CS 325 Spring 2019, Homework 4

April 29, 2019

0.1 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 has 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?

- This problem is similar to the Machine Scheduling Algorithm where the goal is to minimize the number of machines used except in this case we're trying to minimize the number of lecture halls used.

  Given a set C of n classes, each having:

  - A start time, s_i
  - A finish time, f_i (where $s_i < f_i$) Goal: Schedule all the classes using a minimum number of "lecture halls"

  A simple solution would be to consider tasks by their start time and use as few machines as possible with this order. Every time two classes overlap with each other, we have to add another lecture hall used.

  Here's the pseudocode for the algorithm,

```
CLASS-SCHEDULE(C)
 Input: set C of classes with start time s_i and finish time f_i
 Output: non-conflicting schedule with minimum number of lecture halls
 n <- 0 {no. of lecture halls}
 while C is not empty
     remove class i with smallest s_i
     if there's a lecture hall j for i then
         schedule i in lecture hall j
     else
         n <- n+1
         schedule i in lecture hall n
 return n
```

  The running time for the above algorithm is $\Theta(n \log n)$ due to the sorting operation.

0.2 Scheduling Jobs with Penalties

For each $1 \leq i \leq n$ 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?

- Because every job j_i has an equal length of 1 minute, the Job Sequencing problem with deadlines has the following properties:
  - **Greedy-Choice Property**: There is an optimal soution that begins with a greedy choice.
  - **Optimal Substructure**: An optimal solution to the problem contains within it optimal solutions to subproblems.

  Thus, we can use the following greedy algorithm to solve the problem:

  1. Sort all jobs in decreasing order of penalty.
  2. Initialize the result sequence as first job in sorted jobs.
  3. Do the following for the remaining $n - 1$ jobs: ... If the current job can fit in the current result sequence without missing the deadline, add current job to the result. Else ignore the current job.

```
JOB-SCHEDULING(J[1..n])
    Sort J in decreasing order of penalty
    d, _ = max(j for j in J) # maximum deadline
    let r[1..d] be a new array
    for i = 1 to n
        for j = d to 1
            if r[j] is empty
                r[j] = J[i]
                break
    return r
```

  The running time for the above algorithm is $O(n^2)$.

0.3 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.

- Here's the pseudocode for the algorithm

```
LAST-TO-START-ACTIVITY-SELECTOR(A[1..n])
    Sort A in decreasing order of start time
    let r[] be a new array
    r <- A[1]
    i = 1
    for m = 2 to n
        if finish A[m] <= start A[i]
            r <- A[m]
            i = m
    return A
```

  This algorithm is optimal because it has the following properties:

  - **Greedy-Choice Property**: There is an optimal solution that begins with a greedy choice (with activity 1, which has the latest start time).
    - Suppose $A \subseteq S$ is an optimal solution, sort the activities in decreasing order of start time. The first activity in A is k.
      - If $k = 1$, the schedule A begins with a greedy choice.
      - If $k \neq 1$, show that there is an optimal solution B to S that begins with the greedy choice, activity 1.
      - Let $B = A - \{k\} \cup \{1\}$,
      - $s_1 \geq s_k \rightarrow$ activities in B are disjoint (compatible).
      - B has the same number of activities as A.
      - Thus, B is optimal.
  - **Optimal Substructure**: An optimal solution to the problem contains within it an optimal solutions to subproblems.
    - Given that A is optimal to S, then $A' = A - \{1\}$ is optimal to $S' = \{i \in S : f_i \leq s_1\}$.
    - Suppose we could find a solution B' to S' with more activities than A', adding activity 1 to B' would yield a solution B to S with more activities than A, which contradicts the optimality of A.

0.4 Activity Selection Last-to-Start Implementation

Submit a copy of all your files including the txt files and a README file that explains how to compile and run your code in a ZIP file to TEACH. We will only test execution with an input file named act.txt.

You may use any language you choose to implement the activity selection last-to-start algorithm described in problem 3. Include a verbal description of your algorithm, pseudocode and analysis of the theoretical running time. *You do not need to collected experimental running times.*

The program should read input from a file named "act.txt". The file contains lists of activity sets with number of activities in the set in the first line followed by lines containing the activity number, start time & finish time.

- These are the steps for the implementation of the last-to-start activity selection:

  1. Sort all activities in decreasing order of start time.
  2. Initialize the result sequence as first activity in sorted activities.
  3. Do the following for the remaining $n-1$ activities: ... If the current activity finishes before the previous activity starts, add current activity to the result. Else ignore the current activity.

  Here's the pseudocode for the algorithm:

```
LAST-TO-START-ACTIVITY-SELECTOR(A[1..n])
    Sort A in decreasing order of start time
    let r[] be a new array
    r <- A[1]
    i = 1
    for m = 2 to n
        if finish A[m] <= start A[i]
            r <- A[m]
            i = m
    return A
```

  The above algorithm has a running time complexity of $\Theta(n \log n)$ due to the sorting operation.

```
In [1]: def LastToStart(activities):
            activities = sorted(activities, key = lambda x: x[1], reverse = True)
            r = []
            r.append(activities[0][0])
            i = 0
            for m in range(1, len(activities)):
                if activities[m][2] <= activities[i][1]:
                    r.insert(0, activities[m][0])
                    i = m
            return r
```