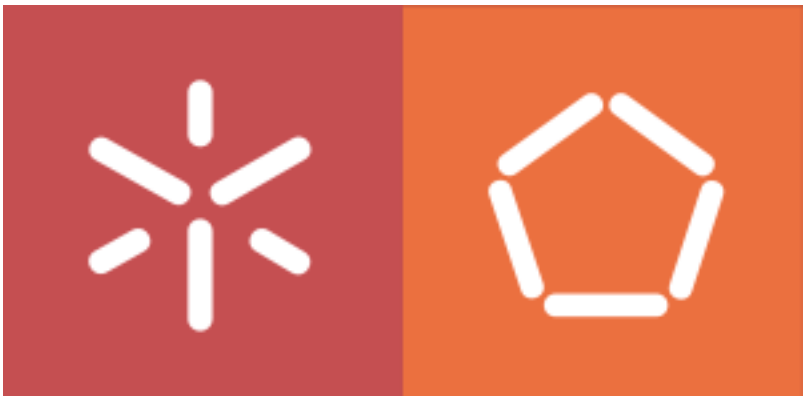

UNIVERSIDADE DO MINHO

Métodos de Resolução de Problemas e de Procura

MIEI

Sistemas de Representação de Conhecimento e Raciocínio

(2º Semestre - 2020/2021)

A89557 Pedro Miguel Dias Veloso

Maio 2021

Índice

Índice Tabelas

Índice Figuras

Introdução

O presente trabalho propõe a determinação de circuitos de recolha de resíduos urbanos na área de Lisboa. Para tal, utilizou-se um *dataset* inicial com um conjunto de dados relativos aos pontos de recolha – coordenadas, localidade, tipo de contentor e respetiva capacidade de armazenamento.

Através da informação anterior, pretende-se deduzir, numa primeira fase, a representação de um grafo onde será inferida a seguinte informação:

1. Circuitos de recolha que cubram um determinado território;
2. Circuitos de recolha com maior abrangência (por tipo de resíduo);
3. Circuitos de recolha com maior índice de produtividade;
4. Circuito de recolha com menor distância percorrida (circuito mais rápido);
5. Circuito de recolha mais eficiente.

De modo a deduzir a informação listada anteriormente, numa segunda fase, serão aplicados algoritmos com métodos de procura informada e não informada.

Para simplificar os algoritmos de procura, optou-se por não limitar a capacidade máxima dos veículos de recolha, assumindo-se assim, que estes possuem capacidade ilimitada de transporte. Além disso, consideram-se as distâncias em linha reta de um nodo a outro do grafo, abstraindo-se do mapa topográfico da cidade.

Descrição do trabalho e análise dos resultados

O presente capítulo pretende descrever o problema em análise, bem como apresentar as diferentes estratégias ou algoritmos de procura para determinar um caminho entre dois pontos de um grafo – procura não informada ou informada.

Por fim, são apresentadas as estatísticas que permitem diferenciar o grau de eficiência dos diferentes algoritmos.

Formulação do problema

No presente trabalho, tal como referido no capítulo anterior, tem-se como objetivo encontrar um circuito válido que começa e termina na garagem, onde o camião se encontra ou deve-se encontrar estacionado. Contudo, durante o circuito o camião deve passar pelos pontos de recolha, deslocando-se, após terminar, ao deposito do lixo, antes de voltar à garagem. Considerando que a localização da garagem, do depósito e dos respetivos pontos de recolha é conhecida, este problema enquadra-se num problema de estado único, onde a sua formulação será apresentada nas secções seguintes.

Estado inicial

Como o camião parte da garagem, o seu estado inicial é dado por essa localização. Contudo, uma vez que no *dataset* fornecido não é apresentada informação sobre a localização da garagem, admitiu-se a seguinte: 38.7073286838222, -9.14255098678099.

Ações

Neste problema, apenas é possível realizar uma viagem de um ponto de recolha/garagem para um outro ponto de recolha/depósito, sendo, por isso, esta a única ação possível.

Modelo de transição

Após realizar uma ação (viajar de um ponto para outro), o estado atual é modificado para a localização do ponto que o camião se moveu.

Estado final

O estado do problema deverá ser a garagem onde o camião deve se estacionar, contudo, uma vez que o depósito tem uma localização fixa e que o deslocamento entre o depósito e a garagem no final do circuito é obrigatória e direta (o camião desloca-se sem passar por nenhum ponto de recolha), considera-se que o estado final do problema é o depósito de descarga de lixo. Uma vez mais como não é fornecida a localização do depósito utilizou-se a seguinte localização: 38.7056911463271, -9.14752862812709.

Custo

O custo de um caminho possível e válido no âmbito do problema descrito é dado pelo somatório das distâncias percorridas ao longo de cada etapa.

Representação do grafo global

Para a resolução do problema foi fornecido um *dataset* com o conjunto de pontos de recolha que um camião deve percorrer. No *dataset* estavam registadas as coordenadas GPS dos pontos, bem como um conjunto de dados adicionais, como por exemplo o tipo de lixo contido no contentor. Uma vez que para uma localização pode existir vários pontos de recolha com tipos diferentes, realizou-se a agregação desses pontos num único, ao qual se associou uma lista com os tipos de contentores existentes. Assim, obteve-se um conjunto de 121 pontos (Figura 1) dos quais se utilizaram dois para representar a garagem e o depósito.

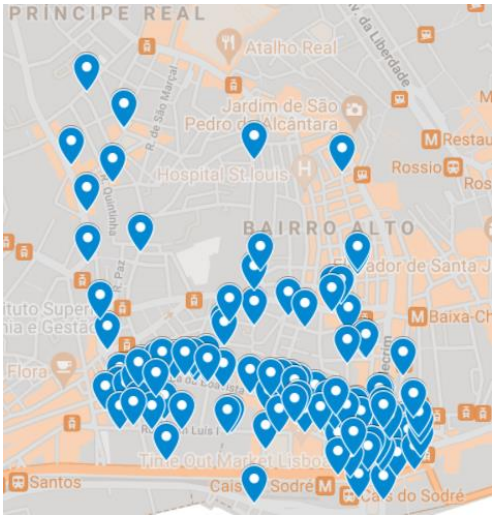

Figura 1: Pontos de recolha agregados por localização

Posteriormente, realizou-se o processo de identificar as arestas que unem cada um dos pontos de recolha. Uma vez que, o grafo a representar corresponde a um conjunto de ligações de coordenadas GPS é lógico considerar a existência de ligações de um nodo para qualquer outro, uma vez que existem sempre estradas que permitem realizar esse deslocamento. Porém, ao realizar esta estratégia obtém-se mais de 14 mil arestas, que tornam a execução dos algoritmos, especificados posteriormente neste relatório, muito demorada. Desta forma, foi necessário diminuir o número de conceções para as 478 identificadas e assumidas durante a realização do trabalho.

Primeiramente, pensou-se em limitar o número de arestas, ao assumir que de cada ponto apenas se atinja os pontos que estão a uma distância inferior a um determinado valor. Contudo, com esta estratégia obteve-se conjuntos de pontos isolados, que para resolver era necessário aumentar a distância considerada, aproximando-se assim do número de arestas já obtidas.

Por fim, decidiu-se unir cada ponto aos 2 pontos acima/abaixo no *dataset*. Com esta estratégia é possível garantir que existe um caminho de um determinado ponto para outro ponto qualquer, e uma vez que se liga aos 2 pontos acima/abaixo garante-se também conjuntos de caminhos diferentes.

Para realizar a união dos pontos, acima explicada, bem como a união das arestas, realizou-se um programa em *python* (Anexo II) capaz de realizar o *parsing* do *datset* fornecido.

