirected graph. It will help to draw the graph. (Do not draw the edges of weight 0).

$$\begin{pmatrix} 0 & 3 & 5 & 11 & 9 \\ 3 & 0 & 3 & 9 & 8 \\ 5 & 3 & 0 & \infty & 10 \\ 11 & 9 & \infty & 0 & 7 \\ 9 & 8 & 10 & 7 & 0 \end{pmatrix}$$

1. Use Kruskal's algorithm to find a minimum spanning tree of the resulting graph. State clearly the order in which edges are placed into the tree, but no other proof is needed.

2. Suppose now we use Prim's algorithm to construct a minimum spanning tree, and we use city 5 as the initial vertex. Now, in which order are edges inserted into the minimum spanning tree?

3. Consider the following network:

$$\begin{pmatrix} 0 & 10 & 20 & \infty & 17 \\ 7 & 0 & 5 & 22 & 33 \\ 14 & 13 & 0 & 15 & 27 \\ 30 & \infty & 17 & 0 & 10 \\ \infty & 15 & 12 & 8 & 0 \end{pmatrix}$$

The entry in the ith row and jth column is the time needed to get from city i to city j. An entry of ∞ means there is no direct route. Use Dijkstra's algorithm to find the shortest route from city 5 to all other cities.

Note that this matrix is *not* symmetric – therefore you will have to represent this as a *directed* graph. Dijkstra's algorithm works equally well for directed and undirected graphs. Write down, after each iteration of the while loop, the values for t_v for all vertices v, and the predecessor p_v.

Also indicate at which point each vertex enters S.

4. In Dijkstra's algorithm, S is the set of all vertices to which the shortest route from the start vertex r is known. When a vertex v gets added to S, how do we know that we have found a shortest path to v?

5. The example that I gave in class, and the one in problem 3, had edges with only nonnegative weights. Show how Dijkstra's algorithm may fail (i.e. produce paths that are not shortest) if we allow edges with a negative weight.

6. Suppose we have a matching in a graph G that saturates some set of vertices $S \subseteq V$. Show that there is a *maximum* matching in G saturating all vertices in S. (Hint: Let M be a matching saturating all vertices in S, and use the augmenting path theorem).

6b) (**NOT FOR SUBMISSION**). In preparation for problem 7, find a graph whose minimum size vertex cover is *strictly larger than* the maximum matching. In a bipartite graph, we can always get equality, so your example must be a non-bipartite graph. What is the smallest nonbipartite graph? What is the size of its smallest vertex cover, and what is the size of its maximum matching?

7. Here is an example of what we call an *approximation algorithm*. I may discuss this in more detail a bit later if there is time. The idea is the following – we have a problem for which we do not know of an efficient algorithm to solve the problem exactly. However, we can get an algorithm that gives an answer reasonably close to the desired value.

The problem of finding a minimum size vertex cover (a set of vertices $S \subseteq V$ that is incident with all edges in E) is what we call NP-hard. I won't define the term here (you don't need to know what it means to solve the problem), but it implies that it is unlikely that we will ever find an efficient algorithm to solve the problem exactly. However, we *can* find an algorithm that produces a

vertex cover that is at most twice the smallest one. Show how finding a maximal (we don't need a *maximum* matching!) matching leads immediately to a vertex cover that is at most twice the size of a minimum vertex cover. Note that we didn't even need to know the exact size of the minimum vertex cover to prove that the cover we created was at most twice as big!

8. Extra Credit: Those of you interested in extra credit can come see me about trying to derive the best known approximation algorithm for the Metric Traveling Salesman Problem. The algorithm itself is not too complicated, but is tricky both to come up with and prove.

Assume that $G = (V, E)$ is the graph we are running each algorithm on:
Kruskal's Algorithm:

1. **Initialize:** Set $G^* = (V, \emptyset)$.
2. **While** G^* is not connected
   (a) Take the smallest edge e in E that has not yet been considered.
   (b) If adding e to G^* forms a cycle, discard e. Else, add e to G^*
3. **End While**
4. **End Algorithm**

Prim's Algorithm for Minimum Spanning Tree, starting at root r Recall $V \setminus S$ is the set of all vertices in V that are not in S.

1. **Initialize:** Set $S = \{r\}$. Set $G^* = (S, \emptyset)$.
2. **While** $S \neq V$.
   (a) Pick the smallest edge (call it uv) from S to $V \setminus S$, where $u \in S$, and $v \in V \setminus S$. Add v to S, and the edge uv to G^* as well.
3. **End While**
4. **End Algorithm**

Dijkstra's Algorithm for shortest path from r to all other vertices v. Remember that c_{uv} is the distance along edge u and v. When using this on a directed graph, remember that uv is the edge that goes "from" u to v.
Here, p_v is the "predecessor" of v on the shortest path from r to v.

1. **Initialize:** Set $S = \{r\}$, $t_r = 0$, $t_v = \infty$ for $v \neq r$. Set $p_r = r$, and set $p_v = -1$ (i.e. a dummy value meaning that we do not have an estimate for the predecessor yet).
2. **While** $S \neq V$
   (a) **For** Each edge uv with $u \in S$, $v \in V \setminus S$. (Really, we only need to check the edges leaving the newest member of S).
      i. **If** $t_v > t_u + c_{uv}$
         A. Set $t_v = t_u + c_{uv}$. Set $p_v = u$.
      ii. **End If**
   (b) **End For**
   (c) Pick the $v \in V \setminus S$ with the smallest value of t_v, and add v to S.
3. **End While**
4. **End Algorithm**