Relatório do Exercício Individual

Carolina Cunha (a80142)

Braga, junho de 2020

Universidade do Minho
Mestrado Integrado em Engenharia Informática
Sistemas de Representação de Conhecimento e Raciocínio
(2º Semestre/2019-2020)

Resumo

Este projeto teve como objetivo o aperfeiçoamento da utilização da linguagem de programação em lógica *Prolog*, no âmbito de métodos de resolução de problemas e no desenvolvimento de algoritmos de pesquisa, recorrendo à utilização de diferentes estratégias de pesquisa (não-informada e informada).

Conteúdo

Lista de Figuras

1 Introdução

O principal objetivo deste projeto consistiu no aperfeiçoamento da utilização da linguagem de programação em lógica, *Prolog*, no âmbito de métodos de resolução de problemas e no desenvolvimento de algoritmos de pesquisa.

O tema em estudo é o sistema de transportes do concelho de Oeiras, através de um repositório de dados abertos, contendo informações relativas às paragens de autocarros de Oeiras, englobando várias características, tais como a sua localização, as carreiras que as utilizam, as respetivas operadoras, entre outras.

Assim, espera-se o desenvolvimento de um sistema, que permita importar os dados relativos às paragens de autocarro, e representá-los numa base de conhecimento. Posteriormente, será desenvolvido um sistema de recomendação de transporte público para o caso de estudo.

2 Preliminares

Previamente à realização deste trabalho, foram estudados dois princípios imprescindíveis ao entendimento dos métodos de resolução de problemas e de procura.

Assim sendo, sabe-se que um problema pode ser resolvido através da pesquisa de um caminho entre o estado inicial e um estado objetivo, pelo que a sua formulação envolve decidir que ações e estados considerar tendo em conta um objetivo. [2]

Um problema pode ser definido formalmente por cinco componentes:

1. Representação do Estado;
2. Estado Inicial (Atual);
3. Estado Objetivo;
4. Operadores (Nome, Pré-Condições, Efeitos, Custo);
5. Custo da Solução.

Existem quatro tipos de problemas [2]:

- **Estado Único**: Caracteriza-se por um ambiente determinístico, totalmente observável. O agente sabe exatamente o estado em que estará, a solução é uma sequência;
- **Múltiplos Estados**: Ambiente determinístico, não acessível. O agente não sabe onde está. A solução é uma sequência;
- **Contingência**: O ambiente é não determinístico e/ou parcialmente acessível. Perceções fornecem novas informações sobre o estado atual;
- **Exploração**: O espaço de estados é desconhecido.

Uma solução para um problema é o caminho do estado inicial para um estado objetivo. A qualidade da solução é medida pela função do custo do caminho e pode ser [2]:

- **Satisfatória**, se é uma qualquer solução;
- **Semi-ótima**, se é aquela que tem aproximadamente o menor custo entre todas as soluções;
- **Ótima**, se é aquela que tem o menor custo entre todas as soluções.

A ideia básica de uma procura passa pela expansão do estado inicial, criando uma lista de todos os possíveis sucessores. Após criação de uma lista de estados não expandidos, é necessário escolher um estado da lista de estados não expandidos para expandir. Este processo deve ser repetido até atingir o estado objetivo, tentando expandir o menor número de estados possível [2].

Uma **estratégia de pesquisa** é definida escolhendo a ordem de expansão do nó.

Existem dois tipos de estratégias de pesquisa: **Não Informadas** - usam apenas as informações disponíveis na definição do problema - **Informadas** - dá-se ao algoritmo dicas sobre a adequação de diferentes estados [2].

No que diz respeito à **pesquisa não-informada (cega)**, tem-se:

- Primeiro em largura (BFS);
- Primeiro em Profundidade (DFS);
- Custo Uniforme;
- Pesquisa Iterativa;
- Pesquisa Bidirecional.

Por outro lado, para a **pesquisa informada (heurísticas)**, tem-se:

- Pesquisa Gulosa;
- Algoritmo A*.

3 Breve Descrição do Enunciado e Análise de Resultados

De seguida, será discutido todo o processo de resolução do projeto proposto pelos docentes da unidade curricular em questão.

Para o efeito, considere-se que o panorama poderá ser caracterizado por conhecimento, por exemplo, dado na forma que se segue:

- paragem: $Gid, Latitude, Longitude, Estado, Tipo, Publicidade, Operadora, Carreira, Codigo, Rua, Freguesia \rightarrow \{V, F\}$
- adjacenciasParagens: $Paragem1, Paragem2, Carreira \rightarrow \{V, F\}$
- adjacenciasNRepetidas: $Paragem1, Paragem2 \rightarrow \{V, F\}$

A elaboração deste caso prático deverá permitir:

1. Calcular um trajeto entre dois pontos;
2. Selecionar apenas algumas das operadores de transporte para um determinado percurso;
3. Excluir uma ou mais operadoras de transporte para o percurso;
4. Identificar quais as paragens com o maior número de carreiras num determinado percurso;
5. Escolher o menor percurso, isto é, com menor número de paragens;
6. Escolher o percurso mais rápido, usando o critério da distância;
7. Escolher o percurso que passe apenas por abrigos com publicidade;
8. Escolher o percurso que passe apenas por paragens abrigadas;
9. Escolher um ou mais pontos intermédios por onde o percurso deverá passar.

