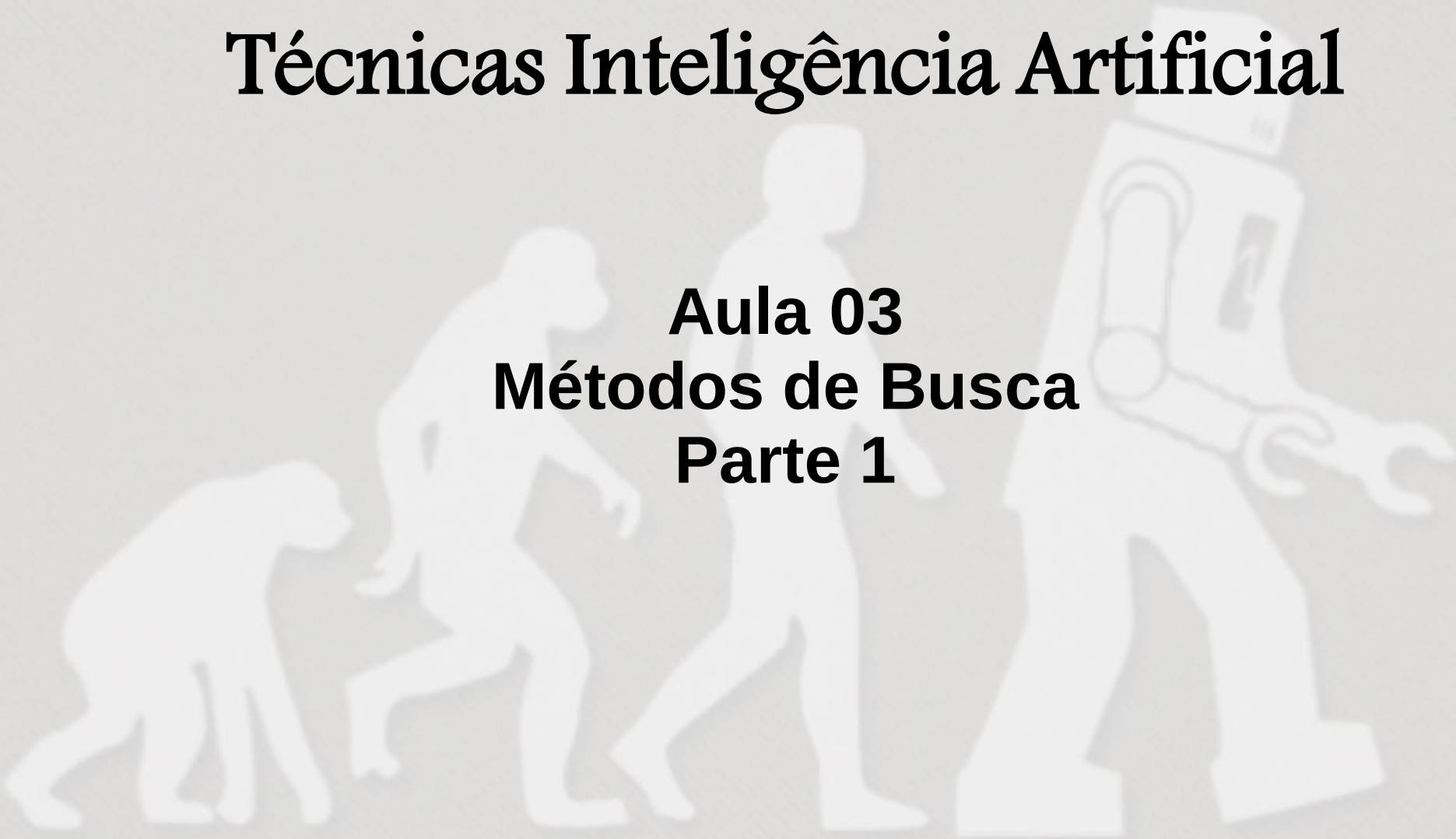
Universidade do Sul de Santa Catarina
Ciência da Computação

Técnicas Inteligência Artificial



Aula 03
Métodos de Busca
Parte 1

Prof. Max Pereira

Solução de Problemas como Busca

Um problema pode ser considerado como consistindo em um **objetivo** e um **conjunto de ações** que podem ser praticadas para alcançar esse objetivo. Em qualquer tempo, consideramos o **estado** do **espaço de busca** para representar aonde chegamos como um resultado das ações aplicadas.

Busca é um método que pode ser utilizado por computadores para examinar o espaço de estados, de modo a encontrar um objetivo. Frequentemente queremos encontrar o objetivo o **mais rápido possível** ou **sem utilizar muitos recursos**.

Gerar e Testar

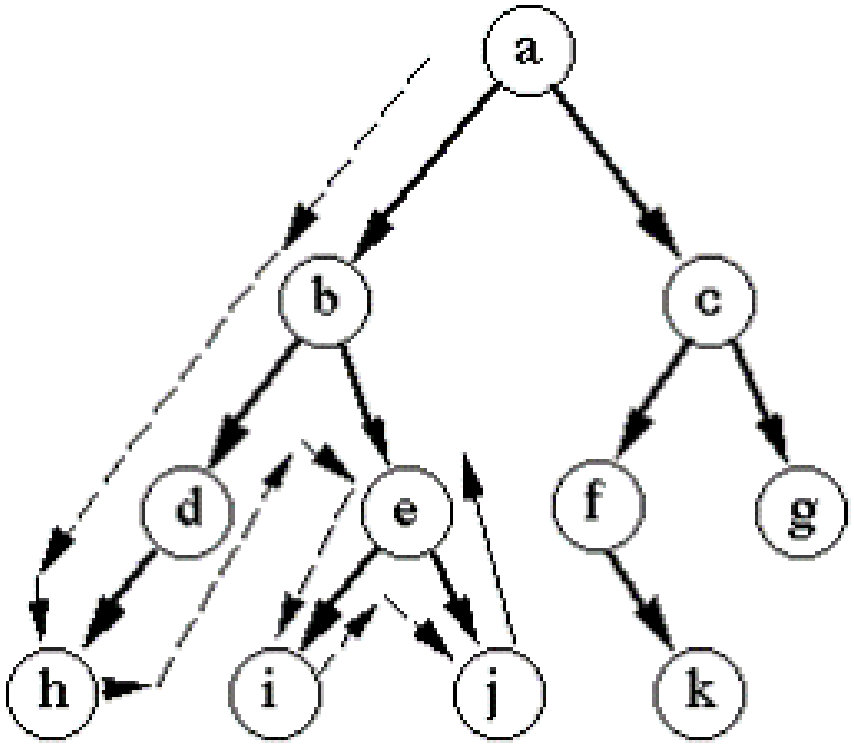
A mais **simples** abordagem de busca é chamada Gerar e Testar. Isto envolve simplesmente gerar cada nó no espaço de busca e testá-lo para verificar se este é um nó objetivo. Essa é a forma mais simples de busca de **força bruta** (busca exaustiva), assim chamada porque ela não pressupõe **conhecimento adicional** além de como percorrer a árvore de busca.

Muitas vezes referido como uma técnica de **busca cega**, devido ao modo pelo qual a árvore de busca é percorrida, sem utilizar qualquer informação sobre o espaço de busca.

Quando não há informação adicional sobre como alcançar a solução

Busca em Profundidade

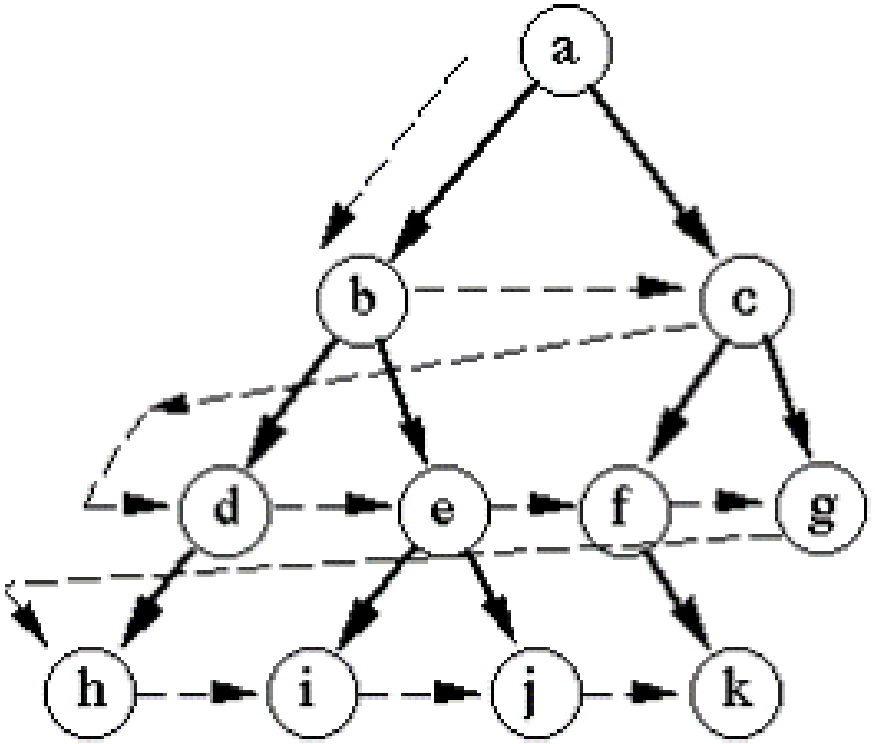
É assim chamada por seguir cada caminho até a sua maior **profundidade** antes de seguir para o próximo caminho. Utiliza um método chamado de **retrocesso cronológico** para voltar na árvore de busca, uma vez que um caminho sem saída seja encontrado.



