Instituto de Ciências Matemáticas e de Computação
Departamento de Ciências de Computação
SCC0503 - Algoritmos e Estruturas de Dados II

Relatório Exercício 04

Alunos:
Eduardo Maciel de Matos, 12563821
João Pedro Ribeiro da Silva, 12563727

Professor: Leonardo Tórtoro Pereira

Junho
2022

Conteúdo

1 Introdução

A proposta do exercício resolvido era simular como funcionava as missões de um jogo RPG, abstraindo essa lógica utilizando um Dígrafo como estrutura de dados e sobre ela fazendo uma busca por profundidade.

Com a implementação conseguimos simular o exercício proposto de dada uma missão, ir até a última missão de cada missão conectada nela, como se fossemos desbloqueando as missões dentro do jogo.

2 Desenvolvimento

Para desenvolver o projeto reutilizamos grande parte dos códigos fornecidos em sala pelo nosso professor. Para a criação do grafo decidimos utilizar a lista de adjacência invés da matriz de adjacência para este escopo do projeto, por conta desse grafo em específico possuir muitos itens, mas poucas conexões entre eles, se utilizassemos uma matriz teríamos muitos espaços vazios ocupando um espaço desnecessário para a alocação da matriz.

Como grande parte da abstração na criação dos grafos já nos foi dada pelo professor, tivemos que implementar a classe relacionada a missão, junto com isso implementar para que ela se conecte com o restante do código, e também implementar a busca por profundidade sobre a lista de adjacência.

Para processar os dados utilizamos a classe *Scanner* do Java, e assim que são processados, adicionamos as missões em um *ArrayList* para iterarmos sobre

```
Scanner in = new Scanner(System.in);

int numVertices = Integer.parseInt(in.nextLine());
ArrayList<Vertex> quests = new ArrayList();

for (int i = 0; i < numVertices; i++) {
  String nome = in.nextLine();
  String descricao = in.nextLine();
  quests.add(new Quest(i, nome, descricao));
}
```

Figura 1: Processamento das informações das missões dadas

Para processar os dados sobre destino e origem do gráfico, utilizamos as classes *AbstractGraph* e *Vertex* dadas em aula, para popular o grafo.

```
Scanner in = new Scanner(System.in);

int numVertices = Integer.parseInt(in.nextLine());
ArrayList<Vertex> quests = new ArrayList();

for (int i = 0; i < numVertices; i++) {
  String nome = in.nextLine();
  String descricao = in.nextLine();
  quests.add(new Quest(i, nome, descricao));
}
```

Figura 2: Processamento das informações das informações sobre o grafo

Após isso pegamos qual o nó devemos começar a busca por profundidade, e utilizamos as classes *TraversalStrategy* e *DepthFirstTraversal* que são as responsáveis pela busca/travessia. Passando para o método de travessia por qual vértice a busca deve começar.

```
int indexStart = in.nextInt();

TraversalStrategy traversalStrategy = new DepthFirstTraversal(graph);
traversalStrategy.traverseGraph(quests.get(indexStart));
```

Figura 3: Processamento da busca por profundidade

Após o método de *traverseGraph* ser chamado começa a lógica implementada por nós. Para isso, iremos explicar e como funciona por debaixo dos panos a classe *DepthFirstTraversal*. O método *traverseGraph* recebe o vértice pelo qual deve começar a travessia, pegamos seu índice, adicionamos na lista de vértices atravessados dentro da classe *TraversalStrategy*, marcamos esse vértice como já visitado, para não ser repetido na busca, marcamos a distância para ele como 0 por se o primeiro, dizemos que ele não tem nenhum predecessor, e por fim chamamos o método *visitVertex*, que implementa uma recursão e após ele mostramos o caminho que nossa busca por profundidade fez.

```
public void traverseGraph(Vertex source) {
    int sourceIndex = getGraph().getVertices().indexOf(source);
    addToPath(source);
    markVertexAsVisited(sourceIndex);
    setDistanceToVertex(sourceIndex, 0);
    setPredecessorVertexIndex(sourceIndex, -1);

    visitVertex(source);

    printPath();
}
```

Figura 4: Método *traverseGraph*

O método *visitVertex* recebe de primeiro momento o vértice inicial da busca, e é o responsável por percorrer os vértices. De primeiro momento pegamos o índice atual, e o primeiro vértice conectado nele. Após isso entramos em um *while* para percorrer todos os filhos do vértice atual, nesse *while* pegamos o índice do vértice adjacente conectado, vemos se já foi visitado, e caso não alteramos para ele ser visitado, fazemos isso chamando o método *updateTraversalInfoForVertex* passando o vértice adjacente, e o vértice atual, após isso o método visitVertex é chamado novamente para fazer uma recursão, para visitar todos os filhos do vértice adjacente. Após a recursão, damos update na variavel *adjacentVertex* para fazer a busca nos próximos vértices conectados ao vértice passado no método *traverseGraph*

```
private void visitVertex(Vertex currentVertex) {
    int currentVertexIndex = getGraph().getVertices().indexOf(currentVertex);

    Vertex adjacentVertex = getGraph().getFirstConnectedVertex(currentVertex);
    while (adjacentVertex ≠ null) {
        int adjacentVertexIndex = getGraph().getVertices().indexOf(adjacentVertex);
        if (!hasVertexBeenVisited(adjacentVertexIndex)) {
            updateTraversalInfoForVertex(adjacentVertexIndex, currentVertexIndex);
            visitVertex(adjacentVertex);
        }
        adjacentVertex = getGraph().getNextConnectedVertex(currentVertex, adjacentVertex);
    }
}
```

