Relatório do Trabalho Prático 2

Pedro Henrique Barbieri – T12

Faculdade de Ciência de Dados e Inteligência Artificial - PUCRS

20 de junho de 2025

**Resumo**

Neste artigo trataremos a resolução do problema disposto no trabalho prático dois da cadeira ‘Algoritmos e Estruturas de Dados II’ (2025/1) do curso ‘Ciência de Dados e Inteligência Artificial’, ‘Os fenícios estão chegando’.

Ao decorrer deste relatório analisaremos o problema proposto, que se trata do cálculo do menor caminho possível de viagem para os fenícios fazerem indo do porto 1 até o porto 9 e voltando ao porto original, evitando áreas inacessíveis. Também apresentaremos uma solução ao enunciado e veremos seus resultados.

# **Introdução**

Neste trabalho, abordamos o planejamento de rotas de navios fenícios em mapas marítimos caracterizados por células navegáveis (“.”) e obstáculos (“\*”). Cada mapa de teste é fornecido em arquivo texto, cujo cabeçalho indica o número de linhas e colunas, seguido pela grade que contém também marcações de portos numerados de 1 a 9. A viagem deve começar no porto 1, seguir na ordem crescente dos portos disponíveis e, ao término, retornar “para casa”, sempre consumindo uma unidade de combustível por movimento ortogonal (Norte, Sul, Leste ou Oeste). Se um porto estiver inacessível, ele é simplesmente ignorado, passando-se ao próximo da sequência.

Para representar a área navegável como um grafo e encontrar distâncias mínimas entre portos, faremos uso das classes **Graph** e **BreadthFirstSearch** (o qual foi necessário fazer algumas correções), disponibilizadas no repositório GitHub fixado na página da disciplina de Algoritmos e Estruturas de Dados II no Moodle. Utilizamos do algoritmo BFS para pesquisa no mapa pelos portos pois ele será o método que nos garantirá o menor caminho possível. Complementarmente, desenvolvemos a classe **Mapa** para ler o arquivo de teste, armazenar suas dimensões, as linhas do mapa e as coordenadas dos portos, além da classe **Main**, que organiza a leitura de casos e a soma dos custos de combustível.

Ao final de tudo, escreveremos na pasta “mapas\_caminho”, em arquivos de texto distintos, cada mapa atualizado com o caminho feito destacado com caracteres ‘X’ e o consumo total de combustível do menor caminho na primeira linha do arquivo, apenas para melhor legibilidade.

# **Solução**

Para a solução do problema proposto utilizaremos algumas classes auxiliares para nos ajudar no programa principal.

“Mapa.java” será uma classe auxiliar que recebe o arquivo de caso de teste e irá transformá-lo em nosso mapa, no qual buscaremos o menor caminho possível para visitar todos os portos possíveis e voltar para o começo.

*public class Mapa {  
 public int height; // Quantidade de linhas do mapa  
 public int width; // Quantidade de colunas do mapa  
 public String[] lines; // O mapa convertido em uma lista de strings de suas // linhas (utilizado para coordenadas)  
 public int[] pos\_nums = new int[10]; // Coordenadas dos portos no mapa  
 Graph grafo; // Grafo do mapa indicando pontos navegaveis  
  
 public Mapa(File arquivo) {  
 LerArquivo(arquivo);  
 initializeGraph();  
 } ... }*

“LerArquivo” é um simples método que recebe um File arquivo, lê sua primeira linha esperando dois números separados por espaço, indicando as dimensões do mapa, e salva elas nas variáveis *height* e *width*. Após isso *lines* é inicializado com o tamanho de *height* e um loop começa que salva cada linha do mapa separadamente na lista de Strings *lines*.

*public void LerArquivo(File arquivo) {  
 try {  
 FileReader fr = new FileReader(arquivo);  
 BufferedReader in = new BufferedReader(fr);  
  
 String line = in.readLine();  
 String[] cord\_xy = line.split(" ");  
 height = Integer.parseInt(cord\_xy[0]);  
 width = Integer.parseInt(cord\_xy[1]);  
  
 lines = new String[height];  
  
 for (int i = 0; i < height; i++) {  
 line = in.readLine();  
  
 lines[i] = line;  
 }  
 in.close();  
 } catch (FileNotFoundException e) {  
 System.out.println("Arquivo não encontrado");  
 } catch (IOException e) {  
 System.out.println("Erro na leitura do arquivo");  
 }  
}*

Já “InitializeGraph” é o método encarregado de criar o grafo e adicionar suas arestas. O algoritmo itera em cada caractere do mapa, da esquerda à direita de cima para baixo. A partir disso ele verifica se o vértice atual é válido (diferente de ‘\*’) para conexão com outros, pulando para o próximo se não for. Ao mesmo tempo, se o vértice é válido, ele confere se o caractere naquela coordenada é um dígito, armazenando-a se for, e verifica se o ponto abaixo e a sua direita também são válidos, criando as conexões necessárias.

Lembrando que o grafo é inicializado utilizando coordenadas em apenas uma dimensão, então, cada coordenada é calculada da seguinte forma: *i*\**width* + *j*, onde *i* é a linha, *j* é a coluna e *width* é o número total de colunas (largura)

*public void initializeGraph() {  
 grafo = new Graph(height \* width);  
  
 for (int i = 0; i < height; i++) {  
  
 for (int j = 0; j < width; j++) {  
 char c = lines[i].charAt(j); // Caractere atual  
  
 // Ignora posições inacessiveis  
 if (c == '\*') continue;*

*// Salva coordenadas de cada porto  
 if (Character.isDigit(c)) {  
 pos\_nums[c - 48] = i\*width + j;  
 }*

*// Conectar vertice atual ao vertice a sua direita  
 if (j+1 < width && lines[i].charAt(j+1) != '\*') {  
 grafo.addEdge(i\*width + j, (i)\*width + j+1);  
 }*

*// Conectar vertice atual ao vertice abaixo  
 if (i+1 < height && lines[i+1].charAt(j) != '\*') {  
 grafo.addEdge(i\*width + j, (i+1)\*width + j);  
 }  
  
 }}}*

Além da classe Mapa também utilizamos a classe Graph para inicializar o grafo com *height*\**width* vértices e cria um objeto bag, que seria basicamente uma linked list que será utilizada para fazer a conexão entre vértices no método ‘addEdge()’, que apenas soma mais um a contagem de arestas e adiciona a conexão passada à linked list de adjacência.

E por último temos a classe BreadthFirstSearch (procura em largura), que foi o método de procura escolhido devido à sua habilidade em encontrar o menor caminho até um vértice garantidamente. Um algoritmo de DepthFirstSearch (procura em profundidade) não garante o caminho mínimo, porque a sua estratégia de busca é focada em explorar o caminho mais fundo possível antes de retroceder, e não em minimizar o número de passos ou o custo do caminho. E algoritmos de Dijkstra também não seriam apropriados para essa situação pois não se trata de um grafo ponderado que estamos trabalhando.

O algoritmo de BFS é inicializado informando o grafo e o ponto para iniciar a busca. Para fazer as consultas de caminho no programa principal utilizamos dois principais métodos, ‘hasPath’ para verificar a existência de caminho para o próximo porto e ‘pathTo’ para obtermos o caminho e fazer a contagem de movimentos.

Agora que já temos nossas classes auxiliares bem definidas, vamos montar nosso arquivo “Main.java”. Iniciamos ele abrindo os arquivos de caso de teste e iterando sobre cada um deles.

*public static void main(String[] args) {  
 File arquivo = new File("src/casosDeTeste");  
 File[] arquivos = arquivo.listFiles();  
 assert arquivos != null;  
  
 int contador = 0; // Servira para dar nomes distintos a cada arquivo de mapa  
 for (File file : arquivos) { ... }*

Então para cada arquivo de teste criaremos um objeto Mapa fornecendo o arquivo que estamos iterando atualmente. Também faremos uma cópia do mapa para posteriormente atualizar com o caminho navegado e armazená-lo para visualização.

*Mapa atual = new Mapa(file);  
  
char[] map2 = new char[atual.height \* atual.width];  
for (int i = 0; i < atual.height; i++) {  
 for (int j = 0; j < atual.width; j++) {  
 map2[i \* atual.width + j] = atual.lines[i].charAt(j);  
 }  
}*

Partimos então para a navegação! Mas antes disso, devemos definir algumas variáveis que serão atualizadas toda vez que chegarmos em um novo porto. ‘combustivel\_a\_gastar’ será referente ao total de combustível que os fenícios gastarão para fazer o menor caminho possível. Também inicializamos o BFS informando o grafo do mapa atual e onde começaremos a viagem (porto 1).

Prontos para iniciar a pesquisa de menor caminho, primeiramente testamos se existe caminho do porto que estamos atualmente para o próximo, caso seja impossível, pulamos este porto e tentamos ir para o seguinte. Assim que identificarmos um porto alcançável, utilizamos do BFS para identificar o caminho até ele e então iteramos sobre as coordenadas que formam o caminho e para cada uma delas somamos 1 ao ‘combustivel\_a\_gastar’ e atualizamos a cópia do mapa para salvamento do caminho, e quando terminamos a iteração sobre as coordenadas removemos 1 pois o caminho para cada porto inclui o ponto de partida, que acaba sendo explorado 2 vezes, no final de um caminho e no começo do outro. Ao final atualiza-se o ponto de partida do bfs para a próxima iteração.

*int combustivel\_a\_gastar = 0;  
BreadthFirstSearch bfs = new BreadthFirstSearch(atual.grafo, atual.pos\_nums[1]);  
for (int i = 2; i < atual.pos\_nums.length + 1; i++) {  
 int j = i == 10 ? 1 : i;  
  
 if (!bfs.hasPath(atual.pos\_nums[j])) {  
 continue;  
 }  
  
 Iterable<Integer> p = bfs.pathTo(atual.pos\_nums[j]);  
  
 for (Integer p1 : p) {  
 System.out.print(p1 + " ");  
  
 if (!Character.isDigit(map2[p1])) map2[p1] = 'X'; // Atualiza o mapa a ser salvo  
  
 combustivel\_a\_gastar++;  
 }  
 combustivel\_a\_gastar--; // Desconta o ponto de partido  
  
 if (i != 10) bfs = new BreadthFirstSearch(atual.grafo, atual.pos\_nums[i]);*

*// Atualiza o bfs para na próxima iteração começar no porto que chegamos agora*

*// (se já não estivermos na última iteração)  
}*

Para concluirmos nossa iteração sobre o arquivo atual, utilizamos a classe Out (também disponível no GitHub da aula) para escrever em um arquivo ‘mapa + contador + .txt’ o consumo de combustível do caminho total e o mapa atualizado contendo o caminho destacado em caracteres ‘X’.

*Out out = new Out("mapas\_caminho/mapa" + contador + ".txt");  
  
out.println("Custo mínimo para a viagem: " + combustivel\_a\_gastar);  
for (int i = 0; i < atual.height; i++) {  
 for (int j = 0; j < atual.width; j++) {  
 out.print(map2[i \* atual.width + j]);  
 }  
 out.println();  
}  
out.close();  
  
contador++;*

*}*

Com isso concluímos a aplicação da solução para o problema. Nas próximas iterações sobre ‘arquivos’ o mesmo processo citado acima será aplicado aos demais arquivos de testes e o programa se encerrará.

# **Resultados**

Gasto de combustível mínimo no caso 0: 248

Gasto de combustível mínimo no caso 1: 598

Gasto de combustível mínimo no caso 2: 1112

Gasto de combustível mínimo no caso 3: 2210

Gasto de combustível mínimo no caso 4: 3510

Gasto de combustível mínimo no caso 5: 11518

# **Conclusões**