Projeto 1

Tiago Loureiro Chaves (187690)
MC906A - Introdução à Inteligência Artificial - 1s2019

14 de Abril, 2019

Resumo

Neste projeto avaliou-se o problema de busca de caminho em um labirinto (*pathfinding*) representado por uma matriz 60×60, descrito com a classe `Problem` da biblioteca de funções do livro AIMA (*Artificial Intelligence: A Modern Approach*) em Python [1], comparando as soluções obtidas por diferentes métodos.

O trabalho consistiu na modelagem do problema, das ações que um robô procurando o caminho poderia executar, na escolha dos algoritmos de busca e na análise dos resultados.

Os modelos utilizados encontram-se na seção 2, as soluções obtidas estão expostas na seção 4 e uma discussão dos resultados é apresentada na seção 5.

1 Objetivos

Este projeto visou a implementação e análise de diferentes métodos de busca na procura de uma solução para o problema de achar um caminho em um mapa, dada uma posição inicial (i_0, j_0) e uma posição objetivo (I, J).

Compararam-se métodos de busca sem informação e de busca informada (heurística), com os seguintes algoritmos (implementados em `search.py` [2]):

- *Breadth First Search* (`BFS`)
- *Depth First Search* (`DFS`)
- *Best First Search* (`BestFS`)
- *A*, A star* (`A*`)

2 Problema

Um robô é inicialmente colocado na posição $(i_0, j_0) = (50, 10)$ do mapa apresentado na Figura 1 e deve chegar à posição $(I, J) = (10, 50)$. As paredes são instransponíveis, e as áreas em roxo representam posições (i', j') válidas, para as quais o robô pode locomover-se.

Por padrão o robô pode se mover apenas para as posições adjacentes em pontos cardeais (N, S, E e W), porém os pontos colaterais (as diagonais) também podem ser considerados para que todas as 8 posições vizinhas sejam alcançáveis em um único passo (veja o parâmetro `diagonal_moves` do construtor da classe `PathfindingRobotProblem`, que modela o problema).

Cada estado retrata a posição atual (i, j) do robô no mapa do labirinto.

2.1 Ambiente e representação do estado

Optou-se por representar o mapa como um *grid*, usando uma matriz, ao invés de um grafo usando um dicionário, pela simplicidade tanto de criação de instâncias para teste, como da visualização do mapa no próprio código (veja o final do arquivo `pathfinding_robot_maps.py`).

O valor de cada posição (i, j) da matriz define uma das seguintes possibilidades para o local no mapa: parede (WALL), posição válida (EMPTY), posição inicial (START) e posição objetivo (GOAL) (constante definidas em `pathfinding_robot_maps.py`).

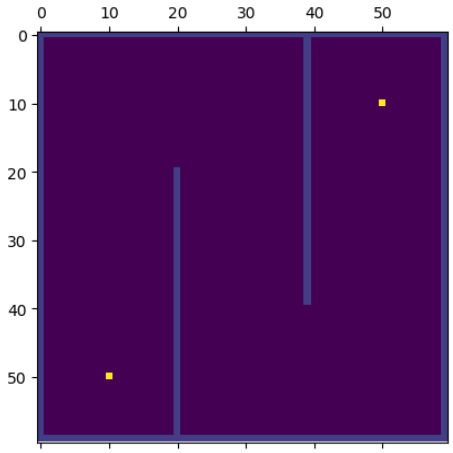

Figura 1: Mapa 60×60 do ambiente, com as posições inicial e final (em amarelo)

De tal modo, o problema pode ser visto de forma mais genérica como a navegação em uma grade (ou malha) bidimensional, com a restrição de intransitabilidade de algumas células.

Assim, os estados do robô são representados por suas referentes posições (i, j) no labirinto, levando a um mapeamento direto para células (i, j) da matriz.

Logo, o estado inicial é dado pela posição (i_0, j_0) da matriz cujo valor é START, e o teste de objetivo é a simples comparação pela igualdade da posição do estado atual (i, j) do robô com a célula (I, J) da matriz que possui valor GOAL (veja e a função `goal_test(self, state)` em `pathfinding_robot.py`).

