

UNIVERSIDADE DO MINHO
DEPARTAMENTO DE INFORMÁTICA
SISTEMAS DE REPRESENTAÇÃO DE
CONHECIMENTO E RACIOCÍNIO

Trabalho Prático Individual

José Martins (A90122)

8 de junho de 2021

Resumo

Este relatório foi realizado no âmbito da disciplina de Sistemas de Representação de Conhecimento e Raciocínio e tem como objetivo apresentar e descrever as soluções implementadas na resolução do projeto individual proposto pela equipa docente. De salientar que toda a implementação foi realizada com recurso à linguagem *Prolog*.

Este documento inicia-se com uma breve introdução, seguida de um conjunto de preliminares que definem o universo sobre o qual o trabalho prático foi desenvolvido, e o método utilizado para a passagem da informação fornecida pelos docentes para a linguagem pretendida. De seguida é feita a descrição do trabalho e análise de resultados, onde se explicam devidamente todos os predicados desenvolvidos e algoritmos desenvolvidos para resolver o problema em questão. Por fim, surge uma breve conclusão com a uma pequena reflexão sobre o trabalho realizado.

Conteúdo

1	Introdução	2
2	Preliminares	3
3	Descrição do Trabalho e Análise de Resultados	9
3.1	Contextualização	9
3.2	Base de Conhecimento	10
3.3	Predicados auxiliares	11
3.4	Formulação do problema	12
3.5	Gerar os circuitos de recolha tanto indiferenciada como seletiva, caso existam, que cubram um determinado território;	12
3.5.1	Gerar os circuitos de recolha indiferenciada	12
3.5.2	Gerar os circuitos de recolha seletiva	14
3.5.3	Análise de Resultados	15
3.6	Identificar quais os circuitos com mais pontos de recolha (por tipo de resíduo a recolher)	17
3.6.1	Análise de Resultados	22
3.7	Escolher o circuito mais rápido (usando o critério da distância)	24
3.7.1	Análise de resultados	29
3.8	Comparar circuitos de recolha tendo em conta os indicadores de produtividade	29
3.8.1	Análise de Resultados	30
3.9	Escolher o circuito mais eficiente (usando um critério de eficiência à escolha)	31
3.9.1	Análise de Resultados	35
3.10	Extra	36
4	Análise comparativa entre as diferentes estratégias de procura	37
5	Conclusão	38

Capítulo 1

Introdução

De forma a incentivar para a utilização da linguagem de programação em lógica *Prolog* e construção de mecanismos de raciocínio para a resolução de problemas, foi-nos proposto um exercício. Este consiste na recolha de resíduos urbanos, constituído por uma garagem, um ponto de despejo e um conjunto de pontos de recolha, do concelho de Lisboa. Este sistema encontra-se disponível em repositório aberto no endereço:

<http://dados.cm-lisboa.pt/no/dataset/circuitos-derecolha-de-residuos-urbanos>.

No contexto de Sistemas de Representação de Conhecimento e Raciocínio, o objetivo principal era desenvolver um programa utilizando o paradigma de programação em lógica para responder a uma série de interrogações sobre caminhos entre uma garagem e um ponto de despejo. Para ser possível a sua realização, foi utilizado o mecanismo de interpretação SWI-Prolog.

Para estabelecer esta pesquisa de caminhos entre uma garagem e um ponto de recolha foram utilizadas duas abordagens: uma com pesquisa informada, que estabelece a procura utilizando algum tipo de informação extra sobre o problema em todas as iterações, a outra com pesquisa não informada, que apenas procura a solução sem saber nada extra sobre o problema em si, apenas para onde tem possibilidade de seguir em cada iteração.

Para comparar os métodos de pesquisa será utilizada uma tabela de comparação e, no final, concluir acerca de qual o mais eficiente em casos mais gerais.

A opção tomada para a realização deste trabalho recaiu sobre a versão mais simplificada.

Capítulo 2

Preliminares

Para ser possível começar a resolução do problema, era necessário saber primeiro do que se tratava a **pesquisa informada** e a **pesquisa não-informada**, pelo que foi necessário um trabalho de pesquisa inicial, que em complemento com as aulas teóricas e práticas ajudou a entender as diferenças entre ambas.

A **pesquisa informada** pretende, de alguma forma, buscar conhecimento específico ao problema para diminuir o tempo de procura e evitar iterações por caminhos que certamente nunca darão corretos, através duma heurística que permita transmitir em cada iteração este conhecimento obtido do problema.

A **pesquisa não informada**, por sua vez necessita de uma heurística que vá procurando as diferentes possibilidades até o estado final ser atingido, porém sem utilizar conhecimento extra sobre o problema em causa.

Após esta fase inicial de pesquisa teórica, foi necessário transferir a informação fornecida pelos docentes para o ambiente onde seria implementado todo o exercício. Foram para isso desenvolvidas três pequenas *scripts* em linguagem de programação python, de forma a facilitar o *parcing* do ficheiro fornecido.

As três *scripts*, criadas de maneira distinta, tem como objetivo criar a Base de Conhecimento constituída e vão ser explicadas e apresentadas de seguida, porém os predicados que derivam do seu output serão melhor explicados apenas no próximo capítulo.

A primeira *script*, que se encontra no ficheiro **LeitorDatasetpontos.py**, diz respeito aos pontos de recolha e demonstra-se de seguida o seu código:

```
import re
import csv

arquivo = open('databasePontos.pl', 'w')

with open('dataset.csv') as ficheiro:
    reader = csv.reader(ficheiro)
    for line in reader:
        latitude = re.sub(r',', '.', line[0])
        longitude = re.sub(r',', '.', line[1])
        sepID = re.split(":", line[4])
        if len(sepID) > 2:
            arquivo.writelines("ponto(" + latitude + "," + longitude + "," + line[2] + "," +
                               + line[3] + "," + sepID[0] + "," + sepID[1] + "," + sepID[2] + "," + line[5] +
                               + "," + line[6] + "," + line[7] + "," + line[8] + "," + line[9] + ").\n")
        elif len(sepID) > 1 and len(sepID) < 2:
            arquivo.writelines("ponto(" + latitude + "," + longitude + "," + line[2] + "," + line[3] +
                               + "," + sepID[0] + "," + sepID[1] + "," + line[5] +
                               + "," + line[6] + "," + line[7] + "," + line[8] + "," + line[9] + ").\n")

arquivo.close()
```

Figura 2.1: Script Python para criação de pontos de recolha

Como podemos observar, este método itera sobre cada uma das linhas do ficheiro CSV, e para cada uma gera um ponto de recolha com toda a informação. Encontra-se de seguida uma amostra do output gerado, que se encontra na sua totalidade conservado no ficheiro **databasepontos.pl**:

```
1 ponto(-9.14330880914792,38.7080787857025,355,'Miseric rdia',15805,' R do
    Alecrim (Par (->)(26->30)', ' R Ferragial - R Ataíde)', 'Lixos', 'CV0090'
    ,90,1,90) .
2 ponto(-9.14330880914792,38.7080787857025,356,'Miseric rdia',15805,' R do
    Alecrim (Par (->)(26->30)', ' R Ferragial - R Ataíde)', 'Lixos', 'CV0240'
    ,240,7,1680) .
3 ponto(-9.14330880914792,38.7080787857025,357,'Miseric rdia',15805,' R do
    Alecrim (Par (->)(26->30)', ' R Ferragial - R Ataíde)', 'Lixos', 'CV0090'
    ,90,1,90) .
4 ponto(-9.14330880914792,38.7080787857025,358,'Miseric rdia',15805,' R do
    Alecrim (Par (->)(26->30)', ' R Ferragial - R Ataíde)', 'Papel e Cartão', '
    CV0240',240,6,1440) .
```

A segunda *script*, que se encontra no ficheiro **LeitorDataSetRuas.py**, diz respeito às ruas. Estas vão corresponder aos nodos do grafo, e acumulam informação sobre os pontos de recolha que as constituem. A cada uma das ruas corresponde um identificador diferente.

```
import re
import csv
#rua(IDrua, latitude, longitude, [contentor_Residuo], soma de contentor_total_litros)
arquivo = open('databaseRuas.pl', 'w')

residuos = []
somaContentor = 0
anterior = None
contador = 0
latitude = None
longitude = None
fsLine = True

with open('dataset.csv') as ficheiro:
    reader = csv.reader(ficheiro)
    for line in reader:
        sepID = re.split(":", line[4])
        if anterior is not None and (anterior==sepID[0]):
            print("passou" + anterior)
            fsLine = False
            residuos.append(line[5])
            somaContentor = somaContentor + int(line[9])

        elif anterior is not None and fsLine:
            fsLine = False
            anterior = sepID[0]
            latitude = re.sub(r',', '.', line[0])
            longitude = re.sub(r',', '.', line[1])
            somaContentor = int(line[9])
            residuos = []
            residuos.append(line[5])
            contador = 1

        else:
            if fsLine:
                anterior = sepID[0]
                print("fsLine")
            else:
                if anterior is not None and longitude is not None and latitude is not None:
                    residuosProlog = re.sub('\n', '\n', str(residuos))
                    arquivo.writelines("rua(" + anterior + "," + latitude + "," + longitude + "," + residuosProlog + "," + str(somaContentor) + ").\n")
                    anterior = sepID[0]
                    latitude = re.sub(r',', '.', line[0])
                    longitude = re.sub(r',', '.', line[1])
                    somaContentor = int(line[9])
                    residuos = []
                    residuos.append(line[5])
                    contador = 1
                residuosProlog = re.sub('\n', '\n', str(residuos))
                arquivo.writelines("rua(" + anterior + "," + latitude + "," + longitude + "," + residuosProlog + "," + str(somaContentor) + ").\n")
            arquivo.close()
```

Figura 2.2: Script Python para criação de ruas

Como podemos observar, este método itera sobre cada uma das linhas do ficheiro CSV, que correspondem a pontos de recolha, e enquanto esses pontos pertencerem à mesma rua, a informação dos mesmos é acumulada de forma a que sejam geradas ruas que contenham informação necessária sobre os pontos que a constituem. Posto isto, uma rua é gerada a partir de de um conjunto de pontos de recolha, que partilham um identificador de rua comum. Para que seja mais perceptível o raciocínio, apresenta-se um exemplo ilustrativo:

Latitude	Longitude	OBJECTID	PONTO_RECOLHA_FREGUESIA	PONTO_RECOLHA_LOCAL	CONTENTOR_RESIDUA*CONTENTOR_TIPO	CONTENTOR_CAP*CONTENTOR_QT	CONTENTOR_TOTAL_LITROS
-9,1433088	38,708079	355	Misericórdia	15805: R do Alecrim (Par (->)>26->30): R Ferragial - R Ataíde)	Lixos	CV0090	90
-9,1433088	38,708079	356	Misericórdia	15805: R do Alecrim (Par (->)>26->30): R Ferragial - R Ataíde)	Lixos	CV0240	240
-9,1433088	38,708079	357	Misericórdia	15805: R do Alecrim (Par (->)>26->30): R Ferragial - R Ataíde)	Lixos	CV0090	90
-9,1433088	38,708079	358	Misericórdia	15805: R do Alecrim (Par (->)>26->30): R Ferragial - R Ataíde)	Papel e Cartão	CV0240	240
-9,1433088	38,708079	359	Misericórdia	15805: R do Alecrim (Par (->)>26->30): R Ferragial - R Ataíde)	Papel e Cartão	CV0090	90

Figura 2.3: Script Python para criação de ruas

Como podemos observar na imagem acima, no *DataSet*, existem pontos de recolha que pertencem à mesma rua, neste caso em concreto à "R do Alecrim". Sendo assim, podemos formar uma rua, que guarde a informação importante sobre esses pontos de recolha, pelo que para o exemplo acima o output gerado pela *script* seria o seguinte:

```
1 rua (15805, -9.14330880914792, 38.7080787857025, ['Lixos ', 'Lixos ', 'Lixos ', 'Papel e Cartão ', 'Papel e Cartão '], 3390).
```

Através da análise, deste predicado rua, que será melhor explicado posteriormente, é fácil identificar que uma rua possui um identificador e coordenadas comuns a todos os pontos de recolha, e em acréscimo possui uma lista com os tipos de resíduo de cada um dos pontos de recolha que a constituem, bem como a quantidade total de lixo que é recolhido nessa rua(correspondente à soma das quantidades de lixo presente em cada ponto).