3.1 Base de Conhecimento

De modo a inserir o conhecimento na base, foi implementado um *parser*, capaz de converter ficheiros *excel* em ficheiros com formato *.csv*, e, posteriormente, capaz de interpretar os dados presentes nos respetivos ficheiros, permitindo o povoamento da base de conhecimento. Neste *parser*, será tido em conta o número de páginas *excel*, representantes de cada carreira.

A conversão dos ficheiros *excel* em ficheiros com formato *.csv* foi conseguida com recurso à linguagem de programação *Python*, usufruindo da simplicidade da biblioteca *Pandas* [3], que permitiu uma fácil abordagem para a resolução deste problema inicial.

```python
import pandas
import xlrd

def excelToCSV():

    workbook = xlrd.open_workbook("lista_adjacencias_paragens.xlsx")
    sheets = workbook.sheet_names()
    name = "carreiras/c_"
    for sheet in range(0, len(sheets)):
        sheet_name = sheets[sheet]
        csv_file = name + sheet_name + ".csv"
        file = pandas.read_excel("lista_adjacencias_paragens.xlsx", sheet_name)
        file.to_csv(csv_file, index = None, header=True)

excelToCSV()
```

Parser em Python.

De seguida, a validação de cada linha e posterior escrita na base de conhecimento foram realizadas recorrendo à linguagem de programação *JAVA*.

3.1.1 Paragens de Autocarros

Para a verificação e validação das linhas resultantes da conversão do ficheiro relativo às paragens dos autocarros, foram tidas as seguintes considerações:

- São eliminadas todas as linhas com um número incorreto de elementos (considerando onze elementos como o valor acertado de constituintes de uma paragem);
- São corrigidas todas as linhas com erros tipográficos, como o excesso de espaços e vírgulas.

Assim sendo, foi criada uma classe, *ParagensOeiras*, em que, para cada linha lida do ficheiro *.csv*, vai validar todos os parâmetros da mesma e, no caso de se tratar de uma linha válida, vai criar um objeto ParagemOeiras, permitindo a sua escrita na base de conhecimento.

```java
public class paragensOeiras {
    private int gid;
    private double latitude;
    private double longitude;
    private String estado;
    private String tipo;
    private String publicidade;
    private String operadora;
    private List<Integer> carreira;
    private int codigo;
    private String rua;
    private String freguesia;
```

Componentes de uma Paragem.

3.1.2 Listas de Adjacências

No que diz respeito à validação das listas de adjacência, foram também corrigidos todos os erros tipográficos, não tendo sido eliminada qualquer linha.

Para a realização deste exercício, foram consideradas adjacências bilaterais dado que, regra geral, as carreiras de transportes públicos realizam os mesmos trajetos nos dois sentidos. Para além disso, a unilateralidade iria impedir, na maior parte dos casos, viagens de regresso para a mesma carreira, pelo que foi então tomada esta decisão.

Deste modo, foram criadas duas classes, *adjacenciasParagens*, onde serão guardados os valores relativos à paragem de origem, a sua paragem adjacente e respetiva carreira, e *adjacenciasNParagens*. A necessidade de criação desta última classe deveu-se à existência de ciclos infinitos aquando da implementação do método de procura A^*. De forma idêntica ao processo anterior, para cada linha lida do ficheiro, vai ser criado um objeto adjacenciasParagens, permitindo a sua escrita na base de conhecimento. No que diz respeito à adjacenciasNParagens, o método é semelhante, no entanto, vai ser verificada a pré-existência dessa adjacência na lista de adjacências, sendo esta verificação bilateral. Caso não se encontre nesta lista, o par de paragens adjacentes será escrito na base de conhecimento.

```java
public class adjacenciasParagens {
    private int paragem1;
    private int paragem2;
    private int carreira;
```

Componentes de uma Adjacência.

```java
public class adjacenciasNRepetidas {
    private int paragem1;
    private int paragem2;

```

Componentes de uma Adjacência sem Carreiras.

Após realizado o *parsing* destes ficheiros, foi obtida uma base de conhecimento com 3332 linhas, em que 801 linhas correspondem às paragens de autocarros dos transportes de Oeiras, 1516 às listas de adjacências de cada paragem e as restantes 1015 às adjacências de paragens não repetidas.

