Report for Programming Problem 2

Team: 2019216646_2019216792

Student ID: 2019216646 **Name:** António Pedro Correia

Student ID: 2019216792 **Name:** Mário Guilherme de Almeida Martins Sequeira Lemos

1. Algorithm description

Our approach can be divided in two steps, the first one is invalidation checking and the second one is the pipeline processing. For the first one, we check the input for possible conflicts: disconnected nodes, multiple initial or terminal nodes and cycles. To do that we begin analyzing when we are receiving the input: if we receive two operations that have no dependencies, we can instantly invalidate the input since we will either have a case of disconnected nodes or multiple initial nodes. Once fully receive the input if we did not receive any node with 0 dependencies, we can also invalidate it since we will have no initial node and, therefore, a loop. After that the loop checking function is called which also checks for multiple terminal nodes. To do that, it recursively iterates through all the nodes and marks them as checked while also checking if any of their children have no dependents (which makes them terminal nodes) and if they have more than one or the current node has already been checked, the input gets invalidated.

After concluding the input is valid, the program will start to process the pipeline depending on the statistic received:

- ⇒ Statistic 0: The program will only print "VALID" since all the invalidation process has been concluded.
- ⇒ Statistic 1: We used Topological Sort Traversal beginning on our initial node to find the path the algorithm would take while processing the pipeline. We also sort the auxiliary queue every pop to make sure the algorithm prioritizes smaller index nodes, as requested. After that we print the total sum and the values on our queue which gives us a feasible order for running the operations.
- ⇒ Statistic 2: Using the Dijkstra algorithm, the program will find the longest path by multiplying all the time values by -1, meaning the shortest path will be the longest. This will give us the minimum amount of time the pipeline takes to process if an infinite number of operations can be processed simultaneously after we again multiply the result for -1, to give us the real value.
- ⇒ Statistic 3: Once again, using Topological Sort Traversal, this time without the sort of the auxiliary queue, the algorithm will check which operations can be ran simultaneously with any other and, the ones that cannot (the ones that are alone on our queue), will be considered bottlenecks and printed on the console.

To be more space and time efficient, while processing statistic 3, instead of allocating more space for another structure to save which operations had already called other operations and which operations had already finished, we re-used our *loopCheck* matrix by saving the finished operations on its diagonal and, on the position i, j, whenever operation i calls operation j.

2. Data structures

The main data structure used in our approach was a HashMap of the type (Integer, *operation)*, the second one being a struct that we created consisting of: an Integer *time* to store the time it takes to perform said operation, a Boolean *checked* used when the program is checking for loops and two vectors *dependencies* and *dependants* which contain, respectively, the HashMap keys for the operations from which the current one depends and the ones that depend on the current one.

There also, in the final submission, two more Booleans *finished* and *marked* that ended up not being used but were forgotten.

3. Correctness

We managed to achieve a score of 200 in *Mooshak* because, instead of coming up with our own algorithms, we altered pre-existing highly effective and highly optimized algorithms so that they would behave the way we wanted. That way, our program is highly efficient in speed and in memory usage, which makes it a correct approach.

4. Algorithm Analysis

The time complexity of our algorithm will depend on the statistic used: when using statistic 2, the time complexity will be O(V^2) (with V being the number of vertices) since it is using the Dijkstra algorithm, when using any other statistic, it will be O(V + E) (with V being the number of vertices and E the number of edges) since all of them are using Topological Sort.

As for space complexity, all of the statistics will have complexity O(V^2 + E) (with V being the number of vertices and E the number of edges) because of both the HashMap and the loop checking Matrix. There are differences between the statistics in terms of space but they are not significant enough to alter the space complexity.

5. References

C++ Cheat Sheets: https://hackingcpp.com/cpp/cheat_sheets.html