Engenharia de Computação

Inteligência Artificial

Trabalho de Implementação

Helena Tavares

Maio de 2022

1. Objetivo

Escolher dois algoritmos de busca, um com informação, outro sem informação e implementar em uma linguagem de preferência, que no caso será Python.

Cada algoritmo deve executar três árvores de busca de tamanhos diferentes, e seus tempos de execução devem ser comparados. Para análise e comparação, serão criados gráficos.

2. Desenvolvimento

Os algoritmos escolhidos foram busca em profundidade (sem informação) e busca gulosa (com informação) pela facilidade de implementação e pelo contraste de desempenho entre eles, para melhor análise.

Busca em profundidade é um algoritmo que percorre os itens dentro das árvores ou grafos de modo que todos os nós filhos do nó raiz, são percorridos até o nó mais profundo possível, e depois retorna para continuar a busca.

Busca gulosa percorre os nós vizinhos que parecem ter a menos distância do objeto com base na estimativa feita pela função heurística h.

Foram implementadas árvores binárias com diferentes números de nodos: 6, 14 e 30, preenchidas em ordem crescente. As distâncias entre os nós em relação ao nó destino também foram definidas.

Os pontos de destino e origem foram respectivamente:

arvore1: 0 e 3

arvore2: 0 e 12

arvore3: 0 e 12

buscaEmProfundidade.py > ...

```
1   #                   arvore1
2   #                     0
3   #
4   #           1                   2
5   #
6   #     3           4           5           6
```

buscaEmProfundidade.py > ...

```
1   #                   arvore2
2   #                     0
3   #
4   #           1                   2
5   #
6   #     3           4           5           6
7   #
8   #  7     8     9     10    11    12    13    14
```

buscaEmProfundidade.py > ...

```
1   #                   arvore3
2   #                     0
3   #
4   #           1                   2
5   #
6   #     3           4           5           6
7   #
8   #  7     8     9     10    11    12    13    14
9   #
10  # 15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30
```

2.1. Implementação BuscaEmProfundidade

buscaEmProfundidade.py

```python
import time

origem = '0'
destino = '12'

dist = 0
testado = set()

def buscaEmProfundidade(testado, graph, node, search):
    global dist
    if node not in testado:
        print (node)
        testado.add(node)
        dist+=1

        if node == search:
            print('Destino Encontrado')
            fim = time.time()
            print('Distância percorrida: %d' %dist)
            print("Tempo de execução: %fs" % (fim - inicio))
        for vizinho in graph[node]:
            buscaEmProfundidade(testado, graph, vizinho, search)
```

A biblioteca time é importada a fim de fornecer meios de contabilizar o tempo de execução.

As variáveis de origem e destino são definidas, assim como a que vai medir a distância percorrida durante a busca e os nodos visitados são armazenados na variável testado.

Na função da busca em profundidade, os nodos visitados, o grafo a ser percorrido, o nodo de origem e de destino são inicializados. A variável do somador da distância inicializada e começam os testes para descobrir se o nodo atual já foi visitado anteriormente, se não foi visitado os testes continuam.

Logo depois, é verificado se o nodo atual é o destino buscado, se sim, o programa imprime o tempo de execução e o caminho percorrido entre origem e destino. Se não, um loop é iniciado recursivamente, chamando o próximo nodo da árvore.

```
arvore1 =  {
    '0' : ['1','2'],
    '1' : ['3', '4'],
    '2' : ['5', '6'],
    '3' : [],
    '4' : [],
    '5' : [],
    '6' : []
}

arvore2 =  {
    '0' : ['1','2'],
    '1' : ['3', '4'],
    '2' : ['5', '6'],
    '3' : ['7', '8'],
    '4' : ['9', '10'],
    '5' : ['11', '12'],
    '6' : ['13', '14'],
    '7' : [],
    '8' : [],
    '9' : [],
    '10' : [],
    '11' : [],
    '12' : [],
    '13' : [],
    '14' : []
}

arvore3 =  {
    '0' : ['1','2'],
    '1' : ['3', '4'],
    '2' : ['5', '6'],
    '3' : ['7', '8'],
    '4' : ['9', '10'],
    '5' : ['11', '12'],
    '6' : ['13', '14'],
    '7' : ['15', '16'],
    '8' : ['17', '18'],
    '9' : ['19', '20'],
    '10' : ['21', '22'],
    '11' : ['23', '24'],
    '12' : ['25', '26'],
    '13' : ['27', '28'],
    '14' : ['29', '30'],
    '15' : [],
    '16' : [],
    '17' : [],
    '18' : [],
    '19' : [],
    '20' : [],
    '21' : [],
    '22' : [],
    '23' : [],
    '24' : [],
    '25' : [],
    '26' : [],
    '27' : [],
    '28' : [],
    '29' : [],
    '30' : []
}
```

Nas linhas de código 24 até a 84, são criadas as árvores binárias de 6, 14 e 30 nodos respectivamente, preenchidas utilizando estruturas de dicionário do Python.

```
86    inicio = time.time()
87    buscaEmProfundidade(testado, arvore2, origem, destino)
```

Após a criação das árvores, o contador de tempo é startado e então a função buscaEmProfundidade é chamada.

