A pista de treinamento Minion

André Luiz Rodrigues,* Bruno Ramos†

Escola Politécnica — PUCRS

19 de novembro de 2021

Resumo

Este trabalho tem como objetivo descrever as soluções encontradas para o problema proposto pelo Prof. Marcelo Cohen na disciplina de Algoritmos e Estrutura de Dados II. O problema consiste em uma pista de obstáculos que devem ser destruídos por uma equipe de minions na ordem correta, até que não reste mais nenhum obstáculo restante. O objetivo do problema consiste em descobrir a menor quantidade de minions a ser empregado para resolver o problema no menor tempo. Para a resolução do problema, trabalhou-se com a linguagem Java.

Introdução

Com o intuito de compreender as soluções retratadas neste artigo, serão descritas as informações e especificações disponibilizadas no enunciado do problema.

1. Um grupo de minions foi encaminhado ao campo de treinamento do Dr. Nefário. Lá eles passarão por atividades que envolvem trabalho duro, esforço em equipe e muitas palavras motivadoras, empreendedoras e disruptivas.
2. Cada equipe será composta por um determinado número de minions.
3. O campo de treinamento consiste em um grande conjunto de obstáculos que devem ser superados pela equipe de minions. Eles precisam desmanchar cada um dos obstáculos, na ordem correta, até que não reste mais nenhum para ser destruído.
4. A esquematização dos obstáculos presentes na pista estão dispostos em uma lista com os nomes de cada obstáculo, conforme imagem abaixo:

```
blabla_213 -> tititi_53
```

Figura 1: **Exemplo de nome e relação de dependência dos obstáculos.**

5. O nome de cada obstáculo é composto por um conjunto de letras, seguido de um número que determina quanto tempo é necessário para destruí-lo;
6. A relação significa que o obstáculo *blabla_213* (vértice pai) deve ser destruído antes que o obstáculo *tititi_53* (vértice filho) possa ser atacado.
7. Apenas um minion pode trabalhar em cada obstáculo, porém diversos minions podem trabalhar em diversos obstáculos diferentes ao mesmo tempo para destruí-los o mais depressa possível;
8. Quando um obstáculo está livre para ser desmanchado um minion ocioso (se houver) é imediatamente enviado para o trabalho;
9. Quando existem vários obstáculos que podem ser escolhidos, o minion é enviado para o que vem primeiro em ordem alfabética.

Pelo fato de apenas um minion poder trabalhar em cada obstáculo, mandar um grupo com minions demais não irá ajudar a diminuir o tempo de execução, pois a grande maioria irá ficar sem trabalhar. Do mesmo modo, não será efetivo encaminhar poucos minions, pois poderá haver obstáculos disponíveis para serem destruídos, porém todos os minions estarão empregados em alguma tarefa, fazendo com que o tempo de execução demore mais do que o necessário. Desse modo, o objetivo consiste em descobrir a menor quantidade de minions a ser empregado para completar a pista de treinamento proposta abaixo no menor tempo possível.

```
ia_48 -> sgg_7
ia_48 -> ioc_37
sgg_7 -> vhmw_89
ioc_37 -> sgg_7
ioc_37 -> phn_96
ioc_37 -> vhmw_89
phn_96 -> sgg_7
phn_96 -> vhmw_89
hax_88 -> ia_48
hax_88 -> sgg_7
hax_88 -> ioc_37
hax_88 -> phn_96
hax_88 -> vhmw_89
dmb_83 -> ia_48
dmb_83 -> sgg_7
ebyr_80 -> ia_48
ebyr_80 -> sgg_7
```

Figura 2: **Arquivo de entrada da composição da pista de treinamento.**

Ao realizar as ligações propostas, obteve-se o seguinte desenho:

Figura 3: **Grafo formado pela ligação dos obstáculos da pista de treinamento.**

Terminologia

Ao analisar o desenho obtido, percebe-se que o problema se trata de um grafo. Diante disso, para compreendermos melhor o funcionamento dos grafos, será feita as associações abaixo:

