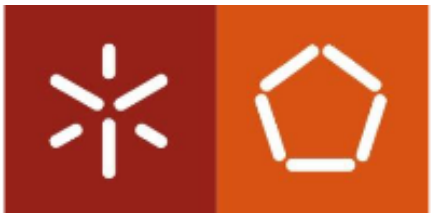

Universidade do Minho

Mestrado Integrado de Engenharia Informática

Sistemas de Representação de Conhecimento e Raciocínio

(2º Semestre – 2020/21)

Relatório SRCR – Trabalho Individual

A89587 Carlos João Teixeira Preto

Braga

5 de junho de 2021

Resumo

No âmbito da Unidade Curricular de Sistemas de Representação de Conhecimento e Raciocínio, foi proposto o desenvolvimento de um trabalho individual, que incidisse na aplicação de problemas de procura, recorrendo ao *Prolog*.

Para o desenvolvimento do trabalho prático, foi fornecido um dataset com várias informações acerca dos diferentes pontos de recolha na cidade de Lisboa, mais concretamente na freguesia da Misericórdia. Com base na informação presente no dataset, e juntamente com a visualização do mapa dos pontos de recolha em Lisboa, foi possível definir diferentes predicados, que posteriormente permitirão realizar operações de procura sobre os diferentes circuitos existentes.

Ao longo do relatório serão abordadas os diferentes pensamentos e decisões tomadas ao longo da realização do trabalho, bem como referir quais os diferentes circuitos desenvolvidos, bem como a fundamentação da sua escolha.

Índice

Lista de Figuras

1. Introdução

O desenvolvimento do trabalho prático teve por base a informação fornecida acerca dos pontos de recolha de resíduos urbanos do concelho de Lisboa, mais concretamente na freguesia da Misericórdia.

Com o trabalho prático, é pretendido que se desenvolva um sistema que permita importar dados relativos aos diferentes circuitos e representá-los numa base de conhecimento. Após importados os dados, será necessário desenvolver um sistema de recomendação de circuitos de recolha para o caso de estudo, tendo em conta os seguintes tópicos:

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

Os indicadores de produtividade serão a quantidade de resíduos recolhidos durante o circuito e a distância média entre os diferentes pontos de recolha. No desenvolvimento do trabalho **não se considerou a capacidade máxima do veículo coletor de Lixo.**

No desenvolvimento das soluções, será necessário considerar diferentes estratégias de procura (não-informada e informada) abordadas nas aulas, sendo que posteriormente deverá ser elaborada uma análise comparativa entre as diferentes estratégias de procura, tendo em conta tempos de execução.

2. Preliminares

2.1. Predicados e Definição do Circuito

Através da análise do *dataset* é possível obter quais as diferentes informações que caracterizam um determinado ponto de recolha. De toda a informação fornecida, apenas algumas representam dados importantes e únicos, capazes de distinguir os diferentes pontos de recolha. Assim, definiu-se que um ***pontoRecolha*** contém a sua latitude e longitude, qual o seu identificador(objectID), a freguesia à qual pertence, o código postal associado ao ponto de recolha, o tipo de resíduo que suporta e por fim, o total de litros de resíduos que contém.

Para se poder contruir os circuitos dos pontos de recolha, definiu-se um predicado, denominado **caminho**, que é definido pelo código de postal ponto de recolha de saída e pelo código postal do ponto de recolha de chegada.

Com base no referido, desenvolveu-se então os seguintes predicados em *prolog*:

```
:- dynamic pontoRecolha/7.
:- dynamic caminho/2.
```

Figura 1: Predicado pontoRecolha e caminho

```
pontoRecolha(-9.14320880914792, 38.7073787857025, 0, 'Misericordia',0, 'Garagem', 0).
pontoRecolha(-9.142812171, 38.706265483955, 1, 'Misericordia', 1, 'Deposito', 0).
pontoRecolha(-9.14330880914792, 38.7080787857025, 355, 'Misericordia', 15805, 'Lixos', 90).
pontoRecolha(-9.14330880914792, 38.7080787857025, 356, 'Misericordia', 15805, 'Lixos', 1680).
pontoRecolha(-9.14330880914792, 38.7080787857025, 357, 'Misericordia', 15805, 'Lixos', 90).
pontoRecolha(-9.14330880914792, 38.7080787857025, 358, 'Misericordia', 15805, 'Papel e Cartao', 1440).
pontoRecolha(-9.14330880914792, 38.7080787857025, 359, 'Misericordia', 15805, 'Papel e Cartao', 90).
pontoRecolha(-9.14337777820218, 38.7080781891571, 364, 'Misericordia', 15806, 'Lixos', 240).
pontoRecolha(-9.14337777820218, 38.7080781891571, 365, 'Misericordia', 15806, 'Lixos', 840).
pontoRecolha(-9.14337777820218, 38.7080781891571, 366, 'Misericordia', 15806, 'Lixos', 120).
pontoRecolha(-9.14337777820218, 38.7080781891571, 367, 'Misericordia', 15806, 'Lixos', 180).
pontoRecolha(-9.14337777820218, 38.7080781891571, 368, 'Misericordia', 15806, 'Lixos', 240).
pontoRecolha(-9.14337777820218, 38.7080781891571, 369, 'Misericordia', 15806, 'Lixos', 840).
pontoRecolha(-9.14337777820218, 38.7080781891571, 370, 'Misericordia', 15806, 'Lixos', 120).
pontoRecolha(-9.14337777820218, 38.7080781891571, 360, 'Misericordia', 15806, 'Lixos', 180).
pontoRecolha(-9.14337777820218, 38.7080781891571, 361, 'Misericordia', 15806, 'Papel e Cartao', 280).
pontoRecolha(-9.14337777820218, 38.7080781891571, 362, 'Misericordia', 15806, 'Papel e Cartao', 90).
pontoRecolha(-9.14337777820218, 38.7080781891571, 363, 'Misericordia', 15806, 'Papel e Cartao', 90).
```

Figura 2: Exemplo Informação pontoRecolha

Além da definição dos predicados do sistema, é também necessário definir os diferentes estados existentes. Através da análise do enunciado, é indicado que:

- Inicial – Percurso entre a garagem e o primeiro ponto de recolha.
- Final – Percurso entre local de deposição e a garagem.

Com base nesta informação, percebe-se que sempre que o veículo sai da garagem, o primeiro ponto de recolha que encontra é sempre o mesmo. Também se percebe que antes de voltar para garagem, terá de passar por um local de deposição, onde será removido todo o lixo do veículo. Assim, pode se definir que, quer **inicial** como **final** são definidos por um identificador, pelo que se obtém:

```
inicial(0).
final(1).
```

Figura 3: Predicado inicial e final

2.2. Definição do Circuito

Uma das maiores adversidades enfrentadas durante a resolução do projeto foi definir um circuito válido. Após algumas tentativas, verificou-se que com grafos muito grandes era praticamente impossível testar os casos práticos, visto que nenhum algoritmo conseguia calcular todos os caminhos possíveis num intervalo de tempo aceitável.

Por isso, optei por desenvolver dois grafos distintos, sendo o primeiro usado para testar a funcionalidade dos casos práticos, e um segundo para determinar os tempos de execução de cada algoritmo, ou seja, para testar a *performance* dos algoritmos de procura.

