
SCC0630
Inteligência Artificial
Buscas Cega e Informada

PROBLEMA DO CAIXEIRO VIAJANTE (*TSP*)
EM UMA APLICAÇÃO WEB USANDO
GOOGLE MAPS API, PROLOG E PYTHON

FEITO POR

PEDRO MORELLO ABBUD
KOLLINS GABRIEL LIMA
CARLA NUNES DA CRUZ
GUSTAVO JOSÉ PEREIRA LEITE
PEDRO HENRIQUE FINI

*Universidade de
São Paulo*

ICMC

2017

1 Introdução

Este trabalho foi desenvolvido para a disciplina SCC0630, Inteligência Artificial, ministrada pela Professora Solange Oliveira Rezende. O objetivo deste é aprofundar e confirmar o conhecimento de nós, alunos, acerca de buscas cegas e buscas informadas, assim como modelagem e solução de problemas. Como o projeto tem também viés didático, escolhemos utilizar a linguagem *Prolog* como seu motor principal, pois esta foi a linguagem de programação de escolha nas aulas ministradas da disciplina. Dividimos a documentação e desenvolvimento deste projeto em três seções:

Modelagem do Problema :

Consiste em como definimos o problema e analisamos suas dificuldades.

Solucao do Problema :

Como foi abordado e resolvido o problema.

Apresentacao do Problema :

Escolhas para apresentar de forma clara, coerente e interessante os resultados obtidos.

Foi escolhido para este projeto o problema do caixeiro-viajante, ou em inglês, *TSP*, Travelling Salesman Problem. Este é um problema clássico em computação: dado um mapa e um número de cidades, encontrar o caminho de menor distância para percorrer todas as cidades, passando uma única vez por cada uma delas. O problema do caixeiro-viajante é um problema de alta complexidade computacional e encontrar a solução ótima para um número elevado de cidades é muito custoso.

De uma forma formal, podemos dizer que o problema se resume em achar o menor caminho hamiltoniano em um grafo, o que caracteriza um problema NP-Difícil.

Neste trabalho, é analisado o uso de dois métodos de busca não informada (busca em profundidade e em largura) e um método de busca informada (A^*) para a solução do problema.

O problema abordado neste trabalho foi modificado em, relação ao problema original, para simplificar as implementações. As modificações feitas foram:

- É definida uma cidade inicial, informada no início do programa para fazer a busca;
- A busca é feita sem considerar o retorno à cidade original.

Ao longo deste documento usaremos algumas definições:

Estados: Um estado é o caminho percorrido da cidade inicial até a cidade atual (com exceção do estado inicial, que não possui caminho mas apenas a cidade inicial).

Transição: Viagem de uma cidade para a outra.

Estado final: Caminho percorrido depois de visitar todas as cidades uma única vez (e que apresenta o menor custo).

Custo: Distância entre as cidades.

2 Modelagem

A figura ?? mostra uma representação das cidades e suas interligações modeladas em um grafo. Cada nó do grafo representa uma cidade e as arestas são os caminhos, cada qual possui um custo associado.

Para uma abordagem mais realista, foi utilizada a API do Google Maps em um script Python para obter as distâncias reais entre cidades reais. A partir deste script, são exportadas as regras para o prolog, onde foi feita a implementação da busca. Desta forma, pôde-se modelar o problema de forma dinâmica e interativa.

Assim, criamos uma função que se comunica com a API do Google Maps, e extraí dela uma matriz de conectividade, que indica o custo de cada transição. O código está representado na Listing 2.

A função `places_distance_matrix()` usa o wrapper Python do google maps e envia a lista de lugares a serem percorridos e retorna a string com a resposta da API.

A função `extract_distance_matrix()` é apenas um filtro para a resposta da API do googlemaps, que devolve a matrix de conectividade desejada.

