Universidade do Minho
Escola de Engenharia

UNIVERSIDADE DO MINHO

TRABALHO PRÁTICO INDIVIDUAL

SISTEMAS DE REPRESENTAÇÃO DE CONHECIMENTO E RACIOCÍNIO

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA

(2º SEMESTRE 20/21)

A89612 - Ana Rita Abreu Peixoto

Braga

Junho - 2021

RESUMO

O presente trabalho tem como objetivo a implementação de diferentes estratégias de procura pertencentes às categorias de procura **informada** e **não informada**, num contexto aproximado do real considerando circuitos de recolha de resíduos urbanos no concelho de Lisboa.

Entre os diversos tipos de procura, os abordados serão a procura em profundidade (com e sem limitação) e em largura, no campo da pesquisa informada. No âmbito da pesquisa não informada, serão alvo de análise as procuras gulosa e A estrela.

O ponto de partida para a realização deste projeto foi um *dataset* que possui informações relativas ao tópico e circuito em questão e, a partir do mesmo, foi possível transforma-lo numa representação em grafo e prosseguir para a implementação dos diferentes algoritmos de procura.

Por último, foi efetuada uma análise de resultados, tendo em conta a complexidade temporal e espacial, de modo a possibilitar a comparação entre os diferentes algoritmos.

ÍNDICE

ÍNDICE FIGURAS

ÍNDICE TABELAS

INTRODUÇÃO

Neste trabalho individual da UC SRCR serão abordadas diferentes estratégias de procura relativas aos circuitos de recolha de resíduos urbanos da câmara municipal de Lisboa.

Para isso, começou-se pela interpretação e *parse* de um *dataset* dos pontos de recolha de uma freguesia da cidade de Lisboa para permitir a sua representação em termos de grafo. Após esta transformação e povoamento da base de conhecimento é possível aplicar os diferentes algoritmos e deste modo responder a algumas questões pertinentes, tais como: circuitos de recolha indiferenciada/seletiva, quais os circuitos com menos pontos de recolha, qual o circuito mais rápido e até mesmo o circuito mais eficiente.

Deste modo, o presente relatório possui uma descrição detalhada do trabalho realizado, acompanhado de imagens representativas do código implementado e das considerações que sustentaram o desenvolvimento do projeto.

PRELIMINARES

Antes de prosseguir para a esposição deste trabalho, considero conveniente apresentar alguns conceitos que careceram de estudo e análise anteriormente à realização deste projeto.

O âmbito deste trabalho individual são os métodos de resolução de problemas e de procura. Nesta área de estudo, as estratégias de procura são avaliadas de acordo com 4 critérios:

- **Completude**: trata da verificar se a procura encontra a solução, caso exista;
- **Complexidade no tempo**: averigua o tempo decorrido para encontrar uma solução;
- **Complexidade no espaço**: verifica a memória necessária para a procura;
- **Otimização**: a solução ótima é encontrada, entre todas as outras?

De modo a responder a estas questões/tópicos, serão consideradas dois tipos de estratégia: procura **não informada e informada**.

No primeiro caso, trata-se de uma procura que não possui informação acerca do número de passos ou custo desde o estado inicial até ao objetivo. Apenas é possível distinguir entre o estado objetivo e o estado não objetivo. Por outras palavras, este tipo de pesquisa encontra soluções através da comparação do estado atual com o estado final, sem ter acesso a outros detalhes do problema. Por esta razão, a **procura não informada** também é comummente denominada por **procura cega.** Exemplos de algoritmos deste tipo de procura que serão abordados neste trabalho são a procura em **profundidade** e em **largura**.

Na **procura informada**, o modo de atuação sobre o grafo é diferente. Os algoritmos **informados** utilizam conhecimento no processo de procura, sendo que não tratam apenas de comparar o estado atual com o final. Contrariamente a este método, a procura informada é capaz de tomar uma decisão entre 2 nodos considerando a heurística implementada. Por exemplo, num problema de caminho mais curto, um algoritmo informado deverá optar por seguir para o nodo que possui menos custo. Esta pesquisa é também designada por **procura heurística**. As estratégias que serão implementadas relativas a esta categoria serão a pesquisa **gulosa** e **A estrela**.

DESCRIÇÃO DO TRABALHO

O presente trabalho tem como objetivo efetuar a recomendação de circuitos de recolha de resíduos urbanos no concelho de Lisboa. Deste modo, a partir de um *dataset* com informações relativas a este tópico, é possível efetuar algoritmos de pesquisa e obter percursos ótimos, consoante um critério desejado.

Deste modo, é de notar que na generalidade, o *dataset* referido possui informações relativas ao nome da rua em questão, coordenadas geográficas (latitude e longitude), contentores, freguesia, tipo de resíduo e capacidade do contentor. A partir destes dados, é possível determinar adjacências entre pontos e transformar o mapa numa representação em grafo.

Além disso, é necessário ter em consideração alguns aspetos relativos aos circuitos entre dois pontos do grafo:

- ⇨ Todos os caminhos começam no mesmo **ponto inicial** (201) que representa a garagem.
- ⇨ Um circuito é constituído por diferentes **pontos de recolha**.
- ⇨ Os últimos pontos de cada percurso deverão ser o **depósito** (200) e a **garagem** (201).
- ⇨ A **garagem** e o **depósito** são adjacentes a todos os outros pontos do grafo e, por isso, é possível ir diretamente para estes pontos a partir de qualquer outro, e vice-versa.

Deste modo, no programa desenvolvido é possível calcular percursos entre 2 quaisquer pontos arbitrários, sendo que todos os percursos começam e acabam nos mesmos pontos, tal como supramencionado. Estes pontos comuns não representam pontos de recolha, e como tal não é possível recolher resíduos nestas localizações.

É de notar que, para a realização deste trabalho, foi considerada a **versão completa**, ou seja, o camião possui capacidade limitada a 15000 litros de lixo. Quando atinge esta quantidade deve deslocar-se ao depósito para deixar os resíduos. De seguida, prossegue com a recolha nos restantes pontos do percurso. No final de percorrer todo o caminho, regressa ao depósito e à garagem.

Por último, é importante referir também que, para cada algoritmo, será abordada e analisada a sua **complexidade** em termos **temporal** e **espacial**, de modo a dar resposta às questões “quanto tempo demora a encontrar a solução?” e “quanta memória é necessária para efetuar a procura?”, respetivamente.

FORMULAÇÃO DO PROBLEMA

Para a concretização concisa deste trabalho foi necessário efetuar a formulação do mesmo indicando e expondo diversos tópicos que considero relevantes, ou seja, decidir quais ações e estados considerar no contexto do problema apresentado.

Primeiramente, é de notar que um problema pode ser definido formalmente em cinco componentes:

- Representação do estado;
- Estado inicial (ponto partida);
- Estado objetivo (define os estados desejados);
- Operadores;
- Custo da solução.

Deste modo, o problema em mãos pode ser formulado como um **problema de procura em grafos**, transformando a informação presente no *dataset* num grafo. Deste modo, cada nodo constituinte do grafo é um registo de ponto de recolha associado a um determinado tipo de resíduo. Este nodo do grafo está presente na base de conhecimento materializando-se no predicado "registo".

A partir deste grafo, é possível obter diferentes percursos entre um ponto inicial e final, dependendo do critério de escolha. Além disso, podemos considerar que o **estado inicial** do problema corresponde ao **ponto inicial** fornecido, que representa o primeiro ponto a ser visitado pelo camião e que o objetivo é chegar ao **ponto final** do percurso. É importante ter em consideração que o ponto inicial e final de cada percurso é a garagem (registo 201), sendo que deverá passar pelo depósito (registo 200) antes de regressar à garagem. No entanto, nos algoritmos de procura estes pontos não são considerados dado que não se tratam de pontos de recolha de resíduos. Além disso, como estes pontos possuem adjacência a todos os outros, existe um caminho que os conecta diretamente.

As soluções obtidas serão avaliadas em termos de custo, que poderá corresponder à distância percorrida, quantidade de lixo recolhida ou até mesmo ao número de idas ao depósito.

LEITURA DO DATASET

O primeiro passo efetuado para a realização deste projeto foi a leitura e *parse* do *dataset* fornecido. Para concretizar tal feito, recorreu-se à linguagem Java que lê o ficheiro em formato CSV e, posteriormente, gera um ficheiro *prolog* com os registos da base de conhecimento, populada com predicados "registo".

