CS 677: Assignment 7

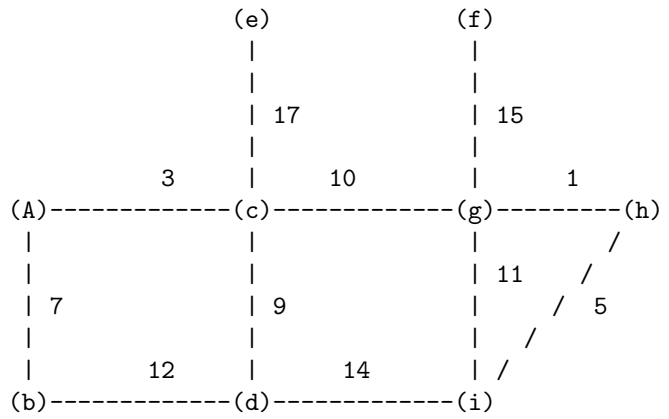
Terence Henriod

May 5, 2014

Abstract

In this assignment, various greedy algorithm problems and related ones are presented.

1. Answer the questions below regarding the following graph:



(a) In what order are edges added to the Minimum Spanning Tree (MST) using Kruskal's Algorithm? List the edges by giving their endpoints.

Solution: Kruskal's algorithm is carried out by first sorting the edges by weight, then selecting edges that have the least weight, but that still add to the span of the tree that is forming without producing a circuit/cycle.

A more canonical way to view Kruskal's algorithm is that one starts with a forrest of trees consisting of a solitary vertex each. Then edges are added light to heavy only if the edge would connect two trees to form a larger tree. This continues until the largest tree(s) possible are formed.

The table below has the edges sorted and the decision of whether or not to include the edge in the spanning tree:

n^{th} edge added	Edge	Weight	Decision
1	(g, h)	1	Keep
2	(A, c)	3	Keep
3	(h, i)	5	Keep
4	(A, b)	7	Keep
5	(c, d)	9	Keep
6	(c, g)	10	Keep
	(g, i)	11	Ignore
	(b, d)	12	Ignore
	(d, i)	14	Ignore
7	(f, g)	15	Keep
8	(e, c)	17	Keep

(b) In what order are edges added to the MST using Prim's Algorithm starting from vertex A? List the edges by giving their endpoints.

Solution: Prim's algorithm essentially starts with an arbitrary vertex and then grows the spanning tree by adding the lightest edge (and its associated vertex) that will grow the most current version of the spanning tree.

The following table displays the edges as they are added to the tree:

n^{th} edge added	Edge	Weight
1	(A, c)	3
2	(A, b)	7
3	(c, d)	9
4	(c, g)	10
5	(g, h)	1
6	(h, i)	5
7	(g, f)	15
8	(c, e)	17

2. Exercise 22.2-9 (page 602): Let $G = (V, E)$ be a connected, undirected graph. Give an $O(V + E)$ time algorithm to compute a path in G that traverses each edge in E exactly once in each direction. Describe how you can find your way out of a maze if you are given a large supply of pennies.

   Solution: By modifying the traditional Depth First Search (DFS) algorithm, we can identify a path that would traverse every edge in a graph precisely twice. The Depth First Search algorithm already explores all edges in a graph. In order to track the order in which the edges are traversed, we simply need to add a list to the algorithm that lists each vertex as it is examined in the DFS-Visit procedure. Once the algorithm terminates, the list will contain an order in which to travel to each vertex, thus crossing each edge twice.

   It is known that DFS is a $\Theta(|V| + |E|)$ algorithm, so it satisfies the problem requirement.

   In the case of the maze, consider every branching or termination of a path a vertex, and thus every path as an edge. Upon embarking on a path, one could leave a penny at the start of that path, and upon leaving the path, drop another penny. After taking a path, if one reaches a dead end, then they must return to the last "vertex" they were at (dropping a penny to show that the path was taken, of course), and at the beginning of the path, they should place a second penny. One should prefer paths not marked by pennies when confronted with a choice, and never take a path that has been marked with 2 pennies. By traversing the maze in this manner, one will eventually find the end without exploring any path more than necessary (not counting making unlucky choices that lead to dead ends, of course).

