Universidade do Minho

Escola de Engenharia

Mestrado Integrado em Engenharia Informática

Unidade Curricular de Sistemas de Representação de Conhecimento e Raciocínio

Ano Lectivo de 2019/2020

Trabalho Prático Individual

Métodos de Resolução de Problemas e de Procura

Flávio Manuel Machado Martins (A65277)

Junho, 2020

Índice

1. Introdução

O desenvolvimento deste projeto tem como objetivos aprofundar os conhecimentos obtidos na unidade curricular de Sistemas de Representação de Conhecimento e Raciocínio, mais concretamente a utilização da Programação em Lógica, utilizando a linguagem de programação PROLOG, a prática de métodos de Resolução de Problemas e o desenvolvimento de algoritmos de pesquisa.

Como caso de estudo foram utilizados dados do sistema de transportes do concelho de Oeiras. Deve ser desenvolvido um sistema capaz de importar dados relativos às paragens de autocarros e carreiras que passam por estas mesmas paragens. Estes dados devem ser representados numa base de conhecimento de forma a desenvolver um sistema de recomendação de transporte público.

O sistema desenvolvido deve ser capaz de calcular um trajeto entre dois pontos, e ainda permitir algumas pesquisas mais complexas para o trajetos, como a titulo de exemplo, a exclusão ou seleção de operadoras nas paragens em que o trajeto passe, o menor percurso(com diversos critérios, por exemplo menor número de paragens e menor distância), a seleção de percursos passando apenas em paragens abrigadas ou com publicidade, e por fim a possibilidade de escolher um percurso a passar entre um ou mais pontos intermédios. O sistema deve ser ainda capaz de identificar quais as paragens com o maior número de carreiras num determinado percurso.

É proposto aos alunos que tentem utilizar estratégias de pesquisa(não-informada e informada) e apresentem uma tabela comparativa das utilizadas.

2. Base de Conhecimento

A base de conhecimento define a base de dados do sistema, antes de importar os dados foi preciso tomar decisões sobre como seria a base de conhecimento de modo a facilitar o desenvolvimento do sistema utilizando a informação inicial.
Tendo em conta que podemos olhar para o exercício como uma procura em grafos, iremos necessitar de definir nodos e arestas. Os nodos serão naturalmente as paragens, enquanto que as arestas são as varias ligação que em cada carreira acontecem de cada vez que se move o autocarro de uma paragem para a seguinte.

2.1. Paragem

Cada Paragem tem um identificador único e a informação relativa a essa paragem.

```
paragem ( gid, Latitude, Longitude, TipoDeAbrigo, AbrigoComPublicidade, Operadora, [Carreira] )
```

2.2. Ligação

Cada ligação consiste da sua carreira, os pontos de origem e destino e por fim a distância percorrida nesta ligação.

```
ligacao ( Carreira, OrigemGid, DestinoGid, Distancia )
```

3. Parser

Após definição da base de conhecimento é desenvolvido um parser, com recurso à linguagem Java, neste é criado um ficheiro *data.pl* onde será colocado o conhecimento, este será importado em duas partes, a partir dum ficheiro paragens.csv contendo a informação relativa às paragens e a partir dum ficheiros *carreiras.xlsx* contendo a informação relativa às carreiras.

3.1. Paragens

Para auxiliar o parse das paragens foi criada uma classe Paragem, nesta podermos armazenar a informação de cada paragem enquanto a vamos processando, nesta classe existe ainda uma função *toString()* que faz transforma a nossa paragem numa String já no formato definido na secção anterior para a paragem na base de conhecimentos.

```
public static class Paragem {
    String gid;
    double latitude;
    double longitude;
    String tipoDeAbrigo;
    String abrigoComPublicidade;
    String operadora;
    String carreiras;
    String codigoDeRua;
    String nomeDeRua;
    String freguesia;

    public String toString() {
        return ("paragem(" + this.gid + ", " + this.latitude + ", " + this.longitude + ", " +
                this.tipoDeAbrigo + ", " + this.abrigoComPublicidade + ", " +
                this.operadora +  ", [" +  this.carreiras + "] " + ").\n");
    }
}
```

Figura 1: Classe Paragem.

De seguida existe ainda uma função fixString(String s) que dada uma String lhe faz alguns reparos de modo a que a mesma seja compativel com o formato pretendido no PROLOG, isto consiste em, retirar os espaços em branco, retirar os caracteres “,” e “ ‘ “ e ainda transformar a letra inical numa minúscula. Caso a string dada seja de tamanho zero e devolvida uma String “invalidValue” que irá permitir prosseguir com o processamento dos dados e inserir na nossa base de conhecimento.

```
public static String fixString(String s) {
    String news;
    if(s.length()==0) return "invalidValue";
    news = s.replaceAll("\\s+","");
    news = news.replaceAll(",","");
    news = news.replaceAll("'","");
    char c[] = news.toCharArray();
    c[0] = Character.toLowerCase(c[0]);
    String str = new String(c);
    return str;
}
```

Figura 2: Função fixString.

Após esta preparação pode então ser realizado o parse das paragens.

Para começar é criado um HashMap onde serão colocadas todas as paragens, este HashMap irá ainda ser necessário na processamento das carreiras. Então criamos o ficheiro *data.pl* para onde iremos escrever a nossa base de conhecimento, como tal, as paragens e criamos um FileWriter para esse mesmo ficheiro. De seguida abrimos o ficheiro csv com as paragens e criamos um BefferedReader para esse mesmo ficheiro. Então iniciamos a leitura desse ficheiro, lendo o mesmo linha a linha, pois em cada linha encontramos uma paragem, e utilizando um delimitador o ";" dividimos cada linha, obtendo assim os vários campos de cada paragem e utilizando um inteiro para contabilizar em que coluna estamos, podemos num ciclo for percorrer toda a linha e adicionar a informação na paragem que é criada a cada linha lida. No final da leitura da linha a paragem é adicionada na HashMap de paragens e escrita no nosso ficheiro com a base de conhecimento com recurso à função toString() criada anteriormente.

```
public static void main(String[] args) throws IOException
{

    HashMap<String,Paragem> paragens = new HashMap<String,Paragem>();

    /* Criação ficheiro destino com os dados */
    File destFile = new File("data.pl");
    destFile.createNewFile();

    /* Escritor para ficheiro */
    FileWriter fWrite = new FileWriter("data.pl");


    /* --- PARSE PARAGENS --- */
    // Input file, formato csv com todas as paragens
    String file = "paragens.csv";
    BufferedReader fileReader = null;

    //Delimiter used in CSV file
    final String DELIMITER = ";";
    try
    {
        String line = "";
        //Create the file reader
        fileReader = new BufferedReader(new FileReader(file));

        if ((line = fileReader.readLine()) != null)
        {
            fWrite.write("%% ----- paragem ( Gid, Latitude, Longitude, TipoDeAbrigo, AbrigoComPublicidade, Operadora, [Carreira])\n")
        }

        int col;
        //Read the file line by line
        while ((line = fileReader.readLine()) != null)
        {
            //fWrite.write("Paragem(");
            Paragem p = new Paragem();
            //Get all tokens available in line
            String[] tokens = line.split(DELIMITER);
            col = 1;
            for(String token : tokens)
            {
                switch(col) {
                    case 1:     /* Id */
                        p.gid = fixString(token);
                        col++;
                        break;
                    case 2:     /* Latitude */
                        if (token.length()>0) p.latitude = Double.parseDouble(token);
                        else p.latitude = Double.parseDouble("-105000");  //Simular posição para o caso de erro encontrado no csv!
                        col++;
                                                    .
                                                    .
                                                    .
                        case 11:     /* Freguesia */
                            p.freguesia = fixString(token);
                            col++;
                            break;
                        default:
                            col++;
                            break;
                    }
                }
                paragens.put(p.gid,p);
                fWrite.write(p.toString());
```