Figura 5: Método *visitVertex*

Já o método *updateTraversalInfoForVertex* atualiza as informações necessárias de cada travessia, ele recebe o vértice que sera visitado, e o que ja foi visitado. Para atualizar as informações, pegamos os índices dos respectivos vértices, calculamos suas distâncias, adicionamos eles ao caminho percorrido, marcamos como visitados, e colocamos a distância calculada nas informações de cada um deles.

```
private void updateTraversalInfoForVertex(int newVertexIndex, int previousVertexIndex) {
    var newVertex = getGraph().getVertices().get(newVertexIndex);
    var oldVertex = getGraph().getVertices().get(previousVertexIndex);
    float newDistance = getGraph().getDistance(oldVertex, newVertex);
    float distance = getDistanceToVertex(previousVertexIndex) + newDistance;
    addToPath(newVertex);
    markVertexAsVisited(newVertexIndex);
    setDistanceToVertex(newVertexIndex, distance);
    setPredecessorVertexIndex(newVertexIndex, previousVertexIndex);
    setSuccessorVertexIndex(previousVertexIndex, newVertexIndex);
}
```

Figura 6: Método *updateTraversalInfoForVertex*

3 Resultados

Tivemos um resultado que apenas diferente na ordem dos vértices visitados pelos casos de teste, por conta da implementação, mas que ainda assim está correta, conversando com o professor ele nos orientou a manter como estava.

Caso de teste	Output	Come

1	Quest{
	ID= '8'
	name= 'Elder Care'
	description= 'Help the village Elder'
	}
	Quest{
	ID= '9'
	name= 'Second Intentions'
	description= 'Make the Elder tell you about the Demon Lord'
	}
	Quest{
	ID= '5'
	name= 'Hell's Door'
	description= 'You must find the entrance to the Demon Lord's Castle'
	}
	Quest{
	ID= '6'
	name= 'Cleaning the House'
	description= 'Kill all the guardians of the Demon Lord'
	}
	Quest{
	ID= '7'
	name= 'Showdown'
	description= 'Face the last fierce battle and kill the Demon Lord'
	}
		Este
		Neste

Caso de teste	Output	Come

2	Quest{
	ID= '0'
	name= 'Slime Killer'
	description= 'You must kill 10 slimes'
	}
	Quest{
	ID= '11'
	name= 'Slime's Revenge'
	description= 'The slime Queen wants revenge for her minions'
	}
	Quest{
	ID= '1'
	name= 'The Woodcutter'
	description= 'Find and talk to Greyson, the woodcutter'
	}
	Quest{
	ID= '2'
	name= 'A New Axe'
	description= 'Search for the legendary axe for Greyson'
	}
	Quest{
	ID= '3'
	name= 'The Blacksmith'
	description= 'Find and talk to Clint, the blacksmith'
	}
	Quest{
	ID= '4'
	name= 'Hammer of Doom'
	description= 'Search for the legendary hammer for Clint'
	}
		Este
		Neste

Caso de teste	Output	Come

3	Quest{
	ID= '10'
	name= 'Hello, World!'
	description= 'Get to the starter town and talk to the villagers'
	}
	Quest{
	ID= '0'
	name= 'Slime Killer'
	description= 'You must kill 10 slimes'
	}
	Quest{
	ID= '11'
	name= 'Slime's Revenge'
	description= 'The slime Queen wants revenge for her minions'
	}
	Quest{
	ID= '1'
	name= 'The Woodcutter'
	description= 'Find and talk to Greyson, the woodcutter'
	}
	Quest{
	ID= '2'
	name= 'A New Axe'
	description= 'Search for the legendary axe for Greyson'
	}
	Quest{
	ID= '14'
	name= 'The Sidequester'
	description= 'Face the final challenge and complete the last sidequest'
	}
	Quest{
	ID= '3'
	name= 'The Blacksmith'
	description= 'Find and talk to Clint, the blacksmith'
	}
	Quest{
	ID= '4'
	name= 'Hammer of Doom'
	description= 'Search for the legendary hammer for Clint'
	}
	Quest{
	ID= '12'
	name= 'Duel Time!'
	description= 'Win the card minigame tournament'
	}
	Quest{
	ID= '13'
	name= 'Master of Duel'
		Este
		Nele
		Neste

Caso de teste	Output	Come

4	Quest{
	ID= '10'
	name= 'Hello, World!'
	description= 'Get to the starter town and talk to the villagers'
	}
	Quest{
	ID= '0'
	name= 'Slime Killer'
	description= 'You must kill 10 slimes'
	}
	Quest{
	ID= '11'
	name= 'Slime's Revenge'
	description= 'The slime Queen wants revenge for her minions'
	}
	Quest{
	ID= '1'
	name= 'The Woodcutter'
	description= 'Find and talk to Greyson, the woodcutter'
	}
	Quest{
	ID= '2'
	name= 'A New Axe'
	description= 'Search for the legendary axe for Greyson'
	}
	Quest{
	ID= '3'
	name= 'The Blacksmith'
	description= 'Find and talk to Clint, the blacksmith'
	}
	Quest{
	ID= '4'
	name= 'Hammer of Doom'
	description= 'Search for the legendary hammer for Clint'
	}
	Quest{
	ID= '12'
	name= 'Duel Time!'
	description= 'Win the card minigame tournament'
	}
	Quest{
	ID= '8'
	name= 'Elder Care'
	description= 'Help the village Elder'
	}
	Quest{
	ID= '9'
	name= 'Second Intentions'
	description= 'Make the Elder talk one last time'
		Este
		Nele
		Neste

Caso de teste	Output	Come

Referências

Anexo

No anexo, pode colocar qualquer coisa extra que achar interessante: detalhes mais completos de resultados, alguma coisa diferente/inovadora que fez, etc.