A função `construct_rules()` a partir da resposta já filtrada da API constrói um arquivo com fatos necessários para a execução da busca em Prolog.

```
1  # coding=utf-8
2  import googlemaps
3
4  def extract_distance_matrix(results):
5      """ Filtra o resultado da api e devolve uma matriz """
6
7      a=[]
8      for row in results["rows"]:
9          b=[]
10         for value in row['elements']:
11             b.append(value['distance']['value'])
12         a.append(b)
13     return a
14
15 def places_distance_matrix(apikey, Places):
16     """Constroi Matriz de conectividade de todos os elementos usando a
17     ↪ Api do maps"""
18     gmaps= googlemaps.Client(key=apikey)
19     results=gmaps.distance_matrix(Places,Places)
20     return (extract_distance_matrix(results))
21
22 def construct_rules(File,apikey,Places):
```

```

22     """Constroi as regras de Prolog apartir da matriz de conectividade
23     ↪ """
24     distance_matrix= places_distance_matrix(apikey,Places)
25     with open(File,'w') as f:
26         for i,origin in enumerate(Places):
27             for j,destination in enumerate(Places):
28                 if(origin!=destination):
29                     tempstring="pode_ir(\'{ }\',\'{ }\',{ }).\n"
30                     f.write(tempstring.format(origin,destination,distance_matrix[i][j]))
31     return distance_matrix
32
33 def test_distance_matrix(apikey):
34     """ Teste da funcao places_distance_matrix"""
35     cities=['Ribeirao Preto','Cravinhos','Batatais','Sao
36     ↪ Carlos','Bauru','Rifaina','Maringa']
37     a= places_distance_matrix(apikey,cities)
38     print(a)
39
40 def test_construct_rules(apikey):
41     cities=['Ribeirao Preto','Cravinhos','Batatais','Sao
42     ↪ Carlos','Bauru','Rifaina','Maringa']
43     b=construct_rules("regras.pl",apikey,cities)

```

3 Solução

Modelado nosso problema e definido as regras, finalmente podemos começar a resolvê-lo. Desenvolvemos dois programas em Prolog que encontram o menor caminho para o TSP. O primeiro programa utiliza uma busca cega sem heurística, visitando todas as possibilidades de seu espaço de busca antes de encerrar sua execução. A segunda estratégia foi utilizar do algoritmo A* que sempre reavalia os possíveis melhores caminhos;

3.1 Busca Cega

Para a busca cega, foi testada tanto a busca em profundidade quanto a busca em largura para avaliar qual apresenta o melhor resultado. O código está representado na Listing 3.1

```
1  buscacega_profundidade(I,Cam,TempoExecucao):-
2      get_time(TempoInicial),
3      findall(TamCaminho,busca_Caminho(I,F,_,_,TamCaminho),ListaTam),m
   ↪ axLista(ListaTam,TamCaminho_Max),
4      findall(CustoCaminho,busca_Caminho(I,F,_,_,CustoCaminho,TamCaminho_
   ↪ _Max),ListaCusto),minLista(ListaCusto,CustoCaminho_Min),
5      busca_Caminho(I,F,Cam,CustoCaminho_Min,TamCaminho_Max),get_tim
   ↪ e(TempoFinal),!,TempoExecucao is
   ↪ TempoFinal-TempoInicial.
6
7  busca_Caminho(I,F,Cam,CustoTotal,Tamanho):-
8      CustoInicial is 0,
9      caminho(I,[F],Cam,[CustoInicial],CustoFinal),somaElem_Lista(Cust
   ↪ oFinal,CustoTotal),tamLista(CustoFinal,Tamanho).
10     caminho(I,[I|Caminho],[I|Caminho],CustoCaminho,CustoCaminho).
11     caminho(I,[Ult_Estado|Caminho_ate_agora],Cam,[Ult_Custo|Custo_at
   ↪ e_agora],CustoFinal):-
12         pode_ir(Est,Ult_Estado,Custo),
13         not( pertence1(Est,Caminho_ate_agora)),
14         caminho(I,[Est,Ult_Estado|Caminho_ate_agora],Cam,[Custo,Ult
   ↪ _Custo|Custo_ate_agora],CustoFinal).
15
16     %Tamanho da lista
17     tamLista([], 0):- !.
18     tamLista([_|L], T):- tamLista(L, X), T is X + 1.
19
20     %Verifica se uma elemento pertence a lista
21     pertence1(E,[E|_]):- !.
22     pertence1(E,[_|T]):-
23         pertence1(E,T).
24
```

```

25 %Soma todos os elementos de uma lista
26 somaElem_Lista([],0).
27 somaElem_Lista([Elem|Cauda],Soma):-
28     somaElem_Lista(Cauda,Som),
29     Soma is Som+Elem.
30
31 %Menor elemento de uma lista
32 minLista([X],X).
33 minLista([X|Y],M):-
34     minLista(Y,N),(X<N -> M=X;M=N).
35
36 %Maior elemento de uma lista
37 maxLista([X],X).
38 maxLista([X|Y],M):-
39     maxLista(Y,N),(X>N -> M=X;M=N).

