Algorithms: Assignment #6

Tanvi Magdum
magdum.t@northeastern.edu

February 20, 2023

Answers

1 Problem 1 Solution :

Let us assume -

1. Each normal computer has the same processing speed.
2. The jobs are independent, and the order in which they are executed does not affect their execution time. There is no prioritizing among jobs.
3. The supercomputer can execute only one job at a time.
4. The execution time of a job is the sum of times it spends on a normal computer and on a supercomputer.

In order to prove that executing jobs in descending order of n_i is an optimal solution, we need prove that there is no other schedule which can complete all jobs in less time.

Suppose that the proposed schedule takes time T to complete all the jobs and the other schedule, where jobs are not sorted in descending order, takes time R to complete all the jobs, where R < T.
Let job i and job j be the first two jobs in other schedule. Let us consider, i starts before j on one of the normal computers, that means, $n_i < n_j$.

Suppose, i completes its execution on supercomputer then it occupies one of the normal computers and starts running. j is yet to complete its execution on supercomputer. Since i requires less execution time than j, it will finish its job on the normal computer first. The same normal computer will remain idle until job j is ready to start. This idle time will increase the overall time required to complete all jobs. So, this schedule is not optimal.

Consider another case. i finishes before j on supercomputer and it starts executing on a normal computer before j. Since i requires less execution time than j, it will finish its job on the normal computer first. Consider that i runs on a different normal computer than j, since i is already using a computer and j cannot use it simultaneously. In this case, the i's normal computer will remain idle after executing i and some another normal computer will be waiting on j until it is ready to execute. Again this will add to overall execution time of all jobs and hence, this schedule also does not prove to be optimal.

Sorting jobs in descending order of n_i is optimal because it prioritizes jobs with longest normal computer time, which are most time-consuming parts. By completing these jobs earlier, we reduce the total time it takes to complete all the jobs. By running the longer jobs first, we free up normal computers earlier, which allows to complete the shorter time taking jobs quickly.

In above example with two jobs i and j, and $n_i < n_j$, assume that job i is executed before job j, then job j will have to wait for job i to finish on the normal computer before it can start. This means that job j will have less time to run on the normal computer, resulting in a longer overall time for both jobs.
Hence, the optimal strategy is to execute the jobs in decreasing order of n_i. This ensures that the jobs with longer normal compute times are given priority and have more time to run on the normal computers, resulting in a shorter overall time.

2 Problem 2 Solution :

Let us assume -

1. There are exactly n pairs of skis and n skiers.
2. The heights of skis and skiers are positive real numbers.
3. There are no two skiers or no two skies having the same heights.

Greedy algorithm:

1. Sort the pairs of skis(s) in the increasing order of their heights.
2. Sort the skiers(h) in the increasing order of their heights.
3. Initialise two pointers j and i to point to point to the first element of the arrays.
4. While i<=n and j<=n:
   (a) If h[i] <= s[j], assign skier i to pair of skis j.
   (b) Increment i by 1.
   (c) If h[i] > s[j], increment j by 1.
5. Return the assignment of skiers to pair of skis.

Proof of correctness:

We can prove the correctness of this algorithm using induction method.

Consider two skis and two skiers, the algorithm assigns each skier a pair of skis with the closest height (minimum height difference in skier and pair of skis). This solution is optimal, since there's only one other possible assignment, which would result in a larger absolute difference for at least one skier.

Assume that algorithm is correct for n-1 pairs of skis and skiers. We will prove that algorithm is correct for n pairs of skis and skiers.

For the inductive step, we assume that the algorithm is correct for n-1 pairs of skis and skiers, and we show that it is also correct for n pairs.

Let i be the tallest skier and j the tallest pair of skis. We have sorted arrays of skis and skiers in increasing order. Thus, $s_j >= s_k$ for all k!=j where k is a some pair of skis other than tallest pair.

Suppose, the algorithm assign skier i to a pair of skis j. We can then apply induction hypothesis to remaining (n-1) pairs of skis and skiers. We know that algorithm will produce an optimal output for those pairs. To prove this, we know that the height of skier i is greater than or equal to any other skier and height of pair of skis j is greater than or equal to any other pair of skis. If we assign any other pair of skis to the tallest skier, then the absolute difference in heights has to increase. There is no way we can assign a different pair of skis to skier i without increasing the absolute difference for atleast one other skier.

Consider, that algorithm assigns skier i with any other pair of skis than j, suppose k (as we introduced k earlier). Since, $s_j >= s_k$ and height of i is also greater than the height of pair of skis k, we have -

$|s_j - h_i| <= |s_k - h_i|$

Thus, assigning skier i to pair of skis j would produce a smaller sum of absolute differences than assigning skier i to pair of skis k. The algorithm produces an optimal solution for n pairs of skis and skiers, so the proof is complete.

While proving, we assumed that the skis and skiers are sorted in ascending order of their heights. If the input is not sorted, the algorithm will need to sort the input first, which takes O(n log n) time. The remaining steps of the algorithm take O(n) time. Therefore, the overall time complexity of the algorithm is O(n log n).

3 Problem 3 Solution :

Dijkstra's algorithm is used to calculate the shortest path in a weighted graph. It works by building a shortest path tree from the starting vertex to all other vertices in the graph. This algorithm works on graphs having non-negative edge weights. It is based on the principle that shortest path to any vertex v in the graph can be found by first finding the shortest path a neighbour u of v, and then extending that path to v by adding the edge (u,v).

The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts by initializing the distance of the starting vertex to 0, and distances of all other vertices to infinity. Then it selects the unvisited vertex with the smallest distance and adds it to the visited set. It then examines all neighbours of newly visited vertex and updates their distances if a shorter path is found. The algorithm repeats this process until all vertices have been visited or until the destination vertex is reached. When the algorithm terminates, the shortest path to any vertex v can be found by tracing back from v to the starting vertex along the shortest path tree.

Now, if there are negative edges in the graph, Dijkstra's algorithm may not work correctly, as the algorithm assumes that all edge-weights are non-negative. However, if the only negative edges are those leaving the source vertex s, the algorithm can still be used to find the shortest path in the graph, as the algorithm will always visit the source vertex s first and correctly update the distances of its neighbours.

To prove this, consider the steps of the algorithm. Since, all edges leaving s are negative, the algorithm will choose a neighbour of s with a negative edge weight as a first vertex to visit. Let this vertex be u. Since, all the other edges are positive, any path from s to u through other vertices will have a greater weight than the direct edge from s to u. Therefore, the distance of u will be correctly updated to the negative edge-weight, and it will added to the visited set.

The algorithm will then update the distances of neighbours of u. Since, all the other edges are positive, the updated distances will be correct. If there is a negative cycle in the graph, it will not affect the distances of vertices other than those in the cycle, because, the cycle can only be reached from s by traversing a negative edge, which is not possible in this case.

Therefore, Dijkstra's algorithm, started at s, will correctly find the shortest paths to all other vertices in a graph with only negative edges leaving s and all other edges positive.