4 Resposta a Questões Colocadas

Para a concretização das questões mais pequenas, foi utilizado um método de procura não informada primeiro em largura, cujo algoritmo base pode ser encontrado em *Notas Introdutórias ao Prolog e Procura IA* [1]. Este método recorre a uma estratégia em que todos os nós de menor profundidade são expandidos primeiro. Por este motivo, é uma pesquisa muito sistemática. No entanto, é normalmente muito demorada e ocupa muito espaço. [2]

Para esta pesquisa, conhecem-se as seguintes propriedades:

- **Completa**: É uma pesquisa completa, caso o fator de ramificação seja finito;
- **Tempo**: $O(b^d)$;
- **Espaço**: Guarda cada nó em memória $O(b^d)$;
- **Otimal**: Sim, se o custo de cada passo for 1.

4.1 Calcular um trajeto entre dois pontos

O cálculo entre duas paragens exigiu o conhecimento das paragens de origem e de destino.

Desta forma, e mantendo uma lista de espera de nodos a visitar, bem como de uma lista de visitados, contendo todos os nodos visitados, foram calculados todos os nodos adjacentes, verificando que estes não se encontravam já nas duas listas mencionadas. Este processo repetiu-se para todos os nodos até ser alcançada a paragem destino.

Para a concretização desta questão, foram declarados os seguintes predicados auxiliares:

- adjacente: Verifica se duas paragens são adjacentes entre si, em ambas as direções;
- apagaNaoCaminho: Após realizadas todas as iterações sobre os nodos adjacentes, vão estar presentes na lista dos nodos visitados, nodos não pertencentes ao caminho entre a paragem de origem e de destino. Por este motivo, é necessário um predicado que elimine da lista todos os nodos não pertencentes ao caminho final;
- nao: Extensão do meta-predicado não.

Foram, ainda, utilizados três predicados pré-definidos, **sort**, permitindo ordenar e lista de adjacentes e eliminar os nodos repetidos, **append**, permitindo adicionar à lista de espera todos os nodos adjacentes calculados, não eliminando os nodos previamente existentes em espera e **member**, que verifica se um elemento pertence a uma lista.

4.2 Selecionar apenas algumas das operadoras de transporte para um determinado percurso

A elaboração da resposta a esta questão foi idêntica à anteriormente especificada. No entanto, houve a necessidade de acrescentar uma nova condição ao predicado de procura, uma vez que se pretende definir as operadoras de transporte que se pretende utilizar.

De forma a cumprir este objetivo, foram implementados três predicados auxiliares:

- getOperadoraGid: Permite saber qual a operada de uma determinada paragem;
- sublista: Predicado utilizado para verificar se uma lista está presente numa outra lista. Este predicado permitirá verificar se a paragem em questão é da operadora desejada;
- elemOperadora: Predicado onde serão aplicados os dois predicados anteriores.
- nao: Extensão do meta-predicado não.

Assim, apenas farão parte do percurso final, todas as paragens de uma(s) determinada(s) operadora(s).

4.3 Excluir um ou mais operadores de transporte para o percurso

Uma vez que esta questão é inversa à prévia, a resposta será idêntica. No entanto, só farão parte do percurso final, todas as paragens que não sejam de determinada(s) operadora(s) especificada(s).

4.4 Escolher o percurso que passe apenas por abrigos com publicidade

De igual modo às anteriores, a resposta à questão colocada apenas necessitará de uma nova condição de paragem.

Dado que são pretendidas apenas paragens com publicidade, é necessário o predicado *gid_Publicidade*, verificando se a paragem em estudo se enquadra nos requisitos pedidos.

4.5 Escolher o percurso que passe apenas por paragens abrigadas

De forma a garantir a cláusula apresentada, foi implementado o predicado *gid_Abrigo* que, dado um **gid** de uma paragem, devolve um valor verdadeiro no caso de se tratar de uma paragem abrigada (o tipo de abrigo deve ser "fechado dos lados"ou "aberto dos lados").

4.6 Identificar quais as paragens com o maior número de carreiras num determinado percurso

Para a realização da resposta a esta questão, o processo teve início com o cálculo de um percurso entre duas paragens, através de uma pesquisa primeiro em largura. De seguida, foram implementados alguns predicados auxiliares, permitindo a obtenção do número de carreiras por paragem, bem como a sua ordenação. Assim, tem-se:

- gidNumeroCarreiras: Devolve o número de carreiras de uma paragem;
- gidNumeroCarreirasCaminho: Para uma dada lista de paragens, calcula quantas carreiras possui cada paragem e guarda os resultados numa lista de duplos *(Paragem, Número de Carreiras)*;
- addOrdenaDecrescente: Dada uma lista de duplos, ordena a mesma por ordem decrescente;
- ordenar: Predicado de ordenação de uma lista. Aplica o predicado anterior;
- ordenaCarreiras: Dado um percurso, calcula o número de carreiras por paragem e ordena-as por ordem decrescente.