Finalmente a terceira e última *script*, que se encontra no ficheiro **LeitorDataSetAdjacencias.py**, diz respeito às adjacências entre ruas(ligação entre ruas), permitindo assim gerar o grafo que seria utilizado para a demonstração dos algoritmos implementados. Antes da implementação deste método foi necessário uma análise mais detalhada, para perceber como poderia ser estabelecida a ligação entre as ruas. Com esta análise foi possível perceber que o campo "PONTO_RECOLHA_LOCAL", possuía a informação necessária, para conectar as mesmas. Extraíndo um exemplo do campo mencionado, é possível explicar o procedimento tomado. Posto isto, no exemplo "15805: R do Alecrim (Par (->)(26->30): R Ferragial - R Ataíde)", considerou-se que as ruas adjacentes à "R do Alecrim" eram a "R Ferragial" e "R Ataíde". Em acréscimo, foi também considerado que cada rua teria um adjacente na rua que aparecesse imediatamente antes e depois no *DataSet* fornecido. Posto isto, o algoritmo pode ser dividido em 3 partes que vão ser apresentadas e explicadas de seguida:

```
import pandas as pd
import re

dados=pd.read_csv("dataset.csv", encoding='utf-8', sep="," ,decimal = ".").dropna()

saveFile = open('arestas.pl','w',encoding='utf-8')
valores = []
valores1 = []
valores2 = []
arestas = []

dict = {}
for i in range(0,dados.shape[0]):
    for j in range(0,dados.shape[1]):
        if j == 4:
            valores = str(dados.iloc[i,j]).split(":")
            informacaoRua=valores[1]
            numeroRua = valores[0]
            if informacaoRua not in dict :
                dict[informacaoRua] = numeroRua
```

Figura 2.4: Script Python para criação de grafo

Como podemos ver o excerto de código em cima apenas serviu para relacionar o nome de uma rua ao identificador respetivo.


```

for i in range(0,dados.shape[0]):
    for j in range(0,dados.shape[1]):
        if(j == 4 ) :

            valores1 = str(dados.iloc[i,j]).split(":")
            numeroRual=valores1[0]
            if len(valores1)> 2 :
                ruas = valores1[2].replace(" ","")[0:-1]

                rua = ruas.split("-")
                for k in rua :
                    for d in dict.keys():
                        if re.search(r'\(',d):
                            ruinha = d.split("(")

                        if re.search(r',',d):
                            ruinha = d.split(",")

                        ruaComparacao = ruinha[0].replace(" ","")

                        if k == ruaComparacao:
                            numeroRuaNova = dict[d]
                            if (numeroRual,numeroRuaNova) not in arestas:
                                saveFile.write('aresta(')
                                saveFile.write(str(numeroRual))
                                saveFile.write(', ')
                                saveFile.write(str(numeroRuaNova))
                                saveFile.write(').')
                                saveFile.write('\n')
                                arestas.append((numeroRual,numeroRuaNova))

```

Figura 2.5: Script Python para criação de grafo

Este excerto de código permite ligar as ruas às suas adjacentes, que estão presentes no campo "PONTO_RECOLHA_LOCAL", como explicado em cima.

```

anterior = None
for v in dict.values():
    if anterior is not None:
        if (anterior,v) not in arestas:
            saveFile.write('aresta(')
            saveFile.write(str(anterior))
            saveFile.write(', ')
            saveFile.write(str(v))
            saveFile.write(').')
            saveFile.write('\n')
            arestas.append((anterior,v))
        anterior = v
saveFile.close()

```

Figura 2.6: Script Python para criação de grafo

Nesta última parte do código, é possível verificar que cada rua é conectada à rua anterior e posterior(pela ordem que aparecem no *DataSet* fornecido).

É então possível visualizar uma pequena amostra do output gerado, que se encontra na sua totalidade no ficheiro **arestas.pl**:

```

1 aresta(15805, 21944).
2 aresta(15806, 21944).
3 aresta(15807, 21944).
4 aresta(15807, 15888).
5 aresta(15807, 15889).
6 aresta(15807, 15890).
7 aresta(15807, 15891).
8 aresta(15808, 15892).
9 aresta(15808, 15893).
10 aresta(15808, 15894).
11 aresta(15808, 15809).
12 aresta(15809, 15808).
13 aresta(15809, 15811).
14 aresta(15809, 15812).

```

Por fim, foi finalmente iniciado o trabalho de desenvolvimento da resolução com todos os dados tratados e convertidos no formato final pretendido.

Capítulo 3

Descrição do Trabalho e Análise de Resultados

3.1 Contextualização

O projecto foi desenvolvido com seis ficheiros distintos após a obtenção de dados antes descrita. Estes ficheiros são:

- **databasePontos.pl** - Ficheiro que armazena todos os pontos de recolha
- **databaseRuas.pl** - Ficheiro que armazena todas as ruas
- **arestas.pl** - Ficheiro que armazena todas as arestas de adjacências que constituem o grafo.
- **arestasReduce.pl** - Ficheiro que armazena apenas uma porção das arestas, que fazem parte do grafo. Este ficheiro foi construído de forma a que os limites de *stack* não fossem ultrapassados e fosse então possível testar a maioria dos predicados. A escolha das arestas foi aleatória.
- **arestasPequeno.pl** - Ficheiro que armazena um número reduzido de arestas, para testes e correção de erros, mais eficaz.
- **individual.pl** - Ficheiro com todos as soluções e algoritmos implementados, de forma a responder aos requisitos pedidos.

Tal como pedido no exercício, foram desenvolvidos vários algoritmos em *prolog* de modo a resolver as diferentes alíneas constituintes, nomeadamente:

Pesquisa não informada:

- **Profundidade (DFS - Depth-First Search)** - Na teoria dos grafos, procura em profundidade é um algoritmo usado para realizar uma procura ou travessia numa árvore, estrutura de árvore ou grafo. Este algoritmo começa num nodo, neste caso arbitrário, e explora tanto quanto possível cada um dos seus ramos, antes de retroceder(backtracking).
- **Largura (BFS - Breadth-First Search)** - Na teoria dos grafos, procura em largura é um algoritmo utilizado para realizar uma pesquisa ou travessia num grafo ou estrutura de dados do tipo árvore. Intuitivamente, começa pelo nodo raiz e explora todos os nodos vizinhos. Então, para cada um desses nodos mais próximos, exploramos os seus nodos vizinhos inexplorados e assim por diante, até que ele encontre o estado objetivo(nodo final).
- **Busca Iterativa Limitada em Profundidade** - Na teoria dos grafos, procura iterativa limitada em profundidade segue a mesma ideia que a pesquisa em profundidade, mas em vez de explorar tanto quanto possível um ramo de um nodo arbitrário, explora esse ramo apenas para um profundidade limitada.

Pesquisa informada:

- **Gulosa** - é uma técnica de algoritmos para resolver problemas de optimização, realizando sempre a escolha que parece ser a melhor no momento; fazendo uma escolha ótima local, na esperança de que esta escolha leve até a solução ótima global.
- **A* (A estrela)** - é um algoritmo para procura de Caminho. Ele procura o caminho em um grafo de um nodo inicial até um nodo final e é a combinação de aproximações heurísticas semelhante ao algoritmo *Breadth First* (Procura em Largura) e da formalidade do Algoritmo de *Dijkstra*. Neste contexto, este algoritmo tem como heurística a escolha da menor soma entre o nodo cuja distância do ponto onde estamos e a distancia ao destino.

3.2 Base de Conhecimento

Neste trabalho existem 3 tipos de predicados que representam o conhecimento, sendo eles apresentados a seguir:

- **ponto**: Latitude, Longitude, Id de ponto de recolha, freguesia, Id da rua, informação da rua, ruas adjacentes, tipo de resíduo, tipo de contentor, capacidade de contentor, quantidade de contentores, quantidade de lixo -> {V,F}
- **rua**: Id de rua, Latitude, Longitude, tipos de resíduos, quantidade a recolher -> {V,F}
- **aresta**: Id de rua 1 , Id de rua 2 -> {V,F}

A Base de Conhecimento apresentada será o ponto de partida de todos os algoritmos implementados bem como de todos os testes realizados.

Para que fosse possível usar estes predicados, foi necessário importar os ficheiros na qual estes estavam armazenados, e por isso no ficheiro **individual.pl** foi necessário os seguintes *includes*, bem como a definição dinâmica do número de argumentos de cada um destes predicados:

```
1 :- dynamic(ponto/12).
2 :- dynamic(aresta/2).
3 :- dynamic(rua/5).
4
5 :- include('databasePontos.pl').
6 :- include('databaseRuas.pl').
7 :- include('arestas.pl').
```

No entanto é importante salientar que para todos os testes realizados neste documento se utilizou o ficheiro **arestasReduce.pl**, pois com o ficheiro principal **arestas.pl** não era possível demonstrar todos os resultados pretendidos, por razões que dificilmente se conseguem contornar, nomeadamente o limite da *stack* utilizada para a execução de programas *prolog*.

3.3 Predicados auxiliares

1. **nao**: negação de um argumento.
2. **membro**: Dado um elemento e uma lista verifica se esse elemento pertence à lista.
3. **membros**: predicado auxiliar do predicado **membro**.
4. **solucoes**: Encontra todas as possibilidades utilizando um **findall**.
5. **concat**: concatena listas dadas como argumentos.
6. **escrever,escrever1**: Imprime no terminal uma lista.
7. **eliminaRepetidos**: Dada uma lista, elimina todos os seus elementos repetidos.
8. **eliminaRepAux**: Predicado auxiliar da **eliminaRepetidos**.
9. **primeiroElementoLista**: Retorna o primeiro elemento de uma lista.
10. **ultimoElementoLista**: Retorna o ultimo elemento de uma lista.
11. **removePrimeiro**: Remove o primeiro elemento de uma lista.
12. **seleciona**: Remove um elemento de uma lista.
13. **ordena**: Ordena uma lista, com a ajuda dos predicados auxiliares **distribui** e **intercala**.
14. **topN**:Imprime no terminal um número de elementos definido como argumento.
15. **listingP**: Mostra todos os pontos de recolha da base de conhecimento.
16. **listingR**: Mostra todas as ruas da base de conhecimento.
17. **listingA**: Mostra todas as arestas da base de conhecimento.

3.4 Formulação do problema

Uma vez que o exercício foi feito para 15 valores e não considera que o camião de recolha de lixo tenha uma capacidade limitada, foi necessário definir um conjunto de requisitos iniciais, para que fosse possível resolver o problema em questão.

Primeiramente definiu-se um ponto/estado inicial, que representasse a garagem de onde o camião de recolha de lixo começa o seu percurso. De seguida definiu-se um ponto de recolha do lixo, para o qual o camião se dirigia para despejar todo o lixo recolhido.

Será para esses dois pontos(garagem e ponto de despejo) que serão calculados os percursos através dos algoritmos desenvolvidos para cada uma das alíneas. Sendo assim podemos dizer que o cálculo dos percursos dependerá do nodo inicial e nodo final bem como da condição imposta para comparar os diferentes percursos criados. Uma nota importante a ter em conta é que não são permitidos ciclos, em qualquer algoritmo implementado.

Em baixo apresenta-se a forma como é estabelecido o ponto de partida, correspondente à garagem e o ponto objetivo, correspondente ao ponto de despejo. Sendo assim, como se pode ver usa-se os predicados garagem, e despejo, que permitem estabelecer naturalmente a garagem e o ponto de despejo respetivamente.

```
1 garagem(21944). %define garagem
2 despejo(15808). %define ponto de despejo
```

Cada um dos predicados recebe o identificador de uma rua, que corresponde à rua na qual se encontram localizados os objetos que representam. Para o exemplo dado, a garagem encontra-se na rua cujo identificador é 21944 enquanto que o ponto de despejo se encontra na rua cujo identificador é 15808. De salientar que será para este exemplo que irão ser efetuados todos o testes apresentados neste relatório, de modo a que se possa comparar os diferentes resultados obtidos.

3.5 Gerar os circuitos de recolha tanto indiferenciada como seletiva, caso existam, que cubram um determinado território;

Este requisito será dividido em dois pontos, um para a recolha indiferenciada e outro para a recolha seletiva.

