Experiments Scheduling

$A = \{A_1, A_2 \ldots A_m \ldots A_n\}$

a) Let Λ be the optimal sequence of students, in order, for the completion of the experiment.

Let $S(i)$ denote the largest continuous step subsequence of a student, A_m (ie. a sequence of continuous 1s) that starts at and includes i. Let j be the last step of $S(i)$, where $1 \leq i \leq j \leq n$. Then $A' = \{A_{m+1} \ldots A_n\}$ is an optimal solution to the subproblem on the steps from $j+1$ to n, determined by $S(j+1)$.

Note:

- Let $S(a) = 0, a > n$

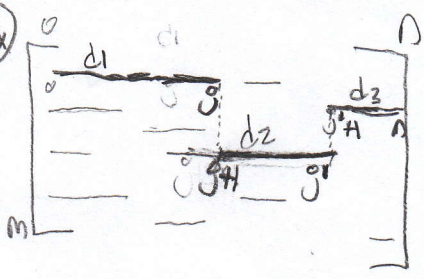
b)

At the start of our greedy algorithm, we will first look at the students who can perform step 1, and choose the student (greedy choice) with the greatest "depth" or in other words, the student who has the longest contiguous subarray of 1s where the start of the subarray is step 1. This will be our first student in our "optimal" sequence for steps 1 to some step j, $1 \leq j \leq n$. This procedure is repeated again where the first step we look at (or consider for the start of our contiguous subarray) is the next step after our previous longest contiguous subarray's last index. The procedure is finished once all students have been selected for such sequence, where the last student in the sequence performs step n of the experiment.

d)

At any given step, j, we are looking at all m students and choosing the student with the greatest "depth". If student k has the greatest depth, d, at step j, then for all students at step j, the algorithm would have to check against, at most, d steps after step j. Therefore at any given step j of the algorithm, we perform $O(m \cdot d)$ operations (as each operation is just a comparison, which is constant). Since each step must be complete, we have to go through all n steps at some point, therefore n can be seen as the sum of $d_1 + d_2 + d_3 \ldots d_\alpha$ ($\alpha = 1$ is the total number of switches we must perform. If $d_1 = n$, then we must perform no switches as there is a student who can do all steps. If all students can do every job (ie) each student's longest contiguous subarray is length $d_1 = n$ at step 1), we would, at worst case, check the entire matrix. Therefore the algorithm runs in $O(m \cdot n)$.

Ex



Note:

Once we choose a student at step j, with the longest contiguous subarray of length d, we do not check any steps before $j+d$ as those steps are completed.

e) Let OPT be our greedy algorithm.

Suppose an algorithm exist, ALG, that is more optimal than OPT. Then, ALG has less switches between students than OPT. For an arbitrary input, let the student sequence chosen be denoted:

$$OPT: A_1, A_2 \dots A_\ell$$
$$ALG: A_1, A_2 \dots A_r$$

where, by definition, $r < \ell$. Then, there exists a point of, A_k, divergence in the OPT and ALG solutions. Let $A_k \in OPT$, $A_m \in ALG$ be the diverging point in the sequence. In other words: OPT: $A_1, A_2 \dots A_{k-1}$

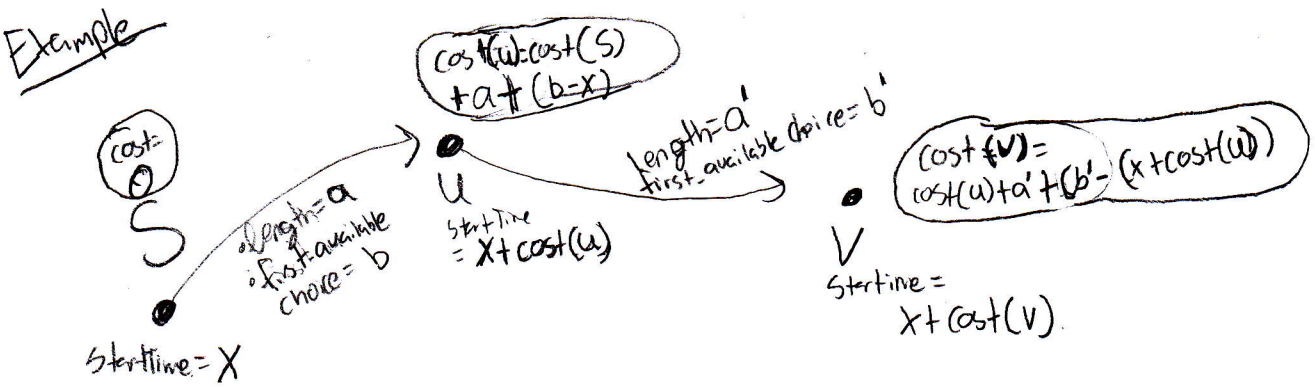
ALG: $\underbrace{A_1, A_2 \dots A_{m-1}}_{\text{All equal}}$. By definition, OPT looks for the student with the longest contiguous subarray (can do most consecutive jobs). Since A_k & A_m is where the algorithms diverge, the current step, j, would be the same at that point. Moreover, since the ALG is more optimal than OPT and $A_k \neq A_m$, A_m would be a student who could perform more consecutive jobs than A_k, or in other words, has a longer contiguous subarray. But OPT always chooses the longest contiguous subarray at a given step. Since ALG & OPT both are on step j before the point of divergence, ALG cannot possibly choose a longer contiguous subarray. Therefore the assumption that ALG exists is contradicted and OPT is the most optimal solution.