4.7 Escolher um ou mais pontos intermédios por onde o percurso deverá passar

A realização desta questão consistiu no cálculo da procura em largura de um nodo inicial para o primeiro nodo intermédio, prosseguindo recursivamente para a procura em largura entre nodos intermédios até que, por fim, é realizada a procura de um percurso entre o último nodo intermédio até ao nodo final. Desta forma, garante-se que o trajeto traçado inclui as paragens intermédias definidas como argumento do predicado. Assim,

- bfsCaminhoParagens: Irá calcular recursivamente a procura em largura para todos os nodos intermédios, até ao nodo final.
- append: Predicado pré-definido que permite a concatenação de duas listas.

4.8 Escolher o percurso mais rápido (usando critério da distância)

Uma vez que esta questão requer a utilização de um critério de distância para calcular o percurso mais rápido entre duas paragens, foi necessário recorrer a um método de procura informada, tendo sido optada a estratégia de pesquisa A^*.

Este algoritmo combina a pesquisa gulosa com a uniforme, minimizando a soma do caminho já efetuado com o mínimo previsto que falta até a solução. Trata-se de um algoritmo ótimo e completo, cuja complexidade no tempo é exponencial, e a sua complexidade no espaço caracteriza-se pelo armazenamento de todos os nodos em memória [2].

O algoritmo da pesquisa A^* rege-se pela função

$$f(n) = g(n) + h(n)$$

onde:

– f(n) = custo estimado da solução mais barata que passa pelo nodo n;

– g(n) = custo total para chegar ao estado n (custo do percurso);

– h(n) = custo estimado para chegar ao objetivo (não deve sobrestimar o custo para chegar à solução).

Assim sendo, é indispensável o cálculo da **estima**, que será caracterizada pela distância Euclidiana entre duas paragens.

Foram implementados dois métodos distintos para o cálculo das distâncias Euclidianas, no entanto, foi apenas utilizado o segundo método.

Distância Eucledіana considerando a Curvatura da Terra

```
distancia(Latitude1, Longitude1, Latitude2, Longitude2, Distancia) :-
  P is 0.017453292519943295,
  A is (0.5 - cos((Latitude2 - Latitude1) * P) / 2 + cos(Latitude1 * P) * cos(Latitude2 * P) * (1 - cos((Longitude2 - Longitude1) * P)) / 2),
  Distancia is (12742 * asin(sqrt(A)) / 1000).
```

Distância Euclidiana 1.

Distância Eucledіana em Linha Reta

```
distanciaEuclidiana(Latitude1, Longitude1, Latitude2, Longitude2, Distancia) :-
 Distancia is (sqrt((Latitude2-Latitude1)^2 + (Longitude2-Longitude1)^2)) / 1000.
```

Distância Euclidiana 2.

Após obtenção do predicado de estima, dá-se início à elaboração do predicado capaz de responder à questão colocada. Para a realização desta questão, foi tida por base a implementação da estratégia de procura de A^* fornecida pelo docente da unidade curricular. Apresentam-se, de seguida, os predicados auxiliares a este:

- estimaDistancia: Predicado estima;
- eliminaNodo: Elimina um nodo de um caminho;
- inverteCaminho: Predicado para inverter uma lista;
- expandeD_aestrela: Predicado que guarda em ExpCaminhos a lista de todos os nodos adjacentes ao MelhorCaminho;
- append: Concatena duas listas;
- obtem_maisRapido: Predicado que calcula o melhor caminho de entre todos os caminhos possíveis;
- adjacenteEstrela: Calcula o próximo nodo adjacente, recalculando o valor da estima;
- adjacente: Verifica se duas paragens são adjacentes;
- aestrela: Predicado que aglomera os restantes predicados auxiliares.

4.9 Escolher o menor percurso (usando critério menor número de paragens)

Para a concretização desta questão, foi utilizado o método de procura primeiro em largura. Deste modo, o algoritmo regeu-se pelo seguinte pensamento:

Dados um ponto de origem e um ponto de destino, vão ser procurados todos os nodos adjacentes do ponto de origem, verificando que estes não se encontram já na lista de nodos visitados nem nos nodos em lista de espera. De seguida, será calculado para a lista resultante, o comprimento (isto é, o número de paragens) entre cada nodo adjacente e o destino, através do método de procura primeiro em largura. Será, então, ordenada a lista de duplos $(Adjacente, \#Paragens)$ por ordem crescente de número de paragens, ao qual será retirado o primeiro elemento. Este processo será repetido recursivamente até ser alcançado o nodo de destino.

4.10 Predicados Auxiliares

- comp: Predicado que calcula o comprimento de uma lista;
- nao: Extensão do meta-predicado não;
- distancia: Predicado que calcula a distância entre dois pontos de coordenadas através dos seus valores de latitude e longitude;
- distanciaEuclidiana: Predicado que calcula a distância Euclidiana entre dois pontos [4];
- primeiroPar: Predicado que devolve o primeiro elemento do primeiro par de uma lista de pares;
- inverteCaminho: Predicado que inverte uma ou várias listas;
- eliminaNodo: Predicado que elimina um nodo de um caminho;
- ordenarC: Predicado que ordena por ordem crescente do segundo elemento de uma lista de pares.

