Report for Programming Problem 3
Team: 2019217030_2019227240
Student ID: 2019227240 Name: Sofia Botelho Vieira Alves
Student ID: 2019217030 Name: João Dionísio

1. Algorithm description

Numa fase inicial, para avaliar se a pipeline é válida ou não usamos a função "**validate**". Para tal, avaliamos as seguintes condições: (1) se esta contém apenas uma tarefa inicial e uma tarefa final; (2) se todo o pipeline se encontra conectado; (3) e se é acíclico.

Para ver se possui apenas um nó inicial, percorremos todos os nós e verificamos se o nó não possui nós pais e se possui filhos. Se este caso se verificar, incrementamos uma variável chamada **init** que contabiliza o número de nós inicial. À medida que vamos percorrendo os nós, verificamos sempre se esta variável é superior a 1. Se assim for, então a condição (1) é violada. Fazemos o mesmo procedimento para os nós finais, verificando desta vez se os nós possuem pais mas não possuem filhos. Neste passo, também verificamos a existência de nós isolados, isto é, que não possuem nem pais nem filhos, pelo que violam a condição (2).

Para verificar se o pipeline é acíclico, usamos a função **"is_cyclic**". Nesta função iremos percorrer todos os nós e considerar que cada nó possui duas variáveis, a **seen** e a **is_used**. A primeira serve para verificar se já passamos por esse nó na recursão, a segunda verifica se o nó atual já foi "processado" ou não, isto é, se já foram analisados todos os seus filhos. À medida que vamos passando pelos nós marcamos estas duas variáveis a 1. De seguida vemos os seus filhos. Se acabarmos de ver todos os filhos, desmarcamos o **is_used** para indicar que aquele nó já foi processado. Se voltarmos a passar por esse nó e verificarmos que o **is_used** ainda se encontra marcado, detetamos um ciclo, pelo que é violada a condição (3).

Validado o pipeline, passamos para a análise e recolha de estatísticas do mesmo. Para a estatística 1 é nos pedido para achar o menor tempo possível que se consegue executar todas as tarefas, correndo uma de cada vez. Para tal, usamos *topological sort* do array **order**, que contém todas as tarefas, de modo a ordenar as tarefas pela ordem que são executadas. À medida que vamos realizando o *sort*, vamos contabilizando o tempo de cada tarefa na variável **totaltime**, que guarda o tempo mínimo.

A estatística 2 funciona como a 1, desta vez sendo possível executar tarefas em simultâneo. Para esta estatística usamos DFS (Depth-first search) para percorrer todas as tarefas, calculando e guardando o tempo acumulado até estas. Quando vemos que temos duas tarefas que podem ser corridas em simultâneo, escolhemos aquela com maior tempo, cobrindo assim a outra.

Por fim, para a estatística 3 iremos identificar quais das tarefas é que são *bottleneck*, isto é, que não são paralelizáveis. Para tal, usamos outra vez *topological* *sort* para ordenar o vetor das tarefas consoante a ordem que são executadas, percorrendo de seguida as mesmas. Para cada tarefa, iremos marcar todos os seus filhos e pais com auxílio do vetor “**marked**”, de modo a identificar se existe alguma tarefa do qual esta não depende ou que não dependa da execução da mesma. De seguida, percorremos o “marked” e caso não encontremos nenhuma tarefa que não tenha sido marcada, a tarefa atual é um *bottleneck*.

2. Data structures

Neste projeto foram usadas como estruturas de dados vetores e estruturas. A seguir, apresenta-se uma breve descrição de cada estrutura utilizada.

- **Vetor <task> “task_list”:** Onde colocamos as task 's.
- **Vetor <int> “seen”:** Onde marcamos os nós ou tarefas visitados em diversos métodos.
- **Vetor <int> “marked”:** Onde marcamos os nós visitados na estatística 3, ao procurar bottlenecks.
- **Vetor <int> “bottlenecks”:** Onde colocamos os id ’s das tarefas consideradas “bottlenecks”.
- **Vetor <int> “in_use”:** Onde colocamos os nós processados durante o processo de identificação de ciclos.
- **Vetor <int> “zeros”:** Onde colocamos os nós sem pais.
- **Vetor <int> “zeros2”:** Vector auxiliar do zeros.
- **Vetor <int> “order”:** Onde colocamos os nós ordenados por ordem topológica.
- **Estrutura ”task”**: Representa uma tarefa do pipeline.