3.5.1 Gerar os circuitos de recolha indiferenciada

Para este ponto, foram gerados circuitos não tendo em conta qualquer tipo de restrição, isto é, apenas foram elaborados algoritmos para gerar circuitos entre a garagem e o ponto de despejo. Só foram, portanto, usados algoritmos de pesquisa não informada, uma vez que uma recolha indiferenciada foi interpretada como uma recolha que não recorre a qualquer tipo de informação dos nodos do grafo. Esta alínea serviu assim de ponto de partida para a implementação mais básica dos algoritmos de pesquisa não informada.

Antes de apresentar os algoritmos criados é importante salientar um predicado que será importante não só nesta alínea mas também nas próximas, que se designa por **adjacente**. Como se pode ver em baixo, este permite obter os nodos adjacentes a qualquer nodo.

```
1 adjacente(X,Y) :- aresta(X,Y).
2 adjacente(X,Y) :- aresta(Y,X).
```

Posto isto passamos à apresentação dos algoritmos criados e sua especificação:

Pesquisa ineficiente:

Foi desenvolvido primeiro um algoritmo simples, implementado numa das aulas, que apenas procura um caminho entre dois nodos através da visita a todos os nodos. Este método é muito ineficiente, daí a não ser utilizado futuramente. O predicado **resolve_ineficiente** permite obter o caminho entre a garagem e o ponto de despejo e o predicado **todos_ineficiente** permite obter todos os caminhos possíveis. Como era de esperar ambos os predicados utilizam o algoritmo desenvolvido.

```
1 caminho(A,B,P):-caminho1(A,[B],P).
2 caminho1(A,[A|P1],[A|P1]).
3 caminho1(A,[Y|P1],P):-adjacente(X,Y),
4                           nao(membro(X,[Y|P1])),
5                           caminho1(A,[X,Y|P1],P).
6
7 resolve_ineficiente(Cam):-garagem(G),despejo(D),caminho(G,D,Cam).
8
9 todos_ineficiente(R):-findall((Cam),resolve_ineficiente(Cam),R).
```

Pesquisa em profundidade:

Partimos então para o desenvolvimento de um algoritmo Depth-First de procura de caminhos entre dois nodos. Este algoritmo é muito mais eficiente do que o método anterior, e procura eficientemente um caminho entre dois nodos, no entanto há determinados conjuntos de procura que expandem a procura a um nível muito elevado e o programa não encontra o caminho. O predicado **resolve_profundidade_Multi_1** permite obter o caminho entre a garagem e o ponto de despejo e o predicado **todos_profundidade_Multi_1** permite obter todos os caminhos possíveis, enquanto que o predicado **resolve_pp_h** corresponde ao algoritmo em questão.

```
1 resolve_pp_h(NodoInicial, NodoFinal, Caminho):-
2     profundidadeH(NodoInicial, NodoFinal, [NodoInicial], Caminho).
3
4 profundidadeH(Destino, Destino, H, D):-reverse(H, D).
5 profundidadeH(Origem, Destino, Historico, C):-
6     adjacente(Origem, Prox),
7     nao(membro(Prox, Historico)),
8     profundidadeH(Prox, Destino, [Prox|Historico], C).
9
10 resolve_profundidade_Multi_1(Cam):-
11     garagem(G),despejo(D),
12     resolve_pp_h(G, D, Cam).
13
14 todos_profundidade_Multi_1(R):-
15     findall((Cam),resolve_profundidade_Multi_1(Cam),R).
```

Pesquisa em largura:

Foi desenvolvido também um algoritmo Breadth-First de procura de caminhos entre dois nodos. Este algoritmo é menos eficiente que o método anterior, pois expande todos os nós adjacentes demorando muito mais tempo, e ocupando muito mais memória. Há portanto neste também determinados conjuntos de procura para o qual o programa não consegue encontrar o caminho. O predicado **resolve_Largura_1** permite obter o caminho entre a garagem e o ponto de despejo. Este utiliza o predicado **expande** como auxiliar, que é responsável por expandir um nodo nos seus adjacentes, retornando os caminhos possíveis a partir do mesmo. Para cada um desses caminhos é verificado se é possível atingir o nodo objetivo. Se não for possível segue-se uma nova iteração.

O predicado **todos_Largura_1** permite obter todos os caminhos possíveis, enquanto que o predicado **largura** corresponde ao algoritmo em questão.

```

1 largura(Inicio,Fim,Caminho):-
2     bfs([[Inicio]],Fim,CaminhoAux),
3     reverse(CaminhoAux,Caminho).
4
5 bfs([[Fim|Caminho]|_],Fim,[Fim|Caminho]).
6 bfs([Caminho1|Caminhos],Fim,Caminho):-
7     expande(Caminho1,NovosCaminhos),
8     append(Caminhos,NovosCaminhos,Caminhos1),
9     bfs(Caminhos1,Fim,Caminho).
10
11 expande([Nodo|Caminho],NovosCaminhos):-
12     findall([Nodo2,Nodo|Caminho],(adjacente(Nodo,Nodo2),\+ memberchk(Nodo2
13         ,[Nodo|Caminho])),NovosCaminhos),!.
14
15 resolve_Largura_1(Cam):-garagem(G),despejo(D),largura(G,D,Cam).
16
17 todos_Largura_1(R):-findall((Cam),resolve_Largura_1(Cam),R).

```

Pesquisa Iterativa Limitada em Profundidade:

Finalmente, foi desenvolvido um algoritmo de pesquisa iterativa limitada em profundidade de procura de caminhos entre dois nodos. Este algoritmo deriva da procura simples em profundidade, seguindo a mesma metodologia, mas procura evitar os casos em que a mesma fica presa em ramos muito grandes ou infinitos. Este algoritmo permite estabelecer um limite para o número de nodos percorridos em profundidade de um ramo. O predicado **resolve_prof_lim_1** permite obter o caminho entre a garagem e o ponto de despejo e o predicado **todos_prof_lim** permite obter todos os caminhos possíveis, enquanto que o predicado **resolve_prof_lim** corresponde ao algoritmo em questão.

```

1 resolve_prof_lim(Nodo,Caminho,L):-
2     profundidade_limitada([],Nodo,InvCaminho,L),
3     reverse(InvCaminho,Caminho).
4
5 profundidade_limitada(Caminho,Nodo,[Nodo|Caminho],_-):-despejo(Nodo).
6 profundidade_limitada(Caminho,Nodo,S,L):-
7     L>0, adjacente(Nodo,NodoL),
8     \+ member(NodoL,Caminho),
9     L1 is L-1,
10    profundidade_limitada([Nodo|Caminho],NodoL,S,L1).
11
12 resolve_prof_lim_1(Cam,L):-garagem(G),resolve_prof_lim(G,Cam,L).
13
14 todos_prof_lim(R,L):-findall((Cam),resolve_prof_lim_1(Cam,L),R).

```

3.5.2 Gerar os circuitos de recolha seletiva

Este requisito foi tido em conta nas alíneas posteriores, uma vez que para que se faça uma recolha seletiva é necessário estabelecer um ou mais critérios de procura, o que até ao momento não foi feito. Sendo assim nos capítulos seguintes deste relatório, serão apresentados algoritmos que permitem uma recolha seletiva, baseado nos critérios propostos pelos docentes.

3.5.3 Análise de Resultados

```
?- resolve ineficiente(Cam),escrever(Cam).
21944
15805
15806
15807
15888
15812
15810
15811
15809
15808
Cam = [21944, 15805, 15806, 15807, 15888, 15812, 15810, 15811, 15809|...] .
```

Figura 3.1: Teste do algoritmo ineficiente

```
?- todos ineficiente(R),escrever(R).
[21944,15805,15806,15807,15888,15812,15810,15811,15809,15808]
[21944,15806,15807,15888,15812,15810,15811,15809,15808]
[21944,15807,15888,15812,15810,15811,15809,15808]
[21944,15805,15806,15807,15889,15812,15810,15811,15809,15808]
[21944,15806,15807,15889,15812,15810,15811,15809,15808]
[21944,15807,15889,15812,15810,15811,15809,15808]
[21944,15805,15806,15807,15890,15812,15810,15811,15809,15808]
[21944,15806,15807,15890,15812,15810,15811,15809,15808]
[21944,15807,15890,15812,15810,15811,15809,15808]
[21944,15805,15806,15807,15891,15812,15810,15811,15809,15808]
[21944,15806,15807,15891,15812,15810,15811,15809,15808]
[21944,15807,15891,15812,15810,15811,15809,15808]
[21944,15805,15806,15807,15888,15812,15811,15809,15808]
[21944,15806,15807,15888,15812,15811,15809,15808]
[21944,15807,15888,15812,15811,15809,15808]
```

Figura 3.2: Amostra dos caminhos encontrados pela pesquisa ineficiente

```
?- resolve profundidade Multi 1(Cam).
Cam = [21944, 15805, 15806, 15807, 15888, 15812, 15809, 15808] .

?- todos profundidade Multi 1(R),escrever(R).
[21944,15805,15806,15807,15888,15812,15809,15808]
[21944,15805,15806,15807,15888,15812,15809,15811,15892,15808]
[21944,15805,15806,15807,15888,15812,15809,15811,15893,15808]
[21944,15805,15806,15807,15888,15812,15809,15811,15894,15808]
[21944,15805,15806,15807,15888,15812,15809,15808]
[21944,15805,15806,15807,15888,15812,15809,15811,15892,15808]
[21944,15805,15806,15807,15888,15812,15809,15811,15893,15808]
[21944,15805,15806,15807,15888,15812,15809,15811,15894,15808]
[21944,15805,15806,15807,15888,15812,15809,15808]
[21944,15805,15806,15807,15888,15812,15809,15811,15892,15808]
[21944,15805,15806,15807,15888,15812,15809,15811,15893,15808]
[21944,15805,15806,15807,15888,15812,15809,15811,15894,15808]
[21944,15805,15806,15807,15888,15812,15809,15808]
```

Figura 3.3: Teste da pesquisa em profundidade e amostra dos caminhos encontrados

```

?- resolve prof lim 1(Cam,4).
Cam = [21944, 15805, 15806, 15807, 15808] .

?- resolve prof lim 1(Cam,3).
Cam = [21944, 15806, 15807, 15808] .

?- resolve prof lim 1(Cam,2).
Cam = [21944, 15807, 15808] .

?- resolve prof lim 1(Cam,1).
false.

```

Figura 3.4: Teste da pesquisa em profundidade limitada, com limite 4,3,2,1

```

?- todos prof lim(R,4).
R = [[21944, 15805, 15806, 15807, 15808], [21944, 15806, 15807, 15808], [21944, 15807, 15808]].

?- todos prof lim(R,3).
R = [[21944, 15806, 15807, 15808], [21944, 15807, 15808]].

?- todos prof lim(R,2).
R = [[21944, 15807, 15808]].

?- todos prof lim(R,1).
R = [].

```

Figura 3.5: Amostra dos caminhos encontrados pela pesquisa em profundidade limitada

```

?- resolve Largura 1(Cam).
Cam = [21944, 15807, 15808] .

?- todos Largura 1(R),escrever(R).
[21944,15805,15806,15807,15891,15812,15810,15811,15809,15808]
[21944,15805,15806,15807,15891,15812,15810,15811,15894,15808]
[21944,15805,15806,15807,15891,15812,15810,15811,15893,15808]
[21944,15805,15806,15807,15891,15812,15810,15811,15892,15808]
[21944,15805,15806,15807,15891,15812,15809,15811,15894,15808]
[21944,15805,15806,15807,15891,15812,15809,15811,15893,15808]
[21944,15805,15806,15807,15891,15812,15809,15811,15892,15808]
[21944,15805,15806,15807,15890,15812,15810,15811,15809,15808]
[21944,15805,15806,15807,15890,15812,15810,15811,15894,15808]
[21944,15805,15806,15807,15890,15812,15810,15811,15893,15808]
[21944,15805,15806,15807,15890,15812,15810,15811,15892,15808]
[21944,15805,15806,15807,15890,15812,15809,15811,15894,15808]
[21944,15805,15806,15807,15890,15812,15809,15811,15893,15808]
[21944,15805,15806,15807,15890,15812,15809,15811,15892,15808]
[21944,15805,15806,15807,15889,15812,15810,15811,15809,15808]

```

Figura 3.6: Teste da pesquisa em largura

Algoritmo	Primeiro Caminho Encontrado
Ineficiente	[21944,15805,15806,15807,15888,15812,15810,15811,15809,15808]
Profundidade	[21944,15805,15806,15807,15888,15812,15809,15808]
Profundidade Limitada(Lim = 4)	[21944,15805,15806,15807,15808]
Profundidade Limitada(Lim = 3)	[21944,15806,15807,15808]
Profundidade Limitada(Lim = 2)	[21944,15807,15808]
Largura	[21944,15807,15808]

Tabela 3.1: Comparação de primeiros caminhos encontrados

3.6 Identificar quais os circuitos com mais pontos de recolha (por tipo de resíduo a recolher)

Para esta alínea, o objetivo é encontrar o caminho entre a garagem e o ponto de despejo, que permita passar por mais pontos de recolha de um determinado tipo de lixo. Nesta foram utilizados todos os algoritmos de pesquisa não informada, e o algoritmo Gulosa de pesquisa informada, para que se possa comparar os resultados obtidos por cada um destes.