4.11 Predicados Extra

4.11.1 Calcular o tempo médio de viagem entre duas paragens

Este predicado permite conhecer o tempo médio de viagem entre duas paragens, fornecendo as paragens de origem e de destino, considerando o tempo de viagem de 5 minutos, acrescido 1 minuto de espera em cada paragem.

4.11.2 Calcular as paragens e o seu número por número de rua

Este predicado permite conhecer quais as paragens disponíveis numa rua.

4.12 Métodos de Comparação

No desenvolvimento das soluções, é pedida a consideração de diferentes estratégias de pesquisa (não-informada e informada), apresentando uma tabela comparativa, com as propriedades das estratégias, com as estratégias utilizadas.

Para este efeito, foram implementados os predicados anteriores em estratégias de procura primeiro em profundidade e em A^*.

A procura não informada primeiro em profundidade é um método que recorre a uma estratégia em que é expandido sempre um dos nós mais profundos da árvore. Consequentemente, tem as vantagens de necessitar de muita pouca memória, e ser bom para problemas com muitas soluções.

De acordo com os estudos realizados sobre as diversas questões, foram verificados os seguintes aspetos:

4.12.1 Algoritmo Ótimo

- O algoritmo A^* obtém sempre soluções ótimas;
- O algoritmo *Primeiro em Largura* obtém ocasionalmente soluções ótimas;
- O algoritmo *Primeiro em Profundidade* nunca obtém soluções ótimas.

4.12.2 Tempo de Execução

No que diz respeito aos tempos de execução das questões, estes dependeram do tamanho dos trajetos selecionados. Com a adição à posteriori de conhecimento onde são retiradas as carreiras de cada paragem, eliminaram-se as probabilidades de criação de ciclos infinitos. Assim sendo, tem-se:

- O algoritmo *Primeiro em Largura* é o algoritmo que obtém um melhor tempo de execução;
- O algoritmo *Primeiro em Profundidade* atinge bons tempos de execução, verificando-se um aumento exponencial deste para trajetos de alguma dimensão.
- O algoritmo A^* atinge os piores tempos de execução, uma vez que se trata do algoritmo mais complexo. Por este motivo, para trajetos de alguma dimensão, a sua performance decresce significativamente.

Algoritmo	Completo	Ótimo	Tempo	Ciclos Infinitos
Primeiro em Largura (BFS)	Sim	Ocasionalmente	Bom	Não
Primeiro em Profundidade (DFS)	Não	Não	Bom	Não
A*	Sim	Sim	Médio	Não

Tabela de Comparação.

5 Conclusões e Sugestões

O desenvolvimento deste projeto permitiu um aperfeiçoamento do conhecimento quanto ao paradigma da programação em lógica, fortalecendo a noção da sua utilidade na resolução de diversos problemas e no desenvolvimento de algoritmos de pesquisa.

Com base nas indicações fornecidas, foi possível o desenvolvimento de alguns algoritmos de pesquisa cruciais à concretização do projeto, permitindo praticar o uso dos mesmos e conhecer a sua necessidade de utilização.

Por fim, a realização deste exercício possibilitou o melhor entendimento da programação em lógica, bem como dos métodos de resolução de problemas e de procura.

Referências

[1] Introductory notes on prolog and ai search. *Introductory Notes on Prolog and AI Search*, page 18.

[2] Métodos de resolução de problemas e de procura. *Métodos de Resolução de Problemas e de Procura*.

[3] Pandas documentation.

[4] Alexander Bogomolny. The distance formula.

