Universidade do Minho
Escola de Engenharia

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Sistemas de Representação de Conhecimento e Raciocínio

Instrumento de avaliação da componente Individual

Métodos de Resolução de Problemas e de Procura

Carolina Gil Afonso Santejo A89500

2020/2021

Índice

Índice de figuras

Introdução

Neste projeto individual no âmbito da unidade curricular de Sistemas de Representação de Conhecimento e Raciocínio foi pedida aplicação de métodos de resolução de problemas e de procura. No contexto deste trabalho, como caso de estudo foram utilizados os dados dos circuitos de recolha de resíduos urbanos do concelho de Lisboa, em particular na freguesia da Misericórdia. Desta forma, recorrendo aos vários algoritmos de procura informada e não informada aprendidos ao longo das aulas, foi possível aplicá-los de modo a gerar os circuitos de recolha que cumpram certos parâmetros, como por exemplo: cobrir um determinado território, percorrer menor distância, recolher maior quantidade de resíduo, etc.

É de notar que para a elaboração do projeto foi escolhida a versão completa, ou seja, é admitida uma capacidade limitada do camião.

Principais Objetivos

Este trabalho prático tem como principais objetivos a consolidação dos conteúdos abordados na aula, bem como a familiarização na utilização de diferentes algoritmos de procura recorrendo à ferramenta *SWI-Prolog*.

Processamento do *Dataset*

Para a realização deste trabalho foi fornecido um ficheiro *xlsx* como *dataset* base. Este ficheiro contém dados referentes aos circuitos de recolha de resíduos na freguesia da Misericórdia, nomeadamente:

- A posição de um ponto de recolha e o seu id;
- Tipo e capacidade de um contentor e número de contentores de um dado tipo e id correspondente;
- Rua pertencente ao ponto de recolha e suas adjacentes;
- Freguesia

Para facilitar a recolha de dados foram eliminadas colunas que não continham informações importantes tais como a freguesia (igual para todos) e tipo do contentor. Por outro lado foi também considerado que cada linha do *dataset* corresponde a um contentor, ou seja, foram agrupados todos os contentores de um dado tipo e considerou-se apenas a capacidade total.

Para ser possível efetuar a leitura do ficheiro foi feita a conversão do mesmo para o tipo *csv*.

Leitura do ficheiro *csv*

Para a leitura e parse dos dados do *dataset* foi utilizado um script em *python*. Com recurso às bibliotecas *re* e *csv* para cada linha foram recolhidos os valores de cada coluna e guardados em estruturas onde foram organizados. Desta forma, foram criadas as estruturas *pontosRecolha*, *contentores*, *ruas* e *ruasAdjacencias* onde estão respetivamente guardados os dados referentes aos pontos de recolha, contentores, ruas e ruas adjacentes.

Para o caso das ruas adjacentes foi admitido que, segundo a estrutura da coluna *PONTO_RECOLHA_LOCAL*, a rua que aparece em primeiro lugar é adjacente àquelas que estão seguidas do caractere ':'.

```python
pontosRecolha = {}
contentores = []
ruas = {}
ruasAdjacencias = {}

with open('datasetCSV.csv', 'r', encoding='utf-8') as file:
    reader = csv.reader(file, delimiter = ';')
    next(reader)
    for row in reader:
        latitude = row[0]
        longitude = row[1]
        contentorID = row[2]
        pontoRecolha = re.search(r'(\d+): ([\w -]+)( \(|,).+', row[3])
        adjacencias = re.search(r'\([^:]+: ([\w ]+) - ([\w ]+)', row[3])
        pontoRecolhaID = pontoRecolha.group(1)
        ruaNome = pontoRecolha.group(2)
        tipoResiduo = row[4]
        capacidade = row[5]
        pR = (pontoRecolhaID,latitude,longitude,ruaNome)
        contentor = (contentorID,tipoResiduo,capacidade,pontoRecolhaID)
        pontosRecolha[pontoRecolhaID] = pR
        contentores.append(contentor)
        if ruaNome in ruas:
            (ruaNome,listaPR) = ruas[ruaNome]
            listaPR.add(pontoRecolhaID)
            ruas[ruaNome] = (ruaNome,listaPR)
        else:
            ruas[ruaNome] = (ruaNome,set([pontoRecolhaID]))
        if adjacencias:
            if ruaNome in ruasAdjacencias:
                listaR = ruasAdjacencias[ruaNome]
                listaR.add(adjacencias.group(1))
                listaR.add(adjacencias.group(2))
                ruasAdjacencias[ruaNome] = listaR
            else:
                ruasAdjacencias[ruaNome] = set([adjacencias.group(1),adjacencias.group(2)])
        else:
            if ruaNome not in ruasAdjacencias:
                ruasAdjacencias[ruaNome] = set([])
```

Figura 1 - Código python para leitura do ficheiro csv

Elaboração da base de conhecimento

A elaboração da base de conhecimento foi feita com recurso ao mesmo script em python onde foi feita a leitura do *csv*. Em primeiro lugar foi criado o ficheiro baseConhecimento.pl, que é onde será escrita toda a base de conhecimento. Desta forma, tendo já os dados organizados em estruturas foi necessário escrevê-los para o ficheiro sob a forma de termos na linguagem *prolog*.

