Universidade Federal do Piauí

Ciência da Computação

Bruno Felipe da Silva Celestino

Inteligência Artificial - 1º Trabalho Prático

Busca em Espaço de Estados

Teresina, 2021

Resumo

O objetivo desta atividade é construir um algoritmo capaz de resolver o problema do "jogo dos 8" (8 puzzle) de forma automática, com o uso de ferramentas de inteligência artificial. São utilizadas quatro abordagens de busca, duas de força brusca (busca em largura e em profundidade), também é utilizada uma abordagem de busca gulosa (juntamente com uma heurística) e uma busca A*. O funcionamento das implementações de cada método de busca é descrito adiante. Também são apresentados dados do desempenho de cada busca, como número de iterações, memória e tempo necessários para chegar à solução, até a própria solução é usada para avaliar o desempenho. Com esses resultados são feitas comparações entre os métodos usados, de modo a determinar e avaliar a eficiência de cada um.

Introdução

O “jogo dos 8” (puzzle 8) é um jogo que consiste em uma malha 3x3, ou seja, com nove espaços, quadrados. Um dos quadrados é vazio, o que permite o movimento de quadrados adjacentes para essa posição. O objetivo do jogo é mover esses quadrados até que a disposição deles seja igual a um “estado final”. A seguir, uma representação de como é o estado final.

1	**2**	**3**
4	**5**	**6**
7	**8**	

Foram implementadas quatro formas de solucionar o jogo. As formas implementadas são: busca em largura, busca em profundidade, busca gulosa com heurística e busca A* (A estrela).

Busca em largura é uma estratégia simples em que o nó raiz é expandido primeiro, em seguida todos os sucessores do nó raiz são expandidos, depois os sucessores desses nós, e assim por diante. A busca é feita em todos os nós de um nível, antes que se possa buscar em qualquer nó do nível seguinte.

Busca em profundidade sempre expande o nó mais profundo na borda atual da árvore de busca. Procura explorar completamente cada ramo da árvore antes de tentar o ramo vizinho. Os últimos nós explorados são os primeiros a serem expandidos, a busca continua nesses sucessores recém expandidos.

Busca gulosa tenta expandir o nó que está mais próximo do objetivo, com o fundamento de que isso pode conduzir a uma solução rapidamente. Com o uso de heurística, esse método avalia os nós apenas a função heurística (h(n)), escolhendo aquele que julga ser o melhor valor.

Busca A* avalia os nós através da combinação de g(n), o custo para alcançar o nó, e h(n), o custo estimado para ir do nó ao objetivo. No caso desse problema, o nó a ser escolhido deve ser aquele que apresentar o menor valor de g(n)+h(n).

Implementação

A implementação de todos os métodos de busca foi feita em Python. O estado inicial é adicionado através de um input de 9 caracteres numéricos, que é transformado em uma matriz que represente um estado do tabuleiro do jogo dos 8.

Uma entrada deste tipo:

```
Informe o problema: 123450786
```

É transformada em um valor deste tipo:

```
{0:[1, 2, 3], 1:[4, 5, 0], 2:[7, 8, 6]}
```

Que representa isso:

1	**2**	**3**
4	**5**	
7	**8**	**6**

Esse valor é usado como atributo de um elemento do tipo NodoArvore, que é uma classe criada para representar os nós usados nas buscas.

```
class NodoArvore:
    def __init__(self, pos0, chave=None, pai=None, depth = None):
        self.chave = chave
        self.pai = pai
        self.filhos = []
        self.pos0 = pos0
        if depth == None:
            self.depth = 0
        else:
            self.depth = depth + 1

    def geraFilho(self, pos0, chave, pai):
        filho = NodoArvore(pos0, chave, pai, pai.depth)
        self.filhos.append(filho)
        return filho

    def getHeu(self, h2):
        tabHeu = []
        for i in range(len(self.chave)):
            for j in range(len(self.chave[0])):
                tabHeu.append(h2[j+3*i][self.chave[i][j]])
        return sum(tabHeu)
```