2.2.1. Circuito para Casos Práticos

Com base no mapa fornecido pelos docentes, bem como o *dataset* dos pontos de recolha da freguesia da Misericórdia, foi então desenvolvido um pequeno circuito, sobre o qual poderão ser aplicados os diferentes casos práticos, e posteriormente visualizar se os algoritmos dos casos práticos desenvolvidos estavam efetivamente corretos.

A primeira tarefa a ser realizada foi definir os nodos correspondentes à Garagem e ao Local de Depósito, que são representados na Figura 4 pelas cores vermelho e preto, respetivamente. Posteriormente, tratou-se de adquirir um conjunto de pontos e conecta-los entre si, de maneira a ter diferentes caminhos. Estes pontos foram escolhidos aleatoriamente, porém, sempre com o cuidado de cada nodo do grafo corresponder a um ponto de recolha no dataset fornecido. No total, foram definidos 40 caminhos, todos bidirecionais, à exceção dos caminhos que saem da garagem e que entram no depósito, ou seja, por exemplo, é possível ir do ponto de

recolha 15855 para o ponto de recolha 15808, e vice-versa, mas já só é possível ir de 0 para 15805. Na Figura 6 encontram-se alguns exemplos dos caminhos definidos.

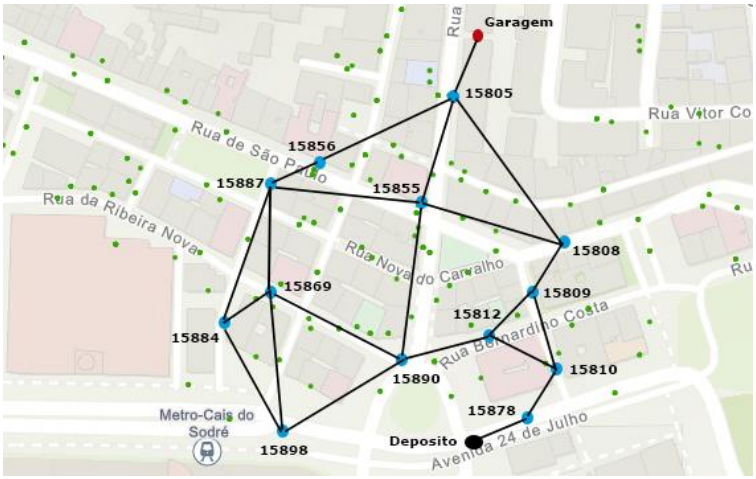

Figura 4: Circuito Casos Práticos

2.2.2. Circuito para *Performance*

Como foi dito anteriormente, este circuito de performance será usado para saber o tempo que cada algoritmo de procura demora a encontrar o primeiro caminho. Este circuito contém todos os pontos de recolha do *dataset* sobre os quais se pode obter informação, sendo que no total haverão 1571 caminhos. Com tal nível de profundidade, já será possível testar casos mais complexos e posteriormente avaliar o desempenho de cada algoritmo.

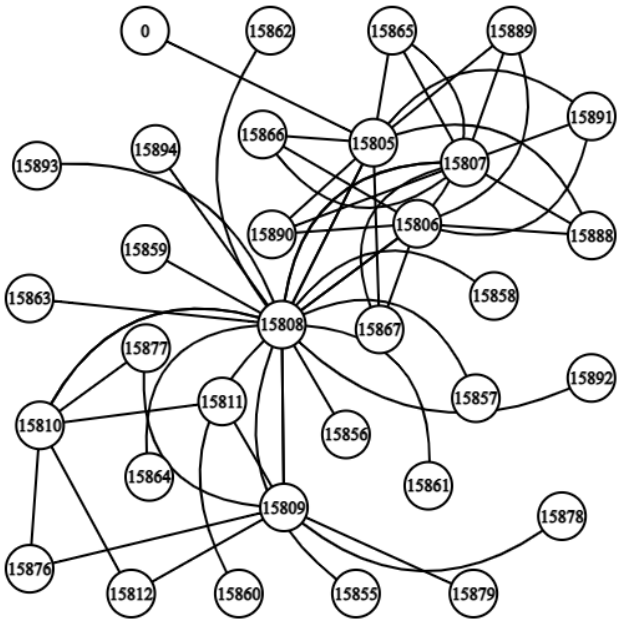

Figura 5: Parte do Circuito para Performance

Na Figura 4 encontra-se parte do grafo do circuito considerado para *performance,* onde apenas se representou 55 dos 1571 caminhos, de maneira a se poder perceber um pouco da composição deste. Facilmente se percebe que a enorme profundidade do grafo faz com que este seja um grafo válido para testar a performance de cada algoritmo.

2.3. Parsing

De maneira a extrair o reconhecimento do dataset para um ficheiro *Prolog*, recorreu-se à linguagem *Python.* O dataset é um documento .**xlsx,** pelo que foi necessário utilizar a biblioteca ***openpyxl*** e posteriormente formar um ficheiro ***.pl*** com a informação obtida.

Como foi referido anteriormente, cada linha do dataset contém a informação corresponde a cada um dos pontos de recolha do circuito, pelo que é possível popular os diferentes predicados **pontoRecolha,** apenas com a informação pretendida.

Além dos pontos de recolha fornecidos, foi necessário definir quais seriam as localizações e identificadores quer da Garagem, quer do Local de Depósito, decidindo-se que a Garagem teria como identificador o valor **0**, e o Local de Deposição teria como identificador o valor **1.** As localizações de ambos foram definidas aleatoriamente, visualizando os valores das latitudes e longitudes dos diferentes pontos de recolha e escolhendo valores que fossem diferentes e lógicos. Por fim, de maneria a distinguir a garagem e o local de deposição dos restantes pontos de recolha, identificou-se os seus tipos como “Garagem” e “Deposito”, respetivamente, bem como colocar as suas quantidades de lixo a 0, visto que nesses locais não há recolha de lixo. Desta forma, é possível obter toda a informação relativa a um determinado ponto de recolha.

Para se escrever os caminhos num ficheiro *prolog*, tem de se considerar qual o circuito que pretendemos, tal como referido no subcapítulo anterior sobre a definição do circuito. Assim, caso se pretenda obter o grafo para testar os casos práticos, terá de se executar o ficheiro ***parserGrafoCompleto.py****,* caso se pretenda obter o grafo para testes dos casos práticos, então será necessário executar o ficheiro ***parserGrafoPequeno.py.*** A diferença nestes 2 ficheiros encontra-se apenas na escrita dos caminhos do circuito, visto que a informação sobre os pontos de recolha nunca vai mudar, independentemente do grafo.

```
import re

file = open("allPaths.txt")

for line in file:
    n1 = line.split(" ")
    n2 = n1[1].split("\n")
    s = "caminho(" + n1[0] + ", " + n2[0] + ").\n"
    prolog.write(s)

prolog.write('caminho(0,15805).\n')
prolog.write('caminho(21961,1).\n')
```

Figura 6: Parser grande

Na figura 5 encontra-se o código para escrever todos os caminhos no ficheiro ***pontosRecolha.pl***, sendo apenas importante destacar que o ficheiro *allPaths.txt* contém todos os caminhos do grafo, obtidos após uma análise extensiva da informação no dataset, e com base na paridade e imparidade de cada rua, isto é, se um dada rua é impar ou par a outras duas, então liga-se esse código postal aos que estavam dentro de parêntesis, já quando não apareciam estes nomes, então liga-se o ponto de recolha ao ponto de recolha quer acima, quer abaixo, desde que não se trate do mesmo código postal o mesmo código postal. No ficheiro ***parserGrafoPequeno.pl*** procedeu-se à escrita manual dos pontos pretendidos.

