Trabalho 2 - Turma 031 - 2021/2

Algoritmos e Estruturas de Dados 2 - Marcelo Cohen

Algoritmo de Simulação de Treinamento de Minions

Cassiano Luis Flores Michel – 20204012-7 – Ciência da Computação

Leonardo Vargas Schilling – 20204008-5 – Ciência da Computação

19 de novembro de 2021

Introdução

Neste artigo busca-se descrever o processo de criação de um algoritmo para leitura de um arquivo de texto, onde simula-se uma pista de obstáculos para minions, onde o primeiro obstáculo descrito na linha é uma dependência para o segundo obstáculo (o primeiro obstáculo deve ser visitado e finalizado antes que o segundo possa ser feito).

Buscamos encontrar o número ideal de minions para cada arquivo de caso, bem como calcular o tempo que os minions levaram, em conjunto, para a execução do treinamento completo.

Seguiu-se as seguintes premissas:

1. Só um minion trabalha por obstáculo

2. Vários minions podem trabalhar em vários obstáculos ao mesmo tempo

3. Quando houver um obstáculo disponível para ser superado, um minion é enviado para execução do mesmo.

4. Os minions priorizam os obstáculos através de ordem alfabética.

Formato da linha do arquivo:
[obstáculo A] -> [obstáculo B]

Exemplo:
ABC_19 -> CDE_20

Significa:
Obstáculo ABC leva 19 segundos, e deve ser executado antes de obstáculo CDE, que leva 20 segundos.

Estratégia

A implementação foi feita em Java 16, em um projeto Maven, com apenas uma dependência na biblioteca Apache Commons 3 (para criação de cópia de objetos).

Entende-se que a lista de obstáculos deve ser organizada em um grafo, que foi implementado da seguinte forma:

Cada obstáculo é um vértice (Que possui um nome e um “custo”, que representa o tempo para execução do mesmo e um inteiro que representa o número de dependências).

Os vértices são armazenados em um mapa (K, V), onde o vértice é a chave (V), e uma lista com os vértices aos quais o vértice chave possui uma aresta (V).

O método MinionTrainingAlgorithm#practice recebe um inteiro que define o número de minions que trabalharão no desafio.

Já o método OptimizeTraining#findIdealMinionsPopulation realiza um laço *for*, que cria uma cópia do MinionTrainingAlgorithm já populado e executa com N minions, até que o tempo de execução com N minions seja maior ou igual a N-1.

Ao final, imprime-se as informações de número de minions e tempo dos minions para executar o percurso.

São impressos:

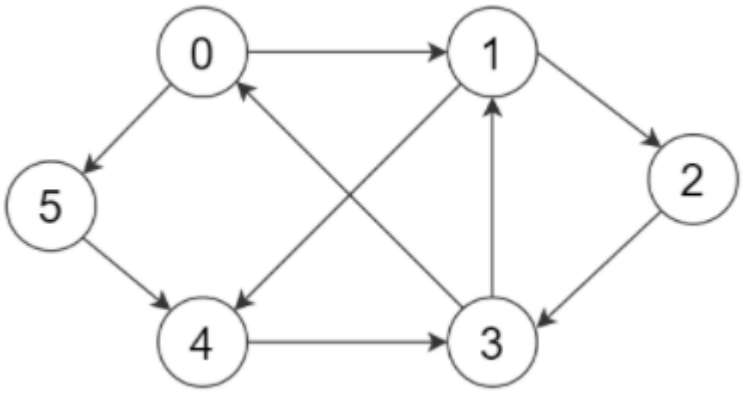
1. Nome do arquivo de caso (armazenado na pasta resources do projeto);
2. Número ideal de minions;
3. Tempo de execução dos minions (soma dos “custos” por obstáculo);
4. Tempo de execução aproximado.

Estruturas de dados

Optou-se por utilizar implementações *próprias* de um digrafo, através da interface Map<K, V>

Onde K é o vértice, e V é a lista de vértices adjacentes a K. Entende-se que as arestas são a relação entre o vértice K e a lista de vértices adjacentes V.

