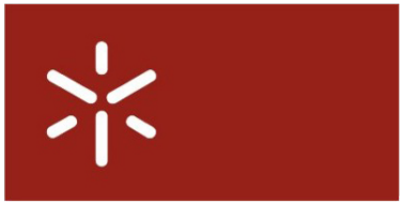

SRCR

Mestrado Integrado em Engenharia Informática

Trabalho Individual

Resolução de Problemas de Procura

A85729 Paulo Jorge da Silva Araújo

Braga, 03 de Junho de 2020

Resumo

Este trabalho desenvolvido incidiu sobre a programação em lógica estendida e sobre os métodos de resolução de problemas de procura através do **PROLOG**. Por isso era pretendido desenvolver conhecimento em relação aos nodos e arcos de uma rede, neste caso particular em relação à rede de transportes do concelho de Oeiras.

Irei portanto ao longo deste relatório descrever e demonstrar os métodos utilizados que me permitiram armazenar o conhecimento e manipular os dados de forma coerente, de forma a atingir os objetivos pretendidos.

Em suma, penso que tenha atingido os objetivos pretendidos para este trabalho, o que me permitiu uma maior assimilação de conhecimento referente à programação estendida em lógica e ao métodos de resolução de algoritmos de procura.

Conteúdo

Capítulo 1

Introdução

Este trabalho tinha como incentivo o aprofundamento de conhecimentos relativos à lógica estendida em **PROLOG** e a definição e manipulação de algoritmos de pesquisa capazes de trabalhar sobre uma amostra real, algoritmos estes que são fundamentais para o desenvolvimento e resolução de problemas de desempenho. Uma forma de complementar ainda mais os conteúdos leccionados ao longo deste semestre relativamente a este tema.

Foi proposto para amostra de dados a rede de transportes do concelho de Oeiras. Irá portanto ser necessário organizar estes dados em formas de conhecimento, para posteriormente serem manipulados consoante os objetivos pretendidos.

Capítulo 2

Preliminares

Primeiramente foi necessário proceder a uma análise cuidadosa dos dados fornecidos para perceber a melhor abordagem a este tipo de problema. Serviram assim como base os dois ficheiros de dados fornecidos pelos docentes da unidade curricular, "Lista de adjacência"e "Dataset: Paragens de Autocarro do concelho de Oeiras". Ficheiros estes aos quais foi aplicado um analisador em *java* que permitiu ao *prolog* trabalhar com estes dados sobre a forma de conhecimento. Feito esta análise o conhecimento resume-se a dois termos compostos "paragem"e "arco", que serão exploradas de forma mais aprofundada ao longo deste relatório.

2.1 Definições relativas ao sistema

Destacamos agora os termos compostos que utilizamos como abordagem para a resolução deste trabalho.

- **paragem**

  O termo composto paragem representa uma paragem física propriamente dita, portanto este termo alberga todos os atributos que eu pensei necessários para a representação do conhecimento de uma paragem.

  Uma paragem é assim identificada pelo seu identificador único (gid), a latitude, a longitude, o estado de conservação, o tipo de abrigo, se é abrigo com publicidade, e a operadora. De realçar que eu não achei pertinente armazenar o conhecimento relativamente às suas carreiras, código de rua e nome da rua visto não ser necessária uma análise posterior destes dados.

```
paragem: #Gid, Latitude, Longitude, Estado,
         Abrigo, Publicidade, Operadora
```

- **arco**

  O termo composto arco representa a "ligação"entre duas paragens associado a uma determinada carreira, ou seja, para conseguirmos fazer um percurso de um nodo inicial para um nodo final temos de ver se existem arcos que liguem estes nodos, quer seja diretamente, quer seja com mais nodos intermédios.

Um arco é assim identificado pelo seu nodo inicial, ou seja paragem inicial, pela carreira associada e pelo nodo final, paragem final respetivamente. Informação pertinente para a representação de um arco ao qual podem ser aplicáveis algoritmos de pesquisa. De salientar que dois nodos inicial e final podem conter carreiras diferentes a ligá-los.

```
paragem: NodoInicial, Carreira, NodoFinal
```

2.2 Definição de Algoritmos de Pesquisa

São apresentados agora os algoritmos de pesquisa pelos quais guiei a resolução dos problemas, para tal foi necessário recorrer à pesquisa de informação para a definição dos mesmos. Não me foi possível testar de forma aprofundada estes algoritmos visto que por si só o *prolog* é um pouco limitador quanto à performance e capacidade de análise de grandes conjuntos de dados.

