Universidade do Minho
Escola de Engenharia

UNIVERSIDADE DO MINHO

TRABALHO INDIVIDUAL

SISTEMAS DE REPRESENTAÇÃO DE CONHECIMENTO E RACIOCÍNIO

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA

(2º SEMESTRE 20/21)

A89574 - Pedro Almeida Fernandes

Braga

Maio - 2021

RESUMO

O seguinte relatório visa abordar as temáticas lecionadas no âmbito da disciplina de Sistemas de Representação de Conhecimento e Raciocínio.

O objetivo do trabalho é realizar um sistema que apresente as rotas pelas quais os camiões do lixo devem percorrer de modo a realizarem com sucesso o seu trabalho. Para isso, foram fornecidos dados dos circuitos de recolha de resíduos urbanos do concelho de Lisboa, mais concretamente, da freguesia da Misericórdia.

Deste modo a primeira fase do trabalho passou por, a partir do *dataset* fornecido, criar uma base de conhecimento estruturada a partir de grafos, que permite albergar toda a informação necessária.

Numa segunda fase o trabalho passou por criar diversos métodos de travessia do grafo de modo a obter os caminhos pretendidos.

Numa terceira fase foram criadas diversas funcionalidades que usam as estratégias de pesquisa anteriormente realizadas, para obter informações específicas à cerca dos caminhos que foram obtidos.

Para finalizar foi realizada um extra que representa uma outra forma de ver o problema, mudando a perspetiva do que é um percurso.

ÍNDICE

ÍNDICE FIGURAS

INTRODUÇÃO

Este trabalho tem como objetivo construir uma representação do sistema de recolha de resíduos da freguesia da Misericórdia em Lisboa recorrendo a *prolog*.

Foi proposto construir um sistema que ajude no cálculo das rotas de recolha de resíduos, sendo que é suposto que as rotas em conjunto passem em todos os pontos de recolha de resíduos presentes nos dados fornecidos.

Assim sendo, foram usados diversos algoritmos, tanto de pesquisa informada como não informada, que embora alguns melhores que outros, calculem os percursos pelos quais os camiões do lixo devem seguir.

PRELIMINARES

Antes de iniciar uma leitura mais aprofundada sobre o presente relatório, é necessário ter alguma noção de alguns conceitos fundamentais para o mesmo. Assim sendo, de seguida encontram-se os tópicos necessários para a total compreensão do trabalho:

- **Linguagem *Prolog*** – todo o trabalho foi escrito nesta linguagem à exceção do *parser* dos dados fornecidos;
- **Pesquisa Não Informada** – método de pesquisa sem informações prévias;
- **Pesquisa Em Profundidade** – expande os nodos mais profundos do grafo em primeiro lugar;
- **Pesquisa Em Largura** – todos os nodos de um nível são procurados antes de passar para o próximo nível;
- **Pesquisa Limitada em Profundidade –** segue o mesmo princípio da pesquisa em profundidade, mas ao contrário dessa tem um limite até ao qual pesquisa;
- **Pesquisa Informada** – utilizam conhecimento do problema (heurística) para resolver o problema de forma mais eficaz;
- **Pesquisa Gulosa** – para escolher o próximo nodo escolhe aquele que tem uma melhor heurística;
- **Pesquisa A Estrela –** semelhante à pesquisa gulosa, mas toma em consideração não apenas a heurística, mas também o custo das arestas que vai escolher;
- **Merge Sort –** método de ordenação de listas bastante eficiente baseado no método *divide and conquer*.

DESCRIÇÃO DO TRABALHO

O trabalho está dividido em diversas secções que representam os diferentes processos pelos quais o trabalho foi passando até à sua conclusão.

A secção **pré-processamento** contém o processo de passagem dos dados do ficheiro *Excel* para um formato que permita ao sistema trabalhar de uma melhor forma toda a informação.

A secção **pesquisa não informada** contém a informação à cerca dos algoritmos de pesquisa não informada presentes no trabalho.

A secção **pesquisa informada** contém a informação de algoritmos de pesquisa informada que utilizam heurísticas para obter melhores resultados.

A secção **funcionalidades** contém todas as funcionalidades aplicadas aos percursos obtidos, obtendo apenas um percurso de acordo com a funcionalidade pretendida.

A secção **extra** contém no seu interior um algoritmo para procurar percursos completos, ou seja que passam em todos os nodos. Para além das funcionalidades anteriormente referidas foi ainda implementada uma outra para este método.

´

PRÉ-PROCESSAMENTO

Para a realização do processamento dos dados, o *dataset* fornecido foi passado para *csv* e de seguida realizada a sua organização em *java*, mais concretamente a partir do ficheiro *Parser.java*.

A estratégia passou primeiramente por retirar a informação importante relativamente a cada caixote. Para isso considerou-se relevante a coluna **Latitude**, **Longitude**, **ObjectID**, **Ponto_Recolha_LocaL** e **Contentor_Total_Litros**.

A primeira tentativa foi tentar agrupar todos os caixotes que têm a mesma localização (Latitude e Longitude), num só caixote, criando os predicados **caixote(id, lat, lon, rua, totalLixos, totalOrganico, totalPapel, totalEmbalagens, totalVidro)** indicados em seguida.

```
% caixote(id, lat, lon, rua, totalLixos, totalOrganico, totalPapel, totalEmbalagens, totalVidro)
caixote(358, -9.143309, 38.708079, 'R do Alecrim', 1860, 0, 1530, 0, 0).
caixote(368, -9.143378, 38.708078, 'R do Alecrim', 2760, 0, 460, 0, 0).
caixote(376, -9.143482, 38.707303, 'R do Alecrim', 1320, 0, 380, 0, 0).
caixote(951, -9.143292, 38.707737, 'R do Alecrim', 0, 0, 0, 90, 0).
```