Estratégias de procura

Para diferenciar os diferentes circuitos de recolha de resíduos, é importante conhecer e distinguir as diferentes estratégias e algoritmos de procura, bem como ter a capacidade de os traduzir para a linguagem *prolog*. Em seguida, serão apresentadas duas estratégias de procura – a não informada e a informada.

Procura não informada

A procura não informada apenas distingue o estado atual do estado de destino, ignorando qualquer outra variável que poderá encurtar o algoritmo. Existem vários algoritmos que podem ser enquadrados neste tipo de procura, entre eles a procura em profundidade, em largura e limitada em profundidade.

Procura em profundidade

Neste tipo de procura, pretende-se que o algoritmo seja capaz de realizar a procura num grafo, partindo sempre de um nodo inicial e progredindo para o nodo adjacente mais profundo possível. Quando chegar ao último nodo do ramo, retrocede ao nodo filho anterior e continua a partir desse, até chegar ao destino (Figura 2). Deste modo, a complexidade deste algoritmo é proporcional ao número de vértices e de arestas do grafo.

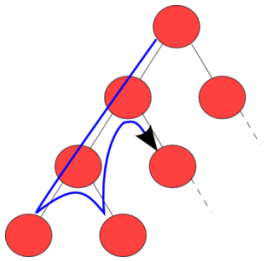

Figura 2: Algoritmo de pesquisa em profundidade

Para o problema apresentado anteriormente, a representação do algoritmo em *prolog* pode ser encontrada no código seguinte:

```
resolve_pp_k(NodosPercorrer, Nodo, Final, [Nodo|Caminho], K) :- profundidadeprimeirok(NodosPercorrer, Nodo,Final,[Nodo],Caminho, 0, K).

profundidadeprimeirok(NodosPercorrer, Nodo,Nodo,_,[],A, A).
profundidadeprimeirok(NodosPercorrer, Nodo,Final, Historico, [ProxNodo|Caminho], A, K) :- adjacenteK(NodosPercorrer, Nodo,ProxNodo, KT),
                                                                                           nao(membro(ProxNodo,Historico)),
                                                                                           N is A + KT,
                                                                                           profundidadeprimeirok(NodosPercorrer,ProxNodo,Final,[ProxNodo|Historico],Caminho, N, K).

adjacenteK(NodosAdmissiveis, Nodo, ProxNodo, K) :- aresta(Nodo,ProxNodo),
                                                   pertence(Nodo,NodosAdmissiveis),
                                                   pertence(ProxNodo,NodosAdmissiveis),
                                                   nodo(Nodo,Lat1,Lon1,_,_,_),
                                                   nodo(ProxNodo,Lat2,Lon2,_,_,_),
                                                   distancia(Lat1,Lon1,Lat2,Lon2,K).
```

Figura 3: Algoritmo de pesquisa em profundidade - prolog

Da figura anterior, é possível extrair que para realizar uma procura em profundidade é necessário introduzir uma lista de todos os nodos que poderão ser percorridos, o nodo de partida e o final. Assim, em cada iteração é realizada a procura através de um nodo adjacente ao atual, escolhendo-

o como nodo de destino dessa iteração. Na iteração seguinte, esse nodo passa a ser o ponto de partida. Desta forma, são realizadas tantas iterações quantas necessárias até ser encontrado o destino do problema ou insucesso da procura.

Procura em largura

Num tipo de procura em largura, o algoritmo começa no nodo inicial e explora todos os nodos vizinhos, para cada nível da árvore, até que seja encontrado o nodo de destino final (Figura 4).

Tal como acontece na estratégia anterior, a complexidade deste algoritmo é proporcional ao número de vértices e de arestas do grafo.

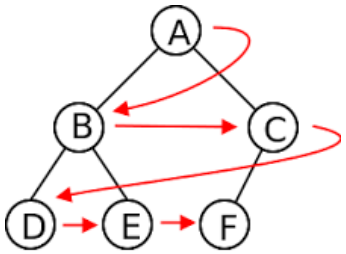

Figura 4: Algoritmo de pesquisa em largura

Em *prolog*, foi aplicado o código da Figura 5 para representar este tipo de procura aplicado ao problema anterior.

```
resolveBF(No, Final, Solucao) :- findall(N,nodo(N,_,_,_,_,_),NA), breadthFirst(NA, Final, [[No]], Solucao1), inverso(Solucao1,Solucao).

breadthFirst(NodosPercorrer, Final, [[No|Caminho]|_], [No|Caminho]) :- Final =:= No.
breadthFirst(NodosPercorrer, Final, [Caminho|Caminhos], Solucao) :- expandirLargura(NodosPercorrer, Caminho, NovosCaminhos),
                                                                    concatena(Caminhos, NovosCaminhos, Caminhos1),
                                                                    breadthFirst(NodosPercorrer, Final, Caminhos1, Solucao).

expandirLargura(NodosPercorrer, [No|Caminho],NovosCaminhos) :- findall([NovoNo,No|Caminho],(aresta(No,NovoNo),
                                                                                            pertence(NovoNo,NodosPercorrer),
                                                                                            \+ pertence(NovoNo,[No|Caminho])),
                                                               NovosCaminhos).
```

Figura 5: Algoritmo de pesquisa em largura - prolog

Procura limitada em profundidade

A procura iterativa limitada em profundidade, pretende combinar as vantagens das duas estratégias anteriores – em profundidade e em largura. Este algoritmo opera de forma semelhante ao da procura em profundidade, contudo a altura da árvore de procura é limitada, de modo a que o algoritmo não esteja a examinar um ramo indefinidamente (Figura 6).

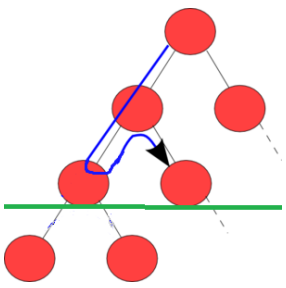

Figura 6: Algoritmo de pesquisa em profundidade limitada

A representação em *prolog* deste algoritmo para a resolução do problema apresentado é a seguinte:

```
profundidade_limitada(NodosPercorrer, LimiteDeProf, Final, [(E,_)|Es], Historia, [E]):- E=:=Final.
profundidade_limitada(NodosPercorrer, LimiteDeProf, Final, [(E,LimiteDeProf)|Es],Historia,Solucao):- !, profundidade_limitada(NodosPercorrer, LimiteDeProf, Final, Es,Historia,Solucao).
profundidade_limitada(NodosPercorrer, LimiteDeProf, Final, [(E,ProfAtual)|Es], Historia, [E|Solucao]):- findall(F,(aresta(E,F), pertence(F,NodosPercorrer)),Filhos),
                                                                                                        ProxProf is ProfAtual + 1,
                                                                                                        expandeComProfundidadeLimitada(Filhos,ProxProf, Historia,FilhosRotulados),
                                                                                                        concatena(FilhosRotulados,Es,ProxEs),
                                                                                                        profundidade_limitada(NodosPercorrer, LimiteDeProf, Final, ProxEs,[E|Historia],Solucao).

expandeComProfundidadeLimitada([E|Es],P,Historia,ER):- membro(E,Historia), !,
                                                       expandeComProfundidadeLimitada(Es,P,Historia,ER).
expandeComProfundidadeLimitada([E|Es],P,Historia,[(E,P)|ER]):- expandeComProfundidadeLimitada(Es,P,Historia,ER).
expandeComProfundidadeLimitada([],_,_,[]):-!.


resolve_pp_com_profundidade_limitada(NodosPercorrer, LimiteDeProfundidade, EstadoInicial, Final, Solucao):-
          profundidade_limitada(NodosPercorrer, LimiteDeProfundidade, Final, [(EstadoInicial,0)],[],Solucao).
```

Figura 7: Algoritmo de pesquisa em profundidade limitada - prolog

Procura informada

A procura informada baseia-se na definição do problema, de modo a reconhecer uma variável que possa fornecer informação para a sua resolução.