Existem métodos auxiliares que ajudam no processo de expansão da árvore de busca e na visualização dos passos.

```
''' acha a posição do zero, a peça vazia '''
def getZero(tabuleiro):
    for i in range(3):
        for j in range(3):
            if tabuleiro[i][j] == 0: return [i, j]

pos0 = getZero(tabuleiro)

''' realiza o movimento, trocando posições de elementos, peças '''
def movimento(tabul, pos, l, c, tree):
    tabuleiro = copy.deepcopy(tabul)
    tabuleiro[pos[0]][pos[1]] = tabuleiro[l][c]
    tabuleiro[l][c] = 0
    return tree.geraFilho([l, c], tabuleiro, tree)

''' verifica quantos movimentos são possíveis e os faz '''
def decisao(tabul, pos, tree):
    tb = copy.deepcopy(tabul)
    nivel = []
    if pos[0] > 0:
        nivel.append(movimento(tb, pos, pos[0]-1, pos[1], tree))
    if pos[0] < 2:
        nivel.append(movimento(tb, pos, pos[0]+1, pos[1], tree))

    if pos[1] > 0:
        nivel.append(movimento(tb, pos, pos[0], pos[1]-1, tree))
    if pos[1] < 2:
        nivel.append(movimento(tb, pos, pos[0], pos[1]+1, tree))

    return nivel

''' visualização de um estado '''
def exibe(tbl):
    for i in range(3):
        print(tbl[i])
```

Ao fim da execução os resultados são exibidos da seguinte maneira:

```
a solucao:  {0: [1, 2, 3], 1: [4, 5, 6], 2: [7, 8, 0]}
            {0: [1, 2, 3], 1: [4, 5, 0], 2: [7, 8, 6]}
            {0: [1, 2, 3], 1: [4, 0, 5], 2: [7, 8, 6]}
            {0: [1, 2, 3], 1: [0, 4, 5], 2: [7, 8, 6]}
            {0: [0, 2, 3], 1: [1, 4, 5], 2: [7, 8, 6]}
            {0: [2, 0, 3], 1: [1, 4, 5], 2: [7, 8, 6]}
Solução na profundidade:  5
número de nós gerados:  28
número de iterações:  11
tempo:  2.034630298614502 s
```

Em resultados temos os passos da solução, a profundidade da solução encontrada, o número de iterações necessários, o número de nós expandidos e o tempo levado.

Busca em Largura

Basicamente é composta por dois loops, o loop mais externo fica armazenando os sucessores do nível atual para serem usados na próxima iteração, enquanto o loop mais interno testa cada nó do nível e gera os sucessores desse nó.

```
inicio = time.time()
while flag == 1:
    if len(lista) < 2:
        print('a solução: ', lista[0].chave)
        print('Fim')
        break
    arvs = copy.deepcopy(lista)
    lista = []
    # testa cada nó de um nível #
    for estado in arvs:
        lacos += 1
        if estado.chave not in explorados:
            print('profundidade: ', estado.depth)
            exibe(estado.chave)
            time.sleep(0.7)
            os.system('clear')
            verificacoes += 1
            explorados.append(estado.chave)
            aux = []
            aux += busca(estado)
            numN += len(estado.filhos)
            if len(aux) < 2:
                verificacoes += 1
                print('Solução na profundidade: ', estado.depth)
                print('a solucao: ', estado.chave)
                # gerando o caminho do início até a solução #
                while(aux[0].pai):
                    print('           ',aux[0].pai.chave)
                    aux[0] = aux[0].pai
                print('fim')
                flag = 0
                break
            lista += aux # os filhos de um nó testado são armazenados aqui para o teste do próximo nível
fim = time.time()
```

Busca em Profundidade

Utiliza de recursividade. Recebe um nó, gera seus sucessores e, para cada sucessor, se chama novamente com um sucessor como parâmetro. Retorna quando achar o estado final ou um nó que não possui sucessores. Neste caso a profundidade é limitada para tentar evitar loops, então caso uma profundidade determinada seja atingida e a solução não é encontrada a função retorna vazio.

```
def solucaoProfundidade(tree):
    global iteracoes
    global verificacoes
    global numN

    print('profundidade: ', tree.depth)
    exibe(tree.chave)
    time.sleep(0.7)
    os.system('clear')

    if tree.chave == tblTeste: return tree

    sol = None
    filhos = decisao(tree.chave, tree.pos0, tree)
    numN += len(filhos)

    for no in filhos:
        iteracoes += 1
        verificacoes += 1

        if no.chave == tblTeste: return no
        if no.depth > 7: continue

        print('profundidade: ', no.depth)
        exibe(no.chave)
        time.sleep(0.7)
        os.system('clear')

        sol = solucaoProfundidade(no)
        if sol != None: return sol
```