Figura 1: Antigo caixote

No entanto o volume de arestas era ainda muito grande visto que todos os nodos estão tanto ligado à garagem e à lixeira. Portanto, de modo a reduzir ainda mais as arestas decidi remover todas as ruas isoladas e agrupar todos os caixotes de uma rua num único predicado caixote, sendo a sua localização a primeira que encontra da rua, gerando o seguinte resultado:

```
caixote(355, -9.143309, 38.708079, 'R do Alecrim', 5940, 0, 2370, 90, 0).
```

Figura 2: Caixote atual

Desta forma reduziu-se a Rua do Alecrim assim como as outras para apenas um nodo sendo que o *dataset* original continha inúmeras linhas :

Latitude	Longitude	OBJECTID	PONTO_RECOLHA_FRE	PONTO_RECOLHA_LOCAL	CONTENTOR_RESÍ	CONTENT	CONTENTO	CONTE	CONTENTOR_TOTAL_LITROS
-9.143308809	38.70807879	355	Misericórdia	15805: R do Alecrim (Par (->)(26->30): R Ferragial - R	Lixos	CV0090	90	1	90
-9.143308809	38.70807879	356	Misericórdia	15805: R do Alecrim (Par (->)(26->30): R Ferragial - R	Lixos	CV0240	240	7	1680
-9.143308809	38.70807879	357	Misericórdia	15805: R do Alecrim (Par (->)(26->30): R Ferragial - R	Lixos	CV0090	90	1	90
-9.143308809	38.70807879	358	Misericórdia	15805: R do Alecrim (Par (->)(26->30): R Ferragial - R	Papel e Cartão	CV0240	240	6	1440
-9.143308809	38.70807879	359	Misericórdia	15805: R do Alecrim (Par (->)(26->30): R Ferragial - R	Papel e Cartão	CV0090	90	1	90
-9.143377778	38.70807819	364	Misericórdia	15806: R do Alecrim (Impar (27->53)(->): R Ataíde - R	Lixos	CV0240	240	1	240
-9.143377778	38.70807819	365	Misericórdia	15806: R do Alecrim (Impar (27->53)(->): R Ataíde - R	Lixos	CV0140	140	6	840
-9.143377778	38.70807819	366	Misericórdia	15806: R do Alecrim (Impar (27->53)(->): R Ataíde - R	Lixos	CV0120	120	1	120
-9.143377778	38.70807819	367	Misericórdia	15806: R do Alecrim (Impar (27->53)(->): R Ataíde - R	Lixos	CV0090	90	2	180
-9.143377778	38.70807819	368	Misericórdia	15806: R do Alecrim (Impar (27->53)(->): R Ataíde - R	Lixos	CV0240	240	1	240
-9.143377778	38.70807819	369	Misericórdia	15806: R do Alecrim (Impar (27->53)(->): R Ataíde - R	Lixos	CV0140	140	6	840
-9.143377778	38.70807819	370	Misericórdia	15806: R do Alecrim (Impar (27->53)(->): R Ataíde - R	Lixos	CV0120	120	1	120
-9.143377778	38.70807819	360	Misericórdia	15806: R do Alecrim (Impar (27->53)(->): R Ataíde - R	Lixos	CV0090	90	2	180
-9.143377778	38.70807819	361	Misericórdia	15806: R do Alecrim (Impar (27->53)(->): R Ataíde - R	Papel e Cartão	CV0140	140	2	280
-9.143377778	38.70807819	362	Misericórdia	15806: R do Alecrim (Impar (27->53)(->): R Ataíde - R	Papel e Cartão	CV0090	90	1	90
-9.143377778	38.70807819	363	Misericórdia	15806: R do Alecrim (Impar (27->53)(->): R Ataíde - R	Papel e Cartão	CA0090	90	1	90
-9.143481807	38.70730262	371	Misericórdia	15807: R do Alecrim (Impar (5->25)(->): R Ferragial - I	Lixos	CV0240	240	1	240
-9.143481807	38.70730262	372	Misericórdia	15807: R do Alecrim (Impar (5->25)(->): R Ferragial - I	Lixos	CV0140	140	3	420
-9.143481807	38.70730262	373	Misericórdia	15807: R do Alecrim (Impar (5->25)(->): R Ferragial - I	Lixos	CV0240	240	1	240
-9.143481807	38.70730262	374	Misericórdia	15807: R do Alecrim (Impar (5->25)(->): R Ferragial - I	Lixos	CV0140	140	3	420
-9.143481807	38.70730262	375	Misericórdia	15807: R do Alecrim (Impar (5->25)(->): R Ferragial - I	Papel e Cartão	CV0240	240	1	240
-9.143481807	38.70730262	376	Misericórdia	15807: R do Alecrim (Impar (5->25)(->): R Ferragial - I	Papel e Cartão	CV0140	140	1	140
-9.143292489	38.70773663	951	Misericórdia	21961: R do Alecrim, 24	Embalagens	CV0090	90	1	90

Figura 3: *Dataset* - Rua do Alecrim

Tanto para a criação da garagem (s) como para a lixeira (t) foram escolhidas duas localizações arbitrárias da freguesia da misericórdia, não coincidindo com as localizações dos caixotes fornecidos no *dataset.* A garagem encontra-se na junta de freguesia da Misericórdia enquanto que a lixeira se encontra no Parque Polivalente da Misericórdia.

Para gerar as arestas foram consideradas todas as ruas adjacentes que aparecem no *dataset,* ou seja, todas as ruas que aparecem em frente ao nome de cada rua. Algumas dessas adjacentes não se encontram nas ruas desta freguesia de modo que foram ignoradas.

```
%---------------------- aresta ---------------------
% aresta(id1, id2, distancia)
aresta(352, 662, 73).
aresta(352, 615, 92).
aresta(345, 435, 38).
aresta(352, 485, 318).
aresta(339, 380, 60).
```

Figura 4: Aresta (incompleto)

A heurística estabelecida para este problema foi a distância de cada nodo à lixeira, gerando o seguinte predicado estimativa:

```
%-------------------- estimativa -------------------
% estimativa(id, distancia)
estimativa(352, 773).
estimativa(389, 873).
estimativa(615, 718).
```

Figura 5: Estimativa (incompleto)

