Atividade Prática 2

Ana Flávia Dias, Camilla Damasceno, Jorge Luiz Mendes, Luiza Ávila, Stefany Gaspar

Algoritmos de busca de caminho são todos aqueles que solucionam o problema do caminho mínimo. No caso do nosso trabalho, os dois algoritmos apresentados, A* (lê-se A estrela) e busca de profundidade, são dois que resolvem o problema. Para a demonstração dos dois algoritmos, usamos o jogo de tabuleiro 8 puzzle (8 fichas).

I. Introdução

O jogo do 8-Puzzle é um jogo de tabuleiro de blocos deslizáveis. O objetivo do jogo é mover as peças a partir de um estado inicial até encontrar seu estado final, quando o Puzzle está ordenado de forma crescente, como na Figura 1. As regras do jogo são bastante simples, a peça vazia é a única que pode movimentar-se, dependendo da situação pode haver de dois a quatro movimentos possíveis (cima, baixo, direita e esquerda). Estes movimentos geram novos estados até encontrar o estado final. O Puzzle possui um espaço de estados no valor de 9!. A solução ótima para este problema pertence a classe NP-Completo.

Existem diversas maneiras de solucionar o problema de 8 puzzle, tais como algoritmos de combinação, busca em largura, profundidade, técnicas de busca direcional, entre diversas outras. Nesse trabalho introduziremos 2 formas de solução: com o algoritmo A* (A estrela) e busca de profundidade.

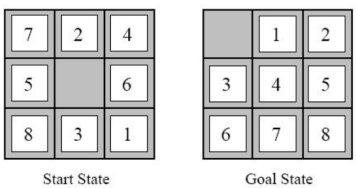

Figura 1: Estados do puzzle

II. Introdução dos algoritmos

A. A*

Um dos algoritmos best-first mais conhecidos é chamado de A* (lê-se “Aestrela”) esse algoritmo é a combinação de aproximações heurísticas, como do algoritmo de busca em largura e do algoritmo de Dijkstra (1959).

O algoritmo de Dijkstra foi concebido pelo holandês Edsger Dijkstra em 1959, basicamente este algoritmo visava solucionar os problemas de caminho mais curto. A principal diferença entre ele e o algoritmo A* é a ausência de uma função heurística que facilite e diminua o número de nós expandidos, pois a cada passo o algoritmo de Dijkstra verificaria os nós adjacentes para efetuar a avaliação, sem se importar com uma ordem ou priorização dos ramos, o que acontece no algoritmo A* devido a utilização da função heurística determinada pelo problema.

O algoritmo A* Foi descrito primeiramente por Hart, Nilsson e Raphael (1968) mas hoje em dia já existem muitas outras variantes do algoritmo proposto originalmente. Ele avalia os nós através da combinação de g(n) que é o custo para alcançar cada nó com a função h(n) que é o menor custo partindo da origem para se chegar ao destino, matematicamente dado na equação: $f(n) = g(n) + h(n)$.

Desse modo f(n) é, portanto, o custo estimado da solução de custo mais baixo passando por n. Esta técnica requer que a estimação do custo restante no próximo nó não seja nunca maior que o custo restante do nó anterior. Diferente das duas técnicas anteriores, sob esta hipótese, sempre é possível encontrar a solução ótima com a busca A*.

Na Figura 2 é ilustrado um fluxograma do algoritmo A*, em que são ilustrados as tomadas de decisões e os passos para o desenvolvimento do algoritmo. O algoritmo se inicia com a declaração das variáveis e dos vetores, em que S é a posição de partida do agente. ABERTOS é o nome da lista de nós ou posições que já foram abertas. FECHADOS é o nome da lista de nós ou posições que já foram fechadas, ou seja, já foram avaliadas e ordenadas. N é a posição ou o nó atual do agente, é com ele que se determina se o objetivo foi alcançado ou se o algoritmo precisa expandir mais nós e avalia-los.

