Inteligência Artificial
Apontamentos para as aulas

Luís Miguel Botelho

Departamento de Ciências e Tecnologias da Informação
Instituto Superior de Ciências do Trabalho e da Empresa

Agosto de 2017

Algoritmos de procura

Descrição, por exemplos, dos métodos de procura em profundidade-primeiro, em largura-primeiro, e do melhor-primeiro; do algoritmo A*, e dos conceitos mais importantes envolvidos nos métodos de procura.

Conteúdo

Na fase inicial da investigação em Inteligência Artificial, houve a forte convicção de que um programa inteligente seria capaz de resolver problemas através de uma abordagem de força bruta em que o programa tentaria todas as possíveis vias para solucionar o problema considerado, até encontrar uma solução. Esta ideia tomou forma nos chamados algoritmos de procura.

Fundamentalmente, um algoritmo de procura receberia uma descrição do problema a resolver e descrições das operações elementares que podem ser feitas para resolver o problema. Um problema é especificado através das especificações formais do estado inicial e de um ou mais estados finais.

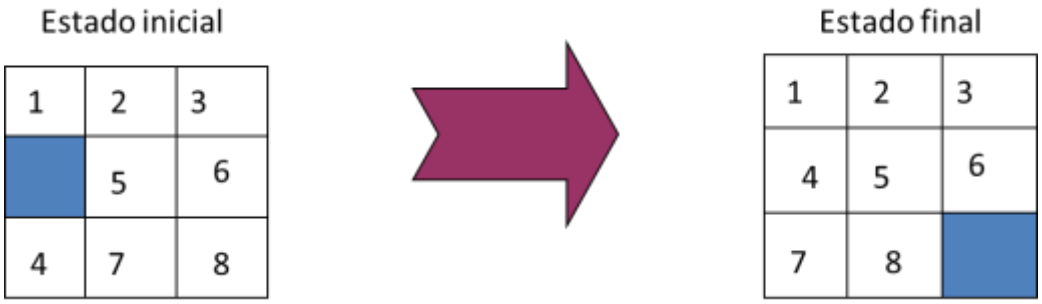


Figura 1 – Especificação de um problema do 8-puzzle

Na figura, o problema consiste em resolver um puzzle (*8-puzzle*): o algoritmo tem de descobrir a sequência de operações que permite passar do estado inicial até ao estado final. Naturalmente, na maioria das utilizações, os estados inicial e final do *8-puzzle* podem ser representados de formas mais práticas do que através de imagens, por exemplo através de listas de números. Neste caso, os dois estados poderiam ser representados pelas duas sequências seguintes:

Estado inicial: [1, 2, 3, 0, 5, 6, 4, 7, 8]

Estado final: [1, 2, 3, 4, 5, 6, 7, 8, 0]

em que o espaço em branco é representado pelo número 0.

As operações que se podem operar neste problema são, por exemplo, as seguintes:

- Mover o espaço em branco para baixo;
- Mover o espaço em branco para cima;
- Mover o espaço em branco para a direita; e
- Mover o espaço em branco para a esquerda.

Cada uma destas operações tem de ser especificada ao algoritmo, usando uma linguagem de representação que ele possa processar, por exemplo:

```
move_right(State1, State2):-
  space_position(State1, Pos, Part1, [X|Part2]),
  \+ member(Pos, [3, 6, 9]),
  append(Part1, [X, 0|Part2], State2).
```

Não é importante perceber o programa apresentado; é importante sim perceber que tem de ser um programa que recebe a representação de um estado e produz a representação do estado que resulta da aplicação da operação ao estado recebido.

Os algoritmos que estão na base de uma enorme variedade de outros algoritmos de procura são os algoritmos de procura em profundidade primeiro (*depth-first search*) e de procura em largura primeiro (*breadth-first search*).

A procura em profundidade primeiro e a procura em largura primeiro são algoritmos de força bruta no sentido em que procuram, às cegas, todas as possíveis sequências de operações até encontrar uma solução para o problema recebido. Rapidamente se percebeu que, mesmo para computadores extremamente rápidos e com memórias enormes, a procura de soluções não deveria ser feita às cegas, pois a maioria dos problemas reais envolve quantidades astronómicas de possibilidades, o que conduz a tempos de resolução inaceitáveis em muitas circunstâncias. Era necessário guiar a procura de soluções na direção mais promissora, o que exige conhecimento do domínio do problema.

A ideia de guiar o processo de procura para a via aparentemente melhor deu origem à procura do melhor primeiro (*best-first search*). Este tipo de procura, em vez de proceder cegamente, em cada decisão que tem de tomar, encaminha-se para aquela que parecer melhor.

As próximas secções ocupam-se destes três tipos de algoritmos: procura em profundidade primeiro, procura em largura primeiro, e procura do melhor primeiro.

1 Conceitos importantes nos algoritmos de procura

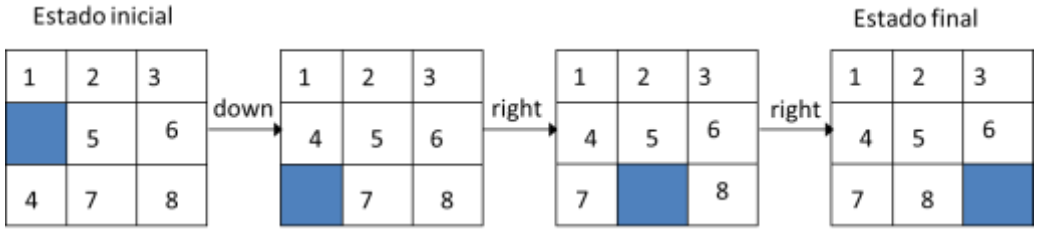
São importantes os seguintes conceitos:

- Estado, estado inicial, e estado final
- Espaço de estados ou de procura
- Operadores
- Solução

- Distinção entre o tempo de execução do algoritmo de procura e o comprimento da solução encontrada
- Distinção entre a ordem pela qual os nós são gerados e a ordem pela qual eles são expandidos / visitados
- Independência entre os algoritmos, em si, e o domínio do problema.