Antes de passar à apresentação e análise de cada um dos algoritmos implementados, é importante mencionar alguns predicados que foram elaborados para ajudar tanto na construção dos mesmos, como para fornecer ao utilizador a oportunidade de ter acesso a mais algumas informações.

Predicado que dado um identificador de rua, devolve o número de pontos de recolha total da mesma.

```

1 %total de pontos de recolha de uma determinada rua
2 numeroPR_Rua(IdRua,R) :- rua(IdRua,Lat,Long,Res,Qt),length(Res,R).
```

Predicado que dado um identificador de rua, devolve a lista de pontos de recolha que a constituem.

```

1 %Dada uma rua, devolve a lista de pontos de recolha da mesma
2 listaDePontos_Rua(IdRua,R) :-
3 findall(ponto(Lat1,Long1,Id1,PRF1,IdRua1,PRL11,PRL12,CR1,CT1,CC1,CQ1,CTL1),
4 (ponto(Lat1,Long1,Id1,PRF1,IdRua1,PRL11,PRL12,CR1,CT1,CC1,CQ1,CTL1),
5 IdRua=IdRua1),R).
```

Predicado que dado um identificador de rua e um tipo de resíduo, devolve o número total de pontos de recolha desse determinado resíduo para a rua em questão.

```

1 %total de pontos de recolha de um determinado resíduo para uma determinada
  rua
2 numeroPR_R_Rua(IdRua,TResiduo,R):-
3     listaDePontos_R_Rua(IdRua,L),
4     quantosResiduoPonto(TResiduo,L,R).
```

Predicado que dado um identificador de rua, devolve a lista de tipos de resíduos a recolher da mesma.

```

1 %Dado um ID de uma rua, devolve a lista de res duos da mesma
2 listaDePontos_R_Rua(IdRua,Res) :- rua(IdRua,Lat,Long,Res,Qt)
```

Predicado que dada uma lista de resíduos e um tipo de resíduo devolve o número de elementos dessa lista que são do mesmo tipo do resíduo dado.

```

1 %conta o numero de elementos de um determinado tipo de residuos na lista de
  residuos
2 quantosResiduoPonto(TResiduo,[ ],0).
3 quantosResiduoPonto(TResiduo,[TResiduo|T],L):-
4     quantosResiduoPonto(TResiduo,T,N1),
5     L is N1 + 1.
6 quantosResiduoPonto(TResiduo,[X|T],L):-
7     X \=TResiduo,
8     quantosResiduoPonto(TResiduo,T,L).

```

Finalmente podemos passar para a apresentação e explicação dos algoritmos desenvolvidos.

Pesquisa não informada em profundidade:

Neste algoritmo, basicamente foi aplicado o algoritmo de pesquisa em profundidade entre dois nodos apresentado anteriormente, porém neste caso para cada nodo ultrapassado contabiliza-se o número de pontos de recolha que cada um tem para um determinado tipo de resíduo. Assim no algoritmo começa-se por calcular a quantidade de pontos de recolha da rua da garagem somando-se a esse valor o número de pontos de recolha de cada uma das ruas em que o camião passa. Uma vez atingido o nodo final(ponto de despejo), obtêm-se não só o caminho mas também a quantidade de lixo recolhida ao longo do mesmo. O predicado que aplica este algoritmo é o **resolve_Profundidade_2**.

Porém sabemos que este algoritmo não é otimal pelo que temos de obter todos os caminhos e compará-los para ver qual o melhor. Para isto foram então desenvolvidos os predicados **todos_Caminhos_TResiduo_Profundidade** que calcula todos os caminhos da garagem até ao ponto de despejo e associa a cada um deles o número de pontos de recolha total, **caminhosOrdena_TResiduo_Profundidade** que ordena todos os caminhos anteriormente calculados por número de pontos de recolha e por fim **bestCaminho_TResiduo_Profundidade**, que dá como resultado o primeiro elemento da lista anteriormente ordenada, correspondente portanto ao caminho com mais pontos de recolha.

```

1 %Dado um ponto inicial e final devolve um caminho e a quantidade de pontos
  de recolha do mesmo para um tipo de residuo
2 resolve_pp_h_PR(NodoInicial, NodoFinal, TResiduo, Caminho/NrPR):-
3     profundidade_PR(NodoInicial, NodoFinal, TResiduo, [NodoInicial],
4         Caminho, NrPR).
5
6 profundidade_PR(Destino, Destino, TResiduo, H,D,NrPR):-
7     reverse(H,D),
8     numeroPR_R_Rua(Destino, TResiduo, NrPR).
9 profundidade_PR(Origem, Destino, TResiduo, Historico, Caminho, NrPR):-
10    numeroPR_R_Rua(Origem, TResiduo, NrPR1),
11    adjacente(Origem, Prox),
12    nao(membro(Prox, Historico)),
13    profundidade_PR(Prox, Destino, TResiduo, [Prox|Historico], Caminho,
14        NrPR2),
15    NrPR is NrPR1+NrPR2.
16
17 resolve_Profundidade_2(TResiduo,Cam):-
18    garagem(G), despejo(D),
19    resolve_pp_h_PR(G,D, TResiduo, Cam).
20
21 %Processo para apresentar o melhor caminho
22 todos_Caminhos_TResiduo_Profundidade(TResiduo,X):-

```

```

21      findall((Cam,NrPR), resolve_Profundidade_2(TResiduo,Cam/NrPR),R),
22      eliminaRepetidos(R,X).
23
24 caminhosOrdena_TResiduo_Profundidade(TResiduo,S):-
25     todos_Caminhos_TResiduo_Profundidade(TResiduo,R),ordena(R,S).
26
27 %Devolve o caminho com mais pontos de recolha para um determinado residuo
28 bestCaminho_TResiduo_Profundidade(TResiduo,Best):-
29     caminhosOrdena_TResiduo_Profundidade(TResiduo,S),
30     primeiroElementoLista(S,Best).

```

Pesquisa não informada em profundidade limitada:

Para este algoritmo, seguiu-se exatamente a mesma estratégia que na pesquisa em profundidade simples, no entanto utilizando-se como era de esperar uma estratégia de pesquisa limitada. O predicado que aplica este algoritmo é o **resolve_prof_lim_2**.

Sabemos também que este algoritmo não é ótimo pelo que temos também de obter todos os caminhos e compará-los para ver qual o melhor. Para isto foram então desenvolvidos os predicados **todos_Caminhos_TResiduo_Prof_Lim** que calcula todos os caminhos da garagem até ao ponto de despejo e associa a cada um deles o número de pontos de recolha total, **caminhosOrdena_TResiduo_Prof_Lim** que ordena todos os caminhos anteriormente calculados por número de pontos de recolha e por fim **bestCaminho_TResiduo_Prof_Lim**, que dá como resultado o primeiro elemento da lista anteriormente ordenada, correspondente portanto ao caminho com mais pontos de recolha.

```

1 resolve_prof_lim_Res(Nodo,TResiduo,Caminho/NrPR,L):-
2     profundidade_limitada([],Nodo,TResiduo,
3     InvCaminho/NrPR,L),
4     reverse(InvCaminho,Caminho).
5
6 profundidade_limitada(Caminho,Nodo,TResiduo,[Nodo|Caminho]/NrPR,-):-
7     despejo(Nodo),numeroPR_Rua(Nodo,TResiduo,NrPR).
8 profundidade_limitada(Caminho,Nodo,TResiduo,S/NrPR,L):-
9     numeroPR_Rua(Nodo,TResiduo,NrPR1),
10    L>0, adjacente(Nodo,NodoL),
11    \+ member(NodoL,Caminho),
12    L1 is L-1,
13    profundidade_limitada([Nodo|Caminho],NodoL,TResiduo,S/NrPR2,L1),
14    NrPR is NrPR1+NrPR2.
15
16 resolve_prof_lim_2(TResiduo,L,Cam):-
17     garagem(G),despejo(D),
18     resolve_prof_lim_Res(G,TResiduo,Cam,L).
19
20 %Processo para apresentar o melhor caminho
21 todos_Caminhos_TResiduo_Prof_Lim(TResiduo,L,X):-
22     findall((Cam,NrPR), resolve_prof_lim_2(TResiduo,L,Cam/NrPR),R),
23     eliminaRepetidos(R,X).
24
25 caminhosOrdena_TResiduo_Prof_Lim(TResiduo,L,S):-
26     todos_Caminhos_TResiduo_Prof_Lim(TResiduo,L,R),
27     ordena(R,S).
28
29 %Devolve o caminho com mais pontos de recolha para um determinado residuo
30 bestCaminho_TResiduo_Prof_Lim(TResiduo,L,Best):-
31     caminhosOrdena_TResiduo_Prof_Lim(TResiduo,L,S),
32     primeiroElementoLista(S,Best).

```

Pesquisa não informada em largura:

Mais uma vez se usou a mesma estratégia que na pesquisa em profundidade simples, no entanto utilizando-se como era de esperar uma estratégia de pesquisa em largura, mais uma vez semelhante à explicada na alínea anterior mas com o agravante de contabilizar os pontos de resíduo de cada uma das ruas por onde passa o caminhão de recolha. O predicado que aplica este algoritmo é o **resolve_largura_2**.

Sabemos também que este algoritmo não é ótimo (neste caso) pelo que temos também de obter todos os caminhos e compará-los para ver qual o melhor. Para isto foram então desenvolvidos os predicados **todos_Caminhos_TResiduo_Largura** que calcula todos os caminhos da garagem até ao ponto de despejo e associa a cada um deles o número de pontos de recolha total, **caminhosOrdena_TResiduo_Largura** que ordena todos os caminhos anteriormente calculados por número de pontos de recolha e por fim **bestCaminho_TResiduo_Largura**, que dá como resultado o primeiro elemento da lista anteriormente ordenada, correspondente portanto ao caminho com mais pontos de recolha.

```
1 largura2 (Inicio ,Fim,TResiduo ,Caminho/PR) :-
2     numeroPR_R_Rua (Inicio ,TResiduo ,Estima) ,
3     bfs2 (TResiduo ,[[ Inicio ]/Estima] ,Fim,CaminhoAux/PR) ,
4     reverse (CaminhoAux,Caminho) .
5
6 bfs2 (TResiduo ,[[ Fim|Caminho]/Pr|-] ,Fim, [Fim|Caminho]/Pr) .
7 bfs2 (TResiduo ,[Caminho1|Caminhos] , Fim, Caminho) :-
8     expande (TResiduo,Caminho1 , NovosCaminhos) ,
9     append (Caminhos , NovosCaminhos , Caminhos1) ,
10    bfs2 (TResiduo ,Caminhos1 , Fim, Caminho) .
11
12 expande (TResiduo ,Caminho1 , NovosCaminhos) :-
13     findall (NovoCaminho ,adjacenteLargura (TResiduo ,Caminho1 ,NovoCaminho) ,
14             NovosCaminhos) ,!.
15
16 adjacenteLargura (TResiduo ,[Nodo|Caminho]/PR1, [ProxNodo,Nodo|Caminho]/PRT):-
17     adjacente (Nodo,ProxNodo) ,
18     nao (membro (ProxNodo ,Caminho)) ,
19     numeroPR_R_Rua (ProxNodo ,TResiduo ,PRNext) ,
20     PRT is PR1 + PRNext .
21
22 resolve_largura_2 (TResiduo ,R):-
23     garagem (G) ,despejo (D) ,
24     largura2 (G,D,TResiduo ,R) .
25
26 %Processo para apresentar o melhor caminho
27 todos_Caminhos_TResiduo_Largura (TResiduo ,X):-
28     findall ((Cam,NrPR) , resolve_largura_2 (TResiduo ,Cam/NrPR) ,R) ,
29     eliminaRepetidos (R,X) .
30
31 caminhosOrdena_TResiduo_Largura (TResiduo ,S):-
32     todos_Caminhos_TResiduo_Largura (TResiduo ,R) ,ordena (R,S) .
33
34 %Develve o caminho com mais pontos de recolha para um determinado residuo
35 bestCaminho_TResiduo_Largura (TResiduo ,Best):-
36     caminhosOrdena_TResiduo_Largura (TResiduo ,S) ,
37     primeiroElementoLista (S,Best) .
```

Pesquisa informada Gulosa:

Aqui surge a primeira implementação de um algoritmo de pesquisa informada, neste caso do tipo gulosa. Assim a heurística utilizada foi a seguinte: em cada iteração, a partir do nodo/rua em que o camião se localiza, são obtidos todas as ruas adjacentes calculando para cada uma delas o número de pontos de recolha para um determinado resíduo. Sabendo isto, o algoritmo irá escolher deslocar-se para o nodo que tem maior número de pontos de recolha, guardando sempre em memória os caminhos alternativos que não foram escolhidos, para que caso chegue a um nodo em que seja impossível chegar ao nodo final(exemplo:estrada sem saída), este possa voltar atrás e escolher o segundo melhor caminho. O predicado que aplica este algoritmo é o **resolve_Gulosa_2**.

