CSE 101 FALL 2019 Homework 3 Due: Thursday, 10/17 at 11:59pm

1. Suppose Dijkstra's algorithm is run on the following graph, starting at node A.

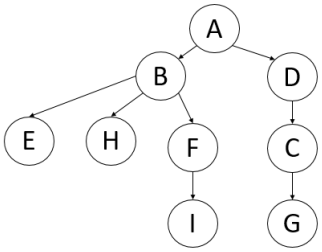
   (a) Draw a table showing the intermediate distance values of all the nodes at each iteration of the algorithm. (5 points)

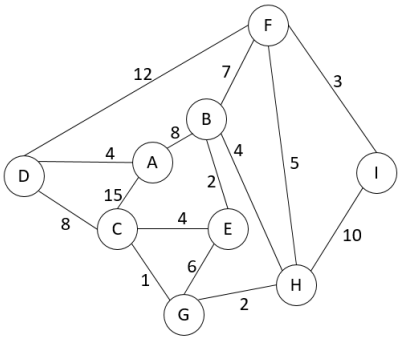
   Solution:

A	B	C	D	E	F	G	H	I
0	∞	∞	∞	∞	∞	∞	∞	∞
	8	15	4	∞	∞	∞	∞	∞
	8	12		∞	16	∞	∞	∞
		12		10	15	∞	12	∞
		12			15	16	12	∞
					15	13	12	∞
					15	13		22
					15			22
								18

   (b) Draw the shortest path tree. (5 points)

   Solution:





2. Suppose you are the newly elected chief maintenance officer for the subway system of your city. The previous maintenance officer did a very bad job and a subset of the stations have not been functional for a long time. The most popular route is going from station S to station T. Your first matter of business is to repair the broken station that will make the commute from S to T the shortest. (you can assume that S and T are both still functional.)

   Problem Statement:

   Input: *directed* graph $G = (V, E)$ with positive edge lengths $\{\ell(e)|e \in E\}$. Starting and ending vertices $S, T \in V$. Subset of vertices $B \subseteq (V - \{S, T\})$.

   Solution type: a vertex $b \in B$

   Objective: Let G' be the graph resulting from G by removing all vertices in B. For each $b \in B$, define *ImprovedLength*(b) to be the shortest path from S to T in the graph G'_b which is G' with the vertex b included (with all of its incoming and outgoing edges.)

   We wish to find the vertex in B such that *ImprovedLength*(b) is the shortest.

(7 points for reasonably efficient correct algorithm (with correctness proof), 5 points for correct time analysis, and 3 points for efficiency of your algorithm.)

Solution:

Algorithm Description: Create G^* from G by removing all outgoing edges from all vertices in B. In other words, remove all edges of the form (b, u) such that b is in B.

Run Dijkstra(G^*, S) and find $dist_S(b)$ for all b in B.

Create G^{**} from G by first reversing G then by removing all outgoing edges from any vertex in B. In other words, remove all edges of the form (b, u) such that b in B in G^R.

Run Dijkstra(G^{**}, T) and find $dist_T(b)$ for all b in B.

Then calculate: $dist_S(b) + dist_T(b)$ for all b in B and return the vertex b with the least such value.

Justification of correctness:

Suppose that the algorithm returned the vertex v. For the algorithm to be correct, we must show that for any other vertex $b \in B$ that *ImprovedLength*(v) is smaller than *ImprovedLength*(b).

proof:

First let's prove the following Claim:

> **Claim:** For any vertex $b \in B$, $dist_S(b)$ is the length of the shortest path from S to b in the graph G'_b.
> *proof of claim:* Any path in G^* that goes from S to b cannot go through any other vertex in B because all vertices in B have been turned into sinks so they cannot be intermediate vertices in the graph. Therefore, any path in G^* from S to b must be a path in G'_b and so $dist_S(b)$ is the length of the shortest path from S to b in G'_b. (similar argument for $dist_T(b)$.)

