Homework 6

Due Wednesday, October 14 at 11:59 PM

Student: Justin DiEmmanuele
Collaborators: Shilpen Patel, George Padavick, Matthew Gilgo

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.)

- **A precise description of the algorithm in English and, if helpful, pseudocode.**

  The main idea of the algorithm is to run Dijkstra's algorithm to get the claimed cheapest path and then for each edge "check" that the distance of the child node is greater than or equal to the weight of the edge added to the distance of the root node as calculated by Dijkstra's.

  For a graph G represented by an adjacency list where each value is a tuple with both the adjacent node and the weight (v, w) and source node s the algorithm is defined as follows.

Algorithm 1: DjikstraTest(G, s)

Input: G A graph in adjacency list form, with weights included
Input: s source node

```
d = Dijkstras(G, s) ;
for i = 0 to len(G) do
    for j = 0 to len(G[i]) do
        parent = i ;
        child = G[i][j][0] ;
        weight = G[i][j][1] ;
    if d[parent] + weight < d[child] then
        Output: "no"
```

Output: d

- **Proof of correctness**

  If we assume that Dijkstra's algorithm returns the shortest distance to each node, then for each edge (u_i, v_i) with weight w_i,

  $$d[u_i] + w_i \geq d[v_i].$$

  Where $d[]$ is the distance returned for that node by Dijkstra's. This must be true because $d[u_i] + w_i = d[v_i]$ is the equation that finds the length of the path to v_i that passes through the shortest path to u_i. If the distance to v_i through u_i is smaller than the returned path length $d[v_i]$ there is clear inconsistency in Dijkstra's result as Dijkstra's should have returned this path.

  Our claim is that the returned Dijkstra's distances d are equivalent to the true shortest distances to each node from the source d' if we check each node in the graph using the algorithm above and the condition is satisfied.

For our base case, level zero of the Dijkstra's tree, we check that the distance from the source (zero) plus the edge weight to each adjacent node is greater than or equal to the claimed shortest path. If this is satisfied we know that there is no inconsistency in level zero.

We assume this holds up to level N. For level $N+1$ we run the same test. If the test is satisfied for each node in level $N+1$ we know Dijkstra's paths are correct.

- **Time Complexity**

  The time complexity of the algorithm is $O(n + m \log n)$ where n is the number of nodes and m is the number of edges. This is the run time for Dijkstra's which the algorithm utilizes. The next significant portion is iterating through each edge to check the path lengths but this operation is only $O(n + m)$ since it can only iterate through all the nodes and edges. All other steps in the algorithm are $O(1)$ time.

- **Space Complexity**

  The space complexity is that of the space required to maintain the adjacency list representation of the graph - $O(m + n)$.

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?)

- **A precise description of the algorithm in English and, if helpful, pseudocode.**

  First, the provided node, dependency and weight values must be converted into a graph format. We will use an adjacency list. The pseudocode to do so is provided below.

Algorithm 2: Schedule(G)

Input: $JobTime$ array with time for each job i
Input: $Dependencies$ array of arrays containing the dependencies for each subproject i

```
AdjList Initialize empty adjacency list ;
for i = 0 to len(JobTime) do
    for j = 0 to len(Dependencies[i] do
        AdjList[Dependencies[i][j]].append((i, JobTime[i]))
```

Output: $AdjList$

The following algorithm takes an adjacency list representing the sub projects and their dependencies. The edges coming out of subproject i will have weight t_i. To start, we must do a topological sort of the jobs. Then, we create an reverse adjacency list so we can look up a nodes parents (dependencies) instead of it's children.

Finally, for each node in the order of the topological sort, we can calculate start time s_i by taking the maximum end time of each of its parents. The end time e_i of a node is $s_i + w_i$. The logic for this is shown below.

Algorithm 3: Schedule(G, t)

Input: G A graph in adjacency list form, with weights included
Input: t array of subproject times for project i

```
sorted = TopologicalSort(G) ;
G' = Reverse(G) ;
start = empty list default zero ;
end = empty list ;
for i = 0 to len(sorted) do
    maxTime = 0 ;
    startTime = start[i] ;
    for j = 0 to len(G'[i]) do
        weight = G'[i][j][1] ;
        newMax = weight + startTime ;
        if newMax > maxTime then
            maxTime = newMax
    start[i] = maxTime ;
    for i = 0 to len(t) do
        end[i] = start[i] + t[i] ;
totalTime = max(end);
```

Output: $start, end, totalTime$

- **Proof of Correctness**

  To prove correctness we need to prove the two following claims: The longest path to each

node gives the earliest possible start time and iterating over the topological sort allows us to calculate these longest paths.

A job can only be run by definition if all of its prerequisites have been met. We argue that finding the longest path of each node gives a time equivalent to the earliest time the job can be started.

- **Base Case**

  For the base case, we take the nodes with no dependencies. The shortest path to node v_i with no dependencies is obviously zero. This value is the expected minimum start time.

- **Induction Step**

  We assume that the longest path for each node n steps from the root nodes represent the earliest time a job can be started. For node $n+1$ we know the earliest start time for each of its parents. The longest time from a parent node to this node is the soonest time the job can be started since all of its dependencies must be completed before start. The claim therefore holds for all n.

Next we prove that we can find this longest path on a topologically sorted set.

- **Base Case**

  The base case is the first node in the topologically sorted set. Since topological sort grantees there will be no dependencies to the left of any node in the array, we know the first node can take the default start time of zero. The algorithm returns the correct value.

- **Induction Step**

  We assume the algorithm works for nodes $0 \ldots, n$. For case $n+1$ we must look at all dependencies earliest start time and add the respective edge weights to find the longest path to node $n+1$‘. This will only fail if we do not have a longest path for a parent of $n+1$. Since the nodes are topologically sorted and being processed in that order, it is impossible that the algorithm will not have earliest starts for parent nodes at level $n+1$. Therefore the algorithm will give the desired result.

Once we know we can find the start times for each node, finding the end time is as simple as adding the provided job runtime. The total time to finish all the jobs is just the maximum end time of the set of jobs.

- **Time and Space Complexity**

  - Time

    The time complexity of this algorithm is $O(m+n)$. The two main portions of the algorithm that are time intensive are the topological sort, the reverse of the graph, and the calculation of the earliest start time iterating over the topologically sorted array. All three of these portions require iterating over all the nodes and edges of the graph giving a time complexity of $O(m+n)$. The final step of calculating the end time and total start time can be done in $O(n)$ time. All other sections of the algorithm are $O(1)$ time.

- Space

  The space complexity of the algorithm is $O(m+n)$ as this is the space required to keep an adjacency list of the graph.