HW #6 CS330 CHANDINI TOLETI U29391556

1A) ALGORITHM

```
def trip_planning(s, train): /*let s be the starting city and train be the n tuples*/
Initialize:
pi { }← /* hash table, current best list for v */
D← { }/* hash table, distance of v */
parents← { }/* parents in shortest paths tree */
Q← PQ/* priority queue to keep track of min ⇡ */

Do:
Time =0;
 $\pi$[s] = 0;
 Q.INSERT(< 0,s>);
 for i=1 to n  in train(i) do: /* n is inclusive and there is no train 0*/
      If depcity(i) $\neq$ s and not in pi do:
            $\pi$ [depcity(i)] = $\infty$;
            Q.INSERT(< $\pi$ [depcity(i)], depcity(i)]  >);
      If arrcity(i) $\neq$ s and not in pi do:
            $\pi$ [arrcity(i)] = $\infty$;
 while Q is not empty do:
       < $\pi$ [city],city> EXTRACT-MIN(Q);
      d[city] = $\pi$ [city];
      time = d[city];
      for i=1 to n in train(i) do:
            if dept(i)<time or depcity(i) $\neq$city :
                  continue;
            if arr(i)< $\pi[arrcity(i)]$ do:
                  DECREASE-KEY(<$\pi$ [$arrcity(i)$], $arrcity(i)$>, <$arr(i), arrcity(i)$>)
                   $\pi[arrcity(i)]= arr$ (i);
return d
```

1B) PROOF OF CORRECTNESS

<u>Assertion 1:</u> For every (destination city, starting city) pair, the algorithm finds the earliest arrival time

Like Dijkstra's algorithm, each earliest arrival time is initialized to infinity for all cities except the starting city, which is 0. This is correct because if you start at a city, you've already arrived there at time 0. Next, the algorithm uses a priority queue to list the arrivals of explored cities by earlierst arrival time. Right now, this just contains the starting city with arrival time 0. Next, the algorithm looks at all the trains departing from city s (since it has

(continued on next page)

the smallest arrival time 0. At this point, we know that that smallest value from the priority queue has the shortest path to that city, and updates all the hash tables to store information about this path. Additionally, we add the destination cities of this city's trains and their arrival times to the sorted queue of earliest arrival times. The process continues, for each of these cities. Each earliest city pulled updates the hash tables again, but as destination cities are explored two conditions may happen. 1) the algorithm discovers an arrival time to a city that's earlier than its current earliest arrival time, so the algorithm checks if that train is boardable (see assertion 2) and updates the earliest arrival time. 2) otherwise, the algorithm doesn't update anything. In the end, the algorithm will eventually terminate after the queue is empty and every shortest path has been discovered.

Assertion 2: For every starting city, the algorithm does not let the passenger board unless they arrive before the departure time.

When exploring a city who's best path has already been determined (say City A), the algorithm checks all paths starting from A. Say there exists a path from A to B where a train arrives in B earlier than B's current earliest time. Now, one of two conditions is true: 1) the train from A to B leaves A before the best path to A arrives or 2) the train from A to B leaves during or after A's best path arrives. In case 1, the passenger cannot make the train and therefore, the algorithm will not update B's earliest arrival path. This is because the Algorithm skips the updating code when dept(i)<time or depcity(i) $\neq$ city, which describe the situation. Otherwise when case 2 occurs, the algorithm updates as Dijkstra's normally does.

All-in-all, the algorithm operates just as Dijkstra's, which we learned in lecture, except for the lack of a Graph, so different iterative techniques and an additional check to see if a train is *boardable* or not. By Assertions 1 and 2, however, we've proved that like Dijkstra's this implementation correctly solves the given problem.

1C) COMPLEXITY

TIME ANALYSIS: To Analyze time complexity, we look at each individual step of the algorithm. Let V be the number of cities and n be the number of trains.

1) Initializing all hash tables will take O(V) time.
2) Inserting the starting city in the min-heap priority queue with the operation Q.INSERT(< 0,s>) will take O(logV) time.
3) In the worst case, for every n train we run the command Q.INSERT(< π [depcity(i)], depcity(i)] >), which would take O(nlogV) time complexity
4) The second loop, in the worst-case, will run for every city the command <π[city],city> EXTRACT-MIN(Q) which takes O(V log V) time.
5) The inner loop, in the worst case, for every train for every city, will run DECREASE-KEY(<π [$arrcity(i)$], $arrcity(i)$>, <*arr(i),* $arrcity(i)$>), which would take O(n log V) time.
6) All other operations take constant time.

(continued on next page)

All-in-all, the overall complexity would look like O(V + logV + n logV * V logV), which we simplify to **O(V * n logV)**.

SPACE COMPLEXITY: To analyze space complexity, we observe the inputs, any temporary structures, and outputs.

