4 Metodologias de Busca

"Se não acharmos qualquer coisa agradável, pelo menos acharemos algo novo."
(Voltaire, *Candide*)

4.1 INTRODUÇÃO

Por causa da natureza sequencial na qual computadores tendem a operar, a busca é necessária para determinar soluções para uma enorme gama de problemas.

Para construirmos um sistema para Solução de um Problema, é necessário:

- Definir o problema precisamente
- Determinar o tipo de conhecimento necessário para resolver o problema.
- Criar um esquema da representação para tal conhecimento
- Escolher (ou desenvolver) técnicas de solução adequadas para resolver o problema.

4.1.1. DEFINIÇÃO DO PROBLEMA

Devemos ter em consideração:

- Representação Computacional do problema
- Objetivo (o que se pretende alcançar)
- Onde Iniciar
- Como modificar os estados
- Como identificar modificações úteis na solução do problema

Existem alguns problemas que podemos considerar **difíceis**: entendimento de outra língua, jogar xadrez, resolver integrais indefinidas, prever o clima, prever mudanças no estoque de uma loja, organizar uma linha de produção, acomodar objetos dentro de um espaço físico limitado.

4.1.2. DECOMPOSIÇÃO E GENERALIZAÇÃO DO PROBLEMA

A complexidade de problemas é uma questão muito relevante, devendo ser sempre considerada; pois pode acontecer do número de prováveis caminhos ser muito grande, ou até mesmo impossível de ser processado, como é o caso de problemas combinatórios de ordem muito elevada, que não podem ser resolvidos em tempos viáveis.

Neste caso, uma das primeiras tarefas a se fazer é verificar se o problema pode ser decomposto em problemas menores, cujas soluções tendem a ser menos complexas. Assim, a resolução do problema principal poderá ser obtida resolvendo problemas menores.

Além disso, é importante generalizar o problema, desenvolvendo métodos gerais para solução de problemas. Dessa forma torna-se fácil comparar problemas (especialmente o tamanho). É necessário, porém, escolher a metodologia mais adequada.

4.2 DEFINIÇÃO DO PROBLEMA COMO UM ESPAÇO DE ESTADOS

Solução de problemas é um importante aspecto da Inteligência Artificial. 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 até então.

Uma possível estratégia para solução de problemas é listar todos os estados possíveis. A solução do problema consiste em percorrer o espaço de estados a partir do estado inicial até o estado objetivo. É necessário desenvolver um conjunto de operadores que modifique um estado para um outro estado.

Por exemplo, consideremos o problema de procurar as lentes de contato no campo de futebol. O **estado inicial** é como começamos, ou seja, sabemos que as lentes estão em algum lugar no campo de futebol, mas não sabemos aonde. Se utilizarmos a representação na qual examinamos o campo em unidades de um centímetro quadrado, então nossa primeira ação será examinar o quadrado no canto superior esquerdo do campo. Se não acharmos as lentes lá, podemos considerar o estado agora como sendo examinamos o quadrado do canto superior esquerdo e não achamos as lentes. Depois de várias ações, o estado deverá ser que examinamos 500 quadrados e agora encontramos as lentes no último quadrado examinado. Esse é um **estado objetivo**, pois ele satisfaz a meta que tínhamos de encontrar as lentes de contato.

Busca é um método que pode se utilizado por computadores para examinar um espaço de problema, como esse, de modo a encontrar um objetivo. Frequentemente, queremos encontrar o objetivo o mais rápido possível ou sem utilizar muitos recursos. Um espaço de problema também pode ser considerado como espaço de busca, pois, de modo a solucionar o problema, faremos uma busca no espaço por um estado objetivo. Continuaremos utilizando o termo **espaço de busca** para descrever esse conceito.

Então para definir o problema como um espaço de estados, é necessário:

- Conjunto de **estados** possíveis.
- Conjunto de **operações** possíveis que modifiquem um estado.
- Especificação de um estado **inicial**
- Especificação de um estado **final**

4.2.1. ESPAÇO DE ESTADOS

Geralmente é impossível listas todos os espaços possíveis:

- Utilização de abstrações para descrever todos os espaços válidos
- Pode ser mais fácil descrever estados inválidos
- Algumas vezes é útil fornecer uma descrição geral do espaço de estados e um conjunto de restrições.