Percebemos facilmente que este algoritmo não é ótimo, pelo que temos também de obter todos os caminhos e compará-los para ver qual o melhor, tal como nos algoritmos anteriores. Para isto, foram então desenvolvidos os predicados **todos_Caminhos_TResiduo_Gulosa** que calcula todos os caminhos da garagem até ao ponto de despejo e associa a cada um deles o número de pontos de recolha total, **caminhos_Ordena_TResiduo_Gulosa** que ordena todos os caminhos anteriormente calculados por número de pontos de recolha e por fim **bestCaminho_Residuo_Gulosa**, que dá como resultado o primeiro elemento da lista anteriormente ordenada, correspondente portanto ao caminho com mais pontos de recolha.

```
1 %soma a estima de um caminho aos seus caminhos adjacentes
2 aumentaEstima(Cam, [], []) .
3 aumentaEstima(_/Estima1, [Nodos/Estima2 | Tail], [Nodos/Estima3 |
   EstimasAumentadas]):-
4     Estima3 is Estima2 + Estima1,
5     aumentaEstima(_/Estima1, Tail, EstimasAumentadas) .
6
7
8 %TResiduo -> Tipo de Residuo
9 %Caminho/PR -> caminho até ao ponto de Residuo / numero de pontos de recolha
   total de um determinado residuo para o caminho
10 %GULOSA GARAGEM -> PONTO DE DESPEJO
11 resolve_gulosa_Ida(Nodo, TResiduo, Caminho/PR):-
12     numeroPR_R_Rua(Nodo, TResiduo, Estima),
13     agulosa_Ida(TResiduo, [[Nodo]/Estima], InvCaminho/PR),
14     reverse(InvCaminho, Caminho) .
15
16 agulosa_Ida(TResiduo, Caminhos, Caminho):-
17     obtem_melhor_g_Ida(Caminhos, Caminho),
18     Caminho = [Nodo|_]/_, despejo(Nodo) .
19
20 agulosa_Ida(TResiduo, Caminhos, SolucaoCaminho):-
21     obtem_melhor_g_Ida(Caminhos, MelhorCaminho),
22     remove(MelhorCaminho, Caminhos, OutrosCaminhos),
23     expande_gulosa(TResiduo, MelhorCaminho, ExpCaminhos),
24     aumentaEstima(MelhorCaminho, ExpCaminhos, AltCaminhos),
25     append(OutrosCaminhos, AltCaminhos, NovoCaminhos),
26     agulosa_Ida(TResiduo, NovoCaminhos, SolucaoCaminho) .
27
28 obtem_melhor_g_Ida([Caminho], Caminho):-!.
29
30 obtem_melhor_g_Ida([Caminho1/Est1, _/Est2 | Caminhos], MelhorCaminho):-
31     Est2 <= Est1,!,
32     obtem_melhor_g_Ida([Caminho1/Est1 | Caminhos], MelhorCaminho) .
33
34 obtem_melhor_g_Ida([_|Caminhos], MelhorCaminho):-
35     obtem_melhor_g_Ida(Caminhos, MelhorCaminho) .
36
37 expande_gulosa(TResiduo, Caminho, ExpCaminhos):-
```

```

38         findall(NovoCaminho, adjacenteGula(TResiduo, Caminho, NovoCaminho),
39               ExpCaminhos).
39
40 adjacenteGula(TResiduo, [Nodo|Caminho]/-, [ProxNodo, Nodo|Caminho]/Estima):-
41     adjacente(Nodo, ProxNodo),
42     nao(membro(ProxNodo, Caminho)),
43     numeroPR_Rua(ProxNodo, TResiduo, Estima).
44
45
46 resolve_Gulosa_2(TResiduo, Cam):-
47     garagem(G), resolve_gulosa_Ida(G, TResiduo, Cam).
48
49 %Processo para descobrir o caminho com mais pontos de recolha
50 todos_Caminhos_TResiduo_Gulosa(TResiduo, X):-
51     findall((Cam, NrPR), resolve_Gulosa_2(TResiduo, Cam/NrPR), R),
52     eliminaRepetidos(R, X).
53
54 caminhos_Ordena_TResiduo_Gulosa(TResiduo, S):-
55     todos_Caminhos_TResiduo_Gulosa(TResiduo, X),
56     ordena(X, S).
57
58 bestCaminho_Residuo_Gulosa(TResiduo, Best):-
59     caminhos_Ordena_TResiduo_Gulosa(TResiduo, S),
60     primeiroElementoLista(S, Best).

```

3.6.1 Análise de Resultados

Demonstra-se agora os resultados obtidos pelos predicados desenvolvidos para esta alínea. Para cada um dos tipos de pesquisa é demonstrado o primeiro caminho encontrado, os 10 melhores caminhos e o melhor caminho de todos os que são possíveis de encontrar (caminho com maior número de pontos de recolha). De notar que todos os predicados recebem como parâmetro o tipo de resíduo para o qual se quer saber o número de pontos de recolha, sendo que neste caso os testes foram feitos para o resíduo 'Lixos'. No caso da pesquisa informada Gulosa, não foi possível apresentar os resultados, para o top 10 de caminhos nem para o melhor caminho, por ser um algoritmo que possui um número enorme de possibilidades de caminhos, demorando assim algum tempo até que seja finalizado.


```

?- resolve largura 2('Lixos',R).
R = [21944, 15807, 15808]/8 .

?- caminhosOrdena TResiduo Largura('Lixos',S),topN(S,10).
[21944,15805,15806,15807,15889,15812,15810,15811,15893,15808],43
[21944,15805,15806,15807,15889,15812,15809,15811,15893,15808],43
[21944,15805,15806,15807,15890,15812,15809,15811,15893,15808],41
[21944,15805,15806,15807,15889,15812,15811,15893,15808],41
[21944,15805,15806,15807,15889,15812,15809,15811,15894,15808],41
[21944,15805,15806,15807,15890,15812,15810,15811,15893,15808],41
[21944,15805,15806,15807,15889,15812,15810,15811,15894,15808],41
[21944,15806,15807,15889,15812,15810,15811,15893,15808],40
[21944,15806,15807,15889,15812,15809,15811,15893,15808],40
[21944,15805,15806,15807,15890,15812,15810,15811,15894,15808],39
S = [[([21944, 15805, 15806, 15807, 15889, 15812, 15810|...], 43), ([21944, 15805, 15806, 15807|...], 41), ([21944, 15805, 15806|...], 41), ([21944, 15805, 15806, 15807|...], 41), ([21944, 15805, 15806, 15807|...], 41), ([21944, 15805, 15806, 15807|...], 41), ([21944, 15805, 15806, 15807|...], 41), ([21944, 15805, 15806, 15807|...], 41), ([21944, 15805, 15806, 15807|...], 41), ([21944, 15805, 15806, 15807|...], 41)].

?- bestCaminho TResiduo Largura('Lixos',(Best,N)),escrever(Best).
21944
15805
15806
15807
15889
15812
15810
15811
15893
15808
Best = [21944, 15805, 15806, 15807, 15889, 15812, 15810, 15811, 15893|...],
N = 43 .

```

Figura 3.9: Teste da pesquisa em largura

```

?- resolve Gulosa 2('Lixos',Cam).
Cam = [21944, 15806, 15807, 15889, 15812, 15811, 15893, 15808]/38 .

```

Figura 3.10: Teste da pesquisa Gulosa

3.7 Escolher o circuito mais rápido (usando o critério da distância)

Para esta alínea, o objetivo é encontrar o caminho mais curto entre a garagem e o ponto de despejo, isto é o caminho que permita ao caminhão percorrer uma menor distância. Nesta, no que toca a algoritmos de pesquisa não informada foi utilizado o algoritmo de pesquisa em profundidade, e em relação a pesquisa informada, foram utilizados os algoritmos da Gulosa e A*(A estrela).

Antes de passar à apresentação e análise de cada um dos algoritmos implementados, é importante mencionar alguns predicados que foram elaborados para ajudar tanto na construção dos mesmos, como para fornecer ao utilizador a oportunidade de ter acesso a mais algumas informações.

Predicado que calcula a distancia entre duas coordenadas geográficas.

```

1 %distancia entre duas coordenadas
2 distancia(N1,N2,N3,N4,R):- N is sqrt((N3-N1)^2+(N4-N2)^2),N=R.

```

Predicado que calcula a distancia entre duas ruas, dados os seus identificadores como argumentos.

```

1 %distancia entre duas ruas pelos seus IDs
2 distanciaEntreRuas_ID (IdRua1, IdRua2, Dist):-
3     rua (IdRua1, Lat1, Long1, Res1, Qt1) ,
4     rua (IdRua2, Lat2, Long2, Res2, Qt2) ,
5     distancia (Lat1, Long1, Lat2, Long2, Dist) .

```

Predicado que calcula a distância entre duas ruas, dadas essas ruas como argumentos.

```

1 %distancia entre duas ruas(nodos)
2 distanciaEntreRuas (rua (IdRua1, Lat1, Long1, Res1, Qt1) , rua (IdRua2, Lat2, Long2,
3     Res2, Qt2) , R):- distancia (Lat1, Long1, Lat2, Long2, W) , R is W.

```

Predicado que dado um caminho só de ida de entre dois nodos, constrói o caminho de ida e volta, refletindo o primeiro.

```

1 %reflete um caminho do ponto inicial ao ponto de despejo para obter o
   caminho de ida e volta
2 caminhoIdaVolta (Caminho, IV):-
3     reverse (Caminho, CR) ,
4     removePrimeiro (CR, R) ,
5     concat (Caminho, R, IV) .

```

Pesquisa não informada em Profundidade

Seguindo a mesma estratégia, para o desenvolvimento deste algoritmo utilizou-se como base a pesquisa em profundidade entre dois nodos, porém neste caso para cada rua ultrapassada calcula-se a distância percorrida, obtendo-se no final a distância total do caminho obtido. O algoritmo funciona da seguinte maneira: uma vez que se trata de um algoritmo de pesquisa não informada, o caminhão não vai ter preferência na escolha da rua adjacente, pelo que vai escolher a primeira rua adjacente (àquela em que se localiza) que encontrar. Para essa rua é calculada a distância a percorrer e de seguida avança-se para a mesma, (adicionando ao conjunto de nodos percorridos o nodo escolhido). Em caso de atingir o nodo final, o algoritmo faz *backtracking* e soma todas as distâncias calculadas. No final, o algoritmo dá como resultado não só o caminho entre a garagem e o ponto de despejo, como também o caminho inverso, e associada a cada um a distância total percorrida. O predicado que aplica este algoritmo é o **resolve_prof_Dist**.