Estado, estado inicial, e estado final

Do ponto de vista do sistema que vai resolver um dado problema (e.g., um *robot* ou um programa de computador), a resolução consiste de uma sequência de operações que causam modificações sucessivas ao estado inicial, enfrentado pelo sistema, conduzindo a um estado final ou objetivo. Relativamente ao problema representado na Figura 1, uma resolução possível é a seguinte sequência de ações: mover o espaço para baixo, mover o espaço para a direita, e tornar a mover o espaço para a direita.



Ao estado inicialmente enfrentado pelo sistema chama-se *Estado Inicial* (ou *Estado de Partida*). Em geral, poderá haver mais que um estado que o sistema tenta atingir. A cada estado desejado chama-se *Estado Final* (ou *Estado Objetivo*). Na solução apresentada, a configuração do mundo (naquilo que é relevante para o problema considerado) atravessa quatro estados ao todo: os estados inicial e final, e dois estados intermédios. Mas há muito mais estados do que os pertencentes à solução apresentada.

Espaço de estados

O *Espaço de Estados* ou *Espaço de Procura* de um problema é o conjunto de todos os estados que representam configurações do mundo relativamente aos aspetos relevantes para o problema.

Como o espaço de procura se pode representar como se fosse um grafo, muitas vezes, em vez de falarmos de estados e de ações que convertem estados noutros, falamos de nós e de arcos de um grafo. Cada nó está associado a um estado, e cada arco representa uma ação.

Quanto maior for o *Espaço de Estados*, mais serão em média os recursos (tempo e memória) que um algoritmo de procura consumirá para resolver um dado problema. No caso do *8-puzzle*, um estado é representado por uma lista de nove números, portanto existe um número de estados igual ao número de permutações dos nove números que representam um estado: [0, 1, 2, 3, 4, 5, 6, 7, 8], [0, 1, 2, 3, 4, 5, 6, 8, 7], [0, 1, 2, 3, 4, 5, 7, 6, 8], [0, 1, 2, 3, 4, 5, 7, 8, 6]…., isto é, 362.880 = 9! estados. O *8-puzzle* é um problema muito pequeno, quando comparado com os problemas da vida real, mas mesmo assim, tem um *Espaço de Procura* com mais de 362 mil estados. Se considerarmos o *15-puzzle*, que é a versão habitual do problema, teremos 16! = 20.922.789.888.000 estados (mais do que 20 milhões de milhões de estados). É

fácil de compreender que os algoritmos que procuram soluções, às cegas, não poderão constituir uma abordagem para problemas da vida real.

Operadores

As ações que se podem aplicar aos estados, em geral, e que conduzem à sua alteração designam-se frequentemente por *Operadores* ou *Operações*. As operações do *8-puzzle* são as seguintes:

- Mover o espaço em branco para baixo;
- Mover o espaço em branco para cima;
- Mover o espaço em branco para a direita; e
- Mover o espaço em branco para a esquerda.

Cada domínio de problemas tem as suas operações.

Para utilizar um algoritmo de procura é necessário dar-lhe os programas que aplicam as operações do domínio aos estados. Concretamente, tem de se dar um programa que, dado um determinado estado, produz o conjunto dos chamados estados sucessores do estado considerado. Os estados sucessores de um estado são aqueles estados que se obtêm do estado considerado pela aplicação de uma operação.

No estado inicial do exemplo que tem servido de ilustração a estes apontamentos, podem aplicar-se três operações:

- Mover o espaço em branco para baixo;
- Mover o espaço em branco para cima; e
- Mover o espaço em branco para a direita.

Nesse estado, não é possível mover o espaço para a esquerda. Isto significa que o estado inicial deste problema ilustrativo tem três sucessores.

Solução

O algoritmo de procura vai aplicando operações até que eventualmente encontra um estado final. Nessa altura, a solução encontrada para o problema é a sequência de operações aplicadas pelo algoritmo desde o estado inicial até ao estado final. É possível que haja diversas soluções para o mesmo problema, quer porque pode haver mais do que um estado final, quer porque, pode haver mais do que um caminho entre o estado inicial e aquele estado final.

Duração da procura e comprimento da solução

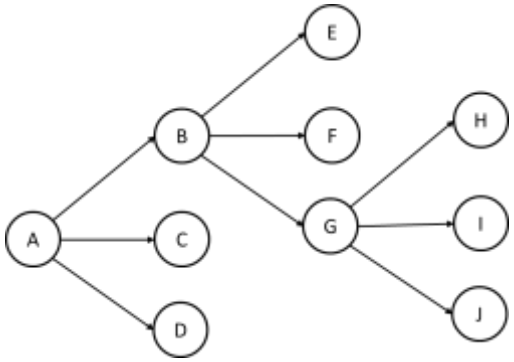
Quando um sistema usa um algoritmo de procura para resolver um problema, há que considerar o tempo e memória gastos na procura da solução, e o tempo e memória gastos para aplicar a solução descoberta pelo algoritmo. Pode acontecer que o algoritmo de procura consuma muitos recursos para encontrar uma solução, mas que a solução encontrada seja uma solução ótima, no sentido em que não há outra solução melhor que ela. Pode também acontecer que o algoritmo de procura consuma poucos recursos na procura da solução, mas que a solução encontrada não seja assim muito boa. Em aplicações reais, há que ponderar os recursos consumidos pelos dois processos: a procura da solução e a aplicação dessa solução. Pode não valer de nada encontrar a melhor solução para um problema, se a fase da procura consumir um tempo inaceitável.