Primeiramente, após a leitura do ficheiro CSV é efetuada a separação de cada linha com recurso à expressão regular ";". Após efetuar este passo, são recolhidos os dados relevantes para a formulação do problema. Os dados considerados pertinentes para a concretização dos objetivos propostos foram: ID Ponto de Recolha, tipo de lixo, latitude, longitude, lista de contentores, lista de adjacentes a esse ponto, quantidade (em litros) presentes nesse ponto e, por último, um identificador desse registo. Esse identificador é um valor inteiro externo ao *dataset* e gerado aquando do *parse,* com o propósito de associar um número único a cada registo.

Nesta fase do projeto, houve algumas considerações importantes que considero relevante destacar. A primeira a referir é o procedimento para o cálculo dos pontos adjacentes, que passou por considerar a primeira coluna de ruas adjacentes do *dataset* e, além disso, considerar também os registos abaixo cujo ID de ponto de recolha ou tipo de lixo fossem distintos do registo atual. Em adição, é de forte utilidade referir também que os dados do *dataset* foram agrupados por ID de Ponto de Recolha e tipo de lixo. A título de exemplo, na seguinte figura está presente uma porção do *dataset* gerado:

```
%--------------------------------- - - - - - - - - - -  -  -  -  -   -
%--------------------------------- - - - - - - - - - -  -  -  -  -   -
% BASE DE CONHECIMENTO
%--------------------------------- - - - - - - - - - -  -  -  -  -   -
% registo: Ponto Recolha, Tipo Lixo, Identificador, Latitude, Longitude, Contentores, IDs adjacentes, Total Litros -> {V,F}

registo(15871,"lixos",74,-9.144417,38.706738,[602, 603],[75, 106, 107],480).
registo(19236,"organicos",132,-9.153608,38.714466,[164],[133],140).
registo(15869,"lixos",73,-9.144479,38.70709,[591, 592, 593, 594],[74, 75, 76],1600).
registo(15868,"lixos",71,-9.144028,38.706894,[584, 585, 586, 587, 588, 589],[72, 92, 93, 94, 95, 96, 97, 98],3980).
registo(15850,"embalagens",46,-9.148354,38.710396,[488],[47],240).
registo(15867,"lixos",70,-9.143192,38.70718,[578, 579, 580, 581, 582, 583],[1, 2, 3, 4, 5, 6, 71, 169],2640).
registo(15866,"lixos",68,-9.143814,38.707256,[573, 574, 575, 576],[69, 74, 75, 76],5560).
registo(15865,"lixos",66,-9.144288,38.70742,[565, 566, 567, 568, 569, 570],[67, 88, 89, 90, 91],2720).
registo(15864,"lixos",65,-9.146958,38.70847,[557, 558, 559, 560, 561, 562, 563, 564],[35, 66],3460).
```

Figura 1: Registos base de conhecimento

Tal como é possível observar, cada predicado "registo" possui um identificador de ponto de recolha, tipo de lixo, identificador de registo, latitude, longitude, contentores, registos adjacentes e quantidade de lixo do ponto.

Por último, é de notar que para a realização/teste de determinadas funcionalidades foi necessário considerar uma versão reduzida do *dataset*, de modo a diminuir o número de arestas entre os nodos e desta forma obter resultados para análise.

PESQUISA NÃO INFORMADA

Para efetuar o cálculo dos diferentes percursos, considerando uma origem e um destino, foram implementados diferentes algoritmos de **pesquisa não informada**. Esta estratégia de procura utiliza apenas informações disponíveis na definição do problema.

É de notar que os algoritmos de pesquisa não informada foram aplicados às diversas funcionalidades propostas no trabalho:

- ⇨ Gerar circuitos de forma indiferenciada/seletiva, caso existam.
- ⇨ Identificar circuitos com mais pontos de recolha (por tipo de resíduo).
- ⇨ Escolher o circuito mais rápido (de menor distância).
- ⇨ Selecionar o circuito mais eficiente (pelo quantidade de lixo recolhida).

PROFUNDIDADE PRIMEIRO

Os algoritmos de profundidade primeiro (*depth first*) expandem sempre um dos nodos no maior nível de profundidade da árvore. Esta estratégia de procura expande nós em níveis de profundidade mais superficiais apenas quando a procura atinge um nodo que não possui expansão e esse mesmo nodo não se trata do nodo objetivo. Dado que o nó expandido era o mais profundo, os seus sucessores serão ainda mais profundos e serão agora os nós mais profundos.

Em termos de memória, a pesquisa em profundidade necessita de armazenar um único caminho desde a raiz até ao último nodo folha e também os nodos irmãos não expandidos de cada nodo do caminho. Deste modo, no que toca à **complexidade** deste algoritmo em termos **temporal** é dada por $\Theta(b^m)$ sendo que b representa o fator de ramificação e m a máxima profundidade da árvore. Em termos **espaciais**, a complexidade é representada por $\Theta(b * m)$.

Este algoritmo DFS deve ser utilizado quando queremos encontrar todos os caminhos possíveis. Para problemas em que existem muitas soluções, os algoritmos em profundidade apresentam melhor performance do que os algoritmos em largura, dado que tem uma maior chance de encontrar a solução após explorar apenas uma pequena porção do grafo.

O lado negativo da pesquisa em profundidade reflete-se na possibilidade de ficar bloqueado no caminho errado. No caso de problemas com elevado nível de profundidade ou infinitas árvores de procura, este algoritmo nunca será capaz de concluir a pesquisa. Nestes casos, o algoritmo pode gerar ciclos infinitos e nunca retornar uma solução ou retornar um caminho mais longo do que o ótimo. Por esta razão, o algoritmo **não é completo nem ótimo**.

Uma das implementações possíveis para este tipo de pesquisa é através da chamada recursiva. Neste caso, a função é invocada para cada nodo filho. Esta foi a estratégia implementada neste projeto, traduzindo-se através de 2 implementações com estruturas semelhantes.

Para a primeira implementação, traduzida através do predicado "resolve_pp" (figura 2) foi necessário considerar uma declaração "goal(Destino)" externa ao predicado de pesquisa, de forma a denotar qual o nodo final. No caso da outra abordagem tomada, que se reflete pela regra "resolve_pp_h" (figura 3), o procedimento foi análogo com a diferença de que o nodo destino é recebido como parâmetro.

Estas abordagens seguem um raciocinio semelhante, na medida em que ambas recorrem à recursividade. Em cada iteração de procura é crucial ter conhecimento acerca do nodo atual, do histórico de nodos visitados e do caminho a percorrer. Em adição, também se mantém registo daa quantidade de resíduos recolhida, sem critério de seleção, correspondendo à recolha de **residuos indiferenciados**. Deste modo, para cada nodo é necessário verificar se este é adjacente ao próximo (predicado "adjacente"), se não é membro do histórico (predicado "membro") e se existe um registo (predicado "registo") desse nodo na base de conhecimento. Se estas condições se verificarem então o nodo pode fazer parte do caminho calculado. A condição de paragem destas implementações é muito semelhante, na medida em que ambas terminam a sua pesquisa ao encontrar o nodo final.

É de notar que nestas regras é implementado um mecanismo de **limitação da quantidade recolhida pelo camião**. Este mecanismo recorre à regra "verificaDeposito" (figura 4) que, a partir do percurso calculado, itera sobre o mesmo e verifica em que situações é necessário ir ao depósito descarregar o lixo.

O modo de procedimento desta regra tem em consideração o predicado núcleo do programa, "registo", e é dotada também de recursividade para iterar sobre os restantes elementos da lista. Como tal, foi necessário considerar um caso de paragem. Esta regra atua com recurso a uma variável "Qnt" que funciona como acumulador da quantidade transportada pelo camião. Deste modo, de cada vez que passa num ponto de recolha, o camião recolhe esse lixo, e essa quantidade é adicionada à variável "Qnt". Em cada iteração, verifica-se se a quantidade está prestes a exceder a quantidade máxima 15000, e em caso afirmativo é adicionada ao percurso uma ida ao depósito,

traduzida através da adição do nodo 200 e é efetuado o *reset* da variável “Qnt” para zero, de forma a efetuar uma nova contagem de quantidade de lixo a partir da próxima iteração.