2.2 Percepção e ações do agente

O robô é capaz de perceber onde se encontra a cada instante de tempo, e cada ação executada sempre o leva para uma posição adjacente, sendo o ambiente determinístico. O resultado de uma ação (di, dj) no estado (i, j) é um novo estado $(i', j') = (i + di, j + dj)$.

Se o parâmetro `diagonal_moves` do problema (`PathfindingRobotProblem`) é falso, as seguintes ações (di, dj) são possíveis (exceto quando levam para uma parede ou para fora do mapa):

- $(-1, 0)$ — cima (N)
- $(0, -1)$ — esquerda (W)
- $(0, 1)$ — direita (E)
- $(1, 0)$ — baixo (S)

Quando verdadeiro, todas as posições vizinhas são representadas como ações possíveis, ou seja:

$$(di, dj) \in \{-1, 0, 1\} \times \{-1, 0, 1\} \setminus \{(0, 0)\}, \quad \text{pois } (0, 0) \text{ representaria o próprio estado.}$$

O custo do caminho, $g(n)$, é sempre 1 para as posições resultantes de ações válidas, e $+\infty$ para as inválidas.

2.3 Heurísticas utilizadas

Para garantir a otimalidade do algoritmo `A*` usou-se a distância Manhattan [3] para que $h(n)$ fosse uma heurística admissível (otimista), ou seja, o custo estimado pela heurística é, no máximo, igual ao custo real do caminho.

Já para o `BestFS`, sabendo que a escolha gulosa da próxima ação não garante a descoberta de uma solução ótima, usou-se a distância Manhattan para comparação com o resultado do `A*`, mas também as distâncias Euclidiana e Chebyshev [4].

O custo de movimentos cima-baixo e esquerda-direita é 1 para todas as heurísticas, elas diferem entretanto no valor de deslocamentos diagonais, conforme mostra a figura a seguir:

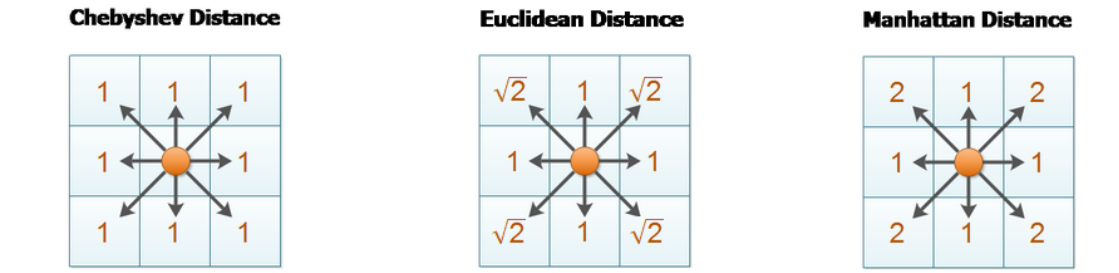

Figura 2: Comparação do custo estimado entre as heurísticas utilizadas [5]

Nota-se que quando movimentos diagonais sejam possíveis com uma única ação (`diagonal_moves = True`) a distância Manhattan deixa de ser uma heurística otimista, portanto a função $h(n)$ implementada na própria classe do problema (`h(self, state)`) muda a heurística utilizada para a distância Chebyshev neste caso (a qual é então admissível) [6].

As funções estão declaradas em `pathfinding_robot_heuristics.py`.

3 Algoritmos de busca

Os algoritmos utilizados foram os implementados para busca em grafo, pois validam se um estado ainda não foi avaliado (evitando *loops* e assegurando a completude dos métodos).

Sendo b o número médio de estados filhos, d a profundidade da solução mais rasa na árvore do espaço de estados (árvore EE) e m a sua profundidade máxima, temos as complexidades teóricas dos métodos:

Tabela 1: Complexidades de pior caso

Método	Complexidade de tempo	Complexidade de espaço
`BFS`	$O(b^{d+1})$	$O(b^d)$
`DFS`	$O(b^m)$	$O(bm)$
`A*`	$O(b^d)$	$O(b^d)$
`BestFS`	$O(b^m)$	$O(b^m)$

Nota-se que a escolha da função heurística $h(n)$ pode diminuir radicalmente tempo real do `BestFS`.

