MC906 A - Trabalho 1

Lucy Miyuki Miyagusiku Narita *
*Ciência da Computação - Graduação
E-mail: ra182851@students.ic.unicamp.br

I. Introdução

Este primeiro projeto teve como objetivo modelar e avaliar soluções para se encontrar o melhor caminho de um ponto inicial A até um ponto final B em um ambiente completamente observável, determinístico e estático.

As soluções são baseadas nos métodos de busca vistos em aula: 2 métodos de busca sem informação (BFS, DFS) e 1 método de busca informada ($A*$) com 2 heurísticas diferentes (distância Manhattan e a distância Euclidiana).

O código pode ser encontrado neste repositório[1].

II. Dependências

O projeto foi desenvolvido em Python 3.7, utilizando-se das implementações dos métodos de busca da biblioteca AIMA[1].

Além das dependências[2]da biblioteca citada acima, foi utilizada o memory_profiler[2] para fazer o *profiling* de memória.

III. Modelo Proposto

Para a representação do estado, utilizamos uma tupla de três elementos: $(x, y, last\_movement)$, ou seja, a posição x atual do robô, a posição y atual do robô e a última ação executada. A última ação executada é utilizada para evitar com que o robô "desfaça" seu último movimento a fim de optimizar a busca pela solução ótima. Se o robô se movimentou para frente, por exemplo, não há porque se mover para alguma das diagonais que o levam para as posições à esquerda e à direita do estado anterior.

Para as soluções iniciais, o robô foi considerado como um ponto no mapa que pode se locomover para frente, para trás, para os lados e para as diagonais, desconsiderando sua rotação. O ângulo θ apresentado no enunciado do problema foi desconsiderado. O conjunto de ações válidas então é representado por:

- Mover-se para a diagonal inferior esquerda
- Mover-se para a diagonal superior esquerda
- Mover-se para a diagonal inferior direita
- Mover-se para a diagonal superior direita
- Mover-se para a direita
- Mover-se para a esquerda
- Mover-se para a cima
- Mover-se para a baixo

[1]https://github.com/lnarita/mc906a/blob/master/01/src/search.py

[2]Foi feito o downgrade do matplotlib[3] para a versão 2.0, pois a biblioteca networkx utiliza funções deprecadas e removidas das versões mais recentes do matplotlib.

Poderíamos ter restringido a direção do movimento somente para a direção em que o robô está virado, adicionando novamente o ângulo θ ao estado e utilizando o seguinte conjunto de ações:

- Mover para frente
- Rotacionar α em sentido horário
- Rotacionar α em sentido anti-horário

Porém, dado que isto não afetaria o caminho encontrado na solução (uma vez que limitaríamos $\theta \in \{0, \frac{\pi}{4}, \frac{\pi}{2}, \frac{3\pi}{4}, \pi, \frac{5\pi}{4}, \frac{3\pi}{2}, \frac{7\pi}{4}\}$), foi dado preferência à versão simplificada do problema.

Todas as ações tem custo igual a 1. Como todas as ações nos levam a uma posição vizinha, o custo do caminho $g(x)$ é a quantidade de posições percorridas.

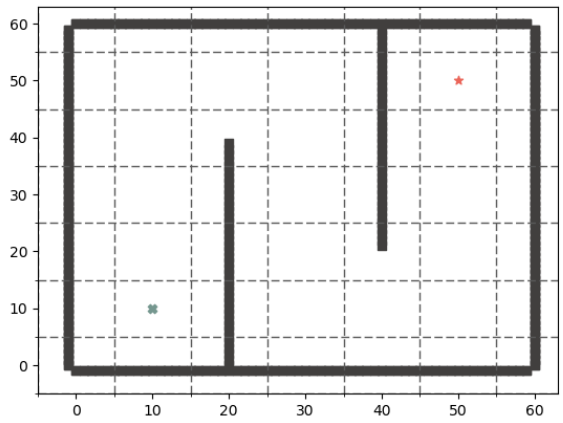

Figura 1. Mapa do ambiente com posição inicial (verde) e objetivo (vermelho)

O mapa (Figura 1) foi representado pelas suas dimensões ($width$, $height$) e um conjunto de tuplas (x, y) que indicam as posições em que existem paredes na sala a fim de economizar memória.