2.2.1 Pesquisa em Profundidade

Intuitivamente, o algoritmo começa num nodo selecionando e explora tanto quanto possível cada um dos seus arcos, antes de retroceder, ou seja realizar o backtracking.

É ainda passado como argumento uma amostra de nodos sobre a qual o algoritmo pode fazer a procura de uma solução.

Sendo assim é necessário definir a função de adjacência de nodos que basicamente se resume à verificação da existência de um "arco"entre o nodo atual e o próximo nodo e a existência deste nodos na amostra.

```
adjacenteProfundidade(Nodo, ProxNodo, Amostra) :-
        arco(Nodo,_,ProxNodo),
        membro(Nodo,Amostra),
        membro(ProxNodo,Amostra).
```

A definição do algoritmo em si passa apenas pela aplicação recursiva desde o seu nodo inicial até ao nodo final, passado pelos nodos intermédios existentes. Sendo assim tem como argumentos, o nodo inicial, a solução/caminho encontrado para os nós dados, o nodo final e a amostra.

```
resolve_pp(NodoInicial, [NodoInicial|Caminho], NodoFinal, Amostra) :-
     profundidadeprimeiro(NodoInicial, Caminho, NodoFinal, Amostra).
```

2.2.2 Pesquisa em Largura

Aqui os nodos em cada nível da árvore são completamente examinados antes de se mover para o próximo nodo adjacente. Um pouco mais rebuscada do que a definição da pesquisa em profundidade visto que é necessário manter um conjunto de nodos candidatos alternativos e não apenas um nodo candidato. Mesmo assim este conjunto de caminhos candidatos não é suficiente se é preciso extrair um caminho solução. Portanto, ao contrário de se manter um conjunto de nodos candidatos, mantém-se um conjunto de caminhos candidatos.

Tal como na pesquisa em profundidade também é passado como argumento uma amostra de nodos sobre a qual o algoritmo deve trabalhar. Como podemos observar no código apresentado

em seguida o primeiro argumento do predicado "larguraprimeiro"contém uma lista de listas, que representa todos os caminhos candidatos. No final é aplicado o predicado inverso ao caminho para este nos ser apresentado desde o nodo inicial até ao final.

```
resolve_lp(NodoInicial, Caminho, NodoFinal, Amostra) :-
     larguraprimeiro([[NodoInicial]], CaminhoInverso, NodoFinal, Amostra),
     inverso(CaminhoInverso, Caminho).
```

2.2.3 A estrela

Neste algoritmo é utilizada uma heurística que estima a distância entre dois nodos através da sua latitude e longitude. Ou seja este algoritmo tem sempre presente uma lista de nodos candidatos e uma lista de nodos visitados, para evitar ciclos. Basicamente os nodos são retirados da lista de candidatos e visitados através da aplicação do predicado heurística que nos dá o nodo ótimo naquele momento da execução.

Como nos dois algoritmos também é passada como argumento a lista amostra de nodos.

```
resolve_aestrela(Nodo, Caminho/Custo, NodoFinal, Amostra) :-
euristica(Nodo, NodoFinal, Estima),
aestrela([[Nodo]/0/Estima], InvCaminho/Custo/_, NodoFinal, Amostra),
inverso(InvCaminho, Caminho).
```

Capítulo 3

Descrição do Trabalho e Análise de Resultados

3.1 Povoamento de conhecimento no sistema

Como já foi referido anteriormente o povoamento de conhecimento no sistema é feito através de um analisador, em java, através dos ficheiros de dados fornecidos.

Achei pertinente utilizar apenas o ficheiro de dados "Lista de adjacência"por ser aquele que apresenta os dados de forma mais estruturada e me ser possível analisar logo as ligações entre paragens e a informação das próprias paragens.

Foi portanto necessário criar vários ficheiro csv um por cada "carreira"para depois serem posteriormente lidos pelo *parser*.

Relativamente ao **parser** em si são aplicadas funções de leitura de ficheiros de texto. Ou seja para cada linha do ficheiro que continha:

183;-103678.36;-96590.26;Bom;Fechado dos Lados;Yes;Vimeca;01;286;Rua Aquilino Ribeiro;Carnaxide e Queijas

foi feito *split* da informação e escrito nos ficheiros destino "paragens.pl"e "arcos.pl".

