Aplicação de A* e Dijkstra em labirintos

Guilherme Henrique Resende de Andrade, Ramon Gonçalves Gonze

04 de julho de 2018

1 Introdução

Este trabalho prático desenvolve e experimenta, de uma forma analítica, a desenvoltura dos algoritmos de Dijkstra e A* para encontrar o caminho mais curto entre dois pontos de um labirinto.

O Algoritmo de Dijkstra encontra o caminho mais curto entre um vértice de origem s e todos os demais vértices de um grafo. Com uma abordagem gulosa, ele utiliza a técnica de relaxamento de arcos como um meio de se arrepender de escolhas erradas feitas previamente.

Já o Algoritmo A* busca encontrar o menor caminho entre dois vértices de um grafo. A versão implementada neste trabalho se baseia no Dijkstra e, em adicional, faz o uso de uma heurística para reduzir o espaço de busca, acelerando o processamento da solução.

No geral, temos que ambos os algoritmos podem ser utilizados para resolver o problema entre o menor caminho de dois pontos em um labirinto. Como ambos se propõem a encontrar menores caminhos em grafos direcionados com pesos não-negativos nas arestas, o labirinto também será modelado dessa forma.

2 Labirintos

Um labirinto é representado neste trabalho na forma de um grafo direcionado $G = (V, E)$ onde V é o conjunto de vértices e E é o conjunto de arcos. Ele pode ser representado graficamente como um retângulo de dimensão $n \times m$ em um plano euclidiano, onde cada quadrado de dimensão 1×1 é um vértice do grafo G.

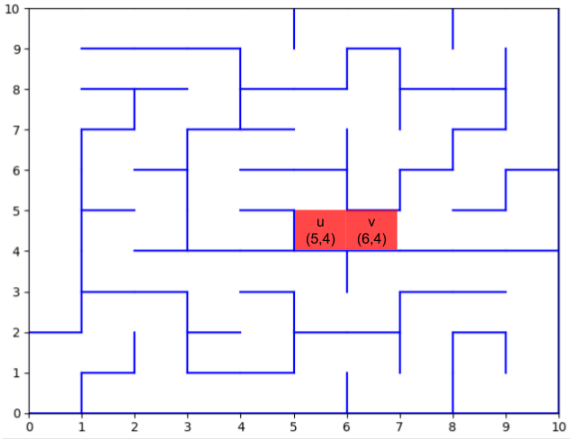


Figura 1: Representação do grafo como um labirinto de tamanho 10×10 no plano euclidiano.

O retângulo que contém as paredes mais externas do labirinto possui um vértice no ponto $(0, 0)$ do plano e outro no ponto (n, m), conforme a Figura 1.

Definição 1 - conjunto E: *Todo vértice com coordenada* (x, y) *pode possuir arcos para os vértices com coordenadas* $(x, y-1)$, $(x+1, y)$, $(x, y+1)$ *e* $(x-1, y)$, *ou seja, os pontos à esquerda, acima, à direita e abaixo no plano, respectivamente, com exceção dos vértices que pertencem às paredes mais externas do retângulo: estes não possuirão arcos para os vértices que estão fora do labirinto.*

Todos os arcos $(u, v) \in E$ *têm peso 1.*

Como citado anteriormente, cada célula do plano (em coordenadas inteiras) é um vértice do grafo. Por exemplo, vértices u e v assinalados em vermelho na Figura 1 são dois vértices adjacentes, logo $(u, v), (v, u) \in E$.

2.1 Construção do labirinto

O algoritmo utilizado para a construção do labirinto é a *busca em profundidade*, ou *DFS*. A contstrução do labirinto se dá da seguinte forma:

Considere cada vértice (ou ponto no grafo) começando com 4 paredes, ou seja, todos os vértices possuem todos os arcos, conforme a Definição 1. Sorteia-se um vértice u aleatório, e adiciona-o à uma pilha P inicialmente vazia. O vértice u é desempilhado e o algoritmo escolhe aleatoriamente um vizinho v de u que ainda não foi visitado. O algoritmo remove a parede entre u e v, marca v como visitado e adiciona-o em P. Continua-se o processo para os vértices que ainda possuem vizinhos não visitados. Quando o algoritmo atinge um vértice em que todos os seus vizinhos já foram visitados (um beco sem saída), ele retorna no caminho percorrido (através de P) até achar um vértice com vizinho não visitado. O processo continua até todo vértice ser visitado, fazendo o algoritmo voltar no caminho percorrido até o vértice inicial. Todas as paredes que foram removidas durante a execução, existirão no conjunto E de arcos do grafo, enquanto que as paredes não removidas, terão seus respectivos arcos removidos de E.

