Inteligência Artificial 2018/1
Trabalho Prático 1

Fábio Dayrell F. M. Rosa[1]

[1]Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Belo Horizonte – Minas Gerais – Brasil

dayrell@dcc.ufmg.br

1. Introdução

Nesta documentação está descrita a solução para o trabalho prático 1 da disciplina Inteligência Artificial da UFMG. Inicialmente, é discutido o problema. Em seguida, serão apresentados testes e análises de cada uma das implementações desenvolvidas. Por fim, são apresentadas informações técnicas sobre a estrutura dos arquivos e sobre a execução do código.

2. O problema

O problema apresentado nesse trabalho é encontrar um caminho entre dois pontos de um mapa dado na entrada. Para encontrar esse caminho, é necessário implementar os algoritmos de busca de custo uniforme (USC), busca gulosa de melhor escolha (BFS), busca de aprofundamento iterativa (IDS) e A*. No caso do A*, é necessário implementar as heurísticas da distância de Manhattan e distância Octile. Para a busca gulosa, apenas a heurística da distância de Manhattan é necessária.

A implementação do código foi feita utilizando a linguagem Python. Apesar de apresentar desempenho inferior a linguagens como C/C++, optei por usar Python para me focar no uso/aperfeiçoamento de apenas uma linguagem nesse semestre (estou utilizando Python em outras três disciplinas).

2.1. Arquivo de entrada

Um arquivo de texto é passado como parâmetro na execução. Nesse arquivo, encontra-se o mapa que deve ser utilizado no problema onde existem apenas os caracteres de ponto final e de arroba. O primeiro simbolo representa um caminho livre e o segundo uma parede.

Para representar esse mapa, é possível utilizar um grafo. A construção desse grafo foi feita utilizando um dicionário de dicionários. Nessa representação, o mapa da entrada abaixo:

```
...
.@.
```

É representado da seguinte forma:

```
{   0: {3: 1.0, 1: 1.0},
    1: {2: 1.0, 0: 1.0},
```

```
2: {5: 1.0, 1: 1.0},
3: {0: 1.0},
5: {2: 1.0}     }
```

No dicionário anterior, também é possível ver algumas restrições de deslocamento no mapa de entrada. Não é possível ir da posição 1 para a 4, já que a posição 2 é uma parede. Outra restrição menos intuitiva é a de não poder ir para o ponto 1 para o ponto 5.

3. Implementações

Todos os algoritmos solicitados na documentação foram implementados. Nesta seção, irei descrever alguns detalhes de implementação desses algoritmos. O passo a passo dos algoritmos em si não será apresentada, já que esse não é o objetivo dessa documentação (todos eles estão descritos detalhadamento nos slides do curso).

Nos algoritmos BFS, UCS e A* foi criada uma lista chamada "pai". Essa lista possui tamanho $height \times width$ e é inicializada com -1 em todas as posições. O objetivo dessa lista é fornecer o caminho do vértice final até o vértice inicial. Na execução de cada algoritmo, essa lista é preenchida de forma que cada posição dela seja preenchida com a posição do vértice que leva até ela.

Nas subseções seguintes, serão descritos os detalhes relevantes de cada implementação.

3.1. Busca de Aprofundamento Iterativo (IDS)

Nesse algoritmo, é usada uma variação do algoritmo de busca em profundidade.

Como no algoritmo de busca em profundidade é possível que a busca entre em um ramo infinito do grafo de entrada, fazendo com que a execução do programa não termine, é preciso definir uma profundidade máxima na qual o programa deve ir. Como não sabemos a profundidade da solução, é possível iterar esse algortmo. Ou seja, começamos com uma profundidade máxima de 0, depois de 1, 2 e assim por diante até que a solução desejada seja encontrada.

3.2. Busca de Custo Uniforme (UCS)

Nesse algoritmo, é utilizada uma variação da busca em largura levando em conta o peso dos nós.

No algoritmo da busca em largura, os pesos das arestas são ignorados. Desse modo, a busca em um nó é feita seguindo uma lógica mais simples, Uma fila é utilizada com a política FIFO. Ou seja, o primeiro nó a ser adicionado nessa fila é o primeiro a ser investigado.

No caso da busca de custo uniforme, os nós são inseridos em uma fila de prioridade, onde nós com peso menor têm maior prioridade. No entanto, nessa implementação, foi usada uma alternativa a essa fila de prioridade. A lista "distancias"e a função "extract min". O uso e funcionamento delas vão ser detalhados mais abaixo.

A função "usc"é usada para realizar esse busca. A lista fechada é representada pela lista "explorado"e a lista aberta é representada pela lista "fronteira". Além disso, foi

criada a lista distância, com tamanho $height \times width$ e todas as posições inicializadas com -1. Cada posição i nessa lista representa a distância do nó inicial até o nó i.

A política de expansão dessa lista é feita no trecho de código a seguir.

```
1. for filho in grafo[node]:
2.     if ((filho not in explorado) and (filho not in fronteira)):
3.         fronteira.append(filho)
4.         [filho] = node
5.         distancias[filho] = distancias[node] + grafo[node][filho]
6.     elif ((filho in fronteira) and ( distancias[filho] > distancias[
                                    node] + grafo[node][filho] )  ):
7.         pai[filho] = node
8.         distancias[filho] = distancias[node] + grafo[node][filho]
```

Código 1 - Expansão da fronteira do UCS

Na primeira linha, é definida a iteração que consiste em percorrer todos os filhos do nó sendo expandido.

Se esse filho não está na lista de nós explorados e ele também não está na lista da fronteira, o filho é adicionado à fronteira. Também, é definido na lista "pai"que o pai desse filho sendo investigado é o nó que está sendo expandido na linha 1 e a distância é simplesmente o peso da aresta que liga esses dois vértices.

Se a condição da linha 2 não for satisfeita, e o filho estiver na fronteira e a distância do nó especificado na linha 1 (grafo[node]) até o filho que está sendo investigado for menos que a distância até esse filho especificado na lista "distâncias", significa que existe um caminho mais curto até esse filho. Então, o pai desse filho passa a ser o node atual e a distância do vértice inicial até esse filho é a distância até o seu novo pai mais a distância do pai até o filho.

Além disso, para que o UCS seja efetivamente implementado, foi criada a função auxiliar "extract min". Essa função receberá as listas "fronteira"e "distancia"e irá selecionar aquele elemento (vértice) da lista "fronteira"com menor valor da lista "distancia". Isso faz com que sempre o elemento da fronteira com o menor custo (para chegar até ele) seja explorado.

3.3. Busca Gulosa de Melhor Escolha (BFS)

Esse algoritmo utiliza uma abordagem gulosa. Ele sempre toma a decisão que é melhor naquele momento. A função responsável por realizar esse algoritmo chama-se bfs.

Para quantificar quão melhor uma escolha é, ele utiliza a heurística da distância de Manhattan, que é calculada como mostrado a seguir:

```
def distancia_manhattan(g, node, goal):
    node_x = math.floor(node / g.width)
    node_y = node % g.width

    goal_x = math.floor(goal / g.width)
    goal_y = goal % g.width

    return (math.fabs(node_x - goal_x) + math.fabs(goal_y - node_y))
```

Código 2 - Cálculo da Distância de Manhattan

O funcionamento da exploração de nós e expansão da fronteira é bastante similar ao da busca de custo uniforme. A novidade aqui é a lista "nova possivel distancia"que é inicializada do mesmo modo que a lista "distancias"($[-1] \times height \times width$). Se na busca de custo uniforme leva-se em conta a distância do primeiro vértice até o filho sendo expandido, mais a distância do pai até esse filho (linha 6 do código 1), na busca gulosa de melhor escolha leva-se em consideração a distância do vértice inicial até o pai do vértice (filho) sendo investigado mais a distância de Manhattan entre esse filho e o vértice final.

Além disso, é importante considerar que movimentos diagonais não são permitidos quando usa-se a heurística da distância de Manhattan porque, com diagonais sendo permitidas, essa heurística poderia superestimar a distância entre o vértice inicial e o final. Por definição, uma heurística que superestima distância não é admissível.

3.4. A*

A função responsável por aplicar o algoritmo A* chama-se "aestrela". Essa função executa tal qual a função da subseção anterior. No entanto, enquanto a função BFS define a "nova possivel distancia"de cada nós expandido como apenas a distância entre o vértice inicial e o pai do vértice investigado mais o resultado da heurística, o A* leva em consideração, também, a distância entre o nó investigado e o seu pai.

Além disso, a execução dessa função é influenciada pela heurística escolhida. Dessa forma, cada heurística pode retornar um valor diferente da outra. As duas heurísticas implementadas no trabalho são descritar abaixo.

3.4.1. Distância de Manhattan

O cálculo da distância de Manhattan já foi descrito mais acima no código 2. Além disso, a limitação de não poder haver deslocamentos verticais continua valendo para o A*.

Como pode ser visto no código 2, a distância de Manhattan é a soma das diferenças absolutas de suas coordenadas. No contexto desse problema, é a soma das diferenças absolutas do vértice investigado e o vértice final.

3.4.2. Distância Octile

O cálculo da distância Octile é o mostrado no código 3.

```
def distancia_octille(g, node, goal):
    node_x = math.floor(node / g.width)
    node_y = node % g.width

    goal_x = math.floor(goal / g.width)
    goal_y = goal % g.width

    dx = math.fabs(node_x - goal_x)
    dy = math.fabs(goal_y - node_y)

    return max(dx, dy) + 0.5 * min(dx, dy)
```

Código 3 - Cálculo da Distância Octile

A vantagem da distância Octile em relação à de Manhattan é que ela permite deslocamentos na diagonal. Como será descrito com mais detalhes na próxima seção, isso faz com que o uso dessa heurística retorne sempre o resultado ótimo.

4. Testes e Análises

Durante o desenvolvimento do trabalho, foi utilizada a biblioteca PIL do Python para visualizar em formato de imagem o caminho percorrido e as áreas visitadas em cada algoritmo. Isso facilitou o entendimento dos algoritmos desenvolvidos.

Na imagem abaixo, é possível ver em vermelho a área visitada durante a busca e em laranja o trajeto escolhido pelo algoritmo.

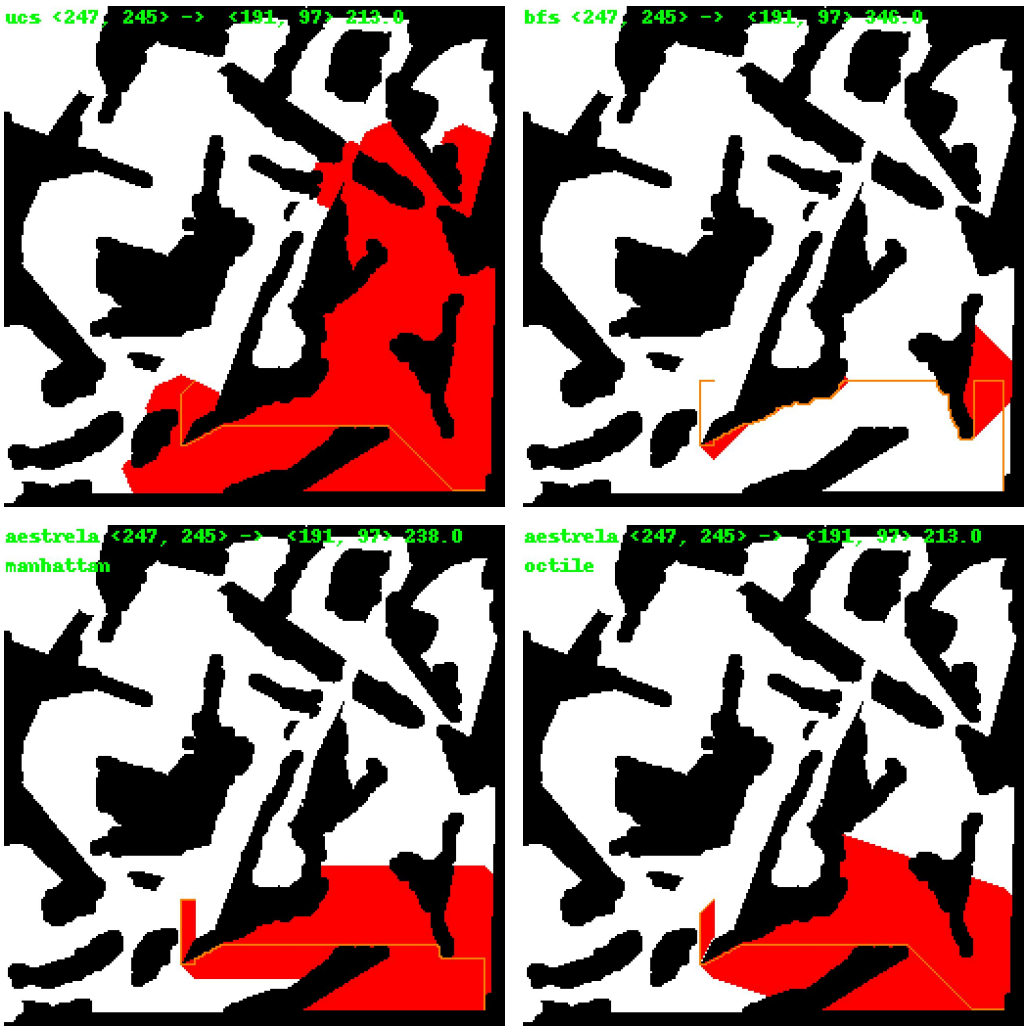

Figura 1. Representação visual do trajeto do ponto (247, 245) - (191, 97)

As imagens geradas para os testes disponibilizados estão disponíveis na pasta “testes”enviada junta aos arquivos do trabalho.

Como gerar as imagens não é o objetivo do trabalho e, também, esse processo deixa o programa ainda mais custoso (em tempo, principalmente), essa “funcionali-

dade"está desabilitada. No entanto, essa implementação ainda está no código como comentários. Para gerar imagens, basta descomentar trechos do código associados à variável img.

4.1. Vértices percorridos e explorados

Na tabela 1, é possível ver a quantidade de vértices do grafo percorridos em deslocamentos no mapa 1 e 2 disponibilizados.

	BFS	A* - Manhattan	A* - Octile	UCS
	map1			
(247, 245) - (191, 97)	9.612	54.222	57.163	112.335
(190, 249) - (175, 249)	75	75	75	2.454
	map2			
(52, 97) - (247, 25)	3.932	30.741	29.571	233.074
(201, 207) - (84, 166)	1.260	22.906	10.175	97.301

Tabela 1. Número de vértices percorridos na execução do programa

	BFS	A* - Manhattan	A* - Octile	UCS
	map1			
(247, 245) - (191, 97)	1.301	7.003	7.383	14.652
(190, 249) - (175, 249)	15	15	15	318
	map2			
(52, 97) - (247, 25)	585	3.889	4.050	30.630
(201, 207) - (84, 166)	158	2.950	1.320	12.723

Tabela 2. Tamanho da lista de vértices explorados

É fácil ver que o BFS apresenta o melhor resultado quando levamos em consideração o número de vértices visitados. No entanto, sabe-se que o BFS não é um algoritmo ótimo e, nos testes feitos, foi possível constatar que é extremamente improvável que ele consiga o resultado ótimo, apesar de quase sempre ser o que tem execução mais rápida.

As heurísticas de Octile e da distância de Manhattan apresentam resultados um pouco parecidos. Já o UCS sempre explora vértices "desnecessários". Ou seja, vértices que não estarão na solução. O IDS, por outro lado, nem mesmo foi usado nesses testes, já que apresentou desempenho muito ruim.

O número de vértices percorridos está diretamente ligado ao tempo de execução.

Dos algoritmos implementados, o UCS, A* Octile e IDS são os que apresentam resultado ótimo. No entanto, para caminhos complexos e distantes, o A* Octile é a melhor opção.

Na tabela 2 estão os tamanhos da lista de vértices explorados para os testes apresentados.

Enquanto o número de vértices percorridos está ligado ao tempo de execução, a tabela de vértices explorados nos diz quanto da memória foi consumido durante a execução do programa.

Em relação à memória consumida, mais uma vez é possível ver que o BFS apresentou melhor desempenho em relação aos demais algoritmos

5. Detalhes técnicos

Nessa seção será discutido rapidamente alguns detalhes técnicos do trabalho, como estrutura dos arquivos e execução do código.

5.1. Estrutura dos arquivos

Na pasta raiz, estão os arquivos $main.py$, que chama os métodos para execução das buscas, $graph.py$, com a classe da implementação do grafo e $utils.py$, com funções específicas que podem ser usadas em outros arquivos.

No diretório $maps$ estão alguns mapas para testes. No diretório $testes$ estão imagens geradas para os testes e análises dos algoritmos. No diretório $shellscripts$ estão os scripts para execução de cada algoritmo de busca. E, por fim, no diretório $searchalgorithms$ estão cada um dos algoritmos de busca.

5.2. Execução do código

Como solicitado, foi criado um script shell para a execução de cada algoritmo. Após dar as permissões necessárias para o arquivo $.sh$, basta executar um código como o a seguir no terminal:

```
./ucs.sh ../maps/map2.map 52 97 247 25
```

Para o algoritmo A*, é necessário informar um parâmetro adicional com a heurística que será utilizada. Como mostra o exemplo abaixo.

```
./aestrela.sh ../maps/map2.map 52 97 247 25 manhattan
```