Figura 3: Parse Paragens.

3.2. Carreiras

Para fazer o parse de carreiras vamos ler um ficheiro .xls e criar um ciclo capaz de percorrer todas as suas páginas, sendo que cada página contém uma carreira cujo seu identificador pode ser de imediato obtido pelo nome da página.

De seguido é criado um Array onde iremos armazenar, ordenadamente os ids de cada ponto da carreira, isto será realizado com recurso a um ciclo for que percorre todas as linhas da página.

Chegando ao fim da página, utilizando um ciclo for podemos percorrer todas as linhas e utilizando o HashMap criado durante o parse das paragens conseguimos ir buscar a localização de cada paragem conseguindo assim calcular a distancia euclidiana entre cada par de paragens e inserindo as nossas ligações(arestas) já com as distâncias de modo a facilitar as pesquisas pelos melhores caminhos posteriormente.

```
/* --- PARSE CARREIRAS --- */
// Creating a Workbook from an Excel file (.xls or .xlsx)
Workbook workbook = WorkbookFactory.create(new File("carreiras.xlsx"));

//fWrite.write("%% ----- Carreira (Id, [gid])\n");
fWrite.write("%% ----- ligacao(Carreira, OrigemGid, DestinoGid, Distancia)\n");
int r = 0;

for(Sheet sheet : workbook) {    /* Iterar pelas várias Pags */
    String carreira = sheet.getSheetName();
    ArrayList<String> gids = new ArrayList<>();
    for (Row row : sheet) {      /* Iterar pelas várias linhas */
        switch(r) {
            case 0:
                r++;
                break;
            default:
                gids.add(String.valueOf((int)Double.parseDouble(row.getCell(0).toString())));
                break;
        }
    }
    for(int i=1; i<gids.size(); i++) {

        double x1 = paragens.get(gids.get(i-1)).latitude;
        double x2 = paragens.get(gids.get(i-1)).longitude;
        double y1 = paragens.get(gids.get(i)).latitude;
        double y2 = paragens.get(gids.get(i)).longitude;

        double distance = Point2D.distance(x1, y1, x2, y2);
        Integer dist = (int)distance;
        fWrite.write("ligacao(" + carreira + ", " + gids.get(i-1) + ", " +
                                  gids.get(i) + ", " + dist + ").\n");
    }

    r = 0;
}
```

Figura 4: Parse Carreiras.

4. Funcionalidades (Pesquisas não-informadas)

4.1. Calcular trajeto

4.1.1 Depth-First Search

A primeira funcionalidade desenvolvida foi o calculo dum trajeto entre dois pontos.
Este trajeto foi calculado nesta primeira fase utilizando uma pesquisa não-informada, mais concretamente do tipo Depth-First Search, isto é, primeiro em profundidade, ou seja, a cada iteração irá ser feita a pesquisa não no próximo nodo adjacente ao original mas a um adjacente ao que já foi processado. Tratando se dum problema de grandes dimensões o algoritmo de Breadth First Search (em Largura) quase nunca é aconselhável, não sendo aqui exceção. Apesar de limitado o Depth-First Search será o mal menor para uma primeira implementação.

```
% ------------------------------------------------------------------------
% |     Caminhos entre 2 pontos (apresentando as ligacoes como resultado)   |
% ------------------------------------------------------------------------

trajeto( Origem, Destino, Caminho ) :-
    write('Percurso no formato:\n[(Carreira,Origem,Destino,Distancia)]\n'),
    percurso( Origem , Destino, [Origem], [], Caminho ).

% Se houver ligacao entre o destino e o gid actual temos um caminho
percurso( GidAtual, GidDestino, Visitados, Ligacoes, Caminho) :-
    ligacao( Car, GidAtual, GidDestino, Dist ),
    reverse([( Car,GidAtual,ProxGid, Dist )|Ligacoes], Caminho).
% --- reverse([GidDestino|Visitados],Caminho).    -- Apresenta resultado os gids onde passou

percurso( GidAtual, GidDestino, Visitados, Ligacoes, Caminho) :-
    ligacao( Car, GidAtual, ProxGid, Dist ),
    ProxGid \== GidDestino,
    \+memberchk( ProxGid, Visitados ),
    percurso( ProxGid, GidDestino, [ProxGid|Visitados], [(Car,GidAtual,ProxGid,Dist)|Ligacoes], Caminho).
```

Figura 5: Calculo de caminhos entre dois pontos.

Como pode ser visto na Figura 5 a implementação do algoritmo é bastante simples, caso exista uma ligação entre a paragem em que estamos(GidAtual no código) e a paragem onde deve terminar o caminho(GidDestino) chegamos a uma solução e podemos devolver um caminho, isto utilizando as opções(Ligacoes) tomadas até aqui, esta lista(Ligacoes) é nada mais do que uma lista de arestas(a ligação definida na base de conhecimento), ou seja, uma lista de caminhos entre pares de paragens contendo alem das paragens a carreira e distancia entre elas. Esta lista deve ser alvo dum 'reverse' pois a adição dos caminhos é feita ao inicio da lista durante o percurso, isto irá contribuir para uma diminuição complexidade da solução.
Caso não estejamos na paragem objetivo, adicionamos esta ligação ao percurso feito até aqui e iremos procurar num nodo adjacente ao atual, não regressando ao nodo anterior até visitar todos os nodos onde podemos chegar a partir do atual.
Ao longo da pesquisa existe uma lista de paragens já visitadas de modo a proteger a pesquisa de entrar em ciclos infinitos.

