CSCI270 Homework 3

Jacob Ma

September 28, 2022

Problem 1

Consider an undirected connected graph $G = (V, E)$ with edge costs $c_e > 0$ for $e \in E$ which are all distinct.

1. Let $E' \subseteq E$ be defined as the following set of edges: for each node v, E' contains the cheapest of all edges incident on v, i.e., the cheapest edge that has v as one of its endpoints. Is the graph (V, E') connected? Is it acyclic? For both questions, provide a proof or a counter-example with explanations.

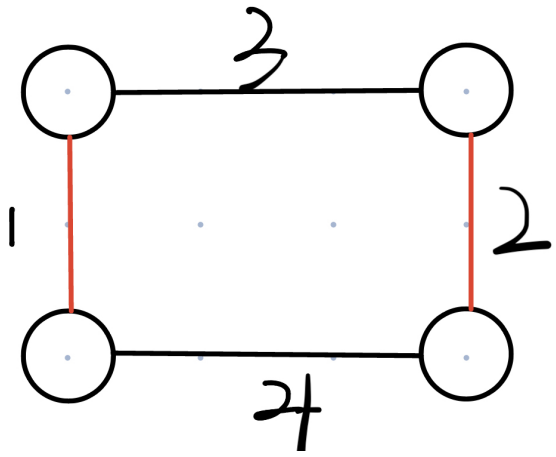
2. Consider the following outline for an algorithm, which starts with an empty set T of edges: Let E' contain the cheapest edge out of each connected component of (V, T). Add E' to T, and repeat until (V, T) is connected. Show that this algorithm outputs a minimum spanning tree of G, and can be implemented in time $O(m \log n)$.

Hints:

- Each iteration of this algorithm can be viewed as applying the operation from part (a) on a "contracted graph", where each connected component of (V, T) corresponds to a node.
- If you want, you are welcome to use the efficient implementation of the Union-Find data structure which supports Find and Union in logarithmic time, as described in Chapter 4.6 of the KT book and briefly outlined in class. However, this implementation is not necessary, and focusing on it may mislead you from the basic idea.
- We recommend thinking about how many iterations it will take until (V, T) is connected.

Solution: Problem 1 (a), Not Connected

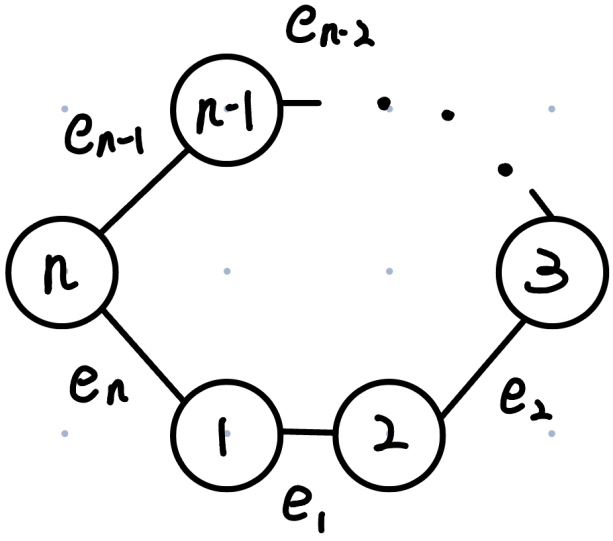
he graph (V, E') is not connected, and we disprove by giving counter-example.



Counter-Example of connected graph.

In the counter example above, red edges are included in E', and contains the cheapest of all edges incident on every vertices. But the graph (V, E') is not connected.

Proof. Proof for Problem $1\,(a)$: the graph is acyclic. We prove by contradiction. Assume the graph (V, E') isn't acyclic, means that there are loops within (V, E'). Consider the graph below, we denote all the vertices as $v_1, v_2, v_3, ..., v_{n-1}, v_n$ representing a graph with n vertices, and $e_1, e_2, ...e_{n-2}, e_{n-1}, e_n$ representing n edges connecting these vertices.



Assume there is a graph (V, E') with loops with n vertices

Let's start with vertices v_1, assume without loss of generality that $e_1 < e_n$, thus for v_1, it will choose e_1 as the cheapest edge that v_1 as one of its endpoint and add into E'.
Thus, as for v_2, v_2 need to add e_2 as the cheapest edge that v_2 as one of its endpoint to maintain both e_1 and e_2

within the loop. So, $e_2 < e_1$.
This will also work for v_3, v_3 choose e_3 as its cheapest adjacent edge, and $e_3 < e_2$.
The same process will continue till v_n. Because e_{n-1} has already been added by v_{n-1}, v_n need to add e_n into E', meaning $e_n < e_{n-1}$.
Now we have the following inequality:

$$e_1 > e_2 > e_3 > \cdots > e_{n-1} > e_n$$

Thus, $e_1 > e_n$. But there exist a contradiction that $e_1 < e_n$ and v_1 chose e_1 as its cheapest adjacent edge.
The conclusion follows the contradiction. □

Proof. Proof of 1 (b).