Public Transportation

(a)

This problem can be solved through a variation of Dijkstra's shortest path algorithm. On top of the edge weights, denoted by time to travel from some station, U, to another station, V, each vertex's time cost should also hold the accumulated wait time between stations based on frequency. At any given station, U, when determining the shortest path for all adjacent stations, we must consider the current cost to get to U from S (path lengths + wait time), along with the given start time we arrive to S in order to determine which train we may catch at each adjacent station. The rest of the algorithm shall follow the general implementation of Dijkstra's shortest path.

Example



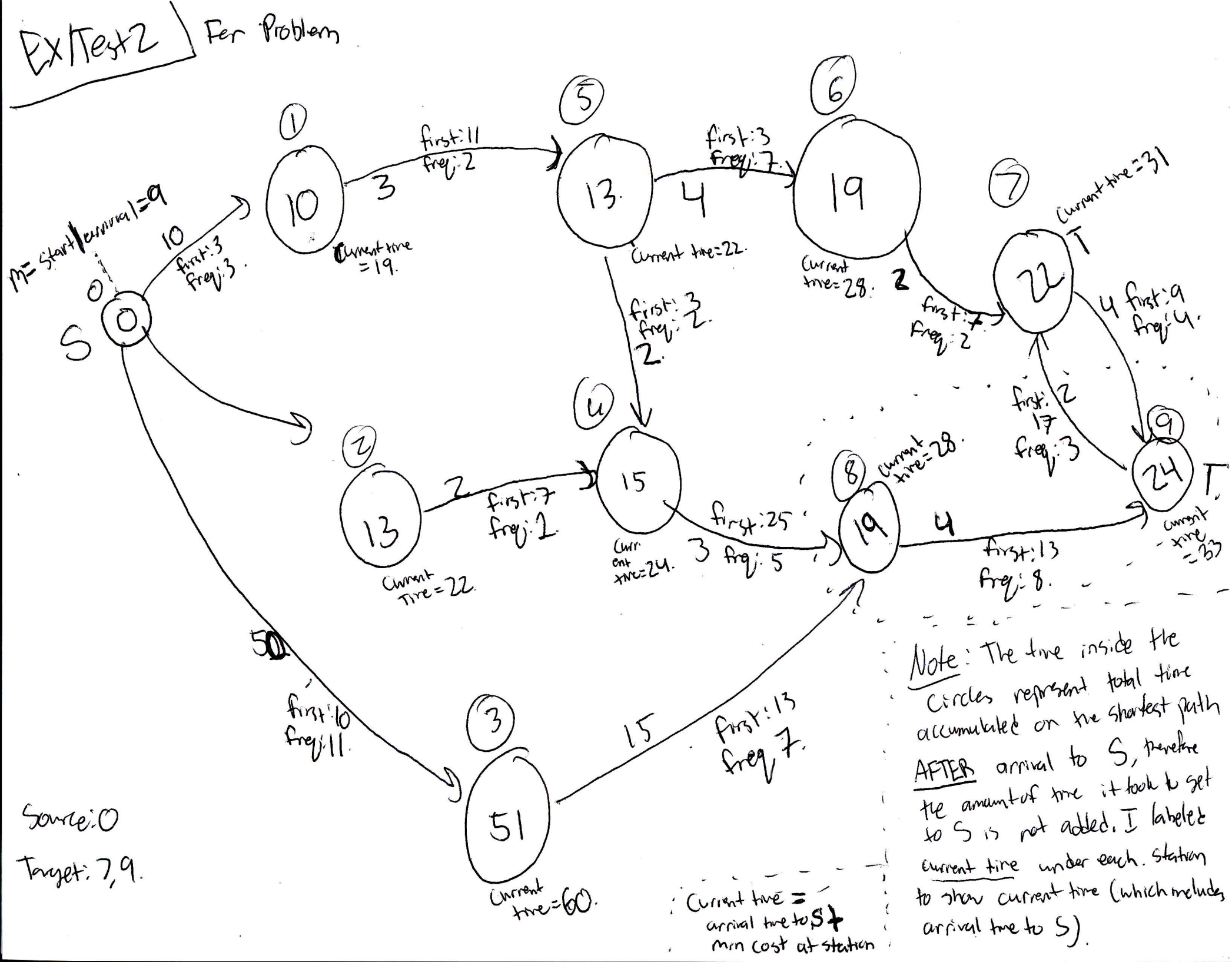
(b) Aside from the implementation for Dijkstra's algorithm, the algorithm in part (a) only does constant amount of work for each station visited, which includes determining the next available train that can be chosen given the current minimum time to get to that station, plus the start time. Since this is only a constant amount of work in addition, for each vertex, the asymptotic complexity of Dijkstra's algorithm does not change, so this algorithm runs in $\boxed{O(V \lg V + E \lg V)}$ using a priority queue or $O(V \lg V + E)$ using a Fibonacci heap.