4.1.2 Depth-First Search com profundidade limitada

Tendo em conta a complexidade do problema foi ainda criada uma pesquisa identica mas com profundidade limitada, isto é, com um limite máximo para a profundidade, podendo assim se proteger o Sistema de algumas procuras demasiado profundas para a capacidade de processamento disponivel.

```
% ------------------------------------------------
% |     DFS (Pesquisa em profundidade limitada)    |
% ------------------------------------------------

trajetoLimProf( Origem, Destino, Limite, Caminho ) :-
    write('Percurso no formato:\n[(Carreira,Origem,Destino,Distancia)]\n'),
    percursoLimProf( Origem , Destino, Limite, [Origem], [], Caminho ).

% Se houver ligacao entre o destino e o gid actual temos um caminho
percursoLimProf( GidAtual, GidDestino, _, Visitados, Ligacoes, Caminho) :-
    ligacao( Car, GidAtual, GidDestino, Dist ),
    reverse([( Car,GidAtual,ProxGid, Dist )|Ligacoes], Caminho).
% --- reverse([GidDestino|Visitados],Caminho).    -- Apresenta resultado os gids onde passou

percursoLimProf( GidAtual, GidDestino, Limite, Visitados, Ligacoes, Caminho) :-
    Limite > 0,
    ligacao( Car, GidAtual, ProxGid, Dist ),
    ProxGid \== GidDestino,
    \+memberchk( ProxGid, Visitados ),
    Lim = Limite-1,
    percursoLimProf( ProxGid, GidDestino, Lim, [ProxGid|Visitados], [(Car,GidAtual,ProxGid,Dist)|Ligacoes], Caminho).
```

Figura 6: Pesquisa em profundidade limitada.

4.1.3 Profundidade Iterativa

De seguida, e recorrendo ao algoritmo anterior foi possível criar uma pesquisa em profundidade iterativa, isto é, faz uma pesquisa de profundidade limitada com um limite cada vez mais, procurando assim soluções com caminhos mais curtos e procura evitar esgotar os recursos disponíveis.

```
% ------------------------------------------------
% |     DFS (Pesquisa em profundidade iterativa)    |
% ------------------------------------------------

trajetoIter( Origem, Destino, Caminho ) :-
    write('Percurso no formato:\n[(Carreira,Origem,Destino,Distancia)]\n'),
    percursoIt( Origem , Destino, 0, Caminho ).

percursoIt( Origem, Destino, Lim, Caminho) :-
    percursoLimProf( Origem , Destino, Lim, [Origem], [], Caminho ),
    !;
    LimN = Lim+1,
    percursoIt( Origem, Destino, LimN, Caminho).
```

Figura 7: Pesquisa em profundidade iterativa.

4.2. Trajeto utilizando apenas algumas operadoras

Este problema pode ser facilmente resolvido recorrendo a qualquer dos algoritmos desenvolvidos anteriormente, apenas é necessário receber uma lista de operadoras e verificar em cada ligação se a paragem para onde é feita a ligação pertence a uma operadora das disponíveis.

```
% ------------------------------------------------------------------
% |     Caminhos entre 2 pontos, utilizando apenas algumas operadoras  |
% ------------------------------------------------------------------

trajetoSelOps( Origem, Destino, Operadoras, Percurso ) :-
    write('Percurso no formato:\n[(Carreira,Origem,Destino,OperadoraDestino)]\n'),
    paragem( Origem,_,_,_,_,Op,_),
    memberchk( Op, Operadoras ),
    paragem( Destino,_,_,_,_,Op2,_ ),
    memberchk( Op2, Operadoras ),
    percursoSelOps( Origem , Destino, Operadoras, [Origem], [], Caminho ),
    Percurso = Caminho.

% Se houver ligacao entre o destino e o gid actual temos um caminho
percursoSelOps( GidAtual, GidDestino, Operadoras, Visitados, Ligacoes, Caminho) :-
    ligacao( Car, GidAtual, GidDestino, Distancia ),
    paragem( GidDestino,_,_,_,_,Op,_ ),
    reverse([(Car,GidAtual,ProxGid,Op)|Ligacoes],Caminho).

percursoSelOps( GidAtual, GidDestino, Operadoras, Visitados, Ligacoes, Caminho) :-
    ligacao( Car, GidAtual, ProxGid, _),
    paragem( ProxGid,_,_,_,_,Op,_ ),
    memberchk(Op,Operadoras),
    ProxGid \== GidDestino,
    \+memberchk(ProxGid,Visitados),
    percursoSelOps( ProxGid, GidDestino, Operadoras, [ProxGid|Visitados], [(Car,GidAtual,ProxGid,Op)|Ligacoes], Caminho).
```

Figura 8: Trajeto selecionando apenas algumas operadoras(DFS).

4.3. Trajeto utilizando excluindo operadoras

Este problema também é facilmente resolvido recorrendo a qualquer dos algoritmos desenvolvidos anteriormente, apenas é necessário receber uma lista de operadoras e verificar em cada ligação se a paragem para onde é feita a ligação não pertence a uma operadora das excluidas.

```
% ------------------------------------------------------------------
% |     Caminhos entre 2 pontos, excluindo algumas operadoras      |
% ------------------------------------------------------------------

trajetoExOps( Origem, Destino, Operadoras, Percurso ) :-
    write('Percurso no formato:\n[(Carreira,Origem,Destino,OperadoraDestino)]\n'),
    paragem( Origem,_,_,_,_,Op,_ ),
    \+memberchk( Op, Operadoras ),
    paragem( Destino,_,_,_,_,Op2,_ ),
    \+memberchk( Op2, Operadoras ),
    percursoExOps( Origem , Destino, Operadoras, [Origem], [], Caminho ),
    Percurso = Caminho.

% Se houver ligacao entre o destino e o gid actual temos um caminho
percursoExOps( GidAtual, GidDestino, Operadoras, Visitados, Ligacoes, Caminho) :-
    ligacao( Car, GidAtual, GidDestino, Distancia ),
    paragem( GidDestino,_,_,_,_,Op,_ ),
    reverse([(Car,GidAtual,GidDestino,Op)|Ligacoes],Caminho).

percursoExOps( GidAtual, GidDestino, Operadoras, Visitados, Ligacoes, Caminho) :-
    ligacao( Car, GidAtual, ProxGid, _),
    paragem( ProxGid,_,_,_,_,Op,_ ),
    \+memberchk(Op,Operadoras),
    ProxGid \== GidDestino,
    \+memberchk(ProxGid,Visitados),
    percursoExOps( ProxGid, GidDestino, Operadoras, [ProxGid|Visitados], [(Car,GidAtual,ProxGid,Op)|Ligacoes], Caminho).
```

Figura 9: Trajeto excluindo algumas operadoras(FDS).