    - **Vetor “parents”**: Onde colocamos os id ’s das tarefas pais.
    - **Vetor “kids”**: Onde colocamos os id ’s das tarefas filhas.

3. Correctness

Para provar que esta solução encontra-se correta, iremos recorrer à negação por absurdo para a estatística 1.

Sub-problema: Dado um pipeline de tarefas, encontrar a sequência que permite executar todas as tarefas no menor tempo possível.

1. **Assunção:** Sendo o fator de otimização a minimização do tempo total, vamos assumir que a solução mais otimizada é a sequência S.
2. **Negação:** Sendo a sequência J uma solução mais demorada que S, vamos assumir que $|J| < |S + 1|$, sendo o módulo igual ao tempo da sequência.
3. **Consequência:** Retirando um segundo a J e $S + 1$, obtemos que $|J - 1| < |S|$.
4. **Contradição:** Contudo isto levaria à contradição do ponto 1.

Desta forma, conclui-se que $|S + 1| = |J|$.

4. Algorithm Analysis

Para o cálculo da complexidade temporal, procedemos à análise da função **main**. Esta pode ser dividida em 4 partes (cada uma das estatísticas): (1) validação da pipeline, que consiste na chamada da função "**validate**"; (2) estatística 1, que corresponde à chamada da função "**TS**"; (3) refere-se à estatística 2 e à chamada da função "**dfs**" e, por fim, (4) onde é tratada a estatística 3, usando a função "**pre_stat3**".

Começando pela análise da função "**validate**", esta inicialmente irá percorrer o vetor "task_list", que terá um tamanho igual ao número de tarefas, **N**. De seguida, iremos chamar a função "**is_cyclic**", cuja complexidade é **E**, isto é, o número de ligações entre as tarefas, já que cada tarefa visita os seus filhos, não repetindo conexões. Assim sendo concluímos que a complexidade da parte (1) é $O(N + E)$.

Na parte (2), analisando a função "**TS**" podemos inferir que esta tem complexidade $O(N)$, já que percorre todas as tarefas do grafo (pipeline) ao realizar chamadas recursivas, certificando-se que todas são percorridas pela ordem desejada.

Relativamente à parte (3), a complexidade da função "**dfs**" é $O(N + E)$. A função é chamada para cada tarefa, portanto *N* vezes, e para cada chamada percorremos os filhos da tarefa atual, o que corresponderá a um total de *E* iterações.

Por fim, para a parte (4) iremos recorrer à análise da função "**pre_stat3**". Nesta função fazemos a chamada da função "TS", de complexidade igual a $O(N)$. De seguida, percorremos todas as tarefas, portanto temos *N* iterações, e para cada tarefa percorremos os seus filhos e os seus pais através das funções "**visit_children**" e "**visit_parents**". Na soma destas funções, a complexidade será *N*, já que estas se complementam e percorremos todas as tarefas, no pior caso. De seguida, temos um ciclo onde iteramos pelo vetor **marked**, de tamanho *N*. Desta forma, consideramos que a função "pre_stat3" irá ter uma complexidade igual a $O(N + 2N^2) = O(N^2)$. Em suma, a complexidade temporal total do algoritmo será $O(N^2)$.

Passando para a complexidade espacial, analisando todas as estruturas de dados criadas, chegamos à expressão:

$$O(\overset{1.}{N(N-1)} + \overset{2.}{4N} + \overset{3.}{N-1}) = O(N^2)$$

(1) Corresponde à complexidade do vetor "**task_list**", constituído por estruturas em que cada uma possui uma complexidade de (N-1) proveniente dos vetores "**parents**" e "**kids**".
(2) Corresponde à complexidade dos vetores "**seen**", "**marked**", "**order**" e "**bottlenecks**".
(3) Refere-se à complexidade do vetor "**zeros**".

5. References

https://www.cplusplus.com/
Powerpoints da disciplina disponibilizados pelo docente.