Relatório do Lab 2 de CT-213

Trabalho 2 – Busca Informada em Grafos
Com Dijkstra, Greedy Search, e A*

Aluno:

Bruno Costa Alves Freire

Turma:

T 22.4

Professor:

Marcos Ricardo Omena de Albuquerque Máximo

Data:

16/03/2019

Instituto Tecnológico de Aeronáutica – ITA
Departamento de Computação

1. Implementação dos algoritmos

1.1 Dijkstra

A implementação do algoritmo de Dijkstra foi feita utilizando uma lista para armazenar os nós do *grid* em visitação, operada com algoritmos de *fila de prioridades*, por meio do pacote `heapq` do Python. Antes de tudo, o *grid* de nós sofre um *reset*, para evitar que as informações de um problema de busca interfiram em outras iterações. O *heap* é inicialmente preenchido com o nó inicial, e em seguida itera-se sobre o *heap* (removendo o elemento de maior prioridade, i.e., menor custo) tomando cada nó, marcando-o como fechado (ou seja, seu custo já está definido) e iterando sobre sua lista de sucessores (dada pelo `NodeGrid`).

Para cada sucessor, atualiza-se o seu custo caso o caminho até ele passando pelo nó atual seja menor que o custo desse sucessor até então. Após essa atualização, o sucessor é adicionado ao *heap* com a sua nova informação de custo. O problema da repetição é evitado pois sempre que um nó é retirado do *heap*, ele é ignorado caso seu atributo `closed` tenha valor verdadeiro.

O algoritmo termina quando, após exaurir os elementos da lista de visitação, e ter definido o custo e os antecessores do nó objetivo, retorna o caminho até este, por meio do método `construct_path()`, da classe `NodeGrid`, e seu custo. Na Figura 1, podemos ver o caminho encontrado por esse algoritmo para o primeiro problema. Note que o nó inicial é marcado por uma estrela amarela, e o objetivo é o X vermelho.

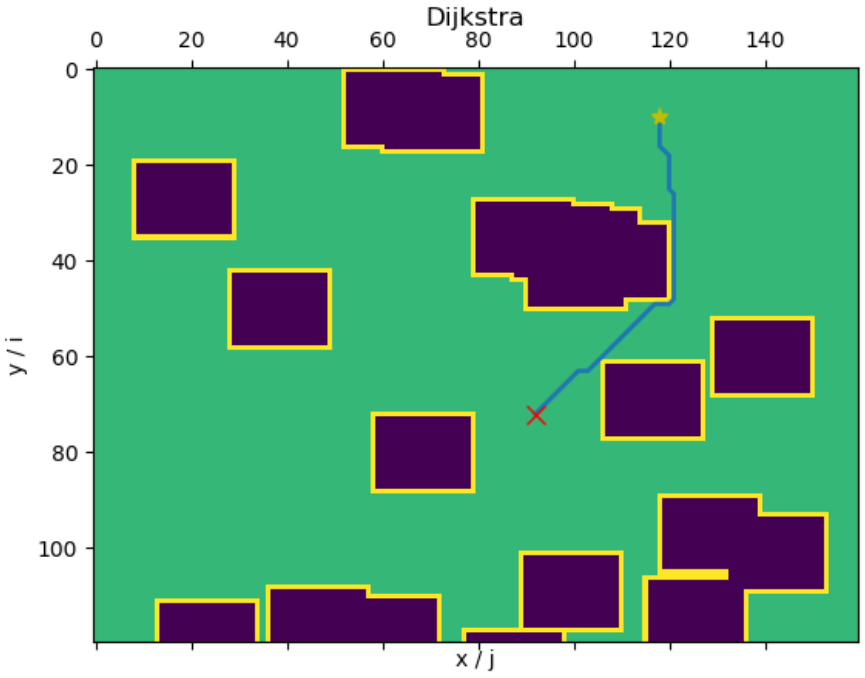

Figura 1: Caminho encontrado pelo algoritmo de Dijkstra.
Custo 78.184 e tempo de execução 0.875 segundos

1.2 Greedy Search

A implementação da busca *greedy* foi feita utilizando a mesma estrutura de lista de prioridades do algoritmo de Dijkstra. Inicialmente, também é feito o *reset* do *grid* de nós, para evitar o uso de informações de outros problemas. Após inserir o nó inicial no *heap*, itera-se sobre o mesmo extraindo sempre o elemento de menor custo. Para um dado nó extraído, itera-se sobre seus sucessores.