1) train(i) is a list of *n* trains that takes O(n) space
2) All hash tables, pi, D, parents, and the Q priority Q each take O(V) space and O(4V) simplifies to O(v). This is because throughout the entire algorithm the worst case is that every city V is added to these structures
3) The hash table D, which we already analyzed is the output
4) All other elementary data take O(1) space

All-in-all, the space complexity can be described as **O(n+ V)** in the worst-case

2.1)

CUT PROPERTY: Let S be a subset of nodes. Let e be the heaviest edge with exactly one endpoint in S. Then every MaxST contains e.
CYCLE PROPERTY: Let C be a cycle, and let f be the lightest edge in C. Then no MaxST contains F.

2.2)
Assertion: For any graph G with unique edge clearance, the path given by using edges from the maximum spanning tree of G gives the maximum clearance route between two warehouses.

Proof: Consider the proof by contradiction. Say the highest clearance path from node u to v is not part of the MaxST. For this to be true, the graph G must have two unique paths from u to v, we'll denote these as P1 and P2: P1 is the path included in the MaxST, while P2 is not included but P1 clearance<P2 clearance. Given the assumption, we know the minimum clearance of P1(P1min) and P2 (P2 min) are such that P1min<P2min. Given the assumption and the definition of a cycle, we know that if P1 and P2 exist and are unique, a cycle must exist including (u,v).

Recall the Cycle Property, say C is the cycle between u and v. In this instance f, the lightest edge in the cycle would be P1min, as we observed before. By the cycle property, this edge must not be included in the MaxST; however, it is included. This contradicts the cycle property and by proof of contradiction proves the assertion to hold true.

(continued on next page)

2.3A) ALGORITHM

```
def max_span_prim(G,w):

Initialize:
parents ← /*empty hash table, will later store MaxST*/
Q←V /* max-heap priority queue initialized with all vertices v ϵ V as values

//initialize priorities for Q
priority[v] ← −∞ for all v ϵ V
priority[s] ← 0 for some arbitrary  s ϵ V

while !(Q.isEmpty) do:
    u ← Q.Extract-Max()
    for v in G[u] do:
        if v ϵ Q AND w(u,v)> priority[v]
            then priority [v] ← w(u,v)
                Q.Increase.Priority(v, priority[v])
                Parent[v]←u /* initialize key if not already in hash*/
Return parent
```

2.3B) PROOF OF CORRECTNESS

Assertion: The parents hash table returned by the algorithm upholds the properties of a MaxST. In other words, the tree contains no cycles and maximizes edge weights.

Consider the inductive proof:

Base Case: the single start node s upholds the properties of the MaxST since one node cannot contain any cycles and has no edge weight, which is at maximum of its edge weights.
Inductive Hypothesis: After each of k iterations, the tree constructured will uphold the properties of a MaxST Tree at every step.
Inductive Step: After k+1 iterations, the algorithm uses the max-at-top heap to remove the node *u* with the largest priority then creates a partial MaxST tree with another node connected to *u*. For all neighboring nodes of *u*, the algorithm checks if adding that edge would increase the current highest weighted path to that node, if so all structures are

(continued on next page)

updated to reflect that. This step 1) maximizes edge weights being added by checking if w(u,v)> *priority*[v], while also 2) ensuring that a cycle is not formed, no non-max weights are added, and the algorithm terminates by using a max-at-top priority queue.

Therefore, by induction, the assertion holds true for all k since the assertion is upheld every k+i step.

2.3C) COMPLEXITY

TIME ANALYSIS: To analyze time complexity, we look at every step of the algorithm. Let V be the number of vertices and E be the number of edges.

1) For the max-at-top heap priority queue, inserting an element takes O(logV) time. Since we must do this operation V times, the time complexity is O(V * logV).
2) Initializing priorities for Q is done by two operations: *priority*[v] $\leftarrow -\infty$ and *priority*[s] $\leftarrow 0$. These operations take O(V + 1) or O(V) time.
3) For the outer-loop, we run u $\leftarrow$ Q.Extract-Max() V times. Therefore, the time complexity of this operation can be given as O(V*logV)
4) For the inner-loop, we run Q.Increase.Priority(v, *priority*[v]) for every edge in the worst-case. Therefore, the time complexity of this operation can be given as O(E * log V)
5) All other operations run in constant time

All-in-all, the total time complexity can be represented by O((V+E)* log V), where V represents the number of vertices and E the number of edges in graph G. This is the same runtime as Prim's algorithm due to the similar nature.

SPACE ANALYSIS: To analyze space complexity, we look at all the inputs, outputs, and other temporary structures utilized by the algorithm.

1) The graph representation G will take O(V+E) space
2) The parent hash table will take O(V) space
3) The priority queue representing the max-at-top heap will also take O(V) space
4) All other simple structures take constant space

All-in-all, the total space complexity can be given by O(V+E), combining like terms and disregarding coefficients.

(continued on next page)