Universidade do Minho

Mestrado Integrado em Engenharia Informática

Sistemas de Representação de Conhecimento e Raciocínio

Trabalho Individual

Daniela Fernandes
A73768

8 de junho de 2021

Resumo

Este documento diz respeito ao Trabalho Prático Individual proposto na unidade curricular de Sistemas de Representação de Conhecimento e Raciocínio.

Tem como objetivo a utlização da Linguagem de Programção em Lógica, utilizando o PROLOG, para o desenvolvimento de métodos para a resolução de problemas e para algoritmos de pesquisa.

Ao longo deste relatório, vamos apresentar as funcionalidades do sistema desenvolvido e as estratégias que utilizamos para a resolução do problema proposto.

Conteúdo

1 Introdução

Na unidade curricular de Sistema de Representação de Conhecimento, foi-nos proposto a resolução de um trabalho prático individual que tem como base de estudo um *dataset* com dados dos circuitos de recolha de resíduos urbanos do concelho de Lisboa.

Para a resolução deste problema, comecei por desenvolver um sistema para a importar os dados relativos aos pontos de recolha onde se encontram os contentores de determinado resíduo e representar esses dados numa base de conhecimento que vamos utilizar. Posteriormente com essa base de conhecimento, desenvolvemos um sistema de recomendação de circuitos.

No enunciado foi proposto que a eloboração deste caso, permitisse:

- Gerar os circuitos de recolha tanto indiferenciada como seletiva, caso existam, que cubram um determinado território;
- Identificar quais os circuitos com mais pontos de recolha (por tipo de resíduo a recolher);
- Comparar circuitos de recolha tendo em conta os indicadores de produtividade;
- Escolher o circuito mais rápido (usando o critério da distância);
- Escolher o circuito mais eficiente (usando um critério de eficiência à escolha);

2 Preliminares

Ao contrário do primeiro projeto desenvolvido para esta cadeira, o sistema com que vamos trabalhar não necessita obrigatoriamente de ser compatível com lógica extendida. Com isto, eu quero dizer que não iremos trabalhar com o "desconhecido", uma vez que todos os parâmetros dos vários predicados são sempre conhecidos.

Pressuposto dos Nomes Únicos: duas constantes diferentes referem-se a duas entidades diferentes.

Pressuposto do Mundo Fechado: todas as afirmações são verdadeiras ou falsas, e nós sabemos se são, respetivamente, verdadeiras ou falsas.

Pressuposto do Domínio Fechado: no universo do discurso apenas existem os objetos que nos são conhecidos.

Apesar da implementação do Conhecimento Desconhecido não ser impossível neste contexto, acredito que a implementação de tal não seria necessária. No universo de discurso, nós sabemos todos os elementos que caracterizam uma cidade. Como tal, não existe informação incompleta a retratar neste sistema, o que torna a implementação de tal desnecessária.

3 Predicados de dados

Para a resolução deste problema resolvi criar três tipos de dados que achei que iriam ser importantes para a resolução dos problemas.

Com o parser criei três ficheiros diferentes que depois manualmente juntei num outro com os três tipos de dados que mais me interessaram:

1. ecoponto: neste tipo de dado está definido pela latitude, longitude, código da rua, nome da rua, tipo de lixo.

2. aresta : neste tipo de dado está definido pelo nome da rua, a sua rua adjacente e a distância entre elas.

3. ponto de recolha : neste tipo está o código da rua, o nome da rua, a sua latitude e longitude.

Apesar ter escolhido inicialmente utilizar este três dados, ao longo da realização do trabalho e pelo seu nível de complexidade decidi utilizar apenas a **aresta**.

Com este tipo de dado já consegui responder às quatro alíneas que nos foram propostas. Para a realização escolhi algumas arestas para conseguir testar o resultado final que pretendia obter com os algoritmos.

3.1 Parsing

Foi-nos fornecido um *excel* com os dados que seriam utilizados para o nosso problema. Para poder importar esses dados, criei um *parser* em python.

