Bacharelado em Sistemas de Informação – Noturno

Instituto de Ciências Matemática e de Computação

Algoritmos e Estruturas de Dados II

Relatório do Exercício 4

Prof.: Leonardo Tortoro Pereira

Rodrigo Teixeira Ribeiro da Silva – 11816164
Rafael Jun Morita – 10845040

26/06/2022

Índice

1. Introdução

O exercício em questão consiste na realização da travessia em profundidade (Depth First Traversal) de um digrafo, onde será utilizada a biblioteca de grafos desenvolvida na disciplina de Algoritmos e Estruturas de Dados II do curso de Sistemas de Informação da USP São Carlos.

Cada um dos vértices do digrafo representará uma quest em um jogo, e as arestas a ordem das quests, ou seja, a ordem com que elas ocorrerão, sendo que cada quest possuirá um id único, um nome e uma descrição.

O número de vértices, o nome e descrição das quests, o número de arestas, a origem e o destino das arestas, bem como o vértice inicial da travessia serão recebidos através da entrada padrão.

Ao final da execução da solução desenvolvida, a saída padrão deve imprimir as questões que farão parte da travessia, desenvolvidas a partir das variáveis de entrada na ordem correta.

2. Desenvolvimento

O desenvolvimento contará com a explicação modular de 3 das classes desenvolvidas e alteradas da biblioteca, que possuem boa parte dos processos relevantes para a travessia em profundidade do grafo.

Travessia em profundidade do grafo de quests

Nome: QuestGraphDFT.java

Função: Essa classe possui a função main do código, e é responsável por instanciar todas as classes necessárias para a resolução do problema, receber as variáveis através da entrada padrão e, de fato, resolver o problema - imprimindo o caminho da travessia através da saída padrão.

Etapas:

1. Inicialização do scanner

```
// Inicialização do Scanner que será utilizado para ler a
// entrada padrão.
Scanner stdin = new Scanner(System.in);
```

2. Instanciação do controlador

```
// Instanciação do controlador do gráfo.
var graphController = new GraphController();
```

3. Definição do número de vértices

```
// Solicita para o usuário a entrada de um inteiro representando
// o número de vértices no gráfo e as atribui para a variável
// numberOfVertices.
System.out.println("Number of vertices: ");
int numberOfVertices = Integer.parseInt(stdin.nextLine());
```

4. Instanciação das quests com nome e descrição

```
// Solicita a entrada do nome e descrição de todas as quests,
// e as adiciona como instância na lista de vértices do
// controlador do gráfo.
System.out.println("Enter all of the " + numberOfVertices + " vertices: ");
for(int i = 0 ; i < numberOfVertices ; i++){

    String questName = stdin.nextLine();
    String questDescription = stdin.nextLine();

    graphController.vertices.add(new Quest(questName, questDescription));
}
```

5. Instanciação da matriz do dígrafo

```
// Após a adição das quests na lista do controlador do gráfo,
// essa variável é utilizada para inicializar a matriz do
// dígrafo.
graphController.g = new DigraphMatrix(graphController.vertices);
```

6. Instanciação da travessia em profundidade

```
// Inicialização da estratégia de travessia do gráfo como uma
// instância da busca em produndidade.
graphController.traversalStrategy = new DepthFirstTraversal(graphController.g);
```

7. Definição do número de arestas

```
// Solicita para o usuário a entrada de um inteiro representando
// o número de arestas no gráfo e as atribui para a variável
// numberOfEdges.
System.out.println("Number of edges: ");
int numberOfEdges = Integer.parseInt(stdin.nextLine());
```

8. Instanciação das arestas com origens e destino das arestas

```
// Solicita a entrada e o destino de todas as arestas,
// e as adiciona no gráfo através do método addEdge da
// matriz do dígrafo.
System.out.println("Enter all of the " + numberOfEdges + " edges: ");
for(int i = 0 ; i < numberOfEdges ; i++){

    int source = stdin.nextInt();
    int destination = stdin.nextInt();

    graphController.g.addEdge(
        graphController.g.getVertices().get(source),
        graphController.g.getVertices().get(destination));
}
```

9. Definição do vértice inicial da travessia

```
// Solicita para o usuário a entrada de um inteiro representando
// o ID do vértice inicial da buscao e atribui para a variável
// initialVertex.
System.out.println("Initial vertex: ");
int initialVertex = stdin.nextInt();
```

10. Realização da travessia em profundidade

```
// Realiza a busca em profundidade apartir do vértice inicial.
graphController.traversalStrategy.traverseGraph(
    graphController.g.getVertices().get(initialVertex)
);
```

11. Fechamento do scanner

```
// Fechamento da instância do scanner
stdin.close();
```

Controlador do grafo

Classe: GraphController.java

Função: Instanciar todas as classes necessárias para a utilização de um grafo com estrutura de dados, vértices e estratégias de travessia genéricas.

Etapas:

1. Propriedades da classe

```java
// As propriedades dessa classe são os elementos fundamentais
// para a utilização do gráfo, definindo as peças para seleção
// da estrutura de dados do gráfo, da estratégia de travessia, e
// também a lista de vértices que é utilizada para instânciar a
// estrutura de dados.
public AbstractGraph g;
public TraversalStrategyInterface traversalStrategy;
public final List<Vertex> vertices;
```

2. Construtor

```java
// Construtor da classe GrphController, que chama a função
// createVertexList
public GraphController() {
    vertices = createVertexList();
}
```

3. Inicialização da lista de vértices

```java
// A função está retornando uma ArrayList para a
// Lista de vertices que é um atributo da classe
private static List<Vertex> createVertexList()
{
    return new ArrayList<>();
}
```

Travessia em profundidade

Classe: DepthFirstTraversal.java

Função: Realiza a travessia do grafo através da profundidade.

Etapas:

1. Inicialização da variável que recebe a instância do StringBuilder.

```
// Definição da propriedade que salvará todo os vértices
// para posteriormente fazer a impressão
StringBuilder traversedPath;
```

2. Construtor

```
// Instancia o StringBuilder, além de receber o grafo
// e "enviá-lo para cima" através do supper, para que
// o grafo receba a operação da busca em produndiade
public DepthFirstTraversal(AbstractGraph g)
{
    super(g);
    traversedPath = new StringBuilder();
}
```

3. Método herdado que chama uma travessia genérica

```
@Override
// Recebe o vértice inicial da busca, chama a chamada recursiva
// de busca em produndidade e imprime o caminho realizado
public void traverseGraph(Vertex source)
{
    depthFirstTraversalRecursion(source);
    printPath();
}
```

4. Realização da travessia em profundidade

```
private void depthFirstTraversalRecursion(Vertex source)
{
    // Marca o vértice recebido por parâmetro como visitado
    markVertexAsVisited(getGraph().getVertices().indexOf(source));

    // Adiciona uma mudança de linha ao StringBuilder
    traversedPath.append(source).append('\n');

    // Adiciona ao caminho o vértice visitado
    addToPath(source);

    // Tentar pegar o primero vértice conectado, caso ele exista, atribui
    // a variável adjacentVertex um valor diferente de null
    var adjacentVertex = getGraph().getFirstConnectedVertex(source);

    while(adjacentVertex != null)
    {
        // Seleciona o indice do próximo vértice adjacente
        int adjacentVertexIndex = getGraph().getVertices().indexOf(adjacentVertex);

        // Caso o próximo vértice ainda não tenha sido visitado, chama a recursão
        if(!hasVertexBeenVisited(adjacentVertexIndex)) {
            depthFirstTraversalRecursion(adjacentVertex);
        }

        // Caso o próximo vértice já tenha sido visitado, visita um vértice vizinho
        adjacentVertex = getGraph().getNextConnectedVertex(source, adjacentVertex);
    }
}
```

3. Resultados

Após a construção da solução para realizar a travessia em profundidade do grafo em questão, foi possível perceber as vantagens latentes da utilização de códigos mais genéricos, facilitando a extensão do código e também a utilização de instâncias de elementos diferentes para realização de uma mesma tarefa, porém combinando módulos diferentes.

Além disso, a utilização da linguagem Java, que possui como característica marcante a orientação a objetos, faz perceber a facilidade da resolução desse paradigma da programação, principalmente quando comparados os códigos em C e Java.

Além de outras facilidades, como a não necessidade de utilização de passagem de parâmetros por referência de ponteiro e também a não necessidade de liberação da memória por conta do Garbage Collector da VM do Java.

4. Referências

- Repositório da biblioteca de grafos da matéria
- Repositório do código da entrega