Nas seguintes figuras é possível visualizar o código em *prolog* relativo aos predicados “resolve_pp”, “resolve_pp_h” e “verificaDeposito”:

```
%-------------------------------------- - - - - - - - - - - - - - - -
% PROFUNDIDADE (DFS - Depth-First Search)
% resolve_pp(1,C,Qnt,Dist,Num). => para goal(175).
% resolve_pp(150,C,Qnt,Dist,Num). => para goal(175).
% resolve_pp(140,C,Qnt,Dist,Num). s=> para goal(175).

resolve_pp(Nodo, Sol,Qnt,Dist,Number) :-
    profundidadeprimeiro1(Nodo,[Nodo],Caminho,Qnt),
    reverse(Caminho,Aux),
    verificaDeposito(Aux,Novo,QntDep,Dep),
    Res = [201 , Origem | Novo],
    insertAtEnd(200,Res,Sol),
    length(Sol,Number),
    calcula_dist(Sol,Dist).

profundidadeprimeiro1(Nodo,_,[],0) :-
    goal(Nodo).

profundidadeprimeiro1(Nodo, Historico, [ProxNodo|Caminho], Qnt) :-
    adjacente(Nodo,ProxNodo), % Os nodos devem ser adjacentes
    nao(membro(ProxNodo,Historico)),
    registo(_,_,Nodo,_,_,_,_,Qnt0),
    profundidadeprimeiro1(ProxNodo, [ProxNodo|Historico], Caminho, Qnt1), % Recursividade
    Qnt is Qnt1 + Qnt0.
```

Figura 2: Regra "resolve_pp"

```
%-------------------------------------- - - - - - - - - - - - - - - -
% Pesquisa em profundidade primeiro Multi_Estados

resolve_pp_h(Origem,Destino,Sol,Qnt,Number,Dep,Dist) :-
    profundidade(Origem,Destino,[Origem],Caminho,Qnt),
    reverse(Caminho,Aux),
    verificaDeposito(Aux,Novo,QntDep,Dep),
    Res = [201 , Origem | Novo],
    insertAtEnd(200,Res,Sol),
    length(Sol,Number),
    calcula_dist(Sol,Dist).

profundidade(Origem,Destino,_,[],0) :- Origem == Destino,!.

profundidade(Origem,Destino,His,[R|Solucao],Qnt) :-
    adjacente(Origem,R),
    nao(membro(R,His)),
    registo(_,_,R,_,_,_,_,Qnt0),
    profundidade(R,Destino,[R|His],Solucao,Qnt1),
    Qnt is Qnt1 + Qnt0.
```

Figura 3: Regra resolve_pp_h

```
%------------------------------- - - - - - - - - - -  -  -  -  -  -
% Predicado que verifica quando se deve ir ao depósito
% verificaDeposito : Caminho , Resultado , Quantidade , Visitas ao depósito -> {V,F}

verificaDeposito(Caminho,Novo,Qnt,Dep) :-
    verificaDeposito2(Caminho,[],Novo,Qnt,Dep).

verificaDeposito2([],Novo,Novo,0,1).
verificaDeposito2([Nodo|Caminho],Aux,Novo,Qnt,Dep) :-
    registo(_,_,Nodo,_,_,_,_,Qnt0),
    verificaDeposito2(Caminho,Lst,Novo,Qnt1,DepAux),
    ( Qnt1 + Qnt0 < 15000 -> Lst = [Nodo|Aux], Qnt is Qnt1 + Qnt0,
      Dep is DepAux;
      Lst = [200,Nodo|Aux], Qnt is 0 , Dep is DepAux + 1).
```

Figura 4: Regra "verificaDeposito"

PESQUISA SELETIVA

A resposta ao requisito de procura seletiva de acordo com o tipo de lixo em cada ponto de recolha foi concretizada recorrendo ao algoritmo de profundidade primeiro. Para isso, foram consideradas duas abordagens complementares: a primeira efetua um percurso passando apenas por pontos de recolha do tipo de lixo requisitado; a segunda efetua um percurso por todos os tipos de lixo, recolhendo apenas o tipo de lixo selecionado. Ambos os métodos efetuam a pesquisa recorrendo a recursividade e de modo análogo à implementação anteriormente referida.

Na figura seguinte é possível observar a implementação da primeira abordagem mencionada:

```
%------------------------------------ - - - - - - - - - - - - - - -
% PESQUISA SELETIVA - de acordo com o tipo de lixo (gera caminhos só com esse tipo de lixo)
% pp_seletiva(81,93,C,"papel e cartão",Qnt,Num,Dist).

pp_seletiva(Origem,Destino,Sol,Tipo,Qnt,Number,Dist) :-
    profundidade_sel(Origem,Destino,[Origem],Caminho,Tipo,Qnt),
    reverse(Caminho,Aux),
    verificaDeposito(Aux,Novo,QntDep,Dep),
    Res = [201 | Novo],
    insertAtEnd(200,Res,Sol),
    length(Sol,Number),
    calcula_dist(Sol,Dist).

profundidade_sel(Destino,Destino,H,D,Tipo,0) :- inverso(H,D).

profundidade_sel(Origem,Destino,His,C,Tipo,Qnt) :-
    adjacente_sel(Origem,Prox,Tipo,Qnt0),
    nao(membro(Prox,His)),
    profundidade_sel(Prox,Destino,[Prox|His],C,Tipo,Qnt1),
    Qnt is Qnt1 + Qnt0.
```

Figura 6: Profundidade primeiro seletiva versão 1

Esta primeira abordagem tem também em consideração a limitação da capacidade do camião a 15000, de modo análogo ao referido anteriormente, recorrendo à regra "verificaDeposito" (figura 4). Além disso, recorre a uma função particular ao feito da seleção "adjacente_sel" que calcula um nodo adjacente que cumpre a restrição de possuir um determinado tipo de lixo (figura 6). Esta última verifica se o próximo nodo é um ponto de recolha cujo tipo de lixo equivale ao tipo pretendido.

```
%--------------------------------- - - - - - - - - - - - - - - - - -
% Predicado que verifica se 2 nodos são adjacentes consoante o tipo de lixo.
% Auxiliar para a pesquisa seletiva.
% adjacente_sel_qtd : Nodo, ProxNodo, Tipo de Lixo, Qtd Recolhida -> {V,F}

adjacente_sel(Nodo,ProxNodo,Tipo,Qnt0) :-
    registo(_,T,Nodo,_,_,_,Adj,Qnt0),
    registo(_,T0,ProxNodo,_,_,_,_,_),
    T == Tipo,
    T0 == Tipo,
    membro(ProxNodo,Adj).

adjacente_sel(Nodo,ProxNodo,Tipo,Qnt0) :-
    registo(_,T,ProxNodo,_,_,_,Adj,Qnt0),
    registo(_,T0,Nodo,_,_,_,_,_),
    T == Tipo,
    T0 == Tipo,
    membro(Nodo,Adj).
```

Figura 7: Regra "adjacente_sel"

Na segunda abordagem a ideia foi efetuar um percurso entre 2 pontos e apenas recolher o tipo de lixo requirido. Para isso criou-se a regra "pp_seletiva2" (figura 7) que, efetuando um algoritmo de pesquisa em profundidade seguindo o raciocínio análogo aos anteriores, calcula a quantidade de lixo recolhido de um dado tipo. Este algoritmo recorre ao predicado "adjacente_sel2" (figura 8). De forma a recolher apenas resíduos do tipo pretendido, é efetuada uma operação condicional que apenas adiciona ao contador caso o tipo seja o pretendido. Esta implementação traduz-se em *prolog* do seguinte modo:

```
%------------------------------------ - - - - - - - - - - - - - - - -
% PESQUISA SELETIVA 2 - de acordo com o tipo de lixo (gera caminhos só com esse tipo de lixo)
% pp_seletiva(81,93,C,"papel e cartão",Qnt,Num,Dist).

pp_seletiva2(Origem,Destino,Sol,Tipo,Qnt,Number,Dep,Dist) :-
    profundidade_sel2(Origem,Destino,[Origem],Caminho,Tipo,Qnt),
    reverse(Caminho,Aux),
    verificaDeposito(Aux,Novo,QntDep,Dep),
    Res = [201 , Origem | Novo],
    insertAtEnd(200,Res,Sol),
    length(Sol,Number),
    calcula_dist(Sol,Dist).

profundidade_sel2(Origem,Destino,_,[],T,0) :- Origem == Destino,!.

profundidade_sel2(Origem,Destino,His,[R|Solucao],Tipo,Qnt) :-
    adjacente_sel2(Origem,R,Qnt0,T),
    nao(membro(R,His)),
    registo(_,_,R,_,_,_,_,Qnt0),
    profundidade_sel2(R,Destino,[R|His],Solucao,Tipo,Qnt1),
    ( T == Tipo -> Qnt is Qnt1 + Qnt0;
    Qnt is Qnt1).
```