Assume toward contradiction that the algorithm returns v but that there exists another vertex b such that *ImprovedLength*(v) > *ImprovedLength*(b).

This means that the shortest path p_v in G'_v from S to T is strictly longer than the shortest path p_b in G'_b from S to T, in other words $len(p_v) > len(p_b)$.

The path p_b does not exist in G'_v because if it did then p_b would potentially be the shortest path in G'_v.

The vertex b must be part of the path p_b because if it wasn't, then p_b would exist in G'_v.

$$len(p_b) = dist_S(b) + dist_T(b) \leq dist_S(v) + dist_T(v) \leq len(p_v) < len(p_b)$$

Explanation: $len(p_b) = dist_S(b) + dist_T(b)$ which is less than or equal to $dist_S(v) + dist_T(v)$ because v was output by the algorithm. But $dist_S(v) + dist_T(v) \leq len(p_v)$ and $len(p_v) < len(p_b)$. Therefore we have shown that $len(p_b) < len(p_b)$ which is impossible.

Runtime: It will take $O(|V| + |E|)$ to create G^* and G^{**}. Running Dijkstra's on G^*, G^{**} takes $O((|V| + |E|) \log |V|)$ each. Then looping through all vertices of B and computing and finding the minimum in terms of $dist_S(b) + dist_T(b)$ is $O(|B|) = O(|V|)$.

Overall the algorithm will have a runtime of $O((|V| + |E|) \log |V|)$ (or $O(|V|^2)$ if you use an array with dijkstra's)

3. Suppose you are planning on traveling from city A to city B by way of a sequence of flights during a particular day. You want to arrive to B early but you also want to save some money. You devise a metric called *cost-time* to measure if one flight sequence is better than the other. Let's say that flight sequence 1 arrives in B at h_1 o'clock (measured in 24 hour time) and costs d_1 dollars whereas flight sequence 2 arrives in B at h_2 o'clock and costs d_2 dollars. We say that the *cost-time* of flight sequence 1 is $h_1 * d_1$ and the *cost-time* of flight sequence 2 is $h_2 * d_2$. Our goal is to find the flight sequence that takes us from A to B with the minimum *cost-time*. (Assume that you can land on one flight and catch the next flight immediately after. Also assume that all flights are on time.)

For example, suppose we are traveling from San Diego to Boston. Suppose the first flight sequence we could take is:

d_i	a_i	DC_i	AC_i	c_i
6:30am	11:00am	SAN	DEN	150
12:30pm	3:30pm	DEN	DFW	50
3:30pm	6:30pm	DFW	BOS	100

This arrives to BOS at 6:30pm and costs a total of 300 dollars.
This gives it a *cost-time* of $(300)(18.5) = 5550$. (6:30pm is 18:30 in 24 hour time.)

Suppose the second flight sequence we could take is:

d_i	a_i	DC_i	AC_i	c_i
9:00am	2:00pm	SAN	ATL	300
3:00pm	5:00pm	ATL	BOS	100

This arrives to BOS at 5:00pm and costs a total of 400 dollars.
This gives it a *cost-time* of $(400)(17) = 6800$. (5:00pm is 17:00 in 24 hour time.)

Design an algorithm that takes as input: departure city A and a destination city B and a list of n flights $F_1, \ldots, F_n$ such that each flight $F_i = (d_i, a_i, DC_i, AC_i, c_i)$ includes the departure time d_i, arrival time a_i, departure city DC_i, arrival city AC_i and cost c_i and the output should be the value of the minimum *cost-time* out of all the flights that go from A to B.

(7 points for reasonably efficient correct algorithm (with correctness proof), 5 points for correct time analysis, and 3 points for efficiency of your algorithm.)