(c) The algorithm implements Dijkstra's algorithm without the use of a priority queue to keep track of non-processed stations in descending order of their cost (time to get there from S).

(d) The current code handles the calculation of the length portion of the minimum cost for each station. Each station does not account for accumulated wait time from the minimum path to S. In order to determine that in the current implementation, we need to first determine the first available train we can grab at each station in the minimum path from S to any vertex, V. Each vertex/station should also determine the current time based on the start time at S and the cost to get to that vertex/station. This info will be used for each adjacent station for the first available train that can be grabbed. Therefore, when calculating the minimum cost for any station, we must include the cost of the adjacent neighbor with least cost to S, the length of that edge, and accumulated wait time from that adjacent neighbor.

(e) The time complexity of "shortestTime" is $O(V^2)$.

With some changes to the data structures, the time can be brought down to $O(V \lg V + E \lg V)$. Instead of traversing over each vertex to determine the next station we haven't processed with the smallest cost, which is $O(V)$, we can use a min-heap to keep track of the minimums for each iteration, which is $O(\lg V)$ per extract operation. The loop that checks for all adjacent nodes to the current iteration uses a adjacency matrix, which is $O(V)$ as the loop will check against all nodes. In order to reduce this, we can use a adjacency list instead, which if implemented with a hash table and arrays that hold neighboring vertices, will give us a runtime of $O(E)$ as each vertex will have access to only the adjacent edges. Assuming each adjacent vertex will have a shorter path through the current vertex being processed, we would need to update all adjacent vertices in the heap, which is an $O(\lg V)$ operation per vertex, which goes to $O(E \lg V)$ for all the vertices. So in total, for each vertex, $O(V)$, we extract the vertex from the heap, $O(\lg V)$, then perform $O(E \lg V)$ operations. Therefore the runtime can be brought down from $O(V^2)$ to $O(V \lg V + E \lg V)$. Using a Fibonacci heap can bring the runtime down further to $O(V \lg V + E)$.


Ex/Test2
For Problem
m= start/arrival=9
S
0
10
first:3
freq:3
1
10
Current time =19.
3
first:11
freq:2
5
13
Current time=22.
4
first:3
freq:7
6
19
Current time=28.
2
first:7
freq:2
7
22
Current time=31
T
4 first:9
freq:4
9
24
T
Current time =33
first:2
17
freq:3
first:3
freq:2
2
4
15
Current time=24.
3
freq:5
first:25
8
19
Current time=28.
4
first:13
freq:8.
2
13
Current Time=22.
2
first:7
freq:2
50
first:10
freq:11
3
51
Current time=60.
15
first:13
freq 7.
Source:0
Target: 7, 9.
Current time = arrival time to S + min cost at station
Note: The time inside the circles represent total time accumulated on the shortest path AFTER arrival to S, therefore the amount of time it took to get to S is not added. I labeled current time under each station to show current time (which includes arrival time to S).