Universidade do Minho
Escola de Engenharia

MÉTODOS DE RESOLUÇÃO DE PROBLEMAS E DE PROCURA

SISTEMA DE REPRESENTAÇÃO DE CONHECIMENTO E RACIOCÍNIO – 2º SEMESTRE – 2020/2021

TRABALHO PRÁTICO INDIVIDUAL

Ana Luísa Lira Tomé Carneiro
A89533 | JUNHO 2021, BRAGA

RESUMO

O seguinte relatório pretende-se descrever como foi desenvolvido o sistema de representação de conhecimento e raciocínio referente a um dos tópicos mais importantes da ciência nas áreas de matemática e computação que são os algoritmos de pesquisa e procura como forma de resolução problemas de *Classical Search,* isto é, um problema de pesquisa.

Para a realização deste projeto fomos incentivados a utilizar a linguagem de programação em lógica PROLOG. Esta ferramenta foi fundamental para a concretização não só das estratégias de pesquisa quer sejam para procura não informada como para a informada como também para todas as funcionalidades e regras implementadas.

Na primeira fase era proposto a formulação do problema em mãos como um problema de pesquisa, contextualizando as 5 componentes que fazem de um problema um problema de *Classical Search*. De seguida implementou-se um conjunto de algoritmos que tinham como objetivo determinar caminhos de um ponto inicial até ao seu destino permitindo ao camião recolher o lixo de cada um desses pontos. Por fim, desenvolveu-se um conjunto de funcionalidades que permitiram a obtenção de vários caminhos avaliados segundo critérios e indicadores de produtividade. Desta forma é possível a avaliação do melhor caminho de acordo com certo tipo de critérios e parâmetros.

Por último é de realçar a análise feita aos diversos tipos de algoritmos implementados que permitiu obter uma análise comparativa entre cada estratégia no que diz respeito ao seu tempo de execução e da memória utilizada por cada algoritmo. Conclui-se este projeto demonstrando qual seria a melhor estratégia a aplicar no problema em questão.

ÍNDICE

ÍNDICE DE FIGURAS

INTRODUÇÃO

O trabalho prático individual tem como objetivo estimular o uso de técnicas de formulação de problemas, a aplicação de diversas estratégias para a resolução de problema com o uso de algoritmos de procura e o desenvolvimento de mecanismos de raciocínio adequados a esta problemática.

No âmbito de sistemas de representação de conhecimento e raciocínio foi proposto a implementação de diversos algoritmos e pesquisa e procura que permitam obter caminhos válidos ao longo de um conjunto de pontos representativos da cidade de Lisboa. Desta forma, era caso de estudo a obtenção de caminhos que fossem percorridos por um camião que ia recolhendo o lixo em cada ponto. Além disso, foi também proposto a implementação de certas regras que fizessem cumprir com a capacidade do camião e que permitissem a obtenção de caminhos avaliados segundo critérios e indicadores de produtividade, entre outros requisitos. Finalmente, era objeto de estudo a análise dos resultados obtidos em cada algoritmo implementado assim como uma análise comparativa entre essas estratégias.

Todo o trabalho foi realizado com recurso à linguagem PROLOG para implementação de todos algoritmos, regras e funcionalidades e com recurso a JAVA a implementação de um programa de *parsing* que permitiu a obtenção de um *dataset* menos redundante e com toda informação essencial.

É de notar que o projeto foi feito de forma que os algoritmos de pesquisa e procura implementados cumprissem com o limite de capacidade do camião, fazendo com que a quantidade de lixo recolhida fosse um indicador de produtividade**. Implementou-se desta forma a versão completa da solução.**

No presente relatório apresenta-se explicações detalhadas sobre todos os algoritmos e funcionalidades que sustentaram este projeto, acompanhadas por código e imagens de forma a ilustrar o raciocínio usado e manter uma documentação exemplificativa do projeto.

FORMULAÇÃO DO PROBLEMA

Na ciência da computação existem muitos problemas que podem ser formulados como problemas de procura onde se primoriza a obtenção de caminhos com menor custo. Um dos problemas mais conhecidos é o problema do caixeiro viagem que tem como objetivo tentar determinar qual a menor rota a percorrer voltanto novamente ao íncio. Além deste problema, toda a àrea da computação e da matemática possui diversos problemas que são facilmente formulados e resolvidos utilizando algoritmos de procura como por exemplo, resolução de puzzles e de outro problemas em contexto competitivo (jogos), deteção de bugs em software e hardware, planeamento de acções feitas por um robot, entre muitos outros.

Desta forma, também o projeto em mãos pode ser formulado como um problema clássico de pesquisa e procura de soluções. Para essa formulação é necessário definir quais as componentes principais aos problemas de procura. Assim, para este problema é necessário contextualizar:

- Estado Inicial;
- Estado Final;
- Representação de Estados;
- Operadores – ações, pré-condições, nomes, entre outros;
- Custo da solução.

Começando pela representação dos estados do nosso problema podemos admitir que cada ponto de recolha que o camião do lixo deve passar pode ser visto como estado. Assim, o camião vai avançando de ponto em ponto da mesma forma que o algoritmo avança de estado em estado, sendo que estados iguais equivalem aos mesmo ponto de recolha. Desta maneira, o conjunto de pontos de recolha disponíveis pode ser visto como grafo do problema.

O estado inicial no contexto do nosso problema pode ser visto como a garagem do camião. Este é definida á custa de um nodo pertencente ao conjunto de pontos de recolha por onde passa o camião. Desta forma, mal o camião sai da garagem existe próximo dela lixo a recolher, uma vez que a garagem também faz parte do grafo representativo da cidade de lisboa.

Consideramos o depósito de lixo do camião como estado final, isto é, onde o camião termina a sua viagem de recolha de lixo. Este depósito é definido no grafo do problema como um ponto único que assumimos estar acessível a todos os outros pontos de recolha. Contudo, é de notar que apesar do depósito ser o destino qualquer que seja o caminho que o camião realize este pode ter de aceder ao depósito ao longo do percurso como forma de descarregar o lixo recolhido cumprido, assim, com a capacidade do camião. Nestes moldes, é necessário definir também o estado final como sendo o ponto exatamente antes do camião terminar o percurso no depósito. Por exemplo, o percurso garagem->A->C->depósito->D->E->depósito além de consideramos o depósito como estado final, na prática também podemos considerar o E como estado final, visto que o caminho realmente considerado é o garagem->A->C->D->E onde as idas a depósito são feitas de forma a descarregar lixo e não de forma cumprir com a adjacência de nodos.

Ao longo dos percursos realizados pelo camião teve-se em consideração algumas premissas, como por exemplo, o depósito é adjacente a qualquer vértice do grafo. Além disso, quando o camião atinge a capacidade máxima é necessário que este vá ao depósito descarregar

o lixo. Assumimos também que é provável que numa recolha não seja possível aceder a todos os pontos do grafo. Assim, o percurso do camião está limitado a um caminho que pode não conter todos os pontos de recolha, sendo que isso faz com que o percurso também seja válido.

No que toca ao custo da solução podemos admitir várias métricas dependendo do contexto a avaliar o percurso. Podemos admitir que o melhor percurso é aquele com mais lixo recolhido, mas também podemos assumir que é o percurso com menos distancia percorrida. Desta forma tentou-se, para cada solução encontrada, devolver vários parâmetros que possam ser posteriormente avaliados de forma independente.

DESCRIÇÃO DO TRABALHO

De modo a melhor organizar e estruturar o trabalho, decidiu-se dividir o projeto em diferentes ficheiros PROLOG que agregam componentes semelhantes do sistema:

- **Main.pl:** onde está presente definições iniciais para o projeto e a função de *load* que faz o carregamento do *dataset* criado para a base de conhecimento do projeto.
- **Algoritmos.pl:** este documento agrega todas os algoritmos de pesquisa e procura que foram implementados.
- **Funcionalidades.pl:** neste ficheiro estão implementadas todas as funcionalidades, isto é, pesquisas por caminhos seletivos de tipo de lixo, regras de obtenção do melhor caminho para certos algoritmos, entre outros.
- **RegrasAuxiliares.pl**: contém todas as regras que serviram de auxílio à implementação de algoritmos e funcionalidades.

DATASET

Como forma de analisar e extrair a informação conveniente do *dataset* sobre os pontos de recolha fornecido pelos docentes, foi criado um programa em java que faça a filtragem da informação essencial à implementação de um grafo representativo do problema em causa. Assim, a informação das linhas na figura 1 foi transformada, após o *parsing*, nas linhas da figura 2.

```
-9.142408356;38.70699663;386;Misericórdia;15811: R Bernardino da Costa (Ambos (7->29)(->22): Lg Corpo Santo - Tv Corpo Santo);Lixos;CV0120;140;3;420
-9.142408356;38.70699663;387;Misericórdia;15811: R Bernardino da Costa (Ambos (7->29)(->22): Lg Corpo Santo - Tv Corpo Santo);Lixos;CV0090;240;4;960
```

Figura 1: Porção do *dataset* original

```
-9.142408356;38.70699663;386;R Bernardino da Costa;333/387;Lixos;140;3;420
-9.142408356;38.70699663;387;R Bernardino da Costa;333/388;Lixos;240;4;960
```

Figura 2: Porção do *dataset* transformada após o *parsing*

Como podemos ver admitiu-se que os pontos adjacentes a um determinado ponto de recolha são o ponto imediatamente abaixo (neste caso o 387 para o ponto 386 e o 388 para o ponto 387) e o ponto que se encontra no início da rua que está à frente da informação sobre a rua atual (tanto para o 386 como o 387 será a rua Lg Corpo Santo). Caso esta rua da frente não exista no *dataset* admite-se que a rua só é adjacente à de baixo. Na última linha do *dataset* caso também não haja informação sobre essa rua à frente então admitimos que este não tem pontos adjacentes. Todos os pontos foram identificados no *dataset* com o identificador do caixote do lixo, no caso das figuras acima será o identificador 386 e 387, por exemplo. Finalmente, tal como já foi dito também se acrescentou ao *dataset* um ponto representativo do depósito tal como está na figura abaixo. O ponto adjacente 0 representa que este depósito é adjacente a todos os outros pontos de recolha.

```
-9.143308809;38.70807879;1;Deposito;0;Lixeira;0;0;0
```

Figura 3: Representação do depósito no *dataset*

De forma que o Prolog conseguisse ler o *dataset* e aplicá-lo á base de conhecimento foi necessário criar um predicado, ***load***, representado na imagem 4 que utiliza a biblioteca csv do Prolog para transformar um ficheiro csv no termo ***ponto*** que será inserida no programa através de um *assert*, criando assim a base de conhecimento. A representação de todo este processamento assim como a implementação dos predicados envolvidos está representado na figura 4.

```
%---------------------------------- - - - - - - - - - - - - - - - - -
% Descarrega o ficheiro csv como base de conhecimento -> {V,F}

load(Data) :- csv_read_file('Dataset.csv', Data, [functor(recolha), arity(9), separator(0';)]), %'
              defineTerms(Data).

%---------------------------------- - - - - - - - - - - - - - - - - -
% Cria os vários predicados inserindos na base de conhecimento -> {V,F}

defineTerms([recolha(A,B,C,D,E,F,G,H,I)]) :- assert(ponto(A,B,C,D,[],F,G,H,I)), !.
defineTerms([recolha(A,B,C,D,E,F,G,H,I)|X]) :- atom(E), split_string(E, "/"," ", L),
                                               fromStringToInt(L,Q,R),
                                               assert(ponto(A,B,C,D,[Q,R],F,G,H,I)), defineTerms(X).
defineTerms([recolha(A,B,C,D,E,F,G,H,I)|X]) :- integer(E),
                                               assert(ponto(A,B,C,D,[E],F,G,H,I)),
                                               defineTerms(X).

%---------------------------------- - - - - - - - - - - - - - - - - -
% Transforma uma lista de strings para inteiros -> {V,F}

fromStringToInt([Y],A,B) :- atom_number(Y,B).
fromStringToInt([X|Y],A,B) :- atom_number(X,A), fromStringToInt(Y,A,B).
```

Figura 4: Predicados utilizados no *load* do *dataset*

O termo ***ponto*** é constituído por dois valores *double* representativos das coordenadas desses pontos, um *integer* que representa o identificador desse ponto de recolha, o nome da rua onde este ponto se encontra, uma lista de pontos que lhe são adjacentes, o tipo de lixo presente nesse local, quantidade máxima que esse contentor de lixo recebe, o número de contentores de lixo e finalmente a quantidade total de lixo nesse ponto.