Algorithm:

- Create a list of all possible pairs of (city,time) that are either departure city and departure time or arrival city and arrival time. Sort all pairs first by city then by time.
- Create a graph G with a vertex for each possible (city,time) pair (Thus creating no more than $2n$ vertices.)
- Suppose that you have the vertices $(K, t_1), \ldots, (K, t_k)$ for a particular city K and times $t_1, \ldots, t_k$ sorted by times. Then for each $i = 1, \ldots, k-1$, create a directed edge from (K, t_i) to (K, t_{i+1}) of weight 0 (Thus creating no more than $2n$ edges.)
- Then create a directed edge from (K, t) to (K', t') if there is a flight that goes from (K, t) to $K', t')$ with the weight of the edge equal to the cost of the flight. (Thus creating exactly n edges.)
- Then run Dijkstra's from the vertex (x, y) where $x = A$ and y is the earliest time of any flights leaving from A (remember they are sorted so it is easy to find.)
- Loop through the vertices (x, y) with $x = B$ and compute the *cost-time* for each vertex and return the minimum *cost-time* among them.

Correctness: It is sufficent to prove that the dist values were set correctly for all vertices. To do this, we must prove the following claim:

Claim: A path in the graph of length c corresponds to a valid sequences of flights of total cost c.

> *proof:* Suppose there is path in the graph $p = e_1, \ldots, e_k$. Then if e_i is an edge between two vertices with the same city, then they only go forward in time so this corresponds to waiting in the same airport from one time to a later time and the cost for these edges is 0. If e_i is a flight edge, then that means that you can take the flight from the departure city,time pair to the arrival city,time pair and that edge will cost the price of the flight. In both cases, these are valid actions and so the entire path corresponds to a sequence of valid actions and since the length of a path is the sum of all the edges, it corresponds to the cost of the sequence of flights.
> (other direction similar)

So, with the claim above, dist values from Dijkstra's will correspond to the cost it takes to get to each vertex. In particular, the dist value for all vertices (x, y) with $x = B$ are correctly set to the minimum cost sequence of flights to get you to B at that time y. So the *cost-time* can be computed for each of these vertices by simple multiplication and then we can just return the minimum.

Runtime Analysis:

Creating the list of possible pairs is $O(n)$. Sorting this list takes $O(n \log(n))$. All in all, there are at most $2n$ vertices and at most $3n$ edges, so running Dijkstra's on this graph takes $O((|V| + |E|) \log |V|) = O((2n + 3n) \log(2n)) = O(n \log n)$. Then computing all of the *cost-time* values and finding the minimum takes at most $O(n)$. The overall runtime is $O(n \log n + n \log n + n) = O(n \log n)$.

4. For some non-negative integer d, we say that a d-regular graph is an undirected graph such that each vertex has a degree of d.

   Suppose you are given access to a *connected* d-regular graph $G = (V, E)$ and two vertices $s \in V$, $t \in V$. You wish to find the shortest path from s to t. (We can assume that the graph is very large and that we want to avoid having to look at the entire graph. So, for any vertex, you can look at its list of neighbors without having to look at the entire graph.)

   We can alter BFS so that it takes both s and t as inputs and when we reach t, we can stop so that we don't have to continue to explore unnecessary vertices.

```
BFS2(G, s, t):
(1)  for all u ∈ V:
(2)      dist(u) = ∞
(3)  dist(s) = 0
(4)  Q = [s]
(5)  while Q is not empty:
(6)      u = eject(Q)
(7)      for all edges (u, v) ∈ E:
(8)          if dist(u) = ∞:
(9)              inject(Q, u)
(10)             dist(v) = dist(u) + 1
(11)             if v == t:
(12)                 break loop
```

   (Notice that `BFS2` is almost identical to `BFS` except for lines 11 and 12.)