Demonstração de um Dígrafo



Fonte: https://www.techiedelight.com/eulerian-path-directed-graph/

Análise do algoritmo

- **Algoritmo para leitura e criação do grafo**

A complexidade de tempo para a inserção na lista auxiliar é (executada N vezes, sendo o pior caso de N o número de linhas do arquivo - 1) é O(1);

A complexidade de tempo para inserção no mapa é O(1) (executada N vezes sendo N o número de arestas do grafo, ou seja, a soma das dependências de cada arquivo da lista auxiliar);

Logo, entende-se que a complexidade para leitura e criação da classe do algoritmo (com a lista auxiliar e seu grafo) é de O(N) sendo N o número de vértices.

- **Algoritmo para treino dos minions (método #practice)**

Cria-se uma lista de obstáculos disponíveis (aqueles com 0 dependências) e uma lista dos obstáculos sendo trabalhados. Enquanto houverem obstáculos sendo superados ou então obstáculos disponíveis, o algoritmo executa o seguinte:

Inicia-se com o obstáculo mais curto (O(n) na lista auxiliar de obstáculos);

Remove-o da lista (O(1), e desconta-se o tempo dos obstáculos da lista de trabalhados que possuem uma dependência neste sendo atual (O(n));

Atualiza-se então todo o grafo, decrementando a dependência daqueles que possuiam dependência no obstáculo sendo removido (isso ocorre também em O(n) na lista auxiliar);

Então, todos os obstáculos que estiverem sem dependências no final dessa iteração são adicionados na lista de disponíveis para execução.

Por fim, em O(1):

1) Remove-se o obstáculo finalizado da lista dos obstáculos sendo trabalhados;

2) Enquanto a lista de sendo trabalhados for menor que a lista de trabalhadores, remove-se o primeiro da lista de disponíveis e adiciona-o na lista de trabalhados.

O método practice será executado enquanto o tempo for otimizao com a adição de mais um minion.

Através da análise dos dados, concluiu-se que a notação do algoritmo é $O(N^2)$.

Resultados

Após a execução do algoritmo com todos os casos de teste, os dados foram tratados no Excel, para análise do tempo de operação e número de operações.

A contagem de operações se deu através da criação de uma classe **Count**, cujo atributo *count* foi incrementado em todos os trechos e iterações do fluxo *(inserção, ordenação, practice...)*.

O tempo de execução foi calculado iniciando antes da leitura do arquivo, terminando apenas após encontrar o número de minions ideal.

O algoritmo se demonstrou ineficiente, e o grupo entende que o algoritmo poderia ser otimizado e não serve para caso na ordem de dezenas de milhares de arestas.