Para que o *load* funcione é necessário que o Prolog esteja na mesma diretoria que o ficheiro csv. Para isso utilizou-se a expressão abaixo para ter ambas entidades na mesma diretoria e conseguir fazer o *load* tal como está na imagem[1]. Sempre que se corra o programa é necessário fazer o *load* do *dataset*, pois caso contrário o termo *ponto* não vai ficar inserido na base de conhecimento.

```
?- working_directory(_,'c:/users/ana luísa carneiro/desktop/uminho/2 semestre 3º ano/srcr/tp-individual').
true.

?-
%  library(csv) compiled into csv 0.02 sec, 177 clauses
% c:/Users/Ana Luísa Carneiro/Desktop/Uminho/2 semestre 3º ano/SRCR/TP-Individual/main.pl compiled 0.02 sec, 137 clauses
?-
|
|
|
|    load(Data).
Data = [recolha(-9.143308809, 38.70807879, 1, 'Deposito', 0, 'Lixeira', 0, 0, 0), recolha(-9.143308809, 38.70807879, 355, 'R
8.70807879, 356, 'R do Alecrim', 357, 'Lixos', 240, 7, 1680), recolha(-9.143308809, 38.70807879, 357, 'R do Alecrim', 358, 'I
do Alecrim', 359, 'Papel e Cartão', 240, 6, 1440), recolha(-9.143308809, 38.70807879, 359, 'R do Alecrim', 368, 'Papel e Cart
 Alecrim', 369, 'Lixos', 240, 1, 240), recolha(-9.143377778, 38.70807819, 369, 'R do Alecrim', 370, 'Lixos', 140, 6, 840), re
```

Figura 5: *Load* do *dataset* para a base de conhecimento

[1] A diretoria a utilizar depende de computador para computador. Basta só que o prolog esteja a trabalhar na mesma diretoria onde está armazenado o ficheiro csv.

IMPLEMENTAÇÃO DA CAPACIDADE DO CAMIÃO

Para a implementação do limite de capacidade do camião houve a necessidade de criar um predicado auxiliar ***adicionaDeposito*** que tem como objetivo determinar as idas ao depósito ao longo do percurso, recebendo como parâmetros um caminho entre a origem e o destino, o novo caminho a realizar com as idas ao depósito e um acumulador que vai contabilizar o lixo recolhida a cada ponto percorrido.

Para isso criou-se um predicado também ele auxiliar, ***adicionaDepositoAux***, que vai recursivamente iterando o caminho recebido como parâmetro acrescentando ao longo das iterações os vários depósitos.

Como em Prolog a recursividade é realizada do fim para o início, então o caso de paragem será quando o percurso for vazio, isto é, quando já todo o caminho tiver sido percorrido e quando o acumulador que conta a quantidade de lixo for zero.

Para o caso recursivo do predicado, este começa por verificar qual a quantidade de lixo a recolher no primeiro ponto do caminho. Caso a soma entre essa quantidade e o acumulador de lixo no camião for inferior a 15000 então o caminho a ser gerado vai conter esse nodo e a quantidade de lixo no camião vai aumentar. Caso contrário então não só vai ser necessário acrescentar ao novo caminho o nodo visitado, mas também a ida ao depósito. No fim disso, o acumulador volta a estar a 0.

```
adicionaDeposito(Caminho,Novo,Acc) :-
    deposito(N),
    adicionaDepositoAux(Caminho,[N],Novo,Acc).

adicionaDepositoAux([],Novo,Novo,0).
adicionaDepositoAux([Nodo|Caminho],Aux,Novo,AccTotal) :-
    ponto(_,_,Nodo,_,_,_,_,_,T),
    adicionaDepositoAux(Caminho,Lst,Novo,AccFinal),
    ( AccFinal + T < 15000 -> Lst = [Nodo|Aux], AccTotal is AccFinal + T;
       deposito(N),
     Lst = [N,Nodo|Aux], AccTotal is 0).
```

Figura 6: Predicado *adicionaDeposito*

Além deste predicado também foi implementado um outro predicado, ***adicionaTipoDeposito***, semelhante à apresentada em cima, contudo esta foi utilizada para a obtenção dos caminhos seletivos no tipo de lixo que recolhe. Desta forma caso um ponto que esteja no caminho não contenha o tipo de lixo selecionado então, esse lixo não vai ser somado ao acumulador. Por outras palavras esse lixo não vai fazer parte da capacidade do camião. Para isso foi implementado duas condições: uma que determina se o tipo de lixo nesse ponto é igual ao tipo de lixo que queremos recolher e outro, que caso o tipo de lixo for igual, vai verificar se a capacidade do camião vai ser ultrapassada ou não.

```
adicionaTipoDeposito(Tipo,Caminho,Novo,Acc) :-
    deposito(N),
    adicionaTipoDepositoAux(Tipo,Caminho,[N],Novo,Acc).

adicionaTipoDepositoAux(Tipo,[],Novo,Novo,0).
adicionaTipoDepositoAux(Tipo, [Nodo|Caminho],Aux,Novo,AccTotal) :-
    ponto(_,_,Nodo,_,_,Select,_,_,T),
    adicionaDepositoAux(Caminho,Lst,Novo,AccFinal),
    (Select==Tipo -> ( AccFinal + T < 15000 -> Lst = [Nodo|Aux],
                                               AccTotal is AccFinal + T;
                                               deposito(N),
                                               Lst = [N,Nodo|Aux], AccTotal is 0);
                                           Lst = [Nodo|Aux] ).
```

Figura 7: Predicado *adicionaTipoDeposito*

Na figura 8 está apresentado um exemplo de um caminho que começa na garagem e que vai ao depósito para descarregar lixo ao longo do percurso, utilizando o algoritmo *Depth Fisrt*.

```
?- resolveDF(386,335,P,Lixo,Distancia,NrPontos).
P = [386,387,388,389,390,463,464,465,466,467,468,469,470,471,472,473,474,482,1,483,484,485,486,487,488,489,490,491,
494,495,496,492,493,497,498,499,500,501,502,503,531,532,533,641,642,643,644,645,646,647,648,649,650,651,652,653,654
,655,656,657,658,659,660,661,662,1,663,664,333,334,335,1],
Lixo = 31210,
Distancia = 0.042750608977637806,
NrPontos = 72 ,
```

Figura 8: Percurso com idas ao depósito

ALGORITMOS

A realização deste projeto passou por implementar diversas estratégias de procura que permitiram resolver o problema em causa de forma consistente e eficiente. Deste modo, foram implementados 2 tipos de estratégias de procura: procura não informada e procura informada.

A procura não informada consiste em estratégias que usam apenas as informações disponíveis na definição do problema. Para esta estratégia foram implementados 3 tipos de procura: a procura em profundidade, a procura em largura e a procura iterativa limitada em profundidade

A procura informada utiliza informação do problema para evitar que o algoritmo de pesquisa fique perdido na procura. Consiste em estratégias que usam informações externas sobre o modo como se deve adequar a diferentes estados. Para esta estratégia foram implementados 2 tipos de procura: a procura gulosa e a procura A*.

De seguida encontram-se descritos os diversos algoritmos implementados ao longo da realização do trabalho. Todas as referências a predicados que não se encontram representados em imagem foram implementados no ficheiro RegrasAuxiliares.pl. Desta forma é conveniente a leitura desse ficheiro caso surja alguma dúvida na implementação desses predicados.

DEPTH FIRST – PROCURA EM PROFUNDIDADE

A busca em profundidade ou *Depth First* é um algoritmo de travessia de árvores ou grafos que intuitivamente começa num nodo raiz e explora tanto quanto possível cada um dos seus ramos priorizando, assim, a exploração de nodos em profundidade.

Para o caso do problema em mão, este algoritmo começa a sua iteração na garagem progredindo a partir da expansão do primeiro caixote de lixo adjacente ao nodo anterior até que seja encontrado o alvo da busca, isto é, chegando ao ponto exatamente antes do depósito.

Este algoritmo exige muita pouca memória e é usado sempre que existem muitas soluções para o problema em causa, o que acontece no caso em estudo. Contudo, este algoritmo não é usado para problemas que tenham profundidade infinita, pois pode ficar presa em ramos errados.

O modo de implementação deste algoritmo para o caso em estudo passou por criar duas predicados principais: ***resolveDF*** e a ***caminhoDF***. O primeiro predicado consiste em inicializar as estruturas necessárias à procura, enquanto a segunda consiste em pesquisar recursivamente os nodos do grafo implementado utilizando a estratégia *depth First*.

Inicialização

Para proceder à inicialização da procura *depth first* implementou-se a predicado ***resolveDF*** que tem como parâmetros o ponto inicial do percurso, isto é a garagem de onde parte o camião, o destino, o percurso realizado entre a fonte e o ponto final, a totalidade de lixo recolhido, distância total realizada e o número de pontos de recolha pesquisados.