4.4. Paragens com mais carreiras num determinado percurso

Para esta funcionalidade, considerando um percurso como sendo a lista de paragens pelas quais o mesmo passa, facilmente se desenvolve um algoritmo que dado uma lista de paragens, e tendo cada paragem associada a si uma lista das carreiras de que faz parte, para cada paragem vamos buscar o tamanho da sua lista de carreiras, e mantendo um contador do tamanho máximo encontrado podemos percorrer a lista, ignorando paragens com menos carreiras, adicionando a uma lista temporária as que tiverem igual à maior até ao momento, e, sempre que aparecer uma paragem com mais carreiras do que o máximo atual, podemos substituir esse valor de tamanho pelo novo máximo e criar uma nova lista temporária apenas com a paragem que acabamos de encontrar, repetindo este processo recursivamente até esvaziar o caminho.

```
% ------------------------------------------------------------
% |     Paragens com mais carreiras num determinado percurso   |
% ------------------------------------------------------------

%%      ([pto1,pto2,...,pton], RES).
paragemMaisCar([P|T],R) :-
    paragem( P,_,_,_,_,_,LP),
    length(LP,Len),
    paragemMaisCarAux(T,Len,P,R).

paragemMaisCarAux([],_,LS,R) :-
    R = LS.

paragemMaisCarAux([P|T],S,X,R) :-
    paragem( P,_,_,_,_,_,LP),
    length(LP,LPS),
    LPS > S,
    paragemMaisCarAux(T,LPS,[P],R).
paragemMaisCarAux([P|T],S,X,R) :-
    paragem( P,_,_,_,_,_,LP),
    length(LP,LPS),
    LPS == S,
    paragemMaisCarAux(T,S,[P|X],R).
paragemMaisCarAux([P|T],S,X,R) :-
    paragem( P,_,_,_,_,_,LP),
    length(LP,LPS),
    LPS < S,
    paragemMaisCarAux(T,S,X,R).
```

Figura 10: Paragens com o maior número de carreiras num determinado percurso.

4.5. Menor percurso(critério menor numero de paragens)

Esta funcionalidade pode ser resolvida de imediato recorrendo à pesquisa em profundidade iterativa desenvolvida anteriormente.

Também é possível resolver o mesmo em profundidade primeiro mas para tal e necessário fazer um findall que encontre todas as soluções, e posteriormente calculado o trajeto com menos paragens. Obviamente, neste tipo de pesquisa, procurar todos os caminhos é bastante mais complexo do que a procura iterativa, que irá parar a pesquisa assim que encontrar um resultado com a certeza que no máximo existe algum resultado equivalente mas nunca menor.

```
% ------------------------------------------------------------
% |     Menor percurso(criterio menor numero de paragens)    |
% ------------------------------------------------------------

trajetoMenosPar(Inicio,Destino,R) :-
    findall(X, trajeto(Inicio,Destino,X), L),
    menosPar(L,R).

menosPar([L|Ls], Min) :-
    menosPar(Ls, L, Min).

menosPar([], Min, Min).
menosPar([L|Ls], Min0, Min) :-
    min(L, Min0, Min1),
    menosPar(Ls, Min1, Min).
min(L,Min0,R) :-
    length(L,LL),
    length(Min0,LM),
    LL>LM,
    R=Min0.
min(L,Min0,R) :-
    length(L,LL),
    length(Min0,LM),
    LL=<LM,                          %>
    R=L.

trajetoMenosParIter(Inicio,Destino,R) :-
    trajetoIter( Inicio, Destino, R ).
```

Figura 11: Menor percurso (menor número de paragens).

4.6. Menor percurso(critério menor distancia)

Para encontrar o percurso mais rápido utilizando como critério a distância total percorrida no percurso podemos recorrer do campo relativo à distância de cada ligação. Assim, uma versão inicial, embora pouco eficaz, de realizar esta procura consiste em, recorrendo ao primeiro trajeto desenvolvido(DFS), podemos fazer um findall para encontrar todos os caminhos.
Assim, a partir da lista de caminhos encontrada podemos calcular o mais rápido(menor distância percorrida) deles, isto é feito de forma recursiva, inicialmente guardamos o primeiro percurso, e a partir daqui para cada percurso da lista fazemos a comparação da distancia total desse percurso com a do percurso guardado, obtendo a menor distância continuamos a fazer o calculo do menor caminho do restante da lista comparando com o menor da iteração anterior.

```
% ------------------------------------------------------
% |     percurso mais rapido(criterio menor distancia) |
% ------------------------------------------------------

trajetoMaisRap(Inicio,Destino,R,Dist) :-
    findall(X, trajeto(Inicio,Destino,X), L),
    maisRap(L,R),
    distCam(R,Dist).

maisRap([L|Ls], Min) :-
    maisRap(Ls, L, Min).

maisRap([], Min, Min).
maisRap([L|Ls], Min0, Min) :-
    menorDist(L, Min0, Min1),
    maisRap(Ls, Min1, Min).
menorDist(L,Min0,R) :-
    distCam(L,LL),
    distCam(Min0,LM),
    LL>LM,
    R=Min0.
menorDist(L,Min0,R) :-
    distCam(L,LL),
    distCam(Min0,LM),
    LL=<LM,                %>
    R=L.

distCam([], R) :-
    R is 0.
distCam([(Car,Or,Dest,Dist)|L], R) :-
    distCam(L,R2),
    R is Dist+R2.
```

Figura 12: Percurso mais rápido(critério de menor distância).

No próximo capitulo irei apresentar duas estratégias de pesquisa informada, sendo que essas são bastante mais eficazes para esta pesquisa, sendo a sua complexidade bastante inferior.

4.7. Percurso a passar apenas em paragens com publicidade

A forma mais básica de resolver esta pesquisa é utilizar novamente a primeira pesquisa desenvolvida, isto é, DFS, adaptando a mesma com a obtenção da existência de publicidade, que facilmente obtemos ao ir buscar as paragens da base de conhecimento.