Busca em profundidade é um exemplo de busca de força bruta ou busca exaustiva

Busca em Largura

Uma alternativa à busca em profundidade. Como o próprio nome sugere, essa abordagem envolve **percorrer a árvore em largura** em vez de em profundidade.



Busca em largura é um exemplo de busca de força bruta ou busca exaustiva

Comparação das buscas em profundidade e em largura

Cenário	Busca em profundidade	Busca em Largura
Alguns caminhos são muito longos ou mesmo infinitos	Funciona mal	Funciona bem
Todos os caminhos têm comprimentos parecidos	Funciona bem	Funciona bem
Todos os caminhos têm comprimentos parecidos e todos os caminhos levam a um estado objetivo	Funciona bem	Desperdício de tempo e memória
Alto fator de ramificação	O desempenho depende de outros fatores	Funciona precariamente

Propriedades dos Métodos de Busca

Há várias propriedades importantes que um método de busca deve ter para ser mais útil.

Em especial, vamos examinar as seguintes:

- ❑ Complexidade
- ❑ Completude
- ❑ Quanto a ser ótimo
- ❑ Admissibilidade
- ❑ Irrevogabilidade

Complexidade

Ao discutir um método de busca, é útil descrever o **quão eficiente** ele é em termos de **tempo e espaço**. A complexidade em termos de tempo está relacionada à **duração de tempo** que o método leva para encontrar um estado objetivo. A complexidade em termos de espaço está relacionada à **quantidade de memória** que o método precisa utilizar.

Um método de busca muito rápido nem sempre encontrará a **melhor solução**, ao passo que, por exemplo, um método de busca que examine cada possível solução garantirá encontrar a melhor solução, mas será **muito ineficiente**.

Completude

Um método de busca é **descrito como completo** se ele **garantir** encontrar um **estado objetivo**, se existir algum. Um método que não seja completo tem a desvantagem de não poder ser necessariamente confiável ao relatar que não há solução.

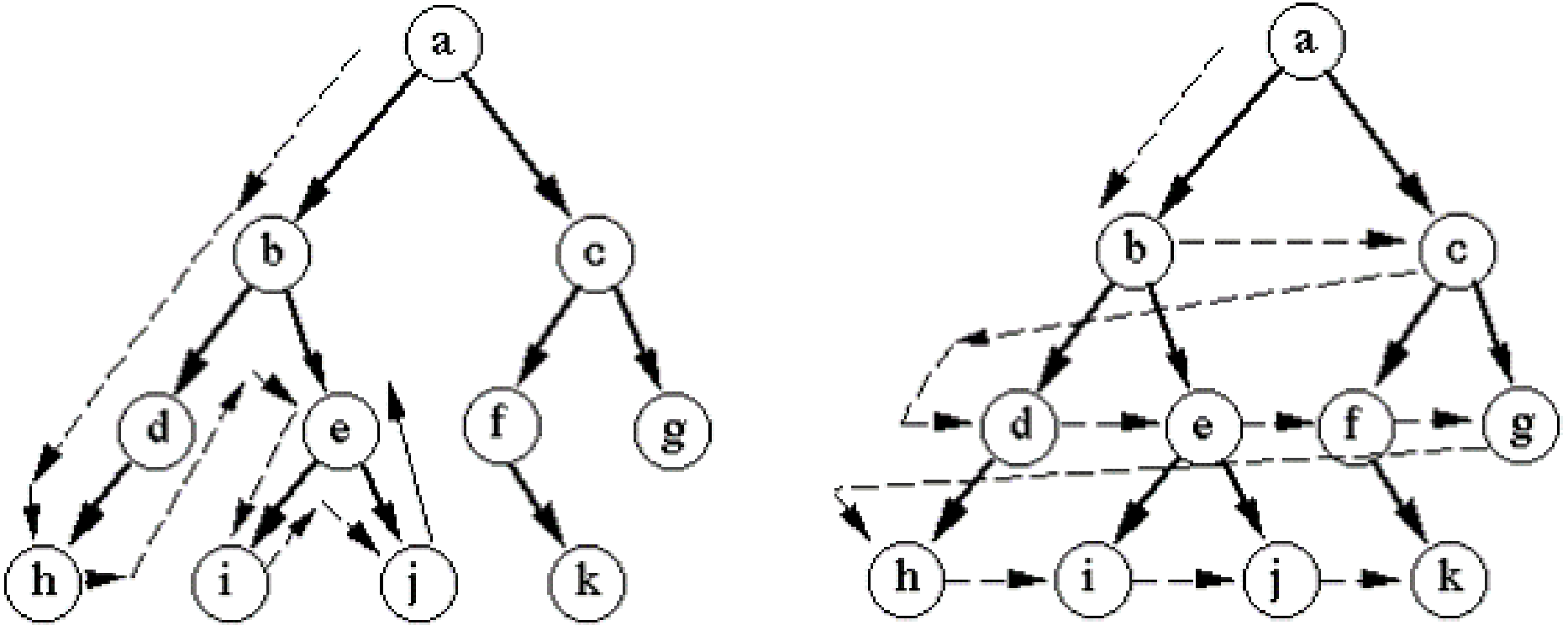
Quanto a ser ótimo

Um método de busca é **ótimo** se ele garantir **achar a melhor solução que exista**. Em outras palavras, ele encontrará o caminho que envolva o **menor número de passos até o estado objetivo.**

Isso não quer dizer que o método de busca propriamente dito seja eficiente – poderia levar muito tempo para um método ótimo de busca identificar a solução ótima.

Em alguns casos, o termo ótimo é utilizado para descrever uma algoritmo que encontre uma solução no **menor tempo possível**, situação na qual o conceito de **admissibilidade** é utilizado no lugar de quanto a ser ótimo. Um algoritmo é então definido como **admissível** se ele garantir encontrar a melhor solução.

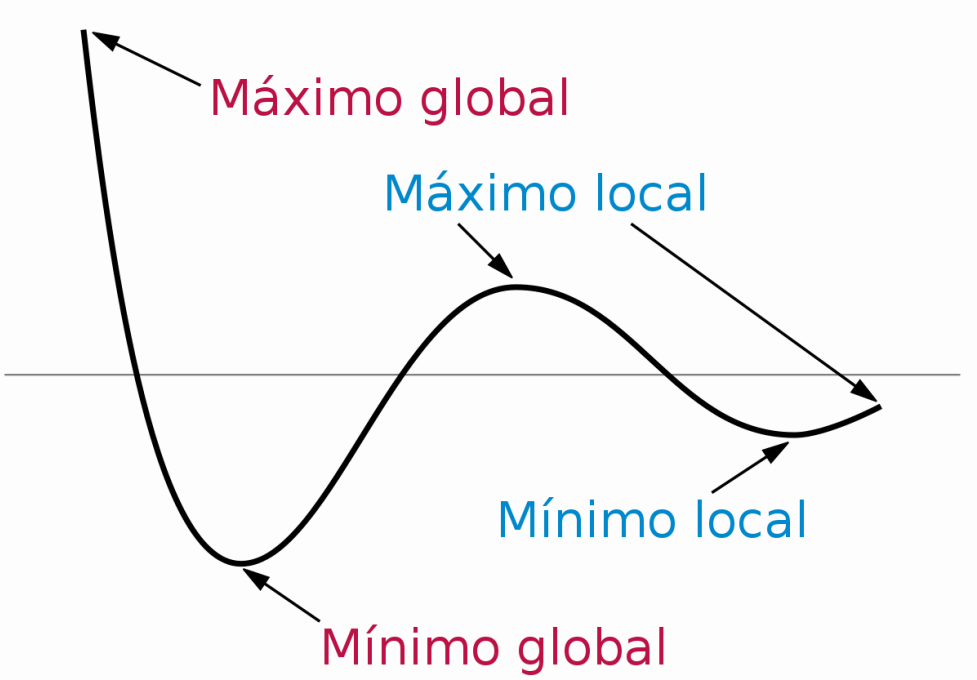
Busca em largura é um método ótimo? Busca em profundidade é um método ótimo?