Todos os cálculos de distâncias foram realizadas no ficheiro Parser.java através do seguinte método:

```
private static int calculateDistance(double lat1, double lon1, double lat2, double lon2) {

    double latDistance = Math.toRadians(lat1 - lat2);
    double lngDistance = Math.toRadians(lon1 - lon2);

    double a = Math.sin(latDistance / 2) * Math.sin(latDistance / 2)
            + Math.cos(Math.toRadians(lat1)) * Math.cos(Math.toRadians(lat2))
            * Math.sin(lngDistance / 2) * Math.sin(lngDistance / 2);

    double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));

    return (int) Math.round(AVERAGE_RADIUS_OF_EARTH_M * c);
}
```

Figura 6: Cálculo Distância

PESQUISA NÃO IFNROMADA

No que toca à pesquisa não informada foram utilizados três métodos diferentes, um de pesquisa em largura e dois de pesquisa em profundidade sendo uma a normal pesquisa em profundidade e a outra uma pesquisa em profundidade limitada. As soluções têm todas em consideração a quantidade máxima do camião (15000).

Os algoritmos implementados em *prolog* tem sempre em conta o tipo que se pretende procurar visto que os camiões do lixo coletam lixo apenas de um tipo. É de notar que um caminho para recolher papel não têm apenas pontos de lixo deste tipo, visto que é preciso passar por outros pontos sem o mesmo para poder chegar a outros pontos de papel.

PROCURA EM PROFUNDIDADE

A procura em profundidade é um algoritmo que começa num nodo de um grafo e antes de fazer *backtracking* tenta explorar ao máximo a profundidade de cada um dos seus nós adjacentes. De seguida encontra-se uma representação visual da mesma:

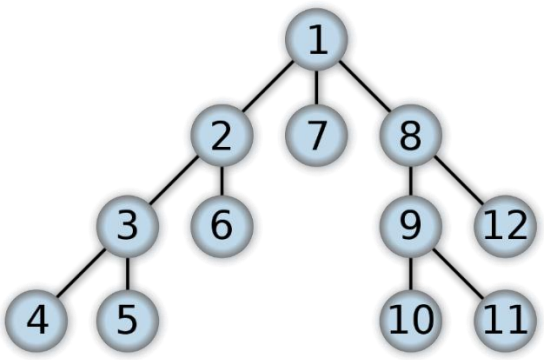

Figura 7: Diagrama - Procura em Profundidade

A implementação começa por encontrar um nodo adjacente, depois é preciso ver se esse nodo já se encontra no histórico de nodos procurados. Caso ainda não esteja é adicionado se a soma da quantidade atual do camião mais a quantidade do tipo em pesquisa desse contentor seja inferior a 15000.

```
dfSolve(Path/Custo/Quantidade, Tipo) :-
    dfCost(s, Tipo, [s], Path, Custo, Quantidade).

dfCost(t, Tipo, Hist, Path, 0, 0):-
    reverse(Hist, Path).

dfCost(Nodo, Tipo, Hist, Path, Custo, Quantidade):-
    adjacenteCusto(Nodo, NodoProx, CustoMovimento),
    getTrashByType(Nodo, Tipo, QuantidadeNodo),
    \+ member(NodoProx, Hist),
    dfCost(NodoProx, Tipo, [NodoProx|Hist], Path, Custo2, Quantidade2),
    Custo is CustoMovimento + Custo2,
    Quantidade is QuantidadeNodo + Quantidade2,
    Quantidade < 15000.
```

Figura 8: Procura em Profundidade

PROCURA EM LARGURA

A pesquisa em largura por sua vez é um algoritmo que ao começar num nodo de um grafo vai percorrendo o mesmo em níveis, ou seja primeiro vê todos os seus adjacentes, de seguida os adjacentes desses e por aí em diante. De seguida apresenta-se uma representação visual do mesmo:

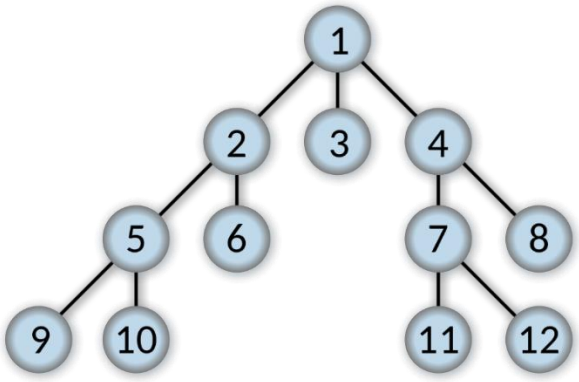

Figura 9: Diagrama - Procura em Largura

De modo a implementar esta pesquisa em *prolog* é necessário ter em consideração que esta funciona como uma *queue* obtendo-se o seguinte resultado:

```
breadth_first(Path/Custo/Quantidade, Tipo):-
    breadth_first_aux([[s]], Path, Tipo, Quantidade),
    calcula_distancia_caminho(Path,Custo).



breadth_first_aux([[t|Visitados]|_], Path, Tipo, Quantidade):-
    Visitados = [Start|_],
    adjacenteCusto(Start,t,_),
    reverse([t|Visitados], Path),
    soma(Path, Tipo, Quantidade),
    Quantidade < 15000.


breadth_first_aux([Visitados|Restantes], Path, Tipo, Quantidade) :-
    Visitados = [Start|_],
    findall( X,
        ( adjacenteCusto(Start,X,_), \+ member(X, Visitados) ),
        [T|Extendido]),
    maplist( adiciona_queue(Visitados), [T|Extendido], VisitadosExtendido),
    append( Restantes, VisitadosExtendido, QueueAtualizada),
    breadth_first_aux( QueueAtualizada, Path, Tipo, Quantidade ).


adiciona_queue( A, B, [B|A]).

calcula_distancia_caminho([Gid1,Gid2|T],Distancia):-
    adjacenteCusto(Gid1,Gid2,DistanciaLigacao),
    calcula_distancia_caminho([Gid2|T],DistanciaAcumulada),
    Distancia is DistanciaLigacao + DistanciaAcumulada.

calcula_distancia_caminho([_],0).
calcula_distancia_caminho([],0).
```

Figura 10: Procura em Largura

PROCURA LIMITADA EM PROFUNDIDADE

Esta procura segue o mesmo princípio da procura em profundidade, no entanto é limitada em termos de profundidade. O algoritmo não limitado, caso esteja a percorrer um grafo de profundidade infinita, vai infinitamente a procurar mais nodos, enquanto que este apenas o faz até ao limite previamente estabelecido.