   (a) i. Give an argument about why `BFS2` will correctly assign $dist(t)$ to the length of the shortest path from s to t.
   (3 points)

   Solution:
   We showed in class that every time the dist(v) value was updated, it was the correct shortest distance. So after we set dist(t), we do not need to look any more in the graph and we can terminate the procedure.

   ii. Assuming that ℓ is the length of the shortest path from s to t, what is the worst-case runtime of `BFS2` in terms of d and ℓ? (your answer should be in big-O notation in terms of d and ℓ.)
   (4 points)
   There is a lemma that states that for every possible distance $i = 1...L$, at some point in BFS, the queue will contain exactly all of the vertices that are distance i away from s.
   Basing our argument off of this: In the worst case there are: 1 vertex of distance 0 (s itself)
   d vertices of distance 1

$d(d-1)$ vertices of distance 2
$d(d-1)^2$ vertices of distance 3 ...
....
$d(d-1)^{\ell-1}$ vertices of distance ℓ
this means that in the worst case, we have to enqueue all of these vertices which is: $1 + d\sum_{k=0}^{\ell-1}(d-1)^k = O(d^\ell)$ times to enqueue. So the algorithm will take $O(d^\ell)$.

(b) Design an algorithm that finds the shortest path from s to t in G that is more efficient than `BFS2`.

i. Algorithm description and correctness argument. (4 points)

Algorithm Description: Run two instances of BFS in the following way:
Initialize two arrays: `dist_s` and `dist_t` that will keep track of shortest distance from s and t, respectively.
Initialize two different queues $Q_s = [s]$ and $Q_t = [t]$.
Then start BFS on s and begin by pushing all neighbors of s into Q_s so that Q_s is filled with all vertices v with dist_s$(v) = 1$. Whenever Q_s tries to push a vertex with dist_s$(v) = 2$, jump to BFS on t. Then push all neighbors of t into Q_t in the same way except that for each vertex v, when you set the dist_t(v) value, check to see if dist_s$(v) \neq \infty$. If so, then add the two dist values together and return that value. Otherwise, jump back to BFS on s.
Jump back and forth in this manner, filling Q_s with all vertices of a particular distance then jumping to Q_t and checking and jumping back.
Justification: This procedure will find the shortest path by way of a vertex v in the middle of the path. (for the sake of simplicity, imagine that ℓ is even so that dist_s(v) $= \ell/2$ and dist_t$(v) = \ell/2$. Suppose by contradiction that there is a shorter path of length ℓ'. Then let v' be the middle vertex of this path. Then dist_s$(v') = \ell'/2 < \ell/2$ and dist_t$(v') = \ell'/2 < \ell/2$ so this vertex would have been found first.

```
BFS2way(G, s, t):
(1)  for all u ∈ V:
(2)      dist_s(u) = ∞
(3)      dist_t(u) = ∞
(4)  dist_s(s) = 0
(5)  dist_t(t) = 0
(6)  Q_s = [s]
(7)  Q_t = [t]
(8)  jump_dist = 0 # this value will keep track of when to jump.
(9)   ##########################(starting from s)
(10) while Q_s is not empty:
(11)     u = eject(Q_s)
(12)     for all edges (u, v) ∈ E:
(13)         if dist_s(u) = ∞:
(14)             dist_s(v) = dist_s(u) + 1
(15)             inject(Q_s, u)
(16)             if dist_s(v) == jump_dist:
(17)                 jump to line (19)
(18)  ##########################(starting from t)
(19) while Q_t is not empty:
(20)     u = eject(Q_t)
(21)     for all edges (u, v) ∈ E:
(22)         if dist(u) = ∞:
(23)             dist_t(v) = dist_t(u) + 1
(24)             inject(Q_t, u)
(25)             if dist_s(v) ≠ ∞:
(26)                 return dist_s(v) + dist_t(v)
(27)             if dist(v) == jump_dist:
(28)                 jump_dist = jump_dist + 1
(29)                 jump to line (10)
```

ii. Runtime analysis in terms of d and ℓ. (4 points)

Runtime: The vertex v that terminates the algorithm will be halfway between s and t ($\ell/2$). by a similar argument in part (a), in the worst case, there will be $1 + d\sum_{k=0}^{\ell/2-1}(d-1)^k$ from each procedure which will result in $O(d^{\ell/2})$.

(This is actually a big improvement. Imagine if $d = 100$ and $\ell = 5$, then $d^\ell = 10,000,000,000$ and $d^{\ell/2} = 10^5 = 10,000$.)