Esta predicado começa por chamar a **caminhoDF** que vai determinar um caminho entre a origem e o destino. De seguida vai determinar o lixo que foi recolhido mal o camião sai da garagem, isto é, o lixo recolhido na origem do percurso. Além disso, de forma a cumprir com a capacidade do camião vamos determinar quantas vezes e em que alturas é que o camião terá de ir ao depósito descarregar o lixo e posteriormente voltar ao ponto onde estava através do predicado ***adicionaDeposito***. Finalmente calculamos a distância total do percurso através do predicado ***distanciaTotal*** e o número de pontos por onde passamos.

```
resolveDF(PInicio,PFim,Percurso,Lixo,DTotal,Custo) :-
                    caminhoDF(PInicio,PFim,[PInicio],SFinal,Lixo1,DFinal),
                    ponto(_,_,PInicio,_,_,_,_,_,L),
                    Lixo is L + Lixo1,
                    inverso([PInicio|SFinal],Caminho),
                    adicionaDeposito(Caminho,Percurso,Acc),
                    distanciaTotal(Percurso,DTotal),
                    length(Percurso,Custo).
```

Figura 9: Predicado *resolveDF*

Pesquisa

O predicado ***caminhoDF*** tem como objetivo encontrar um caminho em profundidade que começa na garagem (origem) e acabe num ponto destino (ponto exatamente antes da última ida ao depósito). Para isso a predicado recebe como parâmetros o ponto inicial, final, histórico dos pontos pesquisados, a solução encontrada, o lixo total recolhido e a distância percorrida pelo percurso encontrado em cada iteração do predicado.

Como em Prolog a recursividade é realizada do fim para o início, então o caso de paragem será um percurso vazio em que o nodo atual é igual ao nodo final. Além disso, os valores da quantidade de lixo recolhida e da distância percorrida serão zero.

Para o caso recursivo da procura, este começa por verificar se o primeiro ponto da solução pesquisada é adjacente ao nodo atual da procura através dos predicados ***adjacente*** que determina os nodos adjacente a um determino vértice e do predicado ***membroLista*** que determina se um ponto está na lista de adjacência recebida como parâmetro. Caso este seja, então vamos verificar se o ponto não está no conjunto dos pontos visitados, evitando deste modo percursos cíclicos. De seguida determinamos o lixo recolhido no ponto adjacente, e a distância que vai desde o ponto atual até ao adjacente, através do predicado ***distanciaPonto***. Finalmente, é chamado o predicado recursivamente acrescentando ao histórico o ponto adjacente encontrado, calculando no final o lixo total recolhido e a distância total percorrida no percurso encontrado.

```
% Caso de paragem:
caminhoDF(Estado,Fim,_,[],0,0) :- Estado == Fim, !.
% Caso Recursivo:
caminhoDF(Estado,Fim,Historico,[P|Solucao],LixoTotal,CustoTotal) :-
    adjacente(Estado,Lista),
    membroLista(P,Lista),
    nao(membro(P,Historico)),
    ponto(_,_,P,_,_,_,_,_,T),
    distanciaPonto(P,Lista,D),
    caminhoDF(P,Fim,[P|Historico],Solucao,LixoFinal,CustoFinal),
    LixoTotal is LixoFinal + T,
    CustoTotal is CustoFinal + D.
```

Figura 10: Predicado *caminhoDF*

Outros Predicados

Como forma de tornar o algoritmo implementado mais versátil e completo implementou-se um conjunto de predicados que tem como objetivo pesquisar todos os percursos calculados pelos predicados explicitadas anteriormente.

```
%Percurso desde da garagem até ao deposito
percursoDF(S,Fim,L,D,C) :- garagem(Inicio),
                           resolveDF(Inicio,Fim,S,L,D,C).

% Todos os Percursos entre a garagem e o deposito
allPercursoDF(Fim,L) :- findall((S,Lixo,D,C),percursoDF(S,Fim,Lixo,D,C),L).

% Todos os Percursos entre uma fonte e um destino
allResolveDF(PInicio,PFim,L) :- findall((S,Lixo,D,C),
                                resolveDF(PInicio,PFim,S,Lixo,D,C),L).
```

Figura 11: Outros predicados implementados no *Depth First*

Além disso, também se implementou os predicados ***resolveTipoDF*** e ***caminhoTipoDF*** que funcionam de forma análoga aos predicados demonstrados em cima, contudo ambos recebem como parâmetro o tipo de lixo a recolher. Assim, além de se calcular um percurso através de uma pesquisa em profundidade vamos, exclusivamente, recolher lixo dos pontos pesquisados que contenham como tipo de lixo o mesmo que recebido como parâmetro. Para cumprir com este objetivo decidiu-se implementar uma condição onde sempre que o ponto atual da pesquisa tenha como tipo de lixo igual ao recebido como parâmetro então a totalidade de lixo recolhido e o número de pontos com aquele tipo de lixo aumenta. Caso contrário, a totalidade de lixo e o número de pontos mantém-se igual.

```
resolveTipoDF(Tipo,PInicio,PFim,Percurso,Lixo,D,C) :-
    caminhoTipoDF(Tipo,PInicio,PFim,[PInicio],SFinal,Lixo1,D,C1),
    ponto(_,_,PInicio,_,_,Select,_,_,L),
    ( Select==Tipo -> Lixo is L + Lixo1, C is C1+1 ; Lixo is Lixo1, C is C1),
    inverso([PInicio|SFinal],Caminho),
    adicionaTipoDeposito(Tipo,Caminho,Percurso,Acc),
    distanciaTotal(Percurso,DTotal).

caminhoTipoDF(_,Estado,Fim,_,[],0,0,0) :- Estado == Fim, !.
caminhoTipoDF(Select,Estado,Fim,Historico,[P|Solucao],LixoTotal,CustoTotal, PontosTotal) :-
    adjacente(Estado,Lista),
    membroLista(P,Lista),
    nao(membro(P,Historico)),
    ponto(_,_,P,_,_,Tipo,_,_,T),
    distanciaPonto(P,Lista,D),
    caminhoTipoDF(Select,P,Fim,[P|Historico],Solucao, LixoFinal, CustoFinal, PontosFinal),
    (Select==Tipo -> LixoTotal is LixoFinal + T,
                     PontosTotal is PontosFinal+1;
                    LixoTotal is LixoFinal, PontosTotal is PontosFinal ),
    CustoTotal is CustoFinal + D.
```

Figura 12: Predicados para obtenção da recolha de lixo seletivo em *Depth First*

Complexidade

Este algoritmo não é completo, pois falha em espaços de profundidade infinita, com repetições. Contudo, podemos modificar o algoritmo evitando estados repetidos ao longo do caminho. Além disso este algoritmo também não é otimal pois devolve a 1º solução encontrada e não a melhor.

No que toca á complexidade temporal, este tem um valor de b^mcom *b* a representar o número máximo de sucessores de um nó e *m* a máxima profundidade do espaço de estados. Ou seja, é proporcional à soma entre o número de vértices e o número de arestas do grafo em questão. Desta forma, caso $m > d$, com *d* a representar a profundidade da melhor solução então a pesquisa *depth first* deixa de ser viável.

No que toca à complexidade espacial, este tem um valor linear de $b * m$ com b a representar o número máximo de sucessores de um nó e *m* a máxima profundidade do espaço de estados.

BREADTH FIRST – PROCURA EM LARGURA

A busca em largura ou *Breadth First* é o oposto do algoritmo *depth first*, pois em vez de pesquisar em profundidade tal como o próprio nome diz faz uma pesquisa em largura. Desta forma, é um algoritmo de travessia de árvores ou grafos que intuitivamente começa num nodo raiz e explora todos os nós de menor profundidade em cada um dos seus ramos priorizando, assim, a exploração de nodos em largura.

Para o caso do problema em mão, este algoritmo começa a sua iteração na garagem progredindo a partir da expansão dos caixotes de lixo adjacentes ao nodo anterior até que seja encontrado o alvo da busca, isto é, chegando ao ponto exatamente antes do depósito.

Este algoritmo tem uma pesquisa muito sistemática e é geralmente usado para pequenos problemas que podem ser resolvidos facilmente com este tipo de algoritmo. Contudo, este algoritmo demora muito tempo e sobretudo ocupa muito espaço.

O modo de implementação deste algoritmo para o caso em estudo passou por criar dois predicados principais: ***resolveBF*** e a ***caminhoBF***. O primeiro predicado consiste em inicializar as estruturas necessárias à procura, enquanto a segunda consiste em pesquisar recursivamente os nodos do grafo implementado utilizando a procura *breadth first*.

Inicialização

Para proceder à inicialização da procura *breadth first* implementou-se o predicado ***resolveDF*** que tem como parâmetros o ponto inicial do percurso, isto é a garagem de onde parte o camião, o destino, o percurso realizado entre a fonte e o ponto final, a totalidade de lixo recolhido, distância total realizada e o número de pontos de recolha pesquisados.

Esta predicado começa por chamar a ***caminhoBF*** que vai determinar um caminho entre a origem e o destino. De seguida vai determinar o lixo que foi recolhido mal o camião sai da garagem, isto é, o lixo recolhido na origem do percurso. Além disso, de forma a cumprir com a capacidade do camião vamos determinar quantas vezes e em que alturas é que o camião terá de ir ao depósito descarregar o lixo e posteriormente voltar ao ponto onde estava através do predicado ***adicionaDeposito***. Finalmente calculamos a distância total do percurso através do predicado ***distanciaTotal*** e o número de pontos por onde passamos.

```
%--- INICIALIZAÇÃO ----%
resolveBF(Inicial, Fim, Percurso,Lixo,DTotal,C) :- caminhoBF(Fim, [[Inicial]], Solucao,Lixo1,D),
                                                   ponto(_,_,Inicial,_,_,_,_,_,LixoAtual),
                                                   Lixo is Lixo1 + LixoAtual,
                                                   inverso(Solucao,Caminho),
                                                   adicionaDeposito(Caminho,Percurso,Acc),
                                                   distanciaTotal(Percurso,DTotal),
                                                   length(Percurso,C).
```

Figura 13: Predicado *resolveBF*

Pesquisa

O predicado ***caminhoBF*** tem como objetivo encontrar um caminho em largura que começa na garagem (origem) e acabe num ponto destino (ponto exatamente antes da última ida ao depósito). Para isso o predicado recebe como parâmetros o ponto final, histórico dos pontos pesquisados, a solução encontrada, o lixo total recolhido e a distância percorrida pelo percurso encontrado em cada iteração do predicado.

Como em Prolog a recursividade é realizada do fim para o início, então o caso de paragem será quando o último ponto encontrado no histórico é igual ao ponto destino. Além disso, os valores da quantidade de lixo recolhida e da distância percorrida serão zero. Para este caso, vamos ter de determinar o inverso do caminho encontrado no histórico de pontos visitadas pois, tal como já foi dito o Prolog, cria caminhos utilizando a recursividade do final para o início fazendo com que o caminho encontrado tenha de ser invertido.