3. Exercise 22.5-6 (page 621): Given a directed graph $G = (V, E)$ explain how to create another graph $G' = (V, E)$ such that (a) G' has the same strongly connected components as G, (b) G' has the same component graph as G, and (c) E is as small as possible. Describe a fast algorithm to compute G'.

Solution: In short, we need a method that will identify the strongly connected components (SCCs) of G, then once the strongly connected components have been identified, then just enough arcs should be added to make the would be SCCs of G' actually be strongly connected, and then just enough arcs to connect the SCCs to one another to produce a component graph similar to that of G. Note that G' does not actually need to be constructed directly from the elements of G, it just has to have matching components; this means that we are actually free to create arcs as needed instead of selecting from the arc set of G, but a similar vertex set is required in order to have the same components.

Intuitively, start with the algorithm from the text that identifies the SCCs of a graph by performing two depth first searches, one on G and one on G^T, STRONGLY-CONNECTED-COMPONENTS. This takes $\mathrm{O}\big(|V| + |E|\big)$ time. Now we have various subsets of vertices that will constitute the SCCs that we need to match.

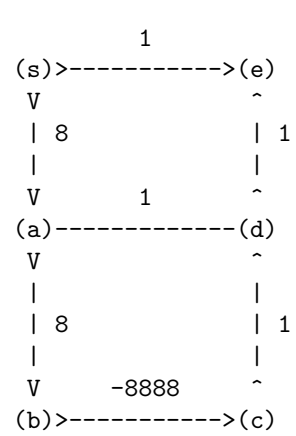
Now, to ensure that the SCCs are actually strongly connected, we need to add arcs that accomplish this. This is most easily done by making one simple cycle that runs through all of the vertices of an SCC. We accomplish this by constructing what is basically a "linked list" of the vertices, which produces —SCC— - 1 arcs, and add an additional arc that connects the last vertex back to the first, giving a total of —SCC— arcs. We do this for all SCCs, meaning we add a total of V arcs, for a running time of $\mathrm{O}\big(|V|\big)$.

Finally, for each pair of SCCs, we create an arc to connect the pair in a way that matches the component graph of G. This operation takes $\mathrm{O}\big(|E|\big)$ time in terms of the size of the arc set of the component graph of G. (If we feel it necessary to create the component graph of G for reference while constructing G', this construction is known to be a $\mathrm{O}\big(|V| + |E|\big)$ operation)

The algorithm described will have a total running time of $\mathrm{O}\big(|V| + |E|\big) + \mathrm{O}\big(|V|\big) + \mathrm{O}\big(|E|\big) = \mathrm{O}\big(|V| + |E|\big)$.

4. Exercise 24.3-2 (page 663): Give a simple example of a directed graph with negative-weight edges for which Dijkstras algorithm produces incorrect answers. Why doesnt the proof of Theorem 24.6 go through when negative-weight edges are allowed?

Solution: Djikstra's algorithm can fail when negative edge weights are allowed due to the fact that vertices are not "re-visited" and the order in which vertices are considered can affect when a negative edge weight is considered. For example, in the graph below, because $w(S, e) < w(S, a)$, e is addressed before a when relaxing edges. This happens again, where d is considered before b in the relaxation step for the same reason as before. This makes it so e cannot be reached again for relaxation since both s and d have been examined in the steps that relax arcs emanating from those respective vertices. This causes the algorithm to fail since the negative edge is considered in the relaxation process too late in the algorithm's execution.