Lemma 0.0.1

The algorithm terminates, and the output contains every element in V, that is all the nodes.

Proof. If the algorithm does not terminate, it means that (V, T) is not connected. Thus, there must exist two component S and $\overline{S}$, and the algorithm could not find the cheapest edge out of S and $\overline{S}$, indicating there is no edge between this two components, and the graph is disconnected, contradicting with the fact that $G = (V, E)$ is connected. Thus, the algorithm must terminate and contains every nodes inside the output. □

Lemma 0.0.2

The algorithm outputs a spanning tree.

Proof. Firstly, we know that from the previous lemma, every node is contained in the output.
We also know that, during each iteration, the number of disconnected component is strictly decreasing because we are containing the bridges between two disconnected components and connecting them and merging them into a new component. Thus, after the program terminates and connecting the originally finite number of components (number of V), the output would be a single component (V, T). Thus, we have a connected single component as an output.
Moreover, we also know that the output is acyclic. Assume the output is cyclic, there would exist a cycle inside the output, indicating there are two path s_1 and s_2 between two components S and $\overline{S}$. According to the algorithm, s_1 added by the algorithm must be the cheapest edge out of S or $\overline{S}$, connecting them into a bigger component S', and $s_1 < s_2$. s_2 would also be the cheapest edge "out of" S or $\overline{S}$, but here is a contradiction that S and $\overline{S}$ is a single component S' after adding s_1, thus it is not "out of" the connected component S'. Thus, there exists a contradiction and the output must be acyclic.
Thus, the algorithm outputs a spanning tree. □

Lemma 0.0.3

The output is minimal.

Proof. Let's denote the output spanning tree of this algorithm as T, and a Minimum Spanning Tress as T' , which is different from T. Thus, we know that there exists edges $e = (v_1, v_2)$ that $e \in T'$ but $e \notin T$. By the algorithm, we know that e is the cheapest edge coming out of the connected component either containing v_1 or v_2. So, e is the cheapest edge connecting the component S containing v_1 and $\overline{S}$ containing v_2. According to the cut property, this edge e must be contained in any MST. However, $e \notin T'$, thus there exist a contradiction. We know that every element in T is in T'.

We also now that if we delete any edge of T, the graph would be disconnected and the algorithm would not terminate, thus we known the number of the edges contained in T equals to the number in MST.

The lemma follows the contradiction. □

Thus, according to the lemmas, we know the output is a minimum spanning trees. The conclusion follows. □

Proof. Runtime for 1 (b)

According to the algorithm, we know that every time we run the iteration, the number of components, say i, would be at least halved into $\lfloor \frac{i}{2} \rfloor$ if for each pair of components containing (v_1, v_2), their shared edge $e = (v_1, v_2)$ is the cheapest edge between both of these two components and both of them choose the same edge e. Thus, there would be at most $\log n$ iterations since there are n nodes.

Inside each iteration, we need to check all m edges if they are the cheapest edge out of this component. We also need to determine if two nodes on both ends of an edge are within the same component.

```
Set label of all node v ∈ V as 'null' , set int LabelNum to 1.
for each v ∈ V do
  if v is labeled then
    continue
  else
    conduct BFS or DFS on v, label all the nodes v' find in this search with LabelNum.
    LabelNum++
  end if
end for
Initially an array A with size LabelNum, set all values originally to a dummy edge with ∞ cost, which
used to later store the shortest edge out of each component
for All edges e = (u, v) ∈ E do
  if u.label == v.label then
    continue
  else if c_e < c_{A[u.label]} then
    A [LabelNum] = e
  end if
  Add all the elements in A into T
end for
```

Here is one way of implementing each iteration, which is picking up an unlabeled node and conduct a BFS or DFS search, labeling all the nodes found in this BFS/DFS with an unique identifier. Repeat this process until all of the nodes are being labeled, which took $O(m+n)$ time in total, because $m \geqslant n-1$, thus $O(m+n) = O(m)$. After the these BFS, all the nodes in the same component would be classified with the same label. Thus, during

each iteration of checking m edges, it only takes constant time to see if they belong to the same component, and constant time to update the current shortest edge in each component. Thus, during the whole process, the whole step would take $O(m) + O(m) = O(m)$.

Thus, the overall runtime is $O(\log n) \cdot O(m) = O(m \log n)$. □

Problem 2

We frequently motivate the shortest path problem, and Dijkstra's Algorithm for solving it, by considering transportation networks, such as driving from one place to another. Other transportation networks are rail or flight networks. Those are a little different from road networks, in that the edges (e.g., trains) are only available at certain times, rather than all the time with a given length.

