CSCI 3110 Assignment 5 Solutions

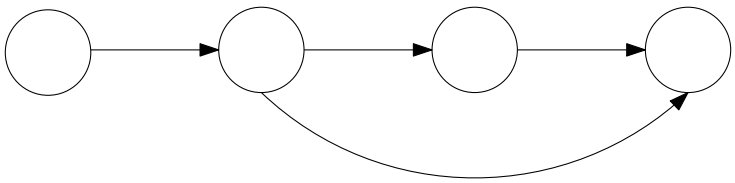
December 5, 2012

4.4 (2 pts) Here's a proposal for how to find the length of the shortest cycle in an undirected graph with unit edge lengths.

When a back edge, say (v, w) , is encountered during a depth-first search, it forms a cycle with the tree edges from w to v . The length of the cycle is level$[v]$ − level$[w] + 1$, where the level of a vertex is its distance in the DFS tree from the root vertex. This suggests the following algorithm:

- Do a depth-first search, keeping track of the level of each vertex.
- Each time a back edge is encountered, compute the cycle length and save it if it is smaller than the shortest one previously seen.

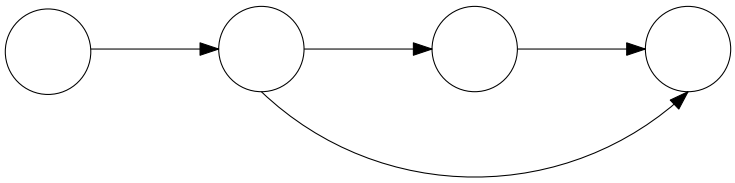
Show that this strategy does not always work by providing a counterexample as well as a brief (one or two sentence) explanation.



This figure shows a graph of 4 vertices a, b, c, d, e. A DFS tree is indicated by marking non-tree edges with dashed lines. The proposed algorithm will only find the length 4 cycle $a \to b \to c \to d \to a$, but will miss the length 3 cycle $a \to d \to e \to a$ because it consists of *two* back edges.

4.8 (2 pts) Professor F. Lake suggests the following algorithm for finding the shortest path from node s to node t in a directed graph with some negative edges: add a large constant to each edge weight so that all the weights become positive, then run Dijkstra's algorithm starting at node s, and return the shortest path found to node t.

Is this a valid method? Either prove that it works correctly, or give a counterexample.



This figure shows a counterexample graph, where a is the constant that we add. When $a = 0$ (ie, the original graph), the shortest path from a to d is $a \to b \to c \to d$. However, when we add $a \geq 2$ to this graph, we penalize longer paths, so the shortest path from a to d is $a \to b \to d$.

4.14 (4 pts) You are given a strongly connected directed graph $G = (V, E)$ with positive edge weights along with a particular node $v_0 \in V$. Give an efficient algorithm for finding shortest paths between all pairs of nodes, with the one restriction that these paths must all pass through v_0.

Any shortest path from two vertices s to t must pass through v_0. Thus, any such path is composed of a path from s to v_0 and a path from v_0 to t. We first use Dijkstra's algorithm to find the shortest path length from v_0 to any other vertex, $B[\cdot]$. We then reverse the graph and find the shortest path length to v_0 from any vertex, $A[\cdot]$. The shortest path from s to t that passes through v_0 has length $A[s] + B[t]$. This takes $O(|V| + |E|) \log |V|)$ time. Note that we do not store all of the shortest path lengths directly, as that would require $O(n^2)$ time.

4.20 (4 pts) There is a network of roads $G = (V, E)$ connecting a set of cities V. Each road in E has an associated length l_e. There is a proposal to add one new road to this network, and there is a list E' of pairs of cities between which the new road can be built. Each such potential road $e' \in E'$ has an associated length. As a designer for the public works department you are asked to determine the road $e' \in E'$ whose addition to the existing network G would result in the maximum decrease in the driving distance between two fixed cities s and t in the network. Give an efficient algorithm for solving this problem.

A newly added edge $e' = (u, v)$ can only decrease the length of the shortest path from s to t if we follow e'. Thus, we would follow the paths $s \to u \to v \to t$. We can solve this problem by first using Dijkstra's algorithm to compute all shortest paths from s to any vertex, $S[\cdot]$ and to t from any vertex, $T[\cdot]$ (using G^R if the graph is directed). For each possible new edge $e' = (u, v)$, the new shortest path length from s to t is $S[u] + l_{e'} + T[v]$. We choose the edge e' that minimizes this (Or none, if $S[t]$ is never improved upon).

5.7 (2pts) Show how to find the maximum spanning tree of a graph, that is, the spanning tree of largest total weight.

Multiply all the edge weights by -1 and find the minimum spanning tree by any of the standard algorithms: Prim's, Kruskal's *etc*

5.9 (10 pts) The following statements may or may not be correct. In each case, either prove it (if it is correct) or give a counterexample (if it isn't correct). Always assume that the graph $G = (V, E)$ is undirected. Do not assume that edge weights are distinct unless this is specifically stated.

(a) If graph G has more than $|V| - 1$ edges, and there is a unique heaviest edge, then this edge cannot be part of a minimum spanning tree.

FALSE. Any unique heaviest edge that is not part of a cycle *must* be in the MST. A graph with one edge is a counterexample.

(b) If G has a cycle with a unique heaviest edge e, then e cannot be part of any MST.

TRUE. An MST has no cycles, so at least one edge of the cycle e' is not in an MST T. If $e' \neq e$ then we could swap e' for e in T and get a lighter spanning tree.

(c) Let e be any edge of minimum weight in G. Then e must be part of some MST.