3.1 Busca cega (sem informação)

Utilizou-se `breadth_first_graph_search` para o `BFS` e `depth_first_graph_search` para o `DFS`. Enquanto o `BFS` garante a otimalidade da solução, pois expande a árvore EE (i.e. a árvore de busca) nível a nível, o `DFS` percorre a árvore em profundidade, assim o caminho obtido não é necessariamente ótimo.

3.2 Busca informada

Utilizou-se `best_first_graph_search` para o `BestFS` e `astar_search` para o `A*` (que é implementado com o mesmo algoritmo do `BestFS`, porém somando $g(n)$ à heurística utilizada, ou seja, o custo do caminho até o estado atual). Como mencionado na seção 2.3, o `BestFS` não garante solução ótima, mas o `A*` sim, quando a heurística utilizada é admissível.

4 Resultados

A seguir são apresentadas figuras representando as soluções encontradas, com *heatmaps* ilustrando as posições exploradas (do início ao fim da busca, em porcentagem com base na quantidade total de células alcançadas), conforme a legenda mostrada na Figura 3.

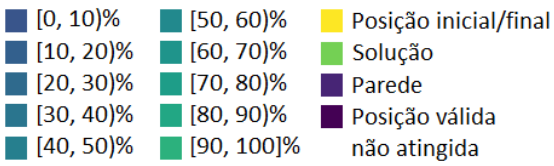

Figura 3: Legenda de cores

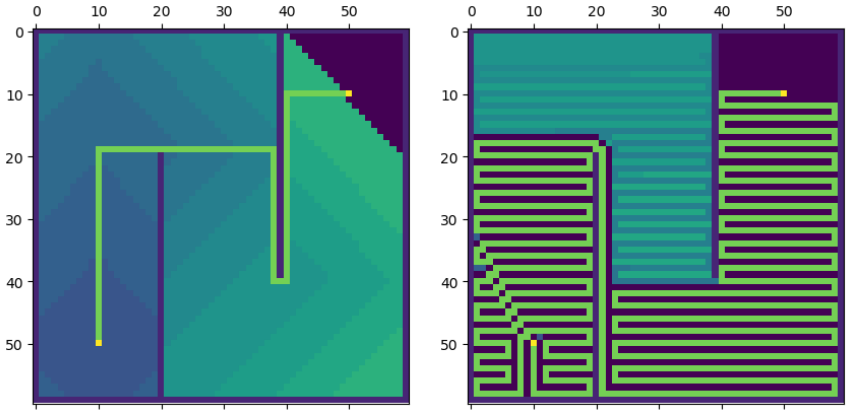

Figura 4: Solução encontrada e caminho explorado pelos algoritmos `BFS` (esquerda) e `DFS` (direita) com ações exploradas na ordem: N, W, E, S

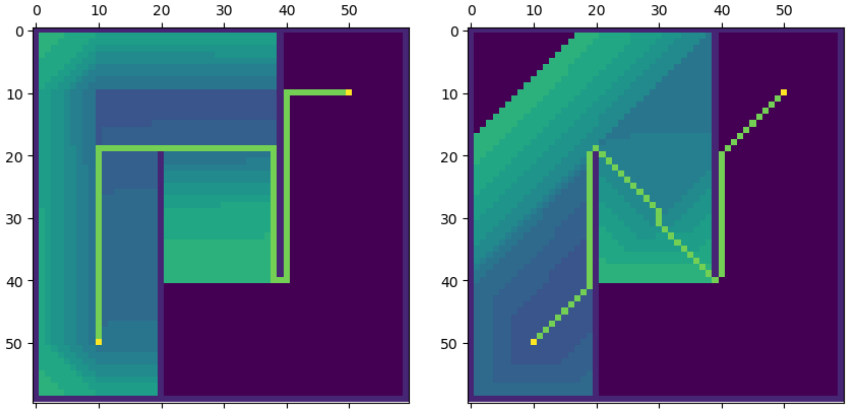

Figura 5: Solução encontrada e caminho explorado pelo `A*` com `diagonal_moves` igual a `False` (esquerda) e `True` (direita)