Figura 8: Pesquisa seletiva em profundidade versão 2

```
%------------------------------ - - - - - - - - - - - - - - - - -
% Predicado que verifica se 2 nodos são adjacentes consoante o tipo de lixo.
% Auxiliar para a pesquisa seletiva.
% adjacente_sel_qtd : Nodo, ProxNodo, Tipo de Lixo, Qtd Recolhida -> {V,F}

adjacente_sel2(Nodo,ProxNodo,Qnt0,T) :-
    registo(_,T,Nodo,_,_,_,Adj,Qnt0),
    registo(_,_,ProxNodo,_,_,_,_,_),
    membro(ProxNodo,Adj).

adjacente_sel2(Nodo,ProxNodo,Qnt0,T) :-
    registo(_,T,ProxNodo,_,_,_,Adj,Qnt0),
    registo(_,_,Nodo,_,_,_,_,_),
    membro(Nodo,Adj).
```

Figura 9: Predicado adjacente_sel2

CIRCUITOS COM MAIS PONTOS DE RECOLHA

De forma a dar resposta a um outro objetivo do trabalho, foi necessário recorrer mais uma vez à pesquisa em profundidade. Esta funcionalidade trata de calcular o circuito com mais pontos de recolha.

Para isto, foi efetuada uma operação de *findall* para obter todos os percursos e de seguida, recorrendo a uma função auxiliar, efetuado o cálculo para verificar qual dos percursos possui mais pontos de recolha. Desta forma, foi implementada a regra "circuito_PR_pp" (figura 8) que após obter a lista de todas as soluções encontradas, invoca a regra "maiorPR" (figura 9) que percorre a lista de soluções e obtém o percurso com mais pontos de recolha.

```
%------------------------------------ - - - - - - - - - - - - - - - -
% Profundidade primeiro

circuito_PR_pp(Origem,Destino, Caminho) :-
    findall((C,Num),resolve_pp_h(Origem,Destino,C,Qnt,Num,Dep,Dist),Lst),
    maiorPR(Lst,Caminho). % predicado para calcular o caminho com mais pontos de recolha
```

Figura 10: Cálculo circuito mais pontos de recolha

```
%--------------------------------- - - - - - - - - - - - - - - - -
% Predicado que calcula o caminho com mais pontos de recolha
% maior : Lista , Elemento -> {V,F}

maiorPR([(P,X)],(P,X)).
maiorPR([(Px,X)|L], (Py,Y)) :- maiorPR(L,(Py,Y)), Y > X.
maiorPR([(Px,X)|L], (Px,X)) :- maiorPR(L,(Py,Y)), X >= Y.
```

Figura 11: Regra que cálcula o circuito com mais pontos de recolha

CIRCUITO MAIS RÁPIDO

Um outro tópico a abordar foi o cálculo do circuito mais rápido, ou seja, o circuito que percorre uma menor distância. Para concretizar tal feito, procedeu-se de modo análogo à abordagem anterior, implementou-se a regra "circuito_Rapido_pp" (figura 10) recorrendo ao predicado *findall* de modo a obter todas as soluções possíveis e de seguida recorre-se à regra "minimo" (figura 11) para obter o caminho que percorre a menor distância.

```
%------------------------------------ - - - - - - - - - - - - - - -
% Profundidade primeiro

circuito_Rapido_pp(Origem, Destino, Caminho) :-
    findall((C,Dist),resolve_pp_h(Origem,Destino,C,Qnt,Num,Dep,Dist),Lst),
    minimo(Lst,Caminho). % predicado para calcular o caminho com mais pontos de recolha
```

Figura 12: Calcular o circuito mais rápido

```
%------------------------------- - - - - - - - - - -  -  -
% Predicado que cálcula o elemento mínimo de uma lista
% minimo : Lista , Elemento -> {V,F}

minimo([(P,X)],(P,X)).
minimo([(Px,X)|L], (Py,Y)) :- minimo(L,(Py,Y)), X > Y.
minimo([(Px,X)|L], (Px,X)) :- minimo(L,(Py,Y)), X =< Y.
```

Figura 13: Calcular o mínimo elemento de uma lista de pares

CIRCUITO MAIS EFICIENTE

Por último, foi implementada a funcionalidade de cálculo de percurso mais eficiente recorrendo a algoritmos *depth first.* Para isso, o critério de eficiência considerado foi a quantidade de lixo recolhido, sendo que um circuito que recolhe mais resíduos é considerado mais eficiente do que um circuito que recolhe menor quantidade. Para isso, procedeu-se de modo análogo às funcionalidades anteriormente referidas, criando a regra "circuito_eficiente" (figura 12) que recorre ao predicado *findall* e posteriormente ao predicado "maiorPR" (figura 9) para calcular o percurso que consegue recolher mais resíduos.

```
%------------------------------------ - - - - - - - - - - - - - - - -
% CIRCUITO MAIS EFICIENTE
% O circuito mais eficiente é aquele que consegue recolher mais quantidade de resíduos

circuito_eficiente(Origem,Destino,Caminho) :-
    findall((C,Qnt),resolve_pp_h(Origem,Destino,C,Qnt,Num,Dep,Dist),Lst),
    maiorPR(Lst,Caminho).
```

Figura 14: Circuito mais eficiente

PROFUNDIDADE PRIMEIRO LIMITADA

Além do algoritmo de pesquisa em profundidade anteriormente referido e as funcionalidades por ele implementadas, foi também relevante considerar a implementação em **profundidade limitada**. Esta procura contorna algumas falhas da procura DFS através da limitação da profundidade.

A sua implementação refletiu-se através do predicado "resolve_pp_limitada" (figura 13) que recorre também a profundidade, tal como a procura em profundidade não limitada. O que as distingue é a existência de um contador "profundidade" que em cada nodo visitado é decrementado. Este contador não pode atingir valores negativos e a procura é terminada quando o nodo objetivo é encontrado ("goal(Destino)"). Além disso, esta regra também calcula a quantidade de resíduos recolhidos, o número de pontos de recolha por onde passa e a distância percorrida. Em adição, é considerado também o método de controlo de capacidade do camião (15000 litros), com recurso à regra "verificaDeposito" (figura 4) exposta anteriormente.

No que toca à **complexidade** deste algoritmo, esta pode ser avaliada em termos termos **temporal** e **espacial**. Para o primeiro, a complexidade é dada por $\Theta(b^l)$ sendo que b representa o fator de ramificação e l a profundidade considerada para a pesquisa. Em termos espaciais, a complexidade é representada por $\Theta(b * l)$. Este algoritmo DLS deve ser utilizado quando temos conhecimento que a solução ótima se encontra num determinado nível de profundidade. Desta forma, existe maior probabilidade de a encontrar e evita-se os problemas causados pela DFS. Esta procura pode ser **completa** e **não ótima** caso a profundidade seja tal que seja possível encontrar uma solução. No caso em que a profundidade seja muito reduzida, este algoritmo **não é completo nem ótimo**.

```
%-------------------------------------- - - - - - - - - - - - - - - -
% Pesquisa em profundidade limitada

resolve_pp_limitada(Profundidade, Nodo, Sol,Qnt,Dist,Number) :-
    pp_limitada(Profundidade,Nodo,[Nodo],Caminho,Qnt),
    reverse(Caminho,Aux),
    verificaDeposito(Aux,Novo,QntDep,Dep),
    Res = [201 , Origem | Novo],
    insertAtEnd(200,Res,Sol),
    length(Sol,Number),
    calcula_dist(Sol,Dist).

pp_limitada(Profundidade,Nodo,_,[],0) :-
    goal(Nodo).

pp_limitada(Profundidade,Nodo,Historico,[ProxNodo|Caminho],Qnt) :-
    Profundidade > 0,
    adjacente(Nodo,ProxNodo),
    ProfNova is Profundidade - 1,
    nao(membro(ProxNodo,Historico)),
    registo(_,_,Nodo,_,_,_,_,Qnt0),
    pp_limitada(ProfNova,ProxNodo,[ProxNodo|Historico],Caminho,Qnt1),
    Qnt is Qnt1 + Qnt0.
```