Ora com a informação da existência de publicidade em cada paragem facilmente alteramos o algoritmo de pesquisa desenvolvimento para verificar que apenas segue ligações em que ambos as paragens tem publicidade.

```
% ------------------------------------------------------------
% |     percurso a passar apenas em paragens com publicidade   |
% ------------------------------------------------------------

trajetoPub( Origem, Destino, Percurso ) :-
    write('Percurso no formato:\n[(Carreira,Origem,Destino,OperadoraDestino)]\n'),
    paragem( Origem,_,_,_,yes,_,_ ),
    paragem( Destino,_,_,_,yes,_,_ ),
    percursoPub( Origem , Destino, Operadoras, [Origem], [], Caminho ),
    Percurso = Caminho.

% Se houver ligacao entre o destino e o gid actual temos um caminho
percursoPub( GidAtual, GidDestino, Operadoras, Visitados, Ligacoes, Caminho) :-
    ligacao( Car, GidAtual, GidDestino, Distancia ),
    paragem( GidDestino,_,_,_,yes,_,_ ),
    reverse([(Car,GidAtual,GidDestino)|Ligacoes],Caminho).

percursoPub( GidAtual, GidDestino, Operadoras, Visitados, Ligacoes, Caminho) :-
    ligacao( Car, GidAtual, ProxGid, Distancia ),
    paragem( ProxGid,_,_,_,yes,_,_ ),
    ProxGid \== GidDestino,
    \+memberchk(ProxGid,Visitados),
    percursoPub( ProxGid, GidDestino, Operadoras, [ProxGid|Visitados], [(Car,GidAtual,ProxGid)|Ligacoes], Caminho).
```

Figura 13: Percurso que passe apenas por abrigos com publicidade.

4.8. Percurso a passar apenas em paragens abrigadas

Esta pesquisa pode ser resolvida de forma idêntica à anterior sendo que o critério será de que o Abrigo da paragem seja diferente de "*semAbrigo*", o que implica a existência de algum abrigo nas paragens.

```
% ------------------------------------------------------------
% |     percurso a passar apenas em paragens abrigadas      |
% ------------------------------------------------------------
trajetoAbrigado( Origem, Destino, Percurso ) :-
    write('Percurso no formato:\n[(Carreira,Origem,Destino,OperadoraDestino)]\n'),
    paragem( Origem,_,_,Abrigo,_,_,_ ),
    Abrigo \== semAbrigo,
    paragem( Destino,_,_,Abrigo2,_,_,_ ),
    Abrigo2 \== semAbrigo,
    percursoAbrigado( Origem , Destino, Operadoras, [Origem], [], Caminho ),
    Percurso = Caminho.

% Se houver ligacao entre o destino e o gid actual temos um caminho
percursoAbrigado( GidAtual, GidDestino, Operadoras, Visitados, Ligacoes, Caminho) :-
    ligacao( Car, GidAtual, GidDestino, Distancia ),
    paragem( GidDestino,_,_,_,yes,_,_ ),
    reverse([(Car,GidAtual,GidDestino)|Ligacoes],Caminho).

percursoAbrigado( GidAtual, GidDestino, Operadoras, Visitados, Ligacoes, Caminho) :-
    ligacao( Car, GidAtual, ProxGid, Distancia ),
    paragem( ProxGid,_,_,Abrigo,_,_,_ ),
    Abrigo \== semAbrigo,
    ProxGid \== GidDestino,
    \+memberchk(ProxGid,Visitados),
    percursoAbrigado( ProxGid, GidDestino, Operadoras, [ProxGid|Visitados], [(Car,GidAtual,ProxGid)|Ligacoes], Caminho).
```

Figura 14: Percurso que passe apenas por paragens abrigadas.

4.9. Percurso a passar em um ou mais pontos intermédios

Esta pesquisa também é facilmente feita com recurso à pesquisa em profundidade inicialmente desenvolvida, que quando encontramos um caminho é feita a verificação de que este passe em todos os pontos pedidos, caso contrário a pesquisa segue a procura por um caminho alternativo.

```
% ------------------------------------------------------------
% |     percurso a passar em um ou mais ptos intermedios     |
% ------------------------------------------------------------
trajetoPtosInt( Origem, Destino, Pontos ,Caminho ) :-
    write('Percurso no formato:\n[(Carreira,Origem,Destino,Distancia)]\n'),
    percursoPtosInt( Origem , Destino, Pontos, [Origem], [], Caminho ).

% Se houver ligacao entre o destino e o gid actual temos um caminho
percursoPtosInt( GidAtual, GidDestino, Pontos, Visitados, Ligacoes, Caminho) :-
    visitaTodos(Pontos,Visitados),
    ligacao( Car, GidAtual, GidDestino, Dist ),
    reverse([( Car,GidAtual,ProxGid, Dist )|Ligacoes], Caminho).
% --- reverse([GidDestino|Visitados],Caminho).    -- Apresenta resultado os gids onde passou

percursoPtosInt( GidAtual, GidDestino, Pontos, Visitados, Ligacoes, Caminho) :-
    ligacao( Car, GidAtual, ProxGid, Dist ),
    ProxGid \== GidDestino,
    \+memberchk( ProxGid, Visitados ),
    percursoPtosInt( ProxGid, GidDestino, Pontos, [ProxGid|Visitados], [(Car,GidAtual,ProxGid,Dist)|Ligacoes], Caminho).

visitaTodos([], _).
visitaTodos([X|Xs], L) :-
    member(X, L),
    visitaTodos(Xs, L).
```

Figura 15: Percurso que passa por um ou mais pontos intermédios.

5. Pesquisas informadas

5.1. Pesquisa Gulosa

Uma pesquisa gulosa consiste em expandir o nodo que aparenta estar mais perto da solução, para tal existe uma função heurística que estima o custo do caminho mais curto do estado n para o objetivo, neste exemplo a nossa heurística será a distância euclidiana, então a cada iteração iremos escolher o nodo ligado ao atual que não tenha sido ainda visitado e esteja mais próximo do nodo objetivo, utilizando a distância euclidiana como fator de decisão.

h(n) = custo estimado do caminho mais curto do estado n para o objetivo **(função heurística)**