Os algoritmos de busca podem ser avaliados sob quatro aspectos; Completeza, se ele encontra a solução se ela existir; Otimização, se ele encontra a solução de menor custo; Tempo, o tempo que ele leva para encontrar a solução e Espaço, que é a quantidade de memória consumida para executar a busca.

A busca A* é completa, ótima e eficiente. É completa a não ser que exista uma quantidade infinita de nós. Ótima, pois, nenhum outro algoritmo tem garantia de expandir um número de nós menor que o A*. Isso porque qualquer algoritmo que não expande todos os nós com f(n) $<$ C* corre o risco de omitir uma solução ótima. Sua complexidade ainda é exponencial e o seu uso de memória é intenso.

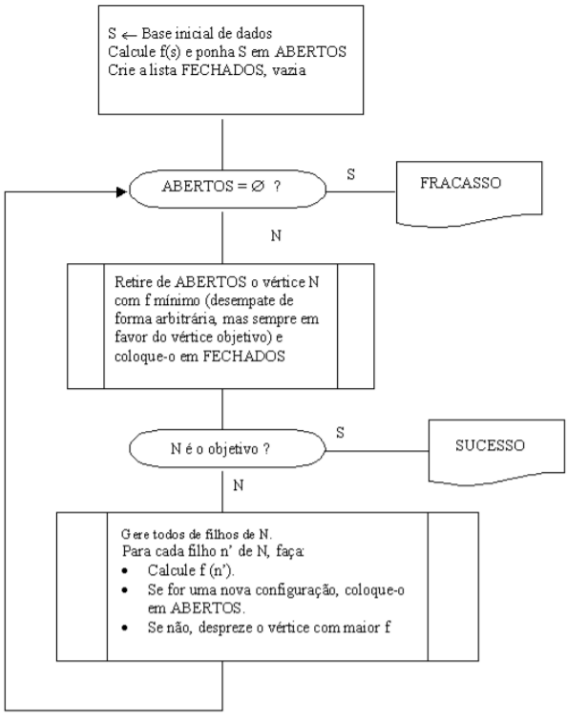

Figura 2: Fluxograma

B. Busca em profundidade

O algoritmo de busca em profundidade é percorrer todos os caminhos de um grafo de forma sistemática. Grosso modo, o algoritmo funciona assim. Começando por um vértice qualquer, e indo "o mais fundo possível". Sempre que encontramos um vértice já visitado, retornamos da busca.

Ela começa a partir de um vértice arbitrário r, que é chamado de raiz da busca (esse vértice é também algumas vezes chamado de vértice fonte). O vértice r é marcado como visitado. A seguir, seleciona-se um vértice qualquer r1, dentre os vizinhos de r que ainda não tenham sido marcados visitados, e uma nova DFS é iniciada recursivamente a partir dele. A recursão termina ao encontrar um vértice v cujos vizinhos estejam todos marcados como visitados. Se após a DFS em r1 terminar todos os outros vértices adjacentes a r já tiverem sido marcados como visi-

tados, a DFS em r também termina. Caso contrário, seleciona-se um outro vértice arbitrário r2 adjacente a r e ainda não marcado como visitado. Uma nova DFS é iniciada em r2. Esse procedimento é repetido até que todos os vértices do grafo tenham sido marcados como visitados.

III. Implementação no jogo

Segue os métodos principais implementados no jogo 8-puzzle de cada algoritmo.

A. A*

```python
import random

objetivo = [[1, 2, 3],
            [4, 5, 6],
            [7, 8, 0]]

def index(item, seq):
    if item in seq:
        return seq.index(item)
    else:
        return -1
```

Figura 3: Função pra validar se o item existe no array e se exisitr retorna o índice se não retorna -1

```python
def __init__(self):
    self.valorHeuristica = 0
    self.profundidade = 0
    self.vizinhos = None
    self.matrizAdjacente = []
    for i in range(3):
        self.matrizAdjacente.append(objetivo[i][:])

def __str__(self):
    res = ''
    for linha in range(3):
        res += ' '.join(map(str, self.matrizAdjacente[linha]))
        res += '\r\n'
    return res

def _clonarMatriz(self):
    p = EightPuzzle()
    for i in range(3):
        p.matrizAdjacente[i] = self.matrizAdjacente[i][:]
    return p
```