3. Descrição do Trabalho e Análise de Resultados

3.1. Algoritmos de Pesquisa

Após realizado o parsing obtém-se um ficheiro com o conjunto de factos que constituem a base de conhecimentos que será utilizada. Com base nesse ficheiro, e manipulando esses dados, será possível desenvolver algoritmos que nos permitam chegar a conclusões acerca dos diferentes circuitos possíveis.

Existem dois tipos de algoritmos de pesquisa em grafos, sendo esses as pesquisas **não-informadas**, onde se destacam algoritmos de procura em profundidade, largura e busca iterativa limitada em profundidade, e as pesquisas **informadas,** composta por algoritmos como a Gulosa e A*(A estrela). No trabalho desenvolveu-se todos estes algoritmos, adaptando-os ao contexto pretendido, e será em seguida explicado como funciona cada um deles, bem como qual a sua complexidade temporal e espacial.

3.1.1. Pesquisa Não-Informada

3.1.1.1. Pesquisa em Profundidade *(Depth-First Search)*

A estratégia de procura em profundidade(*Depth-First Search*) consiste em partir de um dado nodo do grafo e optar por avançar para um nodo adjacente o mais profundo possível, e só depois de chegar à profundidade máxima é que inicia o *backtracking*. Na Figura 6 é possível observar a representação do algoritmo de pesquisa em profundidade num grafo.

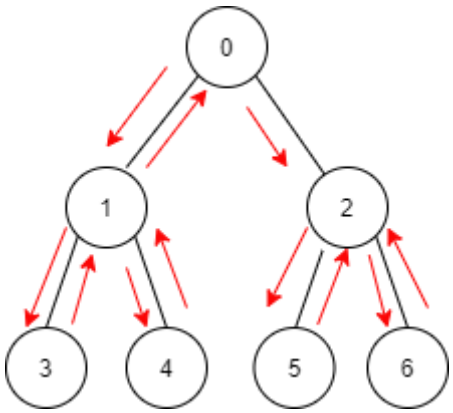

Figura 7: Esquema DFS

Em termos de complexidade, é fácil perceber que quanto maior o número de caminhos existentes no grafo, mais memória terá de ser usada, havendo situações onde, apartir de um determinado número de caminhos, o algoritmo deixa de terminar. Uma maneira de controlar tal problema poderá ser limitar a profundidade até à qual se procura caminhos, tal como acontece no algoritmo de busca iterativa limitada em profundidade, que também será abordado neste relatório**.**

Assim, em termos de **complexidade temporal**, obtemos *O*(Número de Pontos de Recolha) e em termos de **complexidade espacial** obtemos *O*(Profundidade da árvore), ou seja, quanto mais profundo for o nosso grafo, maior será a complexidade espacial deste.

```
depthFirst(NodoInicial, NodoFinal, [NodoInicial|Caminho]) :-
    profundidadePrimeiro(NodoInicial, NodoFinal, [NodoInicial], Caminho).

profundidadePrimeiro(Nodo,Nodo,_,[]).
profundidadePrimeiro(NodoInicial, NodoFinal, Historico, [ProxNodo|Caminho]) :-
    adjacenteDF(NodoInicial,ProxNodo),
    nao(membro(ProxNodo,Historico)),
    profundidadePrimeiro(ProxNodo, NodoFinal, [ProxNodo|Historico],Caminho).

adjacenteDF(Nodo,ProxNodo) :-
    caminho(Nodo,ProxNodo).
```

Figura 8: Algortimo DFS

O algoritmo de ***depthFirst*** tinha sido previamente desenvolvido na aulas práticas da Unidade Curricular de Sistemas de Representação de Conhecimento e Raciocínio, sendo apenas necessário adaptar o algoritmo ao contexto atual do problema. Como se pode observar na Figura 7, um Caminho é formado por todos os conjuntos de pontos de recolha pelos quais o veículo passa, começando num determinado Nodo, denominado *NodoInicial*, e terminando um outro Nodo, denominado *NodoFinal*.

O predicado ***profundidadePrimeiro*** determina quais os nodos adjacentes ao nodo inicial pretendido, e verifica se esse nodo já se encontra na lista de pontos percorridos. Caso não esteja, adiciona esse nodo adjacente ao histórico e volta a repetir o algoritmo, até ter percorrido todos os nodos adjacentes.

Através do predicado ***adjacenteDF*****,** e usando como predicado auxiliar o predicado caminho, será colocado no *ProxNodo* o próximo nodo adjacente a um determinado nodo.

3.1.1.2. Pesquisa em Largura(*Breadth-First Search*)

A estratégia de procura em largura consiste em partir do nodo inicial do grafo e visitar todos os nodos adjacentes, sendo que para cada um desses nodos adjacentes se deverá explorar os seus respetivos nodos adjacentes, até se encontrar o nodo final. Tal algoritmo pode ser representando como a o diagrama da Figura 8.

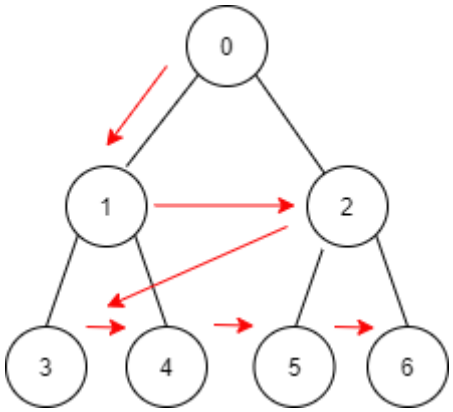

Figura 9: Esquema BFS

Em termos de **complexidade temporal**, o algoritmo de pesquisa em largura apresenta *O*(Número de Pontos de Recolha + Número Caminhos), visto que para cada ponto de recolha terá de se percorrer todos os seus adjacentes. Já em termos de **complexidade espacial**, obtemos *O*(Número de Pontos de Recolha), um vez que quantos mais pontos de recolha tivermos, mais caminhos irão existir.

```
breadthFirst(NodoInicial, NodoFinal, CaminhoFinal) :-
    breadthPrimeiro([[NodoInicial]], NodoFinal, Solucao),
    inverteLista(Solucao, CaminhoFinal, []).

breadthPrimeiro([[Nodo|Caminho]|_], Nodo, [Nodo|Caminho]).
breadthPrimeiro([[Nodo|Caminho]|CaminhoList], NodoFinal, Solucao) :-
    bagof([ProxNodo, Nodo|Caminho],
    (adjacenteBF(Nodo, ProxNodo),\+ membro(ProxNodo, [Nodo|Caminho])), NovosCaminhos),
    append(CaminhoList, NovosCaminhos, Res), !,
    breadthPrimeiro(Res, NodoFinal, Solucao);
    breadthPrimeiro(CaminhoList, NodoFinal, Solucao).

adjacenteBF(Nodo, NodoAdj) :-
    caminho(Nodo, NodoAdj).
```

Figura 10: Predicado Algoritmo BFS

Analisando a Figura 9, é possível perceber que, através do predicado ***bagof*** , se vai buscar todos os nodos adjacentes ao nodo atual, sendo que, aplicando recursividade a cada um desses nodos adjacentes, se irão obter os nodos adjacentes a cada um dos nodos adjacentes atuais, permitindo assim descobrir o caminho do nodo inicial até ao nodo final pretendido.

