CSE 101: Homework #3

Due on Apr 24, 2024 at 23:59pm

Professor Jones

Ray Tsai

A16848188

Problem 1

You are given a schedule for n buses in the city. The schedule is given to you as a 2-dimensional array B of dimensions $n \times k$. Each bus has k stops and each entry of the array is an ordered pair $B[i, j] = (S_{i,j}, t_{i,j})$ which means that the j^{th} stop that bus i makes is at station $S_{i,j}$ at time $t_{i,j}$ (measured in minutes counted from the beginning of the day.) Each row is ordered by times (i.e., $t_{i,1} < t_{i,2} < \cdots < t_{i,k}$).

You can transfer from bus a to bus b at stop X if there exists entries $B[a, j] = (X, t_{a,j})$ and $B[b, j'] = (X, t_{b,j'})$ and $t_{a,j} < t_{b,j'}$.

(a) Given bus stations X and Y and a starting time T, design an algorithm that returns the earliest time you can reach station Y from station X starting any time after time T using a sequence of transfers.

Proof. Let V be the set of stations. Consider the following algorithm:

1. **for all** $v \in V$:
2. $\quad$ $time(v) = \infty$
3. $\quad$ Initialize array $schedule(v)$
4. $time(X) = T$
5. **for each** entry in B:
6. $\quad$ $append(schedule(S_{i,j}), (i, j))$
7. $H = makequeue(V)$ (using $time$ values as keys)
8. **while** $|H| > 0$
9. $\quad$ $u = deletemin(H)$
10. $\quad$ **For each** $(i, j) \in schedule(u)$ such that $t_{i,j} > time(u)$:
11. $\quad\quad$ **For each** l from j to k:
12. $\quad\quad\quad$ $time(S_{i,j}) = min(time(S_{i,j}), t_{i,j})$
13. $\quad\quad\quad$ $decreaseKey(H, S_{i,j})$
14. **return** $time(Y)$

This algorithm is a modified version of Dijkstra's, with an additional step of constructing a bus schedule with respect to each station.

We now give a justification for the correctness of the algorithm. We first note that step 10 of the algorithm ensures any transfers taken to be valid. Thus, given any vertex v, there is always a valid sequence of transfers such that the number of transfers is equal to $step(v)$ in any iteration. It remains to show that the sequence of transfers is the fastest at the end of the algorithm.

Let R_n be $V \backslash H$ at the end of nth iteration of the while loop, and let $\delta(v)$ be the earliest time v can be reached from X, and let $l(S)$ be the time that the sequences of transfers S ends. We show that $time(v) = \delta(v)$ for all $v \in R_n$ by induction on n. Since $R_1 = \{X\}$ and $time(X) = T = \delta(X)$, the base case is done. Suppose $n > 1$. Let u be the last vertex added to R_n. By induction, $time(v) = \delta(v)$ for all $v \in R_{n-1}$, and thus it remains to show that $time(u) = \delta(u)$. Suppose for the sake of contradiction that the fastest sequence of transfer from X to u is S and $l(S) < time(u)$, say $S = s_0 s_1 \dots s_{n-1} s_n$, where $s_0 = X$ and $s_n = u$. Suppose s_i is the first station in S which is not in R_{n-1}, and let $S_k = s_0 s_1 \dots s_k$, for $0 \leq k \leq n$. Note that we may make the subsequential transfers in S if we arrive at s_i earlier than $l(S_i)$, as we can wait at the station. Hence, as $time(s_{i-1}) \leq l(S_{i-1})$ and $s_{i-1} \in R_{n-1}$,

$$time(s_{i-1}) + (l(S_i) - l(S_{i-1})) \leq l(S_{i-1}) + (l(S_i) - l(S_{i-1})) \leq l(S),$$

Since s_i is immediate to stations in R_{n-1}, the while loop forces the $time(s_i)$ to be the earliest time you can travel from X to s_i via the stations in R_{n-1}, and so

$$time(s_i) \leq time(s_{i-1}) + (l(S_i) - l(S_{i-1})).$$

But then the algotithm picked u over s_i at step 9, so

$$time(u) \leq time(s_i) \leq time(s_{i-1}) + (l(S_i) - l(S_{i-1})) \leq l(S) < time(u),$$

contradiction. Hence, $time(u) = \delta(u)$, and this completes the induction. By the time the algorithm terminates, $V = R_n$, and the result follows.