1. Cada obstáculo compreende a um vértice do grafo, ao passo que cada linha traçada corresponde a uma aresta.
2. Uma aresta e um vértice são conectados.
3. Arestas paralelas são arestas associadas ao mesmo conjunto de vértices.
4. Dois vértices que são conectados por uma aresta são chamados de adjacentes.
5. Um vértice que é nó terminal de um laço é dito ser adjacente a si próprio.
6. Duas arestas incidentes ao mesmo vértice são chamadas de adjacentes.
7. Um vértice que não possui aresta incidente é chamado de isolado.
8. Um grafo com nenhum vértice é chamado de vazio.

Na imagem abaixo, é possível verificar algumas das relações descritas acima, como por exemplo:

- O vértice V1 e o vértice V4 são adjacentes;
- As arestas E2 e E3 são paralelas; e
- O vértice V5 é um vértice isolado.

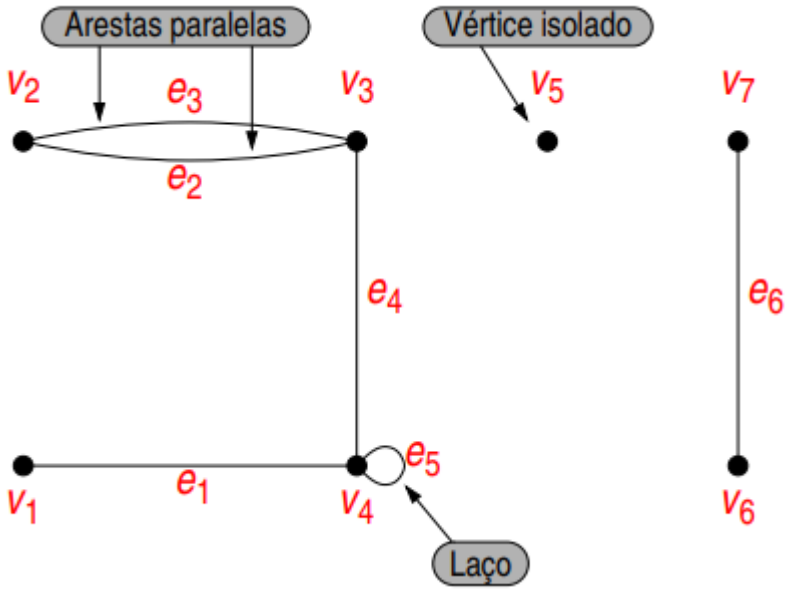


Figura 4: **Exemplo de nomenclaturas de grafos.**

Representação do Grafo

Para o desenvolvimento da solução é necessário primeiramente fazer a representação do grafo do exercício. Sabendo que um grafo nada mais é do que um conjunto de vértices e arestas, uma das possíveis soluções seria a representação através de uma lista de arestas, podendo ser implementada por uma lista encadeada ou vetor. Nessa lista, cada aresta seria representada por dois vértices conforme imagem abaixo:

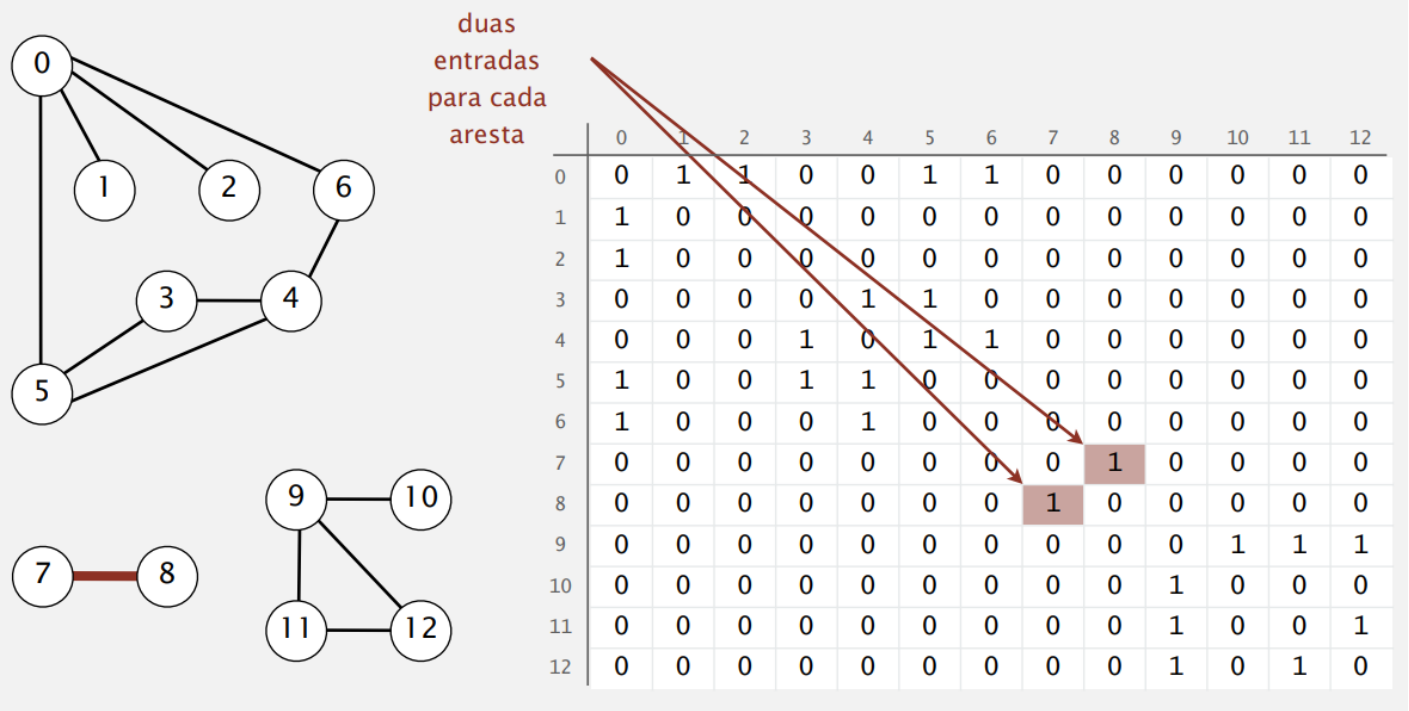
		Vértices	
		V1	V2
Arestas	A1	0	1
	A2	0	2
	A3	0	5
	A4	0	6
	A5	3	4
	A6	3	5
	A7	4	5
	A8	4	6

Figura 4: **Representação de grafo por Lista de Arestas**

O problema é que por ser uma solução de complexidade linear O(n), onde *n* é o número de arestas do grafo, essa solução pode ser muito custosa. Para acessar todas as arestas adjacentes de um

mesmo vértice, por exemplo, seria necessário percorrer todas as arestas do grafo para poder identificá-las. Até mesmo para acessar apenas uma determinada aresta do grafo, como a aresta A7 da figura, seria necessário percorrer todas as seis arestas anteriores da lista. Além disso, se fosse preciso remover um vértice do grafo, como o vértice 3, seria necessário deletar as arestas A5 e A6 e readequar a lista. Ou seja, apesar de ser uma estrutura que funciona na prática, ela acaba sendo muito lenta e ineficiente.

Outra possível solução para representar o grafo do exercício, seria através de uma estrutura de matriz de adjacência, conforme imagem abaixo:



	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

Figura 5: **Representação de grafo por Matriz de Adjacência**

Por meio da matriz, as ligações entre os vértices são representadas pelo numeral 1, enquanto a não existência de ligação é representada pelo numeral 0, ou seja, cada numeral 1 presente na matriz de adjacência representa uma aresta que liga dois vértices (coluna x linha).

Desse modo, para encontrar todos os vértices adjacentes do vértice 9, por exemplo, não precisamos percorrer toda a matriz, e sim basta percorrermos somente a linha do vértice 9 e verificar quais vértices presentes no Grafo possuem conexão (dígito 1), sendo muito mais eficiente do que a representação anterior. No caso da figura, o vértice 9 é adjacente (possui conexão) com os vértices 10, 11 e 12.

