# Homework 6

**Due Wednesday, October 14 at 11:59 PM**

## Homework Guidelines

**Collaboration policy** Collaboration on homework problems, with the exception of programming assignments and reading quizzes, is permitted, but not encouraged. If you choose to collaborate on some problems, you are allowed to discuss each problem with at most 5 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes. Finding answers to problems on the Web or from other outside sources (these include anyone not enrolled in the class) is strictly forbidden.

*You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem.* You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.

**Solution guidelines** For problems that require you to provide an algorithm, you must give the following:

1. a precise description of the algorithm in English and, if helpful, pseudocode,
2. a proof of correctness,
3. an analysis of running time and space.

You may use algorithms from class as subroutines. You may also use any facts that we proved in class.

You should be as clear and concise as possible in your write-up of solutions.

A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand. Points might be subtracted for illegible handwriting and for solutions that are too long. Incorrect solutions will get from 0 to 30% of the grade, depending on how far they are from a working solution. Correct solutions with possibly minor flaws will get 70 to 100%, depending on the flaws and clarity of the write up.

# 1 Djikstra and negative edges

Usually, Djikstra's shortest-path algorithm is not used on graphs with negative-weight edges because it may fail and give us an incorrect answer. However, sometimes Djikstra's will give us the correct answer even if the graph has negative edges. (See Lab 4, question 1.2 for examples.)

You are given graph $G$ with at least one negative edge, and a source $s$. Write an algorithm that tests whether Djikstra's algorithm will give the correct shortest paths from $s$. If it does, return the shortest paths. If not, return 'no.' The time complexity should not be longer than that of Djiksta's algorithm itself, which is $\Theta(|E| + |V| \log |V|)$.

*(Hint: First, use Djikstra's algorithm to come up with candidate paths. Then, write an algorithm to verify whether they are in fact the shortest paths from $s$.)*

**ANSWER**

In order to achieve this requirement, what I'd do is create an algorithm that starts by running the normal Djikstra's algorithm, and after that an altered version of Djikstra's algorithm where the only difference is: instead of extracting the min value from the Heap, we'd extract the maximum value, everything else would stay the same.

By doing this, we are allowing our altered Djikstra's algorithm to begin its search via the widest value instead of the smallest, yet maintain the fact that it would still replace the value if it found smaller ones.

After these two are done running, we'd loop through the responses and compare them. If at any point the distance value between two nodes is smaller in the response for the second Djikstra's algorithm, then we'd return 'no', otherwise, once we are done looping, we'd return the shortest paths (which would be the answer from the first Djikstra's algorithm).

The runtime of this algorithm would be two times the runtime of Djikstra's plus the number of edges, but since the two times, and the addition are nullfied in runtime notation, we'd have an algorithm that runs in $\Theta(|E| + |V| \log |V|)$.

The reasoning behind why this algorithm works is actually surprisingly simply. First we use Djikstra's algorithm regardless of the situation, and, assuming it actually returns, we'd get some set of possible shortest paths s1, from here, if we run our second algorithm, it'd search the same graph, but start from the largest ones. This fixes the issue with Djikstra's algorithm not taking any other paths but the shortest into consideration. Thus giving us an alternative shortest paths s2. Now most of the paths in s2 should be bigger than in s1, with the exception of the negative ones, which s1 didn't catch since it never considered them to begin with because they were hiding behind a bigger positive path.

Based on these two sets, if we compare s1 and s2, and if at any point in s2 we find a shorter path than in s1, then we know s1 has failed, and return 'no', but otherwise (which would be the likely event for most situations where Djikstra doesn't fail), we'd just return s1.

## 2 Scheduling a software project

You are in charge of a software project that has $n$ subprojects. Each subproject $s_i$ takes $t_i$ time to complete. Some subprojects have dependencies, meaning that some subprojects cannot be started until certain other subprojects have been completed. Otherwise, any number of subprojects can be performed simultaneously.

Write an algorithm that finds the shortest possible completion time for the project, as well as a schedule with start and finish times for each $s_i$ that achieves the shortest total time.

