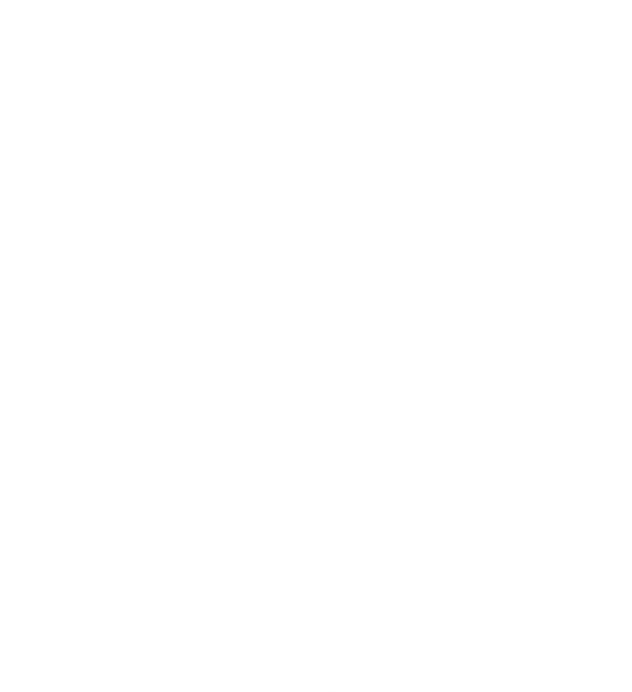
**Problem 1: Collaborators**

**Problem 2: Dijkstra Practice**



|  |  |
| --- | --- |
| **Path** | **Weight** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

1. , , , , , ,

**Problem 3: Vanilla BFS**

Solution

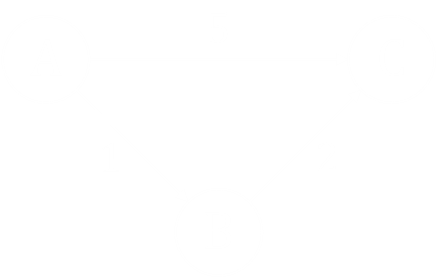
The BFS algorithm gives us one of the shortest paths from the source to any node in the graph. However, BFS does this with unweighted graphs. The definition of ‘shortest path’ in an unweighted graph is the one with the least number of edges between the source and the destination. In a weighted graph, the definition of ‘shortest path’ is the one where the total weight of the edges between the source and destination is the least. BFS does not take the weight of an edge into account, and would thus be unable to find the path with the least weight without some modification.

It is possible to get around this problem by modifying the graph itself. If we were to edit the graph so that the weight of an edge corresponds to the number of edges between the two nodes on either end of the original edge, adding nodes in between, then the unmodified BFS algorithm would work. In this case, the weight of each edge of the modified graph would be .

Note that having edges with negative weights would make the process more complicated, since we would have to consider that the edge with the least weight actually has the value and all other weights would have to be adjusted accordingly. Having a large difference between the minimum and maximum weight in the graph would also cause problems, since a huge number of edges would have to be added. This would negatively affect the time complexity.

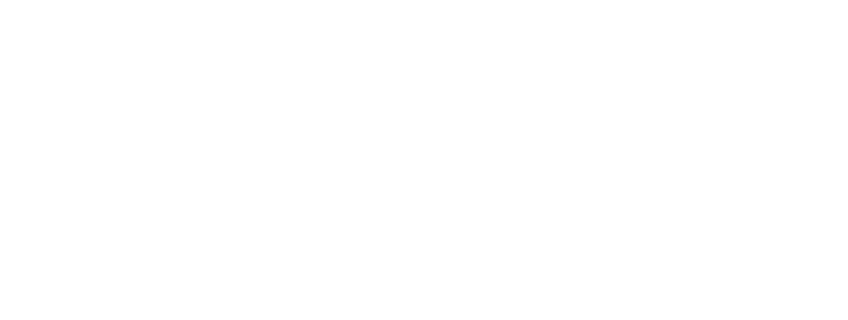
Example

Consider the graph below.



For this graph, the BFS algorithm would fail to recognize that the shortest path between and is , which traverses edges and has a total weight of . It would believe that the shortest path is , which traverses edge, but has a weight of .

Now consider the modified version of the graph below.



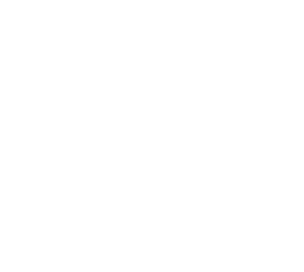
Extra nodes (identifiable by their smaller size) have been added between the original vertices so that the number of edges between the original vertices corresponds to the weights of the original edges. In this case, the unmodified BFS algorithm will correctly identify that the shortest path is (or rather, ), since it has just edges, compared to the edges on the path ().

References

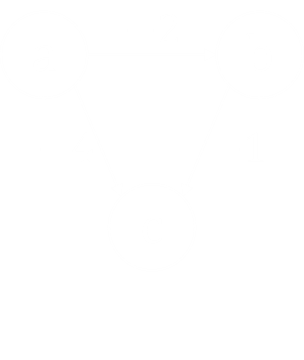
<https://stackoverflow.com/a/30409744>

**Problem 4: Longest Paths**

Consider the graph below:



If we negate all the edge weights of the graph, we will get this diagram:

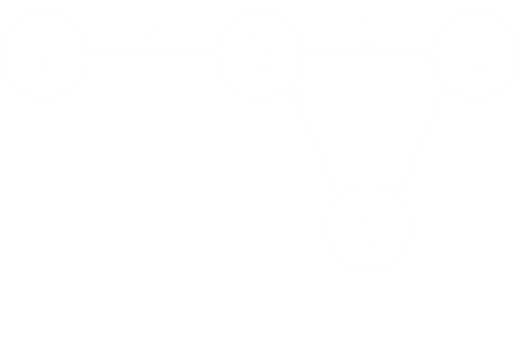


Let’s consider to be the source node and begin our analysis. From , and . From , . This is more than , so the previous value of stays. This proves that negating the edge weights will find the longest path from the source to each vertex, since for the original edge weights, we would have used the smaller positive value.