Sabemos também, que este algoritmo não é ótimo pelo que temos de obter todos os caminhos e compará-los para ver qual o melhor. O predicado responsável por fazê-lo é **melhor_prof_Distancia**, que dá como resultado o caminho de menor distância percorrida.

```

1 resolve_pp_Distancia (Nodo, [Nodo|Caminho] , Distancia , IdaVolta , DistanciaDupl):-
2     profundidadeDistancia (Nodo, [Nodo] , Caminho, Distancia) ,
3     caminhoIdaVolta ([Nodo|Caminho] , IdaVolta) ,
4     DistanciaDupl is Distancia + Distancia.
5
6 profundidadeDistancia (Nodo, -, [], 0):-despejo (Nodo) .
7 profundidadeDistancia (Nodo, Historico , [ProxNodo|Caminho] , Distancia):-
8     adjacenteDistancia (Nodo, ProxNodo, D1) ,
9     nao(membro(ProxNodo, Historico)) ,
10    profundidadeDistancia (ProxNodo, [ProxNodo| Historico] , Caminho, D2) ,
11    Distancia is D1 + D2.
12
13 %Devolve a distancia entre dois nodos adjacentes
14 adjacenteDistancia (Nodo, ProxNodo, Distancia):-

```

```

15     adjacente (Nodo, ProxNodo) ,
16     distanciaEntreRuas_ID (Nodo, ProxNodo, Distancia) .
17
18     minimo ([ (P,X) ] , (P,X) ) .
19     minimo ([ (Px,X) | L ] , (Py,Y) ) :- minimo (L, (Py,Y) ) , X > Y .
20     minimo ([ (Px,X) | L ] , (Px,X) ) :- minimo (L, (Py,Y) ) , X <= Y .
21
22     resolve_prof_Dist (CamIda, DistIda, CamIdaVolta, DistIdaVolta) :- garagem (G) ,
23         resolve_pp_Distancia (G, CamIda, DistIda, CamIdaVolta, DistIdaVolta) .
24
25     melhor_prof_Distancia (Cam, Distancia) :-
26         findall ((Iv, Dist2), resolve_prof_Dist (Ca, Dist, Iv, Dist2), L) ,
27         minimo (L, (Cam, Distancia)) .

```

Pesquisa informada Gulosa

Como sabemos, um algoritmo de pesquisa informada permite que se tomem decisões no momento do deslocamento para um dado nodo do grafo. Posto isto, e seguindo essa ideia, a heurística utilizada foi a seguinte: situando o caminhão numa rua(nodo), o algoritmo analisa todas as ruas adjacentes, e para cada uma delas calcula a distância a percorrer caso se desloque para as mesmas. Assim, uma vez que estamos na pesquisa gulosa, após a análise de todas as adjacentes é escolhida aquela que está mais perto, ou seja a que permite percorrer menor distância. De salientar que para cada iteração se guardam todos os caminhos que não foram escolhidos numa primeira estância, para que na iteração seguinte se possa comparar os mesmos com o novo caminho, pois caso o novo caminho seja pior, o algoritmo tem de ser capaz de escolher então o melhor dos caminhos guardados (e com melhor refere-se o mais curto). Uma vez chegado ao nodo objetivo, têm-se o caminho percorrido desde a garagem até ao mesmo, e associado a si a distância total percorrida. O predicado que aplica este algoritmo é o **resolve_gulosa_Dist**.

Percebemos facilmente que este algoritmo não é otimal, uma vez que escolhe caminhos mais curtos momentaneamente sem ter em conta que se pode estar a afastar do nodo objetivo, pelo que temos também de obter todos os caminhos e compará-los para ver qual o melhor, tal como nos algoritmos anteriores. Para isto, foram então desenvolvidos os predicados **todos_Caminhos_Gulosa_Dist** que calcula todos os caminhos da garagem até ao ponto de despejo e associa a cada um deles a distância total percorrida, **caminhosOrdenaGulosaDist** que ordena todos os caminhos anteriormente calculados por número de pontos de recolha e por fim **bestCaminho_Gulosa_Dist**, que dá como resultado o último elemento da lista anteriormente ordenada, correspondente portanto ao caminho com menor distância percorrida.

```

1     resolve_gulosa (Nodo, Caminho/Dist) :-
2         agulosa ([ [Nodo]/0/0 ] , InvCaminho/Dist/_ ) ,
3         reverse (InvCaminho, Caminho) .
4
5
6     agulosa (Caminhos, Caminho) :-
7         obtem_melhor_g (Caminhos, Caminho) ,
8         Caminho = [Nodo|_] / _ / _ , despejo (Nodo) .
9
10    agulosa (Caminhos, SolucaoCaminho) :-
11        obtem_melhor_g (Caminhos, MelhorCaminho) ,
12        seleciona (MelhorCaminho, Caminhos, OutrosCaminhos) ,
13        expande_gulosa (MelhorCaminho, ExpCaminhos) ,
14        append (OutrosCaminhos, ExpCaminhos, NovoCaminhos) ,
15        agulosa (NovoCaminhos, SolucaoCaminho) .
16
17
18
19

```

```

20 obtem_melhor_g ([Caminho], Caminho) :- !.
21
22 obtem_melhor_g ([Caminho1/Dist1/Est1, _/Dist2/Est2 | Caminhos], MelhorCaminho) :-
23     Est1 =< Est2, !,
24     obtem_melhor_g ([Caminho1/Dist1/Est1 | Caminhos], MelhorCaminho).
25
26 obtem_melhor_g ([_ | Caminhos], MelhorCaminho) :-
27     obtem_melhor_g (Caminhos, MelhorCaminho).
28
29 expande_gulosa (Caminho, ExpCaminhos) :-
30     findall (NovoCaminho, adjacente_Gulosa_Dist (Caminho, NovoCaminho),
31             ExpCaminhos).
32
33 adjacente_Gulosa_Dist ([Nodo | Caminho] / Dist / _, [ProxNodo, Nodo | Caminho] /
34     NovaDist / Distancia) :-
35     adjacenteDistancia (Nodo, ProxNodo, Distancia),
36     \+ member (ProxNodo, Caminho),
37     NovaDist is Dist + Distancia.
38
39 resolve_gulosa_Dist (Cam, Distancia) :-
40     garagem (G),
41     resolve_gulosa (G, Cam / Distancia).
42
43 %primeiro caminho ida e volta
44 resolve_gulosa_Dist_IV (Cam, Distancia) :-
45     resolve_gulosa_Dist (Cam1, Distancia1),
46     caminhoIdaVolta (Cam1, Cam),
47     Distancia is Distancia1 + Distancia1.
48
49 %Processo para descobrir o caminho mais rápido
50 todos_Caminhos_Gulosa_Dist (X) :-
51     findall ((Cam, Dist), resolve_gulosa_Dist (Cam, Dist), R),
52     eliminaRepetidos (R, X).
53
54 caminhosOrdenaGulosaDist (S) :-
55     todos_Caminhos_Gulosa_Dist (X), ordena (X, S).
56
57 bestCaminho_Gulosa_Dist (Best) :-
58     caminhosOrdenaGulosaDist (S), ultimoElementoLista (S, Best).

```

Pesquisa informada pela A*

Aqui surge a primeira implementação de um algoritmo de A*. A heurística associada ao mesmo, para o problema em questão é a seguinte: situando o camião num nodo, o algoritmo irá analisar todos os nodos adjacentes. Para cada um destes é aplicado o predicado **estima**, que calcula a distância mínima de cada um ao nodo objetivo (neste caso o ponto de despejo) para ver o quão bons são esses nodos uns em relação aos outros. Também para estes é guardada a distância total desde o nodo inicial até chegar aos mesmos. Assim a comparação feita entre nodos de forma a saber para qual se deslocar, é feita de acordo com a soma da distância total desde o nodo inicial com a distância mínima que terá de ser percorrida até ao nodo final. O melhor nodo será, naturalmente o que apresentar um menor valor para esta soma. O predicado que aplica este algoritmo é o **resolveEstrela**.

Os algoritmos de A* permitem obter um caminho ótimo, pelo que é desnecessário procurar mais possibilidades, tal como foi feito nos algoritmos anteriores.

```

1 estima(Nodo, Estima):-despejo(R), distanciaEntreRuas.ID(Nodo,R, Estima).
2
3 resolve_estrela_Dist(Nodo, Caminho/Distancia):-
4     estima(Nodo, Estima),
5     aestrela_Dist([[Nodo]/0/Estima], InvCaminho/Distancia/_),
6     reverse(InvCaminho, Caminho).
7
8
9 aestrela_Dist(Caminhos, Caminho):-
10     obtem_melhor_g_estrela_Dist(Caminhos, Caminho),
11     Caminho = [Nodo|_]/_/_ , despejo(Nodo).
12
13 aestrela_Dist(Caminhos, SolucaoCaminho):-
14     obtem_melhor_g_estrela_Dist(Caminhos, MelhorCaminho),
15     seleciona(MelhorCaminho, Caminhos, OutrosCaminhos),
16     expande_estrela_Dist(MelhorCaminho, ExpCaminhos),
17     append(OutrosCaminhos, ExpCaminhos, NovoCaminhos),
18     aestrela_Dist(NovoCaminhos, SolucaoCaminho).
19
20 obtem_melhor_g_estrela_Dist([Caminho], Caminho):-!.
21
22 obtem_melhor_g_estrela_Dist([Caminho1/Distancia1/Est1, _/Distancia2/Est2|
23     Caminhos], MelhorCaminho):-
24     Est1 + Distancia1 <= Est2 + Distancia2, !,
25     obtem_melhor_g_estrela_Dist([Caminho1/Distancia1/Est1|Caminhos],
26     MelhorCaminho).
27
28
29 obtem_melhor_g_estrela_Dist([_|Caminhos], MelhorCaminho):-
30     obtem_melhor_g_estrela_Dist(Caminhos, MelhorCaminho).
31
32 expande_estrela_Dist(Caminho, ExpCaminhos):-
33     findall(NovoCaminho, adjacenteEstrela_Dist(Caminho, NovoCaminho),
34     ExpCaminhos).
35
36
37 adjacenteEstrela_Dist([Nodo|Caminho]/Distancia/_ , [ProxNodo, Nodo|Caminho]/
38     NovaDistancia/Est):-
39     adjacenteDistancia(Nodo, ProxNodo, PassoDistancia), \+ member(ProxNodo
40     ,Caminho),
41     NovaDistancia is Distancia + PassoDistancia,
42     estima(ProxNodo, Est).
43
44
45 resolveEstrela(Cam, Dist):-
46     garagem(G),
47     resolve_estrela_Dist(G, Cam/Dist), !.
48
49 %caminho de ida e volta
50 melhorIdaVolta_Estrela(Cam, Dist):-
51     resolveEstrela(Cam1, Dist1),
52     caminhoIdaVolta(Cam1, Cam),
53     Dist is Dist1 + Dist1.

```

3.7.1 Análise de resultados

```
?- resolve prof Dist(CamIda,DistIda,CamIdaVolta,DistIdaVolta).
CamIda = [21944, 15805, 15806, 15807, 15888, 15812, 15809, 15808],
DistIda = 0.0031372189856496762,
CamIdaVolta = [21944, 15805, 15806, 15807, 15888, 15812, 15809, 15808, 15809|...],
DistIdaVolta = 0.0062744379712993525 .

?- melhor prof Distancia(Cam,Distancia).
Cam = [21944, 15807, 15808, 15807, 21944],
Distancia = 0.0027113128718653564 .
```

Figura 3.11: Teste da pesquisa em profundidade

```
?- resolve gulosa Dist IV(Cam,Distancia),escrever(Cam).
21944
15807
15889
15812
15809
15808
15809
15812
15889
15807
21944
Cam = [21944, 15807, 15889, 15812, 15809, 15808, 15809, 15812, 15889|...],
Distancia = 0.00414585885949579 .
```

Figura 3.12: Teste da pesquisa Gulosa

```
?- melhorIdaVolta Estrela(Cam,Dist).
Cam = [21944, 15807, 15808, 15807, 21944],
Dist = 0.0027113128718653564.
```

Figura 3.13: Teste da pesquisa A*

3.8 Comparar circuitos de recolha tendo em conta os indicadores de produtividade

Uma vez que o trabalho foi realizado para 15 valores, então não se teve em consideração o indicador de produtividade relativo à quantidade, pelo que os algoritmos apresentados nesta alínea apenas dizem respeito ao indicador de produtividade: distância média percorrida entre pontos.

