CS 477: Homework #7

Due on December 8th, 2016 at 2:30pm

Monica Nicolescu

Matthew J. Berger

Problem 1

[20 points](U & G Required)

Answer the questions below regarding the following graph:

a.) **[5 points]** In what order are edges added to the Minimum Spanning Tree (MST) using Kruskal's Algorithm? List the edges by giving their endpoints.

b.) **[5 points]** 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

a.) g-h, A-c, h-i, A-b, c-d, c-g, f-g, c-e

b.) A-c, A-b, c-d, c-g, g-h, h-i, f-g, c-e

Problem 2

[30 points](U & G Required) Exercise 22.2-9 (page 602).

22.2-9) Let $G = (V, E)$ be a connected, undirected graph. Give an $\mathcal{O}(V + E)$-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

A depth-first-search variant will work well here. Each edge is going to be marked the first and second time it is traversed, with each marking being unique for each traversal. Edges that have already been traversed twice may not be taken again.
Our DFS algorithm ensures that all edges are traversed and that each edge is traversed in both directions. By always taking unexplored edges before edges that have already been explored once, we ensure that all edges are explored. We can also ensure that all edges are taken in both directions by backtracking until a new unexplored edge is found. Using this method, the edges are explored in the reverse direction during backtracking.

The complexity of this algorithm is as requested: $\mathcal{O}(V + E)$

The steps are as follows:

- Execute a depth-first search of G starting an arbitrary vertex. DFS will explore the edges in the graph in the same path as requested by the problem.
- While exploring an edge (u, v) that goes to an unvisited node, the edge (u, v) is included for the first time in the path.
- When the DFS backtracks to edge u again after v is marked, the edge (u, v) is included for the 2nd time in the path, but in the opposite direction ((v, u) instead of (u, v)).
- When the DFS explores an edge (u, v) that goes to an already marked (visited) node, we add $(u, v)(v, u)$ to the path. This ensures that each edge is added to the path exactly twice.

Problem 3

[30 points](U & G Required) Exercise 22.5-6 (page 621).

22.5-6) Given a directed graph $G = (V, E)$, explain how to create another graph $G' = (V, E')$ such that

- G' has the same strongly connected components as G
- G' has the same component graph as G
- E' is as small as possible.

Describe a fast algorithm to compute G'.

Solution

First, we generate k connected components and a strongly connected component subgraph. For each connected component, there must be a loop that can go to all the points, and only the loop is added to the new graph. Assume that the i-th connected component has 5 nodes a, b, c, d, and e. We only need to add sides $a \to b$, $b \to c$, $c \to d$, $d \to e$, $e \to a$. Select the edge of the SCC subgraph and add it to the new graph. The nodes can be selected arbitrarily, which makes the algorithm fast.

Problem 4

[20 points](U & G Required) Exercise 24.3-2 (page 663).

24.3-2) Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces incorrect answers. Why doesnt the proof of Theorem 24.6 go through when negative-weight edges are allowed?

Solution

The proof of Theorem 24.6 doesn't go through when negative weight edges are allowed because the conclusion that $\delta(s, y) \leq \delta(s, u)$ because y occurs before u on the shortest path from s to u doesn't hold when there are negative edges. If there are negative weight edges in the path from y to u then $\delta(s, y)$ is not necessarily $\leq \delta(s, u)$. The proof is based on the conclusion above, therefore it will not be correct if negative edges are present in the graph.

Running Dijkstra's algorithm on the graph below with vertex A as the source produces incorrect results. The actual shortest path from A to B is of weight 2 and the weight of the actual shortest path from A to D is 6.

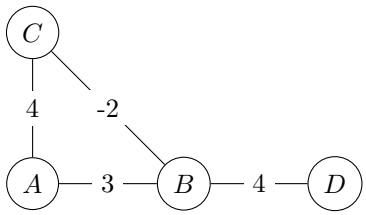

Table 1: The Results of Dijkstra's Algorithm

Step	S	dist[A]	dist[B]	dist[C]	dist[D]
1	{A}	0	3	4	∞
2	{A,B}	0	3	4	7
3	{A,B,C}	0	3	4	7
4	{A,B,C,D}	0	3	4	7

Problem 6

[20 points](Extra Credit) Exercise 24.3-6 (page 663).

24.3-6) 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

The most reliable path between two given vertices would be a path p where the product of the probabilities on that path is maximized. Let s be our source and d be our destination, and let $p = \{v_0, v_1 \ldots, v_k\}$ where $v_0 = s$ and $v_k = t$, then:

$$p = arg\ max\ \Pi_{i=0}^{k}\ r(v_{i-1}, v_i)$$

The problem can be converted to a shortest path problem with a single source by letting the weight on edge (u, v) be $w(u, v) = -\log r(u, v)$. Logarithm doesn't change the monotonicity, and a negative logarithm will convert a minimization problem into a maximization problem. So a shortest path p on the converted graph that satisfies:

$$p = arg\ min\ \sum_{i=1}^{k}\ w(v_{i-1}, v_i)$$

will technically satisfy the initial equation as well.

Dijkstra's algorithm can be used to solve the shortest path problem on the converted graph. Initialization of the weights takes $\mathcal{O}(E)$ time, and the rest are the same as Dijkstra's algorithm. If the graph is mathematically sparse (meaning $E = \mathcal{O}(\frac{V^2}{\log V})$), and all vertices are accessible from the source node, then the algorithm will run in $\mathcal{O}((V + E) \log V + E) + \mathcal{O}(E) = \mathcal{O}(E \log V)$. This algorithm satisfies the problem's requirement for finding an an efficient algorithm to calculate the most reliable path between two given vertices.