Sabendo que todos os vértices foram visitados, temos certeza de que existe sempre um caminho entre dois vértices quaisquer do labirinto, ou seja, tem-se ao final um grafo G fortemente conexo. Com essa propriedade, pode-se aplicar os algoritmos de Dijkstra e A* para encontrar um menor caminho entre dois pontos do labirinto.

A complexidade do algoritmo para gerar o labirinto é a complexidade da busca em profundidade, que se dá por $O(|V|+|E|)$. Para um labirinto de dimensão $n \times m$, tem-se nm vértices, e assumindo que cada vértice tem no máximo 4 vizinhos, a complexidade do algoritmo será $O(nm + 4nm) = O(nm)$.

3 Dijkstra

O algoritmo de Dijkstra calcula a menor distância entre um vértice e os demais vértices de um grafo. Para a solução do menor caminho entre dois pontos de um labirinto, o aloritmo será executado a partir de um dos pontos, e, no momento em que o outro ponto for adicionado ao conjunto solução S, a execução é terminada.

A seguir pode-se visualizar o pseudo-código relativo ao Dijkstra:

Algorithm 1 Dijkstra

1: **function** DIJKSTRA($G = (V, E), s, t$) ▷ s é a origem e t o destino
2: $\forall u \in V : dist(u) = \begin{cases} 0, & \text{se } u = s. \\ \infty, & \text{caso contrário.} \end{cases}$
3: $Q = V$
4: $S = \emptyset$
5: **while** $Q \neq \emptyset$ **do**
6: $u = extraiMinimo(Q)$ ▷ $\forall u \in Q : min(dist(u))$
7: $S = S \cup \{u\}$
8:
9: **if** $u = t$ **then**
10: **return**
11:
12: **for** cada vizinho v de u **do**
13: $dist(v) = min(dist(v), dist(u) + custo(u, v))$

onde $custo(u, v)$ é o peso do arco (u, v) e *extraiMinimo* é uma função que remove o menor elemento da fila de prioridades Q. A fila de prioridades é um heap binário que mantém na raiz o vértice

com a menor distância até s. Sempre que a distância de um vértice até s é atualizada na linha 12 do Algoritmo 1, a fila de prioridades é reordenada.

3.1 Prova

A corretude do algoritmo de Dijkstra se baseia no seguinte teorema:

Teorema 1 - Seja $\delta(s, v)$ o caminho mínimo entre a origem s e o vértice v. Ao final da execução do Dijkstra para um grafo $G = (V, E)$ direcionado ponderado com pesos não-negativos, $dist(u) = \delta(s, u)$ para todo $u \in V$.

Prova (por contradição)
Seja S o conjunto solução. Usa-se o seguinte invariante:

No início de cada iteração do laço **while** (linha 4 do Algoritmo 1), $dist(u) = \delta(s, u)$ para todo $u \in S$.

Se mostrarmos que para cada vértice $u \in V$, tem-se $dist(u) = \delta(s, u)$ no momento em que u é adicionado ao conjunto S, recorre-se à propriedade do limite superior para mostrar que a igualdade é válida em todos os momentos daí em diante.

No início da execução $S = \emptyset$, e tem-se que o invariante é trivialmente verdadeiro. Deseja-se mostrar que, para cada vértice u adicionado ao conjunto S (linha 5 do Algoritmo 1), $dist(u) = \delta(s, u)$. Seja u o primeiro vértice adicionado ao conjunto S onde $dist(u) \neq \delta(u, v)$. Tem-se que $u \neq s$ pois s é o primeiro vértice adicionado à S, e $dist(s) = \delta(s, s) = 0$. Deve-se assumir que exista algum caminho de s a u, caso contrário $dist(u) = \delta(s, u) = \infty$, e neste caso seria violada a suposição inicial de que $dist(u) \neq \delta(u, v)$. Como existe pelo menos um caminho, considere p o caminho mínimo de s até u. Seja y o primeiro vértice ao longo de p tal que $y \notin S$ e $x \in S$ o predecessor de y. Podemos decompor o caminho p em dois caminhos $p_1 \rightarrow p_2$ tal que $p_1 = s \rightsquigarrow x$ e $p_2 = y \rightsquigarrow u$, logo, $p = s \rightsquigarrow x \rightarrow y \rightsquigarrow u$.