Figura 4: Método padrão inicial o valorHeuristica da heurísitica. Profundidade dos movimentos é o mesmo que quantidade. Vizinhos nós vizinhos. MatrizAdjacente

```python
def obterMovimentosPossiveis(self):
    linha, coluna = self.procurarNo(0)
    livre = []

    if linha > 0:
        livre.append((linha - 1, coluna))
    if coluna > 0:
        livre.append((linha, coluna - 1))
    if linha < 2:
        livre.append((linha + 1, coluna))
    if coluna < 2:
        livre.append((linha, coluna + 1))

    return livre

def gerarMovimentos(self):
    free = self.obterMovimentosPossiveis()
    zero = self.procurarNo(0)

    def movimentar(a, b):
        p = self._clonarMatriz()
        p.swap(a, b)
        p.profundidade = self.profundidade + 1
        p.vizinhos = self
        return p

    return map(lambda pair: movimentar(zero, pair), free)

def obterCaminhoDaSolucao(self, path):
    if self.vizinhos == None:
        return path
    else:
        path.append(self)
        return self.vizinhos.obterCaminhoDaSolucao(path)
```

Figura 5: Método para obter os movimentos possíveis com o espaço em branco. Retorna uma lista de tuplas (linha, coluna).

```python
def solve(self, h):
    def is_solved(puzzle):
        return puzzle.matrizAdjacente == objetivo

    openl = [self]
    closedl = []
    movimentos = 0
    while len(openl) > 0:
        x = openl.pop(0)
        movimentos += 1
        if (is_solved(x)):
            if len(closedl) > 0:
                return x.obterCaminhoDaSolucao([]), movimentos
            else:
                return [x]
        succ = x.gerarMovimentos()
        idx_open = idx_closed = -1
        for move in succ:
            idx_open = index(move, openl)
            idx_closed = index(move, closedl)
            hval = h(move)
            fval = hval + move.profundidade
            if idx_closed == -1 and idx_open == -1:
                move.valorHeuristica = hval
                openl.append(move)
            elif idx_open > -1:
                copia = openl[idx_open]
                if fval < copia.valorHeuristica + copia.profundidade:
                    copia.valorHeuristica = hval
                    copia.vizinhos = move.vizinhos
                    copia.profundidade = move.profundidade
            elif idx_closed > -1:
                copia = closedl[idx_closed]
                if fval < copia.valorHeuristica + copia.profundidade:
                    move.valorHeuristica = hval
                    closedl.remove(copia)
                    openl.append(move)
        closedl.append(x)
        openl = sorted(openl, key=lambda p: p.valorHeuristica + p.profundidade)
    return [], 0
```

Figura 6: Executa o A* e retora o numero de movimentações e o caminho

```python
def shuffle(self, step_count):
    for i in range(step_count):
        linha, coluna = self.procurarNo(0)
        free = self.obterMovimentosPossiveis()
        target = random.choice(free)
        self.swap((linha, coluna), target)
        linha, coluna = target

def procurarNo(self, value):
    if value < 0 or value > 8:
        raise Exception("Valo não encontrado")

    for linha in range(3):
        for coluna in range(3):
            if self.matrizAdjacente[linha][coluna] == value:
                return linha, coluna
```

Figura 7: Retorna as coordenadas de um valor específico

```python
def obterValor(self, linha, coluna):
    return self.matrizAdjacente[linha][coluna]

def setarValor(self, linha, coluna, value):
    self.matrizAdjacente[linha][coluna] = value

def swap(self, pos_a, pos_b):
    temp = self.obterValor(*pos_a)
    self.setarValor(pos_a[0], pos_a[1], self.obterValor(*pos_b))
    self.setarValor(pos_b[0], pos_b[1], temp)
```

Figura 8: Obter valores de uma linha e coluna específicos. Método swap para troca de valores de posições.

```python
def heur(puzzle, item_total_calc, total_calc):
    t = 0
    for linha in range(3):
        for coluna in range(3):
            val = puzzle.obterValor(linha, coluna) - 1
            target_coluna = val % 3
            target_linha = val / 3

            # account for 0 as blank
            if target_linha < 0:
                target_linha = 2

            t += item_total_calc(linha, target_linha, coluna, target_coluna)

    return total_calc(t)
```

Figura 9: Parâmetros puzzle: Item total de cálculo recebe linha atual, linha destino, coluna atual e coluna destino. Retorno de um inteiro.

```python
def h_manhattan(puzzle):
    return heur(puzzle,
                lambda r, tr, c, tc: abs(tr - r) + abs(tc - c),
                lambda t: t)


def h_default(puzzle):
    return 0


def main():
    p = EightPuzzle()
    p.shuffle(20)
    print(p)

    path, count = p.solve(h_manhattan)
    path.reverse()
    for i in path:
        print(i)

    print("Resolvido usando a distância de Manhattan", count, "movimentos")
```

Figura 10: Método utilizado para inicializar o algoritmo.

O algoritmo A* utiliza uma fila de prioridades e uma função que tenta prever a quantidade de ações que devem ser realizadas para alcançar o objetivo a partir do estado em questão. A estratégia é explorar primeiro os nós que representam estados com menor custo esperado, pois eles possivelmente estariam mais próximos do objetivo.

O custo esperado para um nó de busca n que representa um estado s qualquer, é a soma entre a quantidade de passos já realizados para chegar em n e a quantidade de passos que ainda faltam para chegar no objetivo. Como não se sabe exatamente quantos passos faltam, utiliza-se o resultado da função como uma estimativa. A fila de prioridades é utilizada para reservar os nós (e caminhos) ainda não visitados pelo algoritmo e para facilitar a escolha do nó mais próximo do objetivo.

B. Busca em Profundidade

```python
if __name__ == '__main__':
    noInicial = [1, 2, 5, 3, 4, 8, 6, 7, 0]
    noFinal = [0, 1, 2, 3, 4, 5, 6, 7, 8]

    encontrado = False
    numeroNo = 0
    listaNosVisitados = pilha = []
    pilha.append(noInicial)
    listaNosVisitados.append(noInicial)
    mostrarNo(noInicial)
    t0 = time.time()
```

Figura 11: Inicialização das variáveis

```
while not encontrado and not len(pilha) == 0:
    noAtual = pilha.pop(0)
    quadradoVazio = noAtual.index(0)
    indiceInsercao = 0
    if quadradoVazio != 0 and quadradoVazio != 1 and quadradoVazio != 2:
        noSuperior = copy.deepcopy(noAtual)
        noSuperior[quadradoVazio] = noSuperior[quadradoVazio - 3]
        noSuperior[quadradoVazio - 3] = 0
        noSuperior.append('cima')
        encontrado = validarFinal(noSuperior)
    if quadradoVazio != 0 and quadradoVazio != 3 and quadradoVazio != 6 and not encontrado:
        noEsquerda = copy.deepcopy(noAtual)
        noEsquerda[quadradoVazio] = noEsquerda[quadradoVazio - 1]
        noEsquerda[quadradoVazio - 1] = 0
        noEsquerda.append('esquerda')
        encontrado = validarFinal(noEsquerda)
    if quadradoVazio != 6 and quadradoVazio != 7 and quadradoVazio != 8 and not encontrado:
        noInferior = copy.deepcopy(noAtual)
        noInferior[quadradoVazio] = noInferior[quadradoVazio + 3]
        noInferior[quadradoVazio + 3] = 0
        noInferior.append('baixo')
        encontrado = validarFinal(noInferior)
    if quadradoVazio != 2 and quadradoVazio != 5 and quadradoVazio != 8 and not encontrado:
        noDireita = copy.deepcopy(noAtual)
        noDireita[quadradoVazio] = noDireita[quadradoVazio + 1]
        noDireita[quadradoVazio + 1] = 0
        noDireita.append('direita')
        encontrado = validarFinal(noDireita)
```

Figura 12: Rodando o algoritmo

A cada interação se remove um nó da pilha e compara com os nós adjacentes. Então é feito o movimento e adicionado no vetor para indicar o movimento. E se chama o método para verificar se encontrou o resultado final desejado. Irá rodar enquanto existirem nós na pilha e enquanto não for encontrado uma solução viável

```
def mostrarNo(no):
    print(no[0], no[1], no[2])
    print(no[3], no[4], no[5])
    print(no[6], no[7], no[8])
    global numeroNo
    print('No:', numeroNo)
    print('Profundidade:', len(no[9:]))
    print('Moveminento:', no[9:])
    numeroNo += 1
```

Figura 13: Método usado para mostrar os nós do puzzle

```
def validarFinal(no):
    if no[:9] == noFinal:
        mostrarNo(no)
        return True
    global indiceInsercao
    if no[:9] not in listaNosVisitados:
        mostrarNo(no)
        pilha.insert(indiceInsercao, no)
        indiceInsercao += 1
        listaNosVisitados.append(no[:9])
    return False
```

Figura 14: Método para validar o tabuleiro

IV. Resultados da Implementação

Segue os métodos principais implementados no jogo 8-puzzle de cada algoritmo.

A. A*

```
1 5 2
4 0 3
7 8 6



1 0 2
4 5 3
7 8 6



1 2 0
4 5 3
7 8 6



1 2 3
4 5 0
7 8 6



1 2 3
4 5 6
7 8 0



Resolvido usando a distância de Manhattan 5 movimentos
```

Figura 15: Retorno da execução do algoritmo A*

O algoritmo A* gera uma matriz randômica a cada execução. A cada movimentação feita, é impresso na tela a nova matriz adquirida, até por fim, chegar na matriz objetivo, finalizando a execução do programa com uma linha impressa informando o número de movimentações realizadas.

```
1 2 5
3 4 8
6 7 0
No: 0
Profundidade: 0
Movimento: []
1 2 5
3 4 0
6 7 8
No: 1
Profundidade: 1
Movimento: ['cima']
1 2 5
3 4 8
6 0 7
No: 2
Profundidade: 1
Movimento: ['esquerda']
1 2 0
3 4 5
6 7 8
No: 3
Profundidade: 2
Movimento: ['cima', 'cima']
1 2 5
3 0 4
6 7 8
No: 4
Profundidade: 2
Movimento: ['cima', 'esquerda']
1 0 2
3 4 5
6 7 8
No: 5
Profundidade: 3
Movimento: ['cima', 'cima', 'esquerda']
0 1 2
3 4 5
6 7 8
No: 6
Profundidade: 4
Movimento: ['cima', 'cima', 'esquerda', 'esquerda']
-------------------------------------------
Tempo de Execução: 0.0019996166229248047
-------------------------------------------
```

Figura 16: Retorno da execução do algoritmo Busca em Profundidade

O algoritmo de busca em profundidade se inicia informando a matriz de início e a desejada. A medida que é feita a execução dos métodos é impresso na tela o nó que o algoritmo está no momento atual, junto a sua profundidade e movimentos necessários para percorrer toda a matriz e chegar no resultado desejado. No fim da execução também é informado através da impressão na tela o tempo de execução do algoritmo.

Referências

BUSCA EM PROFUNDIDADE Kernel Description. [S.l.: s.n.]. http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm. Accessed: 2020-11-29.

GUIMARÃES, Florêncio Junior. Problema 8-Puzzle: Análise da solução usando Backtracking e Algoritmos Genéticos.

ZANCHIN. ANÁLISE DO ALGORITMO A* (A ESTRELA) NO PLANEJAMENTO DE ROTAS DE VEÍCULOS AUTÔNOMOS, 2018.