Figura 15: Pesquisa em profundidade limitada

CIRCUITOS COM MAIS PONTOS DE RECOLHA

Através deste algoritmo de procura foi implementado um mecanismo de cálculo de circuitos com mais pontos de recolha. Para isso, criou-se o predicado "circuito_PR_limitada" (figura 14) que efetua o cálculo de percursos com uma profundidade arbitrária (por exemplo, 100) recorrendo ao predicado *findall* para encontrar todos os caminhos. A partir dessas soluções encontradas é efetuado o cálculo da melhor, que será o que possui o maior número de pontos de recolha. Este cálculo é efetuado com recurso ao predicado "maiorPR" (figura 9).

```
%-------------------------------------- - - - - - - - - - - - - - - -
% Profundidade Limitada

circuito_PR_limitada(Origem, Caminho) :-
    findall((C,Num),resolve_pp_limitada(100,Origem,C,Qnt,Dist,Num),Lst),
    maiorPR(Lst,Caminho). % predicado para calcular o caminho com mais pontos de recolha
```

Figura 16: Circuito mais pontos recolha limitado

CIRCUITO MAIS RÁPIDO

Além do cálculo do circuito com mais pontos de recolha, procedeu-se também ao cálculo do circuito mais rápido (percorre menor distância) com recurso ao algoritmo de profundidade primeiro limitado. Para isso, implementou-se a regra "circuito_Rapido_limitada" (figura 15) que recorreu ao predicado *findall* de forma a obter todas as soluções e, entre essas soluções, calcula a melhor, ou seja, a que percorre a minima distância no percurso (regra "minimo", figura 11). Na figura 15 está presente a implementação adotada e a titulo de exemplo considerou-se a profundidade igual a 100.

```
%-------------------------------------- - - - - - - - - - - - - - - -
% Profundidade Limitada

circuito_Rapido_limitada(Origem, Caminho) :-
    findall((C,Dist),resolve_pp_limitada(100,Origem,C,Qnt,Dist,Num),Lst),
    minimo(Lst,Caminho). % predicado para calcular o caminho com mais pontos de recolha
```

Figura 17: Circuito mais rápido limitado

PESQUISA EM LARGURA

Além da pesquisa em profundidade, foi também implementado um algoritmo de pesquisa em largura. Ao contrário das pesquisas DFS, o nodo origem é expandido primeiro e os nodos adjacentes à raiz são expandidos de seguida. O mesmo acontece para os seus sucessores, e assim em diante. Geralmente, todos os nodos numa profundidade *d* são expandidos primeiro do que os nodos em profundidade *d+1*. Esta estratégia de procura começa por considerar todos os caminhos de comprimento 1, 2, e assim sucessivamente. Caso exista uma solução, é certo que a pesquisa em largura é capaz de a encontrar. Além disso, os primeiros caminhos encontrados estão presentes em níveis mais superficiais da árvore de procura.

Por outro lado, esta procura consome um nível de memória e tempo muito elevado para efetuar a procura, sendo que a sua **complexidade** pode ser avaliada em termos termos **temporal** e **espacial**. Para o primeiro, a complexidade é dada por $\Theta(b^{d+1})$ sendo que *b* representa o fator de ramificação. Em termos espaciais, complexidade é análoga e representada por $\Theta(b^{d+1})$. Este algoritmo BFS deve ser utilizado quando queremos encontrar o menor caminho, por exemplo. Além disso, é de notar que este algoritmo **não é completo nem ótimo**.

Deste modo, considerou-se conveniente implementar para o atual projeto um algoritmo de pesquisa em largura que respeita a estratégia anteriormente referida. Este algoritmo reflete-se na regra "resolve_largura" (figura 16) e recorre a funções como "append" e "bagof" do *prolog* de forma a encontrar as soluções possíveis para todos os nodos adjacentes. Em adição, a sua implementação recorreu a recursividade.

```
%------------------------------------ - - - - - - - - - - - - - - - -
% LARGURA (BFS - Breadth-First Search)

resolve_largura( Origem, Sol, Number, Qnt, Dist)  :-
  breadthfirst( [ [Origem] ], Caminho),
  verificaDeposito(Caminho,Novo,QntDep,Dep),
  Res = [201 | Novo],
  insertAtEnd(201,Res,Sol),
  length(Sol,Number),
  calcula_qnt(Sol,Qnt),
  calcula_dist(Sol,Dist),
  length(Sol,Number).

breadthfirst( [ [Origem | Caminho] | _], [Origem | Caminho])  :-
    goal( Origem).

breadthfirst( [Caminho | Caminhos], Solucao)  :-
    extend( Caminho, NovoCaminho),
    append( Caminhos, NovoCaminho, Paths1),
    breadthfirst( Paths1, Solucao).

extend( [Origem | Caminho], NovoCaminho)  :-
    bagof( [NovoNodo, Origem | Caminho],
         ( adjacente( Origem, NovoNodo), \+ member( NovoNodo, [Origem | Caminho] ) ),
         NovoCaminho), !.

extend( Caminho, [] ).
```

Figura 18: Pesquisa em largura

Tal como é possível observar, este predicado recorre à regra de controlo de capacidade do camião, "verificaDeposito" (figura 4) anteriormente exposta. A implementação adotada segue o princípio referido anteriormente, visitando primeiramente todos os nodos adjacentes e só depois os nodos de outro nível de profundidade. É de notar que este algoritmo efetua também o cálculo da distância percorrida e quantidade de resíduos recolhidos.

PESQUISA INFORMADA

Os algoritmos de **pesquisa informada** são distintos dos de pesquisa não informada na medida em que acrescentam informação nova no cálculo do percurso. É uma pesquisa efetuada com recurso a heurísticas, ou seja, o algoritmo é dotado de pistas sobre a adequação de diferentes estados.

Para este tipo de procura foram implementados dois tipos de algoritmos: **gulosa** (*greedy search*) e **A-Estrela**. Nas seguintes subsecções será abordado como cada um deles foi implementado neste trabalho.

GULOSA

Uma das abordagens de pesquisa informada foi a procura gulosa (*greedy).* O procedimento deste tipo de procura baseia-se em expandir o nó que parece estar mais perto da solução e visita-lo. Deste modo, nos algoritmos deste tipo tenta-se sempre visitar os nodos que à partida são o melhor caminho. Por exemplo, a decisão entre 2 nodos pode ser feita tendo em conta o menor custo. A função que trata do cálculo desse custo é denominada como **função heurística** e é frequentemente denotada da seguinte forma:

$$h(n) = custo\ estimado\ do\ caminho\ mais\ barato\ desde\ n\ até\ ao\ nodo\ objetivo$$

A única restrição para a definição da função heurística *h* é que *h(n) = 0* quando *n* é o nodo objetivo.

Esta pesquisa é dotada dos mesmos problemas da pesquisa em progundidade dado que segue um único caminho até ao destino. Deste modo, a **complexidade** deste algoritmo em termos **temporal** e **espacial** é análoga e, no pior caso, pode ser representada por uma função exponencial *b* elevada a *m* (máxima profundidade do espaço de procura), $\Theta(b^m)$. Trata-se um algoritmo **não ótimo e incompleto** porque pode ficar bloqueado num caminho infinito.

Dada que esta é uma procura que utiliza heurísticas, considero conveniente efetuar o cálculo do percurso mais rápido (mais curto) recorrendo a este tipo de procura, através do desenvolvimento das heurísticas adequadas.

