Hui Lin Project 2 Writeup

Source used: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

1. Experiments Scheduling

(a) Describe the optimal substructure of this problem.

If a decision to schedule a student to particular steps exists in the optimal solution to the problem, then the optimal way to schedule the students to minimize switching also includes the optimal way to schedule the students to the remaining steps that have not been scheduled after the decision is made.

(b) Describe the greedy algorithm that could find an optimal way to schedule the students.

Our optimal choice at each step is as follows: the next student to be scheduled steps must be able to do the step with the lowest index, say step *k*, not already scheduled. So initially, we must pick a student who is able to do the first step. Among the students who are able to do step *k*, we pick the student who is able to do the longest consecutive streak of steps from *k*, i.e. of the students who are able to do steps *k, k + 1, k + 2, ... k + m*, we pick the student for which m is greatest.

(d) What is the runtime complexity of your greedy algorithm? Again, you don't need to factor in the setup of the lookup table, just your scheduling algorithm.

Let *m* be the number of students and *n* be the number of steps. The outermost while loop for my *scheduleExperiment* function runs *O(n)* times, and within each iteration of the while loop there is a for loop which runs *O(m)* times. Each iteration of the for loop calls the helper functions *getFurthestStreak* and *getEarliestJobScheduled* , which both run in *O(n)*. So, the entire solution runs in $O(n^2m)$.

(e) In your PDF, based on your answer to part b, give a full proof that your greedy algorithm returns an optimal solution.

Let G = $\{g_1, g_2, \ldots g_n\}$ be the solution generated by the greedy algorithm.

Let O = $\{o_1, o_2, \ldots o_m\}$ be the solution generated by an optimal algorithm.

Each element g_i of G and o_i of O represents a student doing a contiguous sequence of steps before switching out for another student. Hence, the number of switches using the greedy algorithm is |G| = n, and the number of switches using the optimal algorithm is |O| = m.

Let $f(g_1, g_2, .. g_j)$ be the total number of steps that have been scheduled so far given the scheduling of students from g_1 to g_j, and $f(o_1, o_2, \ldots o_j)$ be the total number of steps that have been scheduled so far given the scheduling of students from o_1 to o_j. We will show that G always stays ahead of O, i.e. $f(g_1, g_2, .. g_n) >= f(o_1, o_2, \ldots o_n)$ for all *n*, using induction.

Initial Step: $f(g_1) >= f(o_1)$. This is true by design of our greedy algorithm, since we schedule the student able to complete the largest contiguous streak of steps from the first step.

Inductive Step: Assume $f(g_1, g_2, \dots g_k) >= f(o_1, o_2, \dots o_k)$ for some $k >= 1$. g_{k+1} selects the student able to perform the longest contiguous steak of steps starting from step with index $f(g_1, g_2, \dots g_k) + 1$, and o_{k+1} selects the student able to perform the longest contiguous steak of steps starting from step with index $f(o_1, o_2, \dots o_k) + 1$.

For the optimal algorithm to catch up with the greedy algorithm, o_{k+1} would have to assign a student who is able to contiguously perform steps from index $f(o_1, o_2, \dots o_k) + 1$ to index $f(g_1, g_2, \dots g_{k+1})$. But if a such a student exists, that student would have been scheduled to g_{k+1} to complete steps $f(g_1, g_2, \dots g_k) + 1$ to $f(g_1, g_2, \dots g_{k+1})$ by the greedy algorithm. Hence the optimal algorithm can at best catch up with, but not overtake, the greedy algorithm.

Thus, we have shown by induction that with each switch, the greedy algorithm's progress as measured by the total number of steps completed so far can never be overtaken by some optimal algorithm.

Therefore, at completion, the greedy algorithm cannot have made more switches than the optimal algorithm, i.e. $|G| <= |O|$, which proves the greedy algorithm is optimal.

2. Public, Public Transit

(a) Describe an algorithm solution to this problem. Feel free to talk about how you would adapt an algorithm we covered in class.

We use Dijkstra's algorithm, while also considering that the weights of the edges are not fixed, but determined by factors such as the starting time, frequency of trains between each station, and the time of arrival at a station. In the code, there were a couple of formulas I used to help calculate the cost of choosing the next edge. The comments in the code explain how I calculated the cost of choosing the next edge by factoring in starting time, train frequency, and time of arrival at current station.

(b) What is the complexity of your proposed solution in (a)?

The calculations done on top of Dijkstra's algorithm are done in constant time, so the complexity of the proposed solution is equivalent to the complexity of Dijkstra's algorithm. In our current implementation, the time complexity is $O(v^2)$, where v is the number of vertices.

(c) See the file FastestRoutePublicTransit.java, the method "shortestTime". Note you can run the file and it'll output the solution from that method. Which algorithm is this implementing?

The method shortestTime is implementing Dijkstra's algorithm.

(d) In the file FastestRoutePublicTransit.java, how would you use the existing code to help you implement your algorithm? The existing code only handles one piece of data per edge, so describe some modifications.

I used the existing the code for Dijkstra's algorithm, but made the following modifications:

- In the original implementation, the edge weights are fixed. Here, the edge weight depends on the starting time, frequency of trains, and time of arrival at station.

- We first calculated the **time of arrival** at the current station u. Since times[u] is the current shortest time in the path up to current vertex u from the vertex S, the amount of time that has elapsed in commute by the time we've arrived at station u is times[u]. Hence, the arrival time at station u would simply be the starting time + amount of time spent in commute (or times[u]).

- Next, we need to calculate **how long we'd need to wait before we can board the next train** from station u to station v. Since the trains come at intervals like clockwork, the use of modulus intuitively comes into the equation. If we arrive the station before the first train from the station has departed, we're forced to wait until the first time arrives. Waiting time would be first[u][v] – arrivalTime. Otherwise, we can calculate how much time has elapsed since the previous train left with (arrivalTime – first[u][v]) % freq[u][v], which is essentially the amount of time which has elapsed since the first train from the station has left mod the frequency of trains. Waiting time can be calculated by subtracting that value from the frequency. Hence, the waiting time for the next train, which factors in our starting time, train frequency, and arrival time at the station, is an additional cost which is added to the edge weight.

- We break out of Dijkstra's once we have processed T, since we've found the shortest travel time from the source vertex to destination vertex T.

(e) What's the current complexity of "shortestTime" given V vertices and E edges? How would you make the "shortestTime" implementation faster? Describe any algorithm changes or data structure changes. What's the complexity of the optimal implementation?

The current complexity of shortestTime is $O(V^2)$. If the input graph is represented using an adjacency list, the algorithm can be improved to O(E log V) with a binary heap.