Introdução

O problema em questão trata-se de organizar fotografias por ordem cronologica não sabendo as datas em que cada fotografia foi tirada, conseguindo apenas comparar duas e dizer qual é mais velha, havendo a possibilidade de os dados fornecidos estejam incoerentes ou insuficientes:

- Incoerente → por exemplo no caso de haverem três fotografias, A, B e C, em que é dito que A é mais recente que B, B mais recente que C e C por sua vez mais recente que A.

- Insuficiente → quando não são fornecidas informações suficientes para conseguir encontrar uma única ordenação cronologica possível, como por exemplo o caso de haverem três fotografias, outra vez A, B e C, sabendo apenas que A é mais recente que B e que C, não é possível chegar a uma solução única.

Proposta de Solução

Posto o problema, a minha proposta de solução será conseguir encontrar uma ordem cronológica por meio da implementação de um **grafo**, neste caso **direcionado** (*directed graph*) e de um algoritmo, **DFS** (*Depth First Search*), a partir do número de fotografias e o número de pares que é conhecida a ordem dado pelo stdin.

- Começa-se por recolher a informação fornecida através do stdin, a primeira linha contendo o número de fotos (Vértices) e número de pares do qual é conhecida a ordem (Ligações), criar um vetor de N nodes e utilizar a função **initializeNodes** para inicializar cada um destes, e utilizar um ciclo para guardar as L ligações, utilizando a função **addNode**.
- De seguida utilizamos o algoritmo **dfs** para organizar o número das fotografias (node→value) num vetor de inteiros, procurando **ciclos** (incoerencias) ao mesmo tempo. Caso for encontrado um ciclo, é escrito para o ecrã “Incoerencia” e termina o programa, no caso de não ser encontrado um ciclo, é acabado o vetor de inteiros.

- Após conseguirmos o vetor de inteiros, verificamos se isto é a única solução possível com a função **isSuficient** que percorre o vetor e verifica se o node correspondente a cada inteiro na posição I do vetor é adjacente ao node correspondente ao inteiro na posição I-1, caso não seja é escrito para o ecrã "Insuficiente" e termina o programa, no entanto se for o caso é imprimido para o ecrã o vetor ao contrário utilizando a função **sortPrints**.
- Finalmente, é utilizada a função **sortFrees** para libertar a memória utilizada e prevenir possíveis perdas de blocos de memória.

Análise Teórica

Ao analisar os algoritmos que poderiam dar auxilio à solução do problema, escolhi com base em dois critérios:

Facilidade de implementação: Um algoritmo simples que conseguisse resolver o problema, não havendo necessidade de perseguir a implementação de um algoritmo muito complexo.

Performance: Um algoritmo que tenha uma baixa complexidade e que resolva o problema rápidamente.

Com base nestes dois critérios, o algoritmo **DFS** encaixou-se perfeitamente, tendo uma fácil implementação e tendo uma excelente performance, com uma **complexidade de *O(V + E)***.

O algoritmo **DFS** é executado em três passos:

- **Inicialização:** todos os vertices que se encontram no array *list* são inicializados de forma a poderem ser visitados pelo algoritmo. ***O(V)***
- **Chamadas a *dfsVisit*:** como inicialmente todos os vertices têm que ser explorados, vai ser iterado o array para serem utilizados como argumento na função **dfsVisit**. ***O(V)***
- **Listas de adjacências de cada vértice:** Cada vertice pode ter zero ou mais ligações, e **vai ser analisado apenas uma vez**. ***O(E)***

Desta forma consideramos que no pior caso, o algoritmo **DFS** é caracterizado por uma complexidade assimptótica de: **O(V) + O(V) + O(E) = O(V + E)**