```

3.2 Busca Informada

Para fazer a busca informada foi utilizada a estratégia A*. Essa estratégia foi utilizada por trabalhar com caminhos, e não apenas com um resultado final, o que é ideal para o problema. Também é interessante por não retornar um resultado aproximado, mas sim um resultado ótimo, desde que seja utilizada uma função de avaliação admissível. Um ponto negativo desta estratégia é a quantidade de memória utilizada já que todos os caminhos tentados são armazenados (embora apenas um caminho esteja sendo desenvolvido por vez). Em nossos testes, não foi possível encontrar a solução para mais de 7 cidades. A heurística utilizada para guiar a busca A* foi a de vizinho mais próximo. Nesta, a função de avaliação é sempre zero (e portanto é admissível) e a função de custo é a distância entre as cidades. (Foi implementada também a heurística de inserção mais barata, fazendo a função de avaliação ser a distância em linha reta entre as cidades, no entanto essa apresentou uma performance pior, certamente devido à maneira como foi implementada.) O código foi representado pelo Listing 3.2.

```

1 buscaCaminho(Inicio,Caminho,Tempo):-
2     get_time(TempoInicial),
3     calculaCusto(Inicio,FCusto),           %Calcula todos os custos a
      ↳ partir do nó inicial
4     calculaAvaliacao(Inicio,FAvaliacao),
5     heuristica(FCusto,FAvaliacao,PossiveisCaminhos), %Calcula todas
      ↳ as heurísticas a partir do nó inicial
6     reordenaCaminhos(PossiveisCaminhos,CaminhosOrdenados), %Coloca
      ↳ caminho ativo no topo da lista

```

```

7      encontraNos(Nos), %Recebe lista com todas as cidades
      ↪ (para saber condição de parada)
8      buscaCaminho(Inicio,Caminho,CaminhosOrdenados,Nos),
9      get_time(TempoFinal),
10     Tempo is TempoFinal-TempoInicial,!
11
12 buscaCaminho(Inicio,CaminhoAtivo,[_|CaminhoAtivo|_],Nos):-
13     verificaFim([Inicio|CaminhoAtivo],Nos).
14
15 buscaCaminho(Inicio,Caminho,[[CustoAtivo|CaminhoAtivo]|Outros],Nos):-
16     removeElemento([[CustoAtivo|CaminhoAtivo]|Outros],[CustoAtivo|Cam_
17     ↪ inhoAtivo],CaminhosRestantes),
18     expandeCaminhoAtivo(CaminhosRestantes,[CustoAtivo|CaminhoAtivo],Cam_
19     ↪ inhosExtendidos),
20     eliminaCaminhosIrregulares(CaminhosExtendidos,CaminhosCorrigido_
21     ↪ s,Inicio),
22     reordenaCaminhos(CaminhosCorrigidos,CaminhosOrdenados),
23     buscaCaminho(Inicio,Caminho,CaminhosOrdenados,Nos).
24
25 %Calcula função de custo de todos os caminhos conectados à Cidade
26 calculaCusto(Cidade,FCusto):-
27     findall([Y,X],pode_ir(Cidade,X,Y),FCusto).
28
29 %Calcula função de avaliação de todos os caminhos conectados à Cidade
30 calculaAvaliacao(Cidade,FAvaliacao):-
31     findall([0,X],pode_ir(Cidade,X,_),FAvaliacao).
32
33 %Calcula função heurística combinando FCusto e FAvaliacao em
34 ↪ PossiveisCaminhos
35 heuristica([[Custo,Cidade]|CaudaC],[[Avaliacao,Cidade]|CaudaA],[[Heur_
36 ↪ ística,Cidade]|CaudaH]):-
37     Heuristica is Custo+Avaliacao,
38     heuristica(CaudaC,CaudaA,CaudaH).
39
40 heuristica([[Custo,Cidade]],[[Avaliacao,Cidade]],[[Heuristica,Cida_
41 ↪ de]]):-
42     Heuristica is Custo+Avaliacao.
43
44 %Coloca caminho ativo no começo da lista
45 reordenaCaminhos(PossiveisCaminhos,[Melhor|Outros]):-
46     menorHeuristica(PossiveisCaminhos,Melhor),
47     removeElemento(PossiveisCaminhos,Melhor,Outros).
48
49 %Retorna caminho ativo
50 menorHeuristica([Melhor],Melhor).

