Fenícios e Grafos

Jocemar Nicolodi Junior[*], Mauro Sachs[†]
Faculdade de Informática — PUCRS

1 de julho de 2023

Resumo

O desafio proposto para este trabalho da disciplina de Algoritmos e Estrutura de Dados II consiste em calcular o combustível gasto na menor rota saindo do primeiro porto representado no mapa e percorrendo os demais portos, respeitando a ordem necessária para que a tarefa seja completada com sucesso. Após visitar todos os portos, conforme as regras especificadas, os fenícios devem retornar para "casa". O algoritmo desenvolvido, calcula o combustível necessário para que cada uma das rotas seja executada com base nos mapas fornecidos. Só é possível executar movimentações para Norte, Sul, Leste e Oeste, e alguns portos são inalcançáveis e, portanto, não devem ser visitados. Desta forma, este artigo busca detalhar como esta tarefa foi realizada, explicando a estrutura do algoritmo, e detalhando a lógica utilizada, assim como justificando-a. A solução foi elaborada em Java, mas o detalhamento acima explicado é feito através de pseudocódigos.

Introdução

O desafio proposto é inerente a utilização de grafos, sendo a busca por menores rotas uma das principais utilizações dos mesmos. Pela facilidade de se trabalhar com arrays e a eficiência para acessar posições, esta metodologia foi escolhida, como será detalhado posteriormente. O material e as classes estudadas em aula, tiveram grande influência sobre o trabalho final. Os casos de teste foram fornecidos em arquivos .txt e possuem em sua primeira linha as quantidades de linhas e colunas do mapa e nas demais linhas as informações de portos, pontos navegáveis e não navegáveis. Números entre 1 e 9 representam os portos e a ordem em que eles devem ser visitados. Os caracteres *"."* e *"*"* representam, nesta ordem, pontos navegáveis e não navegáveis. Os mapas variam em seus tamanhos de 16 linhas e 30 colunas a 1111 linhas e 2000 colunas.

Conforme já detalhado, os grafos são muito utilizados para cálculo de rotas, e portanto, a verificação da menor rota é calculada utilizando esta estrutura. Com base na classe *Graph*, fornecida em aula, foi criada a classe *MapgGraph*. Ainda de acordo com o material apresentado em aula, basedo em Sedgewick e Wayne, a utilização de lista de adjacências é eficaz em casos práticos, pois no geral, os grafos tendem a possuir um baixo grau em cada vértice. A classe *MapGraph*, possui os métodos *buildGraph()* e *setPostPositions()*, para criar o grafo com o mapa informado e verificar as posições dos portos neste mapa. Além destes métodos a classe possui o método *getFuel()*, que retorna de fato quanto de combustível é necessário para que se vá de um ponto ao outro. Na classe *App*, está o método *main()*, que lê as linhas do arquivo, e utiliza os métodos descritos acima para verificar o combustível necessário entre cada um dos portos e, ao final, retornar ao ponto inicial. Assim, é possível retornar a quantidade de combustível considerando que a cada novo vértice visitado, uma unidade do mesmo é utilizado.

[*] jocemar.nicolodi@edu.pucrs.br
[†] mauro.pereira@edu.pucrs.br

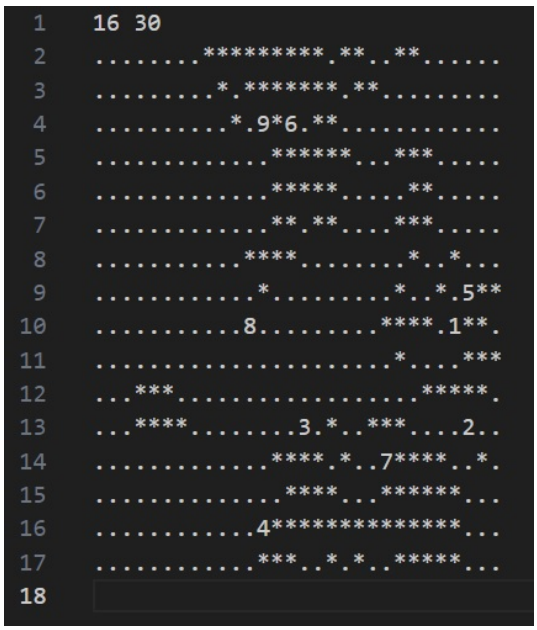

Figura 1: Exemplificação de arquivo txt com um mapa.

Classe MapGraph: Método construtor e buildGraph

Primeiramente, vale explicar as variáveis da classe, seu método construtor e o método *buildGraph*. O *graph*, que é a estrutura utilizada para salvar todas as informações do mapa, vértices e arestas, recebe como parâmetro a quantidade de vértices calculada através da multiplicação entre a quantidade de caracteres que representam as posições possíveis no mapa. A variável *map* representa o array bidimensional com os caracteres do mapa, *rols* e *cols* representam o tamanho deste array e são utilizados para que a instância de Graph seja criada com o tamanho correto. Finalmente, *postPositions* é um array que armazena a localização dos portos a serem visitados.

```
1  procedimento MAPGRAPH(char[][] map)
2      rows ← map.TAMANHO
3      cols ← map[0].TAMANHO
4      graph ← novo Graph(rows ∗ cols)
5      map ← map
6      postPositions ← novo int[9]
7      SETPOSITIONS()
8      BUILDGRAPH()
```

O primeiro método a ser utilizado é o *buildGraph*. Através de um for que percorre todas as colunas de cada uma das linhas, os vértices adjacentes de cada vértice são salvos. Com isto, poder-se-á percorrer apenas caminhos permitidos.

```
1  procedimento BUILDGRAPH()
2      int rows ← map.TAMANHO
3      int cols ← map[0].TAMANHO
4      para i← 0 até rows faça
5          para j← 0 até cols faça
6              se map[i][j] ≠ '∗' então
7                  int v ← i ∗ cols + j
8                  se i > 0 e map[i−1][j] ≠ '∗' então
9                      int w ← (i −1) ∗ cols + j
10                     graph.ADICIONAARESTA(v , w)
```

```
11                      fim
12                      se i < rows −1 e map[i + 1][j] ≠ '∗' então
13                          int w ← (i+1) ∗ cols + j
14                          graph.ADICIONAARESTA(v , w)
15                      fim
16                      se j > 0 e map[i][j−1] ≠ '∗' então
17                          int w ← i ∗ cols + (j −1)
18                          graph.ADICIONAARESTA(v , w)
19                      fim
20                      se j < cols−1 e map[i][j + 1] ≠ '∗' então
21                          int w ← i ∗ cols + (J + 1)
22                          graph.ADICIONAARESTA(v , w)
23                      fim
24                  fim
25              fim
26          fim
27      fim
```

Classe MapGraph: Método setPostPositions

Este método percorre todas as posições e verifica quais delas são portos, salvando no array postPostions sua respectiva localização. A posição do porto 9, por exemplo ficará salva na posição 8 do array. Saber se um vértice é um porto é essencial para que toda a as regras possam ser seguidas e com a metodoliga abordada, cada posição do array pode ser acessada em O(1).

```
1   procedimento SETPOSTPOSITIONS()
2           int postCount ← 0
3           para i← 0 até rows faça
4           para j← 0 até cols faça
5                   char c ← map[i][j]
6                   se c ⩾ '1' e c⩽ '9' então
7                   postPositions[c −'1'] ← i ∗ cols + j
8                   postCount ← postCount + 1
9                   fim
10          fim
11          fim
12          se postCount ≠ 9 então
13                  lançar exceção RuntimeException
14          fim
15  fim
```

Classe MapGraph: Método getFuel

Para calcular o gasto de combustível entre dois pontos, é necessário encontrar a melhor rota entre eles. Para que isso seja possível, todos os vértices do grafo são percorridos em largura, e o custo com combustível, que começa com zero, é incrementado em um a cada novo ponto visitado da rota. O primeiro caminho que atingir o vértice final é o caminho mais curto e assim será possível calcular o combustível necessário para a viagem. Uma fila é utilizada como estruturada para a busca em largura. Um porto é adicionado a fila, removido e marcado como visitado. Se este porto é o destino, a distância

até ele é retornada, caso contrário, seus adjacentes são adicionados à fila, para que o caminhamento em largura continue até o destino ser encontrado.

```
1  procedimento GETFUEL(int i , int k):
2      int[] distTo ← novo int[graph.V()]
3      int[] postPositions ← GETPOSTPOSITIONS()
4      boolean[] marked ← novo boolean[graph.V()]
5      int source ← postPositions[i−1]
6      int target ← postPositions[k−1]
7
8      distTo[source] ← 0
9      Fila< Inteiros>  queue ← nova Fila
10     queue.ACRESCENTA(source)
11     marked[source] ← true
12
13     enquanto fila ≠ vazia faça
14         int v ← fila.REMOVE()
15         se v = target então:
16             retornar distTo[v] // encontrou o ponto de destino , retorna a menor distância
17         fim
18         boolean differentPost = falso
19         para j ← 0 j até 9 faça
20             se v ≠ source e v = postPositions[j] então
21                 differentPost ← verdadeiro
22                 quebrar o loop
23
24         se differentPost = Verdadeiro então
25             continuar
26
27         para cada w em graph.adj(v) faça
28             se marked[w] = falso então
29                 marked[w] ← verdadeiro
30                 distTo[w] ← distTo[v] + 1
31                 fila.ACRESCENTA(w)
32             fim
33         fim
34         retornar −1
35         // não foi possível encontrar um caminho até o ponto de destino ,
36         nenhum custo de combustível será retornado
37     fim
38 fim
```

Com isto, nenhum custo será acrescentado a uma rota que não possa ser realizada. Se um porto for inacessível, por exemplo, o retorno do custo da viagem entre o ponto de origem e este porto inacessível, retornará -1, o que será tratado na classe main. Além disto, quando um caminho passar por um porto, ele será ignorado, uma vez que nos exemplos fornecidos, rotas com portos no caminho são ignoradas.

Classe App

Conforme detalhado anteriormente, os mapas são fornecidos em documentos txt e representados com caracteres. Na classe *App*, os arquivos são lidos para a memória, para que a estrutura de dados detalhada neste documento possa ser aplicada. Após a leitura do arquivo do mapa, as primeiras duas linhas são convertidas para inteiro e salvas em variáveis que serão utilizadas posteriormente para a criação do Grafo, uma vez que elas representam o número de linhas e de colunas, respectivamente. A partir da tercera linha, o mapa é lido e armazenado em um array bidimensional de Strings, *map*, permitindo que a estrutura do mapa se mantenha intacta e viabilizando a navegação conforme as regras de movimentação estipuladas. Então, é criada uma instância de *mapGraph*, tendo como parâmetro *map*. Duas constantes *PRIMEIROPORTO* e *ULTIMOPORTO*, guardam a numeração dos destinos a serem visitados. Para cada trajeto, tendo um porto como origem e um porto como destino, o método *getFuel* retorna o combustível necessário para aquele trajeto e, caso a função *getfFuel* retorne valor negativo, o porto é considerado inacessível e então é pulado, sendo considerado o próximo para o rota. Ao fim, o custo para o retorno ao primeiro porto é calculado.

```
1  int PRIMEIROPORTO ← 1
2  int SEGUNDOPORTO ← 9
3  int combustivel ← 0
4  int i ← PRIMEIROPORTO
5
6  para k = PRIMEIROPORTO + 1 até k←  ULTIMOPORTO faça
7          int custo = mapGraph.GETFUEL(i , k)
8          se custo ⩾ 0:
9          i = k
10         combustivel ← custo + combustivel
11     fim
12 combustivel += mapGraph.GETFUEL(i , PRIMEIROPORTO)
13 ESCREVER("Combustivel Necessario = " + combustivel)
```

Desta forma, utilizando os métodos da classe *MapGraph*, podem ser consultados os custos de cada trecho da viagem total. Vale salientar que a condição que verifica o custo, anula trechos que não poderão ser realizados, desconciderando-os do cálculo de combustível final. Navegando de um porto para outro e somando o combustível gasto, será possível imprimir o custo com combustível da menor rota de cada mapa.

Conclusão

Encontrar o melhor caminho percorrendo o grafo em largura, simplifica a lógica utilizada pois como todos os caminhos avançam igualmente, o primeiro caminho a encontrar o destino final, será o mais curto. A execução do algoritmo é praticamente instântanea, evidenciando sua eficiência para a aplicação proposta. Para reduzir este gasto desnecessário de processamento e memória de percorrer tantos caminhos, precisaríamos aplicar conceitos de inteligência ao algoritmo, analisando quais de fato poderiam reduzir o seu tempo de execução. De qualquer maneira, a solução encontrada soluciona os casos de testes fornecidos com o problema, encontrando as melhores rotas em tempo suficiente para a utilização do mesmo. Sua eficiência se dá pela complexidade de tempo da travessia BFS, que é O(V + E), onde V e E são o número total de vértices e arestas no grafo, respectivamente. A complexidade da solução apresentada é esta para cada rota calculada. Ou seja, para cada mapa com 9 portos, O(V+E) * 9(Número de Portos). Conforme visto acima, pudemos verificar que a saída do algoritmo responde diretamente o desafio final, após aplicar todas as regras da viagem a cada um dos portos, *(...)então sua missão é calcular o quanto de combutível é necessário*. A utilização de arrays também simplificou

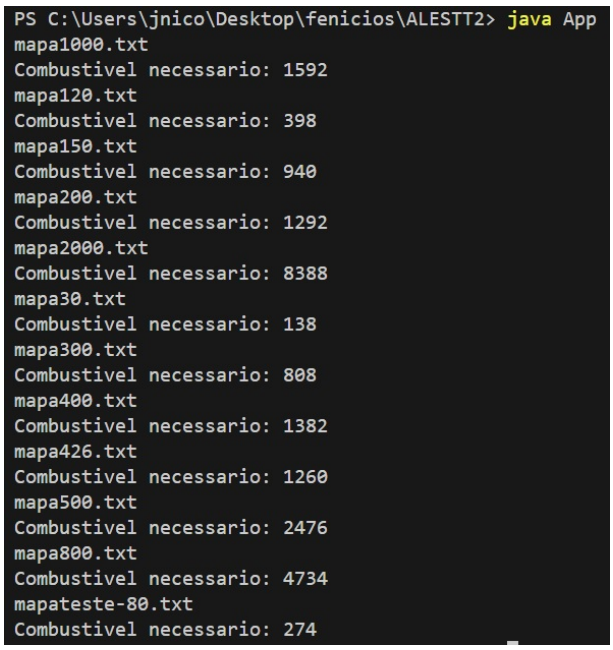

Figura 2: Saída do algoritmo em Java.

o trabalho, permitindo acesso direto às posiçõse de cada vértice e evitando a necessidade de criação de nodos e controle de filhos de cada nodo. A solução proposta se torna simples de se compreender e eficiente, como detalhado acima.

Referências

[1] Sedgewick, Robert ; Wayne, Kevin. . “ **Algorithms, 4th Edition.**”. Addison-Wesley Professional, 2011.

[2] Documentação Java; Disponível em [https://docs.oracle.com/en/java/index.html].

[3] Sedgewick, Robert; Wayne, Kevin; Cohen, Marcelo. Graph.java. Disponível em: https://github.com/mflash/grafos_20231]. Acessado em: [06/06/2023].