Report for Programming Problem 1

Team:
Student ID: 2019216753 Name: João Carlos Borges Silva
Student ID: 2019216826 Name: Pedro Afonso Ferreira Lopes Martins

1. Algorithm description

1.1 Algorithm approach:

1.1.1 Pipeline validation:

Initial tasks verification:
While we are reading the input we check if a task has 0 dependencies, labeling it an initial task. If more than one has this characteristic, we render the pipeline invalid. However, this verification is not enough to catch all the cases, although this isn't a problem because other verifications will detect it (**Cycles verification**).

Final tasks verification:
As we are cycling through the pipeline, using DFS, to validate it, we check the number of tasks with 0 subsequently tasks and if this number is greater than 1, we render the pipeline invalid, returning false in the recursive call

Disconnected pipeline verification:
Whilst cycling through the pipeline, using DFS, we mark each task, starting from the initial task, with a number. After this process is complete, we go through each index corresponding to a task to confirm that all have been marked. If one or more are not marked, we know the pipeline is not fully connected and therefore it's invalid

Cycle verification:
In this step we go through the pipeline, using DFS, and mark each task with a number, only removing it after we go through all the subsequently tasks. If for some reason we end up on the same task again before removing its mark we know that there is a cycle, returning false in the recursive call and diming the pipeline invalid.

1.1.2 Pipeline Statistics:

Statistic 1:
We have a variable that, whilst reading the input accumulates the time required to process each task. Afterwards we do a topological sort in order to uncover the order in which the pipeline is executing the tasks. Each unique task is part of a queue which is sorted every time we go through a new task. Afterall this we output the minimum amount of time the pipeline takes to process if only one task can be processed at a time plus the given order.

Statistic 2:
Here we go through the pipeline using topological sort once again. We start at the initial node, and we go through each subsequently task and compute the total time to get there via this task, storing the highest value obtained. By doing this we assure that even though we can go through multiple tasks at the same time the biggest value to reach a forward task is always stored. In the end we output this value for the final task plus the time cost for the initial task.

Statistic 3:
One more time we use topological sort to iterate over the pipeline, but this time we only remove a task from the queue when at least one of the subsequently tasks can enter in the queue. Each task will only enter on the queue only if the number of dependencies is zero. To output the final result, we only do so when the size of our queue is one.

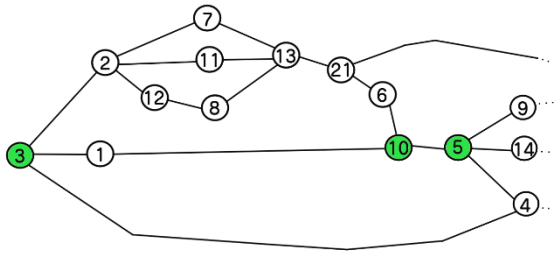

Fig. 1 – Example of a Pipeline and his bottlenecks

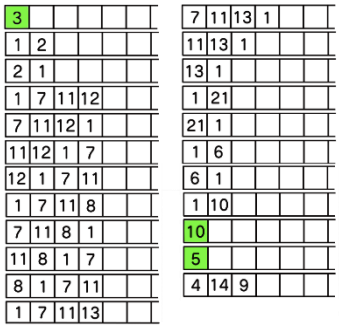

Fig.2 – Queue over the iterations

1.2 Speed-up tricks:

Simultaneous verifications: Whilst verifying the pipeline for final tasks, cycles and disconnection we perform these 3 verifications at the same time, using DFS, which in turn gives us a smaller execution time, although the verification of disconnection also has a cycle after this process if none of the others have returned false.

2. Data Structures

Struct node: This struct represents each task of the Pipeline. Each one is represented by its time to execute (**Integer**), number of dependencies (**Integer**), number of subsequently tasks (**Integer**), maximum amount of time to reach the given task (**Integer**), flag to indicate if the task can or will be deleted in the 3rd statistic (**Boolean**), flag to indicate to indicate if the task is or was in the queue, also used in the 3rd statistic (**Boolean**) and a flag which tells if the task is final or not (**Boolean**). An **array** with 2 **integers** used to mark a task in order to detect if the pipeline has disconnected nodes or cycles, a **vector of integers** to store the indexes of all subsequently tasks and finally a **vector of booleans** to depict whether all the dependencies between the task and all subsequently tasks have been decremented or not (used in the 3rd statistic).

Tasks (pipeline): Represented by an **array** which encapsulates all of the task (**struct node**).

3. Correctness

In order to achieve the minimum execution time, we use algorithms such as DFS, topological sort which are appropriate for the structure of our approach. As these algorithms are well known it's easy for us to get their time complexity further improving and facilitating the correct choice of the implemented solution.
In our Pipeline (which is a graph) DFS is used because its main purpose is to detect cycles and its disconnection, while topological sort's use fits well within the statistic 1, and with some slight variations for the remainder of the statistics.

4. Algorithm analysis

Time Complexity:

Initial tasks verification:
In order to perform this verification, we go through N tasks while reading the input, so the time complexity will be **O(N).**

Final tasks and Cycle verification:
While using DFS to iterate over the pipeline the time complexity is **O(T+L)**, where T represents the number of tasks and L the number of links between tasks.

Disconnected pipeline verification: we also use the DFS to mark the tasks of the pipeline which takes O(T+L) plus O(N) to iterate over all the N tasks to verify the connection. Total complexity will be **O(T+L)**

Statistic 1 and 2:

As we perform a topological sort to calculate the value of these statistics, the time complexity will be **O(T*k*Log(k)+L)** because we will iterate over all of the tasks and therefore their subsequently tasks (links between them) and sort them at each iteration of the while cycle. The k stands as the size of the queue at each iteration.

Statistic 3:

Following the same sequence as before we also use topological sort with a time complexity of **O((T+Tk) + (L+Lk))** to which Tk represents the number of repeated tasks and Lk the number of repeated links necessary to calculate this statistic. This will happen because the algorithm will the

Spatial Complexity:

In order to calculate the spatial complexity of our algorithm we will need to compute the space necessary to store the required information. This will give us (4 + 3 + 2 + 2*k) * N, which is the same as (9+2*k)*N required space for the all the tree. The 4 + 3 + 2 + 2*k corresponds to the size of struct node and k the number of subsequently tasks.
This will in turn give us a spatial complexity of,

$S(n) = (4 + 3 + 2 + 2 * k) * n + c$ **Spatial complexity:** O(k*N)