Irrevogabilidade

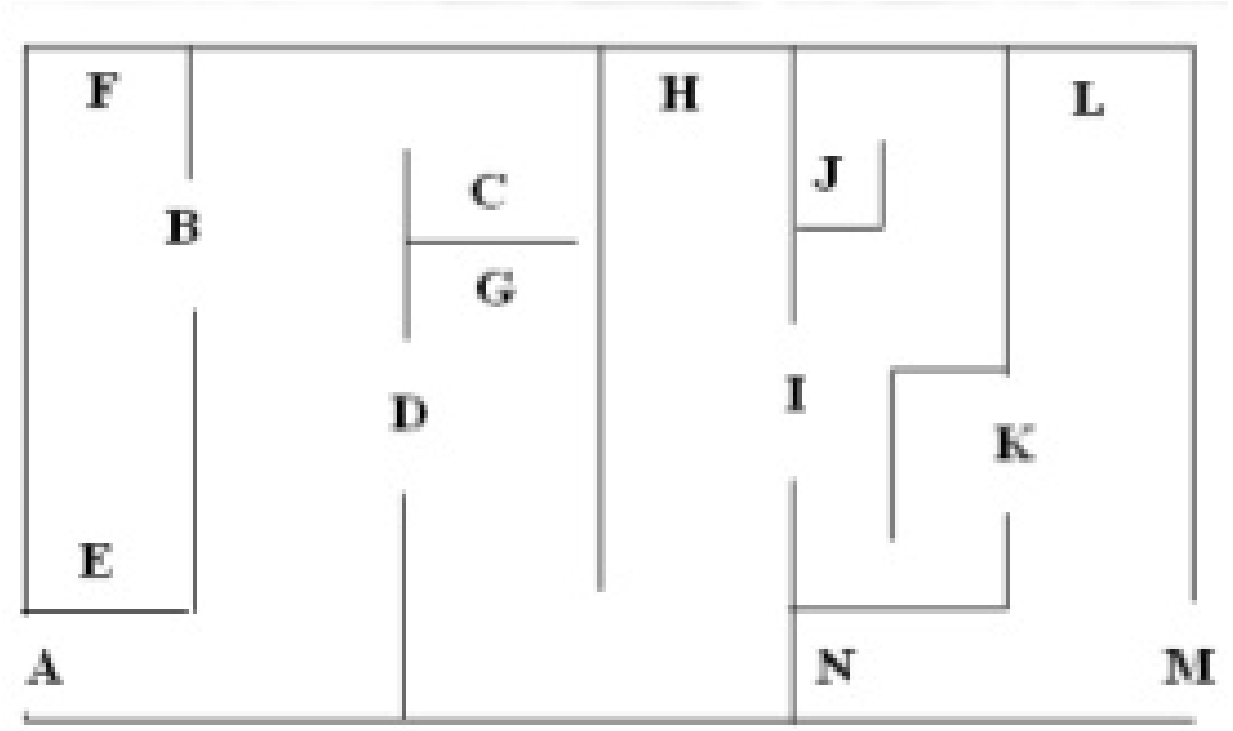
Métodos que **retrocedem** são descritos como uma **tentativa**. Métodos que não retrocedem, e que, portanto, examinam apenas um caminho, são descritos como **irrevogáveis**.

Métodos de busca irrevogáveis frequentemente encontrarão **soluções sub-ótimas**, pois tendem a ser enganados por **ótimos locais** – soluções que parecem localmente boas, porém, são menos favoráveis quando comparadas com outras soluções em outros lugares do espaço de busca.

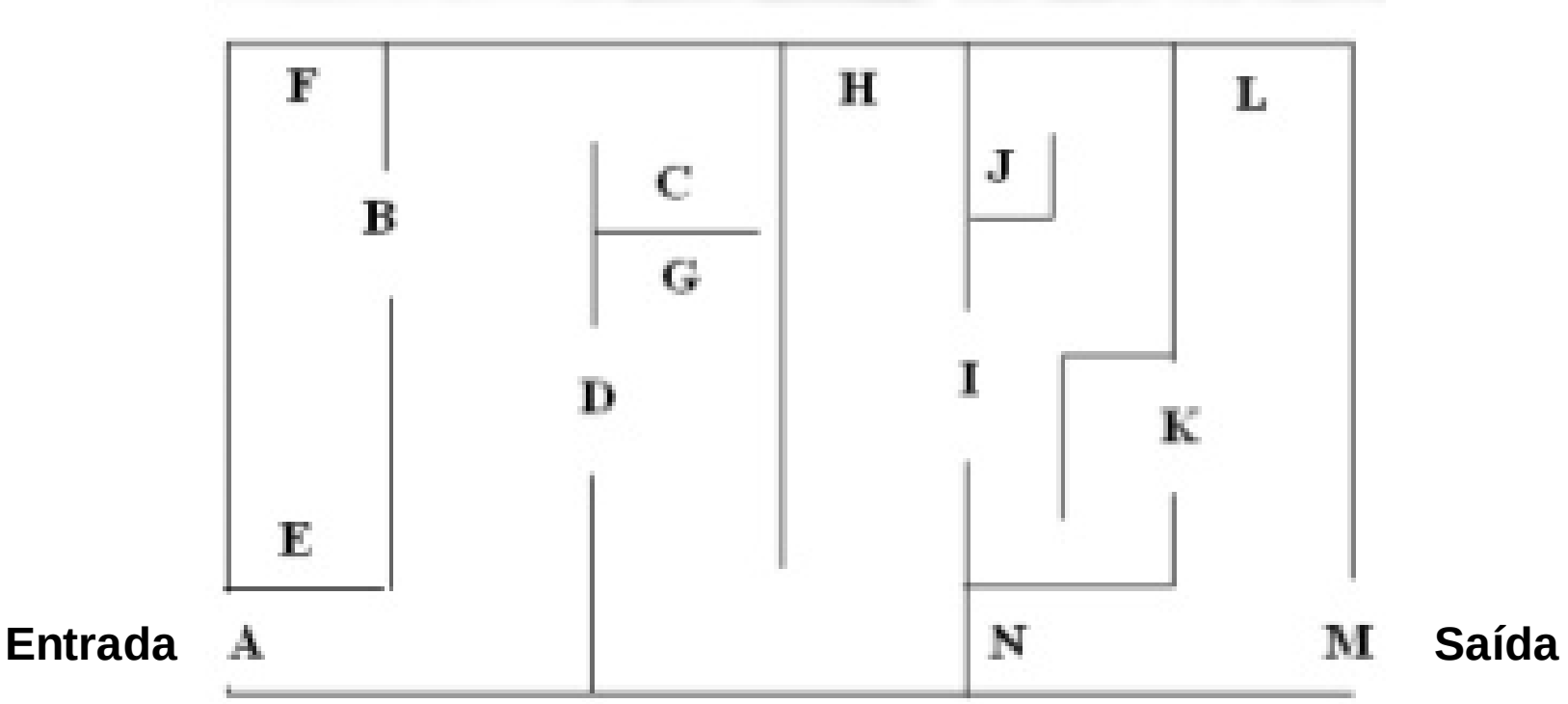


Exemplo: Percorrendo um labirinto

Um método que muitas pessoas conhecem para percorrer um labirinto é começar com a sua mão pelo lado esquerdo do labirinto (ou pelo lado direito, se preferir) e seguir o labirinto de um lado ao outro, sempre indo com a sua mão esquerda no lado esquerdo da parede do labirinto. Desse modo você terá a garantia de encontrar a saída.