6 Anexos

Anexo I - Apresentação do código de *parsing*

Classe paragensOeiras

```
public class paragensOeiras {
    private int gid;
    private double latitude;
    private double longitude;
    private String estado;
    private String tipo;
    private String publicidade;
    private String operadora;
    private List<Integer> carreira;
    private int codigo;
    private String rua;
    private String freguesia;

    public paragensOeiras(int gid, double latitude, double longitude, String estado,
      String tipo, String publicidade, String operadora, List<Integer> carreira, int
      codigo, String rua, String freguesia) {

        this.gid = gid;
        this.latitude = latitude;
        this.longitude = longitude;
        this.estado = estado;
        this.tipo = tipo;
        this.publicidade = publicidade;
        this.operadora = operadora;
        this.carreira = carreira;
        this.codigo = codigo;
        this.rua = rua;
        this.freguesia = freguesia;
    }

    public static paragensOeiras lerLinhaToParagem(String linha){
        int gid = 0;
        double latitude = 0.0;
        double longitude = 0.0;
        String estado, tipo, publicidade, operadora, rua, freguesia;
        List<Integer> carreira = new ArrayList<>();
        int codigo = 0;

        String[] campos = linha.split(";");
        if (campos.length != 11) return null;

        estado = campos[3];
        tipo = campos[4];
        publicidade = campos[5];
        operadora = campos[6];
        String[] carreiras = campos[7].split(",");
```

```
    for (String c : carreiras) {
        carreira.add(Integer.parseInt(c));
    }
    rua = campos[9];
    freguesia = campos[10];

    if (estado == null || tipo == null || publicidade == null ||
        operadora == null || rua == null || freguesia == null ) return null;

    if (carreiras.length == 0) return null;

    try{
        gid = Integer.parseInt(campos[0]);
    }
    catch (NumberFormatException | InputMismatchException exc) {
        return null;
    }

    try{
        latitude = Double.parseDouble(campos[1]);
    }
    catch (NumberFormatException | InputMismatchException exc){
        return null;
    }

    try{
        longitude = Double.parseDouble(campos[2]);
    }
    catch (NumberFormatException | InputMismatchException exc){
        return null;
    }

    try{
        codigo = Integer.parseInt(campos[8]);
    }
    catch (NumberFormatException | InputMismatchException exc){
        return null;
    }

    return new paragensOeiras(gid, latitude, longitude, estado, tipo,
       publicidade, operadora, carreira, codigo, rua, freguesia);
}

@Override
public String toString() {
    return "paragem(" +
                   gid + "," +
                   latitude + "," +
                   longitude + "," +
                   estado.toLowerCase() + "," + '"' +
                   tipo.toLowerCase() + '"' + "," +
                   publicidade.toLowerCase() + "," +
```

```
                        operadora.toLowerCase() + "," +
                        carreira + "," +
                        codigo + "," + '"' +
                        rua.toLowerCase() + '"' + "," + '"' +
                        freguesia.toLowerCase() + '"' +
                        ").";
    }
}
```

Classe adjacenciasParagens

```
public class adjacenciasParagens {
    private int paragem1;
    private int paragem2;
    private int carreira;

    public adjacenciasParagens(int paragem1, int paragem2, int carreira) {
        this.paragem1 = paragem1;
        this.paragem2 = paragem2;
        this.carreira = carreira;
    }

    @Override
    public String toString() {
        return "adjacenciasParagens(" +
                                    paragem1 + "," +
                                    paragem2 + "," +
                                    carreira +
                                    ").";
    }
}
```

Classe baseConhecimento

```
public static int adicionaBaseParagens(String filename) throws IOException {
    File in = new File(filename);
    FileInputStream fis = new FileInputStream(in);
    BufferedReader br = new BufferedReader(new InputStreamReader(fis));

    FileWriter fos = new FileWriter("baseConhecimento.prolog.BB.pl", true);
    BufferedWriter out = new BufferedWriter(fos);

    String line = br.readLine();
    line = br.readLine();

    while(line != null){
        paragensOeiras paragem = paragensOeiras.lerLinhaToParagem(line);
            if (paragem != null){
                out.write(paragem.toString());
                out.newLine();
                out.flush();
            }
```

```
            line = br.readLine();
        }
    out.close();
    return 0;
}

public static int adicionaBaseAdjacentes() throws IOException{
    File folder = new File("/Users/carolina/Desktop/parser/carreiras");
    File[] listOfFiles = folder.listFiles((dir, name) ->
        name.toLowerCase().endsWith(".csv"));

    FileWriter fos = new FileWriter
        ("/Users/carolina/Desktop/parser/baseConhecimento.prolog.BB.pl", true);
    BufferedWriter out = new BufferedWriter(fos);

    for (File file : listOfFiles) {
        if (file.isFile()) {
            FileInputStream fis = new FileInputStream(file);
            BufferedReader br = new BufferedReader(new InputStreamReader(fis));

            adjacenciasParagens a;
            int p1 = 0;
            int p2 = 0;
            int carreira = 0;

            String line = br.readLine();
            line = br.readLine();

            while(line != null) {
                String[] campos = line.split(",");

                if (campos != null) {
                    p1 = Integer.parseInt(campos[0]);
                    carreira = Integer.parseInt(campos[7]);
                }

                line = br.readLine();
                if(line == null) break;
                campos = line.split(",");

                if (campos != null) {
                    p2 = Integer.parseInt(campos[0]);
                }

                a = new adjacenciasParagens(p1,p2,carreira);
                if (p1 == p2) a = null;
                if (a != null) {
                    out.write(a.toString());
                    out.newLine();
                    out.flush();
                }
            }
```

```
        }

    }
    out.close();
    return 0;
}
```

```
public static int adicionaBaseAdjacentesNRepetidas() throws IOException{
    File folder = new File("/Users/carolina/Desktop/parser/carreiras");
    File[] listOfFiles = folder.listFiles((dir, name) ->
    name.toLowerCase().endsWith(".csv"));

    FileWriter fos = new FileWriter("/Users/carolina/Desktop/parser/
    baseConhecimento.prolog.BB.pl", true);
    BufferedWriter out = new BufferedWriter(fos);

    List<adjacenciasNRepetidas> paragens = new ArrayList<>();

    for (File file : listOfFiles) {
        if (file.isFile()) {
            FileInputStream fis = new FileInputStream(file);
            BufferedReader br = new BufferedReader(new InputStreamReader(fis));

            adjacenciasNRepetidas a;
            int p1 = 0;
            int p2 = 0;

            String line = br.readLine();
            line = br.readLine(); //Começar a ler da segunda linha

            while(line != null) {
                String[] campos = line.split(",");

                if (campos != null) {
                    p1 = Integer.parseInt(campos[0]);
                }

                line = br.readLine();
                if(line == null) break;
                campos = line.split(",");

                if (campos != null) {
                    p2 = Integer.parseInt(campos[0]);
                }

                a = new adjacenciasNRepetidas(p1,p2);
                boolean passa = false;

                for (adjacenciasNRepetidas elem : paragens){
                    passa = elem.equals(a);
                    if (passa == true) break;
                }
                if (passa == false) paragens.add(a);

                if (p1 == p2) a = null;
                if (a != null && passa == false) {
                    out.write(a.toString());
                    out.newLine();
                    out.flush();
```

```
                }
            }

        }

    }
    out.close();
    return 0;

}
```