A Pesquisa A* e a Pesquisa Gulosa representam este tipo de procura.

Pesquisa A*

Este algoritmo combina a procura em largura com heurísticas, o que lhe permite encontrar a melhor solução. Contudo, utiliza muita memória, uma vez que regista todos os nodos da pesquisa, escolhendo sempre o que produz uma menor distância ao nodo de destino.

O algoritmo pode ser representado através do seguinte código em *prolog*:

```
resolve_estrela(NodosPercorrer, Nodo, Final, Caminho/Custo) :- dist(Nodo,Final,Estima),
                                                               estrela(NodosPercorrer, Final, [[Nodo]/0/Estima], InvCaminho/Custo/_),
                                                               inverso(InvCaminho,Caminho).

estrela(NodosPercorrer, Final, Caminhos, Caminho) :- obtem_melhor_e(Caminhos,Caminho), Caminho = [Nodo|_]/_/_, Nodo=:=Final.
estrela(NodosPercorrer, Final, Caminhos, SolucaoCaminho) :- obtem_melhor_e(Caminhos, MelhorCaminho),
                                                            seleciona(MelhorCaminho,Caminhos,OutrosCaminhos),
                                                            expande_estrela(NodosPercorrer, Final, MelhorCaminho, ExpCaminhos),
                                                            append(OutrosCaminhos,ExpCaminhos,NovoCaminho),
                                                            estrela(NodosPercorrer, Final, NovoCaminho, SolucaoCaminho).

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

expande_estrela(NodosPercorrer, Final, Caminho, ExpCaminhos) :- findall(NovoCaminho,adjacente3(NodosPercorrer, Final, Caminho, NovoCaminho),ExpCaminhos).
```

Figura 8: Algoritmo de pesquisa A - prolog*

O processo recursivo escolhe o melhor caminho entre os já obtidos, expande esse caminho e adiciona-o aos novos caminhos, através desta ordem específica. Por fim, chama recursivamente o termo, utilizando os novos caminhos obtidos. O melhor caminho é escolhido através de cálculo da heurística do custo do percurso, isto é, através da soma da distância em linha reta do nodo atual até ao final com o custo do caminho já efetuado, como se pode observar na figura seguinte:

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

*Figura 9: Escolha do melhor caminho no algoritmo A**

Expandir o caminho consiste em encontrar todos os nodos adjacentes ao caminho já encontrado, de modo a que as heurísticas possam ser calculadas.

Pesquisa Gulosa

O algoritmo da Pesquisa Gulosa assemelha-se à estratégia anterior, contudo no cálculo do melhor caminho apenas considera a distância do nodo atual ao nodo final, guiando-se pela *otimalidade* local. A única diferença relativamente ao algoritmo Pesquisa A* está no cálculo do melhor caminho, que se apresenta da seguinte forma:

```
resolve_gulosa(NodosPercorrer, Nodo, Final, Caminho/Custo) :- dist(Nodo,Final,Estima),
                                                              agulosa(NodosPercorrer,Final,[[Nodo]/0/Estima], InvCaminho/Custo/_),
                                                              inverso(InvCaminho,Caminho).

agulosa(NodosPercorrer, Final, Caminhos, Caminho) :- obtem_melhor_g(Caminhos,Caminho), Caminho = [Nodo|_]/_/_, Nodo=:=Final.
agulosa(NodosPercorrer, Final, Caminhos, SolucaoCaminho) :- obtem_melhor_g(Caminhos, MelhorCaminho),
                                                            seleciona(MelhorCaminho,Caminhos,OutrosCaminhos),
                                                            expande_gulosa(NodosPercorrer, Final, MelhorCaminho, ExpCaminhos),
                                                            append(OutrosCaminhos,ExpCaminhos,NovoCaminho),
                                                            agulosa(NodosPercorrer, Final, NovoCaminho,SolucaoCaminho).

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

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

Figura 10: Algoritmo de pesquisa gulosa - prolog

Este algoritmo apesar de não encontrar sempre a solução ótima, permite identificar, num número inferior de passos, uma solução próxima da ideal.

Resultados

Após a apresentação das estratégias anteriores para resolução do problema formulado, nesta seção, pretende-se compará-las de modo a perceber qual o algoritmo mais eficiente, por permitir chegar a uma solução no mais curto período de tempo possível e com a menor utilização de memória. Assim, realizaram-se os seguintes testes:

Teste 1: Da aresta 15805 até a aresta 15812;

Teste 2: Da aresta 15805 até a aresta 15827;

Teste 3: Da aresta 15805 até a aresta 21949.

Uma vez que, á medida que o identificador do nodo aumenta, o nível de profundidade da árvore dos caminhos atinge níveis mais profundos, é mais complicado para os algoritmos baseados na pesquisa em largura realizarem uma boa performance. Desta forma, além de se esperar um melhor desempenho dos algoritmos baseados em profundidade, classifica-se o teste 1 como o teste mais fácil e o teste 3 como o mais difícil.

Os resultados obtidos encontram-se compilados na seguinte tabela:

Tabela 1: Comparação de resultados entre os algoritmos de pesquisa

	Estratégia		Tempo (Segundos)	Espaço (MB)	Encontrou a melhor solução?
Não informada	DFS	Teste 1	0	0	Não*
		Teste 2	0	0	Não*
		Teste 3	0	0	Não*
	BFS	Teste 1	0	0	Não*
		Teste 2	1	1.2	Não*
		Teste 3	Imp*	300+	Não*
	DFS - Limitada	Teste 1	0	0	Não*
		Teste 2	0	0	Não*
		Teste 3	0	0	Não*
Informada	A*	Teste 1	0	0	Sim
		Teste 2	2	300	Sim
		Teste 3	Imp	1800+	Não
	Gulosa	Teste 1	0	0	Não
		Teste 2	2	250	Não
		Teste 3	Imp	1800+	Não

Ao observar a tabela, verifica-se que o tempo de execução é instantâneo nos algoritmos de pesquisa em profundidade que possuem o melhor caso para nodos finais mais profundos. Contudo, para algoritmos baseados em largura, a *performance* nestas características é muito inferior, sendo, em alguns casos, impossível obter uma solução devido ao tempo necessário para obter um resultado exceder os 15 minutos. Nos algoritmos de largura também é possível verificar o grande consumo de memória atingindo mais de 300 megabytes para realizar parte do cálculo da solução.

Como a solução explora os nodos mais profundos, a *performance* dos algoritmos de pesquisa em profundidade com ou sem limitações é idêntica, contudo, é necessário utilizar uma profundidade máxima que permita encontrar a solução, caso contrário é impossível identificar um caminho válido. Como já foi referido, o algoritmo de pesquisa em profundidade limitada é mais eficiente sempre que a solução não se encontra a explorar os nodos mais profundados do mesmo ramo.

Os algoritmos de pesquisa informada, baseiam-se na heurística para decidir o ramo a percorrer, desta forma permitem encontrar uma solução com melhores custos. Contudo, necessitam de mais recursos para executar, o que torna impossível encontrar um resultado para o teste 3 (atinge-se o máximo da *stack*). Para o Teste 2 identifica-se a degradação de performance relativamente ao algoritmo DFS. Porém, é de notar que o custo dos caminhos obtidos com estes tipos de algoritmos (Gulosa: 1685.45 e Estrela: 1672.84) é inferior ao custo obtido pelo algoritmo de DFS (2001.93), sendo por sua vez o custo obtido pela pesquisa informada A* o mais baixo, o que evidencia a diferença entre a escolha da *otimalidade* global ou local.

Relativamente à melhor solução para o caminho a realizar, tal como já foi referido e evidenciado nos resultados acima referidos, o algoritmo A* permite obter a melhor solução, pelo que para o Teste 1 e Teste 2 encontrou-se desta forma a solução ótima. Para os outros algoritmos realizou-se uma procura por todos os caminhos, na tentativa de encontrar a melhor solução. Contudo, nenhum dos algoritmos permitiu obter resultados, ou porque atingia-se o máximo da *stack*, ou simplesmente porque não se obtinha resultados num período máximo de 15 minutos (assinalados na tabela com Não*).

Assim, conclui-se que para o grafo considerado o algoritmo de pesquisa em profundidade apresenta uma melhor performance, consumindo menos memória e obtendo resultados mais

rapidamente, embora os resultados obtidos não sejam ótimos. Desta forma os predicados, desenvolvidos e apresentados de seguida, tirarão proveito da pesquisa em profundidade.

Geração de circuitos

Tal como apresentado na introdução deste relatório, este trabalho tem como objetivo encontrar e comparar circuitos. Um circuito é um ciclo que começa na garagem em que o camião de recolha se encontra estacionado e acaba novamente na garagem, após passar pelos pontos de recolha e pelo depósito. Contudo, uma vez que apenas existe um único ponto de depósito de lixo, pode-se generalizar o circuito em um caminho que começa na garagem e termina no depósito. Após se encontrar no depósito é realizado o caminho de volta à garagem através de uma ligação direta.

Com base nos algoritmos já referidos, criou-se os termos que serão apresentados e especificados nas secções seguintes.

Gerar os circuitos de recolha que cubram um determinado território

Para gerar um circuito é necessário determinar o nodo inicial (garagem) e o nodo final (depósito). Posteriormente deve ser calculado, recorrendo a um dos algoritmos anteriormente apresentados, o caminho entre os dois nodos, acrescentando no final o caminho entre o depósito e a garagem.

Assim, o predicado que representa esta funcionalidade é apresentado na figura seguinte.