4.2.2. OPERAÇÕES

- O sistema de solução se move de um estado para outro de acordo com operações bem definidas.
- Geralmente estas operações são descritas como **regras**. Uma operação e as condições que devem ser satisfeitas (restrições) antes da operação poder ser aplicada é chamada de **regra**. Tipicamente é necessário mesclar **regras gerais** e **regras específicas**. Informação prévia sobre a solução tende a produzir regras específicas e aumentar a velocidade da busca.
- Um sistema de controle decide quais regras são aplicáveis em um dado estado e resolve conflitos e/ou ambiguidades.

4.2.3. EXEMPLO – Problema dos Jarros de Água

Considerando que possuo dois jarros de água: um maior com capacidade para 4 litros e outro com capacidade para 3 litros e preciso medir 2 litros.

O **espaço de estados** pode ser representado por dois inteiros x e y: x = litros no jarro de 4 litros e y = litros no jarro de 3 litros. **Espaço de Estados = (x, y) tal que $x \in \{0,1,2,3,4\}$, $y \in \{0,1,2,3\}$**

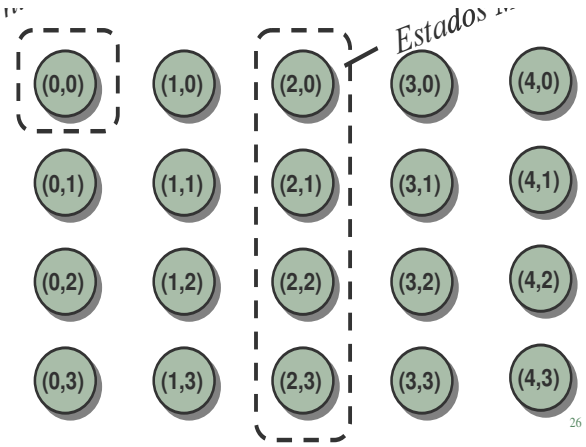
O **estado inicial** ocorre quando ambos os jarros estão vazios: (0,0) e o **estado objetivo** é qualquer estado que possua 2 litros de água no jarro de 4 litros): **(2, n)** para qualquer n

Restrições

- Não é possível colocar água em um jarro cheio.
- Restrições são associadas para que uma operação possa ser aplicada sobre um estado

Domínio de Regras Específicas

- (0,2) → (2,0)
- (x,2) → (0,2)

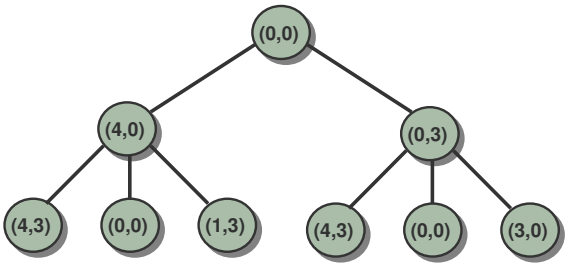


Operações com Jarros de Água:

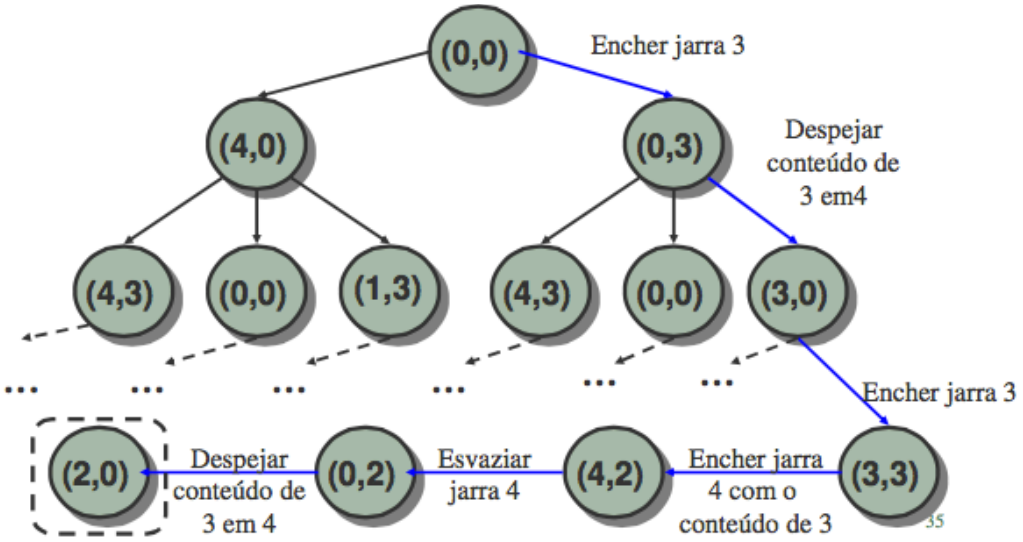
- Colocar 3 lt. no jarro 3: (x,y) → (x,3)
- Colocar 4 lt. no jarro 4: (x,y) → (4,y)
- Esvaziar jarro 3: (x,y) → (x,0)
- Esvaziar jarro 4: (x,y) → (0,y)
- Colocar o jarro 3 no 4:(0,y) → (y,0)

Estratégias de Controle:

- Lista ordenada de regras : aplica a primeira regra adequada para determinada situação.
- Escolhe qualquer regra (randomicamente) adequada para determinada situação.
- Aplica todas as regras adequadas, e armazena o caminho resultante de todos os estados obtidos.
- No próximo passo faz o mesmo para todos os estados.



Exemplo de Solução:



4.3 BUSCA GUIADA POR DADOS OU BUSCA GUIADA POR OBJETIVOS

Existem duas abordagens principais para fazer busca em uma árvore de busca, que aproximadamente correspondem às abordagens de-cima-para-baixo e de-baixo-para-cima. **Busca guiada por dados** parte de um estado inicial e usa ações que são permitidas para ir em frente até que um objetivo seja atingido. Essa abordagem também é conhecida como **encadeamento para frente**.

Alternativamente, a busca pode começar no objetivo e voltar para um estado inicial, vendo quais deslocamentos poderiam ter levado ao estado objetivo. Isso é **busca guiada por objetivos**, também conhecida como **encadeamento para trás**.

A maioria dos métodos de busca que examinaremos é formada por métodos de busca guiada por dados: eles partem de um estado inicial (a raiz na árvore de busca) e trabalham em direção ao nó objetivo.

Busca guiada por objetivos e busca guiada por dados terminam produzindo o mesmo resultado, porém, dependendo da natureza do problema a ser resolvido, um dos métodos pode resolvê-lo mais eficientemente que o outro — em particular, em algumas situações um deles pode envolver examinar mais estados que o outro.

Busca guiada por objetivos é particularmente útil em situações nas quais o objetivo pode ser claramente especificado (por exemplo, um teorema a ser provado ou encontrar uma saída em um labirinto). É também, sem dúvida, a melhor escolha em problemas como diagnósticos médicos onde o objetivo (a condição a ser diagnosticada) é conhecido, mas os outros dados (neste caso, as causas da condição) precisam ser encontrados.

Busca guiada por dados é mais útil quando os dados iniciais são fornecidos e não temos clareza sobre o objetivo. Por exemplo, um sistema que analise dados astronômicos e, assim, faça deduções sobre a natureza de estrelas e planetas, receberia um grande volume de dados, mas não necessariamente lhe seria dado um objetivo direto. Em vez disso, seria esperado que analisasse os dados e tirasse suas próprias conclusões. Esse tipo de sistema tem um imenso número de possíveis objetivos que poderia localizar. Neste caso, busca guiada por dados é mais apropriada.

