**CS 180** Homework 3

**Problem 1**

1. The current algorithm checks all edges connected to at least one node in S and that results in a O(n2) runtime. However, for the shortest path problem, we only need to be concerned about the last node added to our set of explored nodes. We could implement the set of explored nodes with a stack and then inside the while loop as the first step, we would just check the node on the top of the stack and check for a minimum weight among only the edges connected to that node. This way, we would achieve an O(n) runtime for the first step.
2. Below is an algorithm that implements a binary heap to store the distances of the vertices used with Dijkstra’s algorithm. S is the set of explored nodes and V is the set of unexplored nodes (initially the set of all nodes in the graph). This algorithm for sparse graphs would give a O(m log n) runtime where m is the number of edges and n is the number of vertices.

function shortestPath(S, V)

create a min heap, H

create an array of distances, D

push starting node to H

set D[starting node] to 0 and D[n] for n = all other nodes to infinity

while H is not empty

pop the top of the heap into a value, T

place T into set of explored nodes S

for each node N connected to T

if N is in H && D[N] > D[T] + we where we is the weight of the edge T - N

D[N] = D[T] + we

push N onto H

**Problem 2**

Given a set of time intervals, we can construct a Directed Acyclic Graph by creating a node for each interval. The nodes would contain SI, EI, and PI. We can then reduce this by creating a directed edge from one node to another if the node’s EI is less than the other node’s SI *i.e.* if the interval ends before another one starts. From here this becomes a directed graph problem where we have to find the longest path. The algorithm below takes in the intervals as an array. This would yield a topological sorting which we then can use to find the maximum price by iterating through each vertex in linear order and storing all combinations of paths. We then find the max of this to get the longest path.

Dynamic Programming algorithm below:

longestPath(intervals[])

sort intervals by EI (ending time), earliest end time to latest

int table[size of intervals array]

table[0] = intervals[0].price // initial price of first interval

for i in 0… size of intervals array

int currentPrice = intervals[i].price // price with current interval

int nonConflict = -1

for (int j = i-1; j >= 0; j--)

if (interval[j].finish <= interval[i].start) // find latest interval not conflicting with i

nonConflict = j

break

if nonConflict is not -1

currentPrice += intervals[nonConflict].price

table[i] = max(currentPrice, table[i-1]) // table stores solutions to subproblems for dynamic

// programming

int result = table[n-1]

return result

**Problem 3**

If these integer edge weights are all positive, we can then replace all the edges of weight > 1 with intermediate nodes connected by edges of weight 1. For example, if there is a edge with weight 4, we can replace it with 3 nodes connected by 4 edges each of weight 1.

A ----- B ⇒ A --- A2 --- A3 --- A4 --- B

4 1 1 1 1

Doing this to all nodes in the graph would give an “unweighted” graph since all nodes have equal weight. From here, we can utilize BFS to compute the shortest path from a source vertex to all other vertices. This algorithm depends on the maximum weight an edge can have and will be costly with a lot of heavily weighted edges. The runtime would be O(V \* Wmax) where V is the number of vertices and Wmax is the maximum weight an edge has.

**Problem 4**

1. To find the diameter of a tree, one can pick any random node and run a BFS to find the farthest node from it. If we call this end node A, we remember it and run a second BFS to find the farthest node from A. The new resulting node, B and the path from it to A is be the diameter of the tree.

int recursiveDiameter(graph)

if graph.nodes is 1

return 0

if graph.nodes is 2

return 1

return 2 + recursiveDiameter(prune(graph))

graph prune(graph)

for every node in graph

if node.degree is 1

remove the node

-------------------------------------------------------------------------------------------------------------------

int diameter2(root, int\* h) // h is a pointer to an int, representing the height

int lh = 0, rh = 0 // left and right height

int ld = 0, rd = 0 // left and right diameter

if root is null

\*height = 0

return 0

ld = diameter2(root->left, &lh);

rd = diameter2(root->right,&rh);

\*height = max(lh, rh) + 1;

return max(lh + rh + 1, ld, rd);