Specifically, you are given a directed graph $G = (V, E)$ in which each directed edge $e = (u, v, t_u, t_v)$ corresponds to a train connection that goes from u to v, leaves at time t_u, and arrives at time t_v. Of course, there can now be multiple edges from u to v. You know that for each such edge, $t_v > t_u$, since the train cannot arrive before it leaves.[a] You know nothing else about the arrival and departure times; in particular, there could be two trains from u to v with $t'_u > t_u$ and $t'_v < t_v$. For example, these could be an express train and a scenic local train, where the express train leaves later and arrives earlier. You are also given a start node s, start time T, and destination node d. Your goal is to compute the earliest that you can arrive at d when starting at s at time T. You can assume that changing trains takes 0 time, so if you arrive at node u at time t, you can board a train that leaves u at time t.

Give and analyze (i.e., prove correct and analyze the running time) an algorithm that solves this problem in time $O(m \log n)$.

Hint: Obviously, this question is closely related to Dijkstra's Algorithm, which we did not cover in class. We highly recommend that you review the algorithm and — more importantly — its analysis in Section 4.4 of the textbook before attempting this question.

[a] We are talking about trains here, not DeLoreans.

Proof. In order to prove this rough algorithm 1 (complete implementation is algorithm 2, please refer to this one if not clear enough), we apply the analysis of "Greedy Stays Ahead".

Lemma 0.0.4

Consider the set S at any point in the algorithm's execution. For each $u \in S$, the time $d(u)$ is the earliest arrival time among all available path starting from s without time conflict.

We prove this by induction on the size of S. Without loss of generosity, we assume the graph is connected and every node could be guaranteed with a solution. (The algorithm will terminate in another way if the graph is not connected or without a valid solution, by quitting the while loop without adding new element. This will be stated in the second lemma)

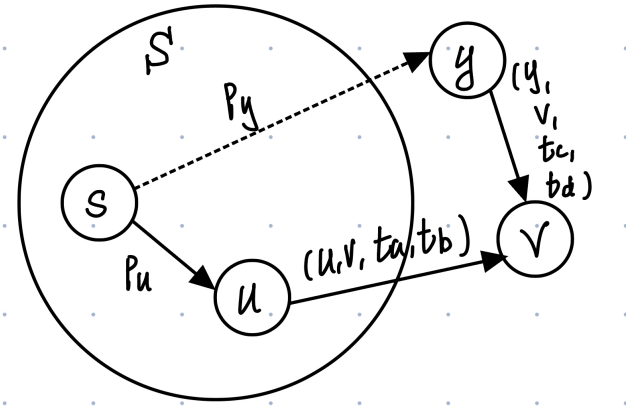
Base Case: $|S| = 1$. We have $S = \{s\}$ and $d(s) = 0$. Thus, $|S| = 1$ holds.

Induction Hypothesis: Suppose the lemma holds true for every integer $i \in \mathbb{N}$ and $i \leqslant k$, for some $k \geqslant 1$ for $|S| = i$.

Algorithm 1 KT Dijkstra's Algorithm Modified (G, t)

Let S be the set of explored nodes.
for Each $u \in S$ **do**
  We store a arriving time $d(u)$
end for
Initially $S = \{s\}$ and $d(s) = T$
while Destination Point $d \notin S$ **do**
  if There does not exist new nodes $v \notin S$ with at least one edge $e = (u, v, t_u, t_v)$ with $u \in S$ and $d(u) \leqslant t_u$ **then**
    Return No Solution
  end if
  Select a node $v \notin S$ with at least one edge $e = (u, v, t_u, t_v)$ with $u \in S$ and $d(u) \leqslant t_u$, such that among all such edges $e = (u, v, t_u, t_v)$, we have $d'(v) = t_v$ as small (early) as possible.
  Add v to S and define $d(v) = d'(v)$.
end while
Return $d(d)$

Induction Step: Consider $|S| = k + 1$, growing S to size $k + 1$ by adding the new node v as the considered node. By induction hypothesis, $d(v)$ is the earliest arrival time among all available path starting from s without time conflict. Now consider another arbitrary $s - v$ path P. We wish to show that through P, the arriving time is at least as early as $d(v)$. In order to reach v, because $s \in S$ but $v \notin S$, the path P must leave S somewhere. Let y be the first node on P that is not in S. As shown below:



Two Possible $s - v$ path

As shown below, the $s - v$ path is consists of two component: the first is $s - u$ path P_u, which is has the earliest arrival time on u because $u \in S$ according tot he induction hypothesis; the second component is the edge $e = (u, v, t_a, t_b)$. According to the algorithm, the arrival time of v, t_b must be greater or equal to the leaving time t_a, and greater or equal to the earliest arrival time of u, which is $d(u)$. Thus we have $d(u) \leqslant t_a \leqslant t_b$.
Based on the algorithm, since we choose v instead of y, it means that

$$t_b \leqslant \text{All possible arrival time from any existing node starting within } S \text{ to } y$$

Because we can only choose an edge leave later than we arrive at y, and we need have a non-negative difference between the arrival and starting time within a single edge, thus we have

$$t_d \geqslant t_c \geqslant \text{All Possible Arrival Time from Any Existing Node} \in S \text{ to } y$$

Thus, in combination, we have

$$t_b \leqslant \text{All Possible Arrival Time from Any Existing Node} \in S \text{ to } y \leqslant t_c \leqslant t_d$$

Thus, any other path P would lead to an at least later time to v.
Because according to the induction hypothesis, P_u does not contain a time conflict, and according to the algorithm, $d(u) \leqslant t_a$, also does not contain a time conflict. Thus, the $s - v$ path based on this algorithm is available.
The lemma follows the induction.

Lemma 0.0.5

The algorithm terminates.

Proof. There are two case: the graph is connected and has a valid solution for the input; or the graph is not connected or without a conflict-free path between the two given destination.

1. The graph is not connected or without a conflict-free path between the two given destination.

   If the graph is not connected or without a conflict-free path between the two given destination, at some point, all the node $v \notin S$ will fail to find an edge $e = (u, v, t_u, t_v)$ with $u \in S$ and $d(u) \leqslant t_u$. Thus, the algorithm would terminate directly within the while loop by the return statement.

2. The graph has a valid and connected solution for the input.

   Because a valid connected solution is guranteed for this graph, there must exist some node $u \in S$, which is connected with some $v \notin S$ and have some edge $e = (u, v, t_u, t_v)$, if $d \notin S$. So in such iteration, v will be added in S, and the number of nodes $\notin$ will minus one. Because there are only finite number of nodes, which is n, the algorithm will eventually terminate.

□

Since the algorithm terminates, we can apply the first lemma after the algorithm terminates, when $S = V$. Thus, every node $\in V$ and has their earliest arriving time.
The conclusion follows. □

Proof. Proof for the runtime.
In order to analyze the runtime, we can do with the data structure using priority queues. We put the nodes V in a priority queue with $d'(v)$ as the key for $v \in V$.
One possible implementation 2 is below.
In this algorithm, we are using a priority queue, which takes $O(\log n)$ for both *ExtractMin* and *ChangeKey*. (*ChangeKey* here not only changes the value of the key, but also help heapify/reorder the priotity queue, similar to how ChangeKey works in max/min-heap). Firstly, the initialization of the priority queue N uses n iterations for all nodes in V, and takes a $O(\log n)$ runtime adding them into the N. At last, we need to initialize n with

ExtractMin, which takes $O(\log n)$. Thus, the initialization of N takes $O(n \log n)$.

Inside the while, we are adding every node u into S, thus there are n iterations in total. Inside each iteration, there is first a *ExtractMin* takes $O(\log n)$. As for the for section, every edges is being checked exactly once because every edge has a starting points, thus takes m iterations in total. Inside each iteration, the *ChangeKey* takes $O(\log n)$ time. So for *ExtractMin*, it has $O(n \log n)$ runtime in total, while *ChangeKey* is $O(n \log m)$.

Adding every node u to S would take $O(n \log n)$ in total.

We know that in the worst case scenario, the algorithm is indeed connected and has a solution (because the algorithm would terminate early if the graph is not connected as shwon in the lemma above). In this case, we know that $m \geq n - 1$, thus $O(m \log n)$ dominates $O(n \log n)$. Thus, the runtime would be $O(m \log n)$. □

Algorithm 2 KT Dijkstra's Algorithm Modified (G, t)

Let N be a priority queue (from small to large), where key is $d(v)$ and value is v.
for each $v \in V$ **do**
  if $v \neq s$ **then**
    We add v in to N with key $d(v) = \infty$
  else
    Add s with key $d(s) = T$
  end if
end for
Initially $S = \{s\}$ and $d(s) = T$
ExtractMin from the N, denote it u
while u is not the destination **do**
  if $d(u) == \infty$ **then**
    Return No Solution
  end if
  for Adjacent edges $e = \{u, v, t_u, t_v\}$ starting form u **do**
    if $d(u) \leqslant t_u$ and $t_v < d(v)$ **then**
      $d(v) = t_v$ using *ChangeKey*
    end if
  end for
  Add u to S
  updateb u with *ExtractMin* from the N
end while
Return $d(destination)$