Os métodos de buscas utilizados foram avaliados com base em:

1) optimicidade da solução
2) tempo de execução
3) uso de memória
4) quantidade de nós visitados
5) quantidade de ações executadas
6) quantidade de verificações de objetivo

Para as buscas cegas, foram escolhidos os métodos *BFS* e *DFS*, pois, após feita uma avaliação preliminar, o método *IDDFS* não conseguiu completar a busca em tempo hábil para um mapa de 60×60. O que faz sentido, uma vez que o número de caminhos partindo de um ponto A até um ponto A' tende a aumentar exponencialmente conforme aumentamos a profundidade máxima da busca e se quisermos garantir que o algoritmo encontrará a solução ótima (se ela existir) não podemos manter um conjunto de estados já visitados, uma vez que não há garantias de que a primeira vez que o nó for encontrado ele terá sido encontrado pelo caminho mais curto. Também não faz sentido avaliar o método de busca com custo uniforme, uma vez que será idêntica ao *BFS*, pois os custos unitários de cada ação são iguais.

Especificamente para o *DFS*, reorganizamos as ações em ordem aleatória a fim de evitar que o robô se mova somente em uma direção. Para todos os outros algoritmos, os movimentos para as diagonais tem preferência em relação aos movimentos que se deslocam para a vizinhança-4 da posição atual, pois como estamos utilizando o tamanho do caminho (número de ações) como critério de optimicidade é interessante priorizar as ações que podem ser compostas por outras (em outras palavras, fazem "mais" com custo menor ou igual).

IV. Resultados e Discussão

A. Solução Ótima

A solução ótima consiste de um caminho de tamanho 81. Qualquer solução com mesmo custo será considerada ótima.

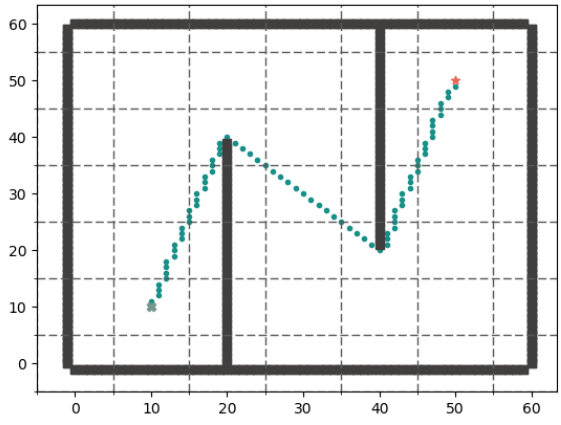

Figura 2. Uma possível solução ótima

B. BFS

Dado que nosso mapa é finito e o caminho existe, o *BFS* nos garante a completude e a optimicidade da solução. Podemos verificar na Figura 3 que o algoritmo realmente

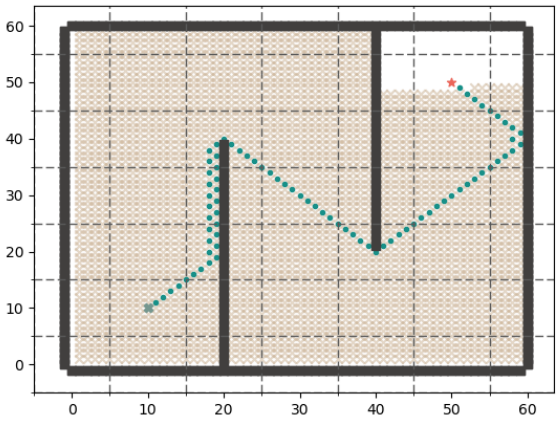

Figura 3. Solução encontrada utilizando BFS

percorre todos as posições até o nível $d = 81$ da solução mais rasa.

C. DFS

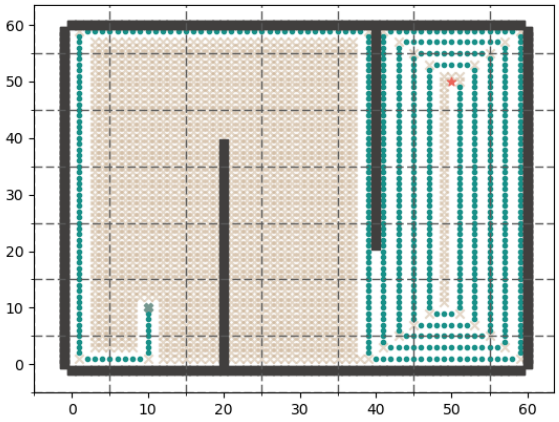

Figura 4. Solução encontrada utilizando DFS (no shuffle)

Considerando a busca em profundidade com verificação de nós visitados, a *DFS* é completa porém não é ótima.