Distância média percorrida entre pontos de recolha

Para este indicador de produtividade, não foi difícil desenvolver os algoritmos, pois estes são exatamente os mesmos que na alínea anterior, para o cálculo das distâncias, porém o resultado final muda, sendo que agora não queremos o caminho com a distância mais curta no total, mas aquele cuja distância média percorrida entre nodos é menor. Por isso, pegando nos algoritmos desenvolvidos anteriormente, que nos dão um caminho e associado a este a distância total do mesmo, apenas foi necessário dividir a distância pelo número de ruas do caminho.

Pesquisa não informada em profundidade

Como podemos observar em baixo, no predicado **resolve_Profundidade_3** reutilizou-se o predicado **resolve_prof_Dist**, de forma a que para um caminho fosse calculada a média de distância percorrida entre as ruas do mesmo. Por sua vez o predicado **melhorProfundidadeMedia**, encontra todos os caminhos com recurso à pesquisa em profundidade e escolhe aquele que possui menor média.

```
1 resolve_Profundidade_3(CamIdaVolta, DistIdaVolta, Media):-
2     resolve_prof_Dist(CamIda, DistIda, CamIdaVolta, DistIdaVolta), length(
3         CamIdaVolta, NrP), Media is DistIdaVolta / (NrP-1).
4 melhorProfundidadeMedia(Cam, Distancia, Media):- findall((Iv, Media),
5     resolve_Profundidade_3(Iv, D, Media), L), minimo(L, (Cam, Media)).
```

Pesquisa informada Gulosa

Seguindo a mesma lógica, temos o predicado **resolve_Gulosa_3** que calcula a média de distância percorrida entre ruas de um caminho e o predicado **melhorGulosa3** que procura todos e escolhe o melhor.

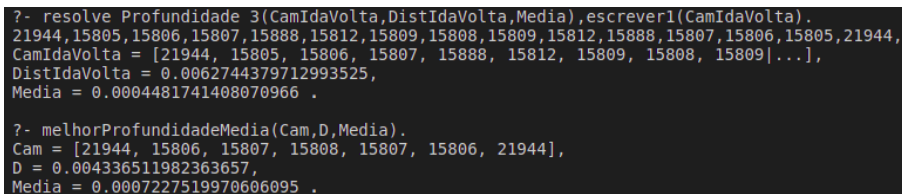
```
1 resolve_Gulosa_3(IV, Distancia, Media):-
2     resolve_gulosa_Dist(Cam, Dist),
3     caminhoIdaVolta(Cam, IV),
4     length(IV, NrP),
5     Distancia is Dist + Dist,
6     Media is Distancia / (NrP-1).
7
8 melhorGulosa3(Cam, Distancia, Media):- findall((Cam, Media), resolve_Gulosa_3(
9     Cam, D, Media), L), minimo(L, (Cam, Media)).
```

Pesquisa informada A*

Da mesma forma para o algoritmo de A*, temos o predicado **resolve_Estrela_3** que calcula a média de distância percorrida entre ruas de um caminho. Como este algoritmo é ótimo não há necessidade de verificar todas as possibilidades, porque a primeira solução encontrada é a ótima.

```
1 resolve_Estrela_3(IV, Distancia, Media):-
2     resolveEstrela(Cam, Dist),
3     caminhoIdaVolta(Cam, IV),
4     length(IV, NrNodos),
5     Distancia is Dist + Dist,
6     Media is Distancia / (NrNodos-1).
```

3.8.1 Análise de Resultados



```
?- resolve_Profundidade_3(CamIdaVolta, DistIdaVolta, Media), escreve1(CamIdaVolta).
21944, 15805, 15806, 15807, 15888, 15812, 15809, 15808, 15809, 15812, 15888, 15807, 15806, 15805, 21944,
CamIdaVolta = [21944, 15805, 15806, 15807, 15888, 15812, 15809, 15808, 15809],
DistIdaVolta = 0.0062744379712993525,
Media = 0.0004481741408070966 .

?- melhorProfundidadeMedia(Cam, D, Media).
Cam = [21944, 15806, 15807, 15808, 15807, 15806, 21944],
D = 0.004336511982363657,
Media = 0.0007227519970606095 .
```

Figura 3.14: Teste da pesquisa em profundidade


```
?- resolve Gulosa 3(IV,Distancia,Media),escrever1(IV).
21944,15807,15889,15812,15809,15808,15809,15812,15889,15807,21944,
IV = [21944, 15807, 15889, 15812, 15809, 15808, 15809, 15812, 15889|...],
Distancia = 0.00414585885949579,
Media = 0.00041458588594957903 .
```

Figura 3.15: Teste da pesquisa Gulosa

```
?- resolve Estrela 3(IV,Distancia,Media).
IV = [21944, 15807, 15808, 15807, 21944],
Distancia = 0.0027113128718653564,
Media = 0.0006778282179663391.
```

Figura 3.16: Teste da pesquisa A*

3.9 Escolher o circuito mais eficiente (usando um critério de eficiência à escolha)

O objetivo desta alínea era desenvolver algoritmos para um critério de eficiência à escolha. Por isso, o critério escolhido foi a **quantidade de resíduos recolhidos**, ou seja um caminho é tanto melhor quanto maior a quantidade de lixo recolhida ao longo do mesmo. Nesta alínea, no que toca a algoritmos de pesquisa não informada foi utilizado o algoritmo de pesquisa em largura e profundidade limitada, e em relação à pesquisa informada, foi utilizado o algoritmo da Gulosa.

Antes de passar à apresentação e análise de cada um dos algoritmos implementados, é importante mencionar alguns predicados que foram elaborados para ajudar tanto na construção dos mesmos, como para fornecer ao utilizador a oportunidade de ter acesso a mais algumas informações.

Predicado que dado um identificador de rua, retorna a quantidade de lixo a recolher na mesma.

```
1 %Devolve a quantidade de lixo a recolher numa rua dado o seu ID
2 quantoLixo_IdRua (IdRua , Qt) :- rua (IdRua , Lat , Long , Res , Qt) .
```

Predicado que dado um caminho(lista de ruas) devolve a quantidade de lixo total a recolher no mesmo.

```
1 quantoLixo_Ruas ([ ] , 0) .
2 quantoLixo_Ruas ([X|XS] , Qt) :- quantoLixo_Ruas (XS, Qt1) , quantoLixo_IdRua (X, Qt2) ,
    Qt is Qt1 + Qt2.
```

Pesquisa não informada em largura

Utilizando o algoritmo de pesquisa em largura mais básico demonstrado no início do relatório, foi facilmente possível desenvolver um predicado que fazendo uma pesquisa em largura, calculasse a quantidade de lixo total do caminho obtido. Neste algoritmo cada nodo é expandido em largura nos seus adjacentes e o caminho obtido até ao momento é expandido em vários com os nodos adjacentes à cabeça. Por sua vez para cada um desses caminhos obtidos, é calculado a quantidade de lixo recolhida, que corresponde à quantidade recolhida de todo o caminho antigo somada com a quantidade do nodo adjacente que deu origem ao caminho. Em cada iteração é verificado se um dos caminhos já atingiu o nodo objetivo(ponto de despejo), pelo que em caso afirmativo se retorna esse mesmo caminho. O predicado que aplica este algoritmo é o **resolve_largura_5**.

Sabemos que este algoritmo não é ótimo pelo que temos também de obter todos os caminhos e compará-los para ver qual o melhor. Para isto foram então desenvolvidos os predicados **todos_Caminhos_Qt_Largura** que calcula todos os caminhos da garagem até ao ponto de despejo e associa a cada um deles a quantidade recolhida total, **caminhosOrdena_Qt_Largura** que ordena todos os caminhos anteriormente calculados e por fim **bestCaminho_Qt_Largura**, que dá como resultado o primeiro elemento da lista anteriormente ordenada, correspondente portanto ao caminho com maior quantidade recolhida.

```

1 largura5(Inicio ,Fim,Caminho/Qt) :-
2     quantoLixo_IdRua(Inicio ,Estima) ,
3     bfs5([[Inicio]/Estima] ,Fim,CaminhoAux/Qt) ,
4     reverse(CaminhoAux,Caminho) .
5
6 bfs5([[Fim|Caminho]/Qt|_] ,Fim, [Fim|Caminho]/Qt) .
7 bfs5([Caminho1|Caminhos] , Fim, Caminho) :-
8     expande_5(Caminho1 , NovosCaminhos) ,
9     append(Caminhos , NovosCaminhos , Caminhos1) ,
10    bfs5(Caminhos1 , Fim, Caminho) .
11
12 expande_5(Caminho1 , NovosCaminhos) :-
13     findall(NovoCaminho , adjacenteLargura_5(Caminho1 , NovoCaminho) ,
14            NovosCaminhos) ,!.
15     expande_5(- , - , []) .
16
17 adjacenteLargura_5([Nodo|Caminho]/Qt1 , [ProxNodo,Nodo|Caminho]/Qt) :-
18     adjacente(Nodo,ProxNodo) ,
19     nao(membro(ProxNodo,Caminho)) ,
20     quantoLixo_IdRua(ProxNodo,QtNext) ,
21     Qt is Qt1 + QtNext .
22
23 resolve_largura_5(R):-garagem(G) , despejo(D) , largura5(G,D,R) .
24
25 %Processo para apresentar o melhor caminho
26 todos_Caminhos_Qt_Largura(X):-
27     findall((Cam,Qt) , resolve_largura_5(Cam/Qt) , R) ,
28     eliminaRepetidos(R,X) .
29
30 caminhosOrdena_Qt_Largura(S):-
31     todos_Caminhos_Qt_Largura(R) ,
32     ordena(R,S) .
33
34 %Develve o caminho com mais quantidade recolhida
35 bestCaminho_Qt_Largura(Best):-
36     caminhosOrdena_Qt_Largura(S) ,
37     primeiroElementoLista(S,Best) .

```

Pesquisa não informada em Profundidade Limitada

Seguindo de forma semelhante, o raciocínio da pesquisa em largura, desenvolveu-se um algoritmo de pesquisa em profundidade limitada, pelo que não será explicado detalhadamente o seu processo. O predicado que aplica este algoritmo é o **resolve_prof_lim_5**.

Este algoritmo não é ótimo pelo que temos também de obter todos os caminhos e compará-los para ver qual o melhor. Foram então desenvolvidos os predicados **todos_Caminhos_Qt_Prof_Lim** que calcula todos os caminhos da garagem até ao ponto de despejo e associa a cada um deles a quantidade recolhida total, **caminhosOrdena_Qt_Prof_Lim** que ordena todos os caminhos anteriormente calculados por quantidade recolhida e por fim **bestCaminho_Qt_Prof_Lim**, que dá

como resultado o primeiro elemento da lista anteriormente ordenada, correspondente portanto ao caminho com maior quantidade recolhida.

```

1 resolve_prof_lim_Qt(Nodo, Caminho/Qt, L) :-                profundidade_limitada_Qt
  ([], Nodo, InvCaminho/Qt, L), reverse(InvCaminho, Caminho).
2
3 profundidade_limitada_Qt(Caminho, Nodo, [Nodo|Caminho]/Qt, _):-
4   despejo(Nodo), quantoLixo_IdRua(Nodo, Qt).
5 profundidade_limitada_Qt(Caminho, Nodo, S/Qt, L) :-
6   quantoLixo_IdRua(Nodo, Qt1),
7   L>0, adjacente(Nodo, NodoL),
8   \+ member(NodoL, Caminho),
9   L1 is L-1,
10  profundidade_limitada_Qt([Nodo|Caminho], NodoL, S/Qt2, L1),
11  Qt is Qt1+Qt2.
12
13 resolve_prof_lim_5(L, Cam):-
14   garagem(G), despejo(D),
15   resolve_prof_lim_Qt(G, Cam, L).
16
17 %Processo para apresentar o melhor caminho
18 todos_Caminhos_Qt_Prof_Lim(L, X):-
19   findall((Cam, Qt), resolve_prof_lim_5(L, Cam/Qt), R),
20   eliminaRepetidos(R, X).
21
22 caminhosOrdena_Qt_Prof_Lim(L, S):-
23   todos_Caminhos_Qt_Prof_Lim(L, R),
24   ordena(R, S).
25
26 %Devolve o caminho com mais quantidade recolhida
27 bestCaminho_Qt_Prof_Lim(L, Best):-
28   caminhosOrdena_Qt_Prof_Lim(L, S),
29   primeiroElementoLista(S, Best).