Tem-se que $dist(y) = \delta(s, y)$ quando u é adicionado à S, pois $x \in S$, e como assumiu-se que u foi o primeiro vértice a ser adicionado a S tal que $dist(u) \neq \delta(s, u)$, então $dist(x) = \delta(s, x)$ quando x foi adicionado a S. Logo $dist(y)$ foi atualizado no momento que x foi adicionado à S, e agora possui o arco (x, y), fazendo com que $dist(y) = \delta(s, y)$.
Como y aparece antes de u em um caminho mínimo de s a u e todos os pesos de arestas são não-negativos, tem-se que $\delta(s, y) \leq \delta(s, u)$, e então

$$dist(y) = \delta(s, y) \leq \delta(s, u) \leq dist(u)$$

Contudo, como $u, y \notin S$ no momento em que u foi adcionado à S, então $dist(u) \leq dist(y)$. Como viu-se que $dist(y) \leq dist(u)$ e $dist(u) \leq dist(y)$, então $dist(u) = dist(y) = \delta(s, y) = \delta(s, u)$. Contradição, pois assumiu-se que u era o primeiro vértice adicionado à S onde $dist(u) \neq \delta(s, u)$.

3.2 Complexidade

O algoritmo de Dijkstra implementado neste trabalho utiliza uma fila de prioridades modelada como um heap binário, que é construído em $O(|V|)$ e faz consultas em $O(\log |V|)$ - função *extraiMinimo* do Algoritmo 1. Essa função é executada $|V|$ vezes, pois cada vértice removido de Q é adicionado exatamente uma vez à S. Toda vez que um vértice é removido pela *extraiMinimo*, o heap deve continuar ordenado, e essa operação tem custo $O(\log |V|)$ no heap binário. Como cada vértice $u \in V$ é adicionado ao conjunto S exatamente uma vez, cada arco correspondente aos vizinhos de u é examinado exatamente uma vez (laço **for** da linha 12 do Algoritmo 1), então, este laço é executado $|E|$ vezes.

Portanto, o tempo de execução total é $O((|V| + |E|) \log |V|)$. Como o labirinto é um grafo fortemente conexo, cada vértice tem no mínimo 1 arco e no máximo 4, logo $|V| \leq |E| \leq 4|V|$. Então a complexidade final será $O(|E| \log |V|)$. Sabendo que $|V| = nm$ (dimensão do labirinto) e que $|E| \leq 4|V|$ (cada vértice tem no máximo 4 vizinhos), a complexidade em termos da dimensão do labirinto é $O(nm \log nm)$.

4 A*

O algoritmo A* é muito utilizado em situações das quais é necessário encontrar de maneira eficiente o menor caminho entre dois pontos de um grafo. Assim como o algoritmo de Dijkstra da seção anterior, este também irá finalizar sua execução assim que atingir o vértice destino.

Uma diferença entre os dois algoritmos é que, enquanto Dijkstra encontra o menor caminho entre um vértice e todos os demais, o A* encontra o menor caminho entre um par de vértices. Outra diferença é a heurística utilizada pelo A* para reduzir o espaço de busca. De forma geral, a função heurística fornece uma indicação do melhor caminho para a solução, ou seja, um custo estimado. Entretanto, deve ser escolhida cuidadosamente visto que a superestimação da mesma pode levar o algoritmo à uma solução não ótima.

O Algoritmo 2 ilustra o funcionamento do A*:

Algorithm 2 A*

1: **function** ASTAR($G = (V, E), s, t$) ▷ s é a origem e t o destino
2: $\forall u \in V : dist(u) = \begin{cases} 0, & \text{Se } u = s. \\ \infty, & \text{caso contrário.} \end{cases}$
3:
4: $\forall u \in V : \Phi(u) = \sqrt{(t_x - u_x)^2 - (t_y - u_y)^2}$ ▷ distância euclidiana
5:
6: $Q = V$
7: $S = \emptyset$
8: **while** $Q \neq \emptyset$ **do**
9: $u = extraiMinimo(Q)$ ▷ $\forall u \in Q : min(dist(u) + \Phi(u))$
10: $S = S \cup \{u\}$
11:
12: **if** $u = t$ **then**
13: **return**
14:
15: **for** cada vizinho v de u **do**
16: $dist(v) = min(dist(v), dist(u) + custo(u, v) + \Phi(v))$

A heurística utilizada pelo A* descrita no começo desta seção é representada por $\Phi(u)$, que indica a distância euclideana entre u e o vértice de destino t. A escolha dessa heurística se deve pelo fato de o objetivo ser executar o algoritmo em labirinos. Como cada vértice do grafo está sendo representado como um ponto no plano euclidiano, é possível calcular a distância euclidiana entre dois vértices.

