Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Curso de Ciência da Computação
Algoritmos e Estruturas de Dados II – Trabalho II
Professor: Edson Moreno
Aluno: Ricardo Süffert

RELATÓRIO DO TRABALHO II

1. DESCRIÇÃO E INSTRUÇÕES:

O programa apresentado neste relatório propõe uma solução para o problema introduzido para o Trabalho II da disciplina de Algoritmos e Estruturas de Dados II. A problemática apresentada para o trabalho consiste, basicamente, no **problema do caixeiro viajante**, bastante conhecido na área da Ciência da Computação. No contexto do trabalho, é entregue ao programa um arquivo texto que simula um mapa-múndi e que segue o seguinte formato:

- A primeira linha contém dois números, separados por um caractere de espaço em branco, sendo o primeiro a quantidade de linhas e o segundo a quantidade de colunas no mapa do arquivo;
- Todas as linhas subsequentes fazem parte do mapa em si, sendo compostas unicamente dos seguintes caracteres:
    - Pontos, que representam uma área navegável;
    - Asteriscos, que representam um obstáculo;
    - Números de 1 a 9, que representam os portos.

Nesse sentido, o programa deve partir do porto de número 1 e avançar até o último porto, passando por todos os portos, em ordem numérica, durante o caminho. Se algum dos portos não for acessível (ou seja, estiver "preso" dentro de obstáculos), o programa deve pulá-lo e buscar alcançar diretamente o porto seguinte. Após chegar ao porto final, o programa deverá retornar diretamente ao porto inicial, desta vez sem passar por nenhum outro porto no caminho. Todos os deslocamentos de um porto a outro devem, necessariamente, ser feitos pelo menor caminho possível, e considera-se que os movimentos possíveis são apenas na direção dos quatro pontos cardeais: norte (cima), sul (baixo), leste (direita) ou oeste (esquerda). Naturalmente, se, ao navegar para alguma dessas direções, o programa for encontrar um obstáculo (asterisco), então ele não deve tomar esse passo. Considerando (1) cada caractere do mapa como uma unidade de distância, (2) que cada unidade de distância consome uma unidade de combustível e (3) que o programa avança de caractere em caractere — isto é, de unidade de distância em unidade de distância —, a saída final do programa deverá ser o mínimo de combustível (que é igual à distância mínima) necessária para todo o trajeto — ou seja, para a ida do porto 1 até o último porto, passando por todos os intermediários, e o retorno do último porto ao porto 1, desta vez sem fazer paradas.

Informações importantes e instruções:

- **Para rodar o programa, basta executar a classe App**;

- A solução elaborada foi desenvolvida na linguagem Java;
- O programa conta com uma interface gráfica implementada com as funcionalidades da classe `JOptionPane` do Java, a fim de facilitar a interação entre o programa e o usuário. Qualquer interação é feita por meio dessa interface gráfica. Para erros na leitura do arquivo ou na execução do programa (exceções), também são apresentadas ao usuário mensagens de erro pertinentes por meio de interface gráfica e, em seguida, o programa é encerrado;
- O programa leva como entrada (*input*) um arquivo em formato de texto — no modelo especificado no enunciado do trabalho e na presente seção deste relatório — que contém as informações necessárias para a montagem da estrutura de armazenamento do mapa na memória do programa e, consequentemente, para o processamento do mapa pela aplicação;
- No código-fonte, os métodos e classes foram documentados no formato Javadoc, bem como o programa conta com uma série de comentários explicativos ao longo do código. A documentação Javadoc pode ser encontrada em `docs/javadoc/index.html`. A partir desse arquivo, é possível acessar a documentação de todas as classes do programa.

---

2. EXPLICAÇÃO DA SOLUÇÃO PROPOSTA E DAS CLASSES DO PROGRAMA:

Para implementar a solução apresentada neste relatório, foi utilizada a estrutura de grafos. Assim, o programa converte o arquivo de entrada em um grafo não dirigido, representado pela classe `Graph`. Ao fazer isso, o programa considera cada caractere do arquivo de entrada como um vértice e, para cada um desses vértices, adiciona como adjacente a ele apenas aqueles vértices alcançáveis (norte, sul, leste e oeste) e navegáveis — ou seja, não adiciona na lista de adjacência dos vértices os vértices representados por asteriscos. No entanto, a implementação apresentada para a estrutura de grafos armazena em seus vértices apenas valores inteiros e sequenciais, começando em zero, resultando em cada vértice do grafo sendo representado por um número inteiro único, gerado sequencialmente de acordo com a ordem de inserção do vértice no grafo. Para manter controle sobre os valores reais de seus vértices, o grafo conta com duas estruturas de dicionário, que são preenchidas no momento em que o grafo está sendo construído. O primeiro dicionário mapeia o código armazenado no grafo para seu respectivo valor `char` (ponto, asterisco ou número de porto), enquanto o outro mapeia cada porto do mapa (número) para seu respectivo código sequencial na estrutura do grafo. Com esses dois dicionários, tem-se total controle sobre os valores reais associados a cada vértice. Os dicionários foram implementados utilizando a classe `HashMap` do Java e, por se tratar de uma estrutura de dados extremamente eficiente, pouco afetam as consultas aos dicionários no desempenho do programa.

A lógica do algoritmo de busca é implementada na classe `BreadthFirstSearch`, que utiliza o algoritmo de **busca em largura (BFS)**. Esse

algoritmo segue o pseudocódigo abaixo, implementado em Java no código-fonte do programa. A partir dessa implementação, que recebe como parâmetro um grafo *g* e um vértice *s*, tem-se um vetor com a distância de cada vértice do grafo até *s*.

```
adicionar s em uma pilha auxiliar
anotar s como visitado
anotar que o vértice que levou a s é -1 (valor inválido)
anotar que a distância de s a s é 0
enquanto a pilha auxiliar não estiver vazia:
      remover o elemento do topo da pilha (top)
      para cada vértice adjacente (adj) a top:
            se adj ainda não foi visitado:
                  anotar que quem levou a adj foi top
                  anotar adj como visitado
                  anotar que distância s→adj = distância top→adj + 1
                  adicionar adj à pilha
```

Pseudocódigo que descreve a implementação do algoritmo BFS no programa

A classe `App` é a classe principal do programa, que efetua todas as interações com o usuário (entrada e saída). Enquanto a classe `BreadthFirstSearch` calcula a distância individual de um porto a outro, a classe `App` também utiliza esses resultados para calcular a distância total a ser percorrida. Para isso, ela conta com dois métodos: o `travelToLastPort`, que calcula a menor distância para ir do primeiro porto até o último porto fazendo parada em cada um dos portos do mapa, e o `returnToFirstPort`, que calcula a menor distância para retornar do último porto ao primeiro, sem fazer paradas. Ao final, apenas é feita a soma das duas distâncias calculadas, cada uma por um dos métodos anteriores, e essa será a saída do programa (lembrando que, pelo enunciado do problema, a distância é diretamente proporcional à quantidade de combustível gasto). Tais métodos efetuam seus cálculos de distância conforme os seguintes pseudocódigos:

```
se não houver portos no mapa, disparar erro
se houver apenas um porto no mapa, retornar que a distância é 0
assumir o porto de origem (origem) como o 1
assumir o porto de destino (destino) como o 2
enquanto destino <= quantidade de portos:
      se existe um caminho origem→destino (verificar usando BFS):
            distância_total = distância_total + distância origem→destino
            origem = destino
      destino = destino + 1
último porto visitado* = origem
agora, se distância_total = 0, lançar erro (não há portos alcançáveis)
```

Pseudocódigo que descreve a implementação do método `travelToLastPort` no programa (OBS.: * → variável global)

```
se não houver portos no mapa, disparar erro
se houver apenas um porto no mapa, retornar que a distância é 0
assumir o porto inicial (início) como o 1
```

```
porto final (fim) = último porto visitado
construir, com BFS, o caminho fim→início
se não houver um caminho fim→início:
      não pode retornar (a distância será 'null'), então lançar erro
caso contrário:
      a distância será a distância fim→início conforme BFS
```

Pseudocódigo que descreve a implementação do método `returnToFirstPort` no programa

---

3. RESULTADOS PARA OS CASOS DE TESTE FORNECIDOS:

3.1. `case0.map`: 248 un. de combustível;

3.2. `case1.map`: 598 un. de combustível;

3.3. `case2.map`: 1.112 un. de combustível;

3.4. `case3.map`: 2.210 un. de combustível;

3.5. `case4.map`: 3.510 un. de combustível;

3.6. `case5.map`: 11.518 un. de combustível;

---

4. CONCLUSÕES:

A partir da construção do programa que implementa a solução para o problema proposto para este trabalho, percebe-se que a implementação do TAD de grafos apresentada por Sedgewick e Wayne, apesar de armazenar apenas valores inteiros sequenciais, pode ser utilizada para armazenar grafos cujos vértices são qualquer objeto. Para isso, utiliza-se um dicionário para mapear o código sequencial de um vértice do grafo para seu real valor. Portanto, os valores dos vértices são armazenados em dicionários indexados pelos códigos sequenciais de cada objeto, e a estrutura de grafo armazena apenas esses códigos.

Além disso, percebe-se a aplicação prática de problemas desse gênero, pois o problema proposto para este trabalho nada mais é que o problema do caixeiro viajante, em que, dados dois pontos e um conjunto de caminhos para chegar de um ponto ao outro, o programa deve analisar os caminhos e escolher avançar por aquele de menor distância. Esse problema é muito comum hoje em dia com os sistemas de *global positioning system* (GPS), que, dentre uma série de caminhos para chegar de um local a outro, deve escolher aquele que acarretará no menor tempo de deslocamento ao usuário.