Entrada **Saída**

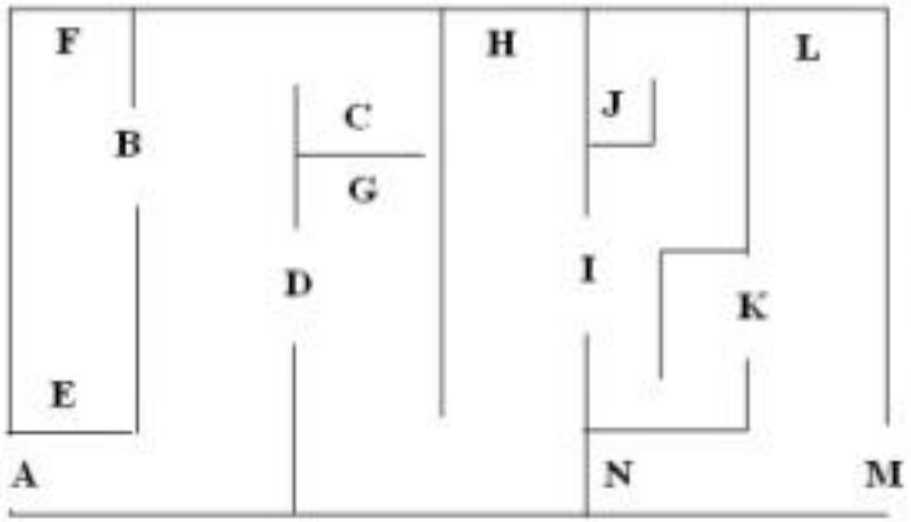


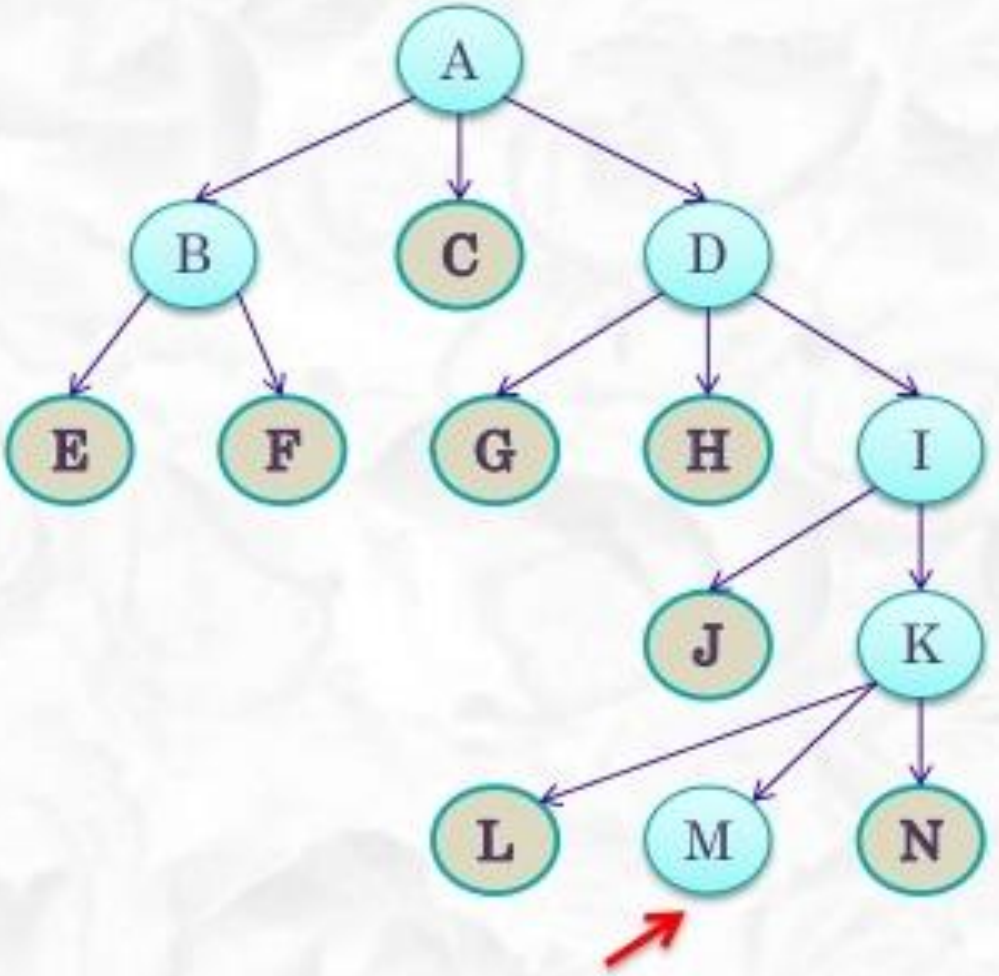
Certos pontos especiais no labirinto foram rotulados:

- A é a entrada do labirinto
- M é a saída do labirinto
- C, E, F, G, H, J, L e N são pontos sem saída
- B, D, I e K são pontos no labirinto onde se pode escolher qual a próxima direção a seguir

Ao seguir o labirinto, percorrendo com a mão pelo lado esquerdo, seria feito o seguinte caminho:
A, B, E, F, C, D, G, H, I, J, K, L, M

Podemos observar que ao seguir a **árvore de busca** utilizando **busca em profundidade** faz-se o mesmo caminho (A, B, E, F, C, D, G, H, I, J, K, L, M).





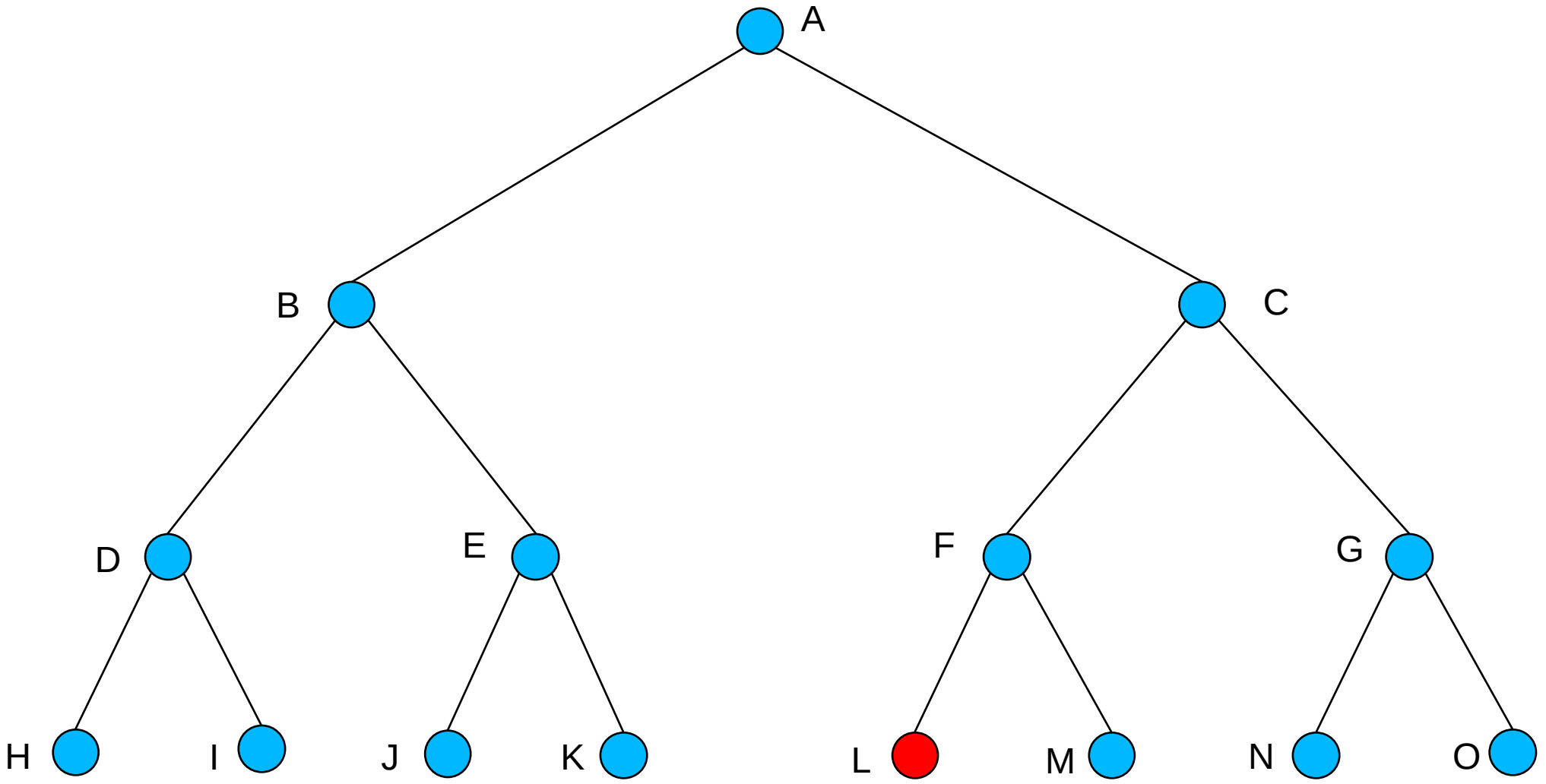
Algoritmo

```
Function profundidade()
{
     fila = [ ]; //inicializa uma fila vazia
     estado = no_raiz; //inicializa nó inicial
     while (true)
      {
           if eh_objetivo (estado)
              then return SUCESSO
           else
               inserir_na_frente_da_fila (sucessores (estado));
           if fila == [ ]
              then return FALHA;
           estado = fila [0]; //estado  = primeiro item na fila
           remover_primeiro_item_da (fila);
       }
}
```

Algoritmo

```
Function largura()
{
     fila = [ ]; //inicializa uma fila vazia
     estado = no_raiz; //inicializa nó inicial
     while (true)
      {
           if eh_objetivo (estado)
              then return SUCESSO
           else
               inserir_no_final_da_fila (sucessores (estado));
           if fila == [ ]
              then return FALHA;
           estado = fila [0]; //estado  = primeiro item na fila
           remover_primeiro_item_da (fila);
       }
}
```

