Relatório do trabalho T2 (Os Fenícios)

Conrado Crestani, Felipe Freitas e Luiza Heller

PUCRS

# 1. Introdução

O trabalho apresentado neste relatório tem como objetivo avaliar o conhecimento dos alunos a respeito do conteúdo de Grafos. Uma vez que grafos representam objetos e as relações entre eles, o trabalho proposto foi a respeito de navegações, um tópico bom para desenvolver e aplicar os conceitos práticos de grafos.

# 2. O problema

O problema proposto foi introduzido com a imagem do povo Fenício, que percorreram mares devido ao seu grande envolvimento com o comércio. O enunciado informa que esse povo está indo atrás de formas de encurtar as viagens e aumentar seus lucros. Para isso, a problemática proposta para este trabalho, foi a de desenvolver um código que realize o cálculo de combustível necessário para realizar o percurso, passando por alguns portos e retornando para o local de origem. É importante ressaltar que alguns detalhes terão que ser levados em conta e são importantes para que o código funcione como deve.

# 3. Processo de solução

Para resolver o problema, o primeiro passo lógico é ler o arquivo com os mapas em questão e interpretá-lo. Então o programa começa lendo o arquivo definido (ou todos na pasta de testes) e precisa entender duas coisas: a quantidade de linhas e colunas do mapa, e seus pontos com suas. Para o primeiro: basta ler as duas palavras (considerando uma palavra por espaço) da primeira linha como um número, e para as informações de cada vértice (ponto) no mapa, percorremos o resto do arquivo, a cada linha passando por todas suas colunas (caracteres) e, caso o ponto em questão seja um porto (definido aqui por um dígito de 1-9), separamos ele em uma estrutura adicional que serve para guardar a posição deste no mapa:

private HashMap<String, Integer> getMapDimensions(Scanner fileScanner) {

   HashMap<String, Integer> dimensions = new HashMap<>();

   String[] header = fileScanner.nextLine().split(" ");

   dimensions.put("lines", Integer.parseInt(header[0]));

   dimensions.put("columns", Integer.parseInt(header[1]));

   return dimensions;

}

private char[][] fillMap(Scanner fileScanner, HashMap<String, Integer> dimensions) {

 char[][] map;

 int lineAmount = dimensions.get("lines");

 int columnAmount = dimensions.get("columns");

  try {

  map = new char[lineAmount][columnAmount];

  int counter = 0;

  for (int i = 0; fileScanner.hasNextLine(); i++) {

   String line = fileScanner.nextLine();

   for (int j = 0; j < columnAmount; j++) {

    char value = line.charAt(j);

    map[i][j] = value;

    try {

     int vertice = Integer.parseInt(String.valueOf(map[i][j]));

     this.harbours[vertice] = counter;

    }

    counter++;

   }

  }

 return map;

}

Depois de criar este mapa “bidimensional”, temos de "viajar” por todo ele e analisar cada um dos pontos, podendo tomar 3 decisões a partir daí; se o ponto em questão for um ‘\*’, entendemos que é um bloqueio no mar e deve ser ignorado, já se for ou água (‘.’) ou um número que representa um porto ([1-9]), devemos tentar guardar seus 4 pontos adjacentes, caso eles não sejam obstáculos, como conexões em um grafo que tem um vértice para cada ponto do mapa.

private Graph linkEdges(char[][] worldMap, HashMap<String, Integer> dimensions) {

    int lineAmount = dimensions.get("lines");

    int columnAmount = dimensions.get("columns");

    Graph graph = new Graph(lineAmount \* columnAmount);

  for (int i = 0; i < lineAmount; i++)

      for (int j = 0; j < columnAmount; j++) {

       if (worldMap[i][j] == '\*') continue;

       int origin = (i \* columnAmount) + j;

      // Olha pra cima

      if (i > 0 && worldMap[i - 1][j] != '\*') {

       graph.insertEdge(origin,

        ((i - 1) \* columnAmount) + j

       );

      }

      // Olha pra baixo

      if (i < lineAmount - 1 && worldMap[i + 1][j] != '\*') {

       graph.insertEdge(origin,

        ((i + 1) \* columnAmount) + j

       );

      }

      // Olha pra esquerda

      if (j > 0 && worldMap[i][j - 1] != '\*') {

       graph.insertEdge(origin,

        (i \* columnAmount) + (j - 1)

       );

      }

      // Olha pra direita

      if (j < columnAmount - 1 && worldMap[i][j + 1] != '\*') {

       graph.insertEdge(origin,

        (i \* columnAmount) + (j + 1)

       );

      }

     }

  return graph;

}

Neste momento, já temos todo mapa mapeado com suas rotas navegáveis, e devemos iniciar a navegação do porto 1 ao porto 2, seguida pela navegação do 2 ao 3 assim por diante, até chegar no 9. Para fazer isso, primeiro vemos se é possível viajar para o próximo porto ou se este está bloqueado; caso esteja bloqueado, tentamos navegar até o próximo porto, partindo do que estamos e, caso contrário, passamos a navegar partindo deste próximo para o porto seguinte. Quando chegamos no porto 9, ou no porto mais longe o possível caso o(s) último(s) esteja(m) bloqueado(s), calculamos a distância do último para o primeiro, que é sempre uma viagem possível, visto que na pior das hipóteses basta voltar por onde viemos.

private void navigate(Dijkstra dijkstra, int origin, int destination) {

    int limit = harbours.length - 1;

    if (origin == limit || destination > limit) return;

  distances[destination] = dijkstra.getDistance(harbours[origin], harbours[destination]);

  if (dijkstra.isUnreachable(harbours[destination])) {

      navigate(dijkstra, origin, destination + 1);

  } else {

      furthest = destination;

      navigate(dijkstra, destination, destination + 1);

  }

}

# 4. Evidências de que o problema foi resolvido

Resultados do arquivo gerado pelo programa em modo de teste:

Tempo decorrido para analisar o arquivo "mapa\_15\_80": 00:63.698600

Distâncias:

Não é possível chegar ao porto 2

Distância para o porto 3: 14

Distância para o porto 4: 3

Distância para o porto 5: 50

Distância para o porto 6: 57

Distância para o porto 7: 49

Distância para o porto 8: 40

Distância para o porto 9: 42

Distância para o porto 1: 49

Distância total: 304

Tempo decorrido para analisar o arquivo "mapa\_30\_120": 00:28.95100

Distâncias:

Distância para o porto 2: 82

Distância para o porto 3: 66

Não é possível chegar ao porto 4

Não é possível chegar ao porto 5

Distância para o porto 6: 38

Distância para o porto 7: 119

Não é possível chegar ao porto 8

Distância para o porto 9: 114

Distância para o porto 1: 13

Distância total: 432

Tempo decorrido para analisar o arquivo "mapa\_500\_1000": 06:6079.911700

Distâncias:

Distância para o porto 2: 178

Distância para o porto 3: 404

Distância para o porto 4: 1017

Distância para o porto 5: 390

Distância para o porto 6: 389

Distância para o porto 7: 174

Distância para o porto 8: 282

Distância para o porto 9: 294

Distância para o porto 1: 298

Distância total: 3426

Tempo decorrido para analisar o arquivo "mapa\_50\_100": 00:20.201600

Distâncias:

Distância para o porto 2: 26

Distância para o porto 3: 74

Não é possível chegar ao porto 4

Distância para o porto 5: 53

Distância para o porto 6: 75

Não é possível chegar ao porto 7

Distância para o porto 8: 85

Distância para o porto 9: 108

Distância para o porto 1: 63

Distância total: 484

Tempo decorrido para analisar o arquivo "mapa\_60\_500": 00:84.421700

Distâncias:

Distância para o porto 2: 238

Distância para o porto 3: 58

Distância para o porto 4: 275

Distância para o porto 5: 110

Distância para o porto 6: 88

Distância para o porto 7: 39

Distância para o porto 8: 51

Não é possível chegar ao porto 9

Distância para o porto 1: 127

Distância total: 986

------------------------ RESULTADOS ------------------------

Total de arquivos analisados: 5

Tempo total de execução: 06:6276.328700

Tempo médio de execução: 01:1255.265740

# 5. Conclusões

Podemos concluir do trabalho diversas coisas, mas principalmente o como é bom utilizar-se de estruturas de dados previamente implementadas; como as classes Grafo e Dijkstra, que permitem uma abstração maior do código e foco principal na resolução do problema. Ainda, percebe-se que sempre há o que pode ser melhorado - por menor que seja - visto que será importante conforme o projeto amplia, e o código ser tão modular permite, por exemplo, utilizar outras estruturas de caminhamento no lugar do Dijkstra sem grandes mudanças ou até mesmo outro tipo de Grafo (valorado, direcional). Para tentar melhorar a complexidade, optamos por uma estrutura auxiliar para guardar a posição dos portos (para evitar percorrer e procurar toda a grade cada vez) e também por utilizar um pouco de matemática na hora de conectar as arestas, multiplicando o tamanho das linhas somada ao número da coluna para determinar o índice/”chave” de cada ponto no mar, mas vale ressaltar que o algoritmo possui uma complexidade de O(2n²), dada a necessidade de percorrer (2 vezes) o arquivo/mapa por altura e por largura, mas o 2 pode ser ignorado, resultando sempre nesta complexidade quadrática.