No *parser* foram criados 3 tipos de ficheiros, os que tinham os dados dos ecopontos, os que tinham os dados das arestas e os que tinham os dados do pornto de recolha.

Ao ler o ficheiro *.xlsx* linha a linha vai retirar as informações que nós consideramos necessárias.

Depois de o parser ser criados, manualmente criei um novo ficheiro *final.txt* que contém todos as informações sobre cada tipo de dados.

3.2 Arestas

Como foi dito anteriormente, optei por utilizar apenas as arestas para a realização do trabalho.

Para as arestas o que vamos ter em conta é a rua em que começa o percurso, a sua rua adjacente e a distância que existe entre elas.

Com estes dados, consegui elaborar a versão mais símplificada do projeto. Isto é, apenas vamos utilizar a distância para a elaboração dos algoritmos que nos foram ensinados sobre Métodos de Resolução de Problemas e de Procura.

4 Descrição do problema e Resolução

4.1 Algoritmos de Pesquisa

Para a resolução do problema formulado anteriormente foi, como já referido, necessária a criação de algoritmos de pesquisa.

4.1.1 Depth First Search (Em profundidade)

Primeiramente, preocupei-me em criar os algoritmos de pesquisa não informada, começando por um algoritmo de pesquisa em profundida, sendo que defini então o seguinte predicado:

```
dfs(Nodo, Destino, [Nodo|Caminho]):-
    dfsAux(Nodo, Destino, [Nodo], Caminho).

dfsAux(Nodo, Destino, _, [Destino]):-
    adjacente(Nodo, Destino).

dfsAux(Nodo, Destino, Visited, [ProxNodo|Caminho]):-
    adjacente(Nodo, ProxNodo),
    \+ member(ProxNodo, Visited),
    dfsAux(ProxNodo, Destino, [Nodo|Visited], Caminho).
```

Este algoritmo escolhe o primeiro elemento e vai percorrer uma lista dos nodos que lhe são adjacentes e visita-os.

4.1.2 Breadth First Search (Em largura)

Para a pesquisa em largura foi necess ´ario o seguinte predicado. Este predicado não consigui utilizar para nenhumas das queries.

```
largura(Origem,Destino,Caminho) :-
    bfs1([[Origem]],Destino,Caminho),
    reverse(CaminhoAux,Caminho).

bfs1([[Destino|Caminho]|_],Destino,[Destino|Caminho]).

bfs1([[Caminho1|Caminhos]|_],Destino,Caminho) :-
    extende(Caminho1, NovosCaminhos),
    append(Caminhos, NovosCaminhos, Caminhos1),
    bfs1(Caminhos1,Destino,Caminho).

extende(_,[]).

extende([Paragem|Caminho], NovosCaminhos) :-
```

```
    findall([Paragem2, Paragem|Caminho], (adjacente(Paragem,Paragem2), \+
    memberchk(Paragem2,[Paragem|Caminho])),NovosCaminhos),!.
```

4.1.3 Gulosa

Após a pesquisa não informada, foi necessária elaboração dos algoritmos de pesquisa informada de acordo coma distância, sendo que comecei por definir o seguinte predicado:

```
solucaoGulosa(Nodo, Caminho/Custo) :-
    estimativa(Nodo, Estimativa),
    agulosa([[Nodo]/0/Estimativa], CaminhoInverso/Custo/_),
    inverso(CaminhoInverso, Caminho).

gulosa(Caminhos, Caminho) :-
    melhorG(Caminhos, Caminho),
    Caminho = [Nodo|_]/_/_,fim(Nodo).

gulosa(Caminhos, SolucaoCaminho) :-
    melhorG(Caminhos, MelhorCaminho),
    escolhe(MelhorCaminho, Caminhos, OutrosCaminhos),
    expandeG(MelhorCaminho, ExpCaminhos),
    append(OutrosCaminhos, ExpCaminhos, NovoCaminhos),
    gulosa(NovoCaminhos, SolucaoCaminho).

melhorG([Caminho], Caminho) :- !.

melhorG([Caminho1/Custo1/Est1,_/Custo2/Est2|Caminhos], MelhorCaminho) :-
    Est1 =< Est2, !,
    melhorG([Caminho1/Custo1/Est1|Caminhos], MelhorCaminho).

melhorG([_|Caminhos], MelhorCaminho) :-
    melhorG(Caminhos, MelhorCaminho).

expandeG(Caminho, ExpCaminhos) :-
    findall(NovoCaminho, gulosaAdj(Caminho,NovoCaminho), ExpCaminhos).

gulosaAdj([Nodo|Caminho]/Custo/_, [ProxNodo,Nodo|Caminho]/NovoCusto/Est) :-
    nodo(Nodo, ProxNodo, PassoCusto),\+ member(ProxNodo, Caminho),
    NovoCusto is Custo + PassoCusto,
    estimativa(ProxNodo, Est).
```