3.1.1.3. Pesquisa Iterativa Limitada em Profundidade

A estratégia de procura em iterativa limitada em profundidade pode ser vista como uma adaptação do algoritmo de procura em profundidade *(Depth-First Search),* uma vez que este algoritmo executa uma versão da pesquisa em profundidade, limitada a um dado nível de profundidade pretendido.

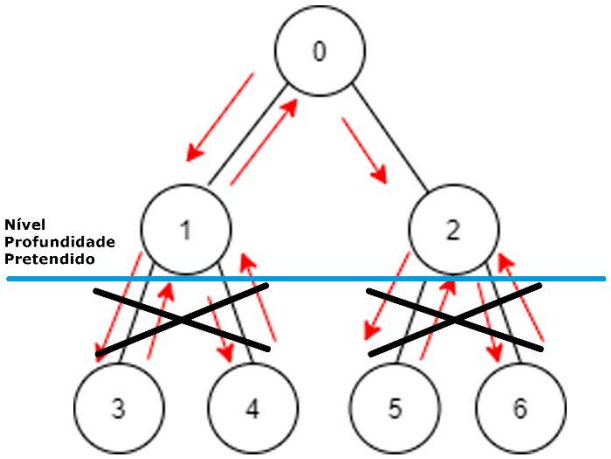

Figura 11: Esquema Profundidade Iterativa

Em termos de **complexidade temporal**, obtemos *O*(Número de Pontos de Recolha), já em termos de **complexidade espacial**, obtemos *O*(Nível Profundidade Pretendida), onde se percebe facilmente que quanto maior a profundidade pretendida, maior será o espaço ocupado pelo algoritmo, tal como se sucedia no algoritmo de procura em profundidade.

```
depthFirstLimited(NodoInicial, NodoFinal, Nivel,[NodoInicial|Caminho]) :-
    profundidadePrimeiroLimited(NodoInicial, NodoFinal, Nivel, [NodoInicial], 0, Caminho).

profundidadePrimeiroLimited(Nodo, Nodo,_,_,[]).
profundidadePrimeiroLimited(NodoInicial, NodoFinal, Nivel, Historico, Counter, [ProxNodo|Caminho]) :-
    Counter < Nivel,
    adjacenteDF(NodoInicial,ProxNodo),
    nao(membro(ProxNodo,Historico)),
    NewCounter is Counter + 1,
    profundidadePrimeiroLimited(ProxNodo, NodoFinal, Nivel,[ProxNodo|Historico], NewCounter, Caminho).
```

Figura 12: Predicado Algoritmo Profundidade Iterativa

O algoritmo de ***depthFirstLimited*** resulta então de uma adaptação do algoritmo ***depthFirst.*** Como se pode observar na Figura 11, o predicado ***profundidadePrimeiroLimited***, para além do nível de profundidade pretendido, recebe também um contador da profundidade atual, sendo esta denominada de *Counter*, inicializada sempre a 0 e, há medida que se progredir na árvore, incrementa-se então o *Counter*, parando apenas quando se tiver num nível de profundidade atual superior ao pretendido.

3.1.2. Pesquisa Informada

3.1.2.1. Pesquisa Gulosa

O algoritmo de pesquisa Gulosa determina qual a travessia ideal de um grafo, com base na escolha local ideal em cada nodo, ou seja, partindo de um determinado nó e sabendo quais os seus adjacentes, o próximo nodo a ser visitado será aquele que apresentar os melhores valores de otimização.

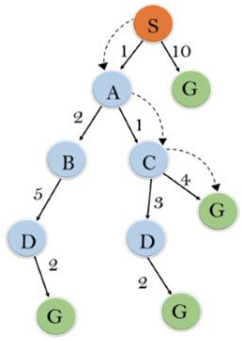

Figura 13: Esquema Gulosa

Como se pode observar na Figura 12, o próximo nodo a ser escolhido para a travessia será aquele com o melhor valor de otimização. No contexto do problema atual, definiu-se que o valor de otimização seria o total de lixo recolhido,

Este algoritmo tem a vantagem de produzir uma solução globalmente ótima num período de tempo razoável, em comparação com outros algoritmos de procura, porém, em muitos problemas, a solução final nem sempre corresponde à melhor solução possível.

Em termos de **complexidade temporal** e **complexidade espacial**, pode-se concluir que ambos são idênticos, sendo estes tempos dependentes do nível de profundidade do grafo ao qual é aplicada, ou seja *O*(Profundidade da árvore), onde quanto maior o a profundidade do grafo, maior a complexidade.

```
gulosaSearch(Partida, Destino, Caminho) :-
    distancia(Partida, Destino,Estimativa),
    agulosa([[Partida]/0/Estimativa],Destino,InvCaminho/Custo/_),
    inverteLista(InvCaminho, Caminho, []).

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

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


expande_gulosa(Caminho, ExpCaminhos, Destino) :-
    findall(NovoCaminho, adjacenteGulosa(Caminho,NovoCaminho,Destino), ExpCaminhos).


adjacenteGulosa([Nodo|Caminho]/Custo/_, [ProxNodo,Nodo|Caminho]/NovoCusto/Est,Destino) :-
    caminho(Nodo, ProxNodo),
    distancia(Nodo, ProxNodo, PassoCusto),
    \+ member(ProxNodo, Caminho),
    NovoCusto is Custo + PassoCusto,
    distancia(ProxNodo,Destino ,Est).

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

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

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

Figura 14: Predicado Algoritmo Gulosa

Como se pode comprovar na Figura 13, o predicado ***obtem_melhor_gulosa,*** escolhe o caminho que tiver menor estima, o predicado ***expande_gulosa*** vai encontrar todos os caminhos que saem de um nodo e chegam até ao próximo nodo da lista de caminhos. O predicado ***adjacenteGulosa*** determina a distância entre o nodo atual e o próximo nodo, e caso esse próximo nodo ainda não exista nos caminhos existentes, então adiciona-se a distância à distância total e coloca-se como estima a distância desse próximo nodo até ao nodo final pretendido.

3.1.2.2. Pesquisa A* (A estrela)

O algoritmo de pesquisa A*(A estrela) é bastante reconhecido devido à sua completude, otimalidade e eficiência. Este consiste em, partindo de um nó inicial, encontrar um caminho para o nodo final pretendido com o menor custo, armazenando uma árvore de caminhos originados no nodo inicial e expandindo esses caminhos, para a esquerda ou para a direita, até que o critério de terminação seja satisfeito.

Apesar de ser de ser bastante eficiente, tem uma grande desvantagem, que se se encontra no facto de armazenar todos os nós gerados na memória, fazendo com que a sua **complexidade espacial** seja bastante grande. Em termos de **complexidade temporal**, o tempo de execução do algoritmo irá depender da heurística do grafo.