Para o caso recursivo da procura, este começa por colocar no histórico o ponto inicial determinado, em seguida, a lista dos nodos adjacentes a esse ponto através do predicado ***adjacente***. De seguida, verificamos quais desses pontos adjacentes ainda não foram visitados através da função ***visitadosmembro***. Determinamos para o primeiro ponto da lista obtida qual a distância que este tem ao ponto adjacente através do predicado ***distanciaPonto*** e qual a quantidade de lixo no local. A partir do predicado ***maplist*** vamos aplicar a predicado ***adicionarFim*** (que adiciona um elemento ao fim da lista) à lista de possíveis pontos adjacentes formando desta forma um caminho inverso que começa no ponto inicial e termina num ponto seu adjacente. O resultado é depois unido com o restante histórico através do predicado ***append***. Finalmente, é feita a chamada recursiva com o resultado do *append* calculado em cima determinando no final o lixo total recolhido e a distância total percorrida pelo percurso encontrado.

```
%Caso de Paragem: Quando o caminho chega ao fim
caminhoBF(Fim, [[Fim| Visitados] | _ ], Caminho,0,0) :-
    inverso([Fim| Visitados],Caminho).
%Caso de Recursivo:
caminhoBF(Fim, [Visitados|Resto], Solucao,LixoTotal,DistanciaTotal) :-
    Visitados = [Inicio|_],
    Inicio \== Fim,
    adjacente(Inicio, Lista),
    visitadosMembro(Lista, Visitados,[],[T|Extensao]),
    ponto(_,_,T,_,_,_,_,_,LixoAtual),
    distanciaPonto(T,Lista,DistanciaAtual),
    maplist(adicionarFim(Visitados), [T|Extensao], ExtensaoVst),
    append(Resto,ExtensaoVst, AtualizaFila),
    caminhoBF(Fim, AtualizaFila,Solucao,LixoFinal,DistanciaFinal),
    LixoTotal is LixoFinal + LixoAtual,
    DistanciaTotal is DistanciaFinal + DistanciaAtual.
```

Figura 14: Predicado *caminhoBF*

Outros Predicados

Como forma de tornar o algoritmo implementado mais versátil e completo implementou-se um conjunto de predicados que tem como objetivo pesquisar todos os percursos calculados pelos predicados explicitadas anteriormente.

```
%Percurso entre uma garagem e um deposito
percursoBF(S,Fim,Lixo,D,C) :- garagem(Inicio),
                              resolveBF(Inicio,Fim,S,Lixo,D,C).

% Todos os Percursos entre a garagem e o deposito
allPercursoBF(Fim,L) :- findall((S,Lixo,D,C),percursoBF(S,Fim,Lixo,D,C),L).

% Todos os Percursos entre a garagem e o deposito
allResolveBF(Inicial,Final,L) :- findall((S,Lixo,D,C),resolveBF(Inicial,Final,S,Lixo,D,C),L).
```

Figura 15: Outros predicados implementados no *Breadth First*

Além disso, também se implementou os predicados ***resolveTipoBF*** e ***caminhoTipoBF*** que funcionam de forma análoga às predicados apresentadas em cima, contudo ambas recebem como parâmetro o tipo de lixo a recolher. Assim, além de se calcular um percurso através de uma pesquisa em largura vamos, exclusivamente, recolher lixo dos pontos pesquisados que contenham como tipo de lixo o mesmo que recebido como parâmetro. Para cumprir com este objetivo decidiu-se implementar uma condição onde sempre que o ponto atual da pesquisa tenha como tipo de lixo igual ao recebido como parâmetro então a totalidade de lixo recolhido e o número de pontos com aquele tipo de lixo aumenta. Caso contrário, a totalidade de lixo e o número de pontos mantém-se igual.

```
resolveTipoBF(Tipo,Inicial, Fim, Percurso,Lixo,D,C) :-
        caminhoTipoBF(Tipo,Fim, [[Inicial]], Solucao,Lixo1,D,C1),
        ponto(_,_,Inicial,_,_,Select,_,_,LixoAtual),
        (Select==Tipo -> LixoTotal is LixoFinal + T, C is C1+1; LixoTotal is LixoFinal, C is C1 ),
        inverso(Solucao,Caminho),
        adicionaTipoDeposito(Tipo,Caminho,Percurso,Acc),
        distanciaTotal(Percurso,DTotal).

caminhoTipoBF(Tipo,Fim, [[Fim| Visitados] | _ ], Caminho,0,0,0) :-
    inverso([Fim| Visitados],Caminho).
caminhoTipoBF(Tipo,Fim, [Visitados|Resto], Solucao,LixoTotal,DistanciaTotal,PontosTotal) :-
    Visitados = [Inicio|_],
    Inicio \== Fim,
    adjacente(Inicio, Lista),
    visitadosMembro(Lista, Visitados,[],[T|Extensao]),
    ponto(_,_,T,_,_,Select,_,_,LixoAtual),
    distanciaPonto(T,Lista,DistanciaAtual),
    maplist(adicionarFim(Visitados), [T|Extensao], ExtensaoVst),
    append(Resto,ExtensaoVst, AtualizaFila),
    caminhoTipoBF(Tipo,Fim, AtualizaFila,Solucao,LixoFinal,DistanciaFinal, PontosFinal),
    (Select==Tipo -> LixoTotal is LixoFinal + T,
                     PontosTotal is PontosFinal +1;
                    LixoTotal is LixoFinal, PontosTotal is PontosFinal ),
    DistanciaTotal is DistanciaFinal + DistanciaAtual.
```

Figura 16: Predicados para obtenção de caminho *Breadth First* de lixo seletivo

Complexidade

Este algoritmo é completo, contudo é possível que não o seja quando o fator de ramificação for infinito. É também otimal se o custo de cada passo for 1.

No que toca à complexidade temporal, este tem um valor de b^dcom *b* a representar o número máximo de sucessores de um nó e *d* a profundidade da melhor solução, ou seja, é exponencial em d.

No que toca à complexidade espacial, este tem um valor exponencial de b^dcom *b* a representar o número máximo de sucessores de um nó e *d* a profundidade da melhor solução. Isto acontece devido ao facto de que cada nodo do grafo é armazenado em memória.

Em suma, a estratégia *depth first* para problemas com várias soluções é bem mais rápida e tem uma complexidade espacial menor que a de um algoritmo de *breadth first*.

DEPTH FIRST LIMITADA – BUSCA ITERATIVA LIMITADA PROFUNDIDADE

A busca como limite de profundidade ou *Depth First Limitada* é um algoritmo de travessia de árvores ou grafos que intuitivamente começa num nodo raiz e explora cada um dos seus ramos até ao limite proposto priorizando, assim, a exploração de nodos em profundidade.

Para o caso do problema em questão, este algoritmo começa a sua iteração na garagem progredindo a partir da expansão do primeiro caixote de lixo adjacente ao nodo anterior até que seja encontrado o alvo da busca, isto é, chegando ao ponto exatamente antes do depósito ou até que se tenha atingido o limite proposto para a pesquisa.

Este algoritmo permite resolver alguns problemas encontrados com a procura *depth first* como por exemplo a incapacidade de lidar com profundidade infinita. Para isso, este algoritmo implementa um limite máximo para a pesquisa de soluções, ou seja, implementa um nível máximo de procura. É um algoritmo, que tal como o *depth first*, exige muita pouca memória e é usado sempre que existem muitas soluções para o problema em causa, o que acontece no caso em estudo.

O modo de implementação deste algoritmo para o caso em estudo passou por criar dois predicados principais: ***resolveDFLimitado*** e a ***caminhoDFLimitado***. O primeiro predicado consiste em inicializar as estruturas necessárias à procura, enquanto o segundo consiste em pesquisar recursivamente os nodos do grafo implementado utilizando a procura *depth first limitado*.

Inicialização

Para proceder à inicialização da procura *depth first limitado* implementou-se o predicado ***resolveDFLimitado*** que tem como parâmetros o ponto inicial do percurso, isto é a garagem de onde parte o camião, o destino, o percurso realizado entre a fonte e o ponto final, a totalidade de lixo recolhido, distância total realizada, o número de pontos de recolha pesquisados e o limite da pesquisa a ser utilizado.

Este predicado começa por chamar a ***caminhoDFLimitado*** que vai determinar um caminho limitado pelo valor limite recebido entre a origem e o destino. O cálculo do lixo recolhido, distância percorrida e o número ponto de recolha visitados é análogo ao utilizado no *depth first*. Da mesma forma, também é análogo o cálculo do número de vezes e em que alturas é que o camião terá de ir ao depósito descarregar o lixo.

```
resolveDFLimitado(PInicio,PFim,Percurso,Lixo,DTotal,C,Limite) :-
        caminhoDFLimitado(PInicio,PFim,[PInicio],S1,Lixo1,Dist,Limite),
        adicionarInicio(PInicio,S1,S),
        ponto(_,_,PInicio,_,_,_,_,_,LixoAtual),
        inverso(S,Caminho),
        adicionaDeposito(Caminho,Percurso,Acc),
        distanciaTotal(Percurso,DTotal),
        length(Percurso,C),
        Lixo is Lixo1 + LixoAtual.
```

Figura 17: Predicado *resolveDFLimitado*

Pesquisa

O predicado ***caminhoDFLimitado*** tem como objetivo encontrar um caminho em profundidade que começa na garagem (origem) e acabe num ponto destino (ponto exatamente antes da última ida ao depósito) limitado ao valor do limite recebido. Para isso o predicado recebe como parâmetros o ponto inicial, final, histórico dos pontos pesquisados, a solução encontrada, o lixo total recolhido, a distância percorrida pelo percurso encontrado em cada iteração do predicado e o limite de procura.

O caso de paragem para este algoritmo é idêntico ao utilizado no *depth first*. Para o caso recursivo da procura, este começa por verificar se a procura atingiu o limite proposto. Caso não tenha acontecido então o processo é semelhante ao utilizado no *depth first*. Finalmente, é chamada o predicado recursivamente acrescentando ao histórico o ponto adjacente encontrado e diminuindo o limite num valor visto que o processo já verificou um nível de procura, calculando no final o lixo total recolhido e a distância total percorrida pelo percurso encontrado.

```
% Caso de paragem:
caminhoDFLimitado(Estado,Fim,_,[],0,0,_) :- Estado == Fim, !.
% Caso Recursivo:
caminhoDFLimitado(Estado,Fim,Historico,[P|Solucao],LixoTotal,DistTotal,Limite) :-
    Limite > 1,
    adjacente(Estado,Lista),
    membroLista(P,Lista),
    nao(membro(P,Historico)),
    ponto(_,_,P,_,_,_,_,_,T),
    distanciaPonto(P,Lista,D),
    caminhoDFLimitado(P,Fim,[P|Historico],Solucao, LixoFinal,DistFinal,Limite-1),
    LixoTotal is LixoFinal + T,
    DistTotal is DistFinal + D.
```

Figura 18: Predicado *caminhoDFLimitado*

Outros predicados

Como forma de tornar o algoritmo implementado mais versátil e completo implementou-se um conjunto de predicados que tem como objetivo pesquisar todos os percursos calculados pelas predicados explicitadas anteriormente.

```
% Percurso desde da garagem até ao deposito
percursoDFLimitado(S,Fim,Lixo,Dist,C,Limite) :- garagem(Inicio),
                                                resolveDFLimitado(Inicio,Fim,S,Lixo,Dist,C,Limite).

% Todos os Percursos entre a garagem e o deposito
allPercursoDFLimitado(Fim,Limite,L) :- findall((S,Lixo,Dist,C),percursoDFLimitado(S,Fim,Lixo,Dist,C,Limite),L).

% Todos os Percursos entre uma fonte e um destino
allResolveDFLimitado(PInicio,PFim,Limite,L) :- findall((S,Lixo,Dist,C),
                                                resolveDFLimitado(PInicio,PFim,S,Lixo,Dist,C,Limite),L).
```