```



```

48 menorHeuristica([Custo1|Melhor1],[Custo2|_],[Custo1|Melhor1):-
49     Custo1<Custo2,!
50 menorHeuristica([Custo1|_],[Custo2|Melhor2],[Custo2|Melhor2):-
51     Custo2<Custo1,!
52 menorHeuristica([Caminho1|OutrosCaminhos],Melhor):-
53     menorHeuristica(OutrosCaminhos,MelhorAux1),
54     menorHeuristica([Caminho1,MelhorAux1],Melhor).
55
56
57 %Remove elemento Melhor de PossiveisCaminhos e gera Outros
58 removeElemento([Melhor|Outros],Melhor,Outros).
59 removeElemento([Cabeca|Cauda],Melhor,[Cabeca|Outros]):-
60     removeElemento(Cauda,Melhor,Outros).
61
62
63 %Retorna todos as cidades do mapa
64 encontraNos(Nos):-
65     setof(TodasCidades,X^Y^pode_ir(TodasCidades,X,Y),Nos).
66
67
68 %Verifica se CaminhoAtivo é a solução.
69 verificaFim([],[]).
70 verificaFim([Cabeca|Cauda],Nos):-
71     contemElemento(Cabeca,Nos),
72     removeElemento(Nos,Cabeca,NovosNos),
73     verificaFim(Cauda,NovosNos).
74
75
76 contemElemento(Elemento,[Elemento|_]).
77 contemElemento(Elemento,[_|Cauda]):-
78     contemElemento(Elemento,Cauda).
79
80
81 %Concatena os novos caminhos possíveis a partir do caminho ativo
82 expandeCaminhoAtivo(CaminhosRestantes,[CustoAtivo|CaminhoAtivo],Cam
83     ↪ inhosExtendidos):-
84     ultimoNo(CaminhoAtivo,Ultimo),
85     calculaCusto(Ultimo,FCustoExtendido),
86     calculaAvaliacao(Ultimo,FAvaliacaoExtendida),
87     heuristica(FCustoExtendido,FAvaliacaoExtendida,PossiveisCaminhosExt
88     ↪ endidos),
89     reconstroiCaminhos([CustoAtivo|CaminhoAtivo],PossiveisCaminhosExt
90     ↪ endidos,CaminhosReconstruidos),
91     concatenaLista(CaminhosRestantes,CaminhosReconstruidos,CaminhosExt
92     ↪ endidos).
93
94 ultimoNo([E],E).

```

```

92 ultimoNo([_|Cauda],E):-
93     ultimoNo(Cauda,E).
94
95
96 reconstroiCaminhos([CustoAtivo|CaminhoAtivo],[[CustoExtendido|Caminho
97     ↳ Extendido]],[[Custo|Caminhos]]):-
98     Custo is CustoAtivo+CustoExtendido,
99     concatenaLista(CaminhoAtivo,CaminhoExtendido,Caminhos).
100
101 reconstroiCaminhos([CustoAtivo|CaminhoAtivo],[[CustoExtendido|Caminho
102     ↳ Extendido]|OutrosCaminhosExtendidos],[[Custo|Caminhos]|OutrosCam
103     ↳ inhos]]):-
104     Custo is CustoAtivo+CustoExtendido,
105     concatenaLista(CaminhoAtivo,CaminhoExtendido,Caminhos),
106     reconstroiCaminhos([CustoAtivo|CaminhoAtivo],OutrosCaminhosExt
107     ↳ endidos,OutrosCaminhos).
108
109
110
111 concatenaLista([],L,L).
112 concatenaLista([Cabeca1|Cauda1],L2,[Cabeca1|Cauda3]):-
113     concatenaLista(Cauda1,L2,Cauda3).
114
115
116
117 %Elimina todos os caminhos expandidos que não podem ser caminhos
118 ↳ finais, criando a lista CaminhosCorrigidos
119 eliminaCaminhosIrregulares([],[],_):-!.
120
121
122
123 eliminaCaminhosIrregulares(Caminhos,[],Inicio):-
124     caminhoValido(Inicio,Caminhos,NovosCaminhos),
125     tamanhoLista(NovosCaminhos,0).
126
127
128
129 eliminaCaminhosIrregulares(Caminhos,[CaminhoCorrigido|OutrosCorrigido
130     ↳ s],Inicio):-
131     caminhoValido(Inicio,Caminhos,CaminhoCorrigido),
132     removeElemento(Caminhos,CaminhoCorrigido,NovosCaminhos),
133     eliminaCaminhosIrregulares(NovosCaminhos,OutrosCorrigidos,Inicio).
134
135
136
137 %Retorna um caminho válido de Caminhos ou lista vazia.
138 caminhoValido(_,[],[]).
139
140
141
142 caminhoValido(Inicio,[[Custo|Caminho]|_],[Custo|Caminho]):-
143     not(repeteElemento([Inicio|Caminho])),!.
144
145
146
147 caminhoValido(Inicio,[_|Cauda],CaminhoCorrigido):-
148     caminhoValido(Inicio,Cauda,CaminhoCorrigido).
149
150
151
152
153