CIRCUITO MAIS RÁPIDO

Considerando a natureza deste algoritmo, foi conveniente realizar esta funcionalidade recorrendo à pesquisa gulosa. Para isso, foi necessário implementar regra "resolve_gulosa" (figura 17) que utiliza uma heurística (figura 18) de modo a calcular a distância euclidiana entre 2 pontos e deste modo escolher o melhor caminho. A implementação adotada segue nas seguintes figuras:

```
%-------------------------------------- - - - - - - - - - - - - - - -
% GULOSA

resolve_gulosa(Nodo,Sol/Custo,Number,Qnt) :-
    registo(_,_,Nodo,_,_,_,_,Estima),
    agulosa([[Nodo]/0/Estima],InvCaminho/Custo/_),
    verificaDeposito(InvCaminho,Novo,QntDep,Dep),
    Res = [201 | Novo],
    insertAtEnd(201,Res,Sol),
    length(Sol,Number),
    calcula_qnt(Sol,Qnt),
    calcula_dist(Sol,Dist),
    length(Sol,Number).


agulosa(Caminhos, Caminho) :-
    obtem_melhor_g(Caminhos,Caminho),
    Caminho = [Nodo|_]/_/_,goal(Nodo).

agulosa(Caminhos,SolucaoCaminho) :-
    obtem_melhor_g(Caminhos,MelhorCaminho),
    seleciona(MelhorCaminho,Caminhos,OutrosCaminhos),
    expande_gulosa(MelhorCaminho,ExpCaminhos),
    append(OutrosCaminhos,ExpCaminhos,NovoCaminhos),
    agulosa(NovoCaminhos,SolucaoCaminho).

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

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

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

expande_gulosa(Caminho, ExpCaminhos) :-
    findall(NovoCaminho,adjacente3(Caminho,NovoCaminho),ExpCaminhos).
```

Figura 19: Algoritmo gulosa menor distância

```
%--------------------------------- - - - - - - - - - - - - - - - -
% Adjacente3 : Elemento, Lista -> {V,F}

adjacente3([Nodo|Caminho]/Custo/_,[ProxNodo,Nodo|Caminho]/NovoCusto/Est) :-
    adjacente2(Nodo,ProxNodo),
    distancia(Nodo,ProxNodo,PassoCusto),
    \+ member(ProxNodo,Caminho),
    NovoCusto is Custo + PassoCusto,
    estima(ProxNodo,Est).

%--------------------------------- - - - - - - - - - - - - - - - -
% Adjacente2 : Elemento, Elemento -> {V,F}

adjacente2(Nodo,ProxNodo) :-
    registo(_,_,Nodo,_,_,_,Adj,_),
    membro(ProxNodo,Adj).

adjacente2(Nodo,ProxNodo) :-
    registo(_,_,ProxNodo,_,_,_,Adj,_),
    membro(Nodo,Adj).
```

Figura 20: Heurística de menor distância

```
%--------------------------------- - - - - - -
% estima : Elemento, Estimativa -> {V,F}

estima(ProxNodo,Est) :-
    goal(Destino),
    distancia(ProxNodo,Destino,Est).
```

Figura 21: Regra para estimar o custo entre 2 nodos

A-ESTRELA

O outro tipo de pesquisa informada considerada foi a procura A-Estrela. Este algoritmo trata-se de uma melhoria do algoritmo gulosa, na medida em que não considera apenas o custo de um nodo para o outro, mas também considera o custo de ir desse nodo até ao nodo final. Assim, a procura A* evita expandir caminhos que se demonstram muito dispendiosos. Com base dois princípios considero conveniente utilizar este algoritmo no cálculo do percurso mais rápido (menor distância) de forma a obter melhores resultados em relação aos outros tipos de procura abordados.

O lado negativo da pesquisa A Estrela é que utiliza uma elevada quantidade de memória. É de notar que a **complexidade** deste algorimo é análoga em termos **temporal** e **espacial**, e dado que foi utilizada a mesma heurística do algoritmo gulosa, então a complexidade revelou se análoga para os dois algoritmos representando-se por uma função exponencial *b* elevada a *m* (máxima profundidade da procura), $\Theta(b^m)$. Este algoritmo é **completo** e **ótimo**. No entanto, para um número de nodos exponencial a memória ocupada é muito elevada.

CIRCUITO MAIS RÁPIDO

Este algoritmo foi aplicado numa das funcionalidades propostas neste trabalho, de modo a calcular o circuito mais rápido, ou seja, aquele que possui uma menor distância, recorrendo às heurísticas e predicados de cálculo de distância (figuras 18, 19 e 20). Em concreto, a implementação deste algoritmo traduz-se em *prolog* da seguinte forma:

```
%------------------------------------ - - - - - - - - - - - - - - - -
% A ESTRELA

resolve_aestrela(Nodo,Sol/Custo,Number,Qnt) :-
    registo(_,_,Nodo,_,_,_,_,Estima),
    aestrela([[Nodo]/0/Estima], InvCaminho/Custo/_),
    verificaDeposito(InvCaminho,Novo,QntDep,Dep),
    Res = [201 | Novo],
    insertAtEnd(201,Res,Sol),
    length(Sol,Number),
    calcula_qnt(Sol,Qnt),
    calcula_dist(Sol,Dist),
    length(Sol,Number).

aestrela(Caminhos,Caminho) :-
    obtem_melhor(Caminhos,Caminho),
    Caminho = [Nodo|_]/_/_,goal(Nodo).

aestrela(Caminhos,SolucaoCaminho) :-
    obtem_melhor(Caminhos,MelhorCaminho),
    seleciona(MelhorCaminho,Caminhos,OutrosCaminhos),
    expande_aestrela(MelhorCaminho,ExpCaminhos),
    append(OutrosCaminhos,ExpCaminhos,NovoCaminhos),
    aestrela(NovoCaminhos,SolucaoCaminho).

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

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

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

expande_aestrela(Caminho,ExpCaminhos) :-
    findall(NovoCaminho,adjacente3(Caminho,NovoCaminho), ExpCaminhos).
```

Figura 22: Algoritmo A Estrela

PREDICADOS AUXILIARES

Durante a realização deste trabalho foi necessário recorrer diversas vezes a alguns predicados auxiliares em diferentes funcionalidades. Por essa razão, nesta secção é possível observar em detalhe a sua implementação.

Considerou-se conveniente implementar a extensão do meta-predicado "não" (figura 21) de modo a poder efetuar a negação de predicados. Deste modo, se o predicado *Questao* possuir o valor **Falso**, a regra *não* irá opor este valor de verdade. O mesmo acontece caso o valor de verdade de *Questao* seja **Verdade**.

```
%------------------------------- - - - - - - - - - -
% Extensao do meta-predicado nao: Questao -> {V,F}

nao( Questao ) :-
    Questao, !, fail.
nao( Questao ).
```

Figura 23: Meta-predicado "não"

Além disso, também foi relevante a implementação do predicado "membro" (figura 22) de modo a averiguar se um dado elemento pertence a uma lista. Caso pertença então o predicado retorna o valor **Verdadeiro**. Caso contrário, o valor deverá ser **Falso.**

```
%------------------------------- - - - - - - - - - - - - - - - -
% Extensão do predicado membro. Verifica se um elemento pertence a uma lista
% membro : Elemento, Lista -> {V,F}

membro(X, [X|_]).
membro(X, [_|Xs]):-
    membro(X, Xs).
```

Figura 24: Extensão do predicado membro

Em adição, foi necessário implementar o predicado "inverso" (figura 23) que trata de calcular o inverso de uma lista recorrendo a recursividade.

```
%------------------------------- - - - - - - - - - -
% Predicado que cálcula o inverso de uma lista
% inverso : Lista , Lista -> {V,F}

inverso(Xs,Ys) :-
    inverso(Xs,[],Ys).

inverso([], Xs, Xs).
inverso([X|Xs], Ys, Zs) :- inverso(Xs, [X|Ys], Zs).
```

Figura 25: Predicado inverso

O predicado “minimo” (figura 24) também se revelou de extrema importância como auxílio às funcionalidades, de modo a calcular o menor caminho de acordo com um determinado critério. A sua implementação segue de seguida, recorrendo a recursividade:

```
%-------------------------------- - - - - - - - - - -  -
% Predicado que cálcula o elemento mínimo de uma lista
% minimo : Lista , Elemento -> {V,F}

minimo([(P,X)],(P,X)).
minimo([(Px,X)|L], (Py,Y)) :- minimo(L,(Py,Y)), X > Y.
minimo([(Px,X)|L], (Px,X)) :- minimo(L,(Py,Y)), X =< Y.
```