2.2. Implementação BuscaGulosa

```
 6   arvore1 =  {
 7       0 : [(1,1),(2,3)],
 8       1 : [(3,0),(4,2)],
 9       2 : [(5,4),(6,4)],
10       3 : [],
11       4 : [],
12       5 : [],
13       6 : []
14   }
15
16   arvore2 =  {
17       0 : [(1,4),(2,3)],
18       1 : [(3,5),(4,5)],
19       2 : [(5,1),(6,2)],
20       3 : [(7,6), (8,6)],
21       4 : [(9,6), (10,6)],
22       5 : [(11,2), (12,1)],
23       6 : [(13,4), (14,4)],
24       7 : [],
25       8 : [],
26       9 : [],
27       10 : [],
28       11 : [],
29       12 : [],
30       13 : [],
31       14 : []
32   }
```

```python
arvore3 =  {
    0 : [(1,4),(2,3)],
    1 : [(3,5),(4,5)],
    2 : [(5,1),(6,2)],
    3 : [(7,6), (8,6)],
    4 : [(9,6), (10,6)],
    5 : [(11,2), (12,0)],
    6 : [(13,4), (14,4)],
    7 : [(15,7), (16,7)],
    8 : [(17,7), (18,7)],
    9 : [(19,7), (20,7)],
    10 : [(21,7), (22,7)],
    11 : [(23,3), (24,3)],
    12 : [(25,1), (26,1)],
    13 : [(27,5), (28,5)],
    14 : [(29,5), (30,5)],
    15 : [],
    16 : [],
    17 : [],
    18 : [],
    19 : [],
    20 : [],
    21 : [],
    22 : [],
    23 : [],
    24 : [],
    25 : [],
    26 : [],
    27 : [],
    28 : [],
    29 : [],
    30 : []
}
```

Os 3 tamanhos de árvores sendo criados e preenchido com duplas de números inteiros que sinalizam o valor armazenado e a distância do nodo até o destino.

```python
def buscaGulosa(graph, node, search):
    nodoAtual = node
    caminho = []
    caminho.append(nodoAtual)
    distancia = 0
    while len(set([nodoVizinho for (nodoVizinho, distance) in graph.get(nodoAtual, [])]).difference(set(caminho))) > 0:
        vizinhoProximo = None
        menorDistancia = None
        for vizinho, vizinhoDistancia in graph[nodoAtual]:
            if vizinho != nodoAtual and vizinho not in caminho:
                if menorDistancia is not None:
                    if menorDistancia > vizinhoDistancia:
                        menorDistancia = vizinhoDistancia
                        vizinhoProximo = vizinho
                else:
                    menorDistancia = vizinhoDistancia
                    vizinhoProximo = vizinho
        vizinhoMaisProximo = (vizinhoProximo, menorDistancia)
        nodoAtual = vizinhoMaisProximo[0]
        caminho.append(nodoAtual)
        distancia += vizinhoMaisProximo[1]
        if nodoAtual == search:
            fim = time.time()
            print("Caminho percorrido: %s" %caminho)
            print("Distância percorrida: %d" %distancia)
            print("Tempo de execução %fs" %(fim - inicio))
            return 0

inicio = time.time()
buscaGulosa(arvore3, origem, destino)
```

A biblioteca time é importada para contabilizar o tempo de execução, as variáveis de origem e destino são iniciadas assim como o contador da distância percorrida e a lista de novos que forem sendo percorridos.

Na busca gulosa o parâmetro de entrada é o grafo a ser percorrido, o ponto de origem e de destino. O vetor de armazenamento do caminho percorrido e o contador da distância percorrida.

Ao longo do loop é testado o caminho, a fim de encontrar o mais curto, em relação aos próximos a serem percorridos até o nodo de destino. A cada teste, também é testado se o nodo atual é o destino, quando for, é impresso o caminho e a distância do mesmo.

O medidor de tempo é startado e a função é chamado.

3. Resultados

3.1. Busca em profundidade

Para encontrar o nodo 3 na árvore de 6 nodos, o caminho percorrido foi 0>1> 3. Totalizando 3 nodos de distância.

Na árvore de 14 nodos, para encontrar o valor 12, o caminho percorrido foi 0>1>3>7>8>4>9>10>2>5>11>12. Totalizando 12 nodos de distância.

Na árvore de 30 nodos, para encontrar o valor 12, o caminho percorrido foi 0>1>3>7>15>16>8>17>18>4>9>19>20>10>21>22>2>5>11>23>24. Totalizando 22 nodos de distância.

3.1.1. Tempo de execução

Tamanho da Árvore	6	14	30
	0.000013	0.000027	0.000049
	0.000027	0.000043	0.000050
	0.000038	0.000031	0.000058

3.2. Busca Gulosa

Para encontrar o nodo 3 na árvore de 6 nodos, o caminho percorrido foi 0 > 1 >3. Totalizando 3 nodos de distância.

Na árvore de 14 nodos, para encontrar o valor 12, o caminho percorrido foi 0>2>5>12. Totalizando 4 nodos de distância.

Na árvore de 30 nodos, para encontrar o valor 12, o caminho percorrido foi 0>2>5>12. Totalizando 4 nodos de distância.

3.2.1. Tempo de execução

Tamanho da Árvore	6	14	30
	0. 000004	0. 000006	0. 000007
	0. 000004	0. 000006	0. 000007
	0. 000004	0. 000006	0. 000007

4. Conclusão

De acordo com os resultados obtidos nas medições de tempo, a busca com informação apresenta um melhor desempenho, pois independente do número de nodos (tamanho da árvores) o tempo de execução não aumenta de forma tão significativa quanto na busca sem informação. Para árvores que não dependam tanto do tempo de execução ou que sejam de tamanho pequeno, a busca sem informação pode ser utilizada, mas para árvores maiores ou situações que o tempo de execução seja importante, é mais adequado utilizar o algoritmo de busca com informação (buscaGulosa).