CS 325 Homework Assignment 4

Peter Moldenhauer

10/16/17

1. Class Scheduling: Suppose you have a set of classes to schedule among a large number of lecture halls, where any class can be placed in any lecture hall. Each class c_j haws a start time s_j and finish time f_j. We wish to schedule all classes using as few lecture halls as possible. Verbally describe an efficient greedy algorithm to determine which class should use which lecture hall at any given time. What is the running time of your algorithm?

   In this problem, we have a certain number of classes and we have to schedule them all. In the case in which the times of classes overlap, then we need to use an additional lecture hall. An efficient greedy algorithm to determine which class should use which lecture hall at any given time can be described as follows: We need to first sort the set of classes based on the start time of each class. After sorting, we need to maintain two lists of lecture halls. The first list will contain the halls that are currently being used at time t. The second list will contain the halls that are currently free at time t. Initially, we need to start with 0 halls (both lists of lecture halls will start at length 0). Then, if there are no halls in the free list, we will create a new hall in this list. Then as we progress through the algorithm, for each new start time which we encounter for each given class, schedule the class to a free lecture hall and then move this hall to the list of the "currently in use" lecture hall list. Similarly, move this in-use lecture hall to the list of the free lecture halls when the corresponding class finishes. This is a "greedy" algorithm because it is always trying to schedule the class with the earliest start time for the next available lecture hall.

   The pseudocode for this algorithm is as follows:

```
ClassScheduling(T) // input T is the set of classes to schedule
    T = sortClasses(T)
    m ← 0 // number of machines
    while T is not empty
        remove class i with the smallest start time
        if there is an available lecture hall j for i then,
            schedule i for lecture hall j
        else
            h ← h + 1 // increment the number of halls used
            schedule i for a new lecture hall h
```

   Overall, this algorithm needs to first sort the classes by start times. This sorting takes time of O(n log n). Then, with each start or finish time we can schedule the classes and move the lecture halls between the lists in constant time O(1). Therefore, the total running time of the ClassScheduling algorithm is dominated by the initial sorting that takes place and an overall running time of O(n log n).

2. Road Trip: Suppose you are going on a road trip with friends. Unfortunately, your headlights are broken, so you can only drive in the daytime. Therefore, on any given day you can drive no more than d miles. You have a map with n different hotels and the distances from your start point to each hotel x1 < x2 < … < xn. Your final destination is the last hotel. Describe an efficient greedy algorithm that determines which hotels you should stay in if you want to minimize the number of days it takes you to get to your destination. What is the running time of your algorithm?

   An efficient greedy algorithm that determines which hotels you should stay in can be described as follows: We can assume that the list of hotels will already be in sorted order (based on ascending distance) from looking at the map. The algorithm starts with a loop that looks at the list of hotels from the current location and then looks at each one until the max travel distance plus the current location is exceeded. It then backs up one hotel to select the hotel at which to stay at for the given day. This process will then be repeated until the current location equals that of the final destination hotel.

   The pseudocode for this algorithm is as follows:

```
RoadTrip(hotels[]) // input is sorted list of hotels by ascending distance
    maxDistance = d
    for(i=1 to final hotel)
        if(current location + hotel[i]  > maxDistance)
            hotel to stay at = hotel[i-1]
        hotels to stay at += hotel to stay at
        current location = hotel to stay at
        if (current location == final hotel)
            break
    return hotels to stay at
```

   The overall running time of this algorithm is O(n). This is because there is a loop whose best case looks at each hotel once if the distance possible to travel always lines up with a hotel. The worst case would be that it looks at each hotel twice if the distance traveled is always more than one hotel’s distance but never more than two. In either case, the running time is O(n).

3. Scheduling jobs with penalties: For each 1 <= i <= n job j_i is given by two numbers d_i and p_i, where d_i is the deadline and p_i is the penalty. The length of each job is equal to 1 minute and once the job starts it cannot be stopped until completed. We want to schedule all jobs, but only one job can run at any given time. If job i does not complete on or before its deadline, we will pay its penalty p_i. Design a greedy algorithm to find a schedule such that all jobs are completed and the sum of all penalties is minimized. What is the running time of your algorithm?

A greedy algorithm to schedule all of the jobs could be as follows: We need to first sort all of the jobs in decreasing order of penalties. The job with the highest penalty would be first then the second highest penalty job and so forth. Next, we need to take the first job in the sorted list (the job with the highest penalty) and add it to the final sequence (schedule) of the jobs that the algorithm returns. Note: we schedule this first job at the latest time that still meets its deadline to avoid the penalty. Nothing is achieved for scheduling this job (or any job) earlier than it has to be. In fact, scheduling a job earlier than it needs to be could prevent other jobs from being completed that have earlier deadlines. For the remaining (n – 1) jobs in the sorted list we then use this logic to schedule the jobs: If the current job can fit in the final sequence (schedule) without missing the deadline then we add it to the schedule. If it cannot be added without missing the deadline then we ignore this current job for now (we will add it at the end). The key in this algorithm is to schedule all of the jobs as late as possible while still meeting their deadline (starting with the highest penalty jobs first). If by the end of this process we needed to skip over any jobs that did not fit in the schedule, we will append these remaining jobs to the end of the schedule. These are the jobs that we will have to pay the penalty for.

The pseudocode for this algorithm is as follows:

```
ScheduleJobs(jobsArray[])
    Sort all the jobs in jobsArray
    create a new int array to store the final schedule of jobs
    Iterate through all given jobs
        Nested loop to find a free slot for this job (start from the last
        possible slot)
            if a free slot found, add this job to schedule[]
            if no free slot, skip this job for now
    Iterate through all of the jobs again
        If job is not already in schedule[] then append to it
    return schedule[]
```

The running time for my algorithm is $O(n^2)$ where n is equal to the number of jobs. This is because in the algorithm there is a nested loop which creates the $n * n = n^2$. Note, there is also an additional loop at the end to append the remaining jobs to the schedule. This makes it appear that the overall running time is $O(n + n^2)$. However, since the n^2 dominates the n, the running time is $O(n^2)$.

4. CLRS 16-1-2 Activity Selection Last to Start: Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm and prove that it yields an optimal solution.

This approach of always selecting the last activity to start is indeed a greedy algorithm. This is because a greedy algorithm always makes the best available choice. As the textbook puts it: “A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.” (p. 414 CLRS) In this case, the algorithm is always choosing the “best choice” activity which in other words, is always choosing the activity has the latest start time while still yet compatible with the previously selected activities. The algorithm finds this best choice activity, adds it to the list of selected activities and then does not remove this activity from the list for the duration of the algorithm. At any “decision point” in the algorithm (or point at which the algorithm has to make a selection), all of the activities that overlap with the already chosen activities are discarded and the latest starting activity among those remaining is the best choice that is picked. This approach is very similar to the approach of always selecting the first activity to finish. However, in this approach of selecting the last activity to start is starting from the “end” rather than from the “beginning”.

We can prove that this “last activity to start” approach yields an optimal solution based on the following: Lets assume that there exists an optimal solution A which does not contain the last starting activity (lets say this is b). We can let a_i be the latest starting activity in the solution of A. Then, a_i must start no later than b (because activity b starts later than activity a_i). This means that b is compatible with all of the activities in the solution of A. If we swap out a_i for b in A, then we get a set with the same maximum activity solution but containing b. We can then keep progressing through until the algorithm adds the last activity.

5. Activity Selection Last to Start Implementation: [Code submitted separately to TEACH] Verbal description of the algorithm, pseudocode and analysis of the theoretical running time below…

   My algorithm that implemented the last-to-start activity selection can be described in the following manner: The algorithm first reads in data from the input file and creates a list of lists from the data. For example, if it reads in data for 11 activities, it would form a list of length 11 with each element its own list of the activity number, start time and finish time for the given activity. Once it has a list of lists for a given set of numbers, it then runs merge sort on the list to sort the activities based on descending start time. After sorting the list, the algorithm then loops through the list and finds the latest activity to start. The loop considers each activity and adds a given activity to the final schedule if it is compatible with all previously selected activities. This logic of looping through the activities to schedule is very similar to the logic of the first-to finish algorithm in the textbook (p. 421). After looping through the list to schedule all of the activities, the schedule is then printed out to the terminal. If there are any other additional sets of numbers in the input file, the algorithm then repeats this whole process over again.

The pseudocode for this algorithm is as follows:

```
ScheduleActivities(activities[])
    activitiesSorted[] = mergeSort(activities[])
    n = activitiesSorted.length
    schedule[] = [activitiesSorted[0]]
    k = 0
    for m in range (1, n)
        if activitiesSorted[m][2] <= schedule[k][1]
            schedule.append(activitiesSorted[m])
            k += 1
    return schedule[]
```

The theoretical running time of this algorithm is Theta(n log n). The algorithm uses merge sort in addition to a for loop. This means that the algorithm is n log n + n. However, the algorithm is dominated by the merge sort, so the overall running time takes that of merge sort which is Theta(n log n).