Deepak Khemraj
CS323 - Professor Yao
Assignment #2 - Greedy and Graph Algorithms

Part One: Greedy Scheduler

The optimal substructure is as follows:

a)We can think of a subproblem as solving for which input schedule does the most tasks in order and then moving on to the next subproblem that does the same thing taking up where the last subproblem left off. Suppose the algorithm was given 6 steps and there was a student that could do steps 1,2,3,4,5,6. That's the best possible answer as it yields no switches. This however is unlikely to be the input of the algorithm and so the best we can do is solve up to a certain longest sequence of consecutive tasks at which point we are forced to switch to another participant schedule. Thus the solution to these subproblems then implies the solution to the whole problem.

b)What we are really looking for at any given iteration of a greedy algorithm is the student that can do the most tasks in consecutive order starting at task one. We locally select this from the pool of schedules. Suppose that t0 represents task 1 initially. We find a student that can perform the most tasks t1,t2…..tN. Once this student has been added to the pool we begin the process again but this time starting at tN + 1.

d)The greedy algorithm is O(n). We have a constant set of schedules so looping through them is only up to a constant time limit. The worst possible input is a set of schedules where each student can only do one unique task. For example if numTasks = 3 and student one can do t1,student 2 can do t2, and student 3 can only do t3. This

introduces the O(n) runtime each iteration the greedy algorithm only advances on one out of N task inputs. Typically the algorithm will perform better as students will be able to do more than one task.

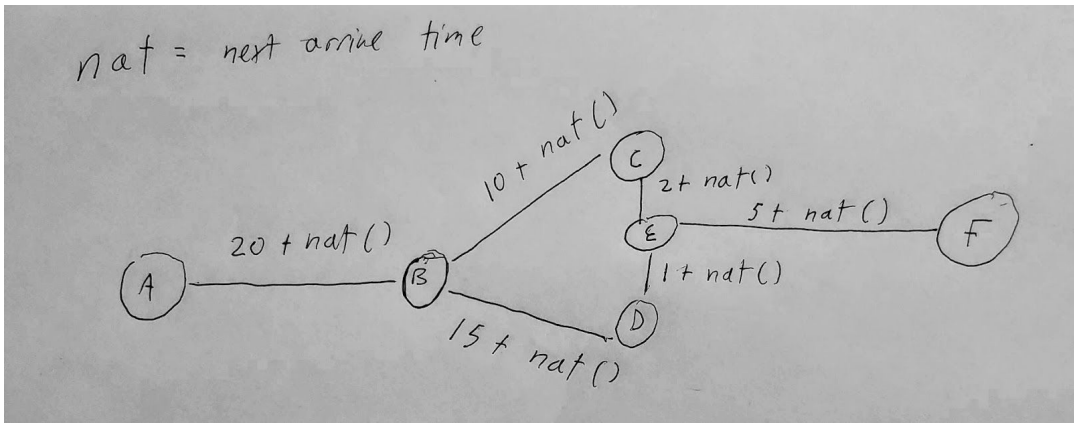
e)On any given iteration the greedy algorithm locally chooses the schedule that can do the most tasks in consecutive order. This implies a globally optimal solution. A switch always happens when we don't have a consecutive sequence as the gap in the sequence needs to be filled in. We pick the schedule that has the longest consecutive sequence to mitigate this as much as possible. Lastly, subsequent iterations of the algorithm pick up where the last iteration and any subsequent iteration is left with the fewest remaining tasks to do in order. Thus the greedy algorithm keeps up with any other implementation and locally selects the globally optimal solution.

Part Two: Oak City Public Transportation

a) I would adapt Dijkstra's shortest path algorithm to solve this problem. The reason for the adaptation is because the edges we have for the Oak City Graph appear to be dynamically generated based on where we last were. If we are at a given station X our time to get to station Y depends on not only the time to travel but the additional time needed for the train to actually arrive at our spot. From here getting from station Y to Z also requires the same calculation but we could get to it from more than one train leaving at Y. In vanilla DSP all the edges are already defined and static. In the Oak city implementation we have **dynamically generated edges** that will be used to calculate the shortest path.

I would define a function nextArriveTime(station,destinationStation) that takes in our start station and our end station and returns the time needed for the next train to arrive. We can derive this information from the adjacency lists that return freq(e) and first(e). Once we have it we can add it to the length(e) to get the **cumulative** time to

travel. This cumulative value can then be used with a standard implementation of dijkstra's algorithm.



Standard Dijkstra's shortest path but our edges are generated based on length + nextArriveTime().

b) The nextArriveTime() function is an O(1) calculation function. Thus we have the same runtime as Dijkstra's shortest path which is O(e + v*log(v)).

c) ShortestTime has to first iterate through each vertex in the graph once to setup other data. Then it has to iterate again over every vertex and inside of this loop it has to iterate over all vertices. It is first processing adjacent vertices and moving on once those are complete. It is storing the initial time to travel to a given vertex as infinite and finding a better one. Thus this is Dijkstras algorithm.

d) Almost all of the existing code is reused. The only difference is that the edge cost is calculated on the fly by nextArriveTime() since it depends on not only the length but the time needed to wait for the next train.

e) After some research one of the fastest implementations of dijkstra comes from this paper entitled *Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks* (http://algo2.iti.kit.edu/download/contract.pdf). The way that it achieves high

speed is by “contracting” nodes, that is replacing edges with shortcuts and also being bidirectional so the number of nodes to search through is dramatically lower, and is stated to be 5 times faster than the next best implementation, though this is by a constant factor only.