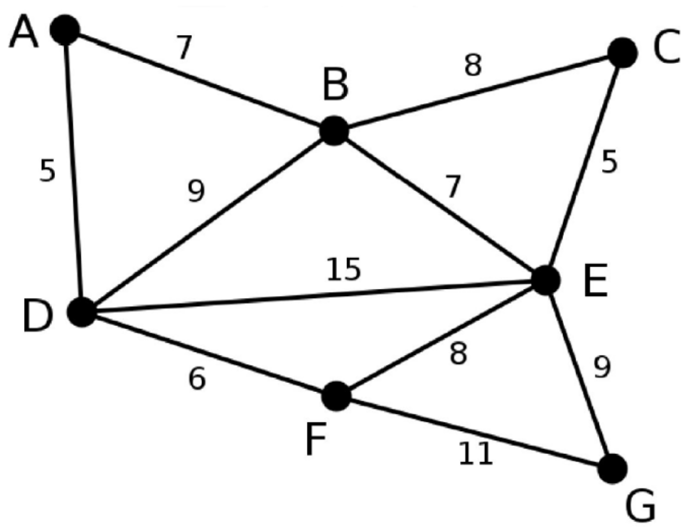
Q1 Graphs

20 Points

Consider the following weighted undirected graph with 7 vertices and 11 edges.



Q1.1

5 Points

Apply Dijkstra's Algorithm on the graph above, to determine the shortest distance from vertex G to each of the other six vertices (A, B, C, D, E, F). Clearly show all of your steps.

First, we have a graph with 7 verticles and 11 edges.
G -> (A, B, C, D, E, F)
We can start from the G and then traverse its neighbor nodes.
G -> F:
We actually have two path. One is 11 and another is G - > E + E - > F, which is 9+8 = 17, so the shortest path is 11.
G -> E: We also have two path G->E and G->F + F->E, and distance(G,E) = 9 and distance(G,F) = 11, so shortest path is 11.
G -> C:
We have a path G->E + E->C, then the shortest path is 9+5 = 14
G -> B:
Since we have already the shortest path G->E is 9 and E-> B, which shortest path is 7,
so, the shortest path for G-> B 9+7 = 16;
G -> D:

We'll have two path from G to D, distance(G,F) = 11 and distance(F,D) = 6, another path is distance(G,E) = 9 and distance(E,D) = 15, then the shortest path is 17;
G -> A:
Since distance(G,F) + distance(F,D) + distance(D,A) which is a path is 11+6+5 = 22, which can get from G to A,
another path is distance(G,E) + distance(E,B) + distance(B,A), which path is 9+7+7 = 23
so, the shortest path for G -> A is 22.

No files uploaded

Q1.2

5 Points

Now suppose we change the weight of edge EF from $+8$ to -8. What happens?

As we all know, the Dijkstra's algorithm can not be applied to the negative weight graph.

As long as the graph does not contain a negative cycle, it will have a shortest path between any two points, but Dijkstra's algorithm is not designed to find them.

If we still want to get the single-source shortest paths in a directed graph with negative edge weights is the Bellman-Ford algorithm will be applied.

HW6-Q1.2.pdf Download

2 / 2

Q1.3

10 Points

Determine a precise Loop Invariant for the Dijkstra's Algorithm, clearly stating your Initialization, Maintenance, and Termination statements. Prove that your loop invariant holds, clearly and carefully justifying each step in your proof.

HW6-Q1.3.pdf Download

1 / 3

Q2 Sunday River

10 Points

A ski rental agency has `n` pairs of skis, where the height of the the i^{th} pair of skis is s_i . There are `n` skiers who wish to rent skis, where the height of the i^{th} 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.

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

Algorithm description:

For the above problem, the optimal solution will be when both the arrays are sorted.
So, sort both the arrays and find the absolute difference of both arrays at each index, the value will be minimum, then we'll match the ith skier to the ith pair of skis.

i.e: If we want to match a taller skier with a long pair of skis, we'll reduce the minimum differences. And because the number of skiers is the same as the number of pairs of skis, we'll sort and match them in order, then we can get the answer.

Program like this to explain my idea:

```
findMinDiff(int[] s, int[] i, int n) {
    int diff = 0;
    Arrays.sort(i);
    Arrays.sort(i);
    for(int m = 0; m < n; m++){
        diff = diff + Math.abs(s[m] - i[m]);
    }
    return diff;
}
```

No files uploaded

Q3 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 that by executing jobs after sorting them in decreasing order by n_i, you have an optimal schedule.

We can have a greedy choice: to choose the first task in each sub-problem to be that with the longest normal computer time. To minimize overall time, the longest normal computer times should be done from earliest to latest, so that the last residual time of the longest unfinished normal computer task is as short a task as possible.

1. Sort the array d[] of normal computer times in decreasing order. For every swap made in d[], we keep track of the ordering by swapping the same elements in s[];
2. s[] is now sorted, and in sense that their tasks are now order of decreasing normal computer time.

I'll have a pseudocode. // the sort is well-defined

```
Input: Array s[] of supercomputer running time and array d[] of normal computer running time.
Output: Array s[] in optimal super-running order.
OptimalTimingArray(s, d)
    1. Perform Mergesort(Largest to smallest) on array d[];
           for every swap in sort, make the same indexed swap in array s[].
    2. return s[]
```

I'll have a simple proof for my idea.

We can suppose that there was an optimal solution that did not consist of the largest normal computer time first.
1. The final ordering of supercomputer task execution is Task[1..n]. Suppose the task with the largest normal computer time is Task[j], where 1<j<=n.
2. We take our desired greedy choice for the first computation, Task[j] and switch it with Task[1], noting how this changes the overall computation time:

3. Since Task[j] is now being completed earlier, clearly Task[j].normal_computer_task can start earlier and will thus finish earlier. If this was the latest finishing task in the original ordering, the problem has been further optimized. If this was not the latest finishing task in the original ordering, the problem's optimization quality has been unchanged.
4. Because supercomputer tasks are performed back-to-back and the time taken for supercomputer tasks is independent of their order, Task[1].normal_computer_task will start at the exact same time that Task[j].normal_computer_task would have started before we made the switch. This is because in each case, the normal computer task starts after the first j supercomputer tasks are complete, which is always constant.
5. Since Task[j].normal_computer _task takes longer time than Task[1].normal_computer_task, by our assumption that Task[j] has the longest normal computer time, Task[1].normal_computer_task will finish sooner than Task[j].normal_computer_task would have finished in original order. Thus, we are guaranteed that moving Task[1] back will maintain the same optimality of the original ordering.
6. so, we note that because of the switch, the normal computer times of Task[1] and Task[j] will both be finished earlier than Task[j] and Task[1], respectively, prior to switching.

Finally, we can get that since switching our greedy choice with the non-greedy choice either improves the solution or does not change the solution, our greedy choice is always equally or more optimal. So, the greedy choice must be provide an optimal solution.

No files uploaded

Problem Set 6

● GRADED

STUDENT

Kejian Tong

TOTAL POINTS

39 / 40 pts

QUESTION 1

Graphs		**20** / 20 pts
1.1	(no title)	**5** / 5 pts
1.2	(no title)	**5** / 5 pts
1.3	(no title)	**10** / 10 pts

QUESTION 2

Sunday River	**9** / 10 pts

QUESTION 3

Super or Normal	**10** / 10 pts