Edge	Weight
(S, a)	8
(S, e)	1
(a, b)	8
(a, d)	1
(b, c)	-8888
(c, d)	1
(d, e)	1

The proof of Theorem 24.6 "doesn't go through" because the proof assumes that because there are no negative edges in the graph that $y.d \leq u.d$, implying that $\delta(s, y) \leq \delta(s, u)$. However, it could be that a negative edge exists in the path from y to u, thus violating the assumption required for the proof. If $y.d > u.d$, then we can't be certain that $\delta(s, y) \leq \delta(s, u) \leq u.d$ when u is added to s, and the constradiction cannot be found.

5. Exercise 25.1-6 (page 692): Suppose we also wish to compute the vertices on shortest paths in the algorithms of this section. Show how to compute the predecessor matrix Π from the completed matrix L of shortest-path weights in $\mathrm{O}(n^3)$ time.

Solution: If we know the weight of the shortest paths for all pairs (since we are given L), we can see if a vertex is a(n) (immediate) predecessor to another on a shortest path. Consider a pair of vertices i and j, where i is the "source" vertex and j is the destination. We can determine if another vertex k is a predecessor to j on that shortest path if the (shortest) weight of the path from k to j complements the path from i to k, that is, if the path from i to k and the final edge from k to j have the same sum as the shortest path from i to j, then k must be a valid predecessor to j on the shortest path.

Thus, by considering every pair of start and end vertices, and then checking every vertex to see if it is a possible predecessor on the shortest path between the pair, we can find the predecessors of every vertex on a shortest path.

Note that it is possible that there could be multiple such k vertices, since there could be multiple equivalent shortest paths from i to j; the algorithm to be presented will only store the last such predecessor considered. The algortihm will produce a possibility for the predecessor matrix. Also note that we don't need to concern ourselves with which shortest path the predecessor matrix will correspond to, since the matrix L does not contain this information anyway.

Since for each of n initial/start vertices we are checking n end vertices to form a pair, and we then check n intermediate vertices, this algorithm is $\mathrm{O}(n * n * n) = \mathrm{O}(n^3)$, as was specified.

Also note that the weights matrix or an edge weight function is also needed for this algorithm; the problem statement neglected to mention this.

The algortihm is defined as follows:

Algorithm *FIND-PREDECESSORS*$(L((1...n)(1...n)), W((1...n)(1...n)))$
($*$ Input: The shortest paths adjacency matrix and the simple edge weights adjacency matrix. $*$)

```
1.  Create Π((1...n)(1...n))
2.  for i ←1 to n
3.      do for j ←1 to n
4.             do π_{i,j} = nil
5.
6.
7.  for  i ←1 to n
8.      do for  j ←1 to n
9.             do for  k ←1 to n
10.                   do
11.                      if  l_{i,j} = l_{ik} + w(k, j)
12.                        then π_{i,j} = k
```

6. Exercise 24.3-6 (page 663): We are given a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has an associated value $r(u, v)$, which is a real number in the range $0 \leq r(u, v) \leq 1$ that represents the reliability of a communication channel from vertex u to vertex v. We interpret $r(u, v)$ as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

   Solution: First, since the probabilities are all independent, we know that all probabilities will have the largest product. This problem in itself displays an optimal substructure, since the largest product will have the largest operands. Further, since we are discussing communication channels that must follow one another, we know that the probability that a channel will not fail will be its own probability multiplied with the largest probability of success up to that point.

   Second, this problem very closely resembles a "shortest paths" problem. We can use the same algorithms for finding a shortest path by replacing addition with multiplication of edge weights, and selecting the maximum result instead of the minimum.

   Finally, it is known that shortest path between a pair algorithms are not better than single source or single destination shortest path algorithms, we will not concern ourselves with the best path between only a particular pair. In fact, since we are not given any indication of which vertices/paths to consider, we may as well just compute the best paths between all vertices.

   Thus, the task can be accomplished by modifying one line of the Floyd-Warshall algorithm from $d_{i,j}^{(k)} = min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)})$ to
   $d_{i,j}^{(k)} = max(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} * d_{k,j}^{(k-1)})$, giving a $\Theta(V^3)$ time algorithm.