Para os pontos de recolha e contentores foi feita a escrita diretamente, sendo que foram acrescentados também os termos *camiao(15000) que indica a capacidade do camião* e *garagem(1,-9.1333,38.69864)* e *deposito(2, -9.149924625043745,38.71161385709514) que indicam o respetivo id, latitude e longitude da garagem e do depósito*. Estes dois últimos foram escolhidos de forma aleatória, no entanto foram escolhidas latitudes e longitudes o mais realistas possíveis, para que a distância aos pontos de recolha não fosse demasiado grande.

```
67  # Escrita para o ficheiro pl
68  fwrite = open("baseConhecimento.pl","w+",encoding='utf-8')
69
70  capacidadeCamiao = 15000
71  fwrite.write(f"% ------ Capacidade do camiao ------\n\n")
72  fwrite.write(f"camiao({capacidadeCamiao}).\n\n")
73
74
75  contentores.sort(key = lambda x: x[3])
76
77  fwrite.write(f"% ------ Pontos de recolha ------\n\n")
78  garagem = (1,-9.1333,38.69864)
79  deposito = (2, -9.149924625043745,38.71161385709514)
80  fwrite.write(f"garagem{garagem}.\n")
81  fwrite.write(f"deposito{deposito}.\n\n")
82  for ptRecolha in sorted(pontosRecolha.values()):
83      fwrite.write(f"pontoRecolha({ptRecolha[0]},{ptRecolha[1]},{ptRecolha[2]},'{ptRecolha[3]}').\n")
84
85  fwrite.write(f"\n\n% ------ Contentores ------\n\n")
86  for cont in contentores:
87      fwrite.write(f"contentor({cont[0]},'{cont[1]}',{cont[2]},{cont[3]}).\n")
88
```

Figura 2 - Escrita dos termos referentes aos pontos de recolha e contentores

Pelo contrário, para o caso das ruas e suas adjacências, devido à elevada complexidade e extensão do *dataset* e de forma a reduzir o número de vértices e arestas do grafo, foram eliminadas ruas segundo alguns critérios:

- Ruas que não possuem qualquer adjacência e não pertencem à lista de adjacências de qualquer outra rua;
- Ruas às quais não pertence nenhum ponto de recolha segundo o *dataset*.

Desta forma foi escrito o um novo termo *adjacenteRuas* onde cada rua possui uma lista das suas adjacentes.

```
91   fwrite.write(f"\n\n% ------ Ruas Adjacentes ------\n\n")
92   ruasAdjacenciasRes = ruasAdjacencias
93   for ruaNome, ruaAdj in sorted(ruasAdjacencias.items()):
94       # Retirar ruas às quais não pertence nenhum ponto de recolha
95       pr = list(sorted(ruaAdj))
96       for p in pr:
97           if p not in ruasAdjacencias.keys():
98               pr.remove(p)
99           elif p == ruaNome:
100              pr.remove(p)
101      # Retirar ruas que nao possuem quaisquer adjacencias
102      ruasAdjacenciasRes[ruaNome] = set(pr)
103      if len(pr) == 0:
104          for key, value in ruasAdjacencias.items():
105              if key != ruaNome and (ruaNome in value):
106                  fwrite.write(f"adjacenteRuas('{ruaNome}',[")
107                  pr.append(key)
108                  fwrite.write(f"'{pr[0]}'")
109                  fwrite.write(f"]).\n")
110
111          if len(pr) == 0:
112              ruasAdjacenciasRes.pop(ruaNome)
113          else:
114              ruasAdjacenciasRes[ruaNome] = set(pr)
115      else:
116          fwrite.write(f"adjacenteRuas('{ruaNome}',[")
117          for i in range(0, len(pr)):
118              if i == (len(pr)-1):
119                  fwrite.write(f"'{pr[i]}'")
120              else :
121                  fwrite.write(f"'{pr[i]}',")
122          fwrite.write(f"]).\n")
```