Como podemos ver na Figura 16 a forma como desenvolvi esta pesquisa é bastante simples, começamos por encontrar todos os nodos para onde é possível deslocar a partir do atual, eliminamos os já visitados e escolhemos então o mais próximo dos disponíveis, isto é, o caminho cujo próximo nodo(paragem) tem menor heurística(distância euclidiana para a paragem objetivo). Após o processamento de cada caminho, caso não seja possível encontrar um caminho até ao objetivo repetimos o passo atual acrescentando o nodo já processado aos visitados.

```
% -----------------
% |    Gulosa     |
% -----------------
gulosa( Origem, Destino, Caminho ) :-
    write('Percurso no formato:\n[(Carreira,Origem,Destino,Distancia)]\n'),
    percursoGulosa( Origem , Destino, [Origem], [], Caminho ).

percursoGulosa( GidAtual, GidDestino, Visitados, Ligacoes, Caminho) :-
    ligacao( Car, GidAtual, GidDestino, Dist ),
    reverse([( Car,GidAtual,ProxGid, Dist )|Ligacoes], Caminho).

percursoGulosa( GidAtual, GidDestino, Visitados, Ligacoes, Caminho) :-
    findall(( C, GidAtual, P, D ),ligacao( C, GidAtual, P, D ),Opcoes),
    removeVisitados(Opcoes,Visitados,[],OpcoesPorVisitar),
    length(OpcoesPorVisitar,L),
    L > 0,
    escolheCaminho(OpcoesPorVisitar,GidDestino,(Car,GidAtual,ProxGid,Dist)),
    write('Escolhido:: '),write(ProxGid),write('\n'),
    ProxGid \== GidDestino,
    \+memberchk( ProxGid, Visitados ),
    ( percursoGulosa( ProxGid, GidDestino, [ProxGid|Visitados], [(Car,GidAtual,ProxGid,Dist)|Ligacoes], Caminho);
    percursoGulosa( GidAtual, GidDestino, [ProxGid|Visitados], Ligacoes, Caminho)).

escolheCaminho([(Car,Origem,Destino,Custo)|Caminhos],GOAL,R) :-
    estima(Destino,GOAL,EOA),
    escolheCaminho(Caminhos,GOAL,(Car,Origem,Destino,Custo),EOA,R).
escolheCaminho([],G,OA,EOA,R) :-
    R = OA.
escolheCaminho([(Car,Origem,Destino,Custo)|Caminhos],GOAL,OA,EOA,R) :-                % OA : Opção atual  | COA: Estimativa OA
    estima(Destino,GOAL,Estima),
    Estima < EOA,
    escolheCaminho(Caminhos,GOAL,(Car,Origem,Destino,Custo),Estima,R).
escolheCaminho([(Car,Origem,Destino,Custo)|Caminhos],GOAL,OA,EOA,R) :-                % OA : Opção atual  | COA: Estimativa OA
    estima(Destino,GOAL,Estima),
    Estima >= EOA,
    escolheCaminho(Caminhos,GOAL,OA,EOA,R).

removeVisitados([], _, TR, TR).
removeVisitados([(C,O,D,Dist)|Caminhos],Vis,TR,R) :-
    \+memberchk(D,Vis),
    removeVisitados(Caminhos,Vis,[(C,O,D,Dist)|TR],R).
removeVisitados([(C,O,D,Dist)|Caminhos],Vis,TR,R) :-
    memberchk( D,Vis ),
    removeVisitados(Caminhos,Vis,TR,R).

estima(O,D,R) :-
    distEuc(O,D,R).
distEuc(O,D,R) :-
    paragem( O,X1,Y1,_,_,_,_ ),
    paragem( D,X2,Y2,_,_,_,_ ),
    R is sqrt((X2-X1)^2 + (Y2-Y1)^2).
```

Figura 16: Pesquisa gulosa de trajeto

Esta pesquisa, embora não seja ótima, é mais eficaz do que as pesquisas não informadas pois tenta se deslocar de forma a aproximar do destino e não aleatória, apesar de que, esta aproximação não garante a escolha do melhor caminho por diversos fatores, como os desvios no percurso seguinte à tomada de decisão, a existência de percursos mais diretos alternativos, entre muitos outros.

5.2. Pesquisa A*

A partir da pesquisa gulosa desenvolvida anteriormente podemos melhorar a mesma, fazendo uma pesquisa A* através da alteração da heurística utilizada. Na pesquisa A* além de considerar a distância do próximo nodo ao objetivo, iremos ainda considerar a distância entre o nodo atual e o próximo.

Assim passaremos ter uma heurística(f(n)):

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

- *g(n) = custo total chegar ao estado n (custo do percurso)*
- *h(n) = custo estimado para chegar ao objetivo (não deve sobrestimar o custo para chegar à solução (heurística))*
- *f(n)= custo estimado da solução mais barata que passa pelo nó n*

```
% -------------
% |    A*     |
% -------------

aestrela( Origem, Destino, Caminho ) :-
    write('Percurso no formato:\n[(Carreira,Origem,Destino,Distancia)]\n'),
    percursoAestrela( Origem , Destino, [Origem], [], Caminho ).

% Se houver ligacao entre o destino e o gid actual temos um caminho
percursoAestrela( GidAtual, GidDestino, Visitados, Ligacoes, Caminho) :-
    ligacao( Car, GidAtual, GidDestino, Dist ),
    reverse([( Car,GidAtual,ProxGid, Dist )|Ligacoes], Caminho).
% --- reverse([GidDestino|Visitados],Caminho).    -- Apresenta resultado os gids onde passou

percursoAestrela( GidAtual, GidDestino, Visitados, Ligacoes, Caminho) :-
    findall(( C, GidAtual, P, D ),ligacao( C, GidAtual, P, D ),Opcoes),
    removeVisitados(Opcoes,Visitados,[],OpcoesPorVisitar),
    length(OpcoesPorVisitar,L),
    L > 0,
    escolheCaminhoAestr(OpcoesPorVisitar,GidDestino,(Car,GidAtual,ProxGid,Dist)),
    ProxGid \== GidDestino,
    \+memberchk( ProxGid, Visitados ),
    ( percursoAestrela( ProxGid, GidDestino, [ProxGid|Visitados], [(Car,GidAtual,ProxGid,Dist)|Ligacoes], Caminho);
      percursoAestrela( GidAtual, GidDestino, [ProxGid|Visitados], Ligacoes, Caminho)).

escolheCaminhoAestr([(Car,Origem,Destino,Custo)|Caminhos],GOAL,R) :-
    estimaAestr(Custo,Destino,GOAL,EOA),
    escolheCaminhoAestr(Caminhos,GOAL,(Car,Origem,Destino,Custo),EOA,R).
escolheCaminhoAestr([],G,OA,EOA,R) :-
    R = OA.
escolheCaminhoAestr([(Car,Origem,Destino,Custo)|Caminhos],GOAL,OA,EOA,R) :-                % OA : Opção atual  | COA: Estimativa OA
    estimaAestr(Custo,Destino,GOAL,Estima),
    Estima < EOA,
    escolheCaminhoAestr(Caminhos,GOAL,(Car,Origem,Destino,Custo),Estima,R).
escolheCaminhoAestr([(Car,Origem,Destino,Custo)|Caminhos],GOAL,OA,EOA,R) :-                % OA : Opção atual  | COA: Estimativa OA
    estimaAestr(Custo,Destino,GOAL,Estima),
    Estima >= EOA,
    escolheCaminhoAestr(Caminhos,GOAL,OA,EOA,R).

estimaAestr(Custo,O,D,R) :-
    distEuc(O,D,DE),
    R is Custo+O.
```

Figura 17: Pesquisa A de trajeto*

Este algoritmo permite a obtenção de uma solução ótima ao fazer as suas opções com base numa estimativa que tem em conta o custo da deslocação para a próxima paragem e também a estimativa do custo dessa paragem até ao destino final.