Figura 26: Extensão do predicado minimo

Adicionalmente foi implementado o predicado “distancia” (figura 25) muito utilizado do longo deste trabalho que trata de calcular a distância entre 2 pontos de recolha. Para o problema em mãos, considerou-se a distância euclidiana.

```
%-------------------------------- - - - - - - - - - -  -  -  -
% Predicado que calcula a distância euclidiana entre dois nodos
% distancia : Nodo, Nodo, Distancia -> {V,F}

distancia(Nodo1, Nodo2, Dist) :-
    registo(_,_,Nodo1,Lat1,Long1,_,_,_),
    registo(_,_,Nodo2,Lat2,Long2,_,_,_),
    Dist is sqrt((Lat2-Lat1)^2 + (Long2-Long1)^2).
```

Figura 27: Predicado distancia

Um outro predicado crucial para o desenvolvimento deste trabalho foi o predicado “adjacente” (figura 26) que trata de verificar se dois pontos são adjacentes.

```
%-------------------------------- - - - - - - - - - -  -  -  -
% Predicado que verifica se 2 quaisquer nodos são adjacentes
% adjacente : Nodo, ProxNodo -> {V,F}

adjacente(Nodo,ProxNodo) :-
    registo(_,_,Nodo,_,_,_,Adj,_),
    membro(ProxNodo,Adj).
```

Figura 28: Predicado adjacente

Também foi necessário efetuar a definição de um predicado que seleciona um elemento de uma lista. Este predicado traduz-se em *prolog* da seguinte forma:

```
%--------------------------------- - - - - - - - - - -  .
% Seleciona : Elemento, Lista, Lista -> {V,F}

seleciona(E,[E|Xs],Xs).
seleciona(E,[X|Xs],[X|Ys]) :- seleciona(E,Xs,Ys).
```

Figura 29: Predicado seleciona

O seguinte predicado trata de calcular a quantidade de resíduos recolhidos, dado um caminho entre 2 pontos:

```
%--------------------------------- - - - - - - - - - .
% Calcula a quantidade de lixo recolhida num caminho
% calcula_qnt : Caminho, Qnt -> {V,F}

calcula_qnt([],0).
calcula_qnt([C|T],Qnt) :-
    registo(_,_,C,_,_,_,_,Qnt0),
    calcula_qnt(T,Qnt1),
    Qnt is Qnt0 + Qnt1.
```

Figura 30: Predicado calcula_qnt

Adicionalmente, foi conveniente implementar um predicado que efetua o cálculo da distância percorrida num dado percurso, "calcula_dist":

```
%--------------------------------- - - - - - - - -
% Calcula a distancia percorrida num caminho
% calcula_dist : Caminho, Dist -> {V,F}

calcula_dist([],0).
calcula_dist([C],0).
calcula_dist([C,C1|T],Dist) :-
    calcula_dist([C1|T],Dist1),
    distancia(C,C1,Dist0),
    Dist is Dist0 + Dist1.
```

Figura 31: Predicado calcula_dist

Por último, é conveniente referir a implementação de um predicado que, dada uma lista e um elemento, insere esse elemento no final da lista (figura 31).

```
%--------------------------------- - - - - - - - - - -  -  -
% Predicado que insere um elemento no final de uma lista
% insertAtEnd : Elemento , Lista , Lista -> {V,F}

insertAtEnd(X,[ ],[X]).
insertAtEnd(X,[H|T],[H|Z]) :- insertAtEnd(X,T,Z).
```

Figura 32: Predicado insertAtEnd

ANÁLISE DE RESULTADOS

De modo a verificar o espaço ocupado e tempo decorrido na execução de cada algoritmo, considero conveniente efetuar uma análise de resultados baseada nestes critérios.

Esta análise foi efetuada com recurso aos predicados "time" e "statistics" do *prolog*, que permitem avaliar o tempo de execução do algoritmo e a memória ocupada pelo mesmo, respetivamente. Deste modo, para efetuar as medições relativas a cada algoritmo foi implementado um predicado que efetua esse cálculo.

Para a **procura em profundidade não limitada**, implementou-se o algoritmo apresentado de seguida:

```
% Procura em profundidade
estatistica_profundidade(C, Mem) :-
    statistics(global_stack, [G1,L1]),
    time(resolve_pp_h(1,35,C,Qnt,Num,Dep,Dist)),
    statistics(global_stack, [G2,L2]),
    Mem is G2 - G1.
```

Figura 33: Predicado estatistica_profundidade

No caso da **procura em profundidade limitada**, o procedimento foi análogo e o predicado implementado "estatistica_profundidade_limitada" pode ser observado de seguida:

```
% Procura em profundidade limitada
estatistica_profundidade_limitada(C, Mem) :-
    statistics(global_stack, [G1,L1]),
    time(resolve_pp_limitada(25,1,C,Qnt,Num,Dist)),
    statistics(global_stack, [G2,L2]),
    Mem is G2 - G1.
```

Figura 34: Predicado estatística_profundidade_limitada

Na **procura em largura**, procedeu-se de modo semelhante e implementou-se o predicado "estatistica_largura" de modo a efetuar o cálculo da memória utilizada e tempo decorrido na execução do algoritmo em largura:

```
% Procura em largura
estatistica_largura(C, Mem) :-
    statistics(global_stack, [G1,L1]),
    time(resolve_largura(1,C,Num,Qnt,Dist)),
    statistics(global_stack, [G2,L2]),
    Mem is G2 - G1.
```

Figura 35: Predicado estatistica_largura

No que toca à **procura gulosa** o raciocinio foi semelhante e pode ser traduzido pelo seguinte predicado:

```
% Procura gulosa
estatistica_gulosa(C, Mem, Dist) :-
    statistics(global_stack, [G1,L1]),
    time(resolve_gulosa(1,C,Num,Qnt,Dist)),
    statistics(global_stack, [G2,L2]),
    Mem is G2 - G1.
```

Figura 36: Predicado estatistica_gulosa

Por último, para a **procura A Estrela** foi criado um predicado que atua de modo análogo aos anteriores e efetua o cálculo do tempo decorrido e memória ocupada na execução deste algoritmo:

```
% Procura a estrela
estatistica_a_estrela(C, Mem, Dist) :-
    statistics(global_stack, [G1,L1]),
    time(resolve_aestrela(1,C,Num,Qnt,Dist)),
    statistics(global_stack, [G2,L2]),
    Mem is G2 - G1.
```

Figura 37: Predicado estatistica_a_estrela

Adicionalmente também foi implementada uma regra para efetuar a medição do tempo e memória ocupados pela funcionalidade de encontrar o melhor caminho com recurso ao algoritmo em profundidade e *findall*. Esta regra traduziu-se em *prolog* da seguinte forma:

```
% Procura profundidade findall
estatistica_profundidade_findall(C, Mem) :-
    statistics(global_stack, [G1,L1]),
    time(circuito_Rapido_pp(1,10,C)),
    statistics(global_stack, [G2,L2]),
    Mem is G2 - G1.
```

Figura 38: Predicado estatistica_profundidade_findall

Invocando os predicados anteriormente expostos e considerando para todas as medições o cálculo do tempo e memória desde o nodo origem 1 até ao nodo destino 35, obteve-se os seguintes resultados:

```
?- estatistica_profundidade(C,Mem).
% 1,433 inferences, 0.000 CPU in 0.000 seconds (?% CPU, Infinite Lips)
C = [201,1,2,3,4,5,6,7,8,9,10,11,12,200,13,14,15,16,17,18,200,19,20,21,22,23,24,25,26,27,28,29,30,200,31,32,
33,34,200,35,200,201],
Mem = 63192 .
```

Figura 39: Medições procura em profundidade

```
?- estatistica_profundidade_limitada(C,Mem).
% 1,967 inferences, 0.000 CPU in 0.000 seconds (?% CPU, Infinite Lips)
C = [201,200,2,3,4,5,6,7,8,9,10,11,12,200,13,14,15,16,17,18,200,19,20,21,22,23,33,200,34,35,200,201],
Mem = 50352 ,
```

Figura 41: Medições procura em profundidade limitada

```
?- estatistica_largura(C,Mem).
% 579,526 inferences, 0.063 CPU in 0.064 seconds (98% CPU, 9272416 Lips)
C = [201,1,2,3,4,5,6,7,99,81,82,65,200,35,200,201],
Mem = 1813528 ,
```