Figura 3 - Escrita do termo adjacenteRuas

Consequentemente, tendo agora as ruas adjacentes foi possível adicionar também o termo *rua* que possui os parâmetros id, nome e lista de pontos de recolha de uma dada rua.

```
127   fwrite.write(f"\n\n% ------ Ruas ------\n\n")
128   idRua = 1
129   ruaRes = {}
130   for ruaN in sorted(ruasAdjacenciasRes.keys()):
131       rua = ruas[ruaN]
132       fwrite.write(f"rua({idRua},'{rua[0]}',[")
133       pr = list(sorted(rua[1]))
134       for i in range(0, len(pr)):
135           if i == (len(pr)-1):
136               fwrite.write(f"{pr[i]}")
137           else :
138               fwrite.write(f"{pr[i]},")
139       fwrite.write(f"]).\n")
140       idRua+=1
141       ruaRes[ruaN] = rua
142
```

Figura 4 - Escrita do termo rua

Por último, para a representação do grafo propriamente dito foi definido o termo aresta, que dados dois vértices indica a distância da aresta que os une. É de notar que, pela interpretação feita ao *dataset* dado, foi considerado que cada ponto de recolha é um vértice do grafo. Assim sendo, podemos dizer que cada aresta é uma ligação entre dois pontos de recolha, sendo que podem ser dentro da mesma rua ou entre ruas adjacentes. Para o cálculo do comprimento da aresta foi determinada a distância geográfica em linha reta entre os dois pontos de recolha, utilizando a sua latitude e longitude. É de realçar que para determinar as arestas não foi tido em conta o sentido das ligações.

Desta forma, em primeiro lugar foi definido que dentro da mesma rua cada ponto de recolha só é adjacente ao seu subsequente e assim sucessivamente, sendo que foram ordenados por ordem crescente pelo seu Id. Por conseguinte, para as adjacências entre ruas, foi considerado que o primeiro ou ultimo ponto de recolha da rua em questão, dependendo se o valor do seu id é menor ou maior respetivamente, liga ao primeiro da sua adjacente.

Para finalizar foi também admitido que qualquer ponto de recolha é adjacente à garagem e ao depósito. Quanto à adjacência entre a garagem e o depósito foi estabelecido que existe apenas uma ligação depósito -> garagem que é quando o camião termina o seu percurso (termo *arestaRegresso*).

```
# Calcular adjacencias entre pontos de recolha
G = nx.Graph()
fwrite.write(f"\n\n% ------ Adjacencias entre pontos de recolha ------\n\n")
pontosAdjacentes = set() # todos os pontos que pertencem ao grafo
for rua in ruaRes.values():
    listaPtR = list(sorted(rua[1]))
    for i in range(0, len(listaPtR)-1):
        id1 = listaPtR[i]
        id2 = listaPtR[i+1]
        dist = distancia(float(pontosRecolha[id1][1]),float(pontosRecolha[id1][2]),float(pontosRecolha[id2][1]),float(pontosRecolha[id2][2]))
        fwrite.write(f"aresta({id1},{id2},{dist}).\n")
        pontosAdjacentes.add(id1)
        pontosAdjacentes.add(id2)
        G.add_edge(id1, id2, weight=dist)
    listaAdj = ruasAdjacenciasRes[rua[0]]
    for nomeRuaAdj in listaAdj:
        if nomeRuaAdj in ruaRes:
            listaPr = list(sorted(ruaRes[nomeRuaAdj][1]))
            if listaPr[0] < listaPtR[0]:
                id1 = listaPr[0]
                id2 = listaPtR[0]
                dist = distancia(float(pontosRecolha[id1][1]),float(pontosRecolha[id1][2]),float(pontosRecolha[id2][1]),float(pontosRecolha[id2][2]))
                fwrite.write(f"aresta({id1},{id2},{dist}).\n")
                G.add_edge(id1, id2, weight=dist)
            else:
                id1 = listaPr[0]
                id2 = listaPtR[len(listaPtR)-1]
                dist = distancia(float(pontosRecolha[id1][1]),float(pontosRecolha[id1][2]),float(pontosRecolha[id2][1]),float(pontosRecolha[id2][2]))
                fwrite.write(f"aresta({id1},{id2},{dist}).\n")
                G.add_edge(id1, id2, weight=dist)
            pontosAdjacentes.add(id1)
            pontosAdjacentes.add(id2)