```

Pesquisa informada Gulosa

Este algoritmo da gulosa é muito semelhante aos algoritmos de gulosa anteriormente explicados pelo que não vai ser detalhadamente especificado. O predicado que aplica este algoritmo é o **resolve_gul_Qt**.

Este algoritmo não é otimal pelo que temos também de obter todos os caminhos e compará-los para ver qual o melhor. Foram então desenvolvidos os predicados **todos_Caminhos_Qt_Gulosa** que calcula todos os caminhos da garagem até ao ponto de despejo e associa a cada um deles a quantidade recolhida total, **caminhosOrdenaGulosa_Qt** que ordena todos os caminhos anteriormente calculados por quantidade recolhida e por fim **bestCaminho_Qt_Gulosa**, que dá como resultado o primeiro elemento da lista anteriormente ordenada, correspondente portanto ao caminho com maior quantidade recolhida.

```

1 resolve_gulosa_Qt(Nodo, Caminho/Qt):-
2   quantoLixo_IdRua(Nodo, Estima),
3   agulosa_Qt([[Nodo]/Estima], InvCaminho/_), reverse(InvCaminho,
4   Caminho),
5   quantoLixo_Ruas(Caminho, Qt).
6
7 agulosa_Qt(Caminhos, Caminho):-
8   obtem_melhor_g_Qt(Caminhos, Caminho),
9   Caminho = [Nodo|_] / _, despejo(Nodo).
10

```

```

11 agulosa_Qt (Caminhos, SolucaoCaminho):-
12     obtem_melhor_g_Qt (Caminhos, MelhorCaminho) ,
13     seleciona (MelhorCaminho, Caminhos, OutrosCaminhos) ,
14     expande_gulosa_Qt (MelhorCaminho, ExpCaminhos) ,
15     aumentaEstima (MelhorCaminho, ExpCaminhos, AltCaminhos) ,
16     append (OutrosCaminhos, AltCaminhos, NovoCaminhos) ,
17     agulosa_Qt (NovoCaminhos, SolucaoCaminho) .
18
19 obtem_melhor_g_Qt ([Caminho] , Caminho) :-!.
20
21 obtem_melhor_g_Qt ([Caminho1/Est1, _/Est2 | Caminhos] , MelhorCaminho):-
22     Est1 >= Est2, !,
23     obtem_melhor_g_Qt ([Caminho1/Est1 | Caminhos] , MelhorCaminho) .
24
25 obtem_melhor_g_Qt ([_ | Caminhos] , MelhorCaminho):-
26     obtem_melhor_g_Qt (Caminhos, MelhorCaminho) .
27
28 expande_gulosa_Qt (Caminho, ExpCaminhos):-                                findall (
29     NovoCaminho, adjacente_Qt (Caminho, NovoCaminho) , ExpCaminhos) .
30
31 adjacente_Qt ([Nodo | Caminho] / _ , [ProxNodo, Nodo | Caminho] / Est):-
32     adjacente (Nodo, ProxNodo) , \+ member (ProxNodo, Caminho) ,
33     quantoLixo_IdRua (ProxNodo, Est) .
34
35
36 resolve_gul_Qt (Caminho, Qt):-
37     garagem (G) ,
38     resolve_gulosa_Qt (G, Caminho/Qt) .
39
40
41 %Processo para descobrir o caminho com mais Quantidade recolhida
42 todos_Caminhos_Qt_Gulosa (X):-
43     findall ((Cam, Qt) , resolve_gul_Qt (Cam, Qt) , R) ,
44     eliminaRepetidos (R, X) .
45
46 caminhosOrdenaGulosa_Qt (S):-
47     todos_Caminhos_Qt_Gulosa (X) ,
48     ordena (X, S) .
49
50 bestCaminho_Qt_Gulosa (Best):-caminhosOrdenaGulosa_Qt (S) ,
    primeiroElementoLista (S, Best) .

```

3.9.1 Análise de Resultados

```
?- resolve largura 5(R).
R = [21944, 15807, 15808]/4660 .

?- caminhosOrdena Qt Largura(S),topN(S,10).
[21944,15805,15806,15807,15890,15812,15810,15811,15893,15808],23550
[21944,15805,15806,15807,15889,15812,15810,15811,15893,15808],23510
[21944,15805,15806,15807,15891,15812,15810,15811,15893,15808],23050
[21944,15805,15806,15807,15888,15812,15810,15811,15893,15808],22970
[21944,15805,15806,15807,15890,15812,15810,15811,15894,15808],22570
[21944,15805,15806,15807,15889,15812,15810,15811,15894,15808],22530
[21944,15805,15806,15807,15891,15812,15810,15811,15894,15808],22070
[21944,15805,15806,15807,15888,15812,15810,15811,15894,15808],21990
[21944,15805,15806,15807,15890,15812,15809,15811,15893,15808],21790
[21944,15805,15806,15807,15889,15812,15809,15811,15893,15808],21750
S = [[([21944, 15805, 15806, 15807, 15890, 15812, 15810|...], 23550), ([21944, 15805, 15806, 15807|...], 22970), ([21944, 15805, 15806|...], 22570), ([21944, 15805, 15806, 15807|...], 22530), ([21944, 15805, 15806, 15807|...], 22070), ([21944, 15805, 15806, 15807|...], 21990), ([21944, 15805, 15806, 15807|...], 21790), ([21944, 15805, 15806, 15807|...], 21750)]

?- bestCaminho Qt Largura((Best,Qt)),escrever1(Best).
21944,15805,15806,15807,15890,15812,15810,15811,15893,15808,
Best = [21944, 15805, 15806, 15807, 15890, 15812, 15810, 15811, 15893|...],
Qt = 23550 .
```

Figura 3.17: Teste da pesquisa em largura

```
?- resolve prof lim 5(4,Cam).
Cam = [21944, 15805, 15806, 15807, 15808]/11270 .

?- caminhosOrdena Qt Prof Lim(4,S),topN(S,10).
[21944,15805,15806,15807,15808],11270
[21944,15806,15807,15808],7880
[21944,15807,15808],4660
S = [[([21944, 15805, 15806, 15807, 15808], 11270), ([21944, 15806, 15807, 15808], 7880), ([21944, 15807, 15808], 4660)]

?- bestCaminho Qt Prof Lim(4,Best).
Best = ([21944, 15805, 15806, 15807, 15808], 11270) .
```

Figura 3.18: Teste da pesquisa em profundidade limitada, limite = 4

```
?- resolve gul Qt(Caminho,Qt).
Caminho = [21944, 15805, 15806, 15807, 15808],
Qt = 11270 .
```

Figura 3.19: Teste da pesquisa Gulosa

```
?- resolve gul Qt(Caminho,Qt).
Caminho = [21944, 15805, 15806, 15807, 15808],
Qt = 11270 .

?- caminhosOrdenaGulosa Qt(S),topN(S,10).
[21944,15805,15806,15807,15888,15889,15890,15891,15892,15893,15894,15808],19030
[21944,15805,15806,15807,15889,15890,15891,15892,15893,15894,15808],18310
[21944,15805,15806,15807,15888,15889,15890,15891,15892,15893,15808],17750
[21944,15805,15806,15807,15890,15891,15892,15893,15894,15808],17050
[21944,15805,15806,15807,15889,15890,15891,15892,15893,15808],17030
[21944,15805,15806,15807,15890,15891,15892,15893,15808],15770
[21944,15805,15806,15807,15891,15892,15893,15894,15808],15750
[21944,15806,15807,15888,15889,15890,15891,15892,15893,15894,15808],15640
[21944,15805,15806,15807,15888,15889,15890,15891,15892,15808],15490
[21944,15806,15807,15889,15890,15891,15892,15893,15894,15808],14920
S = [[([21944, 15805, 15806, 15807, 15888, 15889, 15890|...], 19030), ([21944, 15805, 15806, 15807|...], 17050), ([21944, 15805, 15806|...], 17030), ([21944, 15805, 15806, 15807|...], 15770), ([21944, 15805, 15806, 15807|...], 15750), ([21944, 15806, 15807, 15888, 15889, 15890, 15891, 15892, 15893, 15894, 15808], 15640), ([21944, 15805, 15806, 15807, 15888, 15889, 15890, 15891, 15892, 15808], 15490), ([21944, 15806, 15807, 15889, 15890, 15891, 15892, 15893, 15894, 15808], 14920)]

?- bestCaminho Qt Gulosa((Best,Qt)),escrever1(Best).
21944,15805,15806,15807,15888,15889,15890,15891,15892,15893,15894,15808,
Best = [21944, 15805, 15806, 15807, 15888, 15889, 15890, 15891, 15892|...],
Qt = 19030 .
```

Figura 3.20: Teste da pesquisa Gulosa para ficheiro **arestasPequeno.pl**

3.10 Extra

Como é possível reparar para quase todos os algoritmos feitos, exceto os algoritmos relativos à distância é sempre apresentado apenas o caminho de ida da garagem para o ponto de despejo, uma vez que é o percurso no qual o camião vai fazer a recolha de lixo. Para o caminho de volta, do ponto de despejo para a garagem existem duas possibilidades: ou obtendo um caminho através de um dos predicados desenvolvidos, considera-se que o percurso de retorno à garagem é feito pelas mesmas ruas, e sendo assim com qualquer caminho é fácil de imaginar o percurso inverso(o que não tem muita lógica visto que pode estar a percorrer ruas desnecessárias para voltar à garagem), ou uma vez no ponto de despejo, aplica-se um algoritmo de cálculo da menor distância, que permite saber ao camião qual o melhor caminho para retornar á garagem. De forma a permitir esta segunda possibilidade foi elaborado um predicado que dado um caminho como argumento, da garagem para o ponto de recolha, retorna o caminho mais rápido para voltar à garagem, utilizando uma pesquisa informada do tipo A*. Esse predicado desenvolvido designa-se por **volta**, e encontra-se implementado no fim do ficheiro **individual.pl**.

Capítulo 4

Análise comparativa entre as diferentes estratégias de procura

Embora todos os testes apresentados tenham sido feitos, com recurso ao ficheiro **arestasReduce.pl**, por motivos de limitação de stack, decidi utilizar para a análise das diferentes estratégias o ficheiro de arestas mais completo **arestas.pl** para que as diferenças entre tempos de execução fossem mais nítidas.

Estratégia	Tempo(Segundos)	Espaço	Profundidade/Custo (Inferências)	Encontrou a melhor solução?
Profundidade	0.001	$O(bm)$	2023	Não
Largura	0.003	$O(b^d)$	4038	Apenas se os custos escalonados forem iguais
Profundidade Limitada	0.002	$O(b^d)$	6260	Não
Gulosa	0.005	Pior Caso: $O(b^m)$ Melhor Caso: $O(bxd)$	10 265	Não, mas dá um dos melhores
Estrela	0.001	#Nodos com $g(n)+h(n) \leq C^*$	1590	Sim

Figura 4.1: Tabela de análise comparativa entre estratégias

Assim após a análise da tabela, podemos concluir que as estratégias de pesquisa mais eficientes são a pesquisa em profundidade e a A^* , sendo que a A^* leva vantagem sobre a profundidade ao nível do número de inferências. A estratégia menos eficiente é claramente a Gulosa, que para além de ser a mais lenta, é também a que ocupa mais espaço e provoca maior número de inferências, o que se deve muito provavelmente à capacidade desta estratégia em encontrar uma maior possibilidade de caminhos e a ter que fazer portanto um maior número de comparações. Para o cálculo dos tempos e das inferências foi utilizado o predicado **time** pré-definido após o *include* da biblioteca *statistics*.

Capítulo 5

Conclusão

Durante a realização do trabalho foi possível conhecer melhor o funcionamento da programação simbólica, assim como métodos de representação de grafos como predicados para introduzir aos algoritmos de pesquisa informada e não-informada.

Foi dada a oportunidade de perceber as diferenças entre pesquisas informadas e pesquisas não informadas, assim como a sua implementação num problema real em contexto real.

Em suma, foi um projeto com grande potencial de exploração criativa que penso que foram atingidos os objetivos propostos para o trabalho de 15 valores e, para além do contexto de Sistemas de Representação de Conhecimento e Raciocínio, foi uma enorme mais valia para a formação de mecanismos de raciocínio gerais e incentivar olhar para problemas em prismas diferentes, porque nunca se sabe se só há apenas uma solução.

Resta o desafio de melhorar os algoritmos propostos, visto que há bastantes pontos onde é possível criar uma melhor solução ou utilizar uma heurística diferente de resolução