```
circuito(Regiao,C,K1) :- nodo(Garagem,_,_,Regiao,['Garagem'],_),
                         nodo(Deposito,_,_,Regiao,['Deposito'],_),
                         findall(N,nodo(N,_,_,Regiao,_,_),NA),
                         resolve_pp_k(NA,Garagem,Deposito,C,K), escrever(C), writeln(Garagem),
                         dist(Garagem,Deposito,K2), K1 is K+K2.

circuito(Regiao,Tipo,C,K1) :- nodo(Garagem,_,_,Regiao,['Garagem'],_),
                              nodo(Deposito,_,_,Regiao,['Deposito'],_),
                              findall(N,(nodo(N,_,_,Regiao,AUX2,_), pertence(Tipo,AUX2)),NA),
                              resolve_pp_k([Garagem,Deposito|NA],Garagem,Deposito,C,K), escrever(C), writeln(Garagem),
                              dist(Garagem,Deposito,K2), K1 is K+K2.

circuito(C,K1) :- nodo(Garagem,_,_,_,['Garagem'],_),
                  nodo(Deposito,_,_,_,['Deposito'],_),
                  findall(N,nodo(N,_,_,_,_,_),NA),
                  resolve_pp_k(NA,Garagem,Deposito,C,K), escrever(C), writeln(Garagem),
                  dist(Garagem,Deposito,K2), K1 is K+K2.
```

Figura 11: Predicado circuito

Ao observar o predicado percebe-se que é possível obter um circuito considerando todos os nodos do mapa (pesquisar um circuito que não considere a região ou o tipo de lixo) ou restringindo o circuito a certos tipos de lixo ou certas regiões.

Uma vez que os algoritmos de pesquisa desenvolvidos recebem uma lista com os nodos a considerar no deslocamento, utiliza-se a mesma para eliminar a passagem por certos nodos. Assim, sempre que seja necessário restringir o cálculo do caminho a certas regiões ou tipos de contentor, elimina-se da lista os vértices que não cumpram as restrições. Esta abordagem é utilizada em todos os predicados desenvolvidos e pode ser evidenciada na figura 11.

Identificar os circuitos com mais pontos de recolha

Para gerar o circuito que passa por mais pontos de recolha deve-se gerar todos os circuitos possíveis e posteriormente escolher o circuito com mais pontos de recolha, i.e., escolher o circuito que passa por mais nodos até ao depósito de lixo.

Assim o predicado que verifica o circuito com mais pontos de recolha é representado da seguinte forma:

```
maisPontos([],(A,_,K),(A,K)).
maisPontos([(L,K)|T],(_,PT,_),R) :- comprimento(L,P), P > PT, !, maisPontos(T,(L,P,K),R).
maisPontos([(L,K)|T],(AT,PT,KT),R) :- maisPontos(T,(AT,PT,KT),R).
```

Figura 12:Predicado que devolve o circuito com mais pontos

Por fim, o predicado que calcula o circuito com mais pontos de recolha é o seguinte:

```
circuito_maisPontos(Regiao,Tipo,CP,KP) :- nodo(Garagem,_,_,Regiao,['Garagem'],_),
                                          nodo(Deposito,_,_,Regiao,['Deposito'],_),
                                          findall(N,(nodo(N,_,_,Regiao,AUX2,_), pertence(Tipo,AUX2)),NA),
                                          findall((C,K),resolve_pp_k([Garagem,Deposito|NA],Garagem,Deposito,C,K),L),
                                          maisPontos(L,([],0,0),(CP,K1)),
                                          escrever(CP), writeln(Garagem), dist(Garagem,Deposito,K2), KP is K1+K2.

circuito_maisPontos(Regiao,CP,KP) :- nodo(Garagem,_,_,Regiao,['Garagem'],_),
                                     nodo(Deposito,_,_,Regiao,['Deposito'],_),
                                     findall(N,nodo(N,_,_,Regiao,_,_),NA),
                                     findall((C,K),resolve_pp_k([Garagem,Deposito|NA],Garagem,Deposito,C,K),L),
                                     maisPontos(L,([],0,0),(CP,K1)),
                                     escrever(CP), writeln(Garagem), dist(Garagem,Deposito,K2), KP is K1+K2.

circuito_maisPontos(CP,KP) :- nodo(Garagem,_,_,_,['Garagem'],_),
                              nodo(Deposito,_,_,_,['Deposito'],_),
                              findall(N,nodo(N,_,_,_,_,_),NA),
                              findall((C,K),resolve_pp_k([Garagem,Deposito|NA],Garagem,Deposito,C,K),L),
                              maisPontos(L,([],0,0),(CP,K1)),
                              escrever(CP), writeln(Garagem), dist(Garagem,Deposito,K2), KP is K1+K2.
```

Figura 13: Predicado que gera o circuito que passa por mais pontos de recolha

Uma vez mais é possível restringir o circuito a certas regiões ou a certos tipos de lixo.

Comparar circuitos de recolha tendo em conta a distância

De modo a comparar dois circuitos é necessário perceber o predicado *distancia* (Figura 14) que tem sido usado nos algoritmos já referidos.