```

Figura 5 - Escrita do termo aresta para pontos de recolha

```
fwrite.write(f"\n\n% ------ Adjacencias de pontos de recolha para deposito e garagem ------\n\n")
for idPonto in sorted(pontosAdjacentes):
    dist = distancia(float(pontosRecolha[idPonto][1]),float(pontosRecolha[idPonto][2]),float(garagem[1]),float(garagem[2]))
    fwrite.write(f"aresta({garagem[0]},{idPonto},{dist}).\n")
    dist = distancia(float(pontosRecolha[idPonto][1]),float(pontosRecolha[idPonto][2]),float(deposito[1]),float(deposito[2]))
    fwrite.write(f"aresta({deposito[0]},{idPonto},{dist}).\n")


dist = distancia(float(deposito[1]),float(deposito[2]),float(garagem[1]),float(garagem[2]))
fwrite.write(f"arestaRegresso({deposito[0]},{garagem[0]},{dist}).\n")


```

Figura 6 - Escrita do termo aresta de garagem e deposito para pontos de recolha

Resolução dos problemas de pesquisa

Para a resolução das problemas de pesquisa referidos nos tópicos seguintes, foram utilizados algoritmos de pesquisa informada e não informada. Para cada um foram escolhidos aqueles que mostraram ser mais apropriados para cada situação.

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

Para resolver esta *query* foram utilizados os três algoritmos de pesquisa não informada: profundidade, busca Iterativa Limitada em Profundidade e largura.

Os dois primeiros foram implementados de forma semelhante uma vez que o segundo é uma limitação do primeiro.

Em primeiro lugar foi definido que um território é uma lista de ruas que tem pelo menos uma adjacência entre si.

Os circuitos gerados por estes dois algoritmos só possuem pontos de recolha que pertençam a essas ruas e o caminho total gerado será o resultado de passar em todos esses pontos de recolha. Cada vez que o camião atinge a sua capacidade total é adicionado o deposito ao caminho e a distancia de ir e voltar ao ponto de recolha onde se situava e o algoritmo segue assim normalmente até passar por todos os pontos. É de notar que os pontos inicial e final do caminho são sempre a garagem, sendo que antes do final o camião vai sempre ao deposito.

Para o algoritmo de pesquisa em largura foi necessário adicionar a condição de o ponto de recolha escolhido pertencer à lista de pontos de recolha correspondentes às ruas fornecidas. Por outro lado foi também adicionada a condição do resíduo acumulado ser menor que a capacidade.

Identificar quais os circuitos com mais pontos de recolha

Para calcular os circuitos com mais pontos de recolha foi utilizado apenas o algoritmo de pesquisa em profundidade uma vez que pareceu ser a melhor opção dada a extensão da base de conhecimento.

Ao algoritmo de pesquisa normal foi adicionada a condição do residuo acumulado ser menor que a capacidade do camião, e por outro lado ser maior que 14980 de modo a formar um caminho com o maior numero de resíduo acumulado.

Assim sendo para calcular aquele que tem um maior número de pontos de recolha foram calculados vários circuitos (findnsols) e foi escolhido aquele que possui mais.

Comparar circuitos de recolha tendo em conta indicadores de produtividade

Para comparar circuitos de recolha tendo em conta indicadores de produtividade foram utilizados os algoritmos de pesquisa em profundidade e largura.

Para cada circuito foi calculado tanto a média da distancia entre os pontos como o resíduo acumulado. Para gerar um circuito foi utilizada o mesmo predicado da query anterior (circuito).

Escolher o circuito mais rápido

Para escolher o circuito mais rápido foram utilizados os algoritmos de pesquisa informada A* e gulosa.

Para a resolução deste problema foi modificado o predicado *estima* que dado um nodo do grafo, neste caso indica a sua distância ao deposito. Foi utilizada esta heurística uma vez que o melhor caminho será aquele que terá de ir menos vezes ao deposito. Uma vez que o ponto inicial tem que ser diferente do ponto final foi considerado neste caso que o ponto final é o deposito.

Escolher o circuito mais eficiente

Para escolher o circuito mais eficiente foi utilizado o algoritmo de pesquisa em profundidade.

Foi admitido assim que o caminho mais eficiente é aquele que percorre mais pontos de recolha em uma menor distancia.

Para este algoritmo utilizou-se novamente o predicado *circuito* que calcula um circuito de recolha e utilizando o *findnsols* é possível calcular alguns caminhos e foi escolhido em primeiro lugar o que tem um maior numero de pontos de recolha. Caso sejam iguais, é escolhido aquele que percorreu uma menor distancia.

Conclusão

Finalmente, pode-se concluir que todas as funcionalidades foram implementadas com sucesso tendo em conta os diversos problemas encontrados.

A elaboração deste projeto teve como maiores obstáculos, em primeiro lugar a elaboração da base de conhecimento tendo em conta a complexidade do dataset fornecido, e também a gestão de memoria na execução dos diferentes algoritmos devido à elevada escala do grafo construído.

Uma forma de melhorar o trabalho feito seria implementar todos os algoritmos de pesquisa em todos os problemas dados.