```
astarSearch(Nodo, Destino, Caminho) :-
    distancia(Nodo,Destino,Estima),
    astar([[Nodo]/0/Estima], Destino, InvCaminho/Custo/_),
    inverteLista(InvCaminho, Caminho, []).

astar(Caminhos,Destino, Caminho) :-
    obtem_melhor_astar(Caminhos, Caminho),
    Caminho = [Nodo|_]/_/_,
    Nodo == Destino.

astar(Caminhos,Destino, SolucaoCaminho) :-
    obtem_melhor_astar(Caminhos, MelhorCaminho),
    seleciona(MelhorCaminho, Caminhos, OutrosCaminhos),
    expande_astar(MelhorCaminho, Destino,ExpCaminhos),
    append(OutrosCaminhos, ExpCaminhos, NovoCaminhos),
    astar(NovoCaminhos, Destino,SolucaoCaminho).

expande_astar(Caminho,Destino, ExpCaminhos) :-
    findall(NovoCaminho, move_astar(Caminho,Destino,NovoCaminho), ExpCaminhos).

move_astar([Nodo|Caminho]/Custo/_, Destino,[ProxNodo,Nodo|Caminho]/NovoCusto/Est) :-
    caminho(Nodo, ProxNodo),
    distancia(Nodo, ProxNodo, PassoCusto),
    \+ member(ProxNodo, Caminho),
    NovoCusto is Custo + PassoCusto,
    distancia(ProxNodo,Destino,Est).

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

*Figura 15: Predicado Algoritmo A**

Como se pode observar na Figura 14, no predicado ***astar***, uma vez escolhido o melhor caminho, expande-se esse caminho e adiciona-se este aos novos caminhos, para posteriormente se chamar recursivamente para os restantes caminhos. Um caminho é o melhor caso o somatório da estimativa do custo com o custo atual do caminho seja o menor.

3.2. Elaboração do Caso Prático

Como foi referido no capítulo de introdução, foi proposto a elaboração de diferentes casos práticos que permitissem manipular a informação na base de dados, de maneira a obter diferentes circuitos com base em diferentes parâmetros.

De seguida, será apresentado o raciocínio e excertos de código associados a resolução de cada um desses casos práticos, sendo importante referir, uma vez mais, que para testar a veracidade dos casos práticos, se recorreu ao circuito para casos práticos, obtido através da execução do ficheiro ***parserGrafoPequeno.pl.***

3.2.1. Gerar circuitos que cubram um determinado território

Uma vez que todos os pontos de recolha fornecidos no *dataset* pertencem à freguesia da misericórdia, considerou-se que que todos pontos que cobrem um determinado território serão aqueles que saem da garagem e terminam no local de deposição. Uma vez que foram desenvolvidos 5 algoritmos de procura, desenvolveu-se então 5 predicados associados a cada um desses algoritmos, porém, é fácil concluir que **o melhor algoritmo para descobrir todos os caminhos será o DFS**, visto que não se preocupa com nenhum critério selecção, ao contrário do que acontece nos algoritmos de pesquisa informada. Na Figura 15 encontra-se o exemplo da aplicação do predicado ao algoritmo de procura em profundidade.

```
circuitosRecolha_Depth(Territorio) :-
    inicial(NodoInicial),
    final(NodoFinal),
    pontoRecolha(_,_,_,Territorio,NodoInicial,_,_),
    pontoRecolha(_,_,_,Territorio,NodoFinal,_,_),
    findall(Path, depthFirst(NodoInicial, NodoFinal, Path), Caminho),
    escrever(Caminho).
```

Figura 16: Predicado circuitosRecolha_Depth

De maneira a verificar se os algoritmos estão direitos, podemos então escrever o nome do território pretendido e obter todos os caminhos possíveis nesse mesmo território, tal como se pode observar no exemplo da Figura 16, onde se utiliza o algoritmo de pesquisa em profundidade.

```
| ?- circuitosRecolha_Depth("Misericordia").
Caminho: [0,15805,15856,15887,15884,15869,15898,15890,15855,15808,15809,15810,15878,1]
Caminho: [0,15805,15856,15887,15884,15869,15898,15890,15855,15808,15809,15812,15810,15878,1]
Caminho: [0,15805,15856,15887,15884,15869,15898,15890,15812,15809,15810,15878,1]
Caminho: [0,15805,15856,15887,15884,15869,15898,15890,15812,15810,15878,1]
Caminho: [0,15805,15856,15887,15884,15869,15890,15855,15808,15809,15810,15878,1]
Caminho: [0,15805,15856,15887,15884,15869,15890,15855,15808,15809,15812,15810,15878,1]
Caminho: [0,15805,15856,15887,15884,15869,15890,15812,15809,15810,15878,1]
Caminho: [0,15805,15856,15887,15884,15869,15890,15812,15810,15878,1]
Caminho: [0,15805,15856,15887,15884,15898,15869,15890,15855,15808,15809,15810,15878,1]
Caminho: [0,15805,15856,15887,15884,15898,15869,15890,15855,15808,15809,15812,15810,15878,1]
Caminho: [0,15805,15856,15887,15884,15898,15869,15890,15812,15809,15810,15878,1]
Caminho: [0,15805,15856,15887,15884,15898,15869,15890,15812,15810,15878,1]
Caminho: [0,15805,15856,15887,15884,15898,15890,15855,15808,15809,15810,15878,1]
Caminho: [0,15805,15856,15887,15884,15898,15890,15855,15808,15809,15812,15810,15878,1]
Caminho: [0,15805,15856,15887,15884,15898,15890,15812,15809,15810,15878,1]
Caminho: [0,15805,15856,15887,15884,15898,15890,15812,15810,15878,1]
Caminho: [0,15805,15856,15887,15869,15884,15898,15890,15855,15808,15809,15810,15878,1]
Caminho: [0,15805,15856,15887,15869,15884,15898,15890,15855,15808,15809,15812,15810,15878,1]
```

Figura 17: Resultados circuitosRecolha

3.2.2. Identificar quais circuitos com mais pontos de recolha(por tipo de resíduo a recolher)

Para descobrir quais os circuitos com mais pontos de recolha por tipo de resíduo a escolher, é necessário descobrir todos os caminhos que saem da garagem e terminam no local de depósito e apenas recolhem lixo de um determinado tipo. O tipo de lixo pode ser “Lixos”, “Vidros” , “Papel e Cartao”, “Embalagens” e “Organicos”.

Primeiramente, calculam-se todos os caminhos do território em questão, e depois de obtidos os caminhos, verificam-se quais é que recolhem um dado tipo de lixo, através do predicado ***restringeTipo***. Depois de descobertos todos os caminhos que recolhem um dado lixo, é uma questão de descobrir qual desses tem o maior número de pontos de recolha, sendo tal calculado pelo predicado ***maisPontosRecolha***.