6. Comparação estratégias utilizadas

Critério	**Profundidade Primeiro**	**Profundidade Limitada**	**Iterativa**	**Gulosa**	**A***
Completa	Não	Não	Sim	Não	Sim
Tempo	O(b^m)	O(b^l)	O(b^d)	Pior caso: O(b^m) Melhor caso: O(b^	Número de nodos com g(n)+h(n) =< C*
Espaço	O(b^m)	O(b^l)	O(b^d)		
Ótima	Não	Não	Sim	Não	Sim (depende da heurística)

- B é o fator de ramificação
- d é a profundidade da solução
- m é a máxima profundidade da árvore
- l é a profundidade limite de pesquisa

Regra geral a melhor pesquisa não informada para problemas com um grande espaço de pesquisa e profundidade da solução desconhecida é a iterativa, isto por encontrar uma solução ótima(tendo em conta a profundidade), mas esta acaba por ser naturalmente inferior em relação às pesquisas informadas. Enquanto a pesquisa gulosa não permite uma solução ótima, com uns ajustes podemos criar uma estratégia de pesquisa A*, e neste caso, com a utilização duma heurística aceitável podemos obter uma solução ótima e diminuir bastante a complexidade do tempo pois será possível optar pelos melhores caminhos em cada situação e assim chegar em menos passos à solução ótima.

7. Demonstração

Esta secção serve para demonstrar algumas execuções do sistema desenvolvido e para comparar os resultados de diferentes pesquisas.

```
| ?- trajeto(224,52,R).
Percurso no formato:
[(Carreira,Origem,Destino,Distancia)]
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(2,226,1001,10758),(2,1001,607,12552),(2,607,232,11019),(2,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(2,226,1001,10758),(2,1001,607,12552),(2,607,232,11019),(6,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(2,226,1001,10758),(2,1001,607,12552),(2,607,232,11019),(13,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(2,226,1001,10758),(2,1001,607,12552),(2,607,232,11019),(15,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(2,226,1001,10758),(7,1001,607,12552),(2,607,232,11019),(2,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(2,226,1001,10758),(7,1001,607,12552),(2,607,232,11019),(6,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(2,226,1001,10758),(7,1001,607,12552),(2,607,232,11019),(13,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(2,226,1001,10758),(7,1001,607,12552),(2,607,232,11019),(15,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(6,226,232,8922),(2,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(6,226,232,8922),(6,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(6,226,232,8922),(13,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(6,226,232,8922),(15,232,_A,9209)] ? ;
Prolog interruption (h for help)? a
% Execution aborted
```

Figura 18: percursos calculados por DFS entre 224,52

```
| ?- trajetoLimProf(224,52,10,R).
Percurso no formato:
[(Carreira,Origem,Destino,Distancia)]
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(2,226,1001,10758),(2,1001,607,12552),(2,607,232,11019),(2,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(2,226,1001,10758),(2,1001,607,12552),(2,607,232,11019),(6,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(2,226,1001,10758),(2,1001,607,12552),(2,607,232,11019),(13,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(2,226,1001,10758),(2,1001,607,12552),(2,607,232,11019),(15,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(2,226,1001,10758),(7,1001,607,12552),(2,607,232,11019),(2,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(2,226,1001,10758),(7,1001,607,12552),(2,607,232,11019),(6,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(2,226,1001,10758),(7,1001,607,12552),(2,607,232,11019),(13,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(2,226,1001,10758),(7,1001,607,12552),(2,607,232,11019),(15,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(6,226,232,8922),(2,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(6,226,232,8922),(6,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(6,226,232,8922),(13,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(6,226,232,8922),(15,232,_A,9209)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(13,226,42,10888),(1,42,600,12737),(1,600,602,13606),(1,602,601,14425),(1,601,...
R = [(2,224,239,9686),(2,239,238,10469),(2,238,226,9596),(13,226,42,10888),(1,42,600,12737),(1,600,602,13606),(1,602,601,14425),(1,601,...
```

Figura 19: Percursos calculados com limite de profundidade entre 224 e 52

```
| ?- trajetoIter(224,52,R).
Percurso no formato:
[(Carreira,Origem,Destino,Distancia)]
R = [(6,224,226,8736),(6,226,232,8922),(2,232,_A,9209)]
```

Figura 20: Percurso calculado pela pesquisa iterativa entre 224 e 52

Para começar, e utilizando apenas pesquisas não informadas, utilizando como ponto de partida para um percurso a paragem com gid 224 e como destino a paragem com o gid 52 podemos verificar algumas diferenças entre as três estratégias de pesquisa não informada utilizada.

A pesquisa em profundidade encontra uma serie de percursos, mas como utiliza uma pesquisa em profundidade sem limite começa a tornar se bastante lenta, pois segue caminhos que se estão a afastar do objetivo, e cujo seu limite pode estar a uma distância enorme. De seguida, utilizando uma pesquisa em profundidad limitada a 10, é possível observer que ignorando a pesquisa em profundidade a partir dum nível razoavel, regressamos aos caminhos mais proximos por processor e encontramos novas soluçoes. Por fim, e tendo em conta que com as estratégias anteriores foi possível ver que nenhuma solução era ótima, pois os primeiros caminhos a aparecer eram maiores que os restantes,realizamos uma pesquisa iterative, esta apenas encontra um caminho mas como se pode observer pela comparação com os caminhos encontrados anteriormente esta é uma solução melhor do que todas as encontradas até aqui, ficando assim demonstrado como esta estratégia de pesquisa é geralmente a mais eficaz entre as não informadas.

```
trajetoSelOps(224,232,[vimeca,sCoTTURE],R).
Percurso no formato:
[(Carreira,Origem,Destino,OperadoraDestino)]
no
| ?- trajetoSelOps(224,232,[vimeca,sCoTTUREB],R).
Percurso no formato:
[(Carreira,Origem,Destino,OperadoraDestino)]
no
| ?- trajetoSelOps(224,232,[vimeca,sCoTTURB],R).
Percurso no formato:
[(Carreira,Origem,Destino,OperadoraDestino)]
R = [(2,224,239,vimeca),(2,239,238,sCoTTURB),(2,238,226,vimeca),(6,226,_A,sCoTTURB)] ?
yes
| ?- trajetoSelOps(224,232,[vimeca],R).
Percurso no formato:
[(Carreira,Origem,Destino,OperadoraDestino)]
no
| ?- trajetoSelOps(224,232,[vimeca,sCoTTURB],R).
Percurso no formato:
[(Carreira,Origem,Destino,OperadoraDestino)]
R = [(2,224,239,vimeca),(2,239,238,sCoTTURB),(2,238,226,vimeca),(6,226,_A,sCoTTURB)] ?
yes
| ?- trajetoExOps(224,232,[vimeca],R).
Percurso no formato:
[(Carreira,Origem,Destino,OperadoraDestino)]
no
| ?- trajetoExOps(224,232,[vimeca,sCoTTURB],R).
Percurso no formato:
[(Carreira,Origem,Destino,OperadoraDestino)]
no
| ?- trajetoExOps(224,232,[sCoTTURB],R).
Percurso no formato:
[(Carreira,Origem,Destino,OperadoraDestino)]
no
| ?- trajetoExOps(224,232,[carris],R).
Percurso no formato:
[(Carreira,Origem,Destino,OperadoraDestino)]
R = [(2,224,239,vimeca),(2,239,238,sCoTTURB),(2,238,226,vimeca),(6,226,232,sCoTTURB)] ?
yes
```