Anexo II - Demonstração de Predicados

```
| ?- bfs(467,858,Caminho).
Caminho = [467,9,466,186,465,893,902,472,904,476,482,483,470,473,886,231,241,240,859,858] ?
yes
| ?-
```

Calcular um trajeto entre dois pontos - Procura em Largura.

```
| ?- dfs(467,858,[],S).
S = [467,78,467,9,466,6,652,186,465,893,902,472,904,476,482,483,470,473,886,231,241,240,611,610,859,858] ?
yes
| ?-
```

Calcular um trajeto entre dois pontos - Procura em Profundidade.

```
| ?- maisRapido_aestrela(209,289,Caminho).
Caminho = [209,210,207,206,200,202,211,191,795,796,828,287,1004,290,289]/2.431694343698065 ?
yes
| ?-
```

Calcular um trajeto entre dois pontos - Procura em $A*$.

```
| ?- bfsOperadoras(467,858,[vimeca],Caminho).
Caminho = [467,78,654,655,56,491,490,458,457,335,363,344,345,342,338,86,27,26,339,16,352,351,323,313,360,858] ?
yes
| ?-
```

Selecionar apenas algumas das operadoras de transporte para um determinado percurso - Procura em Largura.

```
| ?- dfsOperadora(467,858,[],[vimeca],Caminho).
Caminho = [467,78,467,9,466,6,652,186,465,957,494,480,462,469,488,487,486,468,460,492,363,345,342,338,341,85,348,86,27,26,339,347
,362,361,637,638,643,349,359,861,332,858] ?
yes
| ?-
```

Selecionar apenas algumas das operadoras de transporte para um determinado percurso - Procura em Profundidade.

```
| ?- operadoras_aestrela(467,858,[vimeca],Caminho).
Caminho = [467,78,654,655,56,491,490,458,457,335,363,345,342,341,85,86,347,339,334,357,336,610,859,858] ?
yes
| ?-
```

Selecionar apenas algumas das operadoras de transporte para um determinado percurso - Procura em $A*$.

```
| ?- bfsNaoOperadoras(467,858,[scotturb],Caminho).
Caminho = [467,78,654,655,56,491,490,458,457,335,363,344,345,342,338,86,27,26,339,16,352,351,323,313,360,858] ?
yes
| ?-
```

Excluir um ou mais operadores de transporte para o percurso - Procura em Largura.

```
| ?- excluiOperadoras_aestrela(467,858,[scotturb],Caminho).
Caminho = [467,78,654,655,56,491,490,458,457,335,363,345,342,341,85,86,347,339,334,357,336,610,859,858] ?
yes
| ?-
```

Excluir um ou mais operadores de transporte para o percurso - Procura em $A*$.

```
| ?- bfsParagens(468,858,C).
C = [468,460,366,365,342,341,85,86,347,339,352,351,370,360,858] ?
yes
| ?-
```

Percurso que passe apenas por abrigos com publicidade - Procura em Largura.

```
| ?- dfsPublicidade(468,858,[],C).
C = [468,486,468,460,366,365,342,341,85,86,347,339,352,351,370,360,858] ?
yes
| ?-
```

Percurso que passe apenas por abrigos com publicidade - Procura em Profundidade.

```
| ?- publicidade_aestrela(468,858,Caminho).
Caminho = [468,460,366,365,342,341,85,86,347,339,352,351,370,360,858] ?
yes
| ?-
```

Percurso que passe apenas por abrigos com publicidade - Procura em $A*$.

```
| ?- bfsAbrigos(468,858,Caminho).
Caminho = [468,460,366,365,342,338,86,347,339,352,351,370,360,858] ?
yes
| ?-
| ?- bfsAbrigos(467,858,Caminho).
no
| ?-
```