Para as paragens foi necessário aplicar uma abordagem para evitar repetir a escrita da mesma paragem mais que uma vez no ficheiro "paragens.pl". Criou-se assim um conjunto de dados(mais propriamente um *hashset*) que armazenava o "gid"das paragens já inserida, portanto antes da inserção de uma paragem no destino era verificado a existência do seu "gid"neste conjunto. Uma paragem seria então escrita nesta forma padrão no ficheiro "paragens.pl":

paragem(183,-103678.36,-96590.26, 'Bom' , 'Fechado dos Lados' , 'Yes' , 'Vimeca').

Já para os arcos considerei que duas linhas seguidas do ficheiro de dados corresponderiam a dois nodos que continham uma ligação entre eles, e portanto seria criado o arco com o nodo inicial e o nodo final e a carreira associada. Um arco seria então escrito nesta forma padrão no ficheiro "arcos.pl":

arco(183,01,791).

Posto isto temos agora os ficheiros "paragens.pl"e "arcos.pl"com todo o conhecimento necessário para a resolução dos problemas propostos.

3.2 Predicados Auxiliares

Durante o trabalho foram utilizadas vários predicados com o intuito de auxiliar e simplificar outros, tais como *insertionSort*, *inter*, *inverso*, entre outros. Para tal foi criado um ficheiro que contém todos estes predicados "predAux.pl". Não vou proceder à explicação dos predicados um a um porque penso que seja algo secundário e de importância menor, queria apenas referir a existência dos mesmos e que estes se encontram devidademente documentados.

3.3 Funcionalidades do sistema

Dependendo do problema apresentado fui aplicando diferente pesquisas, consoante aquela que achava mais pertinente e que a meu ver seria aquela capaz de solucionar o mesmo de forma rápida e eficiente. Para cada funcionalidade irei dar uma breve descrição da implementação e um teste que achei pertinente para análise da mesma, de salientar que os testes estão um pouco limitados à capacidade da máquina e às limitações apresentadas pelo prolog que na maioria das vezes não permitia solucionar percursos com uma grande quantidade de nodos intermédios independentemente da pesquisa.

3.3.1 Calcular um trajeto entre dois pontos

Nesta funcionalidade poderia ser adoptada qualquer tipo de pesquisa para encontrar o percurso entre dois nodos, para efeitos de teste irei usar a pesquisa em profundidade.

```
percurso(NodoInicial, NodoFinal, Caminho) :-
    findall(A, paragem(A,B,C,D,E,F,X), P1),
     resolve_pp(NodoInicial, Caminho, NodoFinal,P1).
```

Para efeitos de teste:

```
| ?- percurso(863,79,D), print(D).
[863,856,857,367,333,846,845,330,364,33,32,60,61,64,63,62,58,57,59,654,78,79]
D = [863,856,857,367,333,846,845,330,364,33|...] ?
```

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

Aqui foi necessário primeiro encontrar todas as paragens que tivessem as operadoras dadas como argumentos, para em seguida esta lista ser passada como o argumento amostra para a pesquisa.

```
comOperadorasParagens([], Acc, Acc).
comOperadorasParagens([X|Cauda], Acc, Paragens) :-
    findall(A, paragem(A,B,C,D,E,F,X), P1),
    append(P1, Acc, R),
    comOperadorasParagens(Cauda, R, Paragens).
```

Em seguida apenas seria necessário fazer a chamada da pesquisa já com a amostra certo e fazer a procura do percurso entre os nodos.

```
comOperadoras(Operadoras,NodoInicial,NodoFinal, Caminho) :-
comOperadorasParagens(Operadoras,[],Paragens),
resolve_pp(NodoInicial, Caminho, NodoFinal,Paragens).
```

Para efeitos de teste:

```
| ?- comOperadoras(['Vimeca','LT'],863,79,C), print(C).
[863,856,857,367,333,846,845,330,364,33,32,60,61,64,63,62,58,57,59,654,78,79]
C = [863,856,857,367,333,846,845,330,364,33|...] ?
| ?- comOperadoras(['Carris'],863,79,C).
no
```

