Instituto Tecnológico de Aeronáutica - ITA
Inteligência Artificial para Robótica Móvel - CT213
Aluno: Arthur de Sousa Vianna

Relatório do Laboratório 2 - Busca Informada

1 Breve Explicação em Alto Nível da Implementação

1.1 Algoritmo Dijkstra

Para todos os algoritmos de busca, utilizou-se uma *priority queue* para armazenar em ordem do menor para o maior custo os nós e, assim, aumentar a eficiência do algoritmo. Inicializou-se o custo do nó inicial para zero e o se adicionou na *priority queue* antes do *loop* principal de busca. Enquanto a lista de prioridades não estiver vazia, visitam-se todos os 8 nós vizinhos ao nó em exploração por meio da função `get_successors` da classe do *grid* e atualiza-se o custo até o nó vizinho em caso de ele não estar fechado e o custo pelo caminho do nó em exploração ao nó vizinho for menor do que o custo já estipulado (inicializam-se os custos para todos os nós como infinito). No caso do Dijkstra, essa comparação é dada somente pelo próprio custo do caminho do nó, identificado por g e atualizada quando necessária utilizando-se a função `get_edge_cost` da classe de mapa de custos. Por fim, quando todos os nós vizinhos ao nó atual forem visitados, o nó atual é fechado. Quando se encontrar o objetivo (nó final), a função retorna o caminho até o nó feito por meio do método `construct_path` e o custo até o nó final. É garantido que o algoritmo de Dijkstra sempre encontre a solução ótima, pois a comparação é feita com os custos de fato do caminho até o nó e com a *priority queue* priorizando os nós de menores custos. Uma observação em relação à implementação da lista de prioridades é que foi utilizada uma *package* na qual não há uma função para atualizar o valor de um item já inserido na lista, ou seja, nesse caso, atualizar o custo de um nó cujo custo já havia sido calculado por outro caminho. Uma alternativa foi em vez de atualizar o custo inserir novamente o nó na lista. Para evitar buscas desnecessárias, quando o nó era extraído da *priority queue*, era verificado se o nó já havia sido fechado. Em caso afirmativo, avançava-se para a próxima iteração do *loop*.

1.2 Algoritmo *Greedy Search*

Implementado de forma semelhante ao algoritmo de Dijkstra, o algoritmo "guloso" apresenta o diferencial de considerar uma estimativa para o cálculo do custo do caminho ótimo do nó atual até o nó objetivo. Essa estimativa é identificada pela função heurística h e, no código, foi considerada como a distância euclidiana do nó atual ao nó final, calculada pela função `distance_to`. A verificação para se atualizarem os custos dos nós é feita da mesma forma do que no Dijkstra. Entretanto, no *loop* de busca, o atributo f de cada nó, nesse caso, igual à estimativa de custo do nó inicial até o nó final (igual a h), é atualizado a cada iteração do nó vizinho ao nó atual em exploração e é considerado como o valor do nó na *priority queue* em vez de somente o custo até o nó atual, como no Dijkstra. Dessa forma, não é garantido que o algoritmo sempre encontra o caminho ótimo, pois não se considera na lista de prioridades o custo de fato do caminho total até

o nó final, mas apenas a menor estimativa de custo até o nó final. No entanto, como o algoritmo preza a todo momento pela menor distância, ele tem uma eficiência significativamente maior do que em relação ao de Dijkstra. É necessário, portanto, analisar o *trade-off* entre um menor custo computacional e um caminho mais custoso, que são informação relevantes no contexto da robótica.

1.3 Algoritmo A*

Implementado de formas análogas aos dois algoritmos anteriores, o algoritmo A*, em contraste ao algoritmo guloso, considera na lista de prioridades não só a função heurística h, calculada da mesma forma que do algoritmo guloso (distância euclidiana), como também o custo do nó inicial ao nó atual. Na prática, o valor f de cada nó é dado pela soma entre o custo g do nó inicial até o nó atual em exploração e a estimativa h do custo do nó atual até o nó final ($f = g + h$). Dado que a estimativa do nó inicial até o nó final é dada por uma função h que satisfaz a alguns requisitos, como de não superestimar o custo real até o nó final ($h^\star$), garante-se que o algoritmo A* sempre encontra a solução ótima. Como é considerada a função heurística, o algoritmo A* é mais eficiente do que o Dijkstra, mas é menos eficiente do que o *Greedy*. Quanto mais próxima a função heurística h é do custo real $h^\star$ até o nó final, mais eficiente é esse algoritmo.

2 Figuras Comprovando Funcionamento do Código

2.1 Algoritmo Dijkstra

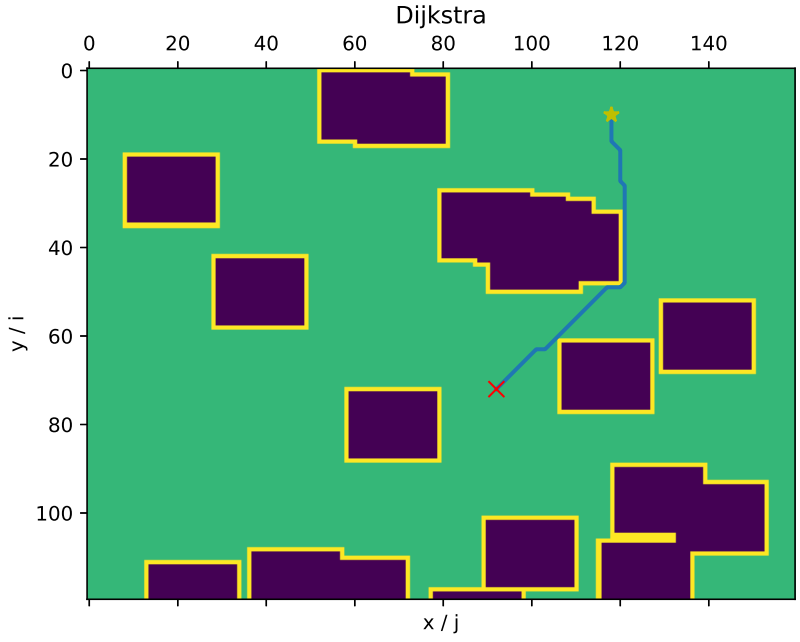

Figura 1: Caminho encontrado pelo algoritmo de Dijkstra para o primeiro problema gerado pelo código.

2.2 Algoritmo *Greedy Search*

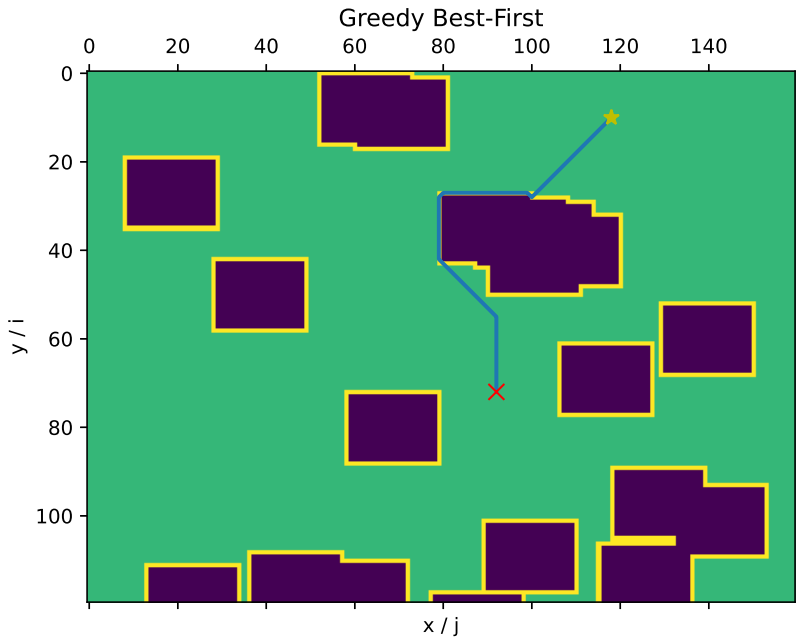

Figura 2: Caminho encontrado pelo algoritmo *Greedy Search* para o primeiro problema gerado pelo código.

2.3 Algoritmo A*

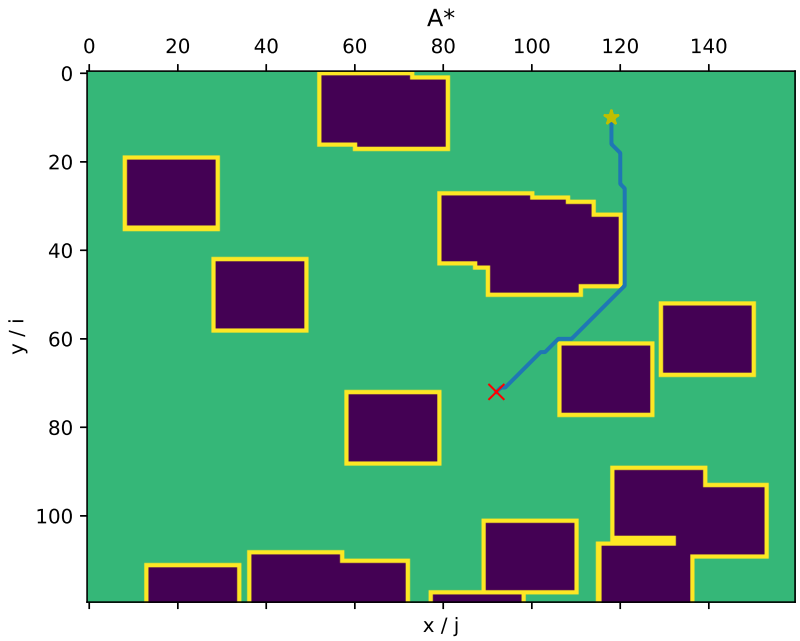

Figura 3: Caminho encontrado pelo algoritmo A* para o primeiro problema gerado pelo código.

3 Comparação entre os Algoritmos

Tabela 1 com a comparação do tempo computacional, em segundos, e do custo do caminho entre os algoritmos usando um Monte Carlo com 100 iterações. Nas simulações, utilizou-se Intel Core i5-1135G7 @ 2.40GHz.

Tabela 1: Tabela de comparação entre os algoritmos de planejamento de caminho.

Algoritmo	**Tempo computacional (s)**		**Custo do caminho**	
	Média	**Desvio padrão**	**Média**	**Desvio padrão**
Dijkstra	0.072100	0.040933	79.829197	38.570962
Greedy Search	0.001895	0.000541	103.117483	58.794021
A*	0.013410	0.012141	79.829197	38.570962