Percurso que passe apenas por paragens abrigadas - Procura em Largura.

```
| ?- dfsAbrigos(468,858,[],Caminho).
Caminho = [468,486,468,460,366,365,342,338,341,85,348,86,347,362,361,637,638,643,349,359,861,332,331,315,312,360,858] ?
yes
| ?-
```

Percurso que passe apenas por paragens abrigadas - Procura em Profundidade.

```
| ?- abrigadas_aestrela(468,858,Caminho).
Caminho = [468,460,366,365,342,341,85,86,347,339,352,351,370,360,858] ?
yes
| ?-
```

Percurso que passe apenas por paragens abrigadas - Procura em $A*$.

```
| ?- bfsMaiorNumeroCarreiras(209,280,Caminho).
Caminho = [(797,5),(339,4),(26,4),(27,4),(86,4),(342,4),(207,4),(323,3),(351,3),(345,3),(363,3),(805,3),(210,3),(209,3),(352,2),(16,2),(338,2),(344,2),(335,2),(654,2),(178,2),(817,2),(813,2),(816,2),(236,2),(223,2),(1009,2),(801,2),(764,2),(781,2),(785,2),(208,2),(280,1),(278,1),(291,1),(457,1),(458,1),(490,1),(491,1),(56,1),(655,1),(59,1),(57,1),(58,1),(62,1),(63,1),(64,1),(61,1),(60,1),(32,1),(33,1),(364,1),(330,1),(845,1),(846,1),(333,1),(367,1),(857,1),(856,1),(863,1),(353,1),(354,1),(983,1),(986,1),(977,1),(1002,1),(954,1),(952,1),(823,1),(818,1),(807,1),(710,1),(792,1),(947,1),(693,1),(692,1),(235,1),(228,1),(768,1),(769,1)] ?
yes
| ?-
```

Paragens com o maior número de carreiras num determinado percurso - Procura em Largura.

```
| ?- dfsMaiorNumeroCarreiras(468,858,Caminho).
Caminho = [(339,4),(26,4),(27,4),(86,4),(342,4),(78,4),(858,3),(351,3),(323,3),(313,3),(312,3),(315,3),(331,3),(332,3),(861,3),(362,3),(347,3),(85,3),(341,3),(487,3),(486,3),(468,3),(360,2),(359,2),(349,2),(643,2),(638,2),(637,2),(361,2),(338,2),(460,2),(492,2),(335,2),(654,2),(467,2),(466,2),(186,2),(465,2),(488,2),(370,1),(348,1),(365,1),(366,1),(457,1),(458,1),(490,1),(491,1),(56,1),(655,1),(9,1),(6,1),(652,1),(957,1),(494,1),(480,1),(462,1),(469,1)] ?
yes
| ?-
```

Paragens com o maior número de carreiras num determinado percurso - Procura em Profundidade.

```
| ?- maiorNumeroCarreiras_aestrela(468,858,Caminho).
Caminho = [(339,4),(86,4),(342,4),(858,3),(351,3),(347,3),(85,3),(341,3),(468,3),(360,2),(352,2),(460,2),(370,1),(365,1),(366,1)] ?
yes
| ?-
```

Paragens com o maior número de carreiras num determinado percurso - Procura em $A*$.

```
| ?- bfsCaminhoXPontos(351,6,[313,246],Caminho).
Caminho = [351,323,313,312,315,331,332,861,860,599,609,799,1010,246,260,985,40,599,860,861,332,858,859,240,241,231,886,473,470,483,482,476,904,472,902,893,465,186,652,6] ?
yes
```

Escolher um ou mais pontos intermédios por onde o percurso deverá passar - Procura em Largura.

```
| ?- caminhoXPontos(351,6,[313,246],Caminho).
Caminho = [351,323,313,360,858,861,860,599,40,985,260,246,260,227,230,234,224,226,232,52,233,231,886,473,470,483,482,476,904,472,902,893,465,186,652,6] ?
yes
| ?-
```

Escolher um ou mais pontos intermédios por onde o percurso deverá passar - Procura em $A*$.

```
| ?- bfsMenorPercurso(193,212,C).
C = [193,233,231,886,473,470,483,482,476,904,472,902,893,465,186,466,467,78,654,59,57,58,62,63,64,61,60,32,33,364,330,845,846,333,367,857,856,863,353,354,983,986,977,1002,954,952,823,818,807,710,792,947,178,693,692,817,813,816,236,1009,801,805,228,764,768,769,781,756,208,797,207,212] ?
yes
```

Escolher o menor percurso (usando critério menor número de paragens) - Procura em Largura.

```
| ?- tempoPorViagem(185,261,Tempo).
Tempo = 24 ?
yes
| ?-
```

Calcula o tempo médio de viagem para um trajeto.

```
| ?- paragensRua(527,Paragens).
Paragens = [554,553,552,551,960]/5 ?
yes
| ?-
```

Conhecer as paragens e o seu número para um código de rua.