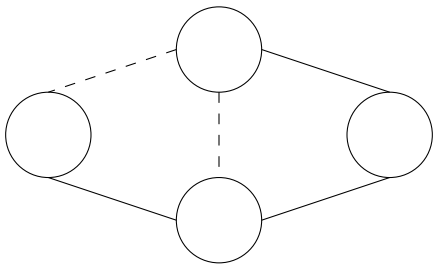
TRUE. An edge of minimum weight is trivially the minimum weight edge of some cut.

(d) If the lightest edge in a graph is unique, then it must be part of every MST.

TRUE. If the lightest edge is unique, then it is the lightest edge of any cut that separates its endpoints.

(e) If e is part of some MST of G, then it must be a lightest edge across some cut of G.

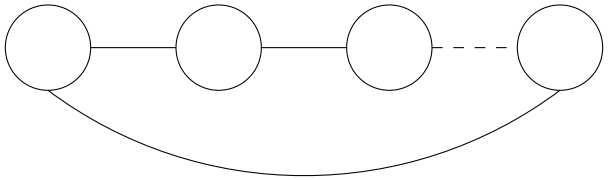
TRUE. If there were a lighter edge e' across some cut of G, then we could replace e with e' and obtain a smaller MST.



(f) If G has a cycle with a unique lightest edge e, then e must be part of every MST.

FALSE. The dashed edge with weight 5 is not part of the MST, but is the lightest edge in the left cycle.

(g) The shortest-path tree computed by Dijkstra's algorithm is necessarily an MST.



FALSE. Dijkstra's algorithm will use the heaviest edge of a cycle if it is on the shortest path from the start s to a node t.

(h) The shortest path between two nodes is necessarily part of some MST.

FALSE. The shortest path between s and t in (g) is not part of any MST.

(i) Prim's algorithm works correctly when there are negative edges.

TRUE. Prim's algorithm always adds the lightest edge between the visited vertices and the unvisited vertices, which is the lightest edge of this cut. Negative weights do not affect this.

(j) (For any $r > 0$, define an r-path to be a path whose edges all have weight $< r$.) If G contains an r-path from node s to t, then every MST of G must also contain an r-path from node s to node t.

TRUE. Suppose that a graph G contains an r-path from a node s to t but that there is an MST T of G that does not contain an r-path from s to t. Then T contains a path from s to t with an edge e of weight $w_e \geq r$. Consider the partition $(S|V-S)$ of vertices made by removing e from T. One vertex e' of the r-path must be along this cut, as the r-path connects s and t. Since $w_{e'} < r$, we can swap e' for e to get a spanning tree that is lighter than T, a contradiction.

5.10 (NOT ASSIGNED- FOR PRACTICE) Let T be an MST of graph G. Given a connected subgraph H of G, show that $T \cap H$ is contained in some MST of H.

Suppose not. Then there is an edge $e \in T \cap H$ across some cut $(S|V-S)$ of H such that another edge across the cut, e', is lighter. However, H is a subgraph of G, so e' is lighter than e across the cut $(S|V-S)$ of G. We could thus swap e' for e in T to get a lighter spanning tree, contradicting that T is an MST.

5.22 (NOT ASSIGNED - FOR PRACTICE) In this problem, we will develop a new algorithm for finding minimum spanning trees. It is based upon the following property:

Pick any cycle in the graph, and let e be the heaviest edge in that cycle. Then there is a minimum spanning tree that does not contain e.

(a) Prove this property carefully.

Suppose that the property is not true. Then there is an edge $e = (u, v)$ of a graph G such that e is the heaviest edge in a cycle and e is part of any MST T of G. Let $e' = (w, x)$ be the lightest edge of the cycle and consider a cut $(S|V - S)$ such that S contains the vertices of the cycle from v to w and $V - S$ contains the vertices of the cycle from u to x. Then the weight of e' is less than or equal to the weight of e across this cut and we can swap e' for e to get an MST T', a contradiction.

(b) Here is the new MST algorithm. The input is some undirected graph $G = (V, E)$ (in adjacency list format) with edge weights $\{w_e\}$.

```
1  sort the edges according to their weights
2  for each edge e ∈ E, in decreasing order of w_e :
3  if e is part of a cycle of G:
4        G = G − e (that is, remove e from G)
5  return G
```

Prove that this algorithm is correct.

We will prove this by induction on the number of edges that have been considered. The base case occurs when we consider the first edge. As this is the largest edge of the graph, it is not part of some spanning tree if its part of a cycle. If it is not part of a cycle, then it is part of any spanning tree.

Now, assume that we have considered k edges and have removed a subset X_k such that there is a spanning T with no edge in X_k. At edge $k + 1$, the algorithm keeps it if and only if it is not part of a cycle of $G \setminus X_k$. By the property, we keep edge $k + 1$ only if it is part of some MST T' of $G \setminus X_k$. By the inductive hypothesis, T' is also an MST of G, so the claim holds.

(c) On each iteration, the algorithm must check whether there is a cycle containing a specific edge e. Give a linear-time algorithm for this task, and justify its correctness.

Let $e = (u, v)$ be an edge of E. If there is a path from u to v that does not use e, then there is such a cycle. Therefore we let $G' = G \setminus e$ and do explore(G, u). If the algorithm visits v then there is such a cycle. This takes linear time.

(d) What is the overall time taken by this algorithm, in terms of $|E|$? Explain your answer.

The overall time taken by the algorithm is the time required to sort the edges, plus the time taken to determine if an edge is part of a cycle, for each edge. This is $O(|E| \log |E| + (|E| \cdot (|E| + |V|)) = O(|E|^2 + |E||V|)$ time.