Figura 19: Outros predicados implementados para o *Depth First Limitado*

Além disso, também se implementou os predicados ***resolveTipoDFLimitado*** e ***caminhoTipoDFLimitado*** que funcionam de forma análoga às predicados apresentadas em cima, contudo ambas recebem como parâmetro o tipo de lixo a recolher. Assim, além de se calcular um percurso através de uma pesquisa em profundidade limitada vamos, exclusivamente, recolher lixo dos pontos pesquisados que contenham como tipo de lixo o mesmo que recebido como parâmetro. Para cumprir com este objetivo decidiu-se implementar uma condição onde sempre que o ponto atual da pesquisa tenha como tipo de lixo igual ao recebido como parâmetro então a totalidade de lixo recolhido e o número de pontos com aquele tipo de lixo aumenta. Caso contrário, a totalidade de lixo e o número de pontos mantém-se igual.

```
resolveTipoDFLimitado(Tipo,PInicio,PFim,Percurso,Lixo,DTotal,C,Limite) :-
        caminhoTipoDFLimitado(Tipo,PInicio,PFim,[PInicio],S1,Lixo1,Dist,C1, Limite),
        adicionarInicio(PInicio,S1,S),
        ponto(_,_,PInicio,_,_,Select,_,_,LixoAtual),
        inverso(S,Caminho),
        adicionaTipoDeposito(Tipo,Caminho,Percurso,Acc),
        distanciaTotal(Percurso,DTotal),
        (Select==Tipo -> LixoTotal is LixoFinal + LixoAtual,
                         C is C1 +1; LixoTotal is LixoFinal, C is C1 ).

caminhoTipoDFLimitado(Tipo,Estado,Fim,_,[],0,0,0,_) :- Estado == Fim, !.
caminhoTipoDFLimitado(Tipo,Estado,Fim,Historico,[P|Solucao],LixoTotal,DistTotal,PontosTotal, Limite) :-
    Limite > 1,
    adjacente(Estado,Lista),
    membroLista(P,Lista),
    nao(membro(P,Historico)),
    ponto(_,_,P,_,_,Select,_,_,T),
    distanciaPonto(P,Lista,D),
    caminhoTipoDFLimitado(Tipo,P,Fim,[P|Historico],Solucao, LixoFinal,DistFinal,PontosFinal,Limite-1),
    (Select==Tipo -> LixoTotal is LixoFinal + T,
                     PontosTotal is PontosFinal +1;
                     LixoTotal is LixoFinal, PontosTotal is PontosFinal ),
    DistTotal is DistFinal + D.
```

Figura 20: Predicados para a obtenção de caminho *Depth First Limitado* com seleção de lixo

Complexidade

Este algoritmo, ou contrário do *depth first*, é completo, pois não falha em espaços de profundidade infinita com repetições. Contudo, pode haver casos em que o algoritmo pode ser incompleto por exemplo quando a solução possível estiver abaixo do limite. Além disso este algoritmo também não é otimal pois devolve a 1º solução encontrada e não a melhor.

No que toca à complexidade temporal, este tem um valor de b^lcom b a representar o número máximo de sucessores de um nó e l o limite. Desta forma, caso $d > l$, com d a representar a profundidade da melhor solução então a pesquisa *depth first* deixa de ser viável.

No que toca à complexidade espacial, este tem um valor linear de $b * l$ com b a representar o número máximo de sucessores de um nó e l o limite

GREEDY – PROCURA GULOSA

A procura *gulosa* ou *Greedy* é um algoritmo de pesquisa informada para travessia de árvores ou grafos que expande o nodo que aparenta estar mais perto da solução através do uso de heurísticas e assim sequencialmente para os seus nodos adjacentes.

Para o caso do problema em mão, este algoritmo começa a sua iteração na garagem progredindo a partir da expansão do caixote de lixo adjacente ao nodo anterior que possua uma menor heurística até que seja encontrado o alvo da procura, isto é, chegando ao ponto exatamente antes do depósito.

Podemos aplicar as heurísticas de forma a gerar decisões sobre que nodo pesquisar ou quais os nodos a serem descartados. Deste modo, as heurísticas são usadas como método de avaliação de nodos ou como método de corte de estratégias. A heurística utilizada neste algoritmo foi implementada de forma a medir a distância euclidiana em linha reta de um ponto ao depósito, pois é este o vértice que consideramos o real destino do percurso. Para isso, através da latitude e longitude tanto do ponto a pesquisar como da garagem calculamos a distância euclidiana entre esses pontos. No caso de estudo o nodo com a heurística menor será aquele que vai ser o primeiro a ser acrescentado á solução.

```
heuristica(Id,Res) :- ponto(La,Lo,Id,_,_,_,_,_,_),
        deposito(N), ponto(La1,Lo1,N,_,_,_,_,_,_),
        pow(La-La1,2,Pot), pow(Lo-Lo1,2,Pot1),
        Res is sqrt(Pot + Pot1).
```

Figura 21: Predicado da heurística implementada

O modo de implementação deste algoritmo para o caso em estudo passou por criar dois predicados principais: ***resolveGulosa*** e o ***caminhoGulosa***. O primeiro predicado consiste em inicializar as estruturas necessárias à procura, enquanto o segundo consiste em pesquisar recursivamente os nodos do grafo implementado utilizando a procura *gulosa*.

Inicialização

Para proceder à inicialização da procura gulosa implementou-se o predicado ***resolveGulosa*** que tem como parâmetros o ponto inicial do percurso, isto é a garagem de onde parte o camião, o destino e o percurso realizado entre a fonte e o ponto final.

Este predicado começa por chamar a ***caminhoGulosa*** que vai determinar um caminho entre a origem e o destino. Além disso, de forma a cumprir com a capacidade do camião vamos determinar quantas vezes e em que alturas é que o camião terá de ir ao depósito descarregar o lixo e posteriormente voltar ao ponto onde estava através do predicado ***adicionaDeposito***. Finalmente calculamos a distância total do percurso através da ***distanciaTotal*** e o número de pontos por onde passamos.

```
%--- INICIALIZAÇÃO ----%
resolveGulosa(Inicio,Fim,Percurso,CustoTotal) :- heuristica(Inicio,Estima),
                caminhoGulosa(Fim,[[Inicio]/0/Estima],InvCaminho/Custo/_),
                adicionaDeposito(InvCaminho,Percurso,Acc),
                distanciaTotal(Percurso,CustoTotal).
```

Figura 22: Predicado *resolveGulosa*

Pesquisa

A predicado ***caminhoGulosa*** tem como objetivo encontrar um caminho utilizando o algoritmo gulosa que começa na garagem (origem) e acabe num ponto destino (ponto exatamente antes da última ida ao depósito). Para isso a predicado recebe como parâmetros o ponto final, os vários caminhos encontrados pelo algoritmo e a solução encontrada, isto é o melhor caminho da lista de caminhos encontrados que tenha como destino o ponto final recebido como parâmetro.

Como em Prolog a recursividade é realizada do fim para o início, então o caso de paragem será quando a o melhor caminho obtido tem como nodo final o destino recebido. A obtenção do melhor caminho é dada pelo predicado ***obtem_melhor_g*** que avalia as heurísticas dos pontos dos caminhos pertencentes à lista e determina quais os pontos que deverão fazer parte da solução encontrada.

Para o caso recursivo da procura, este começa por verificar qual o melhor caminho calculado até ao momento a partir do ***obtem_melhor_g***. De seguida vamos selecionar da lista de caminhos encontrados a cauda da lista que não contenha o melhor caminho encontrado a partir do predicado ***seleciona***. Além disso, vamos expandir o melhor caminho para que este encontre os vários caminhos adjacentes ao ponto final do melhor caminho utilizando o predicado ***expande_gulosa*** que a partir de um *findall* determina os pontos e consequentes caminhos adjacentes. Finalmente vamos adicionar à cauda selecionada os caminhos que foram expandidos através da função ***append*** chamado, de seguida, recursivamente o predicado ***caminhoGulosa*** com o novo caminho calculado. Assim, vamos determinando com base na avaliação fornecida pela heurística quais os caminhos que são melhores em cada ponto convergindo para o melhor caminho calculado.

```
%Caso de Paragem
caminhoGulosa(Fim,Caminhos,Caminho) :- obtem_melhor_g(Caminhos,Caminho),
                    Caminho = [Fim|_]/_/_.
%Caso de Recursividade
caminhoGulosa(Fim,Caminhos,SolucaoCaminho) :- obtem_melhor_g(Caminhos,MelhorCaminho),
                    seleciona(MelhorCaminho,Caminhos,OutrosCaminhos),
                    expande_gulosa(MelhorCaminho,ExpCaminhos),
                    append(OutrosCaminhos,ExpCaminhos,NovoCaminhos),
                    caminhoGulosa(Fim,NovoCaminhos,SolucaoCaminho).
```

Figura 23: Predicado *caminhoGulosa*

Outros Predicados

Como forma de tornar o algoritmo implementado mais versátil e completo implementou-se um conjunto de predicados que tem como objetivo pesquisar todos os percursos calculados pelas predicados explicitadas anteriormente.

```
% Percurso entre um inicio e a garagem
percursoGulosa(Fim,Caminho,Custo) :- garagem(Inicio),
                                    resolveGulosa(Inicio,Fim,Caminho,Custo).

% Todos os Percursos entre da garagem até ao deposito
allPercursoGulosa(Fim,L) :- findall((S,C),percursoGulosa(Fim,S,C),L).

% Todos os Percursos entre uma fonte e um destino
allResolveGulosa(Fim,Inicio,L) :- findall((S,C),resolveGulosa(Inicio,Fim,S,C),L).
```

Figura 24: Outros predicados implementados na procura *Gulosa*

Complexidade

Este algoritmo não é completo, pois é possível que este entre em ciclo e é suscetível de falsos começos e de cálculos repetitivos. Além disso também não é ótima pois nem sempre encontra a solução ótima tende a escolher a solução que á primeira vez aparenta ser a mais económico. Contudo, é simples, de fácil implementação e de rápida execução

No que toca à complexidade temporal, este tem um valor de b^mcom b a representar o número máximo de sucessores de um nó e m a máxima profundidade da árvore. Contudo, com uma boa função heurística pode diminuir consideravelmente.

A complexidade espacial é igual á complexidade espacial. Isto deve-se ao facto de este mantém todos os nós em memória.

A-STAR – PROCURA A*

A procura A* ou *A-star* é um algoritmo combina a pesquisa gulosa com a uniforme, minimizando a soma do caminho já efetuado com o mínimo previsto que falta até a solução. Ou seja, ao contrário da pesquisa gulosa só expande o nodo o que aparenta estar mais perto da solução através do somatório entre a heurística e o seu custo o mesmo processo para os restantes nodos adjacentes. Desta forma evita expandir caminhos que são caros.

Para o caso do problema em mão, este algoritmo começa a sua iteração na garagem progredindo a partir da expansão do caixote de lixo adjacente ao nodo anterior que possua menor somatório entre heurística e o seu custo até que seja encontrado o alvo da procura, isto é, chegando ao ponto exatamente antes do depósito.