Para implementar este método em *prolog* basta usar o mesmo método de pesquisa em profundidade e limitá-lo em termos de profundidade. Assim o código em seguida (***dflSolve***) representa o algoritmo realizado para este tipo de procura não informada.

```
dflSolve(Path/Custo/Quantidade, Tipo, Limit) :-
    dflCost(s, Tipo, [s], Path, Custo, Quantidade, Limit).

dflCost(t, Tipo, Hist, Path, 0, 0, Limit):-
    reverse(Hist, Path).

dflCost(Nodo, Tipo, Hist, Path, Custo, Quantidade, Limit):-
    Limit > 0,
    adjacenteCusto(Nodo, NodoProx, CustoMovimento),
    getTrashByType(Nodo, Tipo, QuantidadeNodo),
    \+ member(NodoProx, Hist),
    Limit2 is Limit - 1,
    dflCost(NodoProx, Tipo, [NodoProx|Hist], Path, Custo2, Quantidade2, Limit2),
    Custo is CustoMovimento + Custo2,
    Quantidade is QuantidadeNodo + Quantidade2,
    Quantidade < 15000.
```

Figura 11: Procura em Profundidade Limitada

PESQUISA INFORMADA

Para a realização da pesquisa informada é necessário definir uma heurística. No meu caso escolhi a distância em linha reta entre cada ponto e a lixeira, gerando assim o predicado **estimativa(x,n)**, cujo **x** corresponde ao id do caixote pretendido e **n** corresponde à distância propriamente dita.

Neste trabalho foram utilizados o algoritmo de pesquisa gulosa e também o algoritmo denominado de A Estrela como se indicam de seguida. As soluções têm todas em consideração a quantidade máxima do camião (15000) e também o tipo de lixo.

GULOSA

O algoritmo da pesquisa gulosa segue a ideia de a cada iteração fazer a melhor escolha no momento tendo em consideração a heurística dos nodos adjacentes. Embora mais eficaz em termos de encontrar a solução ótima do que os métodos de pesquisa não informada ainda peca um pouco visto que não tem em consideração o custo dos nodos adjacentes. A implementação em *prolog* encontra-se em seguida:

```
adjacenteGulosa([Nodo|Caminho]/Custo/_, [ProxNodo,Nodo|Caminho]/NovoCusto/Est) :-
    adjacenteCusto(Nodo, ProxNodo, PassoCusto),
    \+ member2(ProxNodo, Caminho),
    NovoCusto is Custo + PassoCusto,
    estimativa(ProxNodo, Est).

resolveGulosa(Nodo, Caminho/Custo/Quantidade, Tipo) :-
    estimativa(Nodo, Estimativa),
    agulosa([[Nodo]/0/Estimativa], CaminhoInverso/Custo/_, Tipo, Quantidade),
    reverse(CaminhoInverso, Caminho).

agulosa(Caminhos, Caminho, Tipo, Quantidade) :-
    melhorGulosa(Caminhos, Caminho),
    Caminho = [Nodo|Xs]/_/_,
    soma([Nodo|Xs], Tipo, Quantidade),
    Quantidade < 15000,
    fim(Nodo).

agulosa(Caminhos, SolucaoCaminho, Tipo, Quantidade) :-
    melhorGulosa(Caminhos, MelhorCaminho),
    % writeln(MelhorCaminho),
    seleciona(MelhorCaminho, Caminhos, OutrosCaminhos),
    expandeGulosa(MelhorCaminho, ExpCaminhos),
    append(OutrosCaminhos, ExpCaminhos, NovoCaminhos),
    agulosa(NovoCaminhos, SolucaoCaminho, Tipo, Quantidade).

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

melhorGulosa([Caminho1/Custo1/Est1,_/Custo3/Est3|Caminhos], MelhorCaminho) :-
    Est1 =< Est3, !,
    melhorGulosa([Caminho1/Custo1/Est1|Caminhos], MelhorCaminho).

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

expandeGulosa(Caminho, ExpCaminhos) :-
    findall(NovoCaminho, adjacenteGulosa(Caminho,NovoCaminho), ExpCaminhos).
```

Figura 12: Procura Gulosa

A ESTRELA

Este é o melhor algoritmo para encontrar logo à primeira a solução ótima do problema. Tal como a pesquisa gulosa a cada interação faz a melhor escolha que pode efetuar, só que ao contrário dessa, não tem apenas em consideração a heurística, este algoritmo soma o custo da heurística ao custo do nodo para obter realmente o melhor próximo nodo. Em *prolog* o algoritmo materializa-se da seguinte forma:

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

resolveAEstrela(Nodo, Caminho/Custo/Quantidade, Tipo) :-
    estimativa(Nodo, Estimativa),
    aestrela([[Nodo]/0/Estima], CaminhoInverso/Custo/_, Tipo, Quantidade),
    reverse(CaminhoInverso, Caminho).

aestrela(Caminhos, Caminho, Tipo, Quantidade) :-
    melhorAS(Caminhos, Caminho),
    Caminho = [Nodo|Xs]/_/_,
    soma([Nodo|Xs], Tipo, Quantidade),
    Quantidade < 15000,
    fim(Nodo).

aestrela(Caminhos, SolucaoCaminho, Tipo, Quantidade) :-
    melhorAS(Caminhos, MelhorCaminho),
    seleciona(MelhorCaminho, Caminhos, OutrosCaminhos),
    expandeaestrela(MelhorCaminho, ExpCaminhos),
    append(OutrosCaminhos, ExpCaminhos, NovoCaminhos),
    aestrela(NovoCaminhos, SolucaoCaminho, Tipo, Quantidade).