```
distancia(Lat1, Long1, Lat2, Long2, DistanceCost) :- ValPi is pi,
                                                     Fi1 is Lat1 * (ValPi/180),
                                                     Fi2 is Lat2 * (ValPi/180),
                                                     DeltaFi is (Lat2-Lat1) * (ValPi/180),
                                                     DeltaLambda is (Long2-Long1) * (ValPi/180),
                                                     A1 is sin(DeltaFi/2) * sin(DeltaFi/2),
                                                     A2 is cos(Fi1) * cos(Fi2),
                                                     A3 is sin(DeltaLambda/2) * sin(DeltaLambda/2),
                                                     ASum is A1 + A2 * A3,
                                                     C1 is sqrt(ASum),
                                                     C2 is sqrt(1.0 - ASum),
                                                     C is 2 * atan2(C1, C2),
                                                     Dist is 6.371*1000 * C,
                                                     DistanceCost is Dist*1000. % em metros

dist(Nodo,Final,Euristica) :- nodo(Nodo,Lat1,Lon1,_,_,_), nodo(Final,Lat2,Lon2,_,_,_), distancia(Lat1,Lon1,Lat2,Lon2,Euristica).
```

Figura 14:Predicado que calcula a distância entre dois pontos de recolha

Ao observar o predicado *distancia* percebe-se que ao receber duas coordenadas GPS (Latitude e Longitude) obtém-se a distância em linha reta de um ponto ao outro (medida em metros). Contudo, para realizar este cálculo, é necessário considerar a curvatura da terra e a utilização de fórmulas geométricas (considerou-se que o raio da Terra é aproximadamente 6371.0 km). Apesar da fórmula apresentada não ter uma precisão exata, é suficiente para o caso de estudo.

Recorrendo ao predicado *distancia* criou-se o seguinte predicado, que ao receber um circuito, calcula a distância total percorrida ao realizar esse percurso.

```
distanciaPercorridaCircuito([],0).
distanciaPercorridaCircuito([H|T],R) :- nao(empty(T)), distanciaPercorridaCircuito(T,N1), head(T,T1), dist(H,T1,Dist), R is N1+Dist.
distanciaPercorridaCircuito([H|T],R) :- empty(T), R is 0.
```

Figura 15:Predicado que calcula a distância de um circuito

Por fim, verifica-se qual circuito apresenta uma menor distância percorrida, tal como representado na figura 16.

```
comparaCircuitos(C1,C2,C1) :- distanciaPercorridaCircuito(C1,N1), distanciaPercorridaCircuito(C2,N2), N1 < N2, !.
comparaCircuitos(C1,C2,C2).
```

Figura 16:Predicado que compara dois circuitos

Com a realização deste predicado, assume-se que os circuitos recebidos são válidos, não sendo realizada a verificação da sua validade. Caso se pretenda verificar primeiramente a validade dos circuitos dever-se-ia gerar, através de um '*findall*', todos os circuitos válidos e posteriormente comparar ambos os circuitos recebidos com os obtidos.

Encontrar o circuito mais rápido (usando o critério da distância)

Com o objetivo de encontrar o circuito mais rápido, admite-se que a velocidade está diretamente ligada à distância percorrida, sendo que quanto menor for a distância (em metros) a percorrer, mais rápido irá ser o circuito. Assim, para encontrar o circuito mais rápido realiza-se a procura por todos os circuitos possíveis e depois da lista obtida escolhe-se o que percorrer menos metros, tal como se observa na figura seguinte.

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

circuito_maisRapido(Regiao,Tipo,CP,KP) :- nodo(Garagem,_,_,Regiao,['Garagem'],_),
                                          nodo(Deposito,_,_,Regiao,['Deposito'],_),
                                          findall(N,(nodo(N,_,_,Regiao,AUX2,_), pertence(Tipo,AUX2)),NA),
                                          findall((C,K),resolve_pp_k([Garagem,Deposito|NA],Garagem,Deposito,C,K),L),
                                          minimoCusto(L,(CP,K1)),
                                          escrever(CP), writeln(Garagem), dist(Garagem,Deposito,K2), KP is K1+K2.

circuito_maisRapido(Regiao,CP,KP) :- nodo(Garagem,_,_,Regiao,['Garagem'],_),
                                     nodo(Deposito,_,_,Regiao,['Deposito'],_),
                                     findall(N,nodo(N,_,_,Regiao,_,_),NA),
                                     findall((C,K),resolve_pp_k(NA,Garagem,Deposito,C,K),L),
                                     minimoCusto(L,(CP,K1)),
                                     escrever(CP), writeln(Garagem), dist(Garagem,Deposito,K2), KP is K1+K2.

circuito_maisRapido(CP,KP) :- nodo(Garagem,_,_,_,['Garagem'],_),
                              nodo(Deposito,_,_,_,['Deposito'],_),
                              findall(N,nodo(N,_,_,_,_,_),NA),
                              findall((C,K),resolve_pp_k(NA,Garagem,Deposito,C,K),L),
                              minimoCusto(L,(CP,K1)),
                              escrever(CP), writeln(Garagem), dist(Garagem,Deposito,K2), KP is K1+K2.
```

Figura 17:Predicado que gera o circuito mais com menos distância percorrida

Contudo, como já foi referido, o algoritmo A* permite obter o caminho ótimo, sendo por isso possível obter o circuito mais rápido da seguinte forma:

```
circuito_maisRapido2(CP,KP) :- nodo(Garagem,_,_,_,['Garagem'],_),
                               nodo(Deposito,_,_,_,['Deposito'],_),
                               findall(N,nodo(N,_,_,_,_,_),NA),
                               resolve_estrela(NA,Garagem,Deposito,CP/K1),
                               escrever(CP), writeln(Garagem), dist(Garagem,Deposito,K2), KP is K1+K2.
```

*Figura 18:Predicado que gera o circuito com menos distância percorrida – algoritmo A**

Encontrar o circuito mais eficiente

Para encontrar o caminho mais eficiente considera-se a seguinte mediada de eficiência:

- Circuito onde é possível recolher mais resíduos dos contentores.

Posteriormente, basta gerar todos os circuitos possíveis e verificar o que melhor se adequa com a medida apresentada. Este processo pode ser verificado com o seguinte excerto de código:

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

circuito_maisEficiente(Regiao,Tipo,CP,KP) :- nodo(Garagem,_,_,Regiao,'Garagem',_),
                                             nodo(Deposito,_,_,Regiao,'Deposito',_),
                                             findall(N,nodo(N,_,_,Regiao,Tipo,_),NA),
                                             findall((C,K),resolve_pp_1([Garagem,Deposito|NA],Garagem,Deposito,C,K),L),
                                             maxCusto(L,(CP,KP)),
                                             escrever(CP), writeln(Garagem).

circuito_maisEficiente(Regiao,CP,KP) :- nodo(Garagem,_,_,Regiao,'Garagem',_),
                                        nodo(Deposito,_,_,Regiao,'Deposito',_),
                                        findall(N,nodo(N,_,_,Regiao,_,_),NA),
                                        findall((C,K),resolve_pp_1(NA,Garagem,Deposito,C,K),L),
                                        maxCusto(L,(CP,KP)),
                                        escrever(CP), writeln(Garagem).