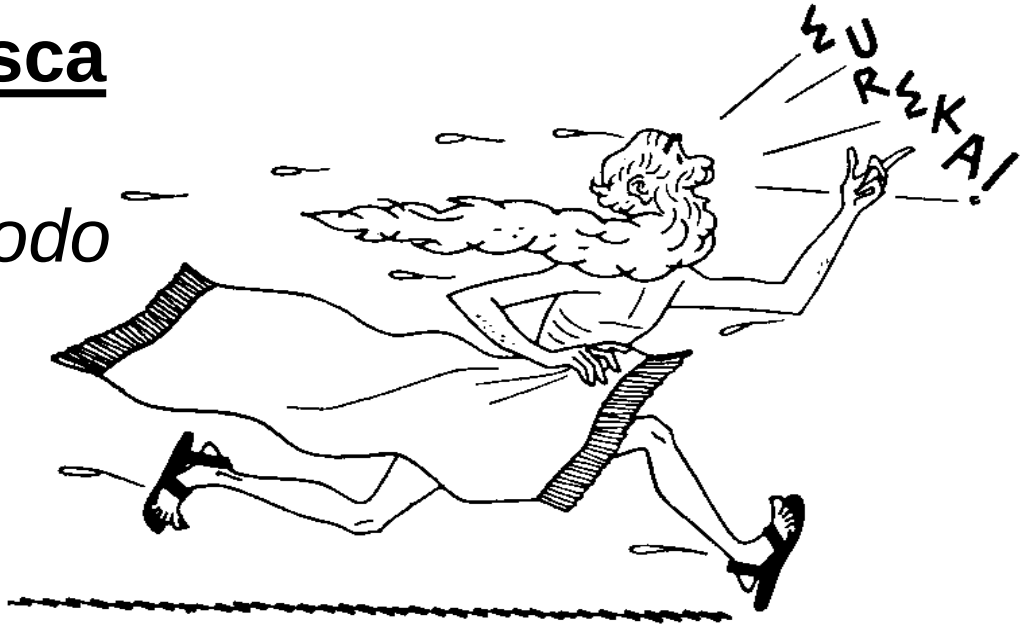
Análise de busca em profundidade e largura na seguinte árvore:



Busca em profundidade frequentemente encontrará o objetivo mais rapidamente que busca em largura se todos os nós folha tiverem a mesma profundidade em relação ao nó raiz. Entretanto, em árvores de busca nas quais haja uma subárvore muito grande que não contenha um objetivo, busca em largura terá quase sempre um desempenho melhor do que busca em profundidade.

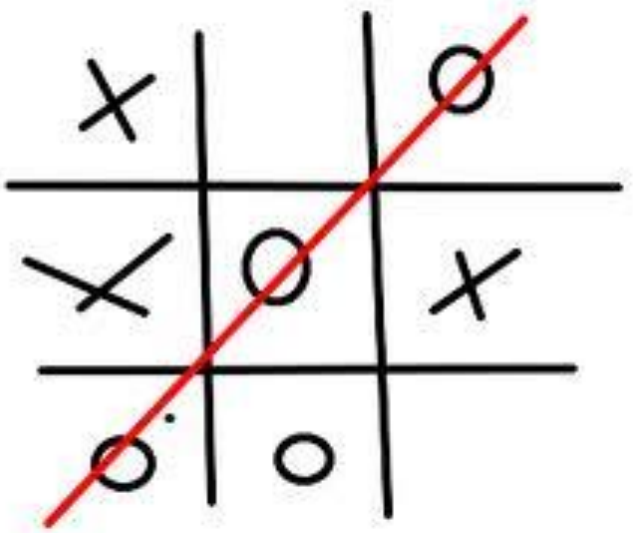
Usando Heurísticas na Busca

Humanos utilizam o tempo todo heurísticas (informações) para resolver problemas.

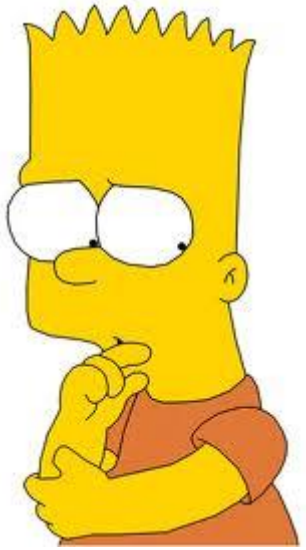


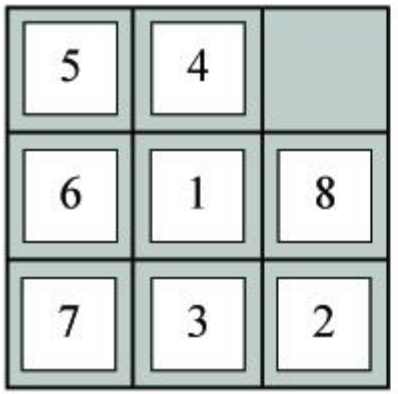
Computadores também podem utilizar heurísticas e, em muitos problemas, heurísticas podem tornar relativamente simples um problema, que de outra forma seria impossível.

Jogo da Velha

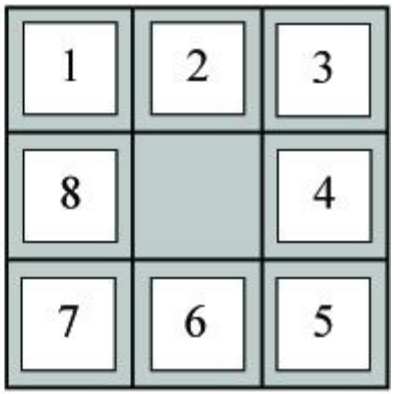


Quebra-cabeça de oito peças



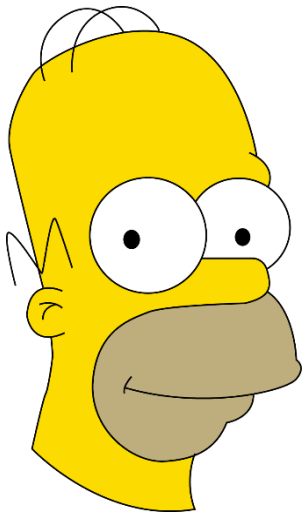


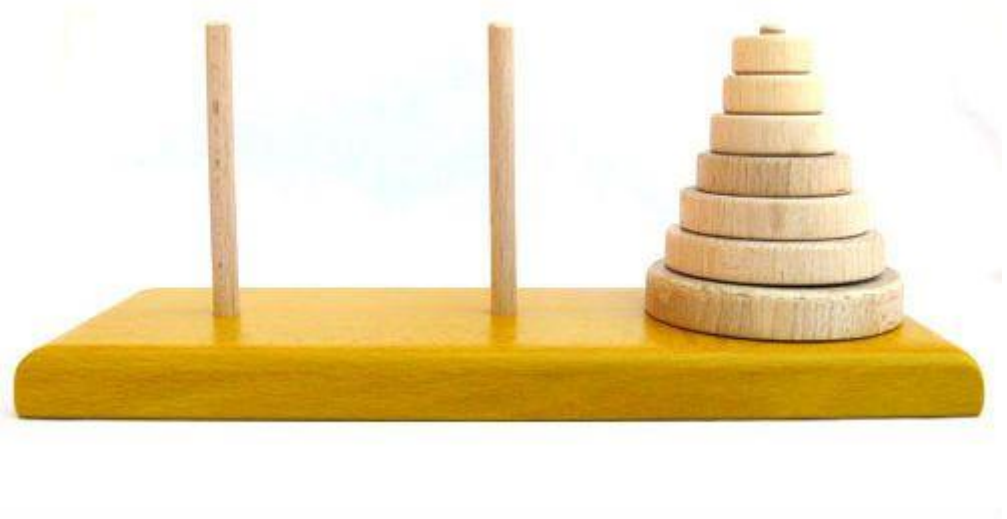
Start State



Goal State

Torres de Hanói





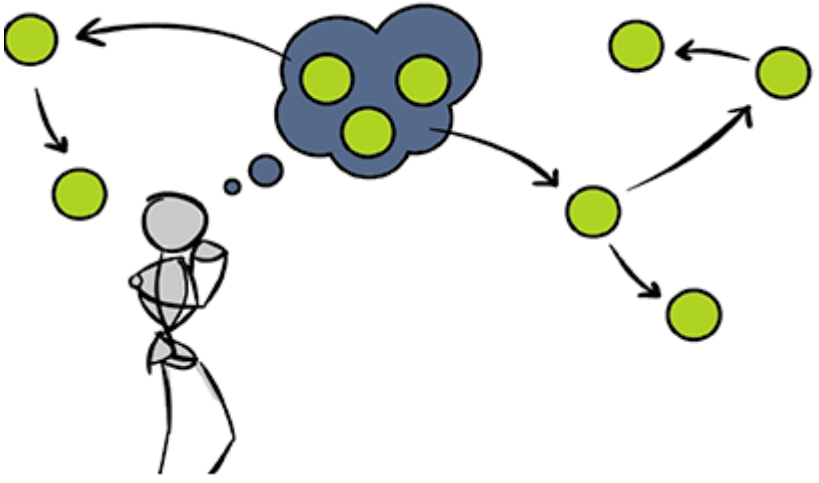
Menor caminho (rota)

Métodos Informados e Não Informados

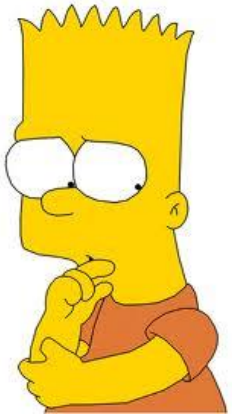
Um método de busca é **informado** se ele usa **informação adicional** sobre nós que ainda não tenham sido explorados para **decidir** quais nós examinar a seguir. Se um método não usa informação adicional ele e **não informado** ou **cego**.