Figura 40: Medições procura em largura

```
?- estatistica_gulosa(C,Mem,Dist).
% 8,389 inferences, 0.000 CPU in 0.002 seconds (0% CPU, Infinite Lips)
C = [201,1,96,106,105,88,65,35,200,201]/0.006608618332031465,
Mem = 56872,
Dist = 0.031496143919777726 ,
```

Figura 42: Medições procura gulosa

```
?- estatistica_a_estrela(C,Mem,Dist).
% 322,110 inferences, 0.063 CPU in 0.067 seconds (93% CPU, 5153760 Lips)
C = [201,1,70,69,68,67,89,36,200,35,200,201]/0.004865966603300689,
Mem = 3175920,
Dist = 0.033764486477425616 ,
```

Figura 43: Medições procura A Estrela

Através dos resultados obtidos é possível efetuar uma análise comparativa em termos de tempo de execução e utilização de memória:

	Tempo (segundos)	Espaço
DFS	0s	63192
DLS	0s	50352
BFS	0.064s	1813528
Greedy	0.002s	56872
A*	0.067s	3175920

Tabela 1: Tempos e memória ocupada por algoritmo

É possível efetuar uma comparação entre os diversos algoritmos dado que os tempos medidos tiveram em consideração os mesmos pontos de origem e destino. Além disso, estas medições foram efetuadas tendo em conta o ***dataset* original** e toda a sua informação. A partir da tabela acima é possível observar que os algoritmos que executaram em menos tempo foram os algoritmos em profundidade (não limitada e limitada), seguido do algoritmo gulosa que executou em 0.002 segundos. A memória ocupada por estas 3 estratégias foi semelhante e na ordem de

grandeza de 10^4. É de notar que, em termos de ocupação de memória, as procuras em largura e A Estrela destacam-se pelo seu uso excessivo, sendo que representam um aumento de cerca de 30 vezes em relação às outras procuras. As estratégias que demoraram mais tempo até encontrar 1 solução foram as em largura e A*, com 0.064 segundos e 0.067 segundos, respetivamente.

Além destas medições de performance de cada algoritmo, considero conveniente efetuar outro tipo de análise de modo a verificar se é possível obter a solução ótima a partir dos algoritmos de procura. Para isso, foram efetuados testes considerando um *dataset* reduzido que contém menor número de arestas e também foram efetuados testes com o *dataset* original.

Deste modo, considerando o ***dataset* reduzido**, foram efetuadas as seguintes medições para o cálculo do caminho mais curto entre os nodos 1 e 10. A distância percorrida entre os pontos de recolha é dada pela variável "Dist" no caso dos algoritmos gulosa e A Estrela**.**

```
?- estatistica_gulosa(C,Mem, Dist).
% 1,850 inferences, 0.000 CPU in 0.001 seconds (0% CPU, Infinite Lips)
C = [201,1,2,3,4,5,6,7,8,9,10,200,201]/0.01446894096845241,
Mem = 18352,
Dist = 0.04167424006029034 ,
```

Figura 44: Cálculo do caminho mais curto através da procura gulosa

```
?- estatistica_a_estrela(C,Mem, Dist).
% 5,690 inferences, 0.000 CPU in 0.001 seconds (0% CPU, Infinite Lips)
C = [201,1,2,3,4,5,6,7,8,9,10,200,201]/0.01446894096845241,
Mem = 65688,
Dist = 0.04167424006029034 ,
```

Figura 45: Cálculo do caminho mais curto através da procura A Estrela

```
?- estatistica_profundidade_findall(C,Mem).
% 285 inferences, 0.000 CPU in 0.000 seconds (?% CPU, Infinite Lips)
C =  ([201,1,2,3,4,5,6,7,8,9,10,200,201],0.04167424006029034),
Mem = 2936 ,
```

Figura 46: Cálculo do caminho mais curto através do findall da procura em profundidade

Os resultados obtidos podem ser visualizados em forma tabular:

	Tempo (segundos)	Espaço	Solução ótima?
DFS *Findall*	0s	2936	Sim
Greedy	0.001s	18352	Sim
A*	0.001s	65688	Sim

Tabela 2: Análise do caminho mais curto *dataset* reduzido

Estes resultados estão de acordo dado que o caminho encontrado por cada um deles foi o mesmo e a distância percorrida também. Além disso, através da análise da tabela 2 é possível observar que o mais rápido foi a procura em profundidade que demorou 0s e as procuras gulosa e A Estrela demoraram 0.001 segundos na sua execução. Em termos de memória ocupada na procura, podemos observar que a memória ocupada pelos algoritmos de procura informada foi muito maior do que a memória ocupada pelo algoritmo em profundidade. Além disso, entre os 2 algoritmos de procura informada é de realçar que o A Estrela ocupou mais memória do que o algoritmo gulosa.

Considerando agora o ***dataset* original** com todas as arestas, é possível observar resultados um pouco distintos. Para este cálculo considerou-se a origem o nodo 1 e o destino o nodo 10. Os resultados obtidos seguem nas seguintes figuras:

```
?- estatistica_gulosa(C,Mem, Dist).
% 6,677 inferences, 0.016 CPU in 0.001 seconds (1511% CPU, 427328 Lips)
C = [201,1,96,79,80,10,200,201]/0.0032514562573377035,
Mem = 37736,
Dist = 0.027010260409834286 ,
```

Figura 47: Estatísticas obtidas procura gulosa *dataset* original

```
?- estatistica_a_estrela(C,Mem, Dist).
% 329,524 inferences, 0.047 CPU in 0.044 seconds (107% CPU, 7029845 Lips)
C = [201,1,96,95,13,10,200,201]/0.0024194959186733304,
Mem = 2008688,
Dist = 0.026178300071169912 ,
```

Figura 48: Estatísticas obtidas A Estrela *dataset* original

Estes dados podem ser representados em forma tabular:

	Tempo (segundos)	Espaço	Solução ótima?
DFS *Findall*	-	-	Não
Greedy	0.001s	37736	Não
A*	0.044s	2008688	Sim

Tabela 3: Análise de resultados *dataset* original

Tal como é possível observar através da tabela 3, não foi possível obter um resultado recorrendo ao *findall* da procura em profundidade, dado que a dimensão do *dataset* é muito elevada. Em relação às procuras informadas, apenas foi possível obter a solução ótima recorrendo ao algoritmo A Estrela ocupando um elevado nível de memória. A procura gulosa ocupa menos memória mas não foi capaz de encontrar a solução ótima. Em relação ao tempo decorrido para cada uma destas procuras, é de notar que o tempo que a procura A Estrela demorou a executar foi maior.

Através desta análise de resultados considero que o algoritmo A Estrela é o melhor no cálculo do menor percurso entre 2 pontos. Caso o *dataset* possua uma dimensão mais reduzida e de menor escala, o algoritmo em profundidade poderia ser uma opção viável. No entanto, no problema atual o algoritmo A Estrela é o que melhor se adequa.

CONCLUSÃO

Após a realização deste projeto considero conveniente efetuar uma reflexão crítica do trabalho realizado.

No espetro positivo, considero relevante referir que foram implementados todos os tipos de procura, tanto a informada como não informada. Além disso, foi implementada também a versão completa do problema e todas as funcionalidades propostas estão operacionais.

A maior dificuldade foi conseguir utilizar os algoritmos dada a grande escalabilidade do *dataset*. Para contornar esse obstáculo, por vezes foi utilizado um *dataset* mais pequeno. Além disso, considero que existem pontos a melhorar, tais como a implementação das funcionalidades a outros algoritmos e a limitação da capacidade do camião em *runtime*.

Por fim, considero que o balanço do trabalho foi positivo e que os pontos fortes superam os negativos.

REFERÊNCIAS

[Russel,Norvig, 2009] RUSSEL, Stuart, NORVIG, Peter

"Artificial Intelligence - A Modern Approach, 3rd edition",

Pearson, 2009.

[Costa, Simões , 2008] COSTA, Ernesto, SIMÕES, Anabela,

"Inteligência Artificial-Fundamentos e Aplicações",

FCA, 2008.

LISTA DE SIGLAS E ACRÓNIMOS

BC Base de Conhecimento

DFS *Depth First Search*

BFS *Breadth First Search*

UC Unidade Curricular

SRCR Sistemas de Representação de Conhecimento e Raciocínio

DLS *Depth Limited Search*