3.3.3 Excluir um ou mais operadores de transporte para o percurso

Similar à funcionalidade apresentada anteriormente é necessário encontrar primeiro a amostra de nodos que não contém as operadoras dadas como argumento.

```
noOperadorasParagens([],Acc,Acc).
noOperadorasParagens([X|Cauda],Acc,Paragens) :-
    findall(A,(paragem(A,B,C,D,E,F,Op), Op \= X), P1),
    inter(P1, Acc, Int),
    noOperadorasParagens(Cauda, Int, Paragens).
```

Sendo assim só falta chamar o predicado de pesquisa com a amostra calculada.

```
noOperadoras(Operadoras,NodoInicial,NodoFinal, Caminho) :-
findall(A,paragem(A,B,C,D,E,F,G),P1),
noOperadorasParagens(Operadoras,P1,Paragens),!,
resolve_lp(NodoInicial, Caminho, NodoFinal,Paragens).
```

Para efeitos de teste:

```
| ?- noOperadoras(['Carris'],863,79,C),print(C).
[863,856,857,367,333,846,845,330,364,33,32,60,61,64,63,62,58,57,59,654,78,79]
C = [863,856,857,367,333,846,845,330,364,33|...] ?
| ?- noOperadoras(['Vimeca','LT'],183,791,C).
no
```

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

Para tal foi construído um predicado auxiliar que nos permite olhar para uma lista de nodos/pargens e transformar esta lista numa nova lista que contém, por ordem decrescente de número de carreiras, um par (nodo, número de carreiras).

```
carreirasParagem([], Acc, Final) :-
    insertionSort(Acc,InvOrd),
    inverso(InvOrd,Final).

carreirasParagem([Nodo|Cauda], Acc, Final) :-
    findall(Nodo,arco(Nodo,_, _), ParagensCarreiras),
    comprimento(ParagensCarreiras,N),
    append([(Nodo,N)],Acc,Int),
    carreirasParagem(Cauda, Int, Final).
```

No predicado principal bastaria portanto fazer a chamada do predicado da pesquisa para encontrar um caminho possível e em seguida fazer a chamada do predicado auxiliar para dar a lista ordenada com o maior número de carreiras.

```
maisCarreirasPercurso(NodoInicial, NodoFinal, Ordenado) :-
    findall(A, paragem(A,B,C,D,E,F,X), P1),
    resolve_lp(NodoInicial, MaisCarreiras, NodoFinal,P1),
    carreirasParagem(MaisCarreiras,[],Ordenado).
```

Para efeitos de teste:

```
| ?- maisCarreirasPercurso(183,595,C).
[(595,7),(183,6),(791,6)]
C = [(595,7),(183,6),(791,6)] ?
```

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

Para esta funcionalidade a estratégia adotada passou por primeiramente calcular todos os percursos possíveis para os nodos dados como argumentos, para tal foi utilizado o predicado *todosPercursos* que consiste em fazer *findall* de todas as soluções. Em seguida é utilizada outro predicado auxiliar que olha para uma lista de lista e retorna a lista de menor comprimento, o que no nosso caso corresponde ao caminho com menos paragens.

```
menorPercursoParagens(NodoInicial, NodoFinal, Caminho) :-
    todosPercursos(NodoInicial, NodoFinal, Solucoes),
    headLista(Solucoes,Head),
    comprimento(Head,N),
    listaMenor(Solucoes, (Head,N), Caminho).
```

Para efeitos de teste:

```
| ?- menorPercursoParagens(183,595,C).
C = [183,791,595] ?
yes
```

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

Aqui como já tinha sido feito a implementação da pesquisa A-estrela com a heurística baseada na distância apenas foi necessário aplicá-la. Apenas fazemos primeiro um *findall* para passar todos os nodos/paragens possíveis para a amostra do predicado pesquisa.

```
menorPercursoDistancia(NodoInicial, NodoFinal, Caminho, Custo) :-
    findall(A, paragem(A,B,C,D,E,F,X), P1),
    resolve_aestrela(NodoInicial, Caminho/Custo, NodoFinal, P1).
```

Para efeitos de teste:

```
| ?- menorPercursoDistancia(354,79,C,D), print(C).
[354,353,863,856,857,367,333,846,845,330,364,33,32,60,61,64,
63,62,58,57,59,654,78,79]
C = [354,353,863,856,857,367,333,846,845,330|...],
D = 178.36634513986803 ?
```