A Figura 4 nos mostra a solução encontrada no caso em que a prioridade das ações são fixas:

1) Mover-se para a baixo
2) Mover-se para a cima
3) Mover-se para a esquerda
4) Mover-se para a direita
5) Mover-se para a diagonal superior direita
6) Mover-se para a diagonal inferior direita
7) Mover-se para a diagonal superior esquerda
8) Mover-se para a diagonal inferior esquerda

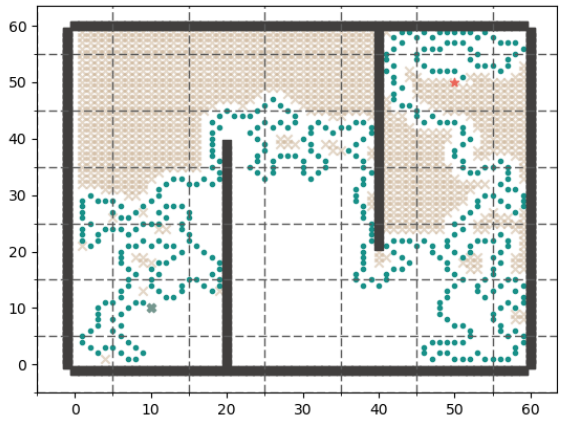

Figura 5. Solução encontrada utilizando DFS (action shuffle)

A Figura 5 é uma das soluções encontradas quando não há prioridade na ordem das ações (ou seja, a escolha é aleatória)

D. A + H1 (Distância Manhattan)*

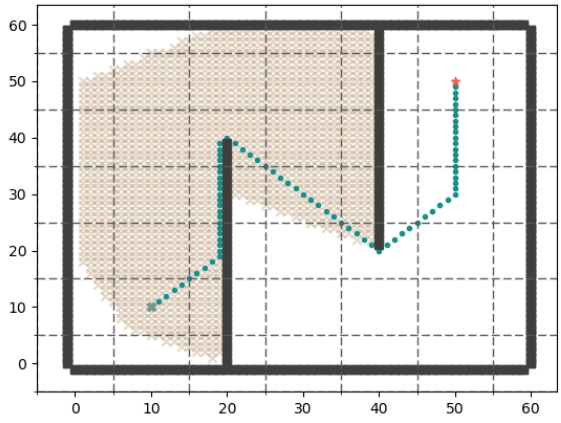

Figura 6. Solução encontrada utilizando A* + H1 (Distância Manhattan)

A distância Manhattan é uma heurística admissível e consistente e continua sendo mesmo quando combinada com o custo do caminho $g(x)$, portanto *A* + H1* será também completa e ótima para o problema.

E. A + H2 (Distância Euclidiana)*

A distância Euclidiana é uma heurística admissível e consistente e continua sendo mesmo quando combinada com o custo do caminho $g(x)$, portanto *A* + H2* será também completa e ótima para o problema.

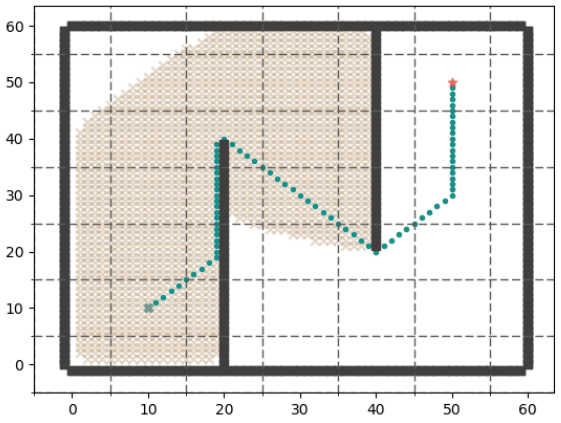

Figura 7. Solução encontrada utilizando A* + H2 (Distância Euclidiana)

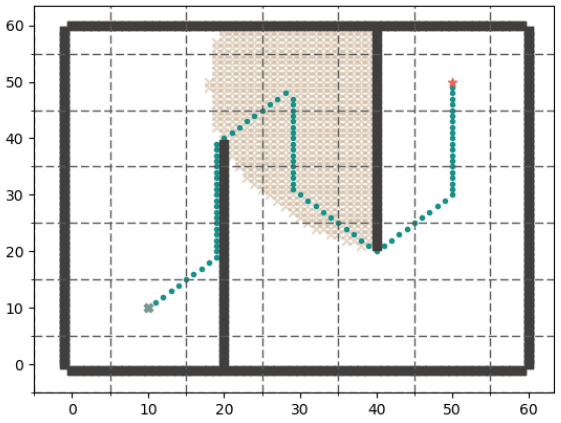

Figura 8. Solução encontrada utilizando A* + H2' (Quadrado da distância Euclidiana)

F. A + H2' (Quadrado da Distância Euclidiana)*

Poderíamos pensar que como estamos preocupados com a relação entre dois valores e não com os valores em si para nossas heurísticas, podemos utilizar o quadrado da distância Euclidiana ao invés da distância em si, uma vez que o cálculo da raíz quadrada de um número costuma ser uma operação custosa.

Como é possível verificar pela Figura 7, *A* + H2'* é completa, porém não é ótima.

O quadrado da distância Euclidiana por si só é uma heurística admissível e consistente, porém uma vez que juntamos com o custo do caminho em nosso mapa discretizado ela deixa de ser consistente. Isso se deve à diferença de escala entre $g(x)$ e $h(x)$, transformando *A** em uma busca gulosa.

Tabela I
MÉTRICAS EXTRAÍDAS POR MÉTODO DE BUSCA

	BFS	DFS	A* + H1	A* + H2
path cost	81	401	81	81
exec time (s)	30.13	10.73	10.12	20.79
mem usage (MiB)	1.3	0.5	0.4	0.4
state explored	3202	1552	1629	1712
actions evaluated	15043	7236	7742	8074
goal checks	3222	1553	1630	1713

G. Comparações

Por fim, temos a tabela I comparativa entre todos os métodos de busca aplicados.

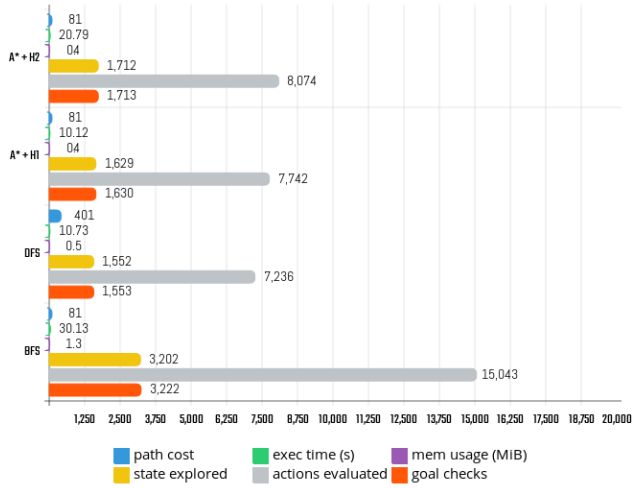

Figura 9. Métricas extraídas por método de busca

Conforme o esperado, o *BFS* foi o método que teve maior consumo de memória, pelo fato de ter que guardar todas as posições visitadas para que consiga traçar o caminho percorrido até a solução mais rasa. Justamente por visitar mais posições e avaliar mais ações, o *BFS* também foi o algoritmo que teve maior tempo de execução.

Podemos verificar que para se encontrar a solução ótima do problema, uma busca informada utilizando a distância Manhattan como heurística seria a melhor opção no caso do modelo matricial adotado. Possívelmente, se tivessemos adotado um modelo em que o robô pode virar para qualquer ângulo a distância Euclidiana teria performado melhor.

V. CONCLUSÕES

A solução proposta é simplista, porém atende as necessidades para a simulação em um mundo discretizado. Dado que conhecemos as dimensões do robô, poderíamos melhorar a solução levando o ângulo $\theta \in [0, 2\pi]$ em consideração e utilizando (x, y) como o ponto onde se encontra o ponto de referência $(0, 0)$ do robô, possibilitando que ele esteja em mais do que uma posição (i, j) da matriz que representa o mapa, aproximando-se do mundo contínuo e tirando máximo proveito da distância Euclidiana.

+————+

REFERÊNCIAS

[1] P. N. Stuart Russell, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009. 1

[2] F. Pedregosa, "Memory profiler." [Online]. Available: https://pypi.org/project/memory-profiler/ 1

[3] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. 1