Noutros problemas, o tempo de procura pode não ser importante, desde que se encontre uma solução eficiente. Tudo depende do problema.

Geração e expansão dos estados

[Talvez fosse melhor apresentar estes conceitos depois da apresentação dos algoritmos de procura]

Na operação de um algoritmo de procura, o algoritmo começa por aplicar operações do domínio do problema ao estado inicial, gerando um conjunto de estados sucessores do inicial. Depois, escolhe um estado sucessor e volta a aplicar todas as operações possíveis a esse estado. O processo repete-se até que um estado final seja encontrado, ou até que já não seja possível aplicar mais operações. Ao processo de aplicação das operações possíveis a um dado estado, originando o conjunto dos seus sucessores, chama-se *Expansão do Estado*. Do surgimento de um determinado estado devido a um processo de expansão diz-se que o estado foi gerado. Não é obrigatório que os estados sejam escolhidos para serem expandidos pela mesma ordem pela qual foram gerados.



Podemos imaginar um processo de procura em que o estado A representa o estado inicial. O A é o primeiro nó a ser expandido e também o primeiro a ser gerado (foi dado inicialmente ao algoritmo). A expansão do estado A origina os seus sucessores B, C e D, os quais podem ter sido gerados por esta ordem. O algoritmo de procura escolhe um destes estados para expandir, por exemplo o estado B, originando os seus sucessores E, F e G, por esta ordem. Nesta altura, o algoritmo pode escolher um de entre vários nós para expandir. Poderia escolher, por exemplo um dos estados C e D, ou um dos estados E, F e G. No nosso exemplo hipotético, o algoritmo escolheu expandir o estado G, gerando assim os estados H, I e J. Se os estados fossem expandidos pela mesma ordem em que são gerados, o algoritmo teria escolhido expandir o estado C em vez do G.

Os algoritmos de procura distinguem-se entre si pela ordem pela qual escolhem os nós para serem expandidos. O algoritmo de procura em largura primeiro expande os estados por níveis de profundidade: nunca expande um estado de um nível mais profundo, se existirem estados menos profundos por expandir. Isto é, o algoritmo escolheria expandir os nós C e D antes de qualquer dos estados E, F e G.

O algoritmo de procura em profundidade primeiro escolhe expandir sempre um dos estados mais profundos. Isto é, expande um dos estados E, F e G antes de expandir qualquer dos estados C e D. Ao escolher, por exemplo, o estado G, originando estados mais profundos na árvore de procura, no próximo passo, expandiria um destes (H, I e J).

A procura do melhor primeiro não escolhe um estado para expandir, com base na sua profundidade. Em vez disso, usa uma função que estima quão promissores são os

estados e expande o mais promissor. A essa função chama-se função heurística, função de avaliação, ou função de custo.

Independência do domínio do problema

Num certo sentido, os algoritmos de procura podem ser definidos de forma totalmente independente do domínio dos problemas que têm de resolver. Tendo algoritmos independentes do domínio do problema é uma grande vantagem porque o mesmo algoritmo pode ser usado para resolver, idealmente, qualquer problema.

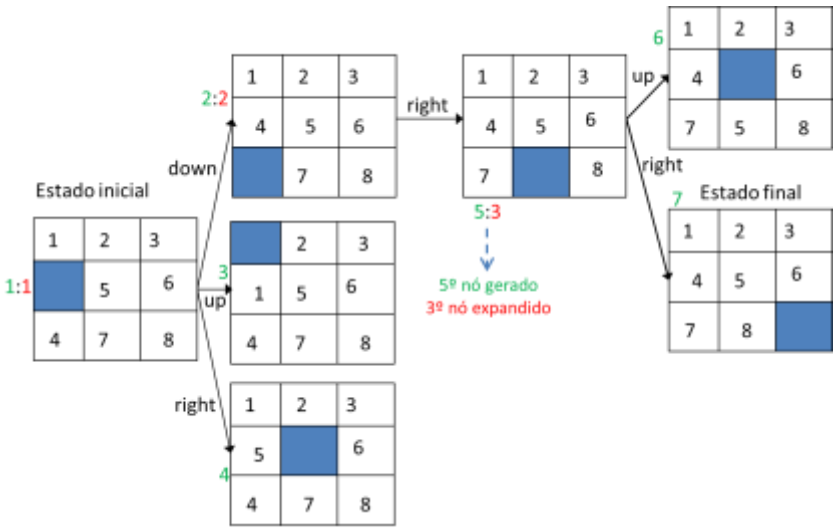
No entanto, na expansão de um estado, os algoritmos de procura têm de lhe aplicar as operações do domínio. Além disso, têm de perceber (i.e., poder determinar) que um dado estado é um estado final. Posto isto, para manter a independência entre o algoritmo e o domínio do problema, os algoritmos têm de receber pelo menos um programa capaz de receber um estado e expandi-lo, originando o conjunto dos seus sucessores; e um programa capaz de identificar estados finais.

Os algoritmos de procura do melhor primeiro têm ainda que receber um programa capaz de avaliar cada estado, determinando quão promissor ele é. Isto é, tem de receber um programa que implemente a função heurística.

Estes três programas encapsulam todo o conhecimento do domínio do problema, permitindo que o resto do algoritmo continue totalmente independente do domínio de aplicação.

2 Procura em profundidade primeiro, e em largura primeiro

A figura que se segue ilustra a aplicação do algoritmo de procura em profundidade primeiro à resolução do problema descrito na Figura 1.



Na figura, cada estado está associado a um número ou a um par de números com o formato i:j. O número associado a um estado ou o primeiro componente do par i:j representa a ordem pela qual o estado foi gerado. Por exemplo, o estado inicial foi o primeiro a ser gerado, por isso o primeiro componente do par 1:1 associado é o 1. Quando os estados estão associados a pares i:j, o j representa a ordem pela qual o estado

foi expandido. Por exemplo, o estado inicial foi o primeiro a ser expandido, daí que o segundo componente do par 1:1 a ele associado tenha o valor 1. Se um estado estiver associado a um número simples, e não a um par i:j, isso significa que esse estado não foi ainda expandido.

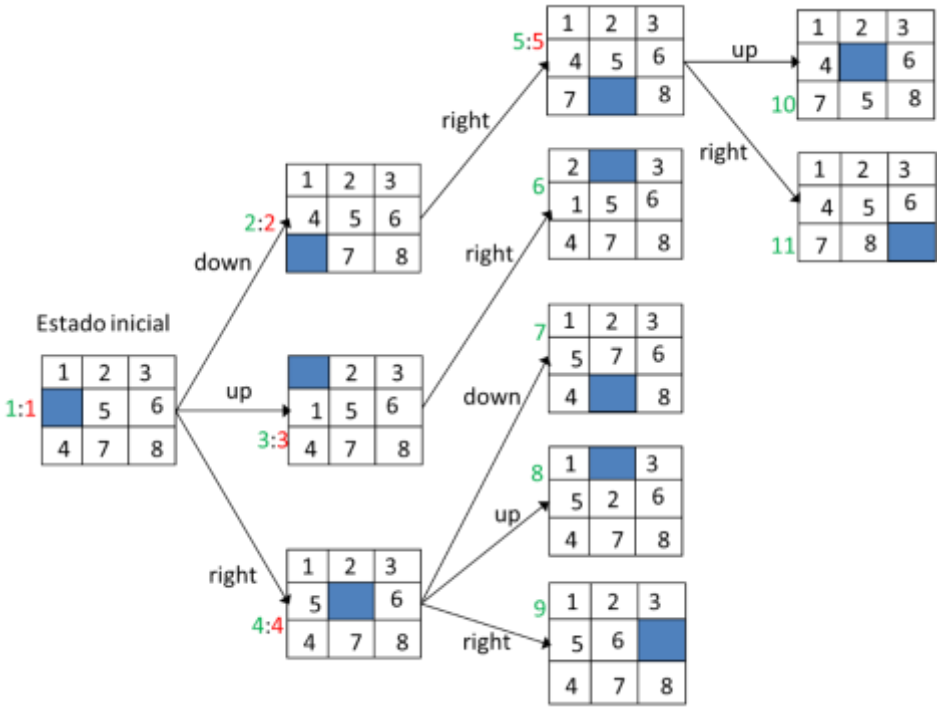
O algoritmo de procura em profundidade-primeiro expande o estado inicial originando os estados 2, 3 e 4, os quais correspondem a mover o espaço para baixo, para cima e para a direita, respetivamente. Neste estado inicial, é impossível mover o espaço para a esquerda.

Seguidamente, o algoritmo escolhe expandir um dos estados não expandidos mais profundos. Nesta altura, todos os estados não expandidos (2, 3 e 4) têm a mesma profundidade. O algoritmo escolhe arbitrariamente expandir o estado 2, originando o único sucessor 5, o qual corresponde a mover o espaço em branco para a direita. A expansão do estado 2 não origina mais sucessores a considerar porque *(i)* neste estado, não se pode mover o espaço nem para a esquerda nem para baixo; e porque *(ii)* o estado que seria originado movendo o espaço para cima é um estado já existente. Nos algoritmos de procura em profundidade primeiro e em largura primeiro, os estados repetidos não são usados para nada. Ou nem chegam a ser gerados ou são descartados.

Nesta altura, os estados não expandidos são 3, 4 e 5, mas o mais profundo deles é o estado 5. O algoritmo expande o estado 5 (o terceiro a ser expandido) originando os sucessores 6 e 7. Mais uma vez, se o espaço em branco fosse movido para a esquerda, seria originado um estado repetido (igual ao estado 2). Sendo repetido, não é considerado.

Nesta altura, os estados ainda não expandidos são 3, 4, 6 e 7, dos quais, os mais profundos são 6 e 7. Havendo um empate (os estados 6 e 7 têm a mesma profundidade), o algoritmo *(i)* escolhe o objetivo, se um deles for o objetivo, ou então *(ii)* escolhe arbitrariamente. Como um dos estados empatados é o estado objetivo (estado 7), o algoritmo escolhe este estado e o processo termina. Há outras implementações do algoritmo de procura em profundidade primeiro que terminam a sua operação assim que um nó objetivo for gerado. Nesse caso, após a expansão do nó 5, a procura terminaria porque um dos seus sucessores é um objetivo.

A figura que se segue mostra a operação do algoritmo de procura em largura primeiro, no mesmo problema.



O algoritmo de procura em largura primeiro começa por expandir o estado 1, isto é, o estado inicial. A expansão de 1 originaria os sucessores 2, 3 e 4. O algoritmo de procura em largura primeiro escolhe sempre expandir um dos nós não expandidos menos profundos. Os únicos nós não expandidos são 2, 3 e 4, os quais têm a mesma profundidade. Como nenhum deles é o objetivo, o algoritmo escolhe arbitrariamente expandir o nó 2, originando o nó 5. Só foi gerado um nó porque, no estado 2, não se pode mover o espaço nem para a esquerda nem para baixo, e porque o estado que se obteria, movendo o espaço para cima, seria igual a um estado já existente.

Nesta altura, os nós não expandidos são 3, 4 e 5. Destes, 3 e 4 são os de menor profundidade. O algoritmo escolhe arbitrariamente expandir o nó 3, originando o nó 6.

Nesta altura, os estados não expandidos são 4, 5 e 6. O algoritmo escolhe expandir o 4 porque é o de menor profundidade. A expansão de 4 origina os sucessores 7, 8 e 9.

Nesta altura, os estados não expandidos são 5, 6, 7, 8 e 9, os quais têm todos a mesma profundidade. O algoritmo escolheu arbitrariamente expandir o nó 5, originando os sucessores 10 e 11.

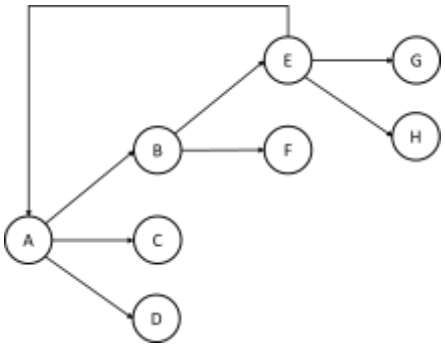
Nesta altura, os nós não expandidos são 6, 7, 8, 9, 10 e 11, dos quais, 6, 7, 8 e 9 são os de menor profundidade. Apesar do nó 11 ser o objetivo, o algoritmo continua a sua operação, escolhendo para expandir um dos nós com menor profundidade (6, 7, 8 e 9). Só depois de expandir todos estes nós é que o algoritmo passa para os nós de profundidade seguinte. Nesse nível é que ele encontra o nó objetivo. Há outras implementações do algoritmo de procura em largura primeiro que terminam a sua operação assim que um nó objetivo for gerado. Nesse caso, após a expansão do nó 5, a procura terminaria porque um dos seus sucessores é um objetivo.

Tanto na procura em profundidade primeiro como na procura em largura primeiro descritas aqui, o processo termina apenas quando o nó escolhido para expandir representar um estado final, ou quando não existirem nós não expandidos.

Apenas o algoritmo de procura em largura primeiro garante que se encontra uma solução (se existir), e que ela é a mais curta. No entanto, os recursos (tempo de

processamento e memória) consumidos pelo algoritmo são, em geral, muito maiores do que os consumidos pelo algoritmo de procura em profundidade primeiro.

Espaços de procura com ciclos: Quando procura um espaço de estados cíclico, isto é, um espaço em que os estados estão organizados como um grafo com caminhos cíclicos, o algoritmo de procura em profundidade primeiro poderá ficar "preso" indefinidamente num desses ciclos, se não tiver determinados cuidados. Consideremos o exemplo que se segue.



Supondo que a procura começa no estado correspondente ao nó A, o algoritmo expande o A, originando os sucessores B, C e D, por esta ordem. Supomos agora que o algoritmo escolhe expandir o nó B, originando os sucessores E e F, por esta ordem. Nesta altura, o algoritmo escolhe expandir o nó E, originando os sucessores A, G e H, por esta ordem. Usando o mesmo raciocínio, o algoritmo irá agora entrar em ciclo, pois voltará a gerar B, C e D, por esta ordem, o que, como já vimos, irá conduzir a geração do nó A. Neste caso e em casos como este, o algoritmo não será capaz de sair do ciclo.

No entanto, o algoritmo de procura em profundidade primeiro pode ser modificado para evitar ficar preso indefinidamente em procuras cíclicas. Para isso, basta memorizar todos os nós já expandidos (ou todos os arcos já percorridos). Nesse caso, quando o algoritmo expande o nó E do exemplo, não gera de novo o nó A (porque já expandiu o nó A anteriormente). Em vez disso, serão gerados apenas os nós G e H, por esta ordem. Agora, o algoritmo irá expandir o nó G (e não o nó A). Infelizmente, quando o espaço de procura é muito grande (ou mesmo infinito), não é possível memorizar todos os nós já expandidos, por limitações de memória. Isto significa que, em espaços de procura muito grandes e com caminhos cíclicos entre os nós, o algoritmo de procura em profundidade primeiro pode ficar refém de processos cíclicos de procura.

Retrocesso na procura em profundidade primeiro: imaginemos, para exemplo, que não é possível expandir o nó G porque não tem sucessores (ou porque só tem sucessores já expandidos). Nesse caso, o algoritmo de procura retrocede para E e vai tentar o nó H. Se o nó H também não tiver sucessores, o algoritmo volta a retroceder, desta vez para o nó B, a partir do qual poderá explorar a via alternativa F, e assim por diante. Repare-se que, quando retrocede para um nó do nível anterior (por exemplo o B), o algoritmo explora um dos nós já gerados (um dos sucessores de B), neste caso, F. No retrocesso não houve nova expansão do nó. Só agora, ao explorar o nó F (já existente) é que o algoritmo o procuraria expandir.

Ao processo de retrocesso chama-se, mais vulgarmente *backtracking*. O algoritmo de procura em largura primeiro não faz *backtracking* porque todos os nós de um nível são explorados antes que algum dos nós do nível seguinte possa ser explorado. Por isso, não há para onde retroceder.

Limitações: a maioria dos problemas reais têm espaços de estados muitíssimo grandes e, em alguns casos, infinitos. Os recursos consumidos por um algoritmo de procura para encontrar uma solução (tempo e memória) dependem do tamanho do espaço do problema.

Conclusão: é preciso que o algoritmo não procure todo o espaço; é preciso que ele se encaminhe por uma trajetória que conduza mais rapidamente ao objetivo. No entanto, para que o algoritmo saiba qual dos caminhos é mais promissor, necessita usar conhecimento específico sobre o problema que está a resolver. Com esse conhecimento, sempre que tem uma escolha a fazer entre vários nós, pode avaliá-los e determinar aquele que parece mais promissor. Desta maneira surge a noção de algoritmo de procura do melhor primeiro (*best-first search*), o qual usa uma função heurística para determinar o valor de cada nó do espaço de procura (quão promissor é o nó).

Na próxima secção, será apresentado o algoritmo A* que se baseia no princípio de avaliar os nós já conhecidos do espaço de procura e expandir o melhor deles.

3 Algoritmo A* (A-Star)

O algoritmo A* é uma das mais conhecidas e mais bem-sucedidas concretizações do conceito de procura do melhor primeiro (*Best-First Search*).

Duas implementações simples do algoritmo de procura do melhor primeiro:

> https://en.wikipedia.org/wiki/Best-first_search
>
> http://courses.cs.washington.edu/courses/cse326/03su/homework/hw3/bestfirstsearch.html

A principal diferença entre a procura do melhor primeiro e a procura cega de força bruta é a utilização de uma função de avaliação dos estados, a qual permite ao algoritmo escolher para expandir, em cada passo, o nó mais promissor. A função de avaliação do nó *n* usada no algoritmo A* (também designada função de custo) tem o seguinte formato geral: f(n) = g(n) + h(n), em que g(n) é o custo do caminho desde o estado inicial ao estado representado no nó *n*; e h(n) é a estimativa do custo do melhor caminho que liga o nó *n* ao estado final.

O que diferencia o A* de outras concretizações de procura do melhor primeiro é justamente a utilização da função g(n) na avaliação de cada nó *n*.

Prova-se que o A* termina sempre, mesmo que não haja solução para um problema, e não fica preso em ciclos infinitos. Prova-se também que, se a estimativa h(n) for sempre menor do que o custo real mínimo do caminho entre o nó *n* e um nó objetivo, então o A* encontra a melhor solução para o problema, se ela existir.

Se h(n)=0 para qualquer nó *n*, o algoritmo A* dá os mesmos resultados que o algoritmo de caminho ótimo de Dijkstra.

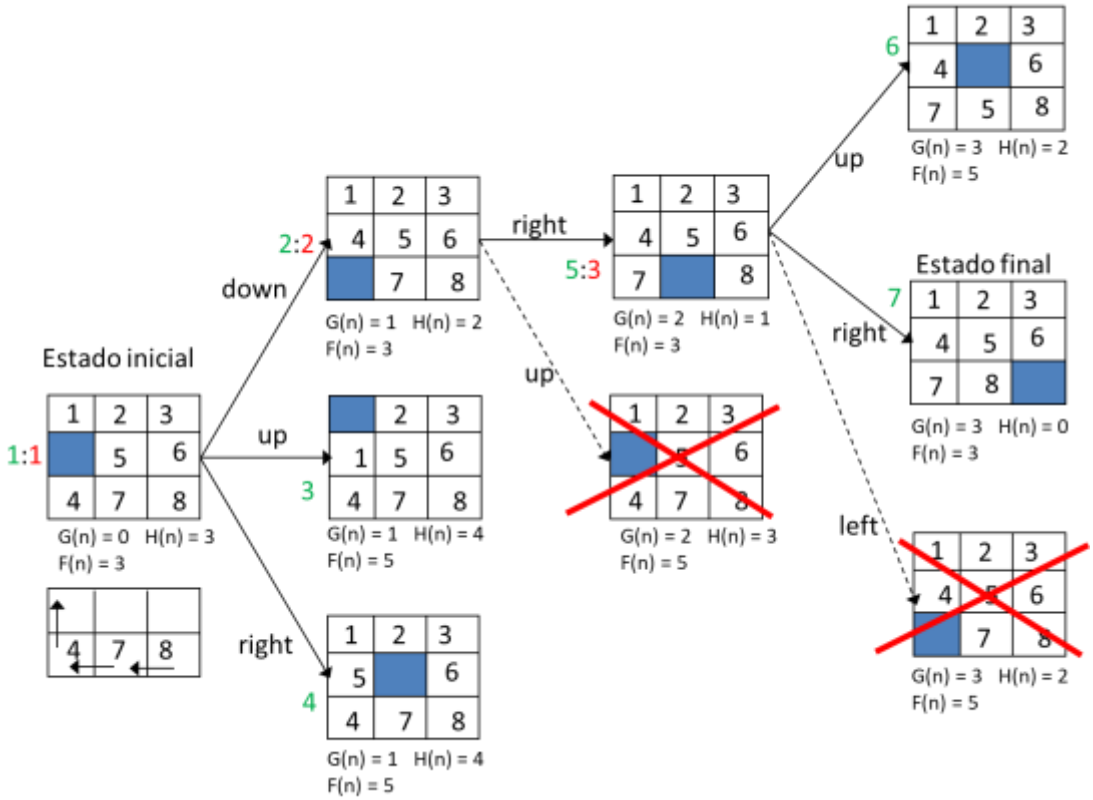
Links para descrições do A*:

- (Wikipedia): https://en.wikipedia.org/wiki/A*_search_algorithm
- (Universidade de Stanford): http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html
- (MIT): http://web.mit.edu/eranki/www/tutorials/search/

A figura que se segue ilustra a aplicação do algoritmo A* para o caso particular do *8-puzzle*. Para isso, é necessário definir as funções g(n) e h(n) usadas no exemplo:

g(0) = 0, g(n) = g(parent(n))+1; e

h(n) = *Manhattan Distance* (soma das distâncias das peças à sua posição final)



3.1 Exemplo de avaliação dos estados

Para calcular a estimativa do custo do nó inicial (i.e., a estimativa da sua "proximidade" ao nó que representa o estado final), usou-se uma heurística conhecida por distância de *Manhattan*. Nesta heurística, o custo de um nó é a soma das distâncias entre a posição de cada uma das suas peças e a posição que ela teria no estado final. A peça 1, no estado inicial está na posição que ela deve ter no estado final, por isso a contribuição da peça 1 para a estimativa do custo do nó é nula. O mesmo acontece com as peças 2, 3, 5 e 6, pois ocupam todas as mesmas posições que no estado final.

A peça 4, no estado inicial, não está na posição desejada: está à distância de uma posição da sua posição final. Assim, a peça 4 contribui com 1 para o custo do nó inicial. A peça 7, no estado inicial, também se encontra à distância de 1 da sua posição final. O mesmo se passa com a peça 8. Para evitar contabilizar a mesma informação duas vezes, a heurística não contabiliza a diferença entre a posição atual do espaço em branco e a sua posição no estado final porque esta diferença é uma consequência das diferenças de posição das outras peças. A estimativa do custo total do nó inicial, considerando esta heurística, é a soma das contribuições das peças 4, 7 e 8, ou seja h(1) = 1+1+1=3.

Existem outras heurísticas que podem ser usadas com este problema. Link para heurísticas para o *8-puzzle*: http://ethesis.nitrkl.ac.in/5575/1/110CS0081-1.pdf

3.2 Funcionamento do A*

O A* começa por expandir e avaliar o nó 1, que representa o estado inicial. A sua expansão origina os nós 2, 3 e 4. O valor de g(1) é 0 pois trata-se do primeiro nó. Os valores de g para todos os nós sucessores de 1 é 1, admitindo que o custo de qualquer das operações aplicadas aos nós é 1. Como vimos (secção 3.1), a estimativa do custo do nó 1 é h(1)=3. As estimativas dos custos de 2, 3 e 4 são as seguintes. No nó 2, apenas os ladrilhos 7 e 8 estão fora do sítio, a uma distância de 1 da sua posição no estado final: h(2)=2. No nó 3, os ladrilhos 1, 4, 7 e 8 estão todas à distância de 1 da sua posição no estado final: h(3)=4. No nó 4, os ladrilhos 4, 5, 7 e 8 estão também à distância 1 da sua posição no estado final: h(4)=4. As avaliações têm os seguintes valores:
f(1) = g(1)+h(1) = 0+3 = 3, f(2) = g(2)+h(2) = 1+2 = 3, f(3) = g(3)+h(3) = 1+4 = 5, e f(4) = g(4)+h(4) = 1+4 = 5.

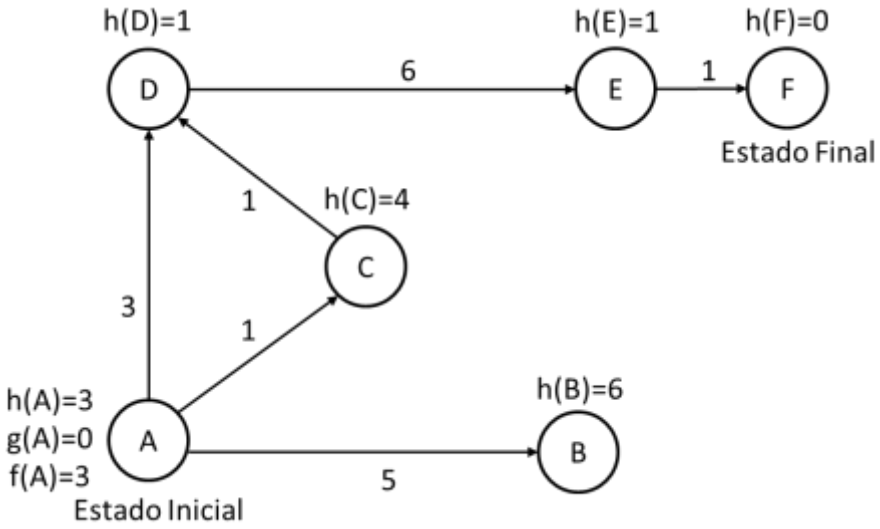
Dos nós não expandidos, o que tem a melhor avaliação é o nó 2, com f(2)=3. O A* expande o nó 2, originando o sucessor 5, cujo valor de g é 2. No nó 5, apenas o ladrilho 8 está fora da sua posição final, a uma distância de 1: f(5) = g(5)+h(5) = 2+1 = 3. Se, no nó 2, o espaço fosse movido para cima, seria gerado um outro nó associado a um estado igual ao do nó 1. Apesar da estimativa do valor deste potencial novo nó ser igual à do nó 1 (porque se trata do mesmo estado), o custo do nó 1 é 0, enquanto que o custo deste nó potencial seria 2. Como a avaliação do nó potencial (5) é pior que a do nó 1 (3), já expandido, o nó potencial é descartado.

Nesta altura, dos nós não expandidos (3, 4 e 5), o que tem a melhor avaliação é o 5, com f(5)=3. O algoritmo expande o nó 5, originando os nós 6 e 7, com as seguintes avaliações f(6) = g(6)+h(6) = 3+2 = 5, e f(7) = g(7)+h(7) = 3+0 = 3. Como o resultado de mover o espaço para a esquerda originaria um nó associado ao mesmo estado que o nó 2, mas com pior avaliação, esse nó não é considerado.

Nesta altura, como o nó não expandido com melhor avaliação é o nó 7, com f(7)=3, o A* escolhe o nó 7 para expansão. Mas, como o 7 é o nó final, a procura termina.

O exemplo apresentado não é suficiente para descrever totalmente o funcionamento do A* porque há aspetos importantes do algoritmo que têm de ser considerados e explicados. A avaliação de um nó depende do caminho percorrido até lá chegar. Por exemplo, os nós 2 e o nó potencialmente descendente do nó 5 resultante do movimento do espaço para a esquerda (chamemos-lhe nó *p*) do exemplo anterior, embora representem o mesmo estado, têm avaliações diferentes: f(2)=3 e f(p)=5. Por outro lado, se um nó gerado *m* tiver descendentes, a avaliação desses descendentes também depende do caminho percorrido desde o nó inicial até *m*.

Consideremos o caso abstrato que se segue, em que os custos das operações estão indicadas junto aos arcos entre nós, e estimativas dos custos dos nós estão indicadas junto a eles.



O A* começa por expandir o estado inicial (nó A), originando os sucessores B, C e D, cujas avaliações são f(B) = g(B)+h(B) = 5+6 = 11, f(C) = g(C)+h(C) = 1+4 = 5, e f(D) = g(D)+h(D) = 3+1 = 4.

Nesta altura, o melhor dos nós não expandidos (B, C e D) é o nó D, com avaliação f(D)=4. O A* expande o D, originando o sucessor E, cuja avaliação é f(E) = g(E)+h(E) = 9+1 = 10.

Agora, dos nós não expandidos (B, C, E), o melhor é o nó C, com f(C)=5. O algoritmo expande o nó C, originando o sucessor D, o qual já havia sido gerado e já tinha uma avaliação prévia de 4. Contudo, pela via atual, a avaliação de D é diferente, g(D)+h(D) = 2+1 = 3. Como há uma via melhor para chegar do nó inicial ao nó D do que a via previamente descoberta, e como estamos interessados na solução mais barata, o algoritmo tem de considerar a melhor das vias para chegar ao nó. Mas alterar a avaliação de D não chega. O A* tem de memorizar que esta avaliação só se obtém, considerando a via em que C é o nó predecessor de D. Além disso, o nó D já havia sido expandido antes, tendo originado o sucessor E com avaliação 10. Porém, pela nova via, a avaliação de E será menor, f(E) = g(E)+h(E) = 8+1 = 9.

Dos nós não expandidos (B e E), E (via C) é o nó com a melhor avaliação, f(E)=9. O algoritmo expande o nó E, originando o sucessor F, cuja avaliação (via C) é f(F) = g(F)+h(F) = 9+0 = 9.

Como o melhor nó é F, o A* escolhe F para expandir. No entanto, como F é um nó objetivo, a procura termina, sendo devolvida a solução [(A, C), (C, D), (D, E), (E, F)], em que usamos a notação (N_1, N_2) para nos referirmos à operação que transforma o estado representado pelo nó N_1 no estado representado pelo nó N_2.

Haveria uma outra solução para chegar do estado inicial (A) ao estado final (F), mas seria uma solução mais dispendiosa, apesar de integrar um menor número de operações.

3.3 Definição do A*

O algoritmo A* usa duas estruturas de dados: a lista dos nós ainda não expandidos (OPEN, mais raramente chamada *"fringe"* – franja – na literatura anglo-saxónica), e a lista dos nós já expandidos (CLOSED).

[Adaptado do artigo original (Hart, Nilsson e Raphael 1968) e da descrição existente no sítio web do MIT (http://web.mit.edu/eranki/www/tutorials/search/)]

1 – Inicializa-se a lista OPEN com o nó inicial, *s* e com a sua avaliação. Inicializa-se a lista CLOSED com a lista vazia.

2 – Se OPEN estiver vazio, terminar com insucesso. Caso contrário, selecionar o nó *n* de OPEN cuja avaliação f(n) é a menor. Os empates resolvem-se arbitrariamente, mas favorecendo sempre os nós objetivo. Remove-se *n* de OPEN e insere-se em CLOSED.

3 – Se *n* for um nó objetivo, termina-se com sucesso. Devolve-se o caminho entre *s* (o nó inicial) e *n*.

4 – Expande-se o nó *n*, obtendo os seus sucessores. Qualquer sucessor, *s*, de *n*, juntamente com a sua avaliação, é acrescentado à lista OPEN apenas se ambas as seguintes condições forem verdadeiras:

(a) não existir um nó na lista OPEN, correspondente ao mesmo estado que *s*, com uma avaliação, f, inferior à avaliação de *s*; e se

(b) não existir um nó na lista CLOSED, correspondente ao mesmo estado que *s*, com uma avaliação, f, inferior à avaliação de *s*.

Sempre que um sucessor de *n* for acrescentado à lista OPEN, é necessário registar que o seu nó pai é *n* (ou alterar o registo, se ele já existir)

5 – Voltar ao passo 2.

(Hart, Nilsson e Raphael 1968) Hart, P.; Nilsson, N.; Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. In *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100-107. DOI:10.1109/TSSC.1968.300136