INSTITUTO TECNOLÓGICO DE AERONÁUTICA
CT-213

Laboratório 2: Busca Informada

Professor:
Marcos Ricardo Omena de Albuquerque Máximo

Aluna:
Thayná Pires Baldão

São José dos Campos, 25 de março de 2021

1 *Dijkstra*

Para implementar o algorimo *Dijkstra*, seguiu-se o pseudo-código apresentado no Slide 40 da Aula 2 de CT-213, com algumas algumas modificações. A primeira modificação realizada diz respeito à utilização do parâmetro `closed` do nó para averiguar se ele já foi extraído da fila de prioridades ou não. Isso porque utilizou-se a estrutura de dados `heappq` para implementar a fila de prioridades no laboratório e ela não admite atualizações. Desta forma, toda vez que o custo de um nó precisa ser atualizado, na verdade, ele é reinserido na fila. Sendo assim, ao extrair um nó da fila de prioridades, seta-se o seu parâmetro `closed` para `True`. Então, antes de se analisar um sucessor de um nó, verifica-se se aquele sucessor não tem o parâmetro `closed` igual a `True`, pois, caso tenha, ele não deve ser explorado.

Já a segunda modificação realizada diz respeito à inserção de uma condição de parada na busca, isto é, logo após extrair um nó da fila de prioridades verifica-se se aquele nó é o objetivo. Em caso positivo, para-se a busca e retorna-se o caminho até o objetivo e seu respectivo custo.

Na Figura 1 é possível observar o caminho encontrado pelo algoritmo de *Dijkstra* para o primeiro problema gerado pelo código do laboratório.

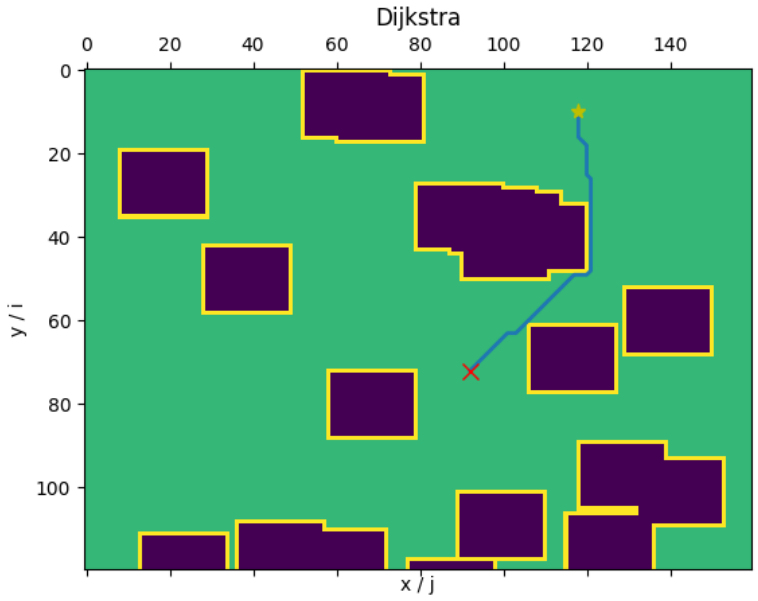

Figura 1: Caminho encontrado pelo algoritmo de *Dijkstra* para o primeiro problema gerado pelo código do laboratório.

2 *Greedy Search*

Para implementar o algorimo *Greedy Search*, seguiu-se o pseudo-código apresentado no Slide 45 da Aula 2 de CT-213, com algumas algumas modificações. A primeira modificação realizada diz respeito à utilização do parâmetro `closed` do nó para averiguar se ele já foi extraído da fila de prioridades ou não, pois utilizou-se a estrutura de dados `heappq` para implementar a fila de prioridades no laboratório e ela não admite atualizações. Desta forma, toda vez que o custo de um nó precisa ser atualizado, na verdade, ele é reinserido na fila. Contudo, diferentemente do *Dijkstra*, setou-se o parâmetro `closed` do nó para `True` logo antes de inseri-lo na fila de prioridades. Isso porque, como o custo do *Greedy Search* não muda, isto é, é sempre igual a distância do nó em questão até o objetivo, a partir do momento que ele entra na fila de prioridades não é necessário explorá-lo novamente. Sendo assim, antes de se analisar um sucessor de um nó, verifica-se se aquele sucessor não tem parâmetro o `closed` igual a `True`, caso tenha, ele não é explorado.

A segunda modificação diz respeito a computar o custo total do caminho encontrado pelo algoritmo. Isso porque para realizar o *Greedy Search* não é necessário calcular o custo real do caminho por ele encontrado.

Não obstante, essa informação foi requisitada como retorno da função de busca do laboratório. Sendo assim, utilizou-se o parâmetro `g` do nó para armazenar o custo real do caminho, enquanto o parâmetro `f` do nó armazenava o custo heurístico necessário para computar o *Greedy Search*. Vale ressaltar que esse custo real (parâmetro `g`) foi inicializado no nó inicial com o valor 0 e a cada sucessor explorado ele foi computado como a soma do parâmetro `g` do antecessor com o custo da aresta que liga o antecessor e o sucessor.

Na Figura 2 é possível observar o caminho encontrado pelo algoritmo *Greedy Search* para o primeiro problema gerado pelo código do laboratório.

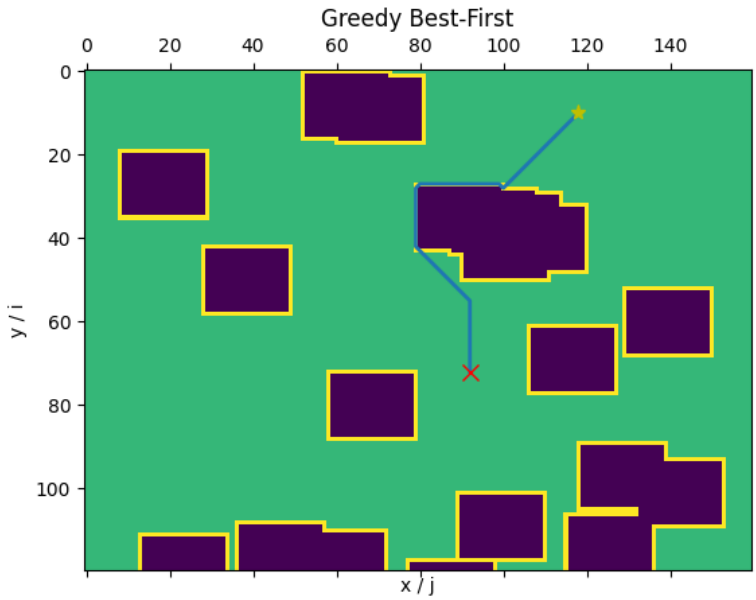

Figura 2: Caminho encontrado pelo algoritmo *Greedy Search* para o primeiro problema gerado pelo código do laboratório.

3 *A**

Para implementar o algorimo *A**, seguiu-se o pseudo-código apresentado no Slide 46 da Aula 2 de CT-213, com uma única alteração. Essa modificação realizada diz respeito à utilização do parâmetro `closed` do nó para averiguar se ele já foi extraído da fila de prioridades ou não. Isso porque utilizou-se a estrutura de dados `heappq` para implementar a fila de prioridades no laboratório e ela não admite atualizações. Desta forma, toda vez que o custo de um nó precisa ser atualizado, na verdade, ele é reinserido na fila. Sendo assim, ao extrair um nó da fila de prioridades, seta-se o seu parâmetro `closed` para `True`. Então, antes de se analisar um sucessor de um nó, verifica-se se aquele sucessor não tem o parâmetro `closed` igual a `True`, pois, caso tenha, ele não deve ser explorado.

Na Figura 3 é possível observar o caminho encontrado pelo algoritmo *A** para o primeiro problema gerado pelo código do laboratório.

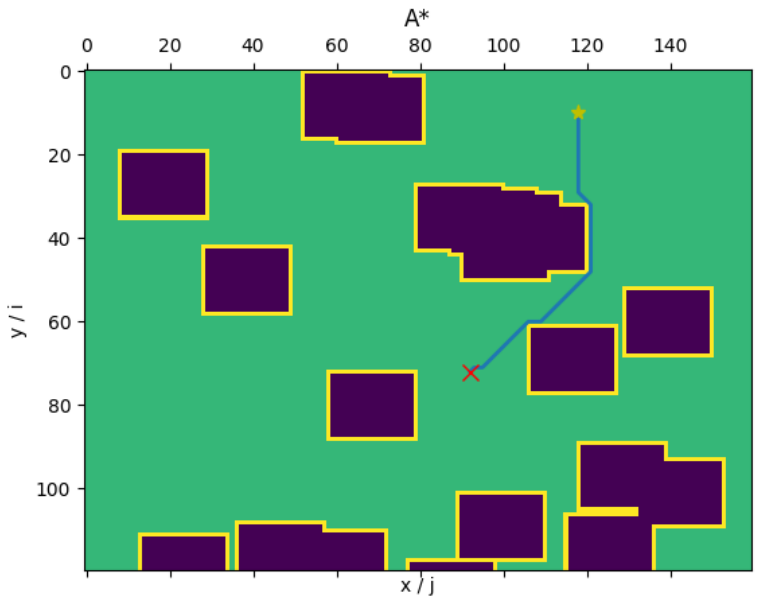

Figura 3: Caminho encontrado pelo algoritmo $A*$ para o primeiro problema gerado pelo código do laboratório.

4 Comparação entre os algoritmos de planejamento de caminho

Na Tabela 1 está apresentada uma comparação entre os três algoritmos de planejamento de caminho implementados neste laboratório.

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	0.1587589716911316	0.0896086636302628	79.8291972826411	38.57096237576532
Greedy Search	0.005274839401245117	0.0008694704961108574	103.34198082325912	59.40972195676692
$A*$	0.03464087963104248	0.032902856358445684	79.8291972826411	38.57096237576532

Note que os algoritmos de *Dijkstra* e $A*$ obtiveram os caminhos com menor custo, visto que ambos são algoritmos de busca ótimos. O *Greedy Search*, contudo, apesar de encontrar um caminho, não necessariamente encontra o menos custoso. Portanto, conforme esperado, ele apresenta custo de caminho maior que os outros dois algoritmos. Essa conclusão também pode ser obtida ao se comparar as Figuras 1 e 3 com a Figura 2, em que é possível perceber que os algoritmos de *Dijkstra* e $A*$ obtiveram os caminhos com menor custo enquanto o algoritmo *Greedy Search* obteve uma solução não-ótima.

Em contrapartida, o algoritmo *Greedy Search* é o mais rápido dentre os três. Isso é esperado, já que o *Greedy Search* não gasta tempo computacional tentando achar a melhor solução. O objetivo desse algoritmo, como seu próprio nome diz, é guloso, isto é, é encontrar a solução o mais rápido possível, mesmo que ela não seja a menos custosa.

Vale dizer ainda que dentre os algoritmo ótimos, o algoritmo $A*$ apresentou melhor performance em termos de tempo computacional. Isso é esperado, dado que o $A*$ se aproveita de uma estimativa do custo entre o nó e o objetivo para avaliar melhor os nós da fila de prioridades. Sendo assim, o algoritmo $A*$ utiliza conhecimento de domínio para encontrar a solução ótima mais rapidamente.