A função *extraiMinimo* no Algoritmo 2, diferentemente do Dijkstra, agora remove o vértice de Q com a menor soma da distância dele ao vértice s e a distância euclidina dele ao vértice t, ou seja, $\forall u \in Q : min(dist(u) + \Phi(u))$.

4.1 Prova

O algoritmo A* fornece solução ótima se a heurística utilizada é **admissível**, isto é

$$\forall v \in V : h(v) \leq \delta(v, t) \tag{1}$$

onde $h(v)$ é a função heurística e $\delta(v, t)$ é o caminho mínimo de v à t.

No caso do labirinto, a função heurística utilizada é a distância euclidiana. Seja $d(v, t)$ a distância euclidiana entre os vértices v e t. Como cada vértice no labirinto está ligado aos vértices à esquerda, acima, à direita e abaixo, o valor de $\delta(v, t)$ está limitado inferiormente pela *distância de Manhattan* entre v e t, ou seja

$$M(v, t) \leq \delta(v, t)$$

onde $M(v, t)$ é a distância de Manhattan entre v e t.

Considere que a distância euclidiana dois vértices no labirinto seja a hipotenusa de um triângulo retângulo, conforme a Figura 2.

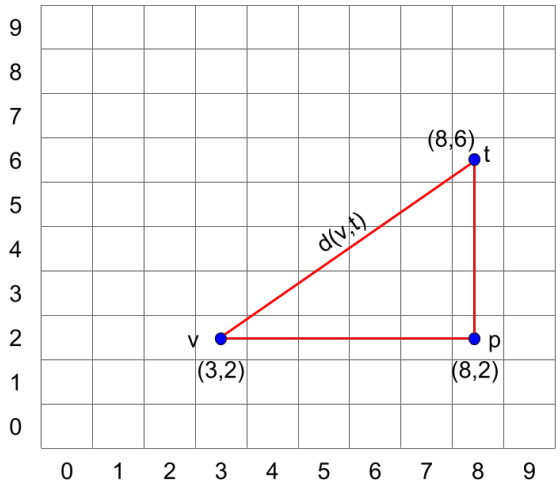


Figura 2: Representação de um triângulo retângulo que pode ser formado ao se analisar os vértices v e t.

Nota-se que a soma dos lados $\overline{VP}$ e $\overline{PT}$ do triângulo da Figura 2 representam a distância de Manhattan entre v e t, logo

$$M(v,t) = \overline{VP} + \overline{PT}$$

A condição de existência de um triângulo diz que $\overline{VT} < \overline{VP} + \overline{PT}$, ou seja,

$$d(v,t) < M(v,t) \tag{2}$$

e como $M(v,t) \leq \delta(v,t)$, então $d(v,t) < \delta(v,t)$, satistazendo a propriedade (1). Portanto, a heurística aplicada é admissível, e o algoritmo de A* encontrará a solução ótima. Há somente dois casos onde não se pode representar o triângulo retângulo:

1. Quando o vértice de origem for o mesmo de destino, ou seja, $v = t$. Neste caso, trivialmente vemos que $d(v,v) = M(v,v) = 0$, que também satisfaz a propriedade (1).
2. Quando o vértice de origem e destino estiverem na mesma linha ou coluna. Neste caso $d(v,t) = M(v,t)$, que também satisfaz a propriedade (1).

Uma outra propriedade da otimalidade do A* é garantir que a heurística seja *consistente*, isto é, quando um nó u for selecionado na linha 9 do Algoritmo 2, o caminho mínimo até ele foi encontrado. Uma heurística é consistente se, para um vértice u e os seus sucessores u', o custo estimado de atingir o destino t a partir de u não é maior que o custo de chegar a u' somado ao custo estimado de u' para t. Isto está sendo satisfeito, conforme a inequação em (2).