É interessante considerar um labirinto que tenha sido criado para ser percorrido de um ponto inicial de modo a chegar a um ponto final específico. É quase sempre mais fácil começar do ponto final e voltar ao ponto inicial. Isso se deve aos diversos caminhos sem saída que foram estabelecidos desde o ponto inicial (dados) e somente um caminho que foi estabelecido para o ponto final (objetivo). Como resultado, vir do objetivo ao início tem somente um caminho possível.

4.4 TÉCNICAS DE BUSCA

Existem muitas técnicas de busca que operam sobre uma estrutura de **árvore**, porém, para que estas técnicas possam ser aplicadas em uma estrutura de rede (**grafo**), precisam de algum mecanismo de controle para que todos os nós possam ser visitados e, também para que não se visite um mesmo nó várias vezes.

Este controle é geralmente realizado com o apoio de uma **lista** de **nós** a visitar. Um procedimento genérico consiste em, inicialmente, colocar o primeiro nó (**nó inicial**) na lista e, em seguida, a cada nó visitado, retirar o nó da lista e aplicar a **função teste** para verificar se é um **nó objetivo** (solução para o problema). Em caso afirmativo, interrompe-se a busca, caso contrário, o nó é expandido, ou seja, seus filhos são colocados na lista. Um nó X é filho de um nó Y se acessado por um único arco a partir do nó Y, e não é filho de nenhum outro nó.

Além das **buscas cegas**, que visitam todos os nós sequencialmente, também existem as técnicas de **buscas heurísticas**, que usam alguma informação para estimar o custo de cada ação, como, por exemplo, uma distância estimada entre o nó atual e o nó objetivo. Assim, tenta-se encontrar uma solução de forma mais rápida, do que percorrer, indiscriminadamente, todos os nós da árvore.

O desempenho dos algoritmos de busca pode ser avaliado em quatro aspectos:

1. **Completeza**: quando o algoritmo sempre encontra uma solução, se ela existe;
2. **Otimização**: quando o algoritmo encontra a solução ótima;

3. **Complexidade Temporal (de tempo)** : se refere ao tempo máximo exigido pelo algoritmo para realizar a busca;
4. **Complexidade Espacial (de memória)**: se refere à quantidade máxima de memória usada pelo algoritmo, geralmente, utilizada para guardar informações sobre os nós visitados e a serem visitados (Lista)

4.5 GERAR E TESTAR – BUSCA CEGA

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. Se for, a busca teve sucesso e não precisa ser levada adiante. Caso contrário, o procedimento segue para o próximo nó. Essa é a forma mais simples de busca de força bruta (também chamada de busca exaustiva), assim chamada porque ela não pressupõe conhecimento adicional além de como percorrer a árvore e busca e como identificar nós folhas e nós objetivos, que terminará por examinar cada nó da árvore até encontrar um objetivo.

Para ter sucesso, Gerar e Testar precisa ter um **Gerador** adequado, que deve satisfazer a três propriedades:

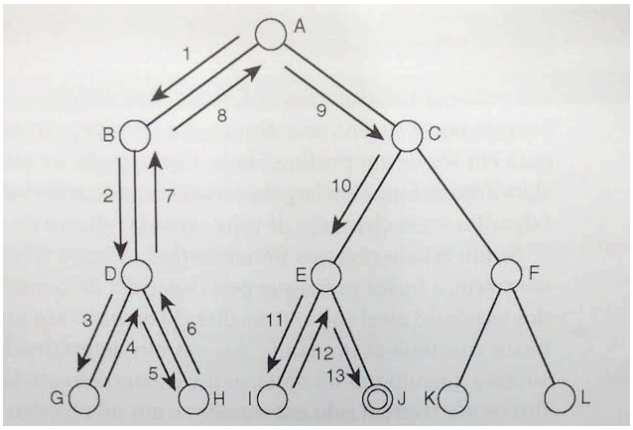
1. Ele deve ser **completo**: em outras palavras, ele deve gerar todas soluções possíveis; caso contrário, poderia descartar uma solução adequada.
2. Ele **não** deve ser **redundante**: isso significa que não deve gerar a mesma solução duas vezes.
3. Ele deve ser **bem informado**: isso significa que só deve propor soluções adequadas e não deve examinar possíveis soluções que não combinem com o espaço de busca.