Uma vez mais, o **algoritmo mais adequado será o DFS**, porque nas procuras informadas, é muito mais provável que o caminho com mais pontos de recolha seja o último a ser encontrado. Na Figura 18 encontra-se um exemplo do cálculo do circuito com mais pontos de recolha, sendo que neste caso se utiliza o algoritmo de pesquisa em profundidade. Este predicado foi posteriormente desenvolvido para cada um dos 5 tipos de algoritmos de procura.

```
circuitoMaisPontosRecolha_Depth(Territorio, TipoLixo) :-
    inicial(NodoInicial),
    final(NodoFinal),
    pontoRecolha(_,_,_,Territorio,NodoInicial,_,_),
    pontoRecolha(_,_,_,Territorio,NodoFinal,_,_),
    findall(Path, depthFirst(NodoInicial, NodoFinal, Path), Solucao),
    restringeTipo(Solucao, TipoLixo, SolucaoFiltrada),
    maisPontosRecolha(SolucaoFiltrada, Caminho, NrNodos), !,
    write('\nMelhor Caminho: '),
    write(Caminho),
    write('\nNumero de Pontos de Recolha: '),
    write(NrNodos),
    write(' Pontos de Recolha.').
```

Figura 18: Predicado circuitoMaisPontosRecolha_Depth

De maneira a verificar se os algoritmos estão direitos, podemos então escrever o nome do território pretendido e qual o tipo de lixo pretendido, e iremos obter assim o maior caminho que recolhem um dado tipo de lixo, nesse mesmo território. Tal pode ser observado no exemplo da Figura 19, onde se utiliza o algoritmo de pesquisa em profundidade acima descrito para descobrir o caminho que recolhe “Lixos” e que cobre o território da freguesia da “Misericórdia”.

```
| ?- circuitoMaisPontosRecolha_Depth('Misericordia', 'Lixos').

Melhor Caminho: [0,15805,15856,15887,15884,15869,15898,15890,15855,15808,15809,15812,15810,15878,1]
Numero de Pontos de Recolha: 15 Pontos de Recolha.
```

Figura 19: Resultados circuitoMaisPontosRecolha_Depth ’Lixos’

3.2.3. Comparar circuitos de recolha tendo em conta os indicadores de produtividade

Há dois tipos de indicadores de produtividade, sendo estes a distância e o lixo recolhido. Considera-se que um caminho é produtivo quanto maior for a distância percorrida ou a quantidade de lixo recolhida. Para se descobrir quais os melhores caminhos para cada um destes indicadores, utilizaram-se os predicados auxiliares ***maiorLixoRecolhido*** e ***maiorDistancia*** , onde o primeiro recebe uma lista de caminhos e determinada qual a quantidade de lixo recolhida em cada um, ficando apenas com aquele que tiver maior quantidade de lixo. Relativamente ao segundo predicado, este recebe na mesma uma lista de caminhos, só que desta vez determina qual a distância associada a cada um desses caminhos, ficando apenas com o que tiver maior distância. Na Figura 20, podemos observar o predicado ***compararCircuitosBreadth,*** onde se calcula os melhores caminhos para um determinado indicador de produtividade, usando o algoritmo de pesquisa em largura. Mais uma vez, foram desenvolvidos 5 predicados, associados a cada um dos algoritmos de procura.

```
compararCircuitos_Breadth(Territorio, IndicadorProdutividade) :-
    inicial(NodoInicial),
    final(NodoFinal),
    pontoRecolha(_,_,_,Territorio,NodoInicial,_,_),
    pontoRecolha(_,_,_,Territorio,NodoFinal,_,_),
    IndicadorProdutividade = 'Distancia',
    findall(Path, breadthFirst(NodoInicial, NodoFinal, Path), Solucao),
    maiorDistancia(Solucao, Caminho, Valor), !,
    write('Caminho: '),
    write(Caminho),
    write('\nDistancia: '),
    write(Valor),
    write(' metros\n').

compararCircuitos_Breadth(Territorio, IndicadorProdutividade) :-
    inicial(NodoInicial),
    final(NodoFinal),
    pontoRecolha(_,_,_,Territorio,NodoInicial,_,_),
    pontoRecolha(_,_,_,Territorio,NodoFinal,_,_),
    IndicadorProdutividade = 'Lixo',
    findall(Path, breadthFirst(NodoInicial, NodoFinal, Path), Solucao),
    maiorLixoRecolhido(Solucao, Caminho, Valor), !,
    write('Caminho: '),
    write(Caminho),
    write('\nLixo Recolhido: '),
    write(Valor),
    write(' cm3 \n').
```

Figura 20: Predicado compararCircuitos_Breadth

```
| ?- compararCircuitos_Breadth('Misericordia', 'Lixo').
Caminho: [0,15805,15856,15887,15884,15869,15898,15890,15855,15808,15809,15812,15810,15878,1]
Lixo Recolhido: 23980 cm3
yes
| ?- compararCircuitos_Breadth('Misericordia', 'Distancia').
Caminho: [0,15805,15808,15855,15887,15884,15898,15869,15890,15812,15809,15810,15878,1]
Distancia: 959.0817627265724 metros
```

Figura 21: Resultados compararCircuitos_Breadth

3.2.4. Escolher o circuito mais rápido

O circuito mais rápido será aquele que, para um determinado território, percorrer uma menor distância. O predicado auxiliar ***menorDistancia*** pega numa lista de caminhos e para cada um deles, vai determinar qual a menor distância. Foram desenvolvidos os 5 predicados para determinar qual a menor distância para cada um dos algoritmos de pesquisa, **porém, o mais adequado a este problema será a procura A*,** visto que o primeiro caminho que este devolve já será aquele com a menor distância. Na Figura 22 encontra-se o exemplo do predicado de procura de caminhos mais curtos, através do algoritmo de procura A*.

```
circuitoMaisCurto_Astar(Territorio) :-
    inicial(NodoInicial),
    final(NodoFinal),
    pontoRecolha(_,_,_,Territorio,NodoInicial,_,_),
    pontoRecolha(_,_,_,Territorio,NodoFinal,_,_),
    astarSearch(NodoInicial, NodoFinal, Caminho),
    distanciaPercorrida(Caminho, Distancia), !,
    write('Menor Caminho: '),
    write(Caminho),
    write('\nDistancia: '),
    write(Distancia),
    write(' metros\n').
```

Figura 22: Predicado circuitoMaisCurto_Astar

```
| ?- circuitoMaisCurto_Astar('Misericordia').
Menor Caminho: [0,15805,15808,15809,15810,15878,1]
Distancia: 330.99962234534576 metros
```

Figura 23: Resultados circuitoMaisCurto_Astar

3.2.5. Escolher circuito mais eficiente

Ficou à escolha do aluno definir qual o critério que define o circuito mais eficiente. Após alguma pesquisa, decidi que o critério de eficiência de um caminho iria ser referente ao número de nodos percorridos, ou seja, quanto menor for o número de nodos percorridos, mais eficiente será um caminho.

