Deveshwar Singh

CS 323

Fall 2019 Assignment Write up

2. Holiday Special - Putting Shifts Together

(a) Describe the optimal substructure of this problem.

The optimal substructure of this problem is to find the most chef with the most back to back steps and choose that chef. After choosing that chef we eliminate the steps he does, you then find the next chef with the most back to back steps that are not done by the previous chefs, adding each chef to the optimal solution.

(b) Describe the greedy algorithm in plain words that could find an optimal way to schedule the volunteers for one recipe.

The greedy algorithm that could find an optimal way to schedule the volunteers for one recipe would be to find the chef with the most back to back steps and assign him those steps for that recipe, this eliminating those steps from the open steps. Then find the next chef with the most back to back steps and assign him those steps for the recipe, eliminating those steps from the open steps. Repeat this process until all steps are done.

(c) Code your greedy algorithm in the file "HolidaySpecial.java" under the "makeShifts" method where it says "Your code here". Read through the documentation for that method. Note that I've already set up everything necessary for the provided test cases. Do not touch the other methods except possibly adding another test case to the main method. Run the code without your implementation first and you should see this:

Done.

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

M = #of cooks
N= # of steps
The runtime complexity of our greedy algorithm is O(M*N) + O(2N). M*N because we have a nested for look that looks at each cook and their steps. Plus 2n because our while would run N times if each step theoretically did 1 step. The last for look would run N if each chef only did 1 step.

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

Suppose our algorithm from answer B produces this Solution, and the optimal solution is:
Algo: S1+S2+S3+....+SN
Opt: W1+W2+W3+....WQ

Let i be the first index where Si and Wi not be equal to each other. By design of our algorithm, S1 should have the longest number of back to back steps. Which means W1<S1. Which means we can replace W1 with S1 to use less chefs. We can use the same argument for Wi+1... WQ. By the end our modified Opt = Algo. We reach a contradiction that Opt was necessarily more optimal.

3. League of Patience

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

The algorithm would be something like prims algorithm, but the shortest edge criteria is changed to be abs(next - duration) the edge with the value closest to 0. If there are any ties, it would be chosen off which of the ties has the shortest duration. That result will be the criteria we use for choosing the "shorter edge".

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

Since our algorithm is a modified prims algorithm. We are using an adjacency matric the complexity of our proposed solution should be $O(V^2)$.

(c) See the file LeagueOfPatience.java, the method "genericShortest". Note you can run the LeagueOfPatience.java file and the method will output the solution from that method. Which algorithm is this genericShortest method implementing?

The genericShortest method seems to be implementing prims algorithm.

(d) In the file LeagueOfPatience.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. Note the helper methods available to you, including one that simulates the game's API that returns the next quest time.

We could use the existing code to help us implement our algorithm by modifying the criteria it uses for choosing the next quest. We would modify it so it would account for the waiting time also by doing abs(next - duration), and choosing the quest that is closest to zero. If there is a tie, we choose the quest with the shortest duration.

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

The complexity of genericShortest is $O(V^2)$. To make it faster we could implement a binary heap and adjacency list to achieve a complexity of $O(E(logV))$.