4.1.4 A estrela(A*)

Por fim, passei para outro algoritmo de pesquisa informada de acordo com a distância, que se traduz pelo seguinte predicado:

```
solucaoEstrela(Nodo, Caminho/Custo) :-
    nodo(Nodo,_,Estimativa),
    estrela([[Nodo]/0/Estima], CaminhoInverso/Custo/_),
    inverso(CaminhoInverso, Caminho).

estrela(Caminhos, Caminho) :-
    melhorE(Caminhos, Caminho),
    Caminho = [Nodo|_]/_/_,fim(Nodo).

estrela(Caminhos, SolucaoCaminho) :-
    melhorE(Caminhos, MelhorCaminho),
    escolhe(MelhorCaminho, Caminhos, OutrosCaminhos),
    expandeEstrela(MelhorCaminho, ExpCaminhos),
    append(OutrosCaminhos, ExpCaminhos, NovoCaminhos),
    estrela(NovoCaminhos, SolucaoCaminho).

melhorE([Caminho], Caminho) :- !.

melhorE([Caminho1/Custo1/Est1,_/Custo2/Est2|Caminhos], MelhorCaminho) :-
    Custo1 + Est1 =< Custo2 + Est2, !,
    melhorE([Caminho1/Custo1/Est1|Caminhos], MelhorCaminho).

melhorE([_|Caminhos], MelhorCaminho) :-
    melhorE(Caminhos, MelhorCaminho).

expandeE(Caminho, ExpCaminhos) :-
findall(NovoCaminho, gulosaAdj(Caminho,NovoCaminho), ExpCaminhos).
```

4.1.5 Tabela Comparativa

Tabela 1: Tabela comparativa

Algoritmo	Completo	Complexidade Temporal	Complexidade Espacial
Depth-First	Não	$O(b^d)$	$O(b^d)$
Gulosa	Sim	$O(b^d)$	$O(b^d)$
A*	Sim	$O(b^d)$	$O(b^d)$

4.2 Queries

Chegando a esta fase do projeto, vou proceder à explicitação das estratégias utilizadas para a elaboração das *queries*.

4.2.1 melhor/3 (DF)

O melhor: Nodo, Cam, Custo -> V,F calcula o percurso que percorra menos metros. Este predicado é um predicado auxilar para o algoritmo em profundidade.

```
melhor(Nodo,Cam,Custo):- findall((Ca,Cus), porCusto(Nodo,Ca,Cus), L),minimo(L
    ,(Cam,Custo)).
```

4.2.2 pior/3 (DF)

O pior: Nodo, Cam, Custo -> V,F calcula o percurso que percorra mais metros. Este predicado é um predicado auxilar para o algoritmo em profundidade.

```
pior(Nodo,Cam,Custo):- findall((Ca,Cus), porCusto(Nodo,Ca,Cus), L),maximo(L,(
    Cam,Custo)).
```

4.2.3 maisPontos/1 (DF)

O maisPontos: L-> V,F calcula o percurso que percorra mais nodos. Este predicado é um predicado auxilar para o algoritmo em profundidade.

```
maisPontos(L):- todasSol(R),
                longest(R,L).
```