Como se pode observar na Figura 24, o predicado ***menosPontosRecolha*** recebe uma lista de caminhos e determina qual terá menor número de pontos de recolha.

```
circuitoMaisEficiente_Astar(Territorio) :-
    inicial(NodoInicial),
    final(NodoFinal),
    pontoRecolha(_,_,_,Territorio,NodoInicial,_,_),
    pontoRecolha(_,_,_,Territorio,NodoFinal,_,_),
    findall(Path, astarSearch(NodoInicial, NodoFinal, Path), Solucao),
    menosPontosRecolha(Solucao, Caminho, Pontos), !,
    write('Menor Caminho: '),
    write(Caminho),
    write('\nNodos: '),
    write(Pontos),
    write(' Pontos de Recolha\n').
```

Figura 24: Predicado circuitoMaisEficiente_Astar

```
| ?- circuitoMaisEficiente_Astar('Misericordia').
Menor Caminho: [0,15805,15808,15809,15810,15878,1]
Nodos: 7 Pontos de Recolha
```

Figura 25: Resultados circuitoMaisEficiente_Astar

3.3. Análise Comparativa

Uma vez concluída a definição dos casos práticos, é necessário fazer uma breve comparação entre os diferentes algoritmos de procura. De maneira a realizar comparações válidas, irá ser comparado os tempos de execução de cada algoritmo associado ao calculo de trajeto entre dois pontos, onde apenas se irá considerar o tempo demorado a encontrar a primeira solução, visto que nenhum predicado é capaz de encontrar todos os caminhos do circuito em tempo válido.

Antes de se prosseguir para os resultados obtidos, é importante notar que o circuito utilizado para obter os tempos a seguir apresentados foi o circuito para testes performance, obtido através da execução do ficheiro ***parserGrafoCompleto.py***. Também é importante referir que o tempo de execução de cada algoritmo irá depender da profundidade do grafo, onde quanto maior esta, mais tempo e memória será despendido até encontrar a solução. Assim, definiram-se três tipos de testes, um fácil, um intermédio e um último difícil, sendo estes:

- ✓ **Teste Fácil (T1):** Caminho de **15805** até **15842**
- ✓ **Teste Intermédio (T2):** Caminho de **15805** até **19293**
- ✓ **Teste Difícil (T3):** Caminho de **15805** até **21949**

Os resultados obtidos para cada um dos algoritmos de pesquisa podem ser visualizados na seguinte tabela:

Estratégia	Tempo
Profundidade(DFS)	T1: 0 s T2: 0 s T3: 0.01 s
Largura(BFS)	T1: 0.016 s T2: Não Termina T3: Não Termina
Profundidade Iterativa(Limite Máximo)	T1: 0 s T2: 0 s T3: 0.01 s
Gulosa	T1: 0 s T2: 0.016 s T3: Não Termina
A*	T1: 0.140 s T2: 0.125 s T3: Não Termina

Figura 26: Tabela Comparativa Algoritmos

Como se pode observar na tabela, para o teste fácil (T1) todos os algoritmos conseguem calcular um caminho de forma instantânea. Relativamente ao teste intermédio, reparamos que o algoritmo BFS é o único não consegue calcular um caminho em tempo válido, uma vez que fica infinitamente a processar. Por fim, percebe-se que apenas o algoritmo DFS consegue obter uma solução para o teste difícil. Tal é percetível, uma vez que este algoritmo apenas vai avançando na profundidade, sem ter em consideração nenhum critério, o que significa que apesar de se obter uma solução, esta muito provavelmente não corresponderá à melhor solução.

O facto do algoritmo BFS não terminar deve-se ao facto de o grafo considerado ter uma profundidade bastante grande. Uma vez que este tem de percorrer primeiro todos os nodos de cada nível de profundidade, quanto maior for a profundidade, maior será o tempo até encontrar o caminho.

Relativamente ao algoritmos de pesquisa informada, verifica-se que o tempo de execução destes é superior ao DFS e inferior ao BFS. O algoritmo da Gulosa, relativamente à procura A*, apenas considera uma solução ótima local, pelo que, em termos de tempo de execução, será mais rápida, tal como se verifica na tabela. A algoritmo A*, uma vez que não se preocupa apenas com a solução ótima local, mas também com a solução ótima global, demorará mais tempo a obter um caminho.

Conclui-se assim que, caso se pretenda obter rapidamente um caminho, sem considerar a nenhum tipo de critério, então o algoritmo DFS será o mais adequado. Porém, caso o objetivo seja obter o possivelmente melhor caminho então os algoritmos de pesquisa informada serão os mais adequados, sendo que destes, o que apresentará a solução mais próxima da solução ótima será a procura A*.

3.4. Estrutura do trabalho

Antes de se passar para as conclusões do trabalho, gostaria de explicar a estrutura do código do trabalho. A pasta enviada contempla o **relatório** do trabalho prático, o ficheiro ***pontosRecolha*.pl,** onde se optou por enviar o circuito mais pequeno para obter os resultados dos casos práticos, o ***allPaths.txt*** com todos os caminhos do dataset, os dois tipos de ficheiros de parsing, sendo esses o ***parserGrafoPequeno.py*** e ***parserGrafoCompleto.py***, apartir dos quais se pode gerar o ficheiro ***pontosRecolha*.pl** com o circuito pretendido, um ficheiro denominado ***predicadosAuxiliares.pl*** que contém todos os predicados auxiliares utilizados para a resolução do trabalho prático, tais como os predicados habituais seleciona, inverteLista, atualizar e escrever. Também serão enviados dois ficheiros denominados ***procuraNaoInformada*.pl** e ***procuraInformada*.pl**, que contém, respetivamente, os predicados dos algoritmos de procura não-informada e informada. Por fim, será enviado o principal ficheiro do trabalho prático, denominado ***main.pl*,** onde estarão desenvolvidos todos os casos práticos, e onde poderão ser invocados todos os outros predicados.

4. Conclusões e Sugestões

Com a realização deste trabalho foi possível aplicar estratégias para a resolução de problemas com o uso de algoritmos de procura. A parte mais desafiante do projeto foi conseguir definir um grafo com uma dimensão relativamente grande, onde se pudesse comparar os diferentes algoritmos. A parte de desenvolver os algoritmos revelou-se relativamente fácil, na medida em que apenas foi necessário adaptar estes ao contexto do problema.

Apesar do desafio, os objetivos traçados para este projeto acabam por ser concluídos com sucesso, tendo sido implementados todos os casos práticos apresentados no enunciado, não só para um único algoritmo, mas sim para todos os algoritmos de pesquisa não-informada e informada desenvolvidos. Assim, para cada caso prático, é possível observar quais os tempos e respostas associados a cada um dos algoritmos.

Por fim, como trabalho futuro, gostaria de tentar considerar a capacidade máxima do veiculo de recolha de lixo, bem como acrescentar mais indicadores de produtividade, para assim ter uma melhor noção de como diferentes fatores irão afetar a escolha dos melhores caminhos.

5. Referências

[RUSSEL,2010], RUSSEL S. J., Norvig, P., & Davis, E. Artificial intelligence: a modern approach. 3rd ed. Upper Saddle River, NJ, 2010

Depth-first search algorithm. Disponível em: https://en.wikipedia.org/wiki/Depth-first_search. Acesso em 25 maio. 2020.

Breadth-first search algorithm. Disponível em: https://en.wikipedia.org/wiki/Breadth-first_search. Acesso em 25 maio. 2020.