4.2 Complexidade

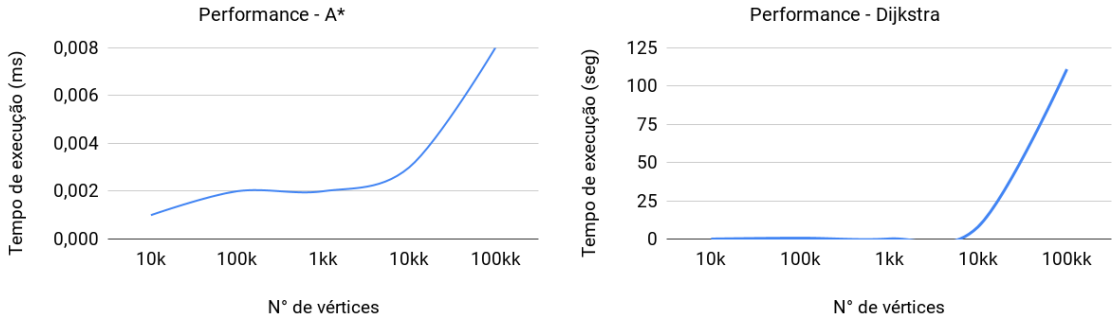
A complexidade do A* depende da heurística. No pior caso de um espaço de busca ilimitado, o número de vértices que são expandidos (vértices da linha 9 do Algoritmo 2) é exponencial na profundidade p da solução (o caminho mínimo até o destino t). A complexidade pode ser escrita como $O(b^p)$, onde b é o fator de ramificação da árvore gerada pelo algoritmo (a média do número de sucessores por estado). Esta complexidade assume que t é alcançável a partir da origem s, pois caso contrário, com um espaço de busca infinito, o algoritmo nunca terminaria a execução.

Em termos do tamanho do grafo, pode-se considerar a complexidade do A* como a mesma do Dijkstra descrita na Seção 3.2: $O(|E| \log |V|)$ ou $O(nm \log nm)$ em termos do tamanho do labirinto. Isto é válido pois a heurística utilizada neste trabalho é a distância euclidiana, que é calculada em $O(1)$. De fato, ao observar os Algoritmos 1 e 2, percebe-se a razão da complexidade de ambos ser a mesma.

5 Experimentos

Apesar da complexidade dos algoritmos depender de $|E|$, o tamanho deste conjunto cresce conforme o número de vértices no labirinto (cada vértice tem no mínimo um arco). Portanto, o experimento para averiguar o tempo de execução dos algoritmos foi realizado variando o número de vértices, ou seja, o tamanho $n \times m$ do labirinto. Foram realizados testes e verificou-se que o formato do labirinto (quadrado ou retângulo) não influencia no comportamento dos algoritmos.

Os algoritmos foram executados para encontrar o caminho mínimo entre o s e t, para $s = (0, 0)$ e $t = (n - 1, m - 1)$.



(a) Algoritmo A*

(b) Algoritmo de Dijkstra

Figura 3: Tempo de execução dos algoritmos em função da variação do número de vértices.

Na Figura 3 pode-se visualizar que ambos os algoritmos assumem um comportamento assintótico tal como descrito em 3.2 e 4.2, ao que diz respeito ao aspecto da curva.

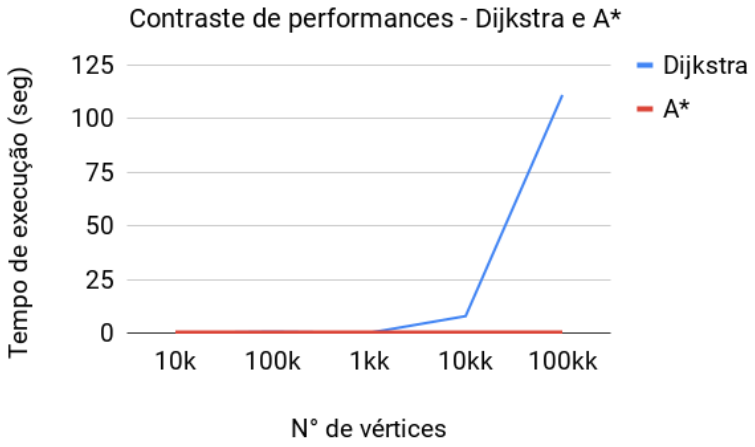


Figura 4: Comparação entre os Algoritmos de Dijkstra e A*

Analisando as projeções de forma conjunta, a performance do algortimo de A* é extremamente superior ao de Dijkstra no dado cenário, permanecendo na ordem dos microssegundos, enquanto que o último ultrapassa minutos de execução com facilidade.

6 Conclusão

O problema do caminho mínimo entre dois pontos de um grafo direcionado é muito bem resolvido pelo algoritmo do A*. A requisição para encontrar a solução ótima é haver uma heurística admissível, e ela pode ser encontrada em problemas de diversos domínios diferentes. E um desses domínios é o de labirintos, aprensentado neste trabalho. Conforme demonstrado pelos experimentos, há uma superioridade muito grande do tempo de execução do A* em relação ao algoritmo de

Dijkstra. Ambos encontram a solução ótima, e portanto, conclui-se que o A* é a melhor opção a ser adotada neste contexto. A implementação utilizada no trabalho se encontra disponível no GitHub [1].

Referências

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[2] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[1] https://github.com/ramongonze/shortest_path_mazes