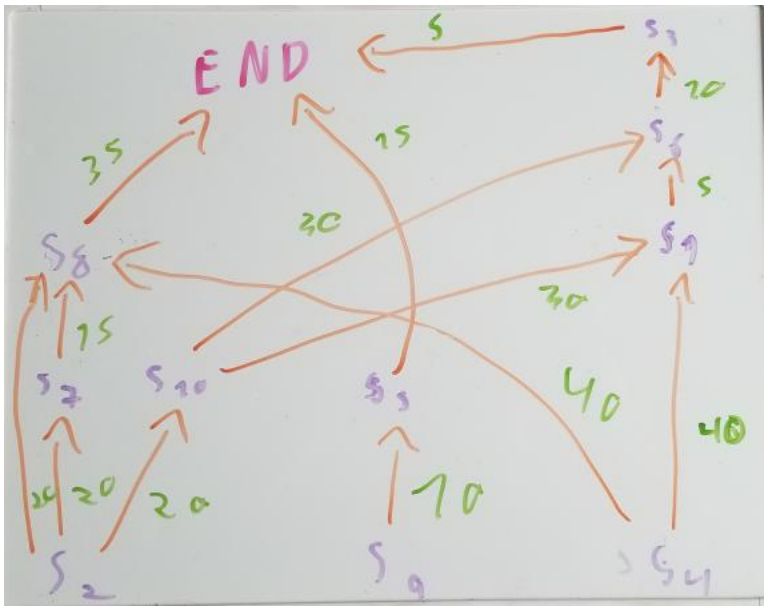
Here is an example:

| **Subproject** | $t_i$ | **Dependencies** |
|---|---|---|
| $s_1$ | 5 | $s_4, s_{10}$ |
| $s_2$ | 20 | None |
| $s_3$ | 5 | $s_6$ |
| $s_4$ | 40 | None |
| $s_5$ | 15 | $s_9$ |
| $s_6$ | 10 | $s_1, s_{10}$ |
| $s_7$ | 15 | $s_2$ |
| $s_8$ | 35 | $s_2, s_4, s_7$ |
| $s_9$ | 10 | None |
| $s_{10}$ | 30 | $s_2$ |

| **Schedule** | |
|---|---|
| **Subproject** | **Start time** |
| $s_2$ | 0 |
| $s_4$ | 0 |
| $s_9$ | 0 |
| $s_5$ | 10 |
| $s_7$ | 20 |
| $s_{10}$ | 20 |
| $s_8$ | 40 |
| $s_1$ | 50 |
| $s_6$ | 55 |
| $s_3$ | 65 |
| **Total time: 75** | |

*(Hint: How can you turn this problem into a graph so that you may use an algorithm you already know as part of the solution?)*

**ANSWER**



If we transform this problem into a graph as seen above, which starts with the nodes without dependencies, and works its way back, then we can transverse the graph in linear time, and mark each node as done, with the correct start time based on its dependencies.

In order for us to do this, we need an algorithm that gets the set of scheduled tasks, transforms it into a graph, and then begins its work. It'll need an empty queue to keep track of all the completed tasks, let this be completedQueue, in which each entry will be a task and its start time, this will be the returning datastructure.

To begin the search, the algorithm will add all the nodes without dependencies into a queue, let this be runQueue, and begin looping through every element of said queue. Each time, it'll dequeue the node that's in first position and check if all its dependencies are done. If they aren't it'll re-queue the node and continue the loop, but if they are done, then it'll add that node to the completed tasks array with the maximum value between all of its dependencies start time + run time. In addition, once a task is done, if it is a dependency of any other task (something that can be easily checked via adjacency list, it'll add all its dependencies to the runQueue.

It'll then continue looping through the other nodes until the runQueue is empty. Thus returning a completedQueue with all the correct start times. The algorithm will always produce the correct response since its looping through all the nodes and only executes them if all the dependencies are done, in addition, it'll always compute the correct start time as said time for any one node is the maximum of (startTime + runTime) for each of its dependencies.

As for runtime, the algorithm will run in about the same time as Djikstra's with $\Theta(|E| + |V| \log |V|)$, since it follows the same loop strategy.