Greedy search algorithm. Disponível em: https://en.wikipedia.org/wiki/Greedy_algorithm. Acesso em 2 junho. 2020.

A* search algorithm. Disponível em: https://en.wikipedia.org/wiki/A* search algorithm. Acesso em 2 junho. 2020.

6. Anexos

6.1. Predicado maisPontosRecolha

```
maisPontosRecolha(Paths, C, NumeroPontosRecolha) :-
    maisPontos(Paths, [], 0, C, NumeroPontosRecolha).

maisPontos([], Caminho, NumeroPontosRecolha, Caminho, NumeroPontosRecolha).
maisPontos([P|Caminhos], MelhorCaminhoAtual, MaiorNrPontosRecolhaAtual, Caminho, NumeroPontosRecolha) :-
    length(P, Tamanho),
    Tamanho > MaiorNrPontosRecolhaAtual,
    maisPontos(Caminhos, P, Tamanho, Caminho, NumeroPontosRecolha).
maisPontos([P|Caminhos], MelhorCaminhoAtual, MaiorNrPontosRecolhaAtual, Caminho, NumeroPontosRecolha) :-
    length(P, Tamanho),
    Tamanho =< MaiorNrPontosRecolhaAtual,
    maisPontos(Caminhos, MelhorCaminhoAtual, MaiorNrPontosRecolhaAtual, Caminho, NumeroPontosRecolha).
```

6.2. Predicado restringeTipo

```
restringeTipo([],_,[]).
restringeTipo([Caminho1], TipoLixo ,[Caminho1]) :-
    temTipo(Caminho1, TipoLixo, Bool),
    Bool == 1, !.
restringeTipo([Caminho1|Caminhos], TipoLixo, [Caminho1|Solucao]) :-
    temTipo(Caminho1, TipoLixo, Bool),
    Bool == 1, !,
    restringeTipo(Caminhos, TipoLixo, Solucao).
restringeTipo([_|Caminhos], TipoLixo, Solucao) :-
    restringeTipo(Caminhos, TipoLixo, Solucao).

temTipo([], _, B) :- B is 1.
temTipo(_,'Garagem',B) :- B is 1.
temTipo(_,'Deposito',B) :- B is 1.
temTipo([Nodo|RestoNodos], TipoLixo, Bool) :-
    findall(Tipo, pontoRecolha(_,_,_,_,Nodo,Tipo,_), Solucao),
    existeTipo(Solucao, TipoLixo, B),
    B == 1,
    temTipo(RestoNodos, TipoLixo, Bool).

existeTipo([],_,0) :- !.
existeTipo([A|B], Tipo, Bool) :-
    A == Tipo,
    Bool is 1, !.
existeTipo([A|B], Tipo, Bool) :-
    A == 'Garagem',
    Bool is 1, !.
existeTipo([A|B], Tipo, Bool) :-
    A == 'Deposito',
    Bool is 1, !.
existeTipo([A|B], Tipo, Bool) :-
    A \= Tipo,
    existeTipo(B, Tipo, Bool).
```

6.3. Predicado maiorLixoRecolhido

```
maiorLixoRecolhido(Paths, C, L) :-
    maisLixo(Paths, [], -1, C, L).

maisLixo([], Caminho, Lixo, Caminho,Lixo).
maisLixo([P|Caminhos], MelhorCaminhoAtual, MenorLixoAtual, Caminho, Total) :-
    lixoRecolhido(P, Lixo),
    Lixo > MenorLixoAtual,
    maisLixo(Caminhos, P, Lixo, Caminho, Total).
maisLixo([P|Caminhos], MelhorCaminhoAtual, MenorLixoAtual, Caminho, Total) :-
    lixoRecolhido(P, Lixo),
    Lixo =< MenorLixoAtual,
    maisLixo(Caminhos, MelhorCaminhoAtual, MenorLixoAtual, Caminho, Total).

lixoRecolhido([], 0).
lixoRecolhido([A|B], Total) :-
    somaLixoCaminho([A|B], 0, Total).

somaLixoCaminho([], A, A).
somaLixoCaminho([A|B], LixoAtual, Total) :-
    findall(L, pontoRecolha(_,_,_,_,A,_,L), Solucao),
    lixoNaLista(Solucao, LixoNoNodo),
    SumLixo is LixoAtual + LixoNoNodo,
    somaLixoCaminho(B, SumLixo, Total).

lixoNaLista([],0).
lixoNaLista([A|B], Total) :-
    somarValores([A|B], 0, Total).

somarValores([], A, A).
somarValores([A|B], LixoAtual, Total) :-
    SumLixo is A + LixoAtual,
    somarValores(B, SumLixo, Total).
```

6.4. Predicado maiorDistancia

```
maiorDistancia(Paths, C, D) :-
    maisLongo(Paths, [], -1, C, D).

maisLongo([], Caminho, Distancia, Caminho, Distancia).
maisLongo([P|Caminhos], MelhorCaminhoAtual, MaiorDistanciaAtual, Caminho, Distancia) :-
    distanciaPercorrida(P, Dist),
    Dist > MaiorDistanciaAtual,
    maisLongo(Caminhos, P, Dist, Caminho, Distancia).
maisLongo([P|Caminhos], MelhorCaminhoAtual, MaiorDistanciaAtual, Caminho, Distancia) :-
    distanciaPercorrida(P, Dist),
    Dist =< MaiorDistanciaAtual,
    maisLongo(Caminhos, MelhorCaminhoAtual, MaiorDistanciaAtual, Caminho, Distancia).

distanciaPercorrida([], 0).
distanciaPercorrida([A|B], Total) :-
    somaDistancia([A|B], A, 0, Total).%saber qual nodo anterior

somaDistancia([], _, D, D).
somaDistancia([A|B], C, DistanciaAtual, Distancia) :-
    distancia(C,A,Dist),
    SumDist is DistanciaAtual + Dist,
    somaDistancia(B, A, SumDist, Distancia).
```

6.5. Predicado menorDistancia

```
menorDistancia(Paths, C, D) :-
    maisCurto(Paths, [], 10000000, C, D).

maisCurto([], Caminho, Distancia, Caminho, Distancia).
maisCurto([P|Caminhos], MelhorCaminhoAtual, MenorDistanciaAtual, Caminho, Distancia) :-
    distanciaPercorrida(P, Dist),
    Dist =< MenorDistanciaAtual,
    maisCurto(Caminhos, P, Dist, Caminho, Distancia).
maisCurto([P|Caminhos], MelhorCaminhoAtual, MenorDistanciaAtual, Caminho, Distancia) :-
    distanciaPercorrida(P, Dist),
    Dist > MenorDistanciaAtual,
    maisCurto(Caminhos, MelhorCaminhoAtual, MenorDistanciaAtual, Caminho, Distancia).
```

6.6. Predicado distancia

```
distancia(Nodo, ProxNodo, DistanciaTotal) :-
    pontoRecolha(Lat1,Long1,_,_,Nodo,_,_),
    pontoRecolha(Lat2,Long2,_,_,ProxNodo,_,_),
    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,
    DistanciaTotal is Dist*1000, !.%para dar em metros
```