A heurística utilizada neste algoritmo é a mesmo que foi aplicada ao algoritmo *greedy* (figura 21), isto é, calcula a distância euclidiana em linha reta de um ponto ao depósito, pois é este o vértice que consideramos o real destino do percurso. Para isso, através da latitude e longitude tanto do ponto a pesquisar como da garagem calculamos a distância euclidiana entre esses pontos. No caso de estudo o nodo com menor soma entre heurística e o custo será aquele que vai ser o primeiro a ser acrescentado á solução.

O modo de implementação deste algoritmo para o caso em estudo passou por criar dois predicados principais: ***resolveAEstrela*** e a ***caminhoAEstrela***. O primeiro predicado consiste em inicializar as estruturas necessárias à procura, enquanto o segundo consiste em pesquisar recursivamente os nodos do grafo implementado utilizando a procura A*.

Inicialização

Para proceder à inicialização da procura A* implementou-se o predicado ***resolveAEstrela*** que tem como parâmetros o ponto inicial do percurso, isto é a garagem de onde parte o camião, o destino e o percurso realizado entre a fonte e o ponto final.

Este predicado começa por chamar o ***caminhoAEstrela*** que vai determinar um caminho entre a origem e o destino. Além disso, de forma a cumprir com a capacidade do camião vamos determinar quantas vezes e em que alturas é que o camião terá de ir ao depósito descarregar o lixo e posteriormente voltar ao ponto onde estava através do predicado ***adicionaDeposito***. Finalmente calculamos a distância total do percurso através da ***distanciaTotal*** e o número de pontos por onde passamos.

```
%--- INICIALIZAÇÃO ----%
resolveAEstrela(Inicio,Fim,Percurso,CustoTotal) :- heuristica(Inicio,Estima),
                caminhoAestrela(Fim,[[Inicio]/0/Estima],InvCaminho/Custo/_),
                adicionaDeposito(InvCaminho,Percurso,Acc),
                distanciaTotal(Percurso,CustoTotal).
```

Figura 25: Predicado *resolveAEstrela*

Pesquisa

O predicado ***caminhoAEstrela*** tem como objetivo encontrar um caminho utilizando o algoritmo A* que começa na garagem (origem) e acabe num ponto destino (ponto exatamente antes da última ida ao depósito). Para isso a predicado recebe como parâmetros o ponto final, os vários caminhos encontrados pelo algoritmo e a solução encontrada, isto é o melhor caminho da lista de caminhos encontrados que tenha como destino o ponto final recebido como parâmetro.

Como em Prolog a recursividade é realizada do fim para o início, então o caso de paragem será quando a o melhor caminho obtido tem como nodo final o destino recebido. A obtenção do melhor caminho é dada pelo predicado ***obtem_melhor_gestrela*** que avalia as heurísticas e o custo dos pontos dos caminhos pertencentes à lista e determina quais os pontos que deverão fazer parte da solução encontrada.

Para o caso recursivo da procura, este é semelhante ao implementado na pesquisa gulosa uma vez que ambas utilizam heurísticas para formação do melhor caminho. Assim, vamos determinando com base na avaliação fornecida pela heurística e pelo custo quais os caminhos que são melhores em cada ponto convergindo para o melhor caminho calculado.

```
%Caso de paragem
caminhoAestrela(Fim,Caminhos, Caminho) :- obtem_melhor_gestrela(Caminhos,Caminho),
                                          Caminho = [Fim|_]/_/_.
%Caso recursivo
caminhoAestrela(Fim,Caminhos, SolucaoCaminho) :- obtem_melhor_gestrela(Caminhos,MelhorCaminho),
                                                 seleciona(MelhorCaminho,Caminhos,OutrosCaminhos),
                                                 expande_estrela(MelhorCaminho,ExpCaminhos),
                                                 append(OutrosCaminhos,ExpCaminhos,NovoCaminhos),
                                                 caminhoAestrela(Fim,NovoCaminhos,SolucaoCaminho).
```

Figura 26: Predicado *caminhoAEstrela*

Outros Predicados

Como forma de tornar o algoritmo implementado mais versátil e completo implementou-se um conjunto de predicados que tem como objetivo pesquisar todos os percursos calculados pelas predicados explicitadas anteriormente.

```
%Percurso desde da garagem até ao deposito
percursoAEstrela(Fim, Caminho,Custo) :- garagem(Inicio),
                                        resolveAEstrela(Inicio,Fim,Caminho,Custo).

% Todos os Percursos entre da garagem até ao deposito
allPercursoAEstrela(Fim,L) :- findall((S,C),percursoAEstrela(Fim,S,C),L).

% Todos os Percursos entre um incio até ao deposito
allPercursoAEstrela(Inicio,Fim,L) :- findall((S,C),resolveAEstrela(Inicio,Fim,S,C),L).
```

Figura 27: Outros predicados implementados para a procura A*

Complexidade

Este algoritmo, ao contrário da pesquisa gulosa é completo e ótimo, pois ao avaliar tanto a heurística como o custo este não entre em ciclo evitando cálculos repetitivos e soluções que aparentam ser ótimas.

A complexidade temporal de A* depende da heurística. No caso de um problema com procura ilimitada, o número de nodos expandidos é exponencial na profundidade da solução (o caminho mais curto) d: b^d, onde b é o fator de ramificação (o número médio de sucessores por estado). Isso pressupõe que existe um estado-objetivo e é alcançável desde o início do estado; se não for, e o espaço o é infinito, o algoritmo não terminará.

A complexidade espacial é aproximadamente a mesma da pesquisa gulosa, pois mantém todos os nós gerados na memória.

ANÁLISE DE RESULTADOS

Nesta secção estão expostos resultados dos diversos algoritmos apresentados anteriormente, assim como uma análise comparativa entre eles e uma avaliação dos critérios que fazem um caminho ser o melhor percurso aplicado ao algoritmo de *depth first*.

AVALIAÇÃO DO MELHOR CAMINHO

A avaliação do melhor caminho através de certos critérios e indicadores passou por implementar certos predicados aplicados aos algoritmos não informados de forma a avaliar o percurso obtido por cada um deles. Neste caso vamos avaliar os percursos e critérios para a estratégia de *depth first* admitindo que para os restantes algoritmos não informados a avaliação é semelhante. Também vamos analisar o percurso do ponto 483, R Dom Luís I até ao ponto 494, R Ribeira Nova através da avaliação dos vários critérios e indicadores de produtividade.

Na figura abaixo encontram-se os vários caminhos de 483 a 494 obtidos pelo algoritmo *depth first* explicitado na secção Algoritmos.

```
?- resolveDF(483,494,L,Lixo,D,C).
L = [483,484,485,486,487,488,489,490,491,494,1],
Lixo = 3790,
D = 0.011622589810185551,
C = 11 ;
L = [483,484,485,486,487,488,489,490,491,494,1],
Lixo = 3790,
D = 0.011622589810185551,
C = 11 ;
L = [483,484,485,486,487,488,489,490,491,494,1],
Lixo = 3790,
D = 0.011622589810185551,
C = 11 ;
L = [483,484,485,486,487,488,489,490,491,494,1],
Lixo = 3790,
D = 0.011622589810185551,
C = 11 ;
L = [483,484,485,486,487,488,489,490,494,1],
Lixo = 3510,
D = 0.011622589810185551,
C = 10 ;
L = [483,484,485,486,487,488,489,494,1],
Lixo = 3230,
D = 0.011622589810185551,
C = 9 ;
```

Figura 28: Lista dos vários caminhos entre o ponto 483 e 494

Através do predicado ***maisPontosRecolhaDF*** conseguimos obter o caminho da lista apresentada na figura 29 que possui mais pontos de recolha. Este critério mostra quantas paragens é que o camião fez avaliando, desta forma, se o percurso foi eficiente no que toca ao número de pontos percorridos. Para a implementação deste predicado foi utilizado o ***allPercursoDF*** que a partir do predicado *findall* cria a lista de caminhos apresentada na figura 28. De seguida através do predicado ***maisPontoRecolha*** determinamos qual o caminho da lista que tem comprimento maior. A implementação deste predicado e dos predicados análogos para os restantes algoritmos não informados estão representados na figura 30.

```
?- maisPontoRecolhaDF(494,L).
L =  ([483,484,485,486,487,488,489,490,491,494,1],3790,0.011622589810185551,11) ;
```

Figura 29: Percurso com maior número de pontos de recolha para a procura *Depth First*

```
% Caminho com mais ponto de recolha apartir do algoritmo Depth First
maisPontoRecolhaDF(Fim,Recolha) :- allPercursoDF(Fim,L),
                                   maisPontoRecolha(L,Recolha).

% Caminho com mais ponto de recolha apartir do algoritmo Breadth First
maisPontoRecolhaBF(Fim,Recolha) :- allPercursoBF(Fim,L),
                                   maisPontoRecolha(L,Recolha).

% Caminho com mais ponto de recolha apartir do algoritmo Depth First Limitado
maisPontoRecolhaDFLimitado(Fim,Limite,Recolha) :-
                                   allPercursoDFLimitado(Fim,Limite,L),
                                   maisPontoRecolha(L,Recolha).
```

Figura 30: Predicados para obtenção dos caminhos com mais pontos de recolha

Além deste predicado também foi implementado um outro, ***maisPontosTipoRecolhaDF*** semelhante ao primeiro só que avalia qual o caminho com mais pontos de recolha percorrendo seletivamente um tipo de lixo. A implementação deste predicado é semelhante ao ***maisPontosRecolhaDF***, utilizando o ***allPercursoTipoDF*** que a partir do predicado *findall* cria a lista de caminhos seletivos no tipo de lixo que recolhem e através do ***maisPontoRecolha*** determinamos qual o caminho da lista que tem comprimento maior. Na figura 32 encontra-se a implementação deste predicado assim como os predicados análogos aplicados a cada uma das estratégias não informadas, já na imagem 31 encontra-se o resultado de aplicar ao caminho a recolha por Papel e Cartão para os algoritmos*Depth First*.

```
    maisPontoTipoRecolhaDF(494,'Papel e Cartão',L).
L =  ([483,484,485,486,487,488,489,490,491,494,1],280,0.009815746693770232,1) ;
```

Figura 31: Caminho com mais pontos de recolha com lixo 'Papel e Cartão'

```
% Caminho com mais ponto de recolha com selecao de lixo apartir do algoritmo Depth First
maisPontoTipoRecolhaDF(Fim,Lixo,Recolha) :-
                                   allPercursoTipoDF(Fim,Lixo,L),
                                   maisPontoRecolha(L,Recolha).

% Caminho com mais ponto de recolha com selecao de lixo apartir do algoritmo Breadth First
maisPontoTipoRecolhaBF(Fim,Lixo,Recolha) :-
                                   allPercursoTipoBF(Fim,Lixo,L),
                                   maisPontoRecolha(L,Recolha).

% Caminho com mais ponto de recolha com selecao de lixo apartir do algoritmo Depth First Limitado
maisPontoTipoRecolhaDFLimitado(Fim,Lixo,Limite,Recolha) :-
                                   allPercursoTipoDFLimitado(Lixo,Fim,Limite,L),
                                   maisPontoRecolha(L,Recolha).
```

