CPS 430/590.06 **Prof. John Reif**
Design and Analysis of Algorithms **Fall 2013**
Homework 5 **Matt Dickenson**

Question 1

We can develop an algorithm to find the shortest path between vertices a and b in a weighted-edge (possibly negative) directed graph, with a guarantee that there will be at most k edges in the shortest path. To develop this algorithm we can use a "greedy" design.

The algorithm takes as input the graph, represented by its vertices and edges, as well as the vertices of interest a and b. We can use two arrays (d and p) to store the distance and path information (regard the values of each vertex to be the real numbers from 0 to $vertices.length - 1$). We also rely on the special values `null` to represent the empty set and `MAX` to represent $+\inf$.

```
1 # initialize with infinite distances from a to each node
2 predecessor = [null] * vertices.length
3 distance = [MAX] * vertices.length
4 distance[a] = 0
5
6 # greedy step
7 for i in 1..k:
8   for e in edges:
9     if distance[e.source] + e.weight < distance[e.destination]:
10      distance[e.destination] = distance[e.source] + e.weight
11      predecessor[e.destination] = e.source
```

This algorithm works because we are guaranteed to find the shortest path that is no longer than k edges by the greedy step. Our double loop takes $k \times |E|$ iterations, for a running time of $O(k|E|)$ as desired.

Question 2

In this problem we want to find the shortest cycle containing an edge e in a graph with positive weighted edges. Because of the postive edge constraint on the graph, it is safe to use a modified version of Dijkstra's algorithm starting at e. We can also reframe the problem of finding the shortest cycle containing e as finding the shortest path from e to e (as long as we ensure that this path contains at least one edge). We use the graph as well as an array of distances from e. Again, regard the values of each vertex to be the real numbers from 0 to $vertices.length - 1$

```
1 modifiedDijkstra(e, vertices, edges):
2   distance = [MAX] * vertices.length
3   distance[e] = 0
4   predecessor = [null] * vertices.length
5
6   Q = {e} # min-queue by distance[]
```

```
 7
 8   until Q is empty:
 9     current = Q.dequeue
10     if current == e and predecessor[e] != null:
11       return
12     else if visited[current]:
13       next
14     visited[current] = true
15
16     for edge in current.neighbors:
17       if distance[current] + edge.weight < distance[edge.destination]:
18           distance[edge.destination] = distance[current] + edge.weight
19           predecessor[edge.destination] = current
20           Q.enqueue(edge.destination)
```

This algorithm has an expected running time less than Dijkstra's algorithm, because we have an earlier stopping point when we get back to the initial node e during our search. However, the worse case running time is still the same: $O(|E| + |V|^2) = O(|V|^2)$, as desired. The correctness of the algorithm is based on the correctness of Dijkstra's algorithm and the insight that the shortest cycle containing e is also the shortest path from e to itself with at least one edge.

Question 3

We wish to prove that in an undirected graph with distinct edge weights there exists a unique minimum spanning tree. We will prove this by contradiction:

1. Assume, for purposes of proof by contradiction, that there exist two distinct minimum spanning trees T_1 and T_2.
2. Identify the minimum edge in T_1 but not T_2, and call it $\underline{e}$.
3. Because T_2 is also a minimum spanning tree, adding the edge $\underline{e}$ to T_2 must induce a cycle
4. The cycle identified in the previous step must contain an edge $\overline{e}$ with a greater weight than $\underline{e}$, because all edges in T_2 with a lower weight are included in T_1 due to our choice of $\underline{e}$ in step 2.
5. Thus, substituting $\underline{e}$ for $\overline{e}$ in T_2 would yield a new spanning tree with smaller weight than T_2. We have arrived at a contradiction of our original assumption that T_2 is a minimum spanning tree.

Question 4

To compute the maximum spanning tree of a graph, we could multiply all of the edges by -1 and find the minimum spanning tree of the modified graph using a known algorithm, such as Prim's.

More formally (and without modifications to the graph), we can reverse the keying of the priority queue Q used in Prim's algorithm (as discussed in class) to prioritize the vertex in Q with the weight of the "heaviest" edge connecting it to a vertex in the subgraph. We denote the vertices of the graph by G.

```
1 Q = V
2 for v in V:
3   key[v] = MAX
4 key[start] = 0
5 predecessor = [null] * V.length
6
7 until Q is empty:
8   u = Q.dequeue
9     for v in u.neighbors:
10      w = weight(u, v)
11      if Q.contains(v) and w < key[v]:
12        key[v] = w
13        predecessor[v] = u
```

When the algorithm is complete, we can begin at the arbitrary `start` vertex and follow the path through the `predecessors` array. Note that the maximum spanning tree is not necessarily unique, just as the minimum spanning tree is not necessarily unique, unless all edges have distinct weights. The correctness of this algorithm is evident from its correspondence to Prim's algorithm. If we let w be the time required by the `weight` operation, the running time is $\Theta(V) \times w + \Theta(E)$.