CS 361 – Lab #8 – Greedy algorithms

Introduction

As you saw in class, greedy algorithms are often used when we have an optimization problem involving a series of choices. In today's lab, we will explore one such example problem. We would like to maximize the number of jobs that can be done over a period of time.

If we are given, say, 10 jobs, each with its start time and finish time, we want to select the largest possible subset of these jobs in such a way that no two jobs in the subset overlap in time. In other words, we can only work on one job at any one time.

For simplicity, assume that time is measured by whole numbers. The smallest interval of time we will measure will be between consecutive integers. It's okay for one job to begin at the same instant another job finishes.

We're going to solve this problem using a greedy algorithm, as we did in class. Here is the pseudocode:

```
1. Sort the jobs ascending by finish time.
2. Let W be our chosen subset of jobs to-do, and initialize it to be empty.
3. for j = 1 to n
   if job j does not overlap in time with W
     add j to W
```

To test this algorithm, we need an initial set of candidate jobs. So, we'll create a collection of 10 jobs. Their start and finish times will be random. For example, the start time for each job can be a random integer value from 0 to 9 inclusive. And we can set the duration of each job to be a random integer from 2 to 6 inclusive. But these constants are just implementation parameters that have no effect on the optimality of the algorithm, and we can tweak these numbers later if necessary.

Implementation details

For today's lab, please create a new directory called lab08. Copy the Java files from my lab08 folder from the class Web site. You will be creating 2 new source files: Job.java and Driver.java. Later in the lab, you will need to sort the jobs in a list, and you should use the appropriate comparators for this purpose.

Job.java should contain the following:

- Attributes for start time, duration of job, and how many other jobs it conflicts with
- Default constructor that sets attributes

- Get methods for start time, duration, finish time, number of conflicts. A set method for the number of conflicts (because at the time of creation we won't know this value).
- toString() – give the start time, ending time and the number of conflicts.

Driver.java should do the following:

- Create a list of 10 jobs.
- Perform the greedy algorithm above to generate a "to-do" subset list of jobs to schedule.
- Output the to-do list.
- Functions to determine:
    - whether a job has a conflict with an existing to-do list
    - for each job, the number of conflicts it has with the other jobs in the entire list.

Empirical optimality

Once you have the above algorithm working, we'll try a little experiment. In step (1) of the algorithm, we could have sorted the jobs in other ways, such as:

a. sort jobs ascending by start time
b. sort jobs ascending by duration (finish – start)
c. sort jobs ascending by the fewest number of *conflicts*: In other words, for each job j, compute the number of other jobs it conflicts with.

We would like to show empirically that all 3 of these alternative ways of sorting jobs will not always produce as "optimal" a solution as sorting by finish time. Run your program a few times on random job sets to observe this take place. For example, you may observe that sorting by finish times lets us schedule 4 jobs, but sorting by start time lets us do only 2. Feel free to change the numerical parameters of the lab: the total number of jobs, the range of possible start times, and the range of possible job durations. You are more likely to see a difference in the sorts if you create a larger number of jobs. For example, you could try 100 jobs, allow the start times to range from 0 to 100, and allow the durations to range from 3 to 7.

Alternatively, rather than relying on random numbers, you can set your own specific start times and durations to generate a case where the finish time sort yields a better schedule than one of the other ways of sorting. In this case, you won't need to create as many jobs.

Run your Driver program a few times. You are looking for a test case where sorting by finish time can schedule more jobs than the other 3 alternate ways of sorting the jobs. Be sure your output clearly shows these results. Then, copy your output to a text file. Turn in your text output file along with your source code.