circuito_maisEficiente(CP,KP) :- nodo(Garagem,_,_,_,'Garagem',_),
                                 nodo(Deposito,_,_,_,'Deposito',_),
                                 findall(N,nodo(N,_,_,_,_,_),NA),
                                 findall((C,K),resolve_pp_1(NA,Garagem,Deposito,C,K),L),
                                 maxCusto(L,(CP,KP)),
                                 escrever(CP), writeln(Garagem).
```

Figura 19: Predicado que gera o circuito mais eficiente

Exemplos de aplicação dos termos

Nesta secção serão apresentados exemplos de utilização dos predicados já expostos, bem como exemplos de utilização dos diferentes algoritmos de procura referidos neste relatório. Contudo, de modo a obter-se resultados de forma rápida com todos os algoritmos de pesquisa, reduziu-se a base de conhecimento para o seguinte grafo não orientado (Figura 20 e Anexo I – representação em *prolog*), com 11 nodos e 38 arestas (a aresta representada a roxo representa a ligação direta entre o depósito – representado a vermelho – e a garagem – representada a verde – percorrida no final do percurso).

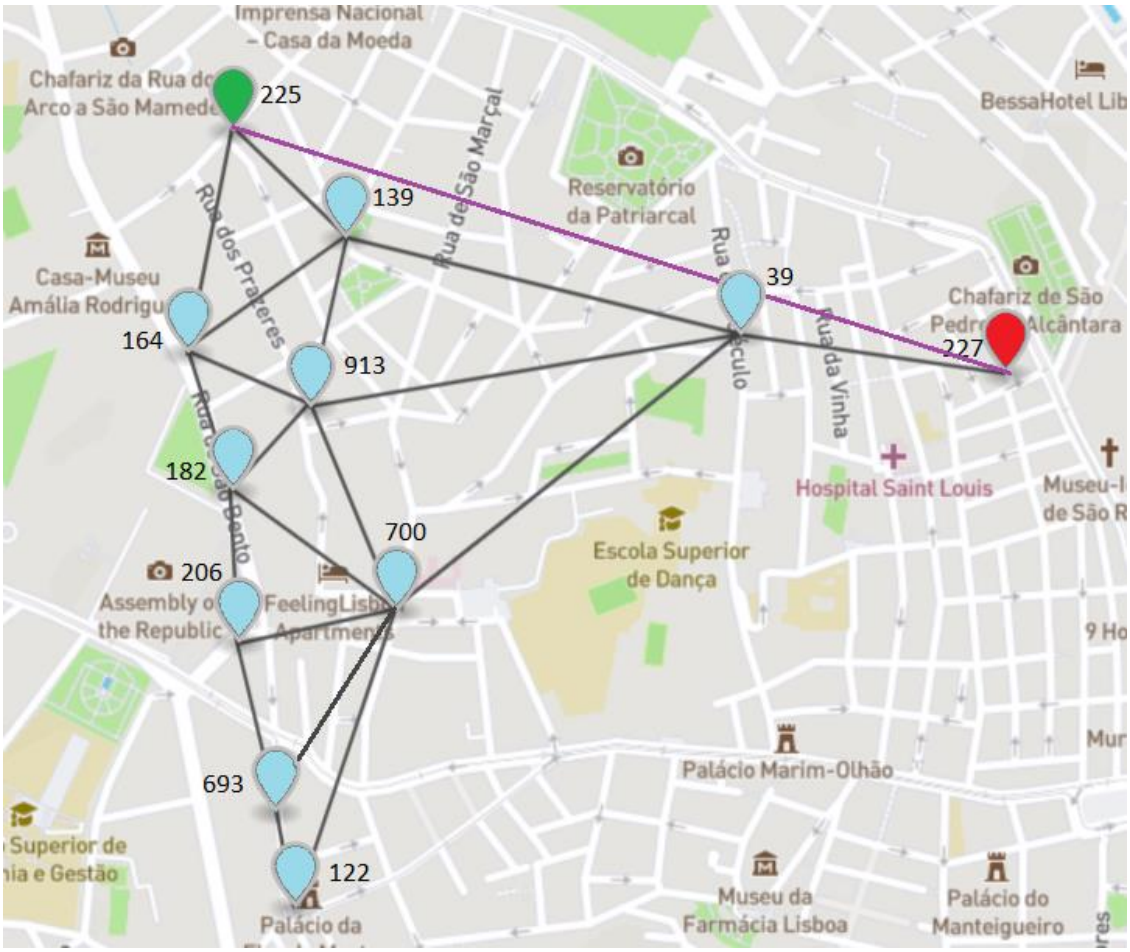

Figura 20: Grafo considerado nos exemplos apresentados

Esta redução de tamanho também possibilita, com o auxílio da figura 20, verificar e validar a devida execução dos algoritmos, que serão apresentados nas secções seguintes.

Teste da pesquisa (não informada) em profundidade limitada

O predicado *resolve_pp_limitada* permite encontrar, através de um algoritmo de pesquisa em profundidade, um caminho válido num determinado grafo, contudo, ao contrário dos outros predicados o nível de profundidade atingido é limitado por um valor definido.

A próxima figura apresenta um exemplo da utilização deste algoritmo.

```
?- resolve_pp_limitada(225,227,3,C), escrever(C).
225
164
139
913
182
139
39
227
C = [225, 164, 139, 913, 182, 139, 39, 227] .
```

Figura 21: Exemplo do algoritmo de pesquisa em profundidade limitada

Teste da pesquisa (não informada) em profundidade sem custos

Para utilizar o predicado *profundidade* (predicado que aplica a pesquisa de um caminho em profundidade, sem considerar o cálculo dos custos) é necessário introduzir o ponto de partida e o ponto de chegada. Um exemplo da aplicação deste termo é:

```
?- profundidade(225,227,C), escrever(C).
225
164
139
913
182
206
693
122
700
39
227
C = [225, 164, 139, 913, 182, 206, 693, 122, 700|...] ;
225
164
139
913
182
206
693
700
39
227
C = [225, 164, 139, 913, 182, 206, 693, 700, 39|...] ;
225
164
139
913
182
206
700
39
227
C = [225, 164, 139, 913, 182, 206, 700, 39, 227] ;
225
164
139
913
182
700
39
227
C = [225, 164, 139, 913, 182, 700, 39, 227] .
```

Figura 22:Exemplo do algoritmo de pesquisa em profundidade

Ao observar a figura 21 percebe-se que todos os caminhos obtidos são válidos, o que prova o correto funcionamento do algoritmo. Contudo, também é possível verificar que após realizar o *backtracking*, obteve-se um caminho diferente, que continua a explorar os mesmos ramos iniciais (225-164-…) pois explora as alternativas no nível de profundidade mais baixo primeiro, apenas mudando a primeira ligação escolhida (225-164) após explorar todos os caminhos alternativos existentes com essa mesma ligação.

Teste da pesquisa (não informada) em profundidade com custos

O predicado *profundidade_custo* permite encontrar, tal como o predicado anterior um caminho entre um nodo inicial e um nodo final, baseando-se no algoritmo de procura em profundidade. Contudo, este predicado além de apresentar um caminho válido, apresenta também a distância percorrida ao realizar o caminho obtido. Um exemplo da aplicação deste termo é dado pela figura seguinte:

```
?- profundidade_custo(225,227,C,K), escrever(C).
225
164
139
913
182
206
693
122
700
39
227
C = [225, 164, 139, 913, 182, 206, 693, 122, 700|...],
K = 2060.6570565971124 ;
225
164
139
913
182
206
693
700
39
227
C = [225, 164, 139, 913, 182, 206, 693, 700, 39|...],
K = 1891.1010454127797 ;
225
164
139
913
182
206
700
39
227
C = [225, 164, 139, 913, 182, 206, 700, 39, 227],
K = 1654.3934124592945 ;
225
164
139
913
182
700
39
227
C = [225, 164, 139, 913, 182, 700, 39, 227],
K = 1549.1520735445595 ;
225
164
139
913
700
39
227
C = [225, 164, 139, 913, 700, 39, 227],
K = 1457.6179052177515 ,
```

Figura 23: Exemplo do algoritmo de pesquisa em profundidade com custos

Teste da pesquisa (não informada) em largura

O predicado *resolveBF* permite encontrar um caminho válido no grafo, contudo, ao contrário dos predicados anteriores o método utilizado é a pesquisa em largura, que apresenta as características já referidas. A figura seguinte apresenta um exemplo da utilização deste algoritmo partindo de um nodo inicial (225) até um nodo final (227).

