Report for Programming Problem 3

Team:
Student ID: 2019216747 Name: João António da Silva Melo
Student ID: 2019216809 Name: Miguel António Gabriel de Almeida Faria

1. Algorithm description

The program starts by reading the input as it's stated in the Problem and saving it in the *pipeline* vector. While storing it, the algorithm checks for invalid connections between nodes: how many initial and final nodes exist and if there are any disconnected ones. To complete the validation of the pipeline, it is verified if there are any cycles in it, using a similar algorithm to DFS (depth-first search) where there are 2 auxiliary arrays (*visited* and *onStack*) that will allow to check for cycles efficiently and correctly by verifying if a node is present on *visited* and *onStack*, meaning that there must be a cycle. After the validation, there are two possible results: the pipeline being valid or invalid.

As it is stated in the Problem, the algorithm needs to be able to process 4 different statistics:

- Statistic 0 - If the pipeline is valid, the string "VALID" is the output.
- Statistic 1 - The most suitable algorithm to achieve the required result is the Topological Sorting Algorithm; since the total running time is also required in the output, it is calculated while reading the input. It's used a priority queue in ascending order, where lower indexes come first, so we can assure that the numerically smallest operation is executed first. Then the nodes in the queue are traversed, removing them from the queue and adding their sons. The output is composed of total running time and the list of operations.
- Statistic 2 - The algorithm chosen is the Dijkstra Algorithm, with a small change: instead of getting the smallest course, it outputs the biggest one. It's used a queue and a vector *path* with its size being the number of nodes present in the graph. This algorithm will work as the original, going through the nodes on the queue and updating *path* if their adjacent nodes have a higher cost. The value on *path* corresponding to the final node is the output.
- Statistic 3 - The algorithm is based on BFS (breadth-first search), with some modifications to complete the task required. It starts at the initial node, inserting it into a queue. Then (after "popping" the first element in the queue) its sons are traversed and, if it is the first time passing on the node, it's subtracted 1 to the number of dependencies of the son. If that number reaches 0, the son is inserted in the queue; else, the father goes back into it. Every time the queue has only one element, that node creates a bottleneck on the pipeline, so the number of the node is in the output.

2. Data structures

- operation – it's a struct that contains all the information of an operation, like time needed to process the operation, number of dependencies for this operation, nodes that depend on the current one and two variables to know if the node is final and if it is the first time passing on the node (for statistic 3)
- pipeline – it's a vector where all operations of the pipeline will be stored
- connectedNodes – *set* to help verify if all operations are connected

3. Correctness

In the resolution of this Problem, our algorithm, as it has been described before, benefits of the usage of graph algorithms, improving the efficiency of the different functionalities developed. In addition to the algorithms, the approach takes advantage of some built-in libraries for storing data, like *sets*, *priority queues* and *queues.* For example, when checking if all the nodes are connected, since there´s the possibility of repetition of nodes to insert, the *set* data structure has the most efficient built-in functionality that checks for repeated nodes while inserting new ones.

Regarding the algorithms, it´s used Topological Sorting since it returns the list of nodes where each one appears before all the ones it points to. The Dijkstra Algorithm is utilized because, if implemented to give the longest path, it finds it between a given node (initial node) and all the other nodes in the graph. It is also used the DFS and BFS Algorithms since they are for searching on graphs/trees; the first starts at the initial node and goes as far down as it's possible in a given branch, backtracking until an unexplored path is found, exploring it after; the second is used to explore the nodes of the graph, starting at the initial node and moving on to visit its sons, analyzing them and, if a given property is satisfied, these nodes are then marked, repeating this process until all the nodes have been visited and marked.

4. Algorithm Analysis

In the resolution of this problem, we used the algorithms explained before, so the analysis will be done separately.

Since our algorithm for the existence of cycles is based on the DFS, it shares the same time complexity:

$$T(n) = O(|V| + |E|)$$

with V and E being the number of vertices and edges, respectively.

For statistic 1, the time complexity, based on Topological Sorting, is also:

$$T(n) = O(|V| + |E|)$$

For statistic 2, the algorithm implemented is reached from the Dijkstra Algorithm. Since the function developed uses an adjacency list and a queue, the time complexity is:

$$T(n) = O(|V|^2)$$

For statistic 3, the time complexity of the implementation, since it uses a similar approach to the BFS algorithm and uses an adjacency list, is:

$$T(n) = O(|V| + |E|)$$

When it comes to spatial complexity, it is used the vector *pipeline* of structs *operation*. This struct has a size 4 + V, with V being the number of vertices, counting with the cases where there are cycles (for example, when a vertex is connected to itself). Therefore, in the worst-case scenario, *pipeline* has a spatial complexity of:

$$S(n) = V * (4 + V) => O(V^2)$$

It is also utilized a set *connectedNodes* that will occupy V of memory.

When checking for cycles, two bool arrays are used, *visited* and *onStack*; each of them will always occupy V of memory.

5. References

[1] PowerPoints given in classes
[2] https://www.geeksforgeeks.org/set-in-cpp-stl/