Como o *greedy* vai a cada passo buscar o nó mais "vantajoso" (segundo a heurística utilizada, que no caso é a distância euclidiana entre os nós), em cada camada iremos preencher os sucessores com informações sobre seus antecessores e seu custo real, mas apenas se eles não estiverem fechados ou com um antecessor já definido. Isso é feito para evitar que se crie um "*loop* de paternidade" entre os nós. Em particular, um contratempo irritante da implementação desse algoritmo foi o estouro de memória que surgiu na hora de construir o caminho até o objetivo. Várias vezes.

Após preencher essas informações dos sucessores, o algoritmo termina caso o sucessor em questão seja o próprio objetivo, e retorna-se o custo real do nó e o caminho até ele dado pelo método `construct_path()`. Caso ainda não seja o objetivo, o sucessor é adicionado ao *heap*, mas valor que define sua propriedade será o valor da heurística apenas, sem levar em conta o custo real do nó.

Na figura 2, podemos ver o caminho encontrado por esse algoritmo para o primeiro problema. Lembre-se que o início é a estrela amarela, e o X vermelho é o objetivo. Note como o custo do caminho obtido foi maior que o do Dijkstra, mas o tempo foi bem menor.

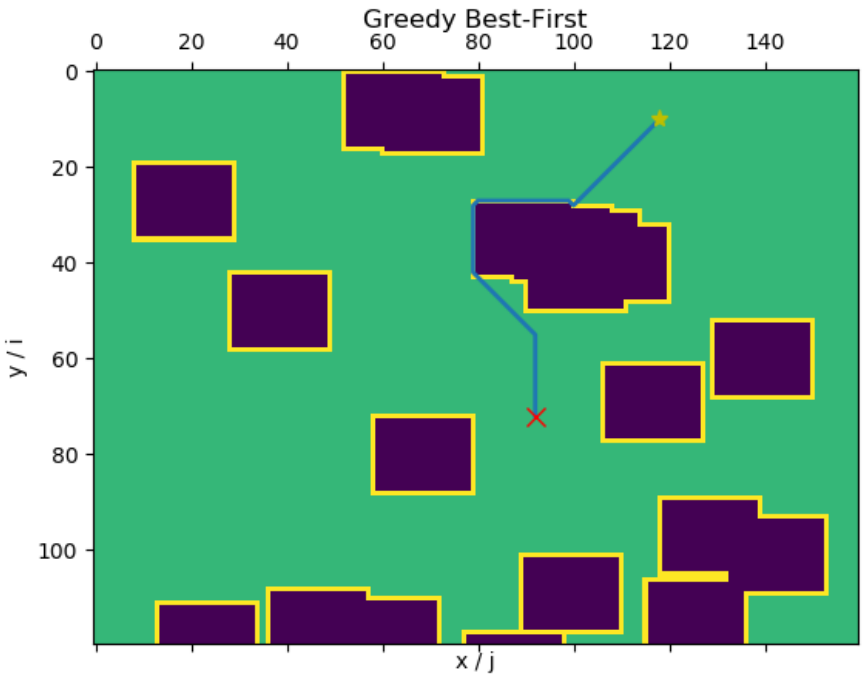

Figura 2: Caminho encontrado pelo algoritmo Guloso.
Custo 135.326 e tempo de execução 0.0156 segundos

1.3 A*

A implementação do algoritmo A* incorpora elementos dos dois algoritmos anteriores. Na inicialização, resetamos o *grid*, e criamos uma fila de prioridades da mesma forma como fizemos anteriormente. Agora utilizamos duas informações relacionadas ao custo de um nó: o atributo `g`, que representa o custo real para chegar até o nó, e o atributo `f`, que incorpora ao atributo anterior o valor da heurística do problema, como uma forma de estimar o custo daquele nó até o objetivo.

A busca ao longo do *heap* é uma mescla das duas formas anteriores. A cada iteração, retira-se o nó de maior prioridade (que aqui equivale ao nó com menor valor de `f`), e caso este esteja *fechado*, tome o próximo, e de todo modo, feche-o. Percorre-se a lista de seus sucessores. Para cada um destes, caso seu atributo `f` seja maior que a estimativa de custo até o objetivo (`g` do nó sendo visitado + distância euclidiana), atualizamos esse valor, definimos o nó sendo visitado como seu antecessor, e colocamos o sucessor no *heap*.

O algoritmo para quando, durante a exploração, encontrar no topo do *heap* o nó objetivo. Nessa ocasião, esse nó já estará com suas informações de antecessores e custo total definidas, de modo que podemos retornar o caminho até o mesmo, e seu custo.

Na figura 3, vemos o caminho resultante da aplicação do algoritmo A*. É importante constatar que o custo do caminho é o mesmo que o do Dijkstra, uma vez que ambos encontram o caminho ótimo. No entanto, vemos que o tempo de execução foi menor aqui (não tanto quanto o do *greedy*, no entanto).

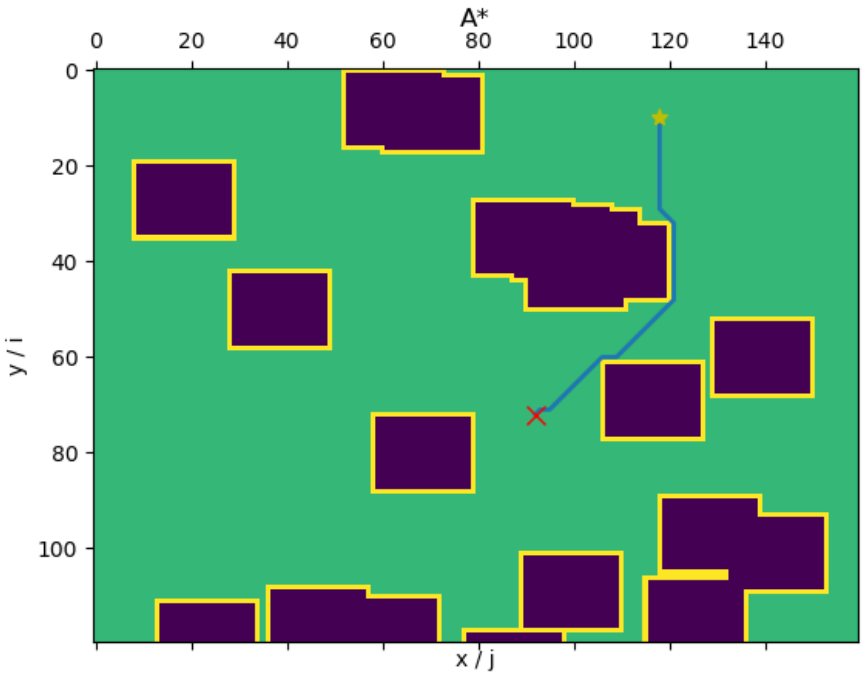

Figura 3: Caminho encontrado pelo algoritmo A*.
Custo 78.184 e tempo de execução 0.0625 segundos

2. Comparação dos Resultados

Para comparar os resultados de tempo de execução e custo do caminho obtido por cada algoritmo, foram simulados 100 casos aleatórios para os quais foram executados os algoritmos de busca, medindo estatísticas do tempo de execução e dos custos totais dos caminhos obtidos. Estes dados foram compilados na tabela 1:

Algoritmo	Tempo Computacional (s)		Custo do caminho	
	Média	Desvio Padrão	Média	Desvio Padrão
Dijkstra	$8.8490 \cdot 10^{-1}$	$9.4572 \cdot 10^{-2}$	79.8292	38.5707
Greedy Search	$1.5626 \cdot 10^{-2}$	$6.9880 \cdot 10^{-3}$	103.3420	59.4097
*A**	$1.0518 \cdot 10^{-1}$	$9.1344 \cdot 10^{-2}$	79.8292	38.5710

Como foram simulados vários casos distintos (pontos inicial e final diferentes), é de se esperar que haja uma certa variância nos custos e nos tempos de execução, para cada método. Contudo, podemos ver que na média, e dentro do desvio padrão, Dijkstra é o mais demorado dos algoritmos, seguido de A* e *greedy*. Levando em conta o desvio padrão, essa ordem de velocidades é corroborada para todos os casos.

No quesito custo, nota-se que o Dijkstra e o A* possuem a mesma média, uma vez que ambos são algoritmos ótimos (A* pode ser subótimo em alguns casos, mas esses não ocorrem para esse problema, com essa heurística). Pensando nisso, a pequena divergência entre os valores de desvio padrão dos dois algoritmos deve ser justificada por *erros numéricos*, uma vez que, diferente do tempo, o custo de um caminho não está sujeito a ruídos. Por fim, o algoritmo *greedy* naturalmente apresenta custos muito maiores, por não ter a garantia de otimalidade na maioria dos casos.

Uma análise estatística mais profunda, comparando por exemplo a variação do custo do caminho e do tempo entre os três métodos em cada problema, permitiria observar que o algoritmo *greedy* é sempre mais rápido, porém comumente subótimo. De fato, para o problema do *grid*, apenas num cenário sem obstáculos entre o objetivo e o início a resposta desse algoritmo seria ótima (sempre que houver obstáculos, o *greedy* vai jogar o robô pro mais perto possível dos obstáculos, agregando custo real maior ao caminho). Além disso, seria possível observar a concordância entre os custos dos caminhos gerados por Dijkstra e A* (em caso de otimalidade de A*), com estes sempre menores ou iguais ao *greedy* (devido à otimalidade).