CS330HW6

ras70

October 2017

Problem 3A

This algorithm requires a similar implementation to Dijkstra's algorithm. Dijkstra's algorithm defines $dis[u]$ to be the minimum distance from the start s to u at a given iteration of the algorithm (Reference Appendix). The algorithm for this problem will define the same $dis[u]$. For an edge $e = (u, v)$ Dijkstra's algorithm only considers the weight of this edge, which in this problem, corresponds to the time it takes to get from u to v, or $l(e)$. However, for this algorithm to work correctly the **weight function** needs to be updated to accommodate for the potential waiting time at each station/vertex.

For example, let's say the algorithm is currently visiting Stop A and attempting to travel to Stop B, where Stop A and B are directly connected and the time to get from Stop A to Stop B is x. More formally, for the edge e, where e is between A and B, $l(e) = x$. If Alice arrive in A, at time y, the soonest time she can get to B is *not* simply $x + y$ as Dijkstra's would compute. Rather the soonest Alice can get to B is the earliest time she can leave A, which might be greater than y, plus x.

To account for this waiting time a couple of cases must be considered. Let the algorithm be at a time step in which u is currently being visited/considered. Let e be a given edge from u, connecting u to v. To calculate $dis[v]$ the earliest time Alice can leave u must be known. Therefore, the algorithm's weight function will be a function of $dis[u], t(e), and int(e)$ - since all these parameters affect the start time.

If Alice arrives before the first bus departs from u to v, then she will take the first bus. In this case, the soonest time she can get to v is the time the first bus leaves u plus the time it takes to travel from u to v, or $t(e) + l(e)$.

If Alice arrives at u just as a bus (first, second, third,..etc) is leaving she will take that bus. Therefore, in this instance her departure time is simply the time she arrives at u, or $dis[u]$, and her time to get to v is $dis[u] + v$. The condition that must be met in order for Alice to arrive as a bus departs is $(t(e) - d(e))\%int(e) = 0$. In other words, if the time Alice arrives minus the time the first bus left is a multiple of the interval length, a bus will be arriving just as Alice gets there.

Lastly, if Alice arrives in the middle of an interval, after the first bus arrives, she must wait for the next bus. When she arrives at u the bus she just missed

left $(dis(u) - t(e))\%int(e)$ minutes ago. Therefore, the next bus will come in $int(e) - (dis(u) - t(e))\%int(e)$ minutes. To account for this wait, the time of departure is $dis(u)+int(e)-(dis(u)-t(e))\%int(e)$ and the time Alice can arrive at v is $dis(u) + int(e) - (dis(u) - t(e))\%int(e) + l(e)$.

The three cases explained above are documented more concisely below:

Case 1: $dis[u] < t(e)$

In this case Alice arrives at u before the first bus departs for u to v. Therefore, the time of departure from u is $t(e)$ and $dis[v] = t(e) + l(e)$.

Case 2: $dis[u] \geq t(e)$ & $(t(e) - d(e))\%int(e) = 0$

In this case Alice arrives at u as a bus departs for u to v. Therefore, the time of departure from u is $dis[u]$ and $dis[v] = dis[u] + l(e)$.

Case 3: $dis[u] \geq t(e)$ & $(t(e) - d(e))\%int(e) \neq 0$

In this case Alice arrives at u as in the middle of an interval and must wait for the next bus. Therefore, the time of departure from u is $dis[u]$ and $dis[v] = dis[u] + l(e) + w_a$, where w_a is the wait time. The wait time is given as $w_a = int(e) - (dis(u) - t(e))\%int(e)$

Given these three cases the modified expression for $dis[v]$ can be given as:

$$dis[v] = \begin{cases} t(e) + l(e) & dis(u) < t(e) \\ dis(u) + l(e) & t(e) \geq dis(u) \text{ \& } (dis(u) - t(e))\%int(e) = 0 \\ dis(u) + l(e) + int(e) - (dis(u) - t(e))\%int(e) & \text{otherwise} \end{cases}$$

With these modifications in mind the algorithm pseudo-code is as follows:

```
shortestCommute(s, t) {
      initialize dis[u] to be all infinity
      FOR all edges e = (s,u)
            IF (dis[s] < t(e))
                initialize dis[u] = t(e) + l(e)
            ELSE IF ((dis(u) - t(e)) % int(e) == 0)
                initialize dis[u] = l(e)
            ELSE
                initialize dis(u) = l(e) + int(e) - (dis(u) - t(e))%int(e)
      FOR i = 2 to n
             Among all vertices that are not visited,
             find the one with smallest distance, call it u.
             Mark u as visited
             FOR all edges e = (u,v)
                    IF (dis[u] < t(e))
                        possibleTime = t(e) + l(e)
```

```
                    ELSE IF ((dis(u) - t(e)) % int(e) == 0)
                        possibleTime= dis(u) + l(e)
                    ELSE
                        possibleTime = dis(u) + l(e) + int(e) - (dis(u) - t(e))%int(e)

                    IF possibleTime < dis[v] THEN
                          dis[v] = possibleTime

    return dis[t]
}
```