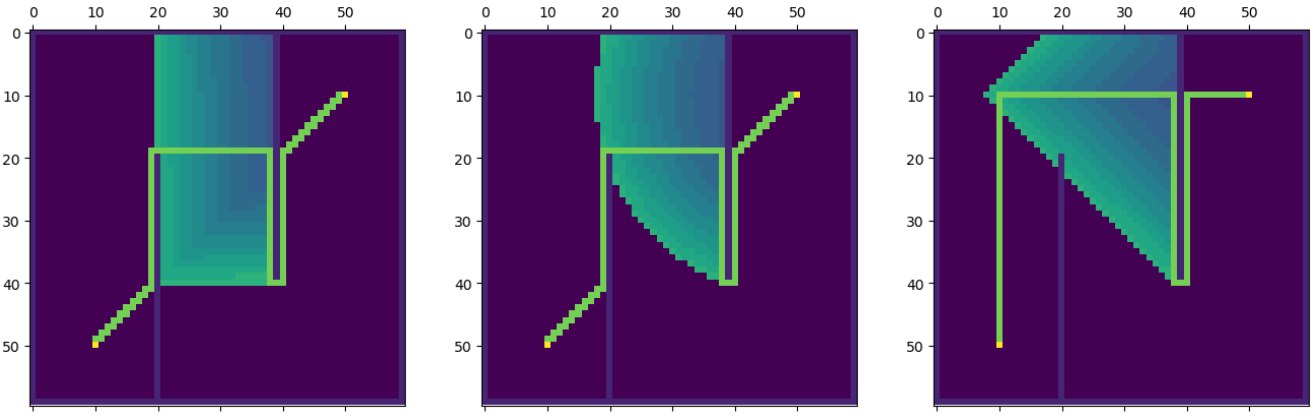

Figura 6: Solução encontrada e caminho explorado pelo `BestFS` com as heurísticas Chebyshev (esquerda), Euclidiana (centro) e Manhattan (direita)

A tabela abaixo mostra o custo computacional (em tempo de execução) e a otimalidade das soluções (no caso, se o caminho encontrado foi mínimo), e é usada posteriormente para análise e discussão. "Nós" referem-se aos estados atingidos pela busca (ou seja, posições do mapa).

Tabela 2: Resultados das buscas

Método	Tempo	Nós explorados	Nós na solução	Otimalidade
`BFS`	150.95 ms	3107	122	Sim
`DFS`	1037.66 ms	2111	1110	Não
`A*` (sem diagonais)	490.11 ms	1883	122	Sim*
`A*` (com diagonais)	694.78 ms	1736	82	Sim*
`BestFS` + Chebyshev	169.18 ms	822	122	Não
`BestFS` + Euclidiana	141.96 ms	733	122	Não
`BestFS` + Manhattan	222.08 ms	807	140	Não

(*) `A*` só garante solução ótima quando a heurística utilizada é admissível [7].

5 Análise e discussão

Conforme esperado [8], a busca em largura (`BFS`) expande sua área de procura radialmente, como mostra a Figura 4 (observe o *heatmap* mostrando sua evolução), de forma que a solução ótima é atingida, porém com isso ela também explora uma quantidade **muito** grande de posições, sendo a que mais o fez (vide Tabela 2).

Já a busca em profundidade (`DFS`), sendo também um método de busca cega, tende a explorar mais nós da árvore de espaço de estados do que as buscas informadas, e além disso não garante otimalidade, tanto que o caminho encontrado foi uma ordem de grandeza maior do que todos os outros.

Entretanto, é importante ressaltar que a ordem em que as ações são exploradas, assim como o próprio mapa, tem uma influência **muito** grande no resultado da `DFS`, sendo interessante randomizar essa ordem para evitar casos patológicos (a avaliação dos resultados para todas as 16 ordens possíveis de se escolher a próxima célula a ser explorada, entre N, S, E e W, fogem do escopo deste relatório, porém podem ser conferidas em [9], assim como também é possível tornar aleatória a sequência escolhida de ações fazendo `shuffle_actions_list = True` na inicialização do problema).

Observando os resultados do algoritmo `A*` vemos que a solução obtida é ótima, pois as heurísticas utilizadas não superestimavam o custo real do caminho. Nota-se por outro lado, que explorou-se duas vezes mais estados do que o `BestFS` (o qual também encontrou soluções de valor ótimo em dois casos), pois todos os nós que tem $g(n) + h(n)$ menor que o valor ótimo são visitados.

Porém, conforme vemos ao utilizar a distância Manhattan com o `BestFS`, o caminho encontrado não é o melhor possível, mesmo sendo ela utilizada com o `A*`. Isto acontece porque a escolha feita no `BestFS` é gulosa, ou seja, o algoritmo tende a seguir um caminho único até o objetivo (diferente do `BFS`, por exemplo), o qual minimiza a função heurística, mas essa estratégia não necessariamente leva ao caminho mínimo.

Logo, apesar de soluções ótimas poderem ser encontradas, elas dependerão da combinação da heurística empregada com o mapa utilizado. Contudo, como uma boa escolha tende a explorar poucos nós, este método de busca pode ser considerado quando quer-se um resultado em pouco tempo, mesmo que o valor seja um pouco distante do ótimo.

Finalmente, ressalta-se um resultado não esperado teoricamente, que o tempo de execução do `BFS` foi inferior ao de praticamente todos os outros, mesmo sendo o que mais explorou estados. Conjectura-se que tal contradição é consequência dos tempos de consulta e alteração de diferentes estruturas de dados utilizadas nos métodos, e do tempo necessário nas buscas informadas para chamar a função heurística a cada nó (tornando-as mais lentas do que o esperado).

6 Conclusões

O uso da classe `Problem` levou a uma implementação natural do problema e de suas especificidades, assim como o uso de matrizes para definição do ambiente. Estas, porém, impõem uma certa restrição de que o mapa deve sempre ser descrito como uma malha retangular (*grid*), de forma que labirintos com formatos irregulares terão uma "borda" que gasta espaço desnecessário de representação.

Os resultados obtidos foram coerentes com o que se esperava de acordo com a teoria (com exceção do rápido tempo de execução do `BFS`, como apontado no final da seção 5, decorrência das formas de implementação dos algoritmos na biblioteca `aima-python`).

Assim, devemos decidir qual método utilizar de acordo com a situação enfrentada, levando em consideração, principalmente, um balanço entre o tempo levado/estados explorados até o alcance de uma solução, a garantia de otimalidade do resultado e as informações conhecidas do problema (o que pode impossibilitar o uso de uma busca heurística).

O código fonte pode ser encontrado em github.com/laurelkeys/ai-intro.

Executando-se `python pathfinding_robot.py` é possível visualizar animações da evolução das soluções que levaram às Figuras 4, 5 e 6 (assim como de outras buscas e mapas).

Referências

[1] AIMA, "Solving problems by searching," accessado em abril de 2019. [Online]. Available: https://github.com/aimacode/aima-python/blob/master/search.ipynb

[2] aima python. [Online]. Available: https://github.com/aimacode/aima-python/blob/master/search.py

[3] "Taxicab geometry — Wikipedia, the free encyclopedia," accessado em abril de 2019. [Online]. Available: https://en.wikipedia.org/wiki/Taxicab_geometry

[4] "Chebyshev distance — Wikipedia, the free encyclopedia," accessado em abril de 2019. [Online]. Available: https://en.wikipedia.org/wiki/Chebyshev_distance

[5] Lyfat, "Euclidean vs chebyshev vs manhattan distance," accessado em abril de 2019. [Online]. Available: https://lyfat.wordpress.com/2012/05/22/euclidean-vs-chebyshev-vs-manhattan-distance/

[6] A. Patel, "Heuristics for grid maps," accessado em abril de 2019. [Online]. Available: http://theory.stanford.edu/ amitp/GameProgramming/Heuristics.html#heuristics-for-grid-maps

[7] E. L. Colombini, "Aula 5 – busca informada e busca local," accessado em abril de 2019. [Online]. Available: https://www.ic.unicamp.br/ esther/teaching/2019s1/mc906/Aula5.pdf

[8] ——, "Aula 4 – agentes do tipo problema e busca sem informação," accessado em abril de 2019. [Online]. Available: https://www.ic.unicamp.br/ esther/teaching/2019s1/mc906/Aula4.pdf

[9] T. L. Chaves. [Online]. Available: https://github.com/laurelkeys/ai-intro/tree/master/dfs