```

```

134 %True se há repetição
135 repeteElemento([]):-fail.
136 repeteElemento(Lista):-
137     setof(Elemento,X^removeElemento(Lista,Elemento,X),Laux),
138     tamanhoLista(Laux,Num),
139     not(tamanhoLista(Lista,Num)).
140
141
142
143 tamanhoLista([],0).
144 tamanhoLista([_|Cauda],N):-tamanhoLista(Cauda,Naux), N is 1+Naux.

```

A fim de testar e avaliar o desempenho dos algoritmos criados, desenvolvemos um programa que cria uma classe que executa todos os códigos anteriores, isto é cria as regras do caminho especificado e executa as buscas. Assim, temos de forma acessível o tempo que cada algoritmo levou para ser executado, o caminho resultante e a distância em metros que é a soma dos custos no estado final. Como o Prolog é limitado, esta comunicação foi feita a partir da criação de arquivos auxiliares. O código está representado na Listing 3.2

```

1  import time,os,subprocess
2  import DistanceMatrix
3  import re
4  import os
5
6  class Busca:
7      """Representa um metodo de busca"""
8      def __init__(self,caminho,tempo):
9          self.caminho=caminho
10         self.tempo=tempo
11
12
13     class ResultadoBuscas:
14         """Representa o resultado de todos os metodos de busca
15         ↪ implementados """
16
17         def __init__(self,cities,inicial,apikey):
18             #Lista com Cidades
19             self.cities=cities
20             #Cidade de partida
21             self.inicial=inicial
22             #Usado para definir nomes em tempo de execução, necessário
23             ↪ para não ter conflitos no server web
24             stamp=time.strftime('%a%d%b%Y%H%M%S')
25             stampedname="regras"+stamp
26             stampedfilecega="buscacega"+stamp+".out"

```

```

25 stampedfilea="buscaa"+stamp+".out"
26 #Matriz de Conectividade e construindo as regras
27 self.distancematrix=DistanceMatrix.construct_rules(stampednam
    ↳ e+".pl",apikey,cities)
28 #Definindo caminhos usando prolog:
29 argstring='consult({}),consult({}),open(\'{}\','write,Stream
    ↳ ),{ }\('{}\','Y,Z),write(Stream,X),nl(Stream),write(Stream
    ↳ ,Y),nl(Stream),write(Stream,Z),close(Stream),halt.'
30 subprocess.call(['swipl','--quiet','-t',argstring.format(stam
    ↳ pedname,'buscacega_profundidade',stampedfilecega,'busca
    ↳ cega_profundidade',cities[inicial]))
31 subprocess.call(['swipl','--quiet','-t',argstring.format(stam
    ↳ pedname,'buscaInformada_A',stampedfilea,'buscaCaminho
    ↳ ',cities[inicial]))
32 #Lendo dos arquivos de output do prolog e salvando em
    ↳ elementos da classe
33 a=[]
34 b=[]
35 with open(stampedfilecega,'r') as f:
36     for line in f:
37         buffer=line
38         a.append(buffer)
39         with open(stampedfilea,'r') as f:
40             for line in f:
41                 buffer=line
42                 b.append(buffer)
43 #Manipulando a string para lista (e excluindo o primeiro no
    ↳ para padronizar Busca A* e Busca Cega)
44 caminhotemp=re.split(',',a[1].strip('[]\n'))
45 caminhotemp.pop(0)
46 self.bcega=Busca(caminhotemp,a[2])
47 caminhotemp=re.split(',',b[1].strip('[]\n'))
48 self.baestrela=Busca(caminhotemp,b[2])
49 #Salvando a distancia total da nossa solucao
50 self.distancia=self.soma_caminho()
51 #Limpendo arquivos auxiliares
52 os.remove(stampedfilea)
53 os.remove(stampedfilecega)
54 os.remove(stampedname+".pl")
55
56 def soma_caminho(self):
57     #Mapeamento das cidades
58     citiesdicio={j:i for i,j in enumerate(self.cities)}
59     #Reinserindo No inicial no nosso caminho
60     caminhocompleto=self.bcega.caminho
61     caminhocompleto.insert(0,self.cities[self.inicial])
62     soma=0
63     for n,cidades in enumerate(caminhocompleto):