Diferentemente da representação do grafo através de uma lista de arestas, onde tínhamos uma solução de complexidade O(n), com a utilização da representação por matriz de adjacência, o grau de complexidade passa a ser O(v), onde *v* representa o número de vértices. O problema dessa solução, é que ela passa a ser muito ineficiente nos casos onde temos muitos vértices e poucas arestas, visto que independentemente do número de arestas do grafo, é necessário consultar todos os vértices para identificar quais os vértices que tem ligação (vértices adjacentes) com um determinado vértice. Devido ao fato de se tratar de uma matriz, outro problema dessa representação é o alto consumo de memória para armazenar a matriz inteira, desse modo, o gasto com memória acaba

sendo diretamente proporcional à quantidade de vértices presentes no grafo.

Para representar o nosso Grafo, é preciso buscar uma estrutura que possa armazená-lo sem um alto custo de memória, e que possa acessar qualquer ligação de um vértice de forma rápida. A melhor solução encontrada é a representação por lista de adjacência.

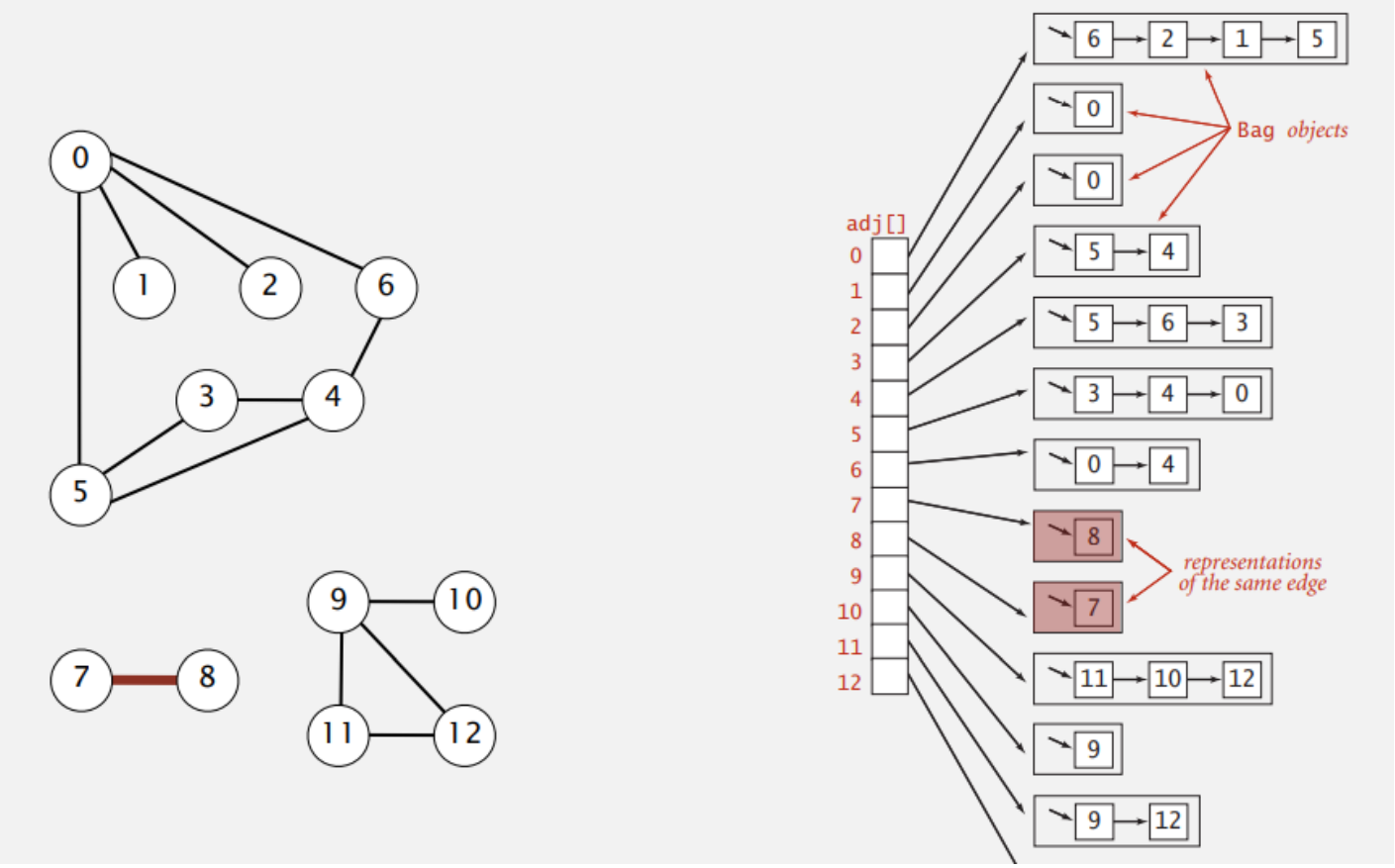


Figura 6: **Representação de grafo por Lista de Adjacência**

Essa solução é composta por uma lista principal, onde os vértices do grafo são colocados em cada índice dessa lista, de modo que seja possível acessar cada vértice diretamente pelo seu índice. Em cada posição da lista, que corresponde a um vértice, adicionamos uma segunda lista contendo todos os vértices adjacentes dele, de modo que possamos acessá-los diretamente pelo índice da lista principal. Na imagem acima, percebe-se que o vértice 9 possui como vértices adjacentes os vértices 11, 10 e 12.

1. ***procedimento*** *CriarGrafo()*
2. ***Para cada*** *vértice diferente criado*
3. ***Adiciona-o*** *em um índice do grafo*
4. ***Adiciona*** *uma lista de vértices adjacentes junto novo vértice criado*
5. ***Fim***
6. ***procedimento*** *CriarAresta()*
7. ***Conectar*** *os vértices do grafo*

8. ***Adiciona*** *o vértice V1 lista de vértices adjacentes do vértice V2*
9. ***Adiciona*** *o vértice V2 lista de vértices adjacentes do vértice V1*
10. ***Fim***
11. ***Fim***
12. ***Fim***

Uma das vantagens dessa representação é a economia de memória, visto que há consumo de memória apenas com o vetor principal que corresponde ao número de vértices, e com as listas dos vértices adjacentes de cada vértice, diferentemente da representação por matriz. O custo para acessar os vértices adjacentes com essa representação, é exatamente o grau do vértice que está sendo feita a consulta.

Na figura abaixo, é possível comparar os custos de cada implementação:

"E" corresponde a quantidade de arestas

"V" corresponde a quantidade de vértices

Representação	espaço	adicionar aresta	aresta entre v e w?	iterar sobre vértices Adjacentes a v?
lista de arestas	E	1	E	E
matriz de adjacência	V^2	1 *	1	V
listas de adjacência	$E + V$	1	*grau(v) / grausaida(v)* **	*grau(v) / grausaida(v)* **

* sem arestas paralelas
** para grafos dirigidos

Figura 6: **Tabela comparativa dos custos implementação de cada representação**

Solução

Após definir a representação do grafo por lista de adjacência, o próximo passo é o desenvolvimento da solução para o problema. A solução proposta para o problema se dá, inicialmente, em interpretar o arquivo de entrada para criar o grafo a ser desenhado. O programa irá fazer a leitura de uma linha por vez, e conforme mostrado anteriormente na introdução, cada linha possui um "vértice pai" e um "vértice filho", cada um contendo suas características próprias, como nome e tempo de execução. Cada novo vértice interpretado é adicionado junto ao grafo, e caso ele seja um "vértice filho",

ganhará um atributo que armazenará todos os “vértices pai” que estiverem ligados à ele. Vale salientar que todos os vértices possuem um atributo contendo todos os vértices adjacentes a ele, desse modo, o grafo é construído utilizando a abordagem de representação por lista de adjacência.

Após a leitura de todos os vértices do arquivo, temos a pista de treinamento (grafo) construída, e devemos implementar um método para os minions percorrerem cada obstáculo (vértice) da pista. Conforme as regras dispostas na introdução, para que um obstáculo esteja disponível para ser destruído(vértice visitado), ele não pode ser dependente de nenhum outro obstáculo, ou seja, todos os seus “vértices pai”(se houver) devem já ter sido destruídos. Desse modo, a solução encontrada foi a criação de uma lista para armazenar todos os vértices disponíveis para serem visitados dentro do grafo. Como pode haver mais de um vértice disponível, a ordem de prioridade de visitação será a ordem alfabética, conforme exigido nas regras expostas anteriormente.

```
1.  procedimento atualizaDisponiveis()
2.      Para cada elemento do ArrayList de Vertices
3.          Se existir vertice no ArrayList de dependencia
4.              Se ArrayList de dependencia for vazio
5.                  Adiciona vertice no verticesDisponiveis
6.              Fim
7.          Fim
8.      Fim
9.  Ordena em ordem alfabetica verticesDisponiveis
```

Logo de cara, percebe-se um problema nessa solução. A quantidade de vértices que podem ser visitados ao mesmo tempo depende diretamente da quantidade de minions encaminhados para a pista. Para isso, criamos uma lista no grafo que armazenará os vértices para execução, que será composta pelos vértices disponíveis para serem visitados no momento, respeitando a quantidade máxima de minions na pista. O vértice que estava que antes pertencia a lista de vértices disponível, passa a compor assim a lista de vértices em execução.

```
1.  procedimento atualizaVerticesEmAndamento()
2.      Enquanto verticesEmAndamento < qtdMinions
3.          Se verticesDisponiveis <= 0
4.              Imprima "Não há vertices com status DISPONIVEIS para estarem
```

```
    EM EXECUÇÃO no momento..."
5.                  Fim
6.          Senao
7.                  adiciona o vertice atual no ArrayList verticesEmAndamento
8.                  remove vertice atual do ArrayList verticesDisponiveis
9.          Fim
```

Com a estrutura montada, podemos iniciar a pista(varredura pelo grafo). A varredura pelo grafo irá acontecer enquanto houver vértices sendo visitados pelos minions. Os vértices da para execução serão visitados e, conforme forem destruídos por completo, tiramos eles da lista de vértices para execução, e armazenamos eles em uma lista de vértices destruídos. Após a destruição de cada vértice e, conforme os minions vão avançando pela pista, atualizamos a lista de vértices disponíveis para serem executados, e a lista de vértices para execução, e além disso, contabilizamos o tempo necessário para a destruição do obstáculo demolido.

Com essa implementação, no momento que não houver mais vértices para serem visitados pelos minions, a varredura será finalizada, e informará ao usuário o tempo total gasto para a execução total da pista pela equipe de minions.

```
1.  procedimento executaCaminhamento(Graph grafo, int posicaoVertice)
2.          Enquanto houver vertices em andamento
3.                  Para cada elemento do ArrayList de VerticeEmAndamento
4.                          Se TempoVertice <= menorTempo
5.                                  menorTempo = TempoVertice
6.                                  verticeMenorTempo = vertice
7.                          Fim
8.                  Fim
9.                  tempoTotal = tempoTotal + menorTempo
10.                 Remove verticeEmAndamento(verticeMenorTempo)
11.                 Adiciona VerticesExecutados(verticeMenorTempo)
12.
13.                 Para cada elemento do ArrayList de verticesAdjacentes
14.                         Se verticeAdjacente = verticePai
15.                                 Remove verticeMenorTempo do ArrayList ListaDependencia
16.                         Fim
```

```
17.                 Fim
18.
19.                 Para cada vertice do ArrayList de VerticesAndamento
20.                         tempoAtual = vertice.getTempo
21.                         tempoRestante = tempoAtual - menorTempo
22.                         vertice.setTempo(tempoRestante)
23.                 Fim
24.
25.             grafo.atualizaDisponiveis();
26.             grafo.atualizaVerticesEmAndamento()
27.         Fim
28.  Fim
```

Resultados obtidos

Com a implementação do algoritmo desenvolvido, após sua execução para os casos de teste disponibilizados, obteve-se as menores equipes de minions para executar as pistas no menor tempo possível:

CASOS DE TESTE		
Arquivo Teste	Quantidade Ideal Minions	Tempo De Execução da Pista
cinco.txt	9	699
oito_enunciado.txt	3	365
dez.txt	5	803
trinta.txt	9	1437
cinquenta.txt	9	2245
cem.txt	19	2327
quinhentos.txt	55	5250
mil.txt	54	7507
mil_e_quinhentos.txt	76	9448
dois_mil.txt	84	13064

Figura 7: **Melhor quantidade de minions para cada arquivo**

Análise do desempenho dos minions

A fim de analisar o desempenho das equipes de minions que participaram do treinamento, juntamos os seguintes dados de desempenho:

	Arquivo teste					
Equipe de minions	cinco.txt	dez.txt	cinquenta.txt	cem.txt	quinhentos.txt	mil.txt
1	3081	1932	12360	26336	126173	252810
2	1669	996	6340	13445	63377	126786
3	1291	1025	4360	9195	42438	84684
5	950	835	3298	5932	26130	51327
10	699	803	2245	3309	13598	26677
20	699	803	1824	2327	7522	13605
50	699	803	1824	2245	5294	7562
100	699	803	1824	2245	5250	6938
200	699	803	1824	2245	5250	6938
500	699	803	1824	2245	5250	6938
800	699	803	1824	2245	5250	6938
1.000	699	803	1824	2245	5250	6938
2.000	699	803	1824	2245	5250	6938
5.000	699	803	1824	2245	5250	6938
	Tempo de execução da pista					

Figura 8: **Resultados de execução dos arquivos**

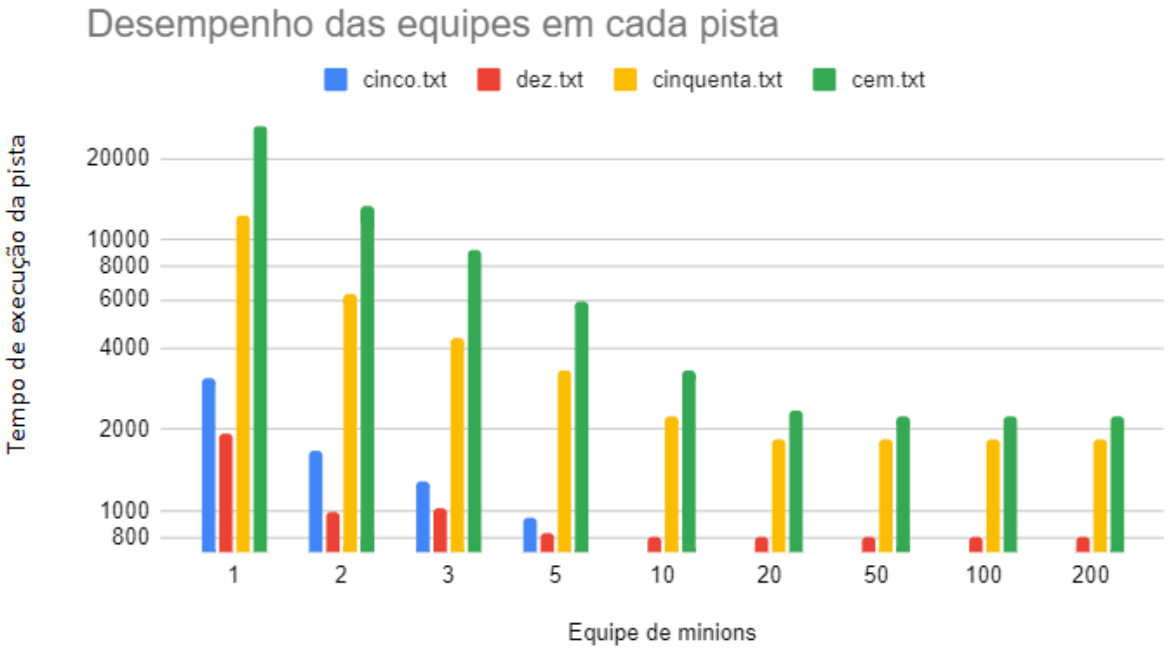


Figura 9: **Gráfico comparador do desempenho das equipes em escala logarítmica**

Os resultados obtidos não são de surpreender. Independentemente da pista escolhida, as equipes

com menos minions levam mais tempo para executar a pista. Devido a regra de apenas um minion poder destruir um obstáculo, não é surpreendente também que equipes muito volumosas tenham desempenho igual, pois mesmo que tenhamos dezenas de obstáculos que podem ser destruídos, todos eles estarão sendo executados e mesmo assim haverão minions procrastinando sem fazer nenhuma tarefa. Devemos observar também que independentemente da quantidade de minions presentes na equipe, quanto mais obstáculos presentes na pista, mais tempo levará para ela ser concluída.

Desse modo, percebemos que o tempo de execução da pista(varredura do grafo) é diretamente proporcional ao número de obstáculos(vértices) presentes nela.

Análise de Complexidade do Algoritmo

Com o algoritmo implementado, e com os dados de desempenho em mãos, podemos então analisar sua complexidade para sabermos o quão eficiente ele é. Analisando nosso algoritmo e os resultados obtidos, podemos afirmar que:

1. Se trata de um algoritmo iterativo;
2. O algoritmo realiza constantes verificações nos vértices do grafo para apurar quais estão disponíveis para serem visitados;
3. A quantidade de minions na equipe influencia (até certo ponto) no tempo de execução da pista;
4. O tempo de execução da pista(grafo) está diretamente ligada a quantidade de obstáculos(vértices) que ela possui;
5. Nosso algoritmo tem tendências de comportamento O(n) linear

Analisando a estrutura da representação do grafo e do algoritmo de caminhamento implementado, percebe-se que o algoritmo percorre cada vértice do grafo. Assim sendo, e com o apoio dos resultados de desempenho obtidos, deduzimos que o consumo de tempo para executar a pista(grafo) por completo é O(V), sendo V o número de vértices(obstáculos) pertencentes ao grafo. Porém, como há vértices adjacentes ligados por arestas, ou seja, há obstáculos que possuem dependência de outros para serem executados, a espera para estarem disponíveis para serem executados também deve ser levada em consideração.

Assim como os vértices, cada aresta do grafo é verificada uma única vez. Pelo fato da quantidade de arestas ser totalmente independente da quantidade de vértices presentes, a quantidade de tempo

utilizada para sua verificação não depende do tamanho do grafo, e portanto, o tempo total gasto nas verificações das arestas, ou destruição dos "vértices pai", é O(A), sendo A o número de arestas do grafo.

Posto isso, se o grafo possui V vértices e A arestas, o algoritmo consome um tempo aproximado de V + A unidades de tempo, ou seja, o tempo de execução do algoritmo é diretamente proporcional ao tamanho do grafo, e a quantidades de arestas existentes.

Constatamos assim que o algoritmo implementado consome O(V+A) unidades de tempo, e pelo fato de V e A estarem ligadas ao tamanho do grafo, conclui-se que o algoritmo é linear.

Conclusões

A solução apresentada se mostrou simples, eficiente e de fácil entendimento. Apesar de ser iterativa, ela foi desenvolvida de forma razoavelmente clara, não precisando de uma implementação complicada. Acreditamos ter escolhido a alternativa de representação mais eficiente para o grafo, sendo uma solução interessante para o problema proposto, conseguindo trazer de maneira satisfatória a interpretação, e o caminhamento de grafos que envolviam uma grande quantidade de vértices.

Referências

[1] CORMEN, T. H.; LEIERSON, E. C.; Rivest, R. L.: **"Introduction to Algorithms"**. Mc-Graw Hill Book Co., The MIT Electrical Engineering and Computer Science Series, Cambridge, 1990.