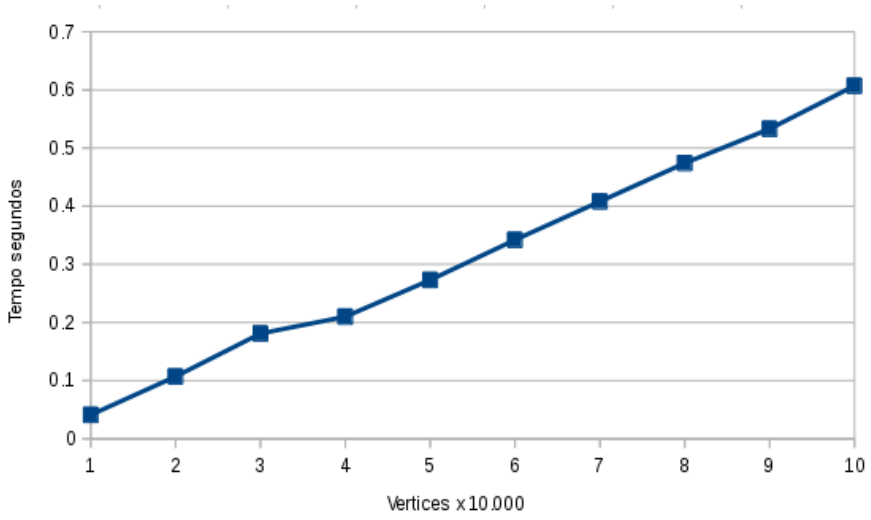
Avaliação Experimental

Como foi demonstrado e explicado anteriormente a complexidade teórica do algoritmo em questão (**DFS**) é de **O(V+E)**, ou seja **é linear em termos de Vertices e Ligações**.

Para comparar a complexidade teórica com a prática realizei dois testes (tempo e memória):

TEMPO

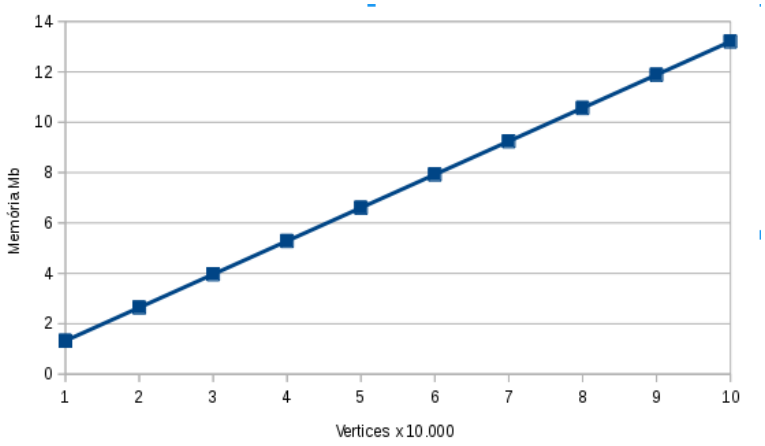
Ligações	Vertices	Tempo (s)
59 967	10 000	0.041
119 953	20 000	0.107
179 966	30 000	0.181
239 960	40 000	0.210
299 968	50 000	0.273
359 967	60 000	0.342
419 968	70 000	0.408
479 973	80 000	0.474
539 966	90 000	0.533
599 961	100 000	0.607



Como podemos ver, o número de ligações é proporcional ao número de vertices, e pelo gráfico podemos ver que se aproxima bastante de ***O(V + E)***.

MEMÓRIA

Ligações	Vertices	Mb
59 967	10 000	1.32
119 953	20 000	2.64
179 966	30 000	3.96
239 960	40 000	5.28
299 968	50 000	6.60
359 967	60 000	7.92
419 968	70 000	9.24
479 973	80 000	10.56
539 966	90 000	11.88
599 961	100 000	13.20



84742 Marcos Faria Grupo 46

Com base na tabela e no gráfico anterior, podemos também verificar um crescimento linear quando se aumenta o número de vertices e ligações.

Concluindo podemos observar que o resultado prático é consistente com a teoria, **evidenciando um crescimento linear com a alteração dos inputs.**