O método Gerar e Testar pode ser aplicado com sucesso a diversos problemas e, na verdade, é o modo pelo qual as pessoas frequentemente solucionam problemas onde não há informação adicional sobre como alcançar uma solução.

Gerar e Testar é também, algumas 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.

4.6 BUSCA EM PROFUNDIDADE

Um algoritmo comumente utilizado é **busca em profundidade**. Busca em profundidade é assim chamada por seguir cada caminho até a sua maior profundidade antes de seguir para o próximo caminho. O princípio subjacente à busca em profundidade é ilustrado na figura a seguir. Supondo que comecemos pelo lado esquerdo e sigamos para o lado direito, a busca em profundidade envolve descer pelo caminho mais à esquerda na árvore até achar uma folha. Se este for um estado objetivo, a busca foi concluída e será relatado sucesso.



Se a folha não representar um estado objetivo, a busca retrocederá ao primeiro nó anterior que tenha um caminho não explorado. Na figura, após examinar o nó G e descobrir que este não é um objetivo, a busca retrocede ao nó D e explora seus outros filhos. Neste caso, só há um outro filho, que é H. Depois que este nó for examinado, a busca retrocederá ao próximo nó não expandido, que é A, pois B não tem filhos não explorados.

Esse processo continua até que todos os nós tenham sido examinados, caso em que a busca termina com falha, ou até que um estado objetivo tenha sido alcançado, caso em que a busca termina com sucesso. Na figura, a busca termina no nó J, que é o nó objetivo. Como resultado, os nós F, K e L não foram examinados.

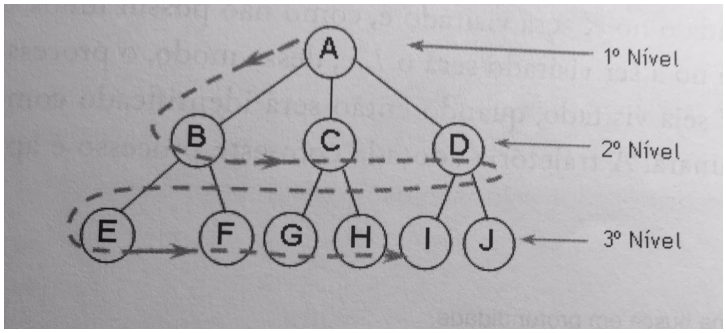
Busca em profundidade utiliza um método chamado de **retrocesso cronológico** para voltar na árvore de busca, uma vez que um caminho sem saída seja encontrado. Retrocesso cronológico é assim chamado porque ele desfaz escolhas em ordem contrária ao momento em que as decisões foram originalmente tomadas.

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

Busca em profundidade é bem utilizada por computadores para problemas de busca do tipo como localizar um arquivo em um disco ou por motores de busca do tipo para **indexação** (spidering) na Internet.

Como qualquer pessoa que tenha utilizado a operação localizar em seu computador sabe, a busca em profundidade pode ter problemas. Em especial, se um ramo da árvore de busca for muito grande, ou mesmo infinito, então o algoritmo de busca gastará uma quantidade descomedida de tempo examinando este ramo, que poderá nunca levar a um estado objetivo.

4.7 BUSCA EM LARGURA



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

Como pode ser visto na figura, o algoritmo de busca em largura começa examinando todos os nós um nível (algumas vezes chamado de uma camada) abaixo do nó raiz.

Se um estado objetivo for encontrado aqui, é relatado sucesso. Caso contrário, a busca prossegue pela expansão de caminhos a partir de todos os nós do nível corrente na direção do próximo nível. Desse modo, a busca continua examinando nós em um determinado nível, relatando sucesso quando um nó objetivo for encontrado e relatando falha se todos os nós tiverem sido examinados e um nó objetivo não tiver sido encontrado.