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

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

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

expandeaestrela(Caminho, ExpCaminhos) :-
    findall(NovoCaminho, adjacenteAS(Caminho,NovoCaminho), ExpCaminhos).
```

Figura 13: Procura A Estrela

FUNCIONALIDADES

Para a realização das funcionalidades propostas foi usada o algoritmo de pesquisa em profundidade indicado em cima, é de notar que todas recebem o tipo de lixo para o qual se pretende realizar a mesma.

As funcionalidades têm todas a mesma base, ou seja, encontrar todos os caminhos usando o predicado ***dfSolve,*** ordenar o mesmo através de um *merge sort* e depois retirar a cabeça da lista resultante. Note que esta implementação facilmente daria por exemplo o top 5 das funcionalidades em vez de apenas a "melhor", trocando apenas o predicado ***head*** por um ***take N***.

```
%--------------------------------- - - - - - - - - - - - - - - - - -
% Ordenação de caminhos através de mergesort
% Número : serve para saber qual é o modo de pesquisa que se quer
% T : Tipo de lixo, usado nas funcionalidades uqe precisam do mesmo
mergesort(1,T,[],[]).
mergesort(1,T,[X],[X]).
mergesort(1,T,List,Sorted):-
    divide(List,L1,L2),
    mergesort(1,T,L1,Sorted1),
    mergesort(1,T,L2,Sorted2),
    mergeTrashQuantity(T,Sorted1,Sorted2,Sorted).

mergesort(2,T,[],[]).
mergesort(2,T,[X],[X]).
mergesort(2,T,List,Sorted):-
    divide(List,L1,L2),
    mergesort(2,T,L1,Sorted1),
    mergesort(2,T,L2,Sorted2),
    mergeAvg(T,Sorted1,Sorted2,Sorted).

mergesort(3,T,[],[]).
mergesort(3,T,[X],[X]).
mergesort(3,T,List,Sorted):-
    divide(List,L1,L2),
    mergesort(3,T,L1,Sorted1),
    mergesort(3,T,L2,Sorted2),
    mergeTrashPoints(T,Sorted1,Sorted2,Sorted).

mergesort(4,T,[],[]).
mergesort(4,T,[X],[X]).
mergesort(4,T,List,Sorted):-
    divide(List,L1,L2),
    mergesort(4,T,L1,Sorted1),
    mergesort(4,T,L2,Sorted2),
    mergeFastest(T,Sorted1,Sorted2,Sorted).

mergesort(5,T,[],[]).
mergesort(5,T,[X],[X]).
mergesort(5,T,List,Sorted):-
    divide(List,L1,L2),
    mergesort(5,T,L1,Sorted1),
    mergesort(5,T,L2,Sorted2),
    mergeEficient(T,Sorted1,Sorted2,Sorted).
```

Figura 14: Merge sort

O ***mergesort*** recebe um número em primeiro lugar simplesmente para escolher o método de ordenação dependendo da funcionalidade pretendida. O predicado é análogo a todos os casos tirando a função ***merge***. Para cada funcionalidade vai ser explicada cada uma destas.

INDICADORES DE PRODUTIVIDADE

Existem dois indicadores de produtividade indicados no enunciado, sendo o primeiro a quantidade recolhida e o segundo a distância média entre os pontos percorridos durante um percurso.

QUANTIDADE RECOLHIDA

Como o nome indica, esta funcionalidade vai devolver o caminho cuja quantidade recolhida é maior.

$$resultado = Max\ (Quantidade)$$

O predicado ***maxTrashQuantity*** procura todas as soluções do problema utilizando ***dfSolve*** cujo resultado para cada caminho é Caminho/Distância/Lixo. Em primeiro lugar, nesta funcionalidade como o que nos interessa é apenas o lixo e o caminho, apenas esses campos são capturados para colocar em L. De seguida L é ordenado e retirada a sua cabeça cujo resultado contém caminho/lixo.

```
% Type - Tipo de lixo pretendido
% H - Caminho escolhido / Lixo máximo
maxTrashQuantity(Type, H) :-
    findall((P,Q), dfSolve(P/C/Q, Type), L),
    mergesort(1,Type,L,R),
    head(R,H).
```

Figura 15: *maxTrashQuantity*

O ***mergesort*** com 1 corresponde a utilizar o predicado ***mergeTrashQuantity*** para ordenar a lista. Este o que faz é organizar uma lista por ordem decrescente de custo. Para isso a cada dois elementos H1 e H2 retira o seu custo usando o predicado ***getSecondTuple*** que pega no segundo elemento de um par.

```
%--------------------------------- - - - - - - - - - - - - -
% Ordena os caminhos por mair quantidade de lixo de um tipo
mergeTrashQuantity(Type,X,[],X):- !.
mergeTrashQuantity(Type,[],X,X):- !.
mergeTrashQuantity(Type,[H1|T1],[H2|T2],[H1|T3]):-
    getSecondTuple(H1,C1),
    getSecondTuple(H2,C2),
    C1 >= C2,
    mergeTrashQuantity(Type,T1,[H2|T2],T3).
mergeTrashQuantity(Type,[H1|T1],[H2|T2],[H2|T3]):-
    getSecondTuple(H1,C1),
    getSecondTuple(H2,C2),
    C1 < C2,
    mergeTrashQuantity(Type,[H1|T1],T2,T3).
mergeTrashQuantity(Type,[H1|T1],[H2|T2],[H1,H2|T3]):-
    mergeTrashQuantity(Type,T1,T2,T3).
```

Figura 16: *mergeTrashQuantity*

DISTÂNCIA MÉDIA ENTRE PONTOS

Neste indicador de produtividade foi escolhido o caminho cuja distância média entre os pontos é mínima, para tal foi considerada a seguinte formula:

$$resultado = Min\left(\frac{Custo\ Caminho}{\#\ Nodos - 1}\right)$$

O predicado ***distanceBetweenAVG*** procura todas as soluções do problema utilizando ***dfSolve*** cujo resultado para cada caminho é Caminho/Distância/Lixo. Primeiramente, o que nos interessa nesta é a distância e o caminho, portanto são esses os campos capturados para colocar em L. De seguida L é ordenado e retirada a sua cabeça cujo resultado contém caminho/lixo. Posteriormente é calculado o tamanho da lista presente no par utilizando o ***lengthTuple*** e retirada a distância da solução através do ***getSecondTuple***. Por fim é calculada e armazenado em A, a distância média entre dois pontos de acordo com a fórmula de cima.