```
?- resolveBF(225,227,C), escrever(C).
225
139
39
227
C = [225, 139, 39, 227].
```

Figura 24: Exemplo do algoritmo de pesquisa em largura

Teste da pesquisa (informada) A*

Para realizar a pesquisa de um caminho válido entre o nodo inicial e um nodo final, utilizando um algoritmo de procura informada, como por exemplo o A*, deve-se recorrer ao predicado *estrela*. Um exemplo da aplicação deste algoritmo é dado pela figura seguinte.

```
?- estrela(225,227,C/K), escrever(C).
225
139
39
227
C = [225, 139, 39, 227],
K = 793.8400497067945 ,
```

*Figura 25: Exemplo do algoritmo de pesquisa A**

Ao observar a figura acima apresentada, juntamente com o grafo (Figura 20) que representa a topologia em estudo, percebe-se que o caminho obtido ao utilizar o algoritmo estrela é o caminho ótimo desde o nodo 225 até ao nodo 227. A capacidade de obter o caminho ótimo entre dois nodos de um grafo é característico deste algoritmo.

Assim, se de todos os caminhos obtidos, apenas se considerar os caminhos que passem por todos os nodos do grafo, obtém-se como primeiro caminho o melhor percurso. Um exemplo da aplicação deste predicado é dado pela seguinte figura:

```
?- estrela_tudo(225,227,C/K), escrever(C).
225
139
913
164
182
206
693
122
700
39
227
C = [225, 139, 913, 164, 182, 206, 693, 122, 700|...],
K = 1964.6213252531395 ,
```

Figura 26: Exemplo do algoritmo de pesquisa A com todos os pontos*

Teste da pesquisa (informada) gulosa

Para além do algoritmo A*, na secção de algoritmos de pesquisa informada apresentados neste relatório, existe o algoritmo de procura gulosa. Como já referido este algoritmo apenas procura a *otimalidade* local, não obtendo por isso sempre o melhor caminho. Assim ao realizar a procura por um caminho qualquer entre o nodo 225 e 227 (tal como apresentado na figura seguinte) obtém-se o caminho ótimo.

```
?- gulosa(225,227,C/K), escrever(C).
225
139
39
227
C = [225, 139, 39, 227],
K = 793.8400497067945 ,
```

Figura 27: Exemplo do algoritmo de pesquisa gulosa

Contudo, ao realizar a procura por um caminho desde o nodo 225 até ao nodo 227 que passe por todos os nodos, obtém-se o seguinte resultado.

```
?- gulosa_todos(225,227,C/K), escrever(C).
225
164
139
913
182
206
693
122
700
39
227
C = [225, 164, 139, 913, 182, 206, 693, 122, 700|...],
K = 2060.6570565971124 .
```

Figura 28: Exemplo do algoritmo de pesquisa gulosa que passa por todos os pontos

Ao comparar este resultado (com uma distância percorrida de 2.06 Km) com o resultado obtido pelo algoritmo A* (com uma distância percorrida de 1.964 Km) percebe-se que apesar de estar próximo, não é a solução ótima. Porém, ao comparar este resultado com o primeiro valor (que passe por todos os nodos) obtido pelo algoritmo de procura em profundidade (2083.8 metros) prova-se que este algoritmo obtém melhores resultados.

Teste do predicado ***distancia***

Para determinar a precisão do predicado *distancia*, recorreu-se a uma calculadora de distâncias entre coordenadas GPS, disponível online. Assim, realizaram-se alguns testes com certas coordenadas e percebeu-se que a precisão, apesar de não ser exata era bastante próxima. Um exemplo dos testes realizados está presente na seguinte figura.

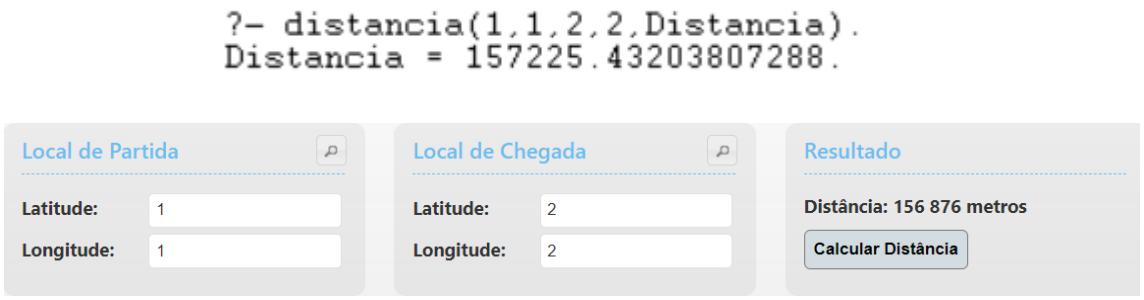

Figura 29: Exemplo do predicado distância

Da figura anterior é possível verificar que existe uma imprecisão de aproximadamente 349 metros nos 157 quilómetros calculados.

Teste do predicado ***circuito***

Com a realização do predicado *circuito* pretende-se encontrar um caminho desde a garagem (neste grafo considerou-se a garagem no nodo 225) até ao depósito de lixo (nodo 227), sendo que uma

vez no depósito do lixo o camião segue a rota direta de volta para a garagem (rota representada a roxo na figura 20). Assim, um exemplo da aplicação deste predicado é dado pela figura seguinte.

```
?- circuito(C,K).
225
164
139
913
700
39
227
225
C = [225, 164, 139, 913, 700, 39, 227],
K = 2232.2930243797127 .
```

Figura 30: Exemplo do predicado que calcula um circuito

Para além de obter um circuito de forma indiferenciada, é possível realizar a seleção da região ou tipo de lixo dos contentores. Um exemplo de um circuito para a região de Misericórdia em Lisboa é o seguinte.

```
?- circuito('Misericordia',C,K).
225
164
139
913
700
39
227
225
C = [225, 164, 139, 913, 700, 39, 227],
K = 2232.2930243797127
```

Figura 31: Exemplo do predicado que calcula um circuito na região da Misericórdia

Teste do predicado ***circuito_maisPontos***

Para obter o circuito que passa por mais pontos de recolha deve-se utilizar o predicado *circuito_maisPontos*, obtendo o seguinte resultado para os contentores do lixo da área da Misericórdia (é possível realizar a pesquisa discriminando apenas o filtro da região ou ambos os filtros aplicados no exemplo).

```
?- circuito_maisPontos('Misericordia','Lixos',C,K).
225
164
139
913
182
206
693
122
700
39
227
225
C = [225, 164, 139, 913, 182, 206, 693, 122, 700|...],
K = 2835.3321757590734.
```

Figura 32: Exemplo do predicado que calcula o circuito com mais pontos de recolha

Ao observar, o circuito obtido, percebe-se que a solução é uma das possíveis, uma vez que passa por todos os nodos do grafo (no exemplo considerado nos testes todos os nodos são contentores do lixo pertencentes à região da Misericórdia).

Teste do predicado ***comparaCircuitos***

Para comparar os dois circuitos válidos, deve-se recorrer ao predicado *compararCircuitos*, obtendo no final o circuito que implica percorrer menos metros.

Para testar este predicado recorreu-se a dois circuitos, que apesar de inválidos, permitem evidenciar o bom funcionamento do predicado. Um exemplo da utilização deste predicado é:

```
?- comparaCircuitos([225,227],[225,122,227],C).
C = [225, 227].