Busca em largura é um método bem melhor para utilizar em situações nas quais a árvore pode ter caminhos muitos profundos, principalmente se o nó objetivo estiver em uma parte mais rasa da árvore. Infelizmente, ela não funciona tão bem quando o fator de ramificação da árvore é muito alto, tais como ao examinar **árvores de jogos** para jogos como Xadrez.

Busca em largura não é uma boa ideia em árvores onde todos os caminhos levam a um nó objetivo com caminhos de comprimentos parecidos. Em situações como esta, a busca em profundidade funciona muito melhor, pois identificaria um nó objetivo quando atingisse o final do primeiro caminho examinado.

4.8 PROPRIEDADES DOS MÉTODOS DE BUSCA

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

As vantagens comparativas das buscas em profundidade e em largura são mostradas acima. A busca em profundidade é geralmente mais fácil de implementar que a busca em largura e, normalmente, exige menos uso de memória, pois somente precisa armazenar informação sobre o caminho que está sendo explorado, enquanto a busca em largura precisa armazenar informação sobre todos os caminhos que atingem a profundi-

dade corrente. Esta é uma das principais razões pela qual a busca em profundidade é amplamente utilizada na solução de problemas computacionais cotidianos.

O problema de caminhos infinitos pode ser evitado na busca em profundidade pela aplicação de um **limiar de profundidade**. Isso significa que um caminho será considerado como terminado quando for atingida uma profundidade específica. Isto tem a desvantagem de alguns estados objetivos (ou, em alguns casos, o único estado objetivo) poderem ser perdidos, mas assegura que todos os ramos da árvore de busca sejam explorados em um tempo razoável.

Métodos de busca diferentes trabalham de modos diversos. Há várias propriedades importantes que um método de busca deve ter para ser mais útil:

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

4.8.1 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** de um método está relacionada à duração de tempo que este leva para encontrar um estado objetivo. A **complexidade em termos de espaço** está relacionada à quantidade de memória que o método precisa utilizar.

É comum utilizar a notação O para descrever a complexidade de um método. Por exemplo, a busca em largura tem complexidade de tempo de $O(b^d)$, onde ***b*** é o fator de ramificação da árvore e ***d*** é a profundidade do nó objetivo na árvore.

Busca em profundidade é muito eficiente em relação a espaço, pois só precisa guardar informações sobre o caminho que está sendo examinado no momento, mas não é eficiente em tempo, pois demora muito examinando ramos muito profundos da árvore.

Obviamente, complexidade é uma importante propriedade a ser compreendida sobre um método de busca. Um método de busca que seja muito ineficiente pode ter um desempenho bem razoável para um pequeno problema de teste, mas diante de um grande problema do mundo real, gastaria um inaceitavelmente longo período de tempo. Pode haver uma grande diferença entre o desempenho de dois métodos de busca e selecionar aquele que seja executado mais eficientemente em uma situação específica pode ser muito importante.

Esta complexidade deve ser frequentemente pesada em relação à adequação da solução gerada pelo método. 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.

4.8.2 COMPLETUDE

Um método de busca é descrito como **completo** se ele garantir encontrar um estado objetivo, se existir algum. Busca em largura é completa, mas busca em profundidade não é, pois pode explorar um caminho de extensão infinita e nunca achar um nó objetivo que esteja em outro caminho.

Completude é geralmente uma característica desejável, pois utilizar um método de busca que nunca ache uma solução não é frequentemente útil. Por outro lado, pode ser o caso (como ao realizar busca em uma árvore de jogos, durante um jogo, por exemplo) em que pesquisar a árvore de busca inteira não seja necessário, ou simplesmente não seja possível, situação na qual um método que pesquise o suficiente da árvore seria bom o bastante.

Um método que não seja completo tem a desvantagem de não poder ser necessariamente confiável ao relatar que não há solução.

4.8.3 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 um 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 —, mas uma vez que a tenha encontrado, há a

garantia de que ela seja a melhor. Isso dá certo se o processo de busca por uma solução consumir menos tempo do que efetivamