Relatório

Projeto Programação Avançada

Variante B 18/19

Docente: Patrícia Macedo

Aluno: Gabriel Ambrósio

Nº: 160221013

Introdução

Este projeto foi desenvolvido no âmbito da UC de programação avançada, que tinha como requisitos criar uma aplicação didática sobre grafos não-orientados. A aplicação tinha de ser capaz de importar um grafo de um ficheiro, visualizá-lo, manipulá-lo, obter informações sobre o mesmo e exportar o grafo resultante, através da linguagem de programação Java com interação do utilizador através de interface visual desenvolvida em JavaFX.

A aplicação foi desenvolvida tendo em conta os princípios da programação orientada por objetos aplicando os conhecimentos aprendidos nas aulas teóricas com o uso do tipo abstrato de dados grafo (TAD Graph) assim como os padrões de software que vão ser descritos mais á frente.

Para criar esta aplicação utilizei como base a TAD Graph disponibilizada pelos docentes e que foi adaptada para ajudar com a implementação necessária. Foram desenvolvidas classes especificas para as arestas e os vértices com base nas disponibilizadas bem como para a classe do grafo.

Como aspeto final de apresentação, toda a persistência de dados foi realizada através da criação de um ficheiro de texto e gson, que guarda todos os vértices, arestas e também a informação contida nelas como a sua descrição e custo.

Classes

Neste projeto usei como base as classes disponibilizadas pelos docentes, nomeadamente as classes genéricas do grafo, aresta e vértice. Na implementação do grafo adicionei alguns métodos que facilitavam a manipulação do mesmo, como na classe MyGraph:

- isolatedVertices()

```
250     /**
251      * Metodo que retorna o número de vertices isolados
252      * Vertices que não contêm edges
253      * @return Integer - numero de vertices isolados
254      */
255     public int isolatedVertices(){
256         HashSet<Vertex<V>> aux = copyIterable(vertices());
257 
258         Iterator<Vertex<V>> it = aux.iterator();
259 
260         while (it.hasNext()) {
261             Vertex<V> v = it.next();
262 
263             for (Edge<E, V> ed : edges()) {
264                 if (ed.vertices()[0] == v || ed.vertices()[1] == v) {
265                     it.remove();
266                     break;
267                 }
268             }
269         }
270         return aux.size();
271     }
```

Este método calcula o número de vértices isolados, é utilizado mais a frente quando é preciso mostrar ao utilizador esta informação na UI.

- getEdgesOfVertex() e vertexDegree()

```
273     /**
274      * Metodo que retorna um Set com os edges de um determinado vertice
275      * @param vertex - Vertex
276      * @return list - set de edges
277      */
278     public HashSet<Edge<E, V>> getEdgesOfVertex(Vertex<V> vertex){
279         HashSet<Edge<E, V>> list = new HashSet<>();
280         Vertex<V> vl = checkVertex(vertex);
281         for(Edge<E, V> e : edges()){
282             if(e.vertices()[0] == vl || e.vertices()[1] == vl){
283                 list.add(e);
284             }
285         }
286         return list;
287     }
```

```java
    /**
     * Metodo que usa o metodo getEdgesOfVertex() para calcular o grau de um vertice
     * @param vertex - vertice a calcular
     * @return Integer - grau, numero de edges do vertice
     */
    public int vertexDegree(Vertex<V> vertex){
        return getEdgesOfVertex(vertex).size();
    }
```

Que são usados juntamente para conseguir uma lista ou o número de arestas de um vértice, é usado também para mostrar na UI o grau do vértice selecionado pelo utilizador.

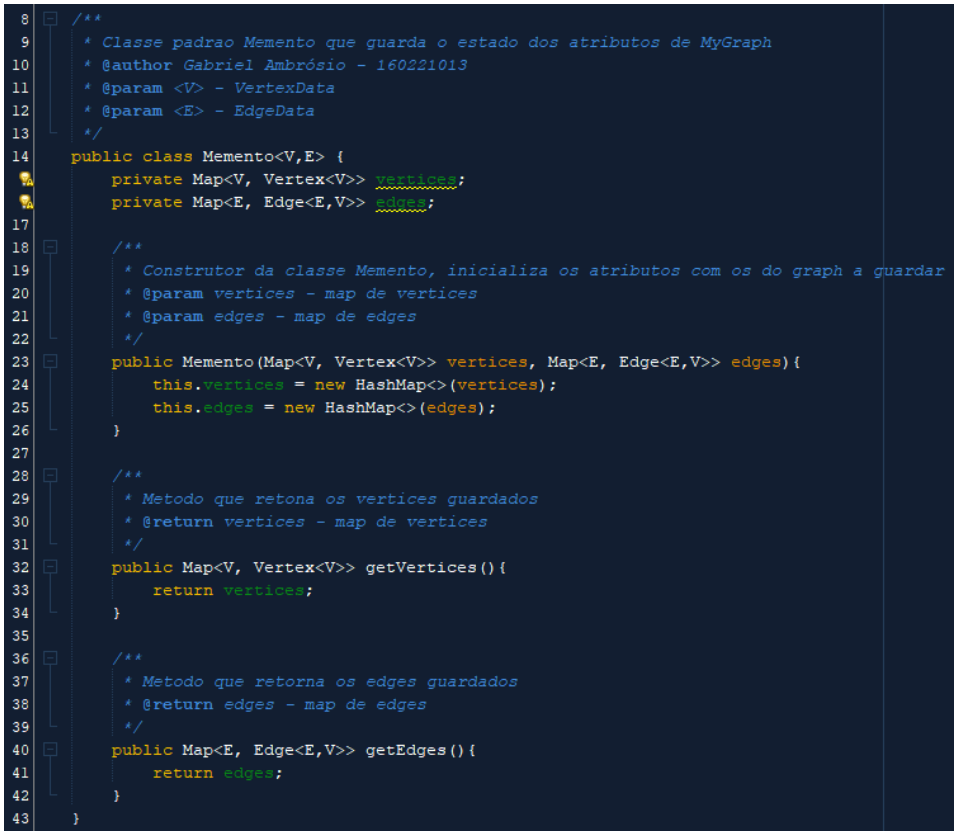
Por fim decidi que para resolver o problema Retroceder (Undo) usar o Padrão Memento, criando os métodos createMemento() e setMemento().

```java
    //Memento

    /**
     * Metodo que cria um novo memento
     * @return
     */
    public Memento createMemento(){
        return new Memento(vertices, edges);
    }

    /**
     * Metodo que restaura o estado dos atributos
     * @param memento - memento
     */
    public void setMemento(Memento memento){
        vertices = memento.getVertices();
        edges = memento.getEdges();
        setChanged();
        notifyObservers(this);
    }
}
```

Estes métodos vão assegurar que a informação contida nesta classe é guardada externamente, e que fica sempre acessível caso seja necessário retroceder algumas alterações.

Para a utilização deste padrão foram criadas mais duas classes:

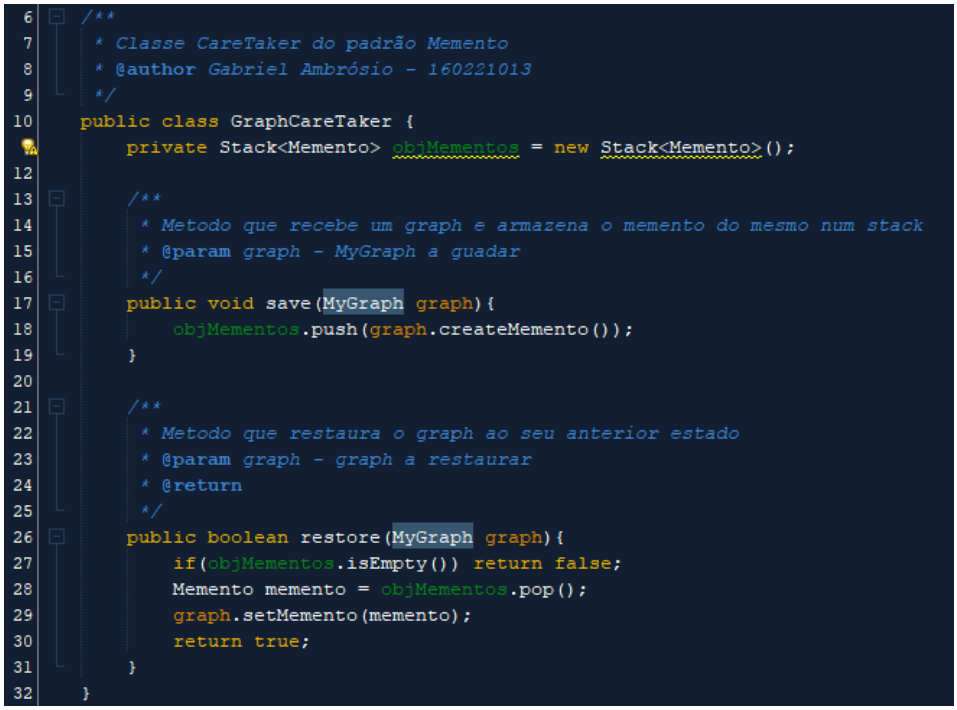
Memento



```java
/**
 * Classe padrao Memento que guarda o estado dos atributos de MyGraph
 * @author Gabriel Ambrósio - 160221013
 * @param <V> - VertexData
 * @param <E> - EdgeData
 */
public class Memento<V,E> {
    private Map<V, Vertex<V>> vertices;
    private Map<E, Edge<E,V>> edges;

    /**
     * Construtor da classe Memento, inicializa os atributos com os do graph a guardar
     * @param vertices - map de vertices
     * @param edges - map de edges
     */
    public Memento(Map<V, Vertex<V>> vertices, Map<E, Edge<E,V>> edges){
        this.vertices = new HashMap<>(vertices);
        this.edges = new HashMap<>(edges);
    }

    /**
     * Metodo que retona os vertices guardados
     * @return vertices - map de vertices
     */
    public Map<V, Vertex<V>> getVertices(){
        return vertices;
    }

    /**
     * Metodo que retorna os edges guardados
     * @return edges - map de edges
     */
    public Map<E, Edge<E,V>> getEdges(){
        return edges;
    }
}
```

Guarda a mesma informação que MyGraph, e a disponibiliza através dos getters.

CareTaker



```java
/**
 * Classe CareTaker do padrão Memento
 * @author Gabriel Ambrósio - 160221013
 */
public class GraphCareTaker {
    private Stack<Memento> objMementos = new Stack<Memento>();

    /**
     * Metodo que recebe um graph e armazena o memento do mesmo num stack
     * @param graph - MyGraph a guadar
     */
    public void save(MyGraph graph){
        objMementos.push(graph.createMemento());
    }

    /**
     * Metodo que restaura o graph ao seu anterior estado
     * @param graph - graph a restaurar
     * @return
     */
    public boolean restore(MyGraph graph){
        if(objMementos.isEmpty()) return false;
        Memento memento = objMementos.pop();
        graph.setMemento(memento);
        return true;
    }
}
```

Que usa um Stack para assegurar que é possível armazenar mais do que um memento e que estes são adicionados e removidos da maneira apropriada.

Para o primeiro requisito do enunciado do projeto II.1 Tipos de Dados criei duas classes que guardam a informação pretendida, uma descrição (em formato String) para o vértice, e uma descrição (String) e custo (um inteiro) para a aresta. Estas classes são bastante simples na sua implementação, mas vai facilitar mais a frente a manipulação do conteúdo de um grafo.

VertexData

```
3  /**
4   * Classe usada para guardar a informação a guardar nos Vertices
5   * @author Gabriel Ambrósio - 160221013
6   */
7  public class VertexData {
8
9      private String str;
10
11     /**
12      * Constutor da classe VertexData, inicializa os atributos
13      * @param str - string
14      */
15     public VertexData(String str){
16         this.str = checkString(str);
17     }
```

Classe que guarda a informação de um vértice

EdgeData

```
3  /**
4   * Classe usada para guardar a informação a guardar nos Edges
5   * @author Gabriel Ambrósio - 160221013
6   */
7  public class EdgeData {
8      private String str;
9      private int cost;
10
11     /**
12      *Construtor da classe EdgeData, inicializa os atributos
13      * @param str - nome
14      * @param cost - custo
15      */
16     public EdgeData(String str, int cost){
17         this.str = checkString(str);
18         this.cost = checkSInteger(cost);
19     }
```

Classe que guarda a informação de uma aresta

No enunciado é também pedido que seja possível, importar e exportar grafos, de duas maneiras, em formato de texto, e em formato Json, onde surgiu o primeiro e único problema na conclusão da aplicação.

Para resolver este ponto recorri novamente a um padrão lecionado nas aulas, o DAO (Data Access Object). Comecei por criar uma interface para ajudar à implementação das duas maneiras:

```
7   /**
8    * Interface para o padrao DAO
9    * @author Gabriel Ambrósio - 160221013
10   */
     public interface DAOInterface {

         public void saveGraph(MyGraph<VertexData, EdgeData> graph, String filename);
         public MyGraph loadGraph(String filename);
15   }
16
```

Depois implementei a classe DAOFile, que importa e exporta os grafos por ficheiros de texto, esta classe foi também fácil na sua conclusão pois foram disponibilizadas várias maneiras da sua implementação. Esta vai ser a única maneira de importar/exportar grafos, porque depois de várias tentativas de implementar em ficheiro Json, não consegui. Deixo aqui na mesma a implementação a que cheguei:

```
16  /**
17   * Classe DAOJson usada para guardar e ler informação em formato gson
18   * @author Gabriel Ambrósio - 160221013
19   */
20  public class DAOOneJson implements DAOInterface{
21
22      private final String path = System.getProperty("user.dir") + "\\src\\Input\\";
23      private final String filename = "graphs.json";
24
25      private JsonData select() {
26          try {
27              BufferedReader br = new BufferedReader(new FileReader(path + filename));
28              Gson gson = new GsonBuilder().create();
29
30              JsonData gh = gson.fromJson(br, new TypeToken<JsonData>() { }.getType());
31              //MyGraph<VertexData, EdgeData> gsonGraph = gson.fromJson(br, new TypeToken<MyGraph<VertexData, EdgeData>>() { }.getType());
32
33              return gh;
34
35          } catch (IOException ex) {
36
37          }
38          return null;
39      }
```

```java
    /**
     * Metodo que guarda um MyGraph num ficheiro gson predefinido
     * @param graph - MyGraph a guardar
     * @param filename - filename
     */
    @Override
    public void saveGraph(MyGraph<VertexData, EdgeData> graph, String filename) {
        FileWriter writer = null;
        try {
            Gson gson = new GsonBuilder().create();
            JsonData data = new JsonData(graph);
            writer = new FileWriter(path + this.filename);
            gson.toJson(data, writer);
            writer.flush();
            writer.close();
        } catch (IOException ex) {

        }
    }
```

Numa das tentativas de por esta classe a funcionar, criei outra que armazenava todos os dados a guardar:

```java
/**
 * Classe que guarda a informaçãoa a ser usada na classe DAOOneJson
 * Criei esta classe porque não estava a conseguir guardar o objeto principal MyGraph
 * @author Gabriel Ambrósio - 160221013
 */
public class JsonData {
    private List<Vertex<VertexData>> vertices;
    private List<Edge<EdgeData, VertexData>> edges;
    private MyGraph<VertexData, EdgeData> graph;

    /**
     * Constutor da classe JsonData, recebe um MyGraph, para guardar os seus atributos (vertives e edges)
     * @param graph - MyGraph, graph a guardar
     */
    public JsonData(MyGraph<VertexData, EdgeData> graph){
        vertices = new ArrayList<>();
        edges = new ArrayList<>();
        this.graph = graph;
        copyArrays();
    }
```

Passando ao próximo ponto do projeto II.3 Visualização e manipulação, pedia para mostrar o grafo graficamente, mostrando também algumas informações sobre o mesmo, pedia para ser possível remover ou adicionar arestas e vértices na UI e que a todo o instante estes pontos anteriores se mantivessem atualizados. Criei então uma classe que ajuda á manipulação no grafo chamada GraphHandling e uma classe que mostrasse o grafo numa UI chamada MainMenu. Para resolver o problema da atualização de informação na UI, decidi usar um padrão lecionado chamado MVC, aqui distribuímos as responsabilidades da lógica de negócio que decidi atribuir à classe MyGraph (Model), da lógica de apresentação, que escolhi atribuir à classe MainMenu (View) e por fim da lógica de controlo, que por exclusão de partes fica atribuída à classe GraphHandling (Controller). Já falamos do Model (MyGraph) acima, por isso vamos falar das outras duas:

MainMenu

```
92      public MainMenu(MyGraph<VertexData, EdgeData> gp){
93          super();
94          this.gp = gp;
95          initialize();
96          scene = new Scene(start(), 1200, 800);
97          plot();
98      }
```

Esta classe é bastante grande, pois é responsável pela criação e inicialização de todos os componentes gráficos a mostrar, desde a janela principal a abrir até aos botões e texto a mostar.

Resumindo, utilizei um BorderPane para inserir todos os componentes, no centro deste colocar o grafo, usando o projeto para representar uma TAD Gaph disponibilizado pelos docentes, na direita inserir todos os componentes que tratam da manipulação do grafo (remover/adicionar), no topo a informação pedida (numero de vértices/arestas, etc), na esquerda as funcionalidades pedidas nos pontos II.6 e II.7 que vamos falar mais á frente.

Tentei explicar ao pormenor o que os métodos desta classe fazem no javadoc.

GraphHandling

```
23  public class GraphHandling extends MyGraph{
24
25      private MyGraph<VertexData, EdgeData> graph;
26      private MainMenu menu;
27      private GraphCareTaker caretaker;
28      private Logger logger;
29
30      /**
31       * Construtor que inicializa os seus atributos com os do MyGraph recebido, também sincroniza o Model (MyGraph) com a View (MainMenu)
32       * e por fim garante que o estado do graph é guardado usando o Memento
33       *
34       * @param graph - Model
35       * @param menu - View
36       */
37      public GraphHandling(MyGraph<VertexData, EdgeData> graph, MainMenu menu){
38          super();
39          this.graph = graph;
40          this.menu = menu;
41          this.menu.setTriggers(this);
42          this.graph.addObserver(this.menu);
43
44          logger = Logger.getInstance();
45          caretaker = new GraphCareTaker();
46          caretaker.save(this.graph);
47      }
```

Para completar o padrão MVC foi criada esta classe, que usa os métodos do Model, implementa também alguns métodos que fornecem as funcionalidades pedidas em II.6 e II.7 e por fim permite usar o padrão Memento com o método restoreGraph().

Para obter a informação sobre os vértices, pedidos no ponto II.5, criei alguns métodos, como o vertexDegree() e getEdgesOfVertex() mostrados anteriormente, para criar o gráfico de barras recorri ao JavaFX Charts, mais concretamente a um BarChart.

```
378     private BarChart<String, Number> createBarChart(){
379         xAxis = new CategoryAxis();
380         yAxis = new NumberAxis();
381         bc = new BarChart<>(xAxis, yAxis);
382         bc.setTitle("Graph Chart");
383         xAxis.setLabel("Vertex");
384         yAxis.setLabel("Degree");
385 
386         XYChart.Series series1 = new XYChart.Series();
387         //por cada vertice adiciona uma nova barra com o seu nome e o seu grau
            for(Vertex<VertexData> v : vertices){
389             series1.getData().add(new XYChart.Data(v.element().getString(), gp.vertexDegree(v)));
390         }
391 
392         bc.getData().add(series1);
393         bc.setPrefHeight(800);
394         bc.setPrefWidth(1000);
395 
396         bc.setLegendVisible(false);
397 
398         return bc;
399     }
```

Para o ponto II.6 e II.7, recorri a slides usados em aulas passadas, que falam dos tipos de pesquisa em profundida e largura, e também do algoritmo de pesquisa chamado Dijkstra, criei então os métodos necessários na classe GraphHandling. Estes métodos são usados depois na classe MainMenu, para poder mostrar ao utilizador a informação obtida nos mesmos.

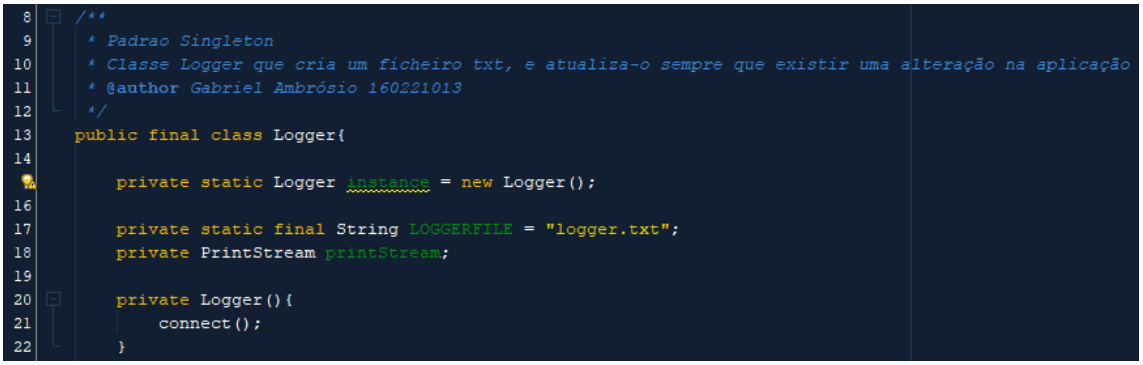
A criação da tabela com a informação do Dijkstra, foi uma a tarefa mais difícil, quando tentei popular a tabela, pois nunca tinha criado uma, recorri informação disponibilizada em documentos da oracle, que mostravam várias maneiras de implementar tabelas. Cheguei a esta implementação:

```java
    private TableView createTableView(){
        table = new TableView();
        table.setEditable(false);

        table.setMinHeight(300);
        table.setMinWidth(510);
        //primeira coluna guarada o nome do vertice escolhido como start
        vertexCol = new TableColumn("Start");
        vertexCol.setCellValueFactory(new PropertyValueFactory<>("start"));
        vertexCol.setMaxWidth(100);
        vertexCol.setMinWidth(100);
        vertexCol.setEditable(false);

        //segunda coluna guarda o caminho
        pathCol = new TableColumn("Path");
        pathCol.setMinWidth(210);
        pathCol.setMaxWidth(210);
        pathCol.setCellValueFactory(new PropertyValueFactory<>("path"));

        //terceira coluna guarada o nome do vertice escolhido como end
        endCol = new TableColumn("End");
        endCol.setMinWidth(100);
        endCol.setMaxWidth(100);
        endCol.setCellValueFactory(new PropertyValueFactory<>("end"));

        //ultima guarda o custo do path de start a end
        costCol = new TableColumn("Cost");
        costCol.setMinWidth(100);
        costCol.setMaxWidth(100);
        costCol.setCellValueFactory(new PropertyValueFactory<>("cost"));

        table.setItems(listofList);

        table.getColumns().addAll(vertexCol, pathCol, endCol, costCol);
        return table;
    }
```

Foi necessário, entretanto criar uma classe que contivesse a informação a mostrar na tabela.

```java
/**
 * Classe usada para guardar a informação necessária a mostrar na table view (chart) no MainMenu
 * @author Gabriel Ambrósio - 160221013
 */
public class DijkstraData {

    private String start, path, end, cost;

    /**
     *Construtor da classe DijkstraData, recebe os valores e inicializa os atributos
     * @param s - Vertex (start), é usado apenas o seu nome, através do metodo getString() de VertexData
     * @param p - Path, string que contêm o caminho completo de s (start) a e (end)
     * @param e - Vertex (end), é usado apenas o seu nome, através do metodo getString() de VertexData
     * @param c - Cost, inteiro que armazena o custo do path recebido (é armazenado como string)
     */
    public DijkstraData(VertexData s, String p, VertexData e, int c){
        start = s.getString();
        path = p;
        end = e.getString();
        cost = ""+c;
    }
```

Por fim o último ponto em que era necessária a implementação de uma classe concreta II.8 Logging, usei o padrão Singleton. A implementação desta classe foi também disponibilizada nas aulas.



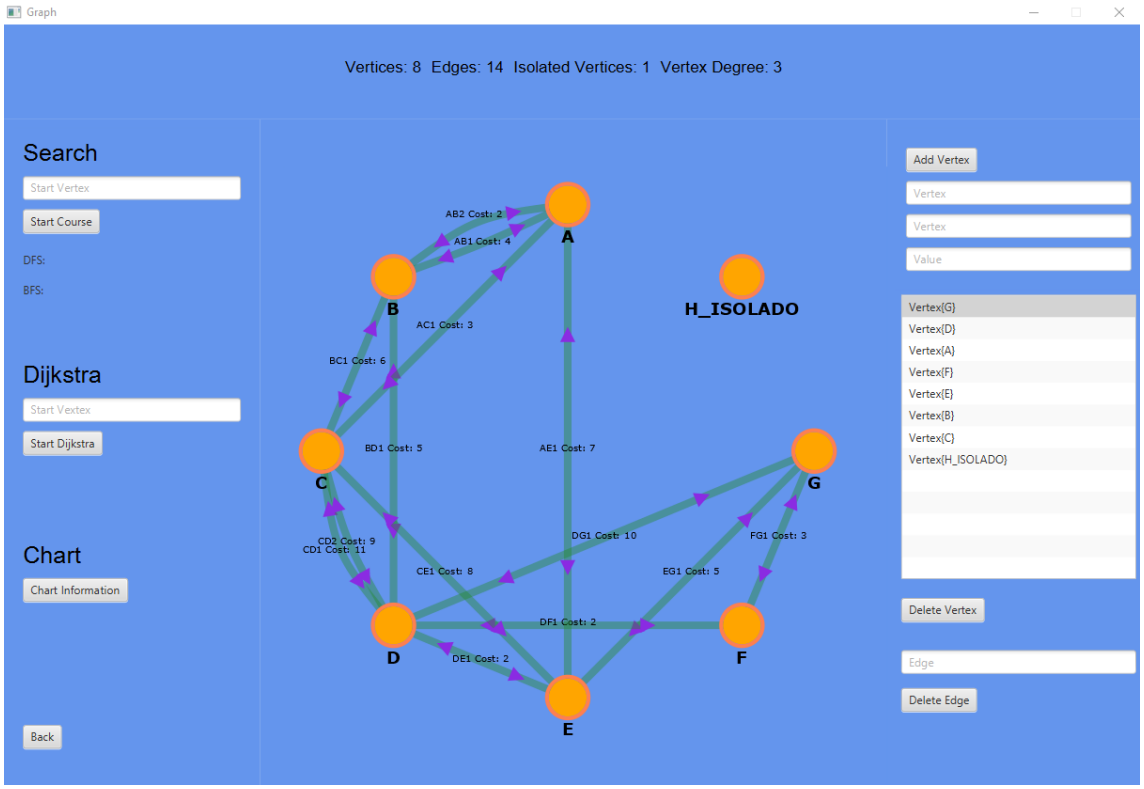
```java
/**
 * Padrao Singleton
 * Classe Logger que cria um ficheiro txt, e atualiza-o sempre que existir uma alteração na aplicação
 * @author Gabriel Ambrósio 160221013
 */
public final class Logger{

    private static Logger instance = new Logger();

    private static final String LOGGERFILE = "logger.txt";
    private PrintStream printStream;

    private Logger(){
        connect();
    }
```

Esta classe responsabiliza-se por escrever para um ficheiro, todas as ações feitas pelo utilizador, por isso é que existe um atributo desta classe em quase todas as outras (sendo estes sempre o mesmo pois é uma classe Singleton).

Funcionalidades em execução

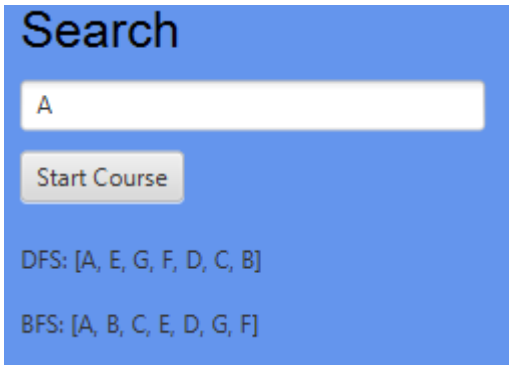
Na execução da aplicação, é mostrado um alerta a pedir o nome no ficheiro de texto onde está o grafo, este ficheiro tem de estar na pasta Input dentro do projeto, está la o ficheiro disponibilizado pelos docentes chamado “grafo”, logo é só escrever grafo nesse campo.

Página principal da aplicação



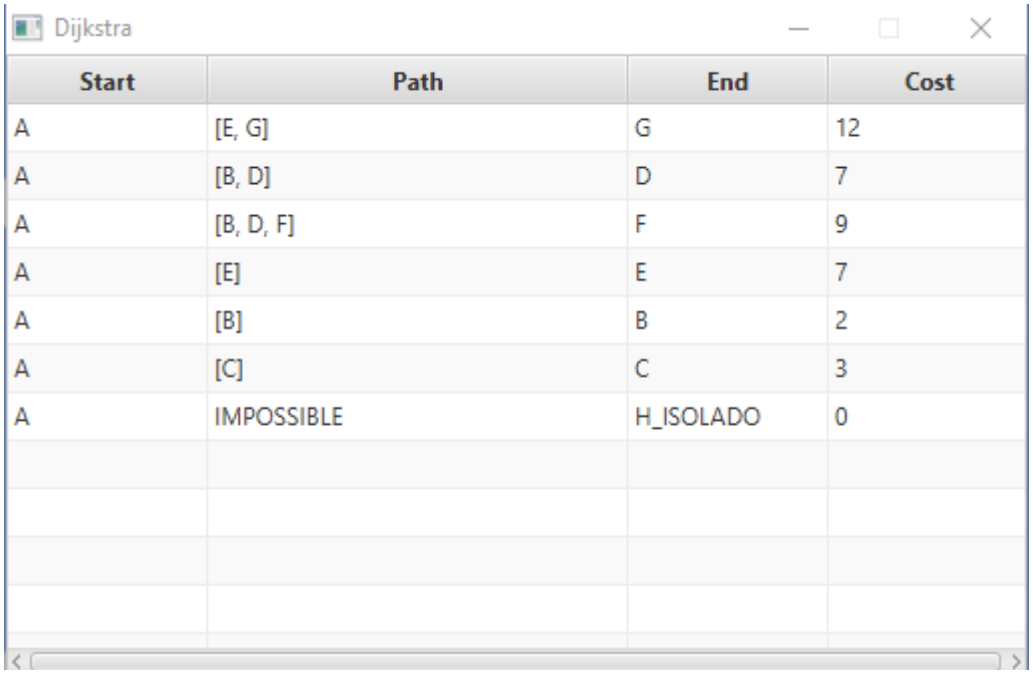
A partir desta janela é possível utilizar todas as funcionalidades pedidas.

O **Search** na esquerda, inserimos um nome de um vertice (ex: A):



Dá-nos os resultados de pesquisa em profundidade e largura.

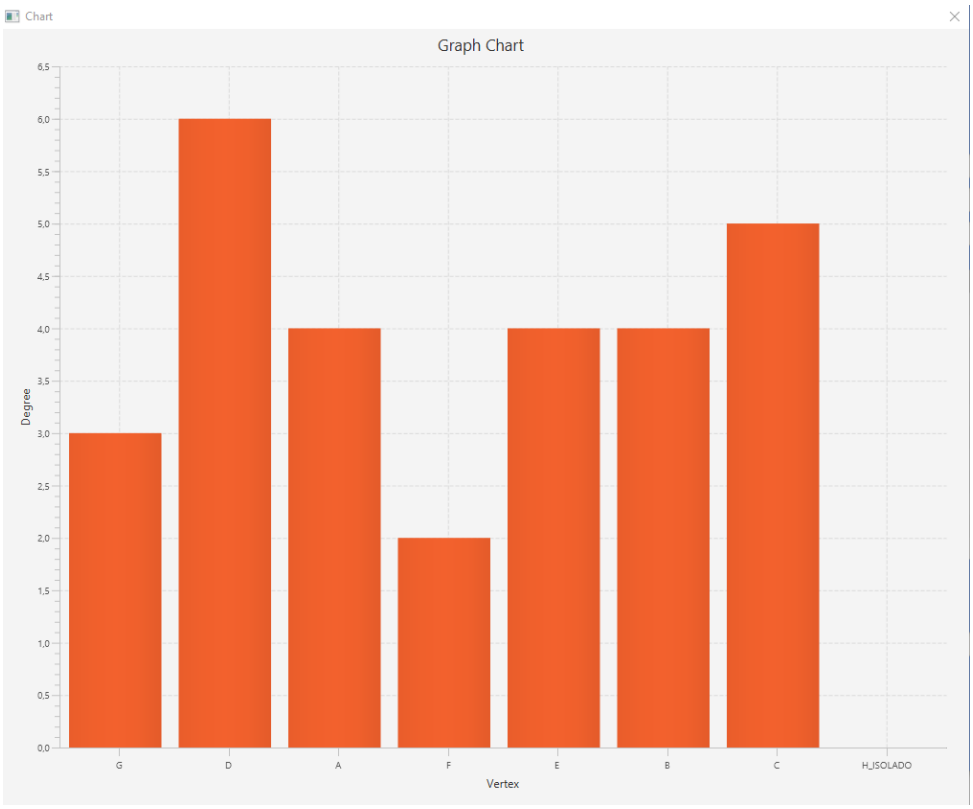
O **Dijkstra**, inserimos um nome de um vértice (ex: A):



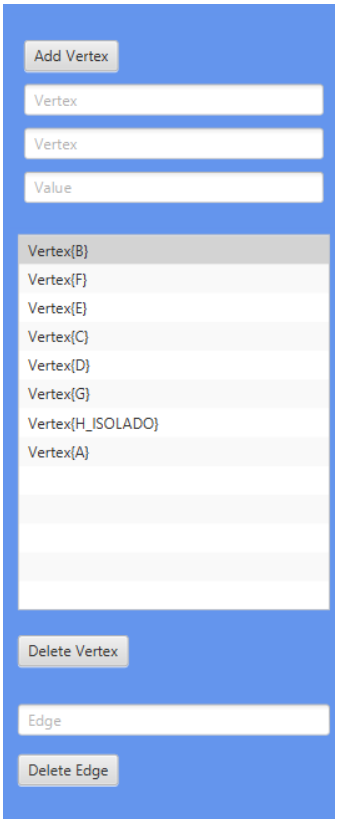
Start	Path	End	Cost
A	[E, G]	G	12
A	[B, D]	D	7
A	[B, D, F]	F	9
A	[E]	E	7
A	[B]	B	2
A	[C]	C	3
A	IMPOSSIBLE	H_ISOLADO	0

E é criada uma nova janela com uma tabela, coma informação toda pretendida, caminhos de A para todos os outros e também o custo do mesmo.

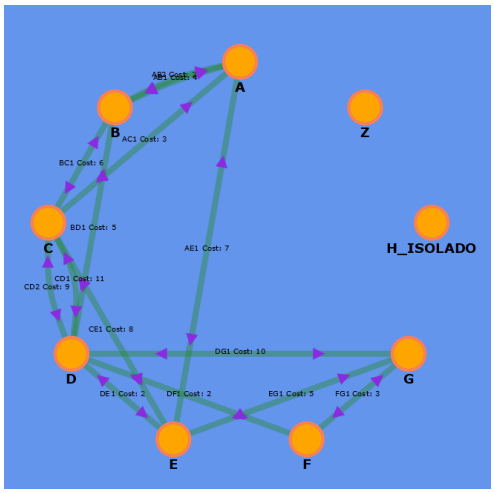
O botão **Chart Information**, abre uma outra janela com o gráfico de barras com os graus de todos os vértices:



Por fim na direita é possível fazer várias coisas…

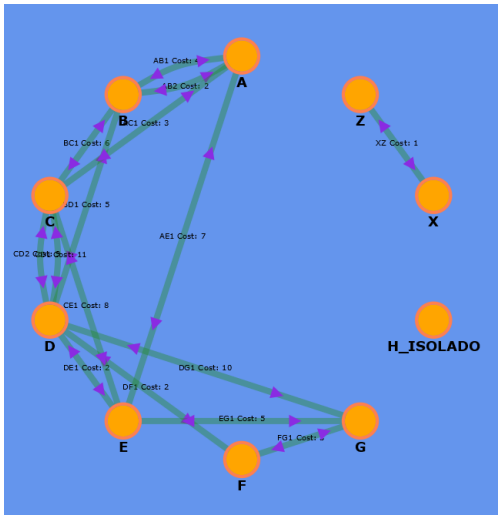


Primeiro podemos **adicionar apenas um vértice**, colocando o nome que lhe pretendemos dar no primeiro ou segundo textfield, deixando os outros vazios, assim criamos apenas um vértice isolado.



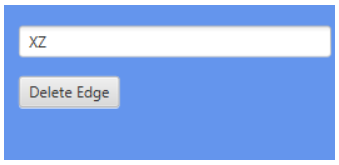
Podemos adicionar uma aresta entre vértices já existentes, preenchendo dois primeiros campos com os nomes dos vértices, e o terceiro com o custo.

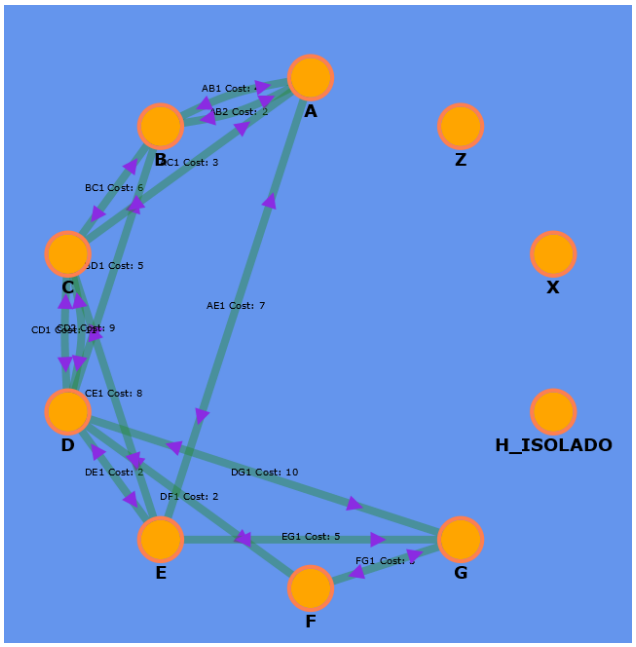
Ou por exemplo criar um vértice novo e conectá-lo com um já existente, ou até criar dois novos e ligá-los (inserindo nos dois primeiros campos, os nomes dos vértices a adicionar e o custo da aresta).



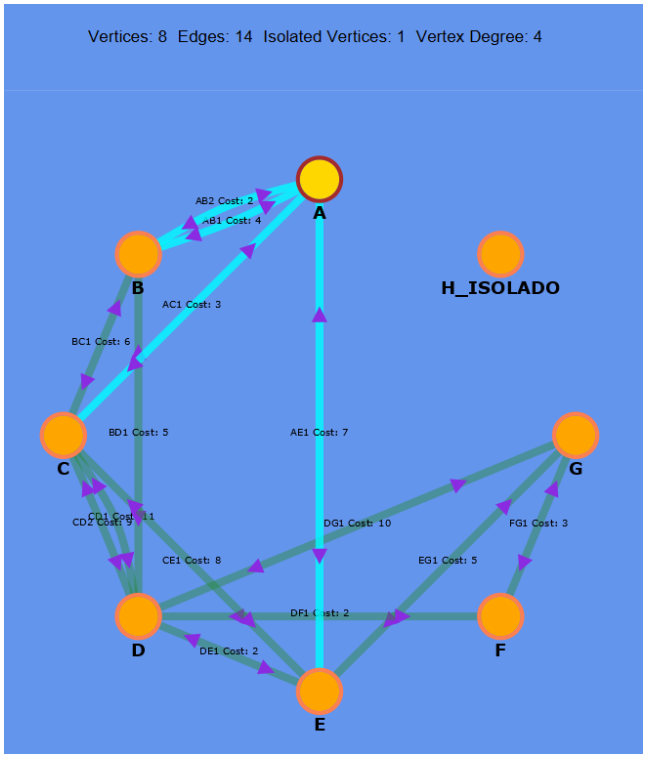
Temos também um botão que permite **remover um vértice** através da list view. Escolhemos o vértice pretendido e carregamos no botão, este remove também as arestas se estas existirem.

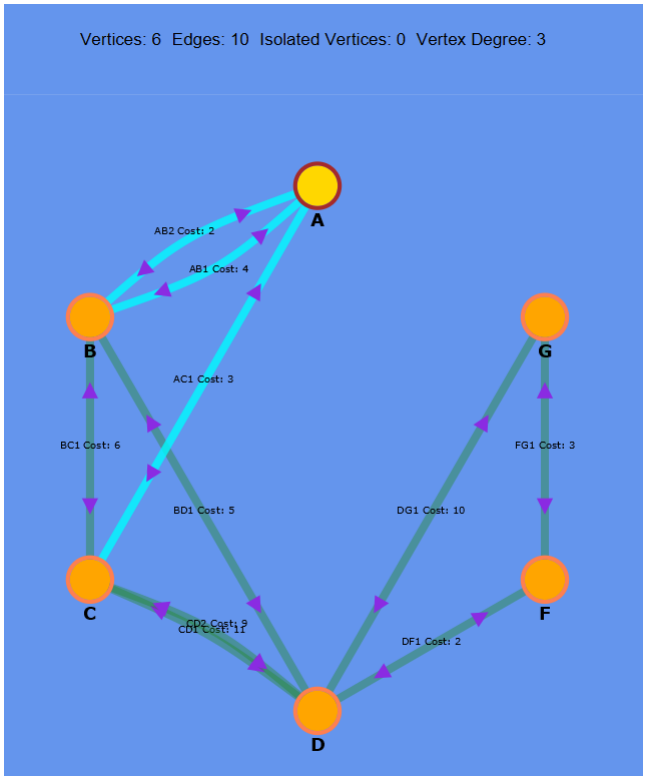
E podemos também apenas **apagar uma aresta** entre vértices usando o campo no canto inferior direito, colocamos o nome da aresta (Ex: XZ)





Por fim no topo podemos ver as informações pedidas a serem **atualizadas**.





Ao fechar a aplicação, esta mostra de novo um alerta, a perguntar se pretende guardar o grafo manipulado, podemos então aceitar, onde escrevemos o nome do ficheiro onde pretendemos guardar, rejeitar onde o grafo manipulado não é guardado, ou cancela que nos redireciona para a aplicação.

Como foi referido e explicado ao longo da explicação das classes foram utilizados diferentes padrões de software como suporte às boas praticas de programação orientada por objetos. Resumindo então o que já foi dito, foram usados 4 padrões, começando pelo Memento usado para a funcionalidade Retroceder, DAO usado para importar e exportar informação requerida pela aplicação, MVC, para distribuir as responsabilidades referidas acima e por fim o Singleton usado na classe Logger.

Refactoring

Depois de concluído o projeto, fiz uma revisão a todo o código escrito, e deparei-me com alguns bad smells comuns e fáceis de corrigir, nomeadamente, o mais detetado foi o Duplicated Code, em que existiam alguns excertos de código idênticos, usados em métodos diferentes, para corrigir usei a técnica Extract Method, em que criei um novo método que continha esse mesmo código, e depois substituí o código com uma chamada do novo método.

Deparei-me também com bad smeels mais complicados, como a Large Class, em referencia á classe MainMenu, que é usada para criar o layout gráfico, é uma classe bastante extensa, poderia usar o método Extract Class, e criar uma outras classes que se dedicavam á criação de uma peça do layout, mas achei melhor ser apenas uma classe a ter essa responsabilidade. Por fim as classes nominadas de Data, podem ser classes que não justifiquem a sua existência, por não fazerem muito alem de guardarem alguma informação necessária, Lazy Class/Data Class, podendo usar os métodos de refactoring Inline Class, mover essa informação para outra classe, criei estas classes por ser recomendado logo de início pelos docentes, para ajudar ao funcionamento da aplicação.

Duplicated Code	Foi o bad smell mais detetado, principalmente na classe MainMenu.	Extract Method
Large Class	Ocorreu uma vez.	Extract Class
Lazy Class/Data Class	Ocorreu 2 vezes, nas classes VertexData e EdgeData	Inline Class