```

```

64         #Queremos calcular a distancia entre o termo n e n+1, sem
        ↳ incluir do ultimo pro primeiro
65     if n!=len(caminhocompleto)-1):
66         #Encontrando os indices correspondentes da matriz de
        ↳ conectividade
67         cidadeorigem=citiesdicio[caminhocompleto[n]]
68         cidadedestino=citiesdicio[caminhocompleto[n+1]]
69         soma=soma+self.distancematrix[cidadeorigem][cida_
        ↳ dedestino]
70     return soma
71
72     def test():
73         a= ResultadoBuscas(['Maracana','Sao
        ↳ Paulo','Bebedouro','Matao'],0)
74         print(a.cities,a.inicial,a.bcega.caminho,a.baestrela.tempo,a_
        ↳ .soma_caminho())
75     return

```

4 Apresentação dos resultados

Como um terminal de query Prolog parece críptico, pouco intuitivo e amigável para o usuário final, escolhemos desenvolver uma aplicação Web que reutiliza-se todos os nosso códigos. Para isso, foi escolhido um micro-framework Web, em Python, chamado *Flask*. Flask define rotas e executa uma função quando uma requisição *HTTP* acontece. Assim, nosso código contido em *PrologIO.py* executa e retorna uma página, com template predefinido por nós, renderizada com todos os dados relevantes contidos da classe *ResultadosBusca*, de forma legível e amigável. O código em Flask é dado abaixo:

```

1  from flask import Flask
2  from flask import render_template
3  from flask import request
4  from PrologIO import ResultadoBuscas
5  from unicode import unicode
6  import re
7  app= Flask(__name__)
8
9  @app.route('/',methods=['GET','POST'])
10 def index():
11     apikey="AIzaSyDnQndjPZDiERjXPdOmA5TAy5sVzk2rFqc"
12     error=None
13     if request.method == 'POST':
14         #Filtrando entrada recebida do usuário
15         a=request.form['resultado'].strip('\["')
16         a=a.strip('\]\ "')

```

```
17         b=re.split('\\"',a)
18         c=[re.sub('\,',',',a) for a in b]
19         cidades=[unicode(d) for d in c]
20         busca=ResultadoBuscas(cidades,0,apikey)
21         return render_template('results.html',busca=busca)
22     return render_template('application.html')
```

Escolhemos o framework front-end (CSS/Javascript) *Materialize* para embelezar nossa aplicação.

5 Conclusão

Podemos concluir que, para o problema do caixeiro-viajante, a estratégia de busca A^* pode ser muito cara pois o consumo de memória cresce muito a partir de 8 nós se não utilizar uma heurística para evitar a expansão de todos os caminhos (o limite do prolog para sistemas 64 bits, onde foi testado, é de 256Mb. Este limite foi expandido para 2GB e mesmo assim com 8 nós ocorreu falha por falta de memória no primeiro teste). A heurística utilizada (vizinho mais próximo) permitiu que o problema fosse resolvido com 9 nós.

Quanto a busca em profundidade, ela possui uma vantagem de não ocupar tanta memória, o que permite encontrar soluções para problemas com mais nós. No entanto, o número de possibilidades a serem testadas é um grande problema (foi possível solucionar problemas com 10 nós utilizando busca em profundidade, mas o tempo gasto ultrapassava 1 minuto, o que tornou inviável fazer diversas medições para tirar uma média de tempo.)