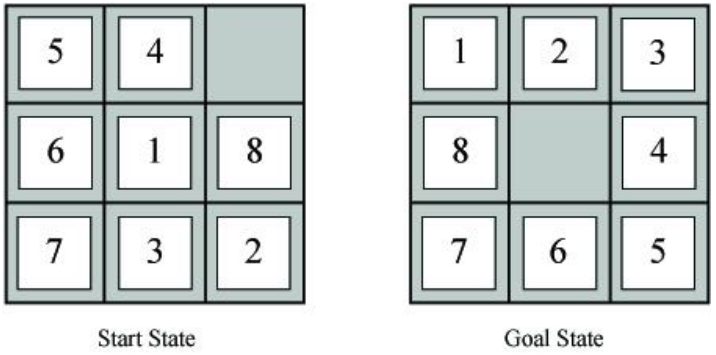
Em outras palavras, métodos de busca que usam **heurísticas** são **informados** e aqueles que não usam são ditos cegos.

Quanto mais informado um método de busca for, mais eficiente será a busca por ele realizada.



Quebra-cabeça de oito peças

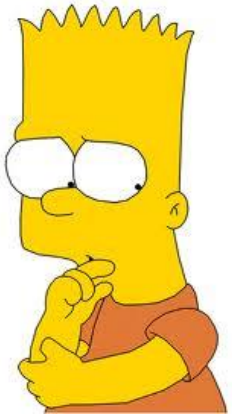


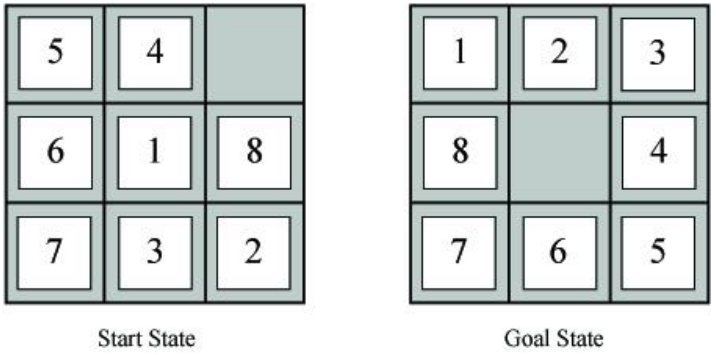


Uma busca exaustiva precisaria examinar cerca de 3^{20} estados, aproximadamente 3,5 bilhões. Devido a existirem apenas 9! ou 362.880 possíveis estados, a árvore de busca pode ser reduzida evitando estados repetidos.

- Tipicamente, são cerca de 20 deslocamentos para ir de um **estado inicial aleatório** ao **estado objetivo** e, assim, a árvore de busca tem uma profundidade em torno de 20.
- O fator de ramificação depende de onde está o espaço vazio. Se ele estiver no meio da grade, o fator de ramificação será de 4; se ele estiver em uma lateral, será de 3 e, se ele estiver em uma quina, será de 2. Assim o fator médio de ramificação da árvore de busca é de 3.

Quebra-cabeça de oito peças

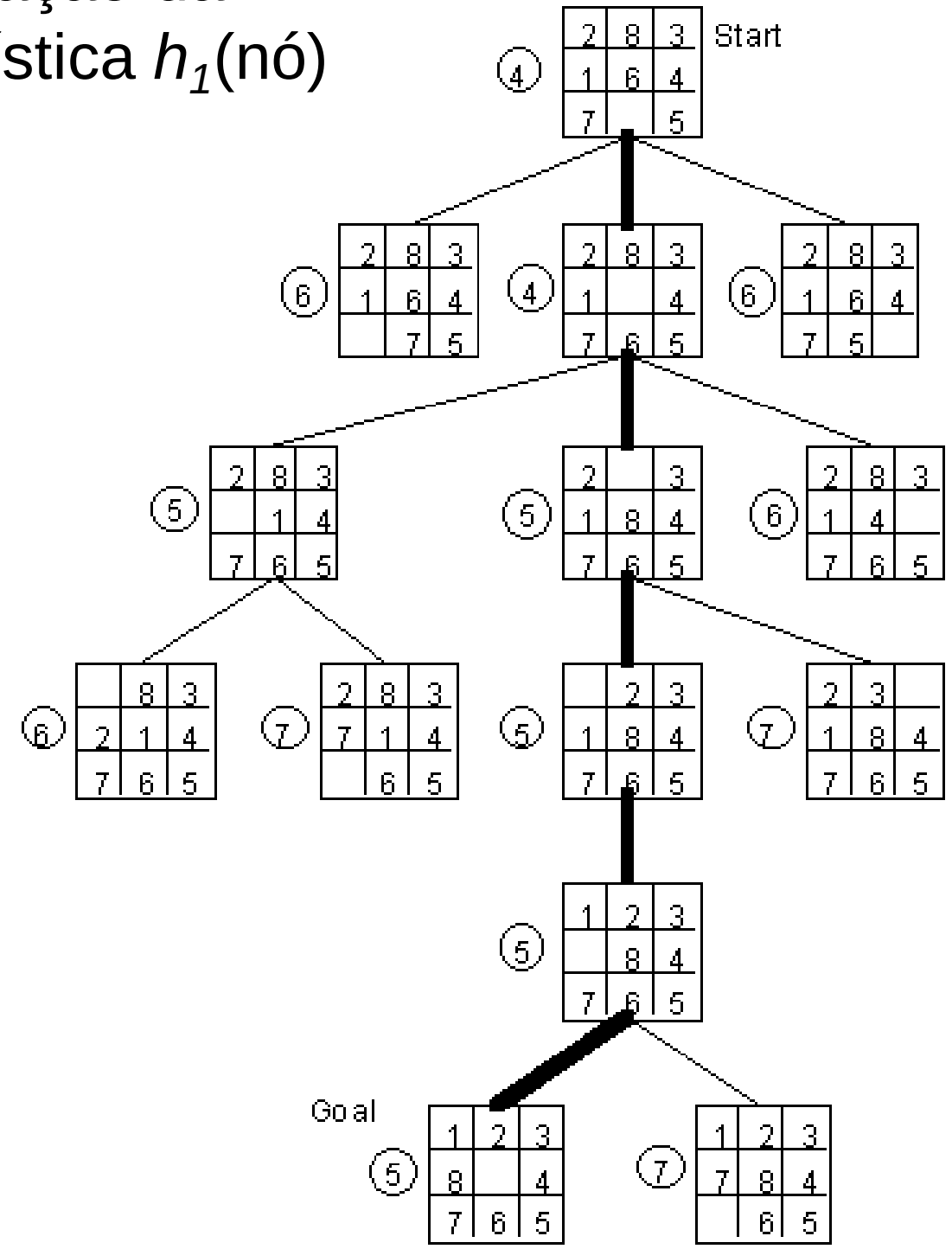




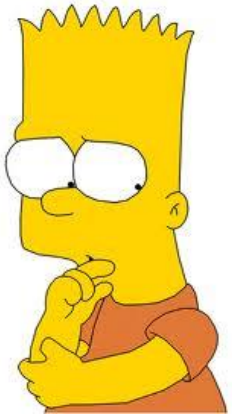
A primeira heurística a considerar é contar **quantas peças estão no lugar errado**. Podemos chamar essa heurística de h_1(nó). No caso do primeiro estado (figura), h_1(nó)=7, pois apenas uma peça está no lugar.

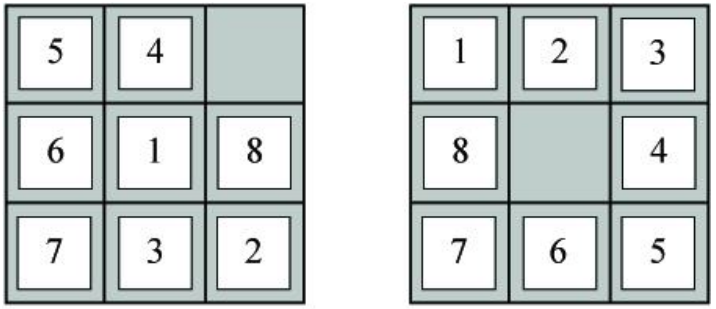
Uma **função de avaliação heurística** é uma função que, quando aplicada a um nó, dá um valor que representa uma boa estimativa da distância entre o nó e o objetivo

Aplicação da heurística h_1(nó)



Quebra-cabeça de oito peças

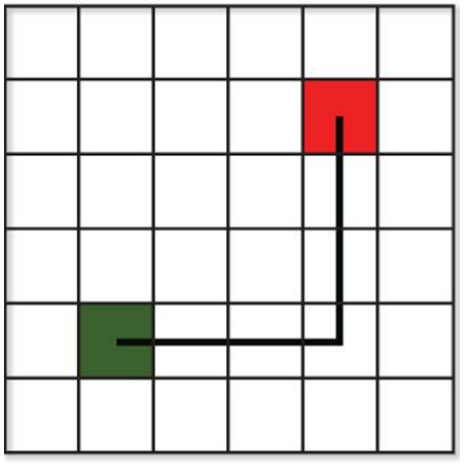




Start State Goal State

Uma heurística aprimorada, h_2, leva em conta **quão longe** cada peça deve ser deslocada para ir para o seu estado correto. Isso é possível somando-se as **distâncias de Manhattan** de cada peça, a partir de sua posição correta.