Figura 32: Predicados para obtenção dos percursos seletivos com mais pontos de recolha

Para avaliação do caminho segundo indicadores de quantidade recolhida e distancia percorrida decidiu-se implementar dois predicadores, ***produtividadeLixoDF*** e ***produtividadeDistanciaDF*** que compara os caminhos obtidos segundo a quantidade de lixo recolhida e distancia percorrida respetivamente. Nas imagens seguintes temos a resultado de cada um dos predicados. Estas predicados foram implementadas de forma semelhante às supramencionadas, tal como podemos ver na figura 33. Avaliando os outputs dos dois predicados podemos ver que o caminho mais longo é a aquele que tem mais lixo recolhido, tal como está na figura 34.

```
%--------------------------------- - - - - - - - - - - - - - - - - -
% Circuitos com Indicadores de produtividade : Quantidade Recolhida
%--------------------------------- - - - - - - - - - - - - - - - - -

produtividadeLixoDF(Fim,Produtividade) :- allPercursoDF(Fim,L),
                                          produtividadeLixo(L,[],Produtividade).

produtividadeLixoBF(Fim,Produtividade) :- allPercursoBF(Fim,L),
                                          produtividadeLixo(L,[],Produtividade).

produtividadeLixoDFLimitado(Fim,Limite,Produtividade) :-
                                          allPercursoDFLimitado(Fim,Limite,L),
                                          produtividadeLixo(L,[],Produtividade).

%--------------------------------- - - - - - - - - - - - - - - - - -
% Circuitos com Indicadores de produtividade : Distancia Media percorrida
%--------------------------------- - - - - - - - - - - - - - - - - -

produtividadeDistanciaDF(Fim,Produtividade) :- allPercursoDF(Fim,L),
                                          produtividadeDistancia(L,[],Produtividade).

produtividadeDistanciaBF(Fim,Produtividade) :- allPercursoBF(Fim,L),
                                          produtividadeDistancia(L,[],Produtividade).

produtividadeDistanciaDFLimitado(Fim,Limite,Produtividade) :-
                                          allPercursoDFLimitado(Fim,Limite,L),
                                          produtividadeDistancia(L,[],Produtividade).
```

Figura 33: Predicados que implementam os indicadores de produtividade

```
?- produtividadeDistanciaDF(494,L).
L = [([483,484,485,486,487,488,489,494,1],0.011622589810185551),([483,484,485,486,487,488,489,490,494,1],0.011622589810185551),
([483,484,485,486,487,488,489,490,491,494,1],0.011622589810185551),([483,484,485,486,487,488,489,490,491,494,1],0.0116225898101
85551),([483,484,485,486,487,488,489,490,491,494,1],0.011622589810185551),([483,484,485,486,487,488,489,490,491,494,1],0.011622
589810185551)].

?- produtividadeLixoDF(494,L).
L = [([483,484,485,486,487,488,489,494,1],3230),([483,484,485,486,487,488,489,490,494,1],3510),([483,484,485,486,487,488,489,49
0,491,494,1],3790),([483,484,485,486,487,488,489,490,491,494,1],3790),([483,484,485,486,487,488,489,490,491,494,1],3790),([483,
484,485,486,487,488,489,490,491,494,1],3790)].
```

Figura 34: Resultado após a aplicação dos predicados para a procura *Depth First*

Para obtenção do caminho mais rápido decidiu-se implementar o predicado ***maisRapidoDF*** que analogamente ao ***maisPontosRecolhaDF*** vai determinar qual o caminho com menos distancia percorrida tal como esta na imagem. Na figura 35 encontra-se a implementação do predicado que tal como podemos é semelhante ao ***maisPontosRecolhaDF***.

```
maisRapidoDF(Fim,Rapido) :- allPercursoDF(Fim,L),
                            maisRapido(L,Rapido).

maisRapidoBF(Fim,Rapido) :- allPercursoBF(Fim,L),
                            maisRapido(L,Rapido).

maisRapidoDFLimitado(Fim,Limite,Rapido) :-
                            allPercursoDFLimitado(Fim,Limite,L),
                            maisRapido(L,Rapido).
```

Figura 35: Predicados para a obtenção do caminho mais rápido

```
?- maisRapidoDF(494,L).
L =  ([483,484,485,486,487,488,489,490,491,494,1],3790,0.011622589810185551,11) .
```

Figura 36: Resultado do predicado *maisRapidoDF*

Finalmente, para a obtenção do caminho mais eficiente considerou-se que a eficiência da recolha de lixo está em percorrer o menor número de pontos recolhendo a maior quantidade de lixo possível. Para isso avaliou-se o lixo recolhido e o tamanho do percurso através de um predicado ***eficiencia*** (figura 37) que determina a eficiência de dois caminhos partir da divisão do lixo recolhido pelo número de pontos de recolha percorridos. Assim quanto maior for este valor da divisão mais eficiente é o caminho. O predicado ***maisEfeciente*** permite determinar qual o percurso da lista obtida pelo ***allPercursoDF*** que tem maior eficiência. Nas figuras 38 e 39 podemos ver a implementação dos predicados como também o caminho mais eficiente entre o ponto 483 e 489, respetivamente.

```
efeciencia((S,Lixo,D,C),(S1,Lixo1,D1,C1),E,E1) :- E is Lixo/C, E1 is Lixo1/C1.
```

Figura 37: Predicado *efeciencia*

```
maisEfecienteDF(Fim,Efeciencia) :- allPercursoDF(Fim,L),
                            maisEfeciente(L,Efeciencia).

maisEfecienteBF(Fim,Efeciencia) :- allPercursoBF(Fim,L),
                            maisEfeciente(L,Efeciencia).

maisEfecienteDFLimitado(Fim,Limite,Efeciencia) :-
                            allPercursoDFLimitado(Fim,Limite,L),
                            maisEfeciente(L,Efeciencia).
```

Figura 38: Predicados para o caminho mais eficiente para estratégias não informada

```
?- maisEfecienteDF(494,L).
L =  ([483,484,485,486,487,488,489,494,1],3230,0.011622589810185551,9) .
```

Figura 39: Resultado do predicado *maisEfecienteDF*

COMPARAÇÃO DE ESTRATÉGIAS

De forma a manter a consistência entre os resultados obtidos foi aplicado a cada estratégia o mesmo ponto de início – 385, Tv Corpo Santo – e de destino – 335, Lg Corpo Santo. Assim, cada estratégia só tem um conjunto limitado de caminhos que pode determinar facilitando, posteriormente, a análise de resultados. Para os pontos usados, através de uma análise rápida do grafo conseguimos ver que o caminho mais curto será 385 (garagem) -> 386 -> 333 -> 334 -> 335 -> 1 (deposito).

No que toca à análise de espaço e tempo utilizados por cada estratégia foi utilizado o predicado ***statistics*** e time de forma a ver a memória utilizada pelo algoritmo e o tempo de produção de uma resulta. Também é possível através do predicado time verificar o uso de CPU. Para isso foi utilizado o predicado na imagem abaixo que é análogo para todos os algoritmos implementados, a única diferença é a chama à predicado que forma o caminho para cada estratégia. Para o caso da pesquisa *depth first* é *resolveDF*, para *breadth first* é *resolveBF* e assim sucessivamente tal como está mencionado na secção Algoritmos. Neste predicado avaliamos o estado da ***global_stack*** antes do início da formação do caminho e logo após essa formação para conseguimos avaliar o estado da memória ao fim do predicado de formação. A cada chamada do predicado de cada estratégia chamamos o predicado time de forma a verificar o tempo de execução do predicado.

```
%--------------------------------- - - - - - - - - - - - - - - - - -
% Estatiscas de tempo de execução e memória utilizada para o algoritmo de Breadth First.

statisticsBF(Inicio,Fim,L,Lixo,D,C,Mem) :-
    statistics(global_stack, [Use1,Free1]),
    time(resolveBF(Inicio,Fim,L,Lixo,D,C)),
    statistics(global_stack, [Use2,Free2]),
    Mem is Use2 - Use1.

%--------------------------------- - - - - - - - - - - - - - - - - -
% Estatiscas de tempo de execução e memória utilizada para o algoritmo de Depth First.

statisticsDF(Inicio,Fim,L,Lixo,D,C,Mem) :-
    statistics(global_stack, [Use1,Free1]),
    time(resolveDF(Inicio,Fim,L,Lixo,D,C)),
    statistics(global_stack, [Use2,Free2]),
    Mem is Use2 - Use1.

%--------------------------------- - - - - - - - - - - - - - - - - -
% Estatiscas de tempo de execução e memória utilizada para o algoritmo de Depth First Limitado.

statisticsDFLimitado(PInicio,PFim,Percurso,Lixo,DTotal,C,Limite,Mem) :-
    statistics(global_stack, [Use1,Free1]),
    time(resolveDFLimitado(PInicio,PFim,Percurso,Lixo,DTotal,C,Limite)),
    statistics(global_stack, [Use2,Free2]),
    Mem is Use2 - Use1.
```

Figura 40: Predicado *statistics* para as estratégias não informadas

```
%--------------------------------- - - - - - - - - - - - - - - - -
% Estatiscas de tempo de execução e memória utilizada para o algoritmo de Pesquisa Gulosa

statisticsGulosa(Inicio,Fim,Percurso,CustoTotal,Mem) :-
    statistics(global_stack, [Use1,Free1]),
    time(resolveGulosa(Inicio,Fim,Percurso,CustoTotal)),
    statistics(global_stack, [Use2,Free2]),
    Mem is Use2 - Use1.

%--------------------------------- - - - - - - - - - - - - - - - -
% Estatiscas de tempo de execução e memória utilizada para o algoritmo de Pesquisa A*

statisticsAEstrela(Inicio,Fim,Percurso,CustoTotal, Mem) :-
    statistics(global_stack, [Use1,Free1]),
    time(resolveAEstrela(Inicio,Fim,Percurso,CustoTotal)),
    statistics(global_stack, [Use2,Free2]),
    Mem is Use2 - Use1.
```

Figura 41: Predicado *statistics* para as estratégias informadas

Depth First – Procura em profundidade

O algoritmo *depth first* tal como foi referido em cima não é ótimal e por isso o pode não conseguir determinar o caminho mais curto entre os pontos inicial e destino. Neste caso a estratégia calcula em primeiro lugar o caminho obtido na imagem 42. Como podemos ver não é o caminho mais curto, contudo é um caminho com bastantes pontos atingidos e lixo recolhido.

```
?- resolveDF(385,335,L,Lixo,D,C).
L = [385, 386, 387, 388, 389, 390, 463, 464, 465|...],
Lixo = 31450,
D = 0.038404031654967016,
C = 73
```

Figura 42: Resultado do predicado *resolveDF*

Ao longo dos vários caminhos que o *depth first* determina, só o último caminho encontrado é que é o melhor caminho em termos de distância tal como podemos ver na imagem 43.

```
L = [385, 386, 333, 334, 335, 1],
Lixo = 1450,
D = 0.002657685363666583,
C = 6 ;
```