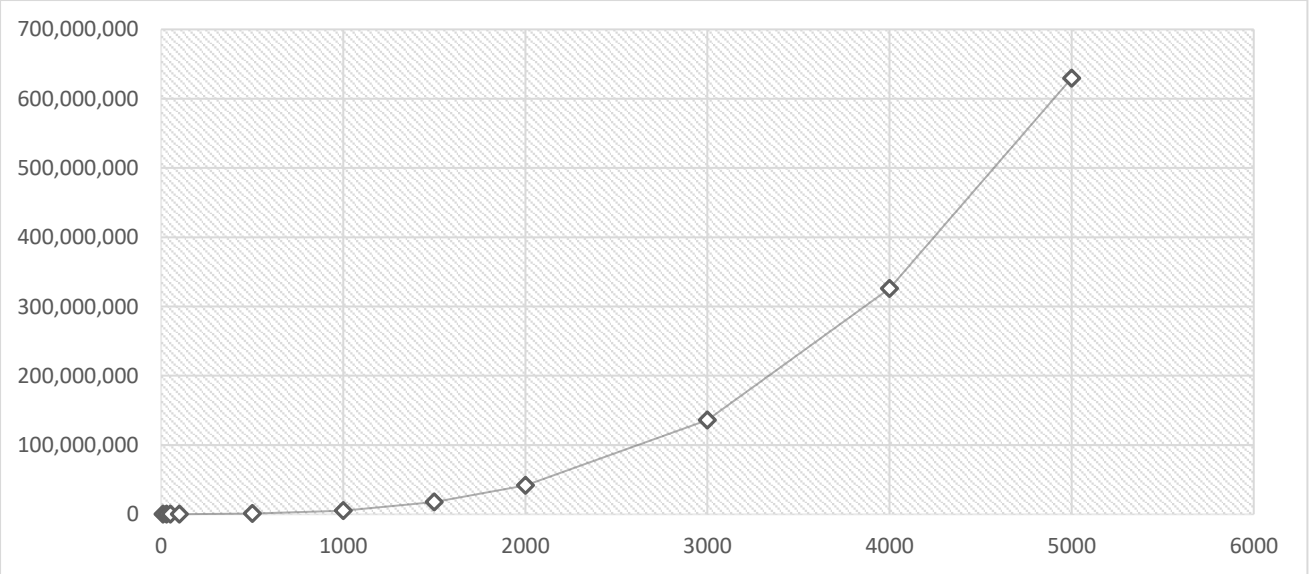
Tabela de Resultados

Nome	Arestas	Operações	Tempo Exec.	Minions	Tempo dos Minions
dez.txt	10	186	0,0100	2	996
trinta.txt	30	2.782	0,0400	9	1.437
cinquenta.txt	50	6.321	0,0300	9	2.245
cem.txt	100	40.391	0,0720	19	2.327
quinhentos.txt	500	1.285.320	0,4210	14	9.478
mil.txt	1000	5.191.895	1,1790	20	13.605
mil_e_quinhentos.txt	1500	17.619.230	2,3590	28	14.882
dois_mil.txt	2000	41.567.774	6,3820	42	14.115
tres_mil.txt	3000	136.178.009	13,9810	30	27.302
quatro_mil.txt	4000	326.033.779	34,0700	40	28.564
cinco_mil.txt	5000	629.664.977	57,5510	36	39.275

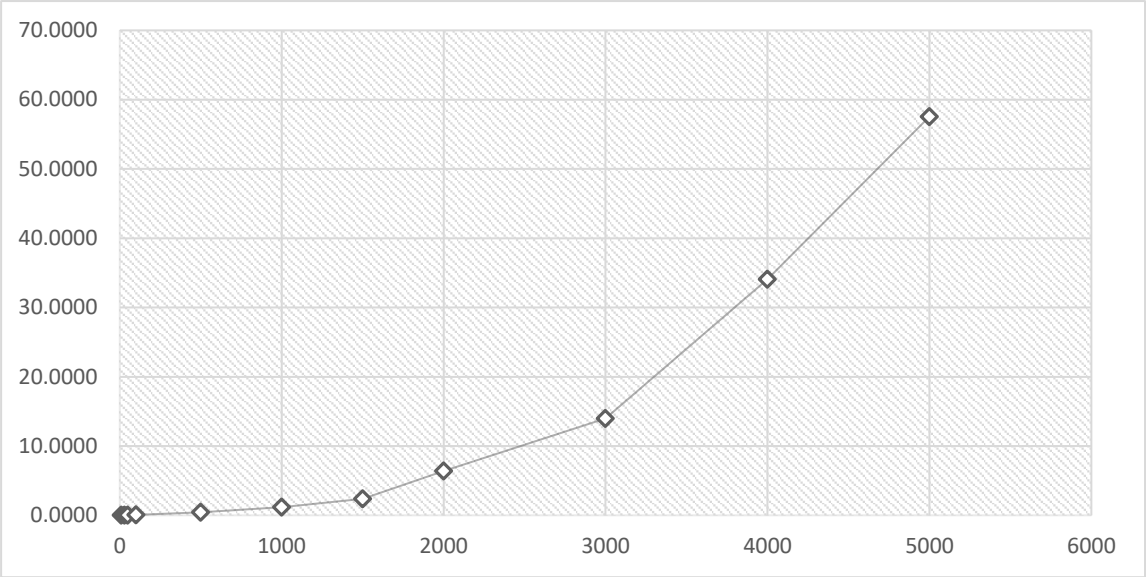
Fonte: autores

Com os dados de tempo de execução, número de operações e número de ordens (linhas do arquivo de texto), foi possível esboçar dois gráficos:

Número de Operações x Número de Arestas



Tempo de Execução (s) x Número de Arestas



Referências

[1] Sedgewick, Robert; Wayne, Kevin: "**Directed Graphs**". Last modified on December 29, 2017.
Available in: https://algs4.cs.princeton.edu/42digraph

[2] Gurgul, Adam: "**How to Make a Deep Copy of an Object in Java**". Last modified on October 25, 2021.
Available in: https://www.baeldung.com/java-deep-copy