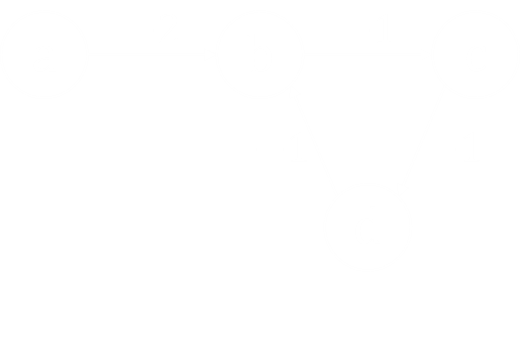
The traversal of the graph is complete, and we can negate all of the distances we have found. Thus, and . From this, we can check for the maximum value and know that the longest path in the graph is .

Starney’s algorithm makes use of the Bellman-Ford algorithm, and like the Bellman-Ford algorithm, it will not work for graphs that have a negative weight cycle. More specifically, if a graph has a cycle whose edge weights sum up to a positive value, then negating the edge weights will cause them to sum up to a negative value. This means that the longer we keep looping over that cycle, the less the distance of each of the vertices in the cycle will become.

Consider this graph:



If we negate the edge weights, we get this diagram:



Let’s go over what happens if we run Bellman-Ford’s algorithm on this graph, starting at . , , . At , we find that following the edge will give us , which is smaller than the existing value of . Thus, we will change the value of and then try to traverse all the edges of . Again, we will find that the edge can be , which is smaller than the existing value of . So, we will change the value of and traverse the edges of again. We have fallen into an infinite loop, and we cannot give a definitive value for the longest simple path in this graph.

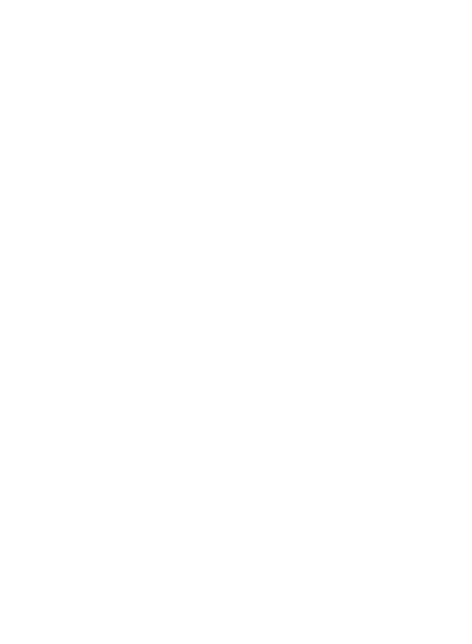
This shows how Starney’s algorithm does not work with positive weight cycles. However, Bellman-Ford’s algorithm is only set to run times. By that time, it is guaranteed that the minimum values for all paths are detected. If there is one more loop and the values decrease again, that indicates that a loop like the one above is present. As such, we can detect the existence of a positive weight cycle, even if we cannot do anything about it.

**Problem 5: Weight! That’s Possible?**

Algorithm

It is possible to solve this problem using a simple depth first search, along with a process to keep track of the minimum time needed at each stage. Before going into how to do that, first let’s see how the scenario will actually work.

Consider this graph:



The starting point of the graph is a blank node. This is not actually a task, but just somewhere for us to start working from. From this point, there are outgoing edges to all nodes that actually have no dependencies. Notice that there are two tasks from the starting point, and . Since and are not dependant on each other, they can both be executed simultaneously.

has two children, and , which can again be executed simultaneously. From ’s perspective, the time taken to complete all activities in its own branch is its own time, plus the maximum time taken by its children. Thus, if and each need second to complete, but takes seconds, the shortest time in which these three activities can be completed is seconds. Notice why this makes sense. and each have to wait second, while is completed, and then they can both work together. finishes after second, but takes seconds. Thus, the total time is the second of wait time, plus the seconds of execution time for .

Going further back, from the initial node, we can again check the maximum value of all the children of that node. If we consider to have taken second, then the branch of took seconds, which is the larger value. Thus, the shortest time in which all tasks can be completed is seconds.

Essentially, we are performing a DFS traversal, and at each recursive call, we are asking a node to return the sum of its own value, and the maximum value of its descendant branches.

Algorithm Proof

At a node of the th level, we are taking the maximum value of the branches starting at the th level, and adding the value of the th node itself. Thus, at every node, we have the shortest time taken to complete that branch. This means, at the root of the graph, we will have the shortest time to complete the entire graph.

Time Complexity

The algorithm follows simple DFS, which has a time complexity of , since we are visiting all the vertices and traversing all the edges. If we keep track of the shortest time for the branch starting at each node using a simple array, we will not have to traverse the same branch more than once, which will keep the time complexity at . We can keep track of the maximum time amongst child branches at each node using an extra variable, which will prevent any effect on the time complexity of finding the maximum value using a loop. Thus, the overall time complexity is .