Figura 43: Caminho mais curto obtido pelo *resolveDF*

Em termos de tempo e de espaço utilizado conseguimos ver na imagem que demorou cerca de 0.002 segundos utilizando cerca de 111176 espaço na *global_stack.*

```
|    statisticsDF(385,335,L,Lixo,D,C,Mem).
% 5,085 inferences, 0.000 CPU in 0.002 seconds (0% CPU, Infinite Lips)
L = [385,386,387,388,389,390,463,464,465,466,467,468,469,470,471,472,473
643,644,645,646,647,648,649,650,651,652,653,654,655,656,657,658,659,660,
Lixo = 31450,
D = 0.038404031654967016,
C = 73,
Mem = 111176 .
```

Figura 44: Resultado das estatísticas para a procura *Depth First*

Breadth First – Procura em largura

No caso do algoritmo *breadth first* tal como foi referido em cima pode ser ótimal se todos os custos escalonados forem iguais. Como a distancia entre os pontos que estamos a usar e euclidiana e como os pontos encontram-se muito próximos uns dos outros podemos considerar que a distancia entre os diversos pontos de recolha é sempre semelhante. Assim, a estratégia de procura em largura vai determinar, á primeira, o melhor caminho em termos de distância que esta apresentado na imagem a baixo.

```
|    resolveBF(385,335,L,Lixo,D,C).
L = [385, 386, 333, 334, 335, 1],
Lixo = 1820,
D = 0.0026576853636666583,
C = 6 ;
```

Figura 45: Resultado do algoritmo *Breadth First*

Em termos de tempo e de espaço utilizado conseguimos ver na imagem que demorou cerca de 0.000 segundos utilizando cerca de 12424 espaço na *global_stack.*

```
?- statisticsBF(385,335,L,Lixo,D,C,Mem).
% 337 inferences, 0.000 CPU in 0.000 seconds (?% CPU, Infinite Lips)
L = [385,386,333,334,335,1],
Lixo = 1820,
D = 0.0026576853636666583,
C = 6,
Mem = 12424 .
```

Figura 46: Estatísticas para a estratégia *Breadth First*

Depth First Limitada – Procura em profundidade Limitada

O algoritmo *depth first limitado*, tal como acontece para a estratégia *depth first* não é otimal e por isso pode não conseguir determinar o caminho mais curto entre o ponto inicial e o destino. Como, no caso do caminho entre 385 e 335 o mínimo de pontos por onde o camião tem, obrigatoriamente de percorrer é 5 então caso o limite de profundidade seja menor que 5 este não vai encontrar solução, tal como acontece na imagem 47.

```
?- resolveDFLimitado(385,335,L,Lixo,D,C,4).
false.
```

Figura 47: Resultado do predicado *resolveDFLimitado* com limite 4

Caso limitemos a profundidade da procura para 10 já vamos obter não só o caminho mais curto como também outros caminhos tal como podemos ver na imagem abaixo. Desta forma quanto maior for o limite mais soluções obtemos.

```
?- resolveDFLimitado(385,335,L,Lixo,D,C,10).
L = [385, 386, 387, 388, 389, 390, 333, 334, 335|...],
Lixo = 4030,
D = 0.002657685363666583,
C = 10 ;
L = [385, 386, 387, 388, 389, 333, 334, 335, 1],
Lixo = 3790,
D = 0.002657685363666583,
C = 9 ;
L = [385, 386, 387, 388, 333, 334, 335, 1],
Lixo = 2830,
D = 0.002657685363666583,
C = 8 ;
L = [385, 386, 387, 333, 334, 335, 1],
Lixo = 2410,
D = 0.002657685363666583,
C = 7 ;
L = [385, 386, 333, 334, 335, 1],
Lixo = 1450,
D = 0.002657685363666583,
C = 6 ;
false.
```

Figura 48: Resultado do predicado *resolveDFLimitado* com limite 10

Em termos de tempo e de espaço utilizado conseguimos ver na imagem que demorou cerca de 0.000 segundos utilizando cerca de 15336 espaço na *global_stack*, para um limite baixo de profundidade. Para um limite de profundidade maior, imagem x, podemos ver que o tempo utilizado aumenta para 0.001 e o espaço também aumenta para 112840.

```
?- statisticsDFLimitado(385,335,L,Lixo,D,C,10,Mem).
% 741 inferences, 0.000 CPU in 0.000 seconds (?% CPU, Infinite Lips)
L = [385,386,387,388,389,390,333,334,335,1],
Lixo = 4030,
D = 0.002657685363666583,
C = 10,
Mem = 15336 .
```

Figura 49: Estatísticas para a estratégia *Depth First Limitado* com limite 10

```
?- statisticsDFLimitado(385,335,L,Lixo,D,C,70,Mem).
% 5,154 inferences, 0.000 CPU in 0.001 seconds (0% CPU,
L = [385,386,387,388,389,390,463,464,465,466,467,468,4
643,644,645,646,647,648,649,650,651,652,653,654,655,65
Lixo = 31450,
D = 0.038404031654967016,
C = 73,
Mem = 112840 .
```

Figura 50: Estatísticas para a estratégia *Depth First Limitado* com limite 70

Greedy – Procura Gulosa

No caso da pesquisa gulosa tal como foi referido em cima nem sempre esta converge para a solução ótima, contudo no caso do problema em questão o primeiro caminho obtido foi o caminho mais curto, tal como esta na imagem abaixo.

```
?- resolveGulosa(385,335,L,C).
L = [385, 386, 333, 334, 335, 1],
C = 0.0026576853636665 83 ;
```

Figura 51: Resultado do predicado *resolveGulosa*

Em termos de tempo e de espaço utilizado conseguimos ver na imagem que demorou cerca de 0.000 segundos utilizando cerca de 12232 espaço na *global_stack.*

```
|    statisticsGulosa(385,335,L,D,Mem).
% 717 inferences, 0.000 CPU in 0.000 seconds (?% CPU, Infinite Lips)
L = [385,386,333,334,335,1],
D = 0.0026576853636665 83,
Mem = 12232 .
```

Figura 52: Estatísticas da procura gulosa

A-Star – Procura A*

No caso da pesquisa A* tal como foi referido em cima esta converge para uma solução ótima caso a heurística seja considerado boa. No caso em questão, o primeiro caminho obtido a partir desta estratégia é o caminho mais curto, tal como podemos ver na imagem seguinte. Desta forma podemos considerar que a heurística utilizada (distancia euclidiana em linha reta entre um ponto e o deposito) é uma boa heurística a utilizar.

```
?- resolveAEstrela(385,335,L,C).
L = [385, 386, 333, 334, 335, 1],
C = 0.0026576853636665 83 ;
L = [385, 384, 386, 333, 334, 335, 1],
C = 0.0026576853636665 83 ;
L = [385, 386, 387, 333, 334, 335, 1],
C = 0.0026576853636665 83
```

Figura 53: Resultado do predicado *resolveAEstrela*.

Em termos de tempo e de espaço utilizado conseguimos ver na imagem que demorou cerca de 0.001 segundos utilizando cerca de 22024 espaço na *global_stack*.

```
?- statisticsAEstrela(385,335,L,D,Mem).
% 1,310 inferences, 0.000 CPU in 0.001 seconds (0% CPU, Infinite Lips)
L = [385,386,333,334,335,1],
D = 0.0026576853636665 83,
Mem = 22024 .
```

Figura 54: Estatísticas da procura A*

ANÁLISE COMPARATIVA

De seguida está apresentado uma tabela com os vários parâmetros avaliadas em cima para cada estratégia de forma a facilitar a comparação dos vários algoritmos. No que toca ao parâmetro de melhor solução assume-se um caminho é melhor que o outro caso a sua distância seja melhor. Também podíamos avaliar em termos de lixo recolhido ou eficiência, entre outros. Caso a estratégia não tenha convergido para a melhor solução ao fim da primeira chamada à predicado então consideramos que esta não convergiu para a melhor solução.

Estratégia	Tempo (segundos)	Espaço	Profundidade/ Custo	Encontrou a melhor solução?
Depth First	0.002	111176	0.034	Não
Breadth First	0.000	12424	0.00266	Sim
Depth First Limitada[2]	0.000	15336	0.002658	Não
Greedy Search	0.000	12232	0.00265768	Sim
A* Search	0.001	22024	0.00265768	Sim

Tabela 1: Resumo com as estatísticas das estratégias

Para o caminho com início em 385, Tv Corpo Santo e de destino em 335, Lg Corpo Santo, podemos ver que o *depth first* é mais lento e utiliza mais memória que o *breadth fisrt*, sendo esta última uma boa estratégia para solucionar o problema em mãos. Além disso, podemos ver que as estratégias de procura informada usando a heurística previamente mencionada também são bons algoritmos a aplicar no problema, pois ambas atingem a melhor solução em pouco tempo utilizando pouco espaço.

Concluímos assim, que para o problema em causa considera-se que o algoritmo que deve ser aplicado de forma a obter o melhor caminho entre a garagem e o deposito de forma mais eficiente é o algoritmo *greedy*, pois o espaço de memória ocupado e tempo de execução foi o menor e conseguiu encontrar a melhor solução segundo a distância percorrida.

[2] Para limite de 10, pois como este limite a predicado encontra facilmente a melhor solução.

CONCLUSÃO

Dada por concluída a projeto individual, considera-se relevante efetuar uma análise crítica do trabalho realizado, apontando aspetos positivos e negativos, resumindo os resultados finais e dando ênfase a possíveis evoluções futuras do trabalho.

É de realçar os aspetos positivos o trabalho desenvolvido, entre eles a implementação do limite da capacidade do camião tornando assim solução encontra mais completa e realista. Além disso, a existência de diversas funcionalidades que permitem a obtenção dos melhores caminhos segundo critérios e indicadores produtividade. Finalmente, existem uma boa documentação quer no código implementado como também no presente relatório. Desta forma, torna-se mais fácil a leitura e a compreensão de tudo aquilo que foi desenvolvido ao longo do trabalho.

No que toca às dificuldades sentidas, considera-se que a maior de todas foi a implementação do algoritmo de *breadth first* na linguagem Prolog assim como implementação eficiente do limite da capacidade para o camião. Além disso, a maneira como se ia interpretar e filtra a informação e como se ia lidar com a elevada quantidade de informação contida no *dataset* fornecido pelos docentes foi uma das dificuldades sentidas.

Considera-se que o resultado final se revelou bastante satisfatório dado que alberga todas as funcionalidades e os vários algoritmos propostos. Numa versão futura do trabalho, poder-se-ia melhorar a implementação do algoritmo *breadth first* assim como a implementação da limitação do camião visto que este último não foi implementado durante o processo recursivo das estratégias, mas sim fora deste. Além disso, a implementação dos critérios e indicadores de avaliação para os vários caminhos deviam ser estendidos para as estratégias informadas.

Para concluir, considera-se que houve um balanço positivo do trabalho realizado dado que as dificuldades sentidas foram superadas e obtendo um sistema completo de acordo com o proposto.

REFERÊNCIA

REFERÊNCIAS BIBLIOGRÁFICAS

[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.