```
% T - Tipo de lixo
% H - Caminho escolhido
% A - Distância média entre dois pontos
distanceBetweenAVG(T,H,A) :-
    findall((Path,Cost), dfSolve(Path/Cost/Quant, T), L),
    mergesort(2,T,L,R),
    head(R,H),
    lengthTuple(H, Len),
    getSecondTuple(H,Cost),
    A is Cost / (Len - 1).
```

Figura 17: *distanceBetweenAVG*

O ***mergesort*** com 2 corresponde a utilizar o predicado ***mergeTrashQuantity*** para ordenar a lista. Este vai organizar a lista fornecida por ordem crescente de distância média entre pontos. Assim sendo, para cada dois pontos H1 e H2 obtém o seu número de pontos a partir de ***lengthTuple*** e o seu custo a partir de ***getSecondTuple***. Para realizar a comparação aplica a fórmula descrita em cima.

```
%--------------------------------- - - - - - - - - - - - - - - - - -
% Organiza os caminhos de acordo com menor distância media entre pontos
mergeAvg(Type,X,[],X):- !.
mergeAvg(Type,[],X,X):- !.
mergeAvg(Type,[H1|T1],[H2|T2],[H1|T3]):-
    getSecondTuple(H1,C1),
    getSecondTuple(H2,C2),
    lengthTuple(H1,L1),
    lengthTuple(H2,L2),
    C1 / (L1 - 1) =< C2 / (L2 - 1),
    mergeAvg(Type,T1,[H2|T2],T3).
mergeAvg(Type,[H1|T1],[H2|T2],[H2|T3]):-
    getSecondTuple(H1,C1),
    getSecondTuple(H2,C2),
    lengthTuple(H1,L1),
    lengthTuple(H2,L2),
    C1 / (L1 - 1) > C2 / (L2 - 1),
    mergeAvg(Type,[H1|T1],T2,T3).
mergeAvg(Type,[H1|T1],[H2|T2],[H1,H2|T3]):-
    mergeAvg(Type,T1,T2,T3).
```

Figura 18: *mergeAVG*

GERAR PERCURSOS INDIFERENCIADOS OU SELETIVOS

Nesta funcionalidade os próprios algoritmos de pesquisa já a implementam através do uso do tipo de lixo nos seus argumentos.

CIRCUITO COM MAIS PONTOS DE RECOLHA

Esta funcionalidade vai retornar o percurso cujo número de pontos de recolha seja o maior a partir e todos os caminhos calculados.

$$resultado = Max\ (\#Pontos\ de\ Recolha)$$

O predicado ***maxTrashPointsType*** procura todas as soluções do problema utilizando ***dfSolve*** cujo resultado para cada caminho é Caminho/Distância/Lixo. Em L apenas é necessário colocar o caminho, visto este ser o único elemento necessário para realizar os cálculos pretendidos. Em seguida é realizada a sua ordenação e retirada a sua cabeça H. Por fim é contado o número de pontos de lixo em H (através de ***contTrashPoints***) e colocado em Q para apresentar o resultado.

```
% Type - Tipo de lixo pretendido
% H - Caminho escolhido
% Q - Pontos do tipo Type
maxTrashPointsType(Type, H, Q) :-
    findall(Path, dfSolve(Path/Cost/Quant, Type), L),
    mergesort(3,Type,L,R),
    head(R,H),
    countTrashPoints(Type,H,Q).
```

Figura 19: *maxTrashPointsType*

O ***mergesort*** com 3 corresponde a utilizar o predicado ***mergeTrashPoints*** para ordenar a lista. Este vai organizar a lista por ordem decrescente de quantidade de pontos de lixo. Assim para cada dois caminhos H1 e H2, calcula o número de pontos de um determinado tipo (***countTrashPoints***) e compara os mesmos.

```
%------------------------------ - - - - - - - - -
% Ordena por mais pontos de um determinado tipo
mergeTrashPoints(Type,X,[],X):- !.
mergeTrashPoints(Type,[],X,X):- !.
mergeTrashPoints(Type,[H1|T1],[H2|T2],[H1|T3]):-
    countTrashPoints(Type,H1,TP1),
    countTrashPoints(Type,H2,TP2),
    TP1 >= TP2,
    mergeTrashPoints(Type,T1,[H2|T2],T3).
mergeTrashPoints(Type,[H1|T1],[H2|T2],[H2|T3]):-
    countTrashPoints(Type,H1,TP1),
    countTrashPoints(Type,H2,TP2),
    TP1 < TP2,
    mergeTrashPoints(Type,[H1|T1],T2,T3).
mergeTrashPoints(Type,[H1|T1],[H2|T2],[H1,H2|T3]):-
    mergeTrashPoints(Type,T1,T2,T3).
```

Figura 20: *mergeTrashPoints*

CIRCUITO MAIS RÁPIDO

Para calcular o circuito mais rápido, assumindo que a velocidade média do camião do lixo é aproximadamente a mesma em todo o percurso, através da fórmula física seguinte pode-se dizer que o caminho mais rápido será aquele cuja distância percorrida seja a menor.

$$v = \frac{\Delta x}{\Delta t}$$

Desta forma sabendo que o custo no resultado da função de procura representa a distância percorrida para realizar o percurso, basta saber qual é o percurso cujo custo é o menor.

O predicado ***fasttestPath*** procura todas as soluções do problema utilizando ***dfSolve*** cujo resultado para cada caminho é Caminho/Distância/Lixo. Em L vai colocar todas as componentes relevantes, neste caso o caminho e distancia. Por fim é realizada a sua ordenação e retirada a sua cabeça que já vai conter o resultado pretendido, Caminho/Distância.

