CS 577

Assignment 3: Greedy Algorithms

Fall 2023

Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: Riya Kore

Wisc id: rykore

Greedy Algorithms

1. In one or two sentences, describe what a greedy algorithm is. Your definition should be informal, something you could share with a non computer scientist.

A greedy algorithm is like making the best immediate choice at each step without worrying about the future consequences, aiming to reach a solution that seems optimal but might not always be the best overall. So, a greedy algorithm treats every request as the last one, so that the output is optimal.

2. There are many different problems all described as "scheduling" problems. In the following questions, pay attention to the details of the problem setup, as they will change each time!

(a) Let each job have a start time, an end time, and a value. We want to schedule as much value of non-conflicting jobs as possible. Use a counterexample to show that Earliest Finish First (the greedy algorithm we used for jobs with all equal value) does NOT work in this case.

Let's consider three jobs A, B, and C.

	start time	end time	value
A	3	11	20
B	7	14	30
C	15	21	20

By greedy algorithm used for jobs with all equal values, we schedule the job with earliest finish time. This will be A. Now, B is conflicting with A, so it cannot be scheduled. Finally we schedule C. We would schedule 2 jobs, that are A and C. This gives us the value = 20 + 20 = 40. But, if we schedule B and C, then we get value = 20 + 30 = 50. Therefore, the greedy algorithm used for scheduling jobs with equal value will not work in this case. Jobs A, B, C provide a counterexample.

(b) *Kleinberg, Jon. Algorithm Design (p. 191, q. 7)* Now let each job consist of two durations. A job i must be preprocessed for p_i time on a supercomputer, and then finished for f_i time on a standard PC. There are enough PCs available to run all jobs at the same time, but there is only one supercomputer (which can only run a single job at a time). The completion time of a schedule is defined as the earliest time when all jobs are done running on both the supercomputer and the PCs. Give a polynomial time algorithm that finds a schedule with the earliest completion time possible.

1) Create a list of jobs, including their preprocessing times (p_i), standard PC finishing times (f_i), and job numbers.
2) Sort the list of jobs in ascending order of preprocessing times (p_i). If two jobs have the same preprocessing time, sort them by their standard PC finishing time (f_i) in ascending order.
3) Initialize the completion time (CT) to 0 and create an empty set S for supercomputer jobs.
4) Iterate through the sorted list of jobs:
   (a) If a job can be assigned to the supercomputer (i.e. it does not conflict with jobs in set S in terms of processing time), assign it to the supercomputer, update the completion time to (CT + p_i) and add it to the set S.
   (b) Assign all other jobs that can run in parallel on standard PCs simultaneously, without affecting the completion time.
5) The completion time CT will represent the earliest time when all jobs are done running on both supercomputers and all the PCs.

(c) Prove the correctness and efficiency of your algorithm from part (b).

1) Correctness.

Correctness can be proven by showing that the algorithm always produces a valid schedule that minimizes the completion time.

First, we sort the list of jobs based on their preprocessing times (p_i) and, if tied, by their finishing times on regular PCs (f_i). This sorting makes sure that we process jobs that require the least time on the supercomputer first, which is an optimal strategy.

When assigning jobs to the supercomputer, we ensure that there are no conflicts in terms of preprocessing times since we have sorted the jobs accordingly. Thus, we select the jobs optimally for the supercomputer.

For jobs that can run on regular PCs, they can be executed in parallel, which is an efficient way of utilizing the available resources.

By following this strategy, we guarantee that we finish all jobs with the earliest completion time because we optimize the use of the supercomputer and simultaneously run tasks on regular PCs without causing delay.

2) Efficiency

The efficiency of the algorithm can be analyzed in terms of its time complexity.

Sorting the list of jobs initially takes $O(n \log n)$ time, where n is the number of jobs. this the most time consuming step.

The subsequent steps involve iterating through the sorted list once and performing simple calculations and comparisions (steps 4 and 5 in part(b)).

These steps are linear in time complexity, meaning they take $O(n)$ time.

Therefore, the overall time complexity of the algorithm is dominated by the sorting step, making it $O(n \log n)$, which is polynomial time. This shows that the algorithm is efficient and can handle a reasonable number of jobs in a practical time frame.

3. *Kleinberg, Jon. Algorithm Design (p. 190, q. 5)*

   (a) Consider a long straight road with houses scattered along it. We want to place cell phone towers along the road so that every house is within four miles of at least one tower. Give an efficient algorithm that achieves this goal using the minimum possible number of towers.

1) Sort the co-ordinates of all the houses in ascending order along the road.
2) Initialize an empty list to store the positions of the towers.
3) Start from the first house and keep moving along the road.
4) Place a tower at the position of the first house.
5) For each subsequent house, check if it is within four miles of the last tower's position. If it is not, place a new tower at the current house's position.
6) Repeat step 5 for all houses.
7) Return the positions of all towers as the final solution.

   (b) Prove the correctness of your algorithm.

Proof of Correctness:

To prove the correctness of the algorithm, we need to show that every house is within four miles of at least one tower, and we are using the minimum possible number of towers.

1) Every house is within four miles of at least one tower.
Since we are placing a tower at the position of each house, or at the position of the closest house, if it is not within four miles of the last tower, every house will have a tower within four miles of its position. This is guaranteed by the algorithm design.

2) Using the minimum possible number of towers:
Le's assume that there is a better solution that uses fewer towers. In this hypothetical solution, there is at least one house without a tower within four miles of its position.
Using proof by contradiction,
Now, let's consider the first house without a tower nearby in this better solution. In our algorithm, we placed a tower at the position of this house (or the closest house before it). Therefore, the distance between this house and the nearest tower is less than or equal to four miles. This contradicts our assumption that there is at least one house without a tower within four miles in the better solution. Since, the assumption lead to a contradiction, our initial assumption that there is a better solution with fewer towers must be false. Thus, our algorithm gives the minimum possible number of towers needed to cover all the houses within four miles.

Dijkstra's Algorithm.

4. *Kleinberg, Jon. Algorithm Design (p. 197, q. 18)* Your friends are planning to drive north from Madison to the town of Superior, Wisconsin over winter break. They have drawn a directed graph with nodes representing potential stops and edges representing the roads between them.

   They have also found a weather forecasting site that can accurately predict how long it will take to traverse one of the edges on their graph, given the starting time t. This is important because some of the roads on their graph are affected strongly by the seasons and by extreme weather. It's guaranteed that it never takes negative time to traverse an edge, and that you can never arrive earlier by starting later.

   (a) Design an algorithm your friends can use to plot the quickest route. You may assume that they start at time $t = 0$, and that the predictions made by the weather forecasting site are accurate.

To find the quickest route, we can use the Dijkstra's algorithm with a modification. Instead of keeping track of the shortest distance from the starting node to all other nodes, we will keep track of the shortest time it takes to reach each node from the starting node. We can represent each node as a tuple (node, time). Here, node is the name of the node and time is the shortest time it takes to reach that node from the starting node.

We start by setting the shortest time to reach Madison to 0, and adding it to a priority queue. For each node u that we dequeue from the priority queue, we relax all its outgoing edges (v, w). Here, v is the node at the other end of the edge and w is the time taken to traverse that edge.

If the shortest time to reach v is greater than the shortest time to reach u plus w, we update the shortest time to reach v to be the shortest time to reach u plus w and add (v, shortest time to reach v) to the priority queue.

We continue this process until we dequeue Superior from the priority queue.

(b) Demonstrate how your algorithm works using a small example with 6 nodes. Your demonstration should include any data structures you maintain during the execution of your algorithm and any queries you make to the weather forecasting site. For example, if your algorithm maintains a "current path" that grows from (M)adison to (S)uperior, you might show something like the following table:

Path	Total time
M	0
M,A	2
M,A,E	5
M,A,E,F	6
M,A,E	5
M,A,E,H	10
M,A,E,H,S	13

Let the graph be:

$$\begin{tikzcd}
 & B \arrow[r, "2"] & D \arrow[r, "1"] & E & \\
\text{(Madison)}\ A \arrow[ur, "2"] \arrow[dr, "1"'] & & & & \\
 & C \arrow[r, "3"] & F \arrow[r, "1"] & G \arrow[r, "2"] & H\ \text{(superior)}
\end{tikzcd}$$

We can represent each node as a tuple(node, time). Let the starting node (Madison) be A.
We start be setting the shortest time to reach Madison to 0 and adding it to the priority queue.

1) Priority queue: [(A,0)]

We dequeue (A,0) from the priority queue and relax all its outgoing edges. they are (B,2), and (C,1). we update the shortest time to reach B and C to be 2 and 1, respectively, and add them to the priority queue.

2) Priority queue: [(C,1),(B,2)]

We dequeue (C,1) from the priority queue and relax all its outgoing edges. For C, there is just one edge going out from it: (F,3). We update the shortest time to reach F to be 1+3 = 4 and add that to the priority queue.

3) Priority queue: [(B,2), (F,4)]

We dequeue (B,2) from the priority queue and relax all its outgoing edges. The outgoing edge is (D,2). We update the shortest time to reach D to be 2+2 = 4 and add it to the priority queue.

4) priority queue: [(F,4), (D,4)]

We dequeue (F,4) from the priority queue and relax all its outgoing edges. The outgoing edge is (G,1). We update the shortest time to reach G to be 4+1=5 and add it to the priority queue.

5) priority queue: [(D,4), (G,5)]

we dequeue (D,4) from the priority queue and relax all its outgoing edges. the outgoing edge is (E,1). We update the shortest time to reach E to be 4+1 = 5 and add it to the priority queue.

6) priority queue: [(G,5), (E,5)]

We dequeue (G,5) from the priority queue and relax all its outgoing edges. The outgoing edge is (H,2). we update the shortest time to reach H to be 5+2 = 7 and add it to the priority queue.

7) priority queue: [(E,5), (H,7)]

We dequeue (E,5) from the priority queue. (E,5) doesn't have any outgoing edges, so we don't do anything.

8) priority queue: [(H,7)]

We dequeue (H,7) from the priority queue. which means we have found the shortest path time to superior. The shortest time to reach superior is 7.
The quickest route from Madison to Superior is A → C → F → G → H, which takes 7 time units

Coding Question

5. Implement the optimal algorithm for interval scheduling (for a definition of the problem, see the Greedy slides on Canvas) in either C, C++, C#, Java, or Python. Be efficient and implement it in $O(n \log n)$ time, where n is the number of jobs.

   The input will start with an positive integer, giving the number of instances that follow. For each instance, there will be a positive integer, giving the number of jobs. For each job, there will be a pair of positive integers i and j, where $i < j$, and i is the start time, and j is the end time.

   A sample input is the following:

```
  2 ← no. of instances
→ 1
  1 4
→ 3
  1 2
  3 4
  2 6
```

   The sample input has two instances. The first instance has one job to schedule with a start time of 1 and an end time of 4. The second instance has 3 jobs.

   For each instance, your program should output the number of intervals scheduled on a separate line. Each output line should be terminated by a newline. The correct output to the sample input would be:

```
1
2
```