?- comparaCircuitos([225,122,227],[225,227],C).
C = [225, 227].
```

Figura 33: Exemplo do predicado que compara circuitos

Ao analisar o grafo presente na Figura 33, percebe-se que dos dois caminhos introduzidos, o obtido como resultado é o que implica percorrer menos metros.

Teste do predicado ***circuito_maisRapido***

Como referido de modo a obter o percurso mais rápido (menos distância percorrida), deve-se utilizar o predicado *circuito_maisRapido*. Assim, para o grafo utilizado obtém-se o seguinte percurso:

```
|    circuito_maisRapido(C,K).
225
139
39
227
225
C = [225, 139, 39, 227],
K = 1568.5151688687556 .
```

Figura 34: Exemplo do predicado que calcula o circuito mais rápido

Observando este circuito percebe-se que é o mesmo obtido pelo algoritmo de procura informada A* (Figura 25), tendo apenas no final a aresta direta do depósito para a garagem, que implica percorrer mais 774.6 metros.

Teste do predicado ***circuito_maisEficiente***

Como a medida de eficiência utilizada para verificar o circuito mais eficiente é a quantidade de lixo, em litros, recolhida no final do circuito, é fácil perceber que o melhor percurso, uma vez que todos os nodos pertencem à mesma região e tem o mesmo tipo de lixo, irá conter todos os nodos do grafo. Assim obtém-se o seguinte exemplo:

```
?- circuito_maisEficiente(C,K).
225
139
913
164
182
206
693
122
700
39
227
225
C = [225, 139, 913, 164, 182, 206, 693, 122, 700|...],
K = 3050 .
```

Figura 35: Exemplo do predicado que calcula o circuito mais eficiente

Conclusão

Ao longo deste trabalho descreveram-se e implementaram-se estratégias de procuras de caminhos entre dois pontos de um grafo. Assumindo um sistema fortemente ligado através de um grafo, aplicado a um caso real (pontos de recolha de lixo), foi possível implementar estratégias de procura não informada (pesquisa em largura ou profundidade), bem como estratégias de procura informada (A* e gulosa).

Após a implementação de todas as estratégias, foi possível realizar a comparação entre todas elas, do qual se evidenciou que os algoritmos baseados na pesquisa em largura e os algoritmos de pesquisa informada consomem mais memória do que os algoritmos de pesquisa em profundidade. Contudo, com a utilização dos algoritmos de pesquisa informada é possível obter uma solução ótima (algoritmo A*) ou próximo da ótima (algoritmo gulosa).

Posteriormente, aplicou-se os algoritmos de pesquisa de modo a obter circuitos a partir da garagem que guarda os camiões do lixo. Dos circuitos obtidos é possível escolher entre o primeiro encontrado, ou o mais rápido, o mais eficiente (permite recolher mais lixo), bem como o que passa por mais pontos de recolha. Também se criou um predicado que permite comparar circuitos, indicando o percurso que permite ir do ponto inicial ao final percorrendo a menor distância possível.

Neste trabalho, considerou-se que a capacidade do camião era ilimitada o que torna o problema irreal, sendo por isso necessário, no futuro, introduzir essa restrição ao problema. Também será necessário, de modo a obter melhores resultados, considerar uma representação do grafo diferente (as alternativas à representação atual já foram referidas no capítulo 2.2) o que implicará um aumento das arestas a considerar, contudo uma melhor representação da realidade.

Anexos

I – Representação do grafo reduzido

nodo(225,38.7163797615419,-9.15311830964257,'Misericordia',['Garagem'],0).

nodo(139,38.7154539083141,-9.15186340541146,'Misericordia',['Lixos'],240).
nodo(164,38.7144658214995,-9.15360843944475,'Misericordia',['Lixos'],140).
nodo(913,38.7140002169301,-9.15225663484226,'Misericordia',['Lixos'],140).
nodo(182,38.713272150325,-9.1531085611251,'Misericordia',['Lixos'],280).
nodo(206,38.7119754868767,-9.15305561262562,'Misericordia',['Lixos'],240).
nodo(700,38.7122430297428,-9.15130039259538,'Misericordia',['Lixos'],240).
nodo(39,38.7146089969724,-9.14751690960422,'Misericordia',['Lixos'],960).
nodo(693,38.7105108085557,-9.15264387969663,'Misericordia',['Lixos'],90).
nodo(122,38.7096839983153,-9.15242517945081,'Misericordia',['Lixos'],720).

nodo(227,38.7142829757734,-9.14460348898318,'Misericordia',['Deposito'],0).

aresta(225,164).
aresta(225,139).
aresta(139,225).
aresta(139,164).
aresta(139,913).
aresta(139,39).
aresta(164,225).
aresta(164,139).
aresta(164,913).
aresta(164,182).
aresta(913,139).
aresta(913,164).
aresta(913,182).
aresta(913,700).
aresta(913,39).
aresta(182,164).

aresta(182,913).

aresta(182,206).

aresta(182,700).

aresta(206,693).

aresta(206,700).

aresta(206,182).

aresta(700,206).

aresta(700,693).

aresta(700,182).

aresta(700,39).

aresta(700,913).

aresta(700,122).

aresta(39,139).

aresta(39,913).

aresta(39,700).

aresta(39,227).

aresta(227,39).

aresta(693,122).

aresta(693,206).

aresta(693,700).

aresta(122,693).

aresta(122,700).

II – Programa que realiza o parsing ao dataset

```
import openpyxl
import math
import re

path = "dataset.xlsx"

wb_obj = openpyxl.load_workbook(path)

sheet_obj = wb_obj.active
m_row = sheet_obj.max_row
max_col = sheet_obj.max_column
```

```
fich = []
for i in range(2, m_row + 1):
    s = []
    for j in range(1, max_col + 1):
        cell_obj = sheet_obj.cell(row = i, column = j)
        s.append(cell_obj.value)
    fich.append(s)

codRua = {}
codLoc = {}
adjacencias = {}
tipoContentor = {}
capacidadeContentor = {}
for line in fich:
    match = re.search(r'(\w+)\:([^\(]+)\((Par|Impar|Ambos)[^\:]+:([^-]+)-
([^)]+)',line[4])
    if(match):
        codRua[match[1]] = match[2].strip()
        codLoc[match[1]] = (line[0],line[1])
        if(match[1] not in adjacencias):
            adjacencias[match[1]] = set()
        if(match[1] not in tipoContentor):
            tipoContentor[match[1]] = set()
        if(match[1] not in capacidadeContentor):
            capacidadeContentor[match[1]] = 0
        capacidadeContentor[match[1]] += line[9]
        tipoContentor[match[1]].add(line[5].strip())
        adjacencias[match[1]].add(match[4].strip())
        adjacencias[match[1]].add(match[5].strip())
    else:
        match2 = re.search(r'(\w+)\:([^\,]+)',line[4])
        codRua[match2[1]] = match2[2].strip()
        codLoc[match2[1]] = (line[0],line[1])
        if(match2[1] not in tipoContentor):
            tipoContentor[match2[1]] = set()
        if(match2[1] not in capacidadeContentor):
            capacidadeContentor[match2[1]] = 0
        capacidadeContentor[match2[1]] += line[9]
        tipoContentor[match2[1]].add(line[5].strip())


for (x,y) in codRua.items():
    (lon,lat) = codLoc[x]
    tipo = list(tipoContentor[x])
    capacidade = capacidadeContentor[x]
    termo = f"nodo({x},{lat},{lon},'Misericordia',{tipo},{capacidade})."
    print(termo)

print('\n%--------------------------------------\n')
```

```
i = 0
codRua2 = list(codRua)
for x in codRua2:
    (lon,lat) = codLoc[x]
    anterior = (i>=1)
    anteriorA = (i>=2)
    proximo = i < (len(codRua)-1)
    proximoP = i < (len(codRua)-2)

    if(anteriorA):
        print(f"aresta({x},{codRua2[i-2]}).")

    if(anterior):
        print(f"aresta({x},{codRua2[i-1]}).")

    if(proximo):
        print(f"aresta({x},{codRua2[i+1]}).")

    if(proximoP):
        print(f"aresta({x},{codRua2[i+2]}).")

    i = i + 1
```