3.3.7 Escolher o percurso que passe apenas por abrigos com publicidade

Neste problema foi primeiro necessário calcular a amostra sobre a qual íamos trabalhar, ou seja encontrar todos os nodos/paragens com publicidade para em seguida passar como argumento no predicado de pesquisa.

```
publicitadas(Paragens) :-
    findall(A, paragem(A,B,C,D,E,'Yes',G), Paragens).
```

Feito isto bastava apenas chamar o nosso algoritmo de pesquisa com a amostra calculada.

```
percursoPublicitado(NodoInicial, NodoFinal, Caminho) :-
    publicitadas(Paragens),
    resolve_pp(NodoInicial, Caminho, NodoFinal,Paragens).
```

Para efeitos de teste:

```
| ?- percursoPublicitado(353,333,C).
C = [353,863,856,857,367,333] ?
yes
| ?- percursoPublicitado(353,846,C).
no
```

3.3.8 Escolher o percurso que passe apenas por paragens abrigadas

A ideia por detrás deste problema é análogo ao apresentado ao anterior mas agora queremos que a nossa amostra apenas apresente nodos/paragens abrigadas, ou seja que não contenham o campo tipo de abrigo com esta entrada: 'Sem Abrigo'.

```
abrigadas(Paragens) :-
    findall(A,(paragem(A,B,C,D,E,F,G), E \= 'Sem Abrigo'), Paragens).
```

E em seguida aplica-se a pesquisa para esta nova amostra.

```
percursoAbrigado(NodoInicial, NodoFinal, Caminho) :-
    abrigadas(Paragens),
    resolve_lp(NodoInicial, Caminho, NodoFinal,Paragens).
```

Para efeitos de teste:

```
| ?- percursoAbrigado(863,32,C), print(C).
[863,856,857,367,333,846,845,330,364,33,32]
C = [863,856,857,367,333,846,845,330,364,33|...] ? yes
| ?- percursoAbrigado(863,60,C).
no
```

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

Nesta última funcionalidade foi pedido um percurso entre dois nodos que tinha de passar de forma obrigatória num conjunto de nodos. A estratégia utilizada para a resolução deste problema foi encontrar todos os caminhos possíveis entre o nodo inicial e o nodo final. Depois seria apenas necessário verificar se dos caminhos encontrados pelo menos um continha todos os nodos intermédios obrigatórios.

```
percursoComPontos(Pontos,NodoInicial,NodoFinal) :-
    todosPercursos(NodoInicial, NodoFinal, Solucoes),
    temNaListaListas(Pontos,Solucoes).
```

Para efeitos de teste:

```
| ?- percursoComPontos([78,364,33,61,64,58],863,79).
yes
| ?- percursoComPontos([3000],863,79).
no
```

Capítulo 4

Conclusões e Sugestões

Em suma neste trablho foi-me possível colocar os conhecimentos adquiridos ao longo deste semestre relativamente à lógica estendida em *prolog* e aos métodos de resolução de problemas de procura. Foi ainda possível aplicar os próprios predicados de procura realizados nas aulas fundamentais para a resolução deste trabalho. Assim sendo penso que tenha realizado todos os objetivos pretendidos para este trabalho individual, apesar de ao longo do mesmo ter encontrado vários obstáculos os quais penso ter ultrapassado com um certo nível de competência. Volto a frisar o facto de a linguagem utilizada *prolog* ser um pouco limitante no poder computacional, e portanto não me ser possível a realização de grandes testes aos meus predicados, tais como a procura por caminhos que contenha grande número de nodos intermédios. De salientar ainda que o relatório ficou um pouco extenso pelo facto de eu achar necessário descrever os métodos e a forma de pensar por mim utilizada ao pormenor, para melhor compreensão por parte dos leitores. Sendo assim penso que sou agora capaz de no futuro abordar problemas idênticos a este e com maior grau de dificuldade visto que desenvolvi novas capacidades de raciocínio e aprofundei o meu conhecimento relativamente à utilização de algoritmos de procura.

Referências

[1] Russell and Norvig(2009). Artificial Intelligence -A Modern Approach, 3rd edition, ISBN-13: 9780136042594.

[2] Ivan Bratko (2000), PROLOG: Programming for Artificial Intelligence, 3rd Edition, Addison-Wesley Longman Publishing Co., Inc.