Busca Gulosa

Guarda informação de todos os nós expandidos e testa o melhor nó dentre eles. O melhor nó é aquele que possui o menor custo, baseado em uma heurística. A heurística adotada é a Manhattan, que atribui um valor ao nó, sendo esse valor o somatório das distâncias cada peça para a posição correta em relação ao estado final, ou seja, o menor h(n). É assim que funciona, o melhor nó é testado, depois seus sucessores são gerados e armazenados em uma variável que contém os nós que serão avaliados na próxima iteração.

```
def solucaoGulosa(tree):
    global explorados
    global numN
    global h2
    global iteracoes

    if tree.chave not in explorados: explorados.append(tree.chave)

    print('profundidade: ', tree.depth, ', custo: ', tree.getHeu(h2))
    exibe(tree.chave)
    #time.sleep(0.8)
    os.system('clear')

    if tree.chave == tblTeste: return tree
    filhos = decisao(tree.chave, tree.pos0, tree)
    numN += len(filhos)

    # testa o nó de menor custo e expande #
    while True:
        iteracoes += 1
        a = custoMin(filhos)
        if a.chave == tblTeste: return a
        if a.chave not in explorados:

            print('profundidade: ', a.depth, ', custo: ', a.getHeu(h2))
            exibe(a.chave)
            #time.sleep(0.8)
            os.system('clear')

            filhos += decisao(a.chave, a.pos0, a)
            explorados.append(a.chave)
            numN += len(a.filhos)
```

Busca A*

Semelhante à busca gulosa, guarda informação de todos os nós expandidos e testa o melhor nó dentre eles. O melhor nó agora será aquele com menor valor g(n)+h(n), que é o custo para se chegar até esse nó mais o seu custo heurístico.

```
def solucaoAStar(tree):
    global explorados
    global h2
    global iteracoes
    global numN

    if tree.chave not in explorados: explorados.append(tree.chave)

    print('profundidade: ', tree.depth, ', custo: ', tree.getHeu(h2))
    exibe(tree.chave)
    time.sleep(0.8)
    os.system('clear')

    if tree.chave == tblTeste: return tree
    filhos = decisao(tree.chave, tree.pos0, tree)
    numN +=len(filhos)

    # testa o nó de menor custo e expande #
    while True:
        iteracoes += 1
        a = custoMin(filhos)
        if a.chave == tblTeste: return a
        if a.chave not in explorados:
            print('profundidade: ', a.depth, ', custo: ', a.getHeu(h2))
            exibe(a.chave)
            time.sleep(0.8)
            os.system('clear')
            filhos += decisao(a.chave, a.pos0, a)
            explorados.append(a.chave)
            numN += len(a.filhos)
```

Resultados e Comparações

Desempenho dos métodos de busca com o seguinte estado inicial:

2		**3**
1	**4**	**5**
7	**8**	**6**

	Busca em largura	Busca em profundidade	Busca gulosa	Busca A*
Nº de iterações	79	497	11	8
Nós gerados	144	505	28	25
Tempo	11.364s	80.086s	1.963s	1.783s
Profundidade da solução	5	7	5	5

Desempenho dos métodos de busca com o seguinte estado inicial:

1	**3**	**6**
4	**2**	**8**
	7	**5**

	Busca em largura	Busca em profundidade	Busca gulosa	Busca A*
Nº de iterações	406	4144	10	11
Nós gerados	664	4144	23	23
Tempo	54.018s	1220.002s	1.666s	1.650s
Profundidade da solução	8	8	8	8

Desempenho dos métodos de busca com o seguinte estado inicial:

4	**3**	**6**
8	**7**	**1**
	5	**2**

	Busca em largura	Busca em profundidade	Busca gulosa	Busca A*
Nº de iterações	38635	4680	1002	495
Nós gerados	58648	4680	1373	753
Tempo	5091.321s	750.076s	113.391s	62.164s
Profundidade da solução	18	_	56	18

Conclusão

Com uma análise dos resultados verifica-se que as buscas em largura, gulosa e A* são completas, que atingem a solução ótima temos a busca em largura e a busca A*. Em relação à utilização de memória, à medida em que a dificuldade do problema cresce, a busca em largura é a que apresenta o maior custo, a busca A* tem o menor custo. Quanto ao tempo, a busca A* é o mais eficiente, a busca em profundidade é o mais lento. Esses resultados também dependem da forma de como os métodos foram implementados, mas mesmo levando isso em consideração é possível concluir que a melhor abordagem de busca é a A*.

Referências

RUSSEL, Stuart; NORVIG, Peter. Inteligência Artificial. 3 ed. Rua Sete de Setembro, 111 – 16º andar 20050-006 – Centro – Rio de Janeiro – RJ – Brasil: Elsevier Editora Ltda. 2013.