Figura 21: Demonstração de utilização dos filtros de paragens entre 224 e 232

```
| ?- paragemMaisCar([224,239,238,226],R).
R = [226|224] ?
yes
```

Figura 22: Demonstração de calculo das paragens com mais carreiras num determinado percurso

No exemplo acima, as paragens 225 e 224 tem 4 carreiras enquanto que as paragens 239 e 238 tem 3, pelo que o resultado se confirma.

```
| ?- trajetoMenosPar(224,226,R).
Percurso no formato:
[(Carreira,Origem,Destino,Distancia)]
Prolog interruption (h for help)? a
% Execution aborted
| ?- trajetoMenosParIter(224,226,R).
Percurso no formato:
[(Carreira,Origem,Destino,Distancia)]
R = [(6,224,_A,8736)] ?
yes
| ?-
```

Figura 23: Calculo de menor percurso pelo criterio de paragens

O calculo de percurso com menor número de paragens, utilizando uma estratégia de pesquisa em profundidade torna-se bastante exaustivo, isso fica demonstrado no exemplo da figura 23, que o sistema se torna tão lento que acabo por abortar o pedido, faço entao um pedido identico mas utilizando a pesquisa iterativa, e muito facilmente obtenho o resultado.

```
| ?- trajetoPub(224,226,R).
Percurso no formato:
[(Carreira,Origem,Destino,OperadoraDestino)]
no
| ?- trajetoAbrigado(224,226,R).
Percurso no formato:
[(Carreira,Origem,Destino,OperadoraDestino)]
no
```

Figura 24: Pesquisa trajeto paragem abrigada e com publicidade

Na figura 24 podemos verificar, utilizando um percurso que está demonstrado que o Sistema consegue encontrar, que, sendo a paragem de partida, 24, uma paragem sem abrigo e sem publicidade, de imediato se torna impossível realizer um percurso apenas passando em paragens abrigadas ou realizer um percurso apenas passando por paragens com publicidade.

```
| ?- trajetoLimProf(224,226,5,R).
Percurso no formato:
[(Carreira,Origem,Destino,Distancia)]
R = [(6,224,_A,8736)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(2,238,_A,9596)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(13,238,46,11587),(13,46,_A,10810)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(13,238,46,11587),(15,46,_A,10810)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(15,238,46,11587),(13,46,_A,10810)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(15,238,46,11587),(15,46,_A,10810)] ? ;
R = [(2,224,239,9686),(13,239,238,10469),(2,238,_A,9596)] ? ;
R = [(2,224,239,9686),(13,239,238,10469),(13,238,46,11587),(13,46,_A,10810)] ?
yes
| ?- trajetoPtosInt(224,226,[46],R).
Percurso no formato:
[(Carreira,Origem,Destino,Distancia)]
R = [(2,224,239,9686),(2,239,238,10469),(13,238,46,11587),(13,46,_A,10810)] ? ;
R = [(2,224,239,9686),(2,239,238,10469),(13,238,46,11587),(15,46,_A,10810)] ?
```

Figura 25: Percurso passando por pontos intermédios

Na figura 25 podemos verificar como requesitando um percurso que passe numa lista de pontos intermédios alguns percursos que não passavam por esses pontos(46 neste caso), são ignorados e a procura continua até encontrar percursos que passem nos pontos pedidos.

```
yes
| ?- trajetoMaisRap(224,226,R,D).
Percurso no formato:
[(Carreira,Origem,Destino,Distancia)]
Prolog interruption (h for help)? a
% Execution aborted
| ?- gulosa(224,226,R).
Percurso no formato:
[(Carreira,Origem,Destino,Distancia)]
R = [(6,224,_A,8736)] ? ;
no
| ?- aestrela(224,226,R).
Percurso no formato:
[(Carreira,Origem,Destino,Distancia)]
R = [(6,224,_A,8736)] ?
yes
```

Figura 26: Procura trajeto menor distância.

Na figura 26 podemos demonstrar como a primeira estratégia utilizada para pesquisa do caminho mais curto era limitada, ora, a necessidade de percorrer todos os caminhos para depois calcular o mais curto torna a complexidade desta pesquisa imensa, e torna bastante dificil de processar o pedido em várias circunstancias. Pelo contrário, as pesquisas informadas, gulosa e aestrela, conseguem rapidamente identificar uma solução, que é ótima para ambas neste caso mas podia não ser para a pesquisa gulosa.

8. Conclusão

Entre os vários objetivos deste projeto foi possível praticar de forma eficaz e profunda a utilização da Programação em Lógica, mais concretamente da linguagem de programação PROLOG. Além disso permitiu praticar métodos de Resolução de Problemas e o desenvolvimento de algoritmos de pesquisa.

Todos os requisitos iniciais para o caso prático foram concretizados com sucesso, embora alguns estejam ainda sujeitos a melhoria com a utilização de algoritmos de pesquisa mais eficazes.

Ao longo do desenvolvimento do projeto foi possível concretizar o objetivo de ponderar sobre quais algoritmos de pesquisa utilizar e realizar comparações entre eles, obtendo assim um maior grau de conhecimento sobre os mesmos e a possibilidade de realizar opções mais eficazes e de forma mais rápida quando necessitar de utilizar estratégias de pesquisa.

Um próximo passo neste projeto seria a utilização de estratégias de pesquisa informada em algumas das funcionalidades que apenas foram implementadas com recurso a pesquisas não informadas numa fase inicial, tendo em conta as implementações já realizadas a sua implementação seria muito simples e irá melhorar consideravelmente o desempenho geral do sistema. De imediato estas estratégias permitem descobrir o caminho mais curto, pelo que respondem ao problema principal deste sistema sem qualquer alteração ao estado atual, quanto aos restantes problemas apenas se trata de pequenos filtros na pesquisa.