```
% H - Caminho escolhido com respetiva distância
fasttestPath(Type,H) :-
    findall((Path,Cost), dfSolve(Path/Cost/Quant, Type), L),
    mergesort(4,Type,L,R),
    head(R,H).
```

Figura 21: *fasttestPath*

O ***mergesort*** com 4 corresponde a utilizar o predicado ***mergeFastest*** para ordenar a lista. Este vai organizar a lista por ordem crescente de distância de percurso. Para cada dois caminhos H1 e H2 é obtido a sua distância recorrendo a ***getSecondTuple*** e posteriormente comparadas.

```
%-------------------------------- - - - - - - -
% Ordena caminhos por menor distancia
mergeFastest(Type,X,[],X):- !.
mergeFastest(Type,[],X,X):- !.
mergeFastest(Type,[H1|T1],[H2|T2],[H1|T3]):-
    getSecondTuple(H1,C1),
    getSecondTuple(H2,C2),
    C1 =< C2,
    mergeFastest(Type,T1,[H2|T2],T3).
mergeFastest(Type,[H1|T1],[H2|T2],[H2|T3]):-
    getSecondTuple(H1,C1),
    getSecondTuple(H2,C2),
    C1 > C2,
    mergeFastest(Type,[H1|T1],T2,T3).
mergeFastest(Type,[H1|T1],[H2|T2],[H1,H2|T3]):-
    mergeFastest(Type,T1,T2,T3).
```

Figura 22: *mergeFasttest*

CIRCUITO MAIS EFICIENTE

Para circuito com maior eficiência achei por bem considerar que o circuito mais eficiente é aquele que em menos distância percorrida consegue recolher uma maior quantidade de lixo. Para isso considerei a fórmula seguinte:

$$resultado = Max\left(\frac{quantidade}{distância}\right)$$

O predicado ***mostEfficient*** procura todas as soluções do problema utilizando ***dfSolve*** cujo resultado para cada caminho é Caminho/Distância/Lixo. Nesta funcionalidade todos os campos do algoritmo de pesquisa vão ser necessários, de modo que são todos colocados em L. De seguida é ordenada a lista por ordem crescente de eficiência e retirado o primeiro elemento da mesma, H. Desse triplo vai ser retirada a distância através do ***getSecondTriplet*** e a quantidade de lixo através do ***getThirdTriplet***. Por fim é realizada a divisão da quantidade pela distância para devolver a fração do mesmo.

```
% Type - Tipo de lixo pretendido
% H - Caminho/Custo/Quantidade lixo
% Res - Resultado da divisão Q/C
mostEfficient(Type, H, Res) :-
    findall((P,C,Q), dfSolve(P/C/Q, Type), L),
    mergesort(5,Type,L,R),
    head(R,H),
    getSecondTriplet(H,Cost),
    getThirdTriplet(H,Quant),
    Res is Quant/Cost.
```

Figura 23: *mostEfficient*

O ***mergesort*** com 5 corresponde a utilizar o predicado ***mergeEficient*** para ordenar a lista. Este vai organizar a lista por ordem decrescente de eficiência. Para cada dois caminhos H1 e H2 é obtido a sua distância recorrendo a ***getSecondTriplet*** e a sua quantidade de lixo com ***getThirdTriplet.*** Por fim é realizada a comparação com base na fórmula apresentada em cima.

```
%-------------------------------- - - - - - - -
% Ordena caminhos por maior eficiencia
mergeEficient(Type,X,[],X):- !.
mergeEficient(Type,[],X,X):- !.
mergeEficient(Type,[H1|T1],[H2|T2],[H1|T3]):-
    getSecondTriplet(H1,C1),
    getSecondTriplet(H2,C2),
    getThirdTriplet(H1,TQ1),
    getThirdTriplet(H2,TQ2),
    TQ1 / C1 >= TQ2 / C2,
    mergeEficient(Type,T1,[H2|T2],T3).
mergeEficient(Type,[H1|T1],[H2|T2],[H2|T3]):-
    getSecondTriplet(H1,C1),
    getSecondTriplet(H2,C2),
    getThirdTriplet(H1,TQ1),
    getThirdTriplet(H2,TQ2),
    TQ1 / C1 < TQ2 / C2,
    mergeEficient(Type,[H1|T1],T2,T3).
mergeEficient(Type,[H1|T1],[H2|T2],[H1,H2|T3]):-
    mergeEficient(Type,T1,T2,T3).
```

Figura 24: *mergeEfficient*

EXTRA – ALGORITMO PERCURSO QUE PASSA EM TODOS OS NÓS

Para esta parte foram considerados percursos completos, ou seja, um percurso é o conjunto de todos os caminhos que começam numa garagem, recolhem lixo, vão para a lixeira e se ainda existirem caixotes por descarregar volta a garagem e faz o mesmo processo. Toda esta parte encontra-se à parte no ficheiro *extra.py*.

Para este método foram realizados os métodos de pesquisa em profundidade, gulosa e A estrela. Para além disso foram realizadas todas as funcionalidades como foram descritas anteriormente, no entanto em algumas como por exemplo a que devolve o caminho com mais pontos de recolhas, foi dividido o resultado pelo número de nós em que passa o camião, visto que com este método o camião passa sempre em todos os pontos.

No entanto este método tem uma desvantagem enorme, não permite realizar estes algoritmos para grafos muito grandes, daí no ficheiro extra.pl conter no seu interior um grafo pequeno onde se consegue realizar tudo que foi implementado. Esse grafo é o seguinte:

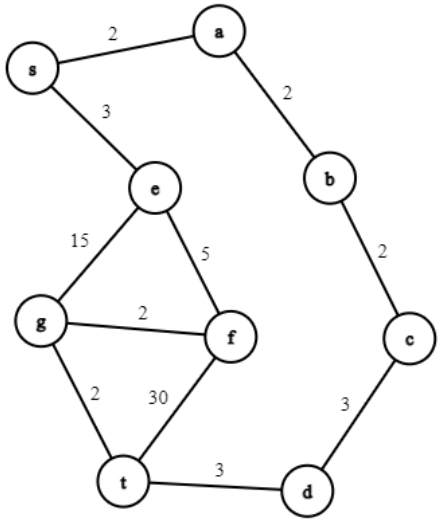

Figura 25: Grafo utilizado

Foi ainda adicionada uma funcionalidade extra para este método cuja função é calcular o circuito que passa menos vezes em pontos em que não existe lixo, ou seja, na garagem e lixeira.

Primeiramente foi necessário realizar um predicado que conte o número de pontos que não são pontos de lixo (***countNotTrashPoints***), ou seja que são a garagem ou a lixeira.

```
%------------------------------ - - -
% Conta pontos que não são de recolha
countNotTrashPoints([],0).
countNotTrashPoints([T|List], N) :-
    isNotTrash(T),
    countNotTrashPoints(List, N1),
    N is N1 + 1.
countNotTrashPoints([T|List], N) :-
    \+ isNotTrash(T),
    countNotTrashPoints(List, N).
```

Figura 26: *countTrashPoints*

Em seguida é, pois, necessário ver numa lista de percursos qual é o que tem uma menor quantidade deste tipo de pontos, aí entra o ***minNotTRASH:***

```
%------------------------------ - - - - - - - - - - - - - - -
% Encontra o caminho que passa menos vezes na garagem e lixeira
minNotTRASH([Path], Path) :- !, true.
minNotTRASH([Path | List], Path) :-
    length(Path,L1),
    countNotTrashPoints(Path, T1),
    minNotTRASH(List, NewPath),
    countNotTrashPoints(NewPath, T2),
    length(NewPath,L2),
    T1 / L1 =< T2 / L2, !.
minNotTRASH([_ | List], New) :-
    minNotTRASH(List, New).
```

Figura 27: *minNotTRASH*

Para finalizar o predicado ***minNotTrashPoints*** vai calcular os percursos presentes no sistema e aplicar o predicado anterior para obter a resposta pretendida.

```
% R - Caminho escolhido
% Q/Len - Pontos do tipo Type / Pontos totais de R
minNotTrashPoints(R, Q/Len) :-
    findall(Path, dfSolve5(Path/Cost, []), L),
    minNotTRASH(L, R),
    countNotTrashPoints(R,Q),
    length(R,Len).
```

Figura 28: *minNotTrashPoints*

RESULTADOS

Em seguida apresentam-se os resultados e comparações entre os diversos tipos de pesquisa utilizados no trabalho. É de notar que para obter o tempo e a memória de cada um dos métodos de pesquisa procedeu-se ao use de **time(predicado)** e de **statistics(global_stack, [G1,L1])**, conforme é apresentado em seguida:

```
%-------------------------------- - - -
% Estatísticas Procura em Profundidade
statisticsDF(R,T) :-
    statistics(global_stack, [G1,L1]),
    time(dfSolve(R,T)),
    statistics(global_stack, [G2,L2]),
    Res is G2 - G1,
    write("Memory: "),
    write(Res).
```

Figura 29: *statisticsDF* (funções análogas para as outras pesquisas)

É de notar que foi corrido 3 vezes o predicado de estatística de cada uma das procuras e feita a média entre os resultados obtidos.

Estratégia	Tempo (segundos)	Espaço (bytes)	Custo Solução(tipo lixo)	Garante Ótima (à primeira)
BF	0.000	78272	1295	Não
DF	0.000	3256	1628	Não
DF Limitada	0.000	2928	1381 (limite = 3)	Não
Gulosa	0.000	12320	588	Não
A*	0.000	14776	588	Sim

Através de uma comparação mais atenta conseguimos observar que os métodos de pesquisa informada acabam por ser melhores a encontrar a solução ótima, uma vez que estes têm algo que os tenta direcionar para o caminho certo (heurística) enquanto que os outros agem completamente às cegas. Além disso é possível observar que o melhor para encontrar uma solução ótima é o A* o que faz bastante sentido visto que é aquele que para tomar cada passo tem em consideração mais parâmetros (tanto o custo as arestas como a heurística de cada ponto).

É possível observar que os métodos de pesquisa em profundidade gastam muito menos memória que todos os outros. Para além disso a procura em largura revelou-se bastante *memory hungry* devido ao seu método de pesquisa.

A pesquisa em profundidade revelou-se a mais rápida a procurar todas as suas soluções, daí que as funcionalidades tenham sido implementadas na sua maioria com este método de pesquisa.

CONCLUSÃO

Em conclusão todas as funcionalidades foram implementadas com sucesso, sendo que para a implementação de cada uma delas foi tomada bastante atenção e cuidado a cada detalhe.

Durante a realização do trabalho a maior dificuldade foi claramente gerar um *dataset* que não fosse demasiado extenso, visto que dependendo deste, o *prolog* poderia não conseguir correr os algoritmos realizados. No entanto essas dificuldades acabaram por ser superadas com sucesso.

Penso que o extra incluído é muito interessante, visto que permitia gerar uma rota completa para que o camião possa coletar o lixo todo de um grafo, no entanto é um algoritmo bastante mais complexo e não corre em grafos de maiores dimensões.

Em suma considero que foi um trabalho bem-sucedido, sendo que como trabalho futuro ficaria acrescentar mais algumas funcionalidades e maximizar a eficiência de todos os algoritmos.

REFERÊNCIAS

[Novais, 2020] Novais, Paulo,
"Métodos de resolução de problemas e de procura",
Universidade do Minho, Portugal, 2020.

[RUSSEL, 2020] RUSSEL, S. J., Norving, P., & Davis, E.
"Artificial intelligence: a modern approach", 4th ed.
Upper Saddle River, 2020.

[COSTA, 2008] COSTA, E., Simões, A.,
"Inteligência Artificial-Fundamentos e Aplicações".
2008.