Finally, we give a runtime analysis of the algorithm. Suppose we use the binary heap, which takes $O(\log |V|)$ for *insert*, *decreaseKey*, and *deletemin*. The first for loop runs over V for initialization, which takes $O(|V|)$ time. The second for loop runs through all entries of B, which takes $O(nk)$ time. *makequeue* takes $O(|V|)$ time. The while loop goes through all $|V|$ stations, and the inner for loop at 10. goes through at most $\sum_{u \in V} |schedule(u)| = |B| = nk$ entries, each entry runs *decreaseKey* at most k times. Hence, the while loop takes $O(|V|(\log |V| + nk^2 \log |V|)) = O(nk^2 |V| \log |V|)$ time. In total, the algorithm takes $O(nk^2 |V| \log |V|)$ time. □

(b) Given bus stations X and Y and a starting time T, design an algorithm that returns the fewest number of transfers necessary to reach station Y from station X starting any time after time T using a sequence of transfers.

Proof. Let V be the set of stations. Consider the following algorithm:

1. **for all** $v \in V$:
2. $\quad time(v) = \infty$
3. $\quad step(v) = \infty$
4. $\quad$ Initialize array $schedule(v)$
5. $time(X) = T$
6. $step(X) = 0$
7. **for each** entry in B:
8. $\quad append(schedule(S_{i,j}), (i,j))$
9. $H = makequeue(V)$ (using *step* values as keys)
10. **while** $|H| > 0$
11. $\quad u = deletemin(H)$
12. $\quad$ **For each** $(i,j) \in schedule(u)$ such that $t_{i,j} > time(u)$:
13. $\quad\quad$ **For each** l from j to k:
14. $\quad\quad\quad step(S_{i,j}) = min(step(S_{i,j}), step(u) + 1)$
15. $\quad\quad\quad decreaseKey(H, S_{i,j})$
16. **return** $step(Y)$

This algorithm is a modified version of Dijkstra's, with an additional step of constructing a bus schedule with respect to each station.

We now give a justification for the correctness of the algorithm. We first note that step 12 of the algorithm ensures any transfers taken to be valid. Thus, given any vertex v, there is always a valid sequence of transfers such that the number of transfers is equal to $step(v)$ in any iteration. It remains to show that the sequence of transfers is the shortest at the end of the algorithm.

Problem 1 continued on next page...

Let R_n be $V \backslash H$ at the end of nth iteration of the while loop, and let $\delta(v)$ be the minimum number of transfers v can be reached from X, and let $|S|$ be the number of transfers in a sequences of transfers S. We show that $step(v) = \delta(v)$ for all $v \in R_n$ by induction on n. Since $R_1 = \{X\}$ and $step(X) = 0 = \delta(X)$, the base case is done. Suppose $n > 1$. Let u be the last vertex added to R_n. By induction, $step(v) = \delta(v)$ for all $v \in R_{n-1}$, and thus it remains to show that $step(u) = \delta(u)$. Suppose for the sake of contradiction that the minimum sequence of transfers from X to u is S and $|S| < step(u)$, say $S = s_0 s_1 \dots s_{n-1} s_n$, where $s_0 = X$ and $s_n = u$. Let $S_k = s_0 s_1 \dots s_k$, for $0 \leq k \leq n$. Suppose s_i is the first station in S which is not in R_{n-1}. Since $s_{i-1} \in R_{n-1}$, by induction,

$$step(s_{i-1}) + 1 \leq |S_{i-1}| + 1 = |S_i| = |S| < step(u).$$

Since s_i is immediate to stations in R_{n-1}, the while loop forces $step(s_i)$ to be the minimum steps you can travel from X to s_i via the stations in R_{n-1}, and so

$$step(s_i) \leq step(s_{i-1}) + 1 \leq |S_{i-1}| + 1 = |S_i| = |S| < step(u).$$

But then the algorithm picked u over s_i at step 11, so $step(u) < step(s_i)$, contradiction.

Finally, we give a runtime analysis of the algorithm. Suppose we use the binary heap, which takes $O(\log |V|)$ for *insert*, *decreaseKey*, and *deletemin*. The first for loop runs over V for initialization, which takes $O(|V|)$ time. The second for loop runs through all entrys of B, which takes $O(nk)$ time. *makequeue* takes $O(|V|)$ time. The while loop goes through all $|V|$ stations, and the inner for loop at 10. goes through at most $\sum_{u \in V} |schedule(u)| = |B| = nk$ entries, each entry runs *decreaseKey* at most k times. Hence, the while loop takes $O(|V|(\log |V| + nk^2 \log |V|)) = O(|V| \log |V| nk^2)$ time. In total, the algorithm takes $O(|V| \log |V| nk^2)$ time. □

Problem 2

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):
for all u ∈ V:
   dist(u) = ∞
dist(s) = 0
Q = [s]
while Q is not empty:
   u = eject(Q)
   for all edges (u, v) ∈ E:
      if dist(v) = ∞:
         inject(Q, v)
         dist(v) = dist(u) + 1
         if v == t:
            break loop
```

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

Proof. Note that the only difference between `BFS2` and `BFS` is that `BFS2` stops once $dist(t)$ is modified. Since the $dist(t)$ in both `BFS2` and `BFS` would only be modified at most once, both algorithms would output the same value given the same input. Since `BFS` is already proven to be correct, then so is `BFS2`. □

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.)

(Note: You can use the fact 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.)

Proof. Since at some point in BFS, the queue will contain exactly all of the vertices that are distance $l-1$ away from s, which includes the node preceding t in its shortest path to s, `BFS2` is guaranteed to have visited t after processing all vertices in the queue at that moment. By the fact 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, BFS will visit all vertices of distance less than l before visiting vertices of distance l. Since each vertex $v \neq s$ has at most $d-1$ nonvisited neighbors (excluding the preceding vertex), there are at most $d(d-1)^{k-1}$ vertices that has distance k from s by induction. But then there are at most

$$1+\sum_{k=1}^{l} d(d-1)^{k-1} = 1 + d\cdot\frac{(d-1)^l-1}{d-2} = 1-\frac{d}{d-2}+\frac{d(d-1)^l}{d-2}$$

vertices which have distance at most l from s. Hence, in the worst case, t would be the $(1-\frac{d}{d-2}+\frac{d(d-1)^l}{d-2})$th visited vertex, and thus the algorithm has a worst case runtime of

$$O\left(|V|+1-\frac{d}{d-2}+\frac{d(d-1)^l}{d-2}\right) = O(d^l).$$

□

(b) Design an algorithm that finds the shortest path from s to t in G that runs in time $O(d^{l/2})$.

i. High-level algorithm description (No correctness proof necessary.)

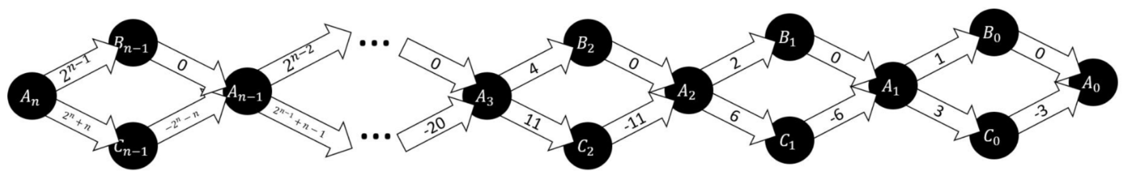
Proof. We run `BFS2` simultaneously from both s and t, and terminate once both `BFS2` have visited the same node. □

ii. Runtime analysis.

Proof. At the worst case, both `BFS2` meet at the middle of the shortest path between s and t, say vertex v. But then the shortest path between s, v and s, t have length $l/2$. By (a).ii, the runtime for each `BFS2` is $O(d^{l/2})$, and thus the total runtime is $O(2d^{l/2}) = O(d^{l/2})$. □

Problem 3

For all integers $n \geq 0$, consider the graph $H(n)$ pictured below:



That is, A_0 has no outgoing neighbors and the adjacency list for each other vertex is:

- $A_k : (B_{k-1}, 2^{k-1}), (C_{k-1}, 2^k + k)$
- $B_k : (A_k, 0)$
- $C_k : (A_k, -2^k - k)$

(a) Let $maxdist(v)$ be the maximum dist value that $dist(v)$ is set to (besides ∞) after running Dijkstra's on $H(n)$ starting at A_n.

Prove by induction that $maxdist(v) \leq 2^n + n$ for all $v \in H(n)$.

Proof. We proceed by induction on n. Since A_0 has no out going edge, $maxdist(A_0) = 0 \leq 2^n + n$ and the base case is done. Suppose $n \geq 1$. Since there is only one path to either B_{n-1} or C_{n-1} from A_n, $maxdist(B_{n-1}) = 2^{k-1}$ and $maxdist(C_{n-1}) = 2^k + k$, both of which are at most $2^n + n$. There are two possible paths, $A_nB_{n-1}A_{n-1}$ and $A_nC_{n-1}A_{n-1}$, from A_n to A_{n-1}, each having weights 2^{n-1} and 0, respectively. Since $2^{n-1} < 2^k + k$, $maxdist(A_{n-1})$ would be set to 2^{n-1}. But then $2^{n-1} < 2^n + n$, so the algorithm would prioritize A_{n-1} over C_{n-1}. By induction,

$$maxdist(v) \leq maxdist(A_{n-1}) + (2^{n-1} + n - 1) \leq 2^{n-1} + (2^{n-1} + n - 1) = 2^n + n - 1 \leq 2^n + n,$$

for all $v \in H(n-1)$. Hence, the inequality holds true for all $v \in H(n)$. $\square$

(b) Prove by induction that after running Dijkstra's on $H(n)$, starting at A_n, the vertex A_0 has been ejected from the priority queue 2^n times. (You can use the previous part.)

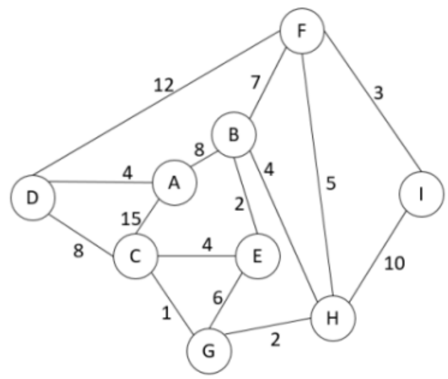
Proof. We proceed by induction on n. For $n = 0$, $H(0)$ consists of a single vertex A_0 and thus the algorithm would only eject it once. Suppose $n \geq 1$. Since $dist(A_{n-1})$ would initially be assigned $2^{n-1} < 2^n + n$, A_{n-1} would be prioritised over C_{n-1}. By (a), $maxdist(v) \leq 2^{n-1} + n - 1$ for all $v \in H(n-1)$. But then $2^{n-1} + 2^{n-1} + n - 1 = 2^n + n - 1 < 2^n + n$, so all vertices of subscripts less than $n - 1$ would be prioritised over C_{n-1}. Hence, the process of the algorithm before revisiting C_{n-1} is equivalent to running Dijkstra's algorithm on $H(n-1)$, which would ejected A_0 2^{n-1} times, by induction. But then after revisiting C_{n-1}, the algorithm push A_{n-1} back to the priority queue, and the remaining process is also equivalent to running Dijkstra's algorithm on $H(n-1)$, which would eject A_0 another 2^{n-1} times. Hence, A_0 would be ejected 2^n times in total. $\square$

(c) Give a lower bound for the runtime of Dijkstra's on this graph in Ω notation in terms of the number of vertices N.

Proof. Given N vertices in the graph, there would be $(N-1)/3$ diamonds. Thus, by (b), the algorithm would at least process A_0 $2^{(N-1)/3}$ times, which has a lower bound of $\Omega(2^{N/3})$. $\square$

Problem 4

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.

Proof. Consider the following table:

Iteration	A	B	C	D	E	F	G	H	I
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	0	8	15	4	∞	∞	∞	∞	∞
2	0	8	12	4	∞	16	∞	∞	∞
3	0	8	12	4	10	15	∞	12	∞
4	0	8	12	4	10	15	16	12	∞
5	0	8	12	4	10	15	13	12	∞
6	0	8	12	4	10	15	13	12	22
7	0	8	12	4	10	15	13	12	22
8	0	8	12	4	10	15	13	12	18
9	0	8	12	4	10	15	13	12	18

$\square$

(b) Draw the shortest path tree.