4.2.4 caminhoDepthFirst/2 (DF)

O caminhoDepthFirst: Inicio, Fim -> V, F é responsável por calcular o trajeto entre dois pontos. Escolhi o algoritmo depth first para esta query, uma vez que acho que é o ideal para quando o problema tem muitas soluções e usada muito pouca memória.

```
caminhoDepthFirst(Origem, Destino) :-
    statistics(walltime, [_|T1]),
    dfs(Origem,Destino,R),
    statistics(walltime, [_|T2]),
    pathCost(R, Custo),
    printCost(Custo),
    printTime(T2).
```

4.2.5 custoTodasSolucoesDF/1 (DF)

O custoTodasSolucoes: Lista -> V,F é responsável por imprimir o custo total de todos os percursos possíveis.

```
custoTodasSolucoesDF(L) :-
    statistics(walltime, [_|T1]),
    todasSolCusto(R),
    statistics(walltime, [_|T2]),
    write(R),
    printTime(T2).
```

4.2.6 caminhoMaisCurtoDF/1 (DF)

O caminhoMaisCurto: Origem -> V,F vai imprimir qual é o percurso mais curto, utilizando o predicado desenvolvido anteriormente *melhor*.

```
caminhoMaisCurtoDF(Origem) :-
    statistics(walltime, [_|T1]),
    melhor(Origem,Cam,Custo),
    statistics(walltime, [_|T2]),
    write(Cam),
    write(Custo),
    printTime(T2).
```

4.2.7 caminhoMaisLongoDF/1 (DF)

O caminhoMaisCurto: Origem -> V,F vai imprimir qual é o percurso mais longo, utilizando o predicado desenvolvido anteriormente *pior*.

```
caminhoMaisCurtoDF(Origem) :-
    statistics(walltime, [_|T1]),
    melhor(Origem,Cam,Custo),
    statistics(walltime, [_|T2]),
    write(Cam),
    write(Custo),
    printTime(T2).
```

4.2.8 caminhoComMaisPontosRecolhaDF/1 (DF)

O caminhoMaisCurto: Origem -> V,F vai imprimir qual é o percurso com mais pontos de recolha, utilizando o predicado desenvolvido anteriormente *maisPontos*.

```
caminhoComMaisPontosRecolhaDF(R) :-
    statistics(walltime, [_|T1]),
    maisPontos(L),
    statistics(walltime, [_|T2]),
    write(L),
    printTime(T2).
```

4.2.9 gulosaa/1

O gulosaa : Origem -> V,F vai imprimir utilizando o algoritmo desenvolvido anteriormente, sobre o percurso em que o seu custo é o mínimo não passando por todos os pontos.

```
gulosaa(Origem) :-
    statistics(walltime, [_|T1]),
    solucaoGulosa(Origem,Caminho/Custo),
    statistics(walltime, [_|T2]),
    write(Caminho),
    write(Custo),
    printTime(T2).
```

4.2.10 aEstrela/1

O aEstrela : Origem -> V,F vai imprimir o percurso de acordo com o algoritmo escolhido, evita expandir caminhos que tenham mais custo, que neste caso é a distância.

```
aEstrela(Origem) :-
    statistics(walltime, [_|T1]),
    solucaoEstrela(Origem,Caminho/Custo),
    statistics(walltime, [_|T2]),
    write(Caminho),
    write(Custo),
    printTime(T2).
```

5 Conclusão

Consegui implementar algumas funcionalidades pedidas, na maioria dos casos com mais do que um algoritmo de procura. Mais ainda, consegui arranjar uma estratégia bastante adequada para traduzir os dados, embora no final não tenha sido capaz de a utilizar.

O único problema que o meu trabalho apresenta é o facto de não ter consigo utilizar os dados que inicialmente idealizei e também porque não consegui implementar corretamente o algoritmo em largura.

Em suma, apesar de apresentar algumas falhas, acredito que fiz um bom trabalho que corresponde ao pedido e esperado.