It is important to note that the above algorithm calculates the minimum distance from every vertex s to every other vertex u in the graph. Since the destination t is known this is not strictly necessary. Rather, the algorithm can commence once t is visited. This is the case because once t is visited it is guaranteed to be the vertex of shortest distance from s, out of the remaining vertices that have yet to be visited. Since, all times of travel, earliest departure times, and intervals are positive, if t contains the shortest path from s considering the remaining set of unvistied vertices, no shorter path can be made from s to t using the remaining unvisited vertices.

Although the algorithm can exit after visiting t I thought it was best to calculate the distances for all vertices. This makes the algorithm more flexible as it can be run once and then be used to calculate the shortest path from s to any vertex in the graph. This may be helpful, for example, if s is Alice's home. She might want to know the shortest commute to a variety of locations. However, I did want to acknowledge at the very least that this was a stylistic choice and not the only possible implementation.

Running Time

This algorithm uses Dijkstra's algorithm with a modified transition function that has no effect on the running time. For Dijkstra's the closest vertex needs to be found n time and the edges need to be updated m times. The best implementation uses a Fibonacci heap, that provides access to vertices in $O(\log n)$ time and to edges in constant, $O(1)$, time. Therefore, the overall running time of this algorithm is,

$$O(m + n \log n)$$

Problem 3B

For every vertex u let $d[u]$ be the earliest time that Alice can reach u from s (starting at time 0). Suppose there is a subset of vertices S, we have already computed $d[u]$ for $u \in S$, and we know for any $v \notin S$, $d[v] \geq d[u]$.

Now, let v be a vertex not in S that has the minimum $d[v]$ value, prove that the best route to get to v only leaves the set S in the last step.

Consider the public transit system to be a graph G that connects stops as vertices to each other through edges (streets). Assume towards contradiction that a shortest path P exists from s to v that leaves the set S before the final step, where v has the minimum $d[v]$ value, as given in the problem statement. Since v has the minimum $d[v]$ value it must hold that:

$$d[v] \leq d[b]$$

$$\text{for all vertices } b \notin S$$

Let edge $e = (u_d, v_d)$ be the edge in which P first leaves S. Since the edge leaves S it must hold that $v_d \notin S$. Additionally, since it was assumed P leaves S before the last step, v_d cannot be v. If v_d and v were the same vertex then e would be the last step in the path P and P would leave S on the last step.

Let P_d be the part of the path that connect s to v_d and P_f be the part of the path from v_d to v. Given this,

$$w[P] = w[P_d] + w[P_f]$$

where $w[A]$ is equal to the weight of path A, which is equal to the sum of weights of all edges in A.

Since P is the shortest path from s to v, $w[P] = d[v]$. P_d is also a shortest path because any connected edges of a shortest path are also a shortest path. Therefore, $w[P_d] = d[v_d]$. This results in the expression,

$$d[v] = d[v_d] + w[P_f]$$

The path from any one vertex to another in G is non-negative. This is due to the fact that it is not possible to travel from one location to another in no or negative time. To see this more clearly please reference how the weight calculations between stops are formulated in *Problem 3A*. Therefore, since no negatively or zero weighted edges exist in G each path in G must have positive weight. This means $w[P_f] > 0$. Given this information,

$$d[v] > d[v_d]$$

However, since $v_d \notin S$ and the assumption made that $d[v]$ is minimal for all edges $\notin S$, it must hold that $d[v] \leq d[v_d]$. Therefore, a contradiction has been identified and the optimal path from s to v cannot leave S before the last step.

Appendix

```
Dijkstra(s)
      initialize dis[u] to be all infinity, prev[u] to be NULL
      For neighbors of s,
      initialize dis[u] = w[s,u], prev[u] = s
      Mark s as visited
      FOR i = 2 to n
             Among all vertices that are not visited,
             find the one with smallest distance, call it u.
             Mark u as visited
             FOR all edges e = (u,v)
                IF dis[u]+w[u,v] < dis[v] THEN
                    dis[v] = dis[u]+w[u,v]
                    prev[v] = u
```