Saleh Alkhalifa
10/18/2022

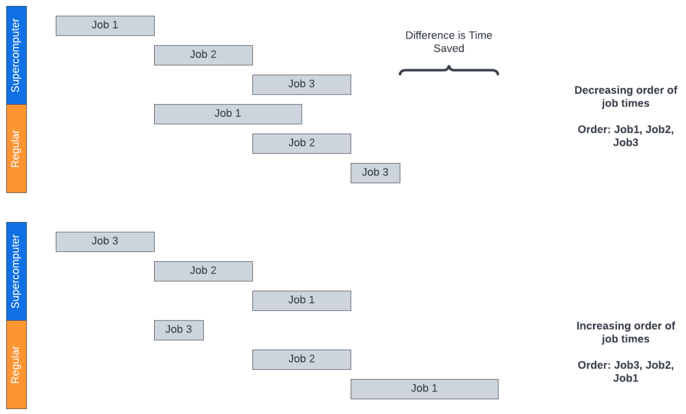
CS 5800
Module 6

Homework#6

Problem 1 Super or Normal (10 points)

- You have one supercomputer and *n* normal computers on which you need to run *n* jobs.
- Each job *i* first spends s_i time on the supercomputer and then n_i time on the normal computer.
- A job can only start running on any of the normal computers after it has finished on the supercomputer. However, as soon as any job finishes on the supercomputer, it can immediately start on one of the free normal computers. The goal is to finish running all the jobs as soon as possible.
- Note that since there is only one supercomputer, you'll always have to wait $\sum_{i=1}^{i=n} s_i$ time so that the jobs finish running on the supercomputer. However, you can optimize when you run the jobs on the normal computers to try to finish running all the jobs as soon as possible.
- Show the following schedule is optimal: execute jobs after sorting them in decreasing order by n_i. Note: Include any assumptions on this problem to clarify the justification. But do not allow your assumption(s) to contradict what is in the problem statement.

Answer:



Our objective here is to show a schedule is optimal when we execute jobs after sorting them in decreasing order by n_i in which n_i represents the time on the normal computer, which essentially marks the time in which the job has its earliest completion.

Since all jobs must spend s_i time on the supercomputer first, we are only able to optimize the algorithm when it comes to the n_i time spent on the normal computers. The first step here is to sort the jobs in order of decreasing n_i time, allowing the jobs with the longest durations to go first, and the ones with the shortest durations to go last. Depending on which sorting algorithm we use, such as MergeSort, this will establish the complexity of the algorithm which in the case of MergeSort is O(n logn). Once sorted, the idea here now is that with more jobs running in parallel allows us to complete more jobs at the same time as demonstrated in the figure above. The greedy nature of this algorithm is to use the jobs time as our metric and select the longest jobs first. We assume that the jobs once run in the supercomputer, and then move immediately to the normal computers without any downtime or additional delays or steps. We also assume that the amount of time spent on the super computer is the same for each job. Given these assumptions, and method above, we can assume that this algorithm for scheduling is optimal, which can be demonstrated:

Proof by Contrapositive.

Let us assume that there exists some solution for this scheduling problem where the longest job did not go first as suggested above. Let us also suppose that there is job with the longest duration denoted by J_{Longest} , and another job some other duration denoted by $J_{\text{NotLongest}}$. If we run our computations with our greedy implementation twice, once where the longest duration is set first, and switch them where the *NotLongest* duration is set first, we can compute the difference between the two. When J_{Longest} starts earlier, it is able to finish first on the super computer and begin running on the normal computers earlier. If $J_{\text{Non-Longest}}$ were to run earlier, it would finish much faster since we assumed it was shorted in duration, therefore we can move this task later in the queue and still have an optimal solution. Therefore, we can state that our greedy solution we will either be equal or more optimal than its counterparts.

References: [1], [2], [3], [4]

Problem 2 Skiing Agency (10 pts)

- A ski rental agency has n pairs of skis, where the height of the ith pair of skis is s_i.
- There are n skiers who wish to rent skis, where the height of the ith skier is h_i.
- Ideally, each skier should obtain a pair of skis whose height matches her/his own height as closely as possible.
- We would like to assign skis to skiers so that the sum of the absolute differences of the heights of each skier and her/his skis is minimized. Design a greedy algorithm for the problem.
- **Prove** the correctness of your algorithm. Note: Include any assumptions on this problem to clarify your proof. But do not allow your assumption(s) to contradict what is in the problem statement.

- (Hint: Start with two skis and two skiers. How would you match them? Continue to three skis and three skiers, and identify a strategy.)

Answer:

Our objective here is to match the skiers and skies as closely as possible when it comes to height. Since we have n skiers and n pairs of skis, we can assume that the number of skiers and the number of skis are the same, which we can essentially think of as two lists.

- First, we will need to sort both lists using a sorting algorithm of our choice, such as MergeSort. We can sort them in either ascending or descending order, but we will need to be consistent and sort them similarly.

- Next, we can assign the ith ski with the ith skier such that:

$$ski\_list[i] \quad is\ matched\ with\ skier\_list[i]$$

- Since the lists are equal in length, we are guaranteed that each ski will be matched to a skier.

Let us assume that we have two skiers with heights h_1 and h_2, and two sets of skis which we will denote s_1 and s_2. Let us assume that $h_1 > h_2$ and that $s_1 > s_2$. If we calculate the differences, we will see a greater value when we compute the absolute differences when h_1 is matched with s_2, or if h_2 is matched with s_1. In other words, we can minimize the sum of differences when each skier is matched with the appropriate ski. Let us assume that we have four lists, representing skis and skiers in a sorted and sorted manner. Using a function to compute the differences for each index value in each pair of lists, we can determine the absolute differences. If we go ahead and calculate the cumulative sum of differences across both sets of lists, we will see that the sorted lists yield a sum of 3.0, whereas the unsorted lists yield a sum of 8.0. In other words, we are able to minimize this sum by sorting our lists. We can prove this further as shown below:

```
ski_list = [8.5, 6.0, 6.5, 8.0, 7.5, 7.0]
skier_list = [8.0, 8.5, 9.0, 6.5, 7.0, 7.5]

sorted_ski_list = [6.0, 6.5, 7.0, 7.5, 8.0, 8.5]
sorted_skier_list = [6.5, 7.0, 7.5, 8.0, 8.5, 9.0]
```

```
def calculate_difference(list1, list2):
    sum = 0
    for i in range(0, len(list1)):
        print(i)
        print(sum)
        sum = sum + abs(list1[i] - list2[i])
    return sum
```

Proof by contradiction.

- Let us assume we have 2 skiers where h_1 and h_2 have heights of 5.0 and 6.0, respectively.
- Let us assume we also have 2 skies where s_1 and s_2 have lengths of 5.0 and 6.0, respectively.
- If we match h_1 with s_1, and h_2 with s_2, we will yield differences of 0.0 and 0.0 (our greedy approach)
- If we match h_1 with s_2 and h_2 with s_1, we will yield differences of 1.0 and 1.0 (alternative approach)
- Since the minimum value between these two cases is 0.0, we show that our greedy approach was most optimal.
- Since our algorithm is predominately governed by the sorting of two lists, we will see a time complexity of approximately O(n log n).

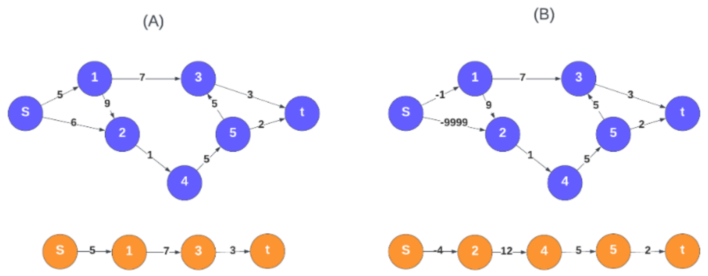
References: [3], [4], [5]

Problem 3 Negative Edges (10 pts)

Consider a directed graph in which the only negative edges are those that leave *s*; all other edges are positive. Can Dijkstra's algorithm, started at *s*, fail on such a graph? Prove your answer.

Answer:

Dijkstras algorithm is a shortest path algorithm commonly used to determine the shortest path within a directed graph consisting of non-negative weights. The algorithm operates by maintaining a list of visited nodes and their associated weights. If we look at the diagram below, we can see case A where Dijkstras algorithm would function by finding the shorted path S -> 1 -> 3 -> t. This would happen by computing the path with the lowest possible weights, which in this case would be 5+7+3 yielding a total of 15. On the other hand, if we included negative weights in the edges leaving s, it is possible that one of the weights would have a largely negative value such that a path that is not the shortest path would be the end result of the algorithm. In case B shown below, we would get a result of a path such as S -> 2 -> 4 -> 5 -> t, which is clearly not the shortest path.



Alternatively, Dijkstras algorithm can fail on graphs when a negative cycle is encountered. Essentially, this occurs when a cycle in a given graph when the sum of the weights is a negative value. Even in the event that it does not cause a negative cycle I would imagine that the downstream calculation of the sum of weights would still struggle with the negative values.

Proof.

- Let's assume that s-> v_1 -> … -> v_k is the shortest path from a given node s to the final node t.
- If assumption is true, then the statement that $d(s, vi) \leq d(s, v_k)$ for all $i < k$, is now false.
- In this case, we can say that a cycle I negative if the sum of the edge lengths is less than 0.
- Given the negative cycle in the process of going from s to t, we can sate that the shortest path length here is negative infinity.

References: [3], [4], [6]

References:

[1] https://northeastern.instructure.com/courses/117409/pages/module-6-6-dot-2-interval-scheduling-2?module_item_id=7833143

[2] https://en.wikipedia.org/wiki/Interval_scheduling#:~:text=Interval%20scheduling%20is%20a%20class,%2C%20scheduled%20on%20some%20resource).

[3] Introduction to Algorithms, Cormen, Third Edition. (CLRS)

[4] https://web.stanford.edu/class/archive/cs/cs161/cs161.1138/handouts/120%20Guide%20to%20Greedy%20Algorithms.pdf

[5] https://www2.edc.org/makingmath/mathtools/contradiction/contradiction.asp

[6] http://web.stanford.edu/class/archive/cs/cs161/cs161.1182/Lectures/Lecture11/CS161Lecture11.pdf