A distância de Manhattan é a soma dos deslocamentos horizontais e verticais que precisam ser feitos para ir de uma posição a outra.

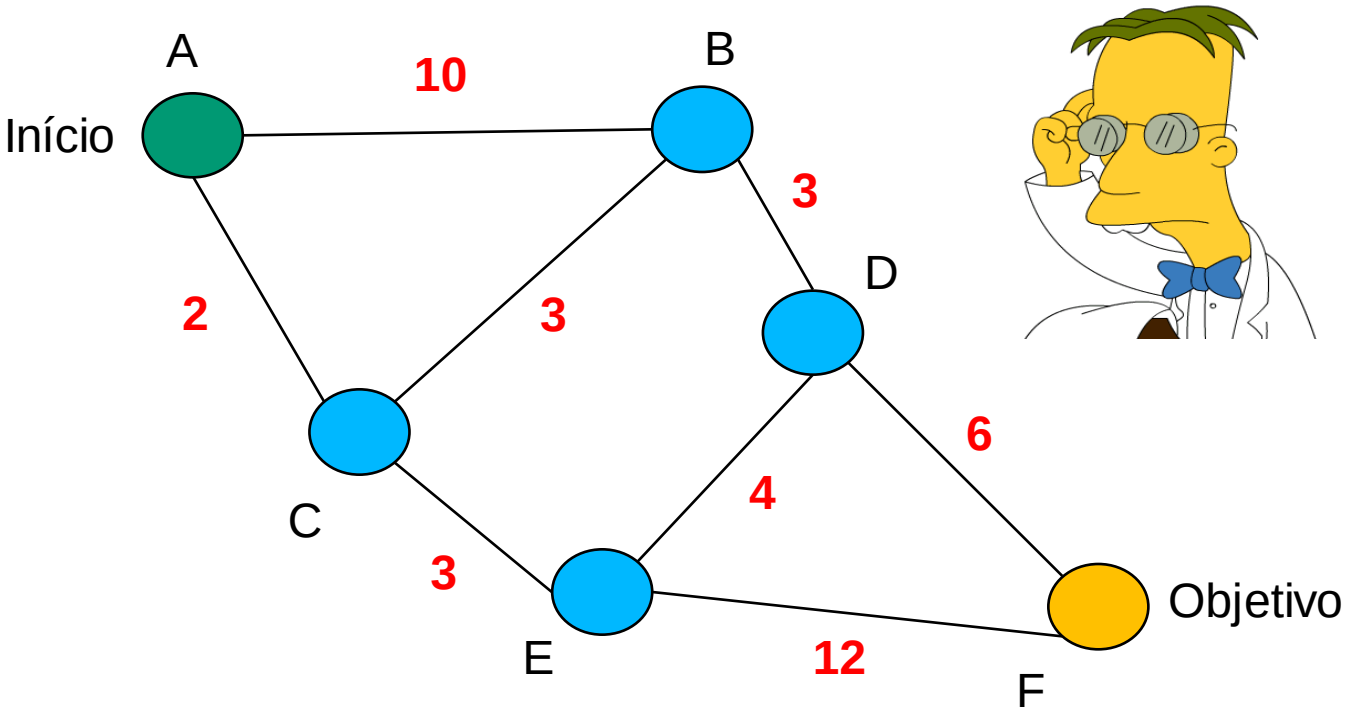


Manhattan Distance
h(*x*) = 6

<u>Primeiro estado</u>:

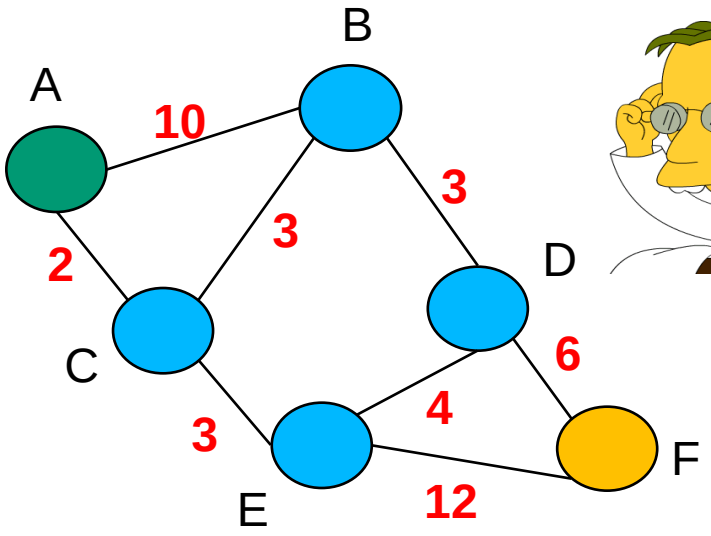
h_2(nó) = 4 + 2 + 2 + 2 + 2 + 0 + 3 + 3 = 18

Menor caminho (rota)



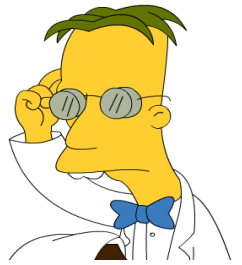
Cada **nó** representa uma cidade e as **ligações** entre os nós representam estradas que ligam as cidades. Cada ligação possui uma **distância**. O objetivo é encontrar o **caminho mais curto** possível desde a cidade *A* até a cidade *F*.

Menor caminho (rota)



Representação do espaço de busca como uma árvore de busca

Menor caminho (rota)



Os caminhos possíveis de *A* até *F*.

1. A,B,D,E,F
2. A,B,D,F
3. A,B,C,E,D,F
4. A,B,C,E,F
5. A,C,E,F
6. A,C,B,D,E,F
7. A,C,B,D,F
8. A,C,E,D,F

Busca em Profundidade?
Busca em Largura?

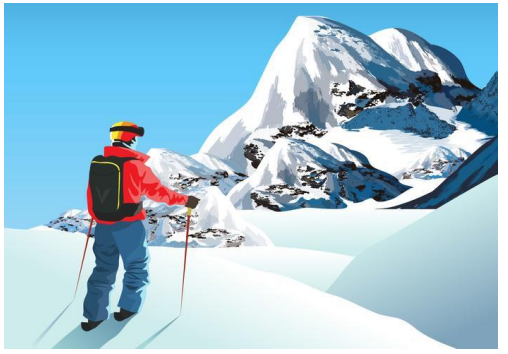
Subida da Colina (*Hill Climbing*)

É um **método de busca informado**, pois ele utiliza informação sobre o espaço de busca de uma maneira razoavelmente eficiente.

Se tentarmos escalar uma montanha, em dia de neblina, com um altímetro, mas sem mapa, utilizaríamos uma abordagem de subida do tipo **Gerar e Testar**.

- ✓ Verificar a altura a alguns centímetros da nossa posição corrente em cada direção: norte, sul, oeste e leste.
- ✓ Se houver uma posição com altura maior que a altura atual, vamos para lá.
- ✓ Se todas as direções levam para mais baixo do que a posição atual, então assumimos que chegamos ao topo.

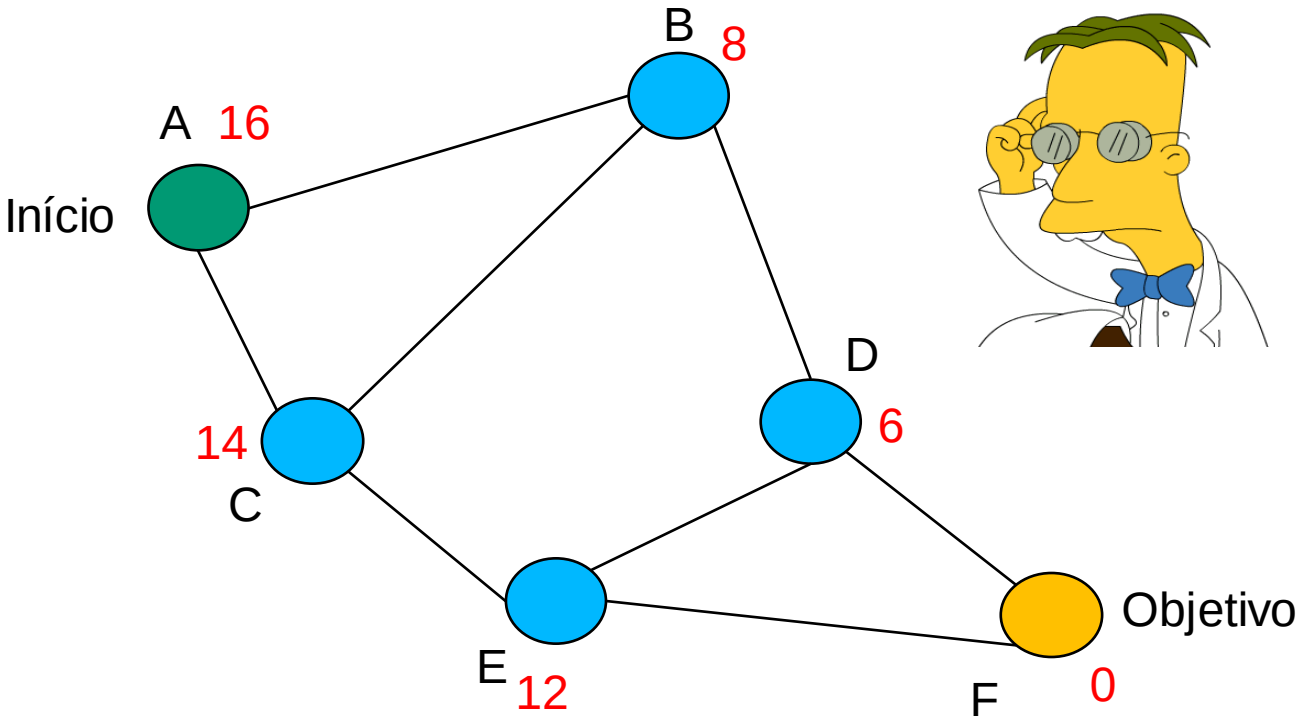
Ao examinar um árvore de busca, o algoritmo subida da colina irá para o primeiro nó sucessor que seja “melhor” que o nó corrente – em outras palavras, o primeiro nó que aparecer com um **valor heurístico inferior aquele do nó corrente**.



Subida da Colina (*Hill Climbing*)

Esse método também pode ser visto como uma **variação da busca em profundidade**, que utilize informação sobre **quão longe** cada nó está do nó **objetivo** para escolher qual caminho seguir em qualquer ponto.

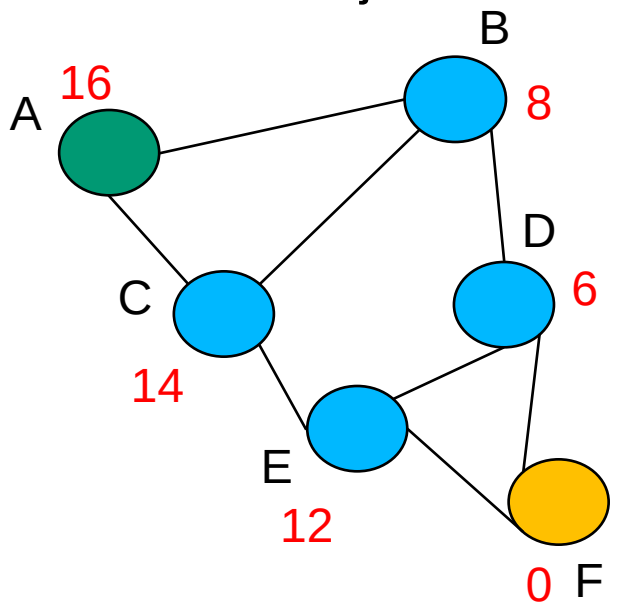
Para esse método vamos aplicar uma **heurística** à arvore de busca, que é a **distância em linha reta** de cada cidade à cidade objetivo.



Subida da Colina (*Hill Climbing*)

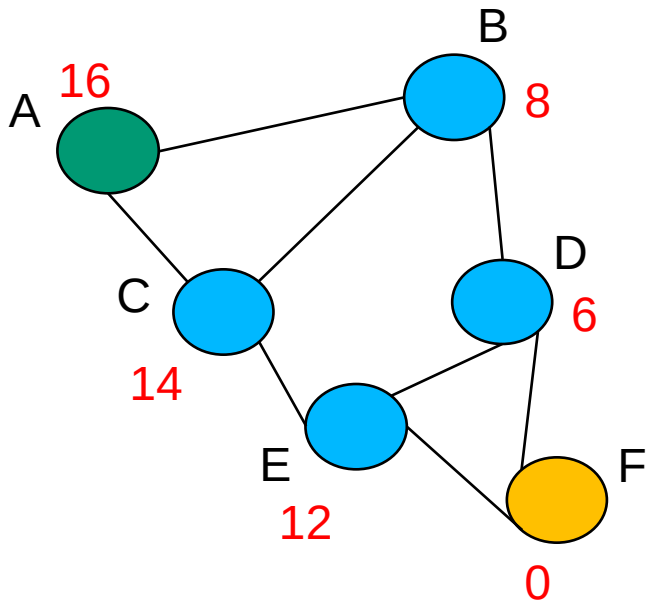
Agora o algoritmo segue como a busca em profundidade, mas, a cada passo, os novos nós a serem inseridos na fila são **ordenados pela distância** até o objetivo.

```
Function colina()
{
      fila = [ ]; //inicializa uma fila vazia
      estado = no_raiz; //inicializa nó inicial
      while (true)
       {
            if eh_objetivo (estado)
               then return SUCESSO
            else
                ordenar (sucessores (estado));
                inserir_na_frente_da_fila (sucessores (estado));
            if fila == [ ]
               then return FALHA;
            estado = fila [0]; //estado  = primeiro item na fila
            remover_primeiro_item_da (fila);
       }
}
```



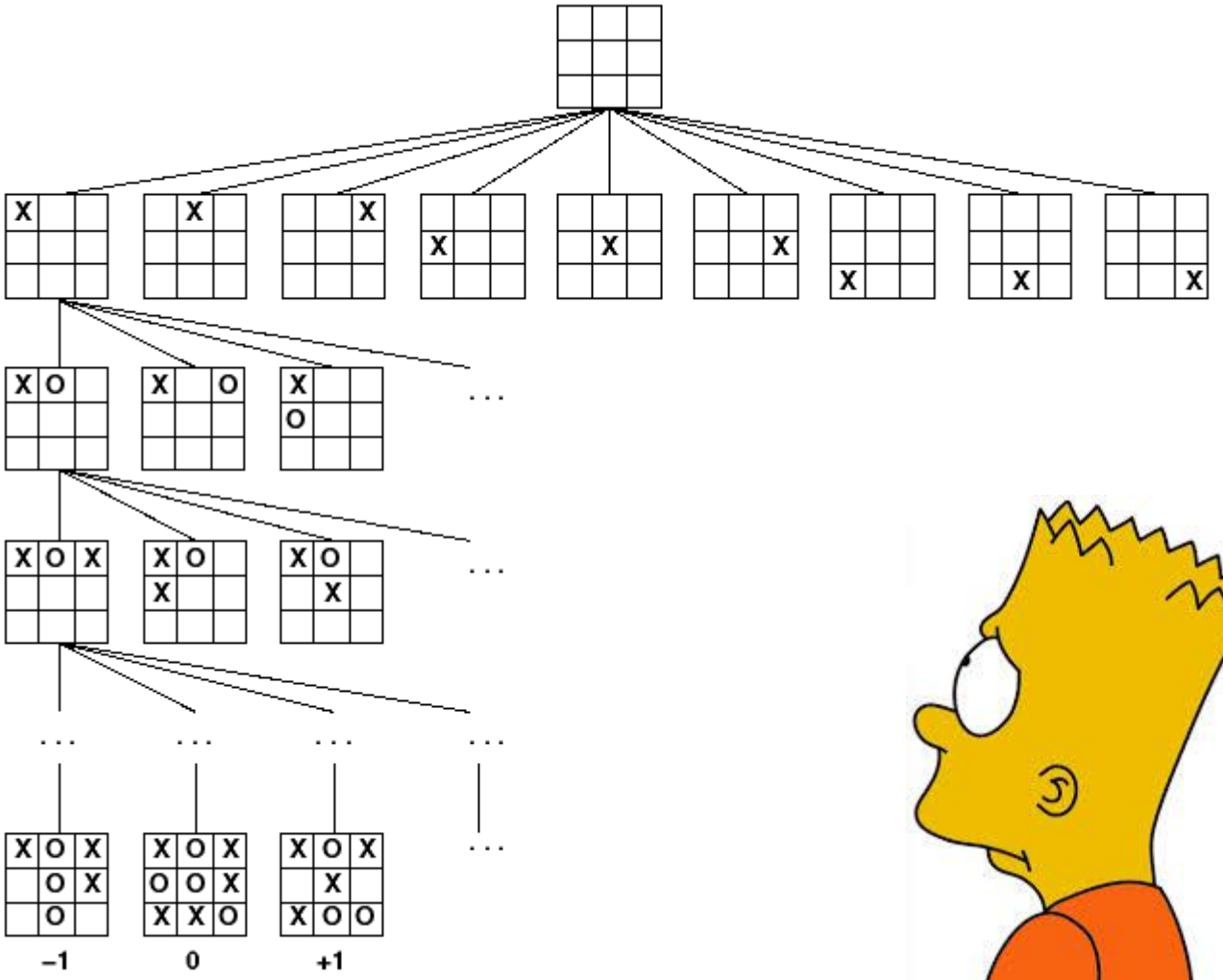
Assim, o algoritmo percorre a árvore como uma busca em profundidade, a cada passo escolhendo caminhos que parecem ser mais prováveis de levar ao objetivo.

Subida da Colina (*Hill Climbing*)



A subida da colina utiliza heurísticas para identificar caminhos eficientemente, mas não necessariamente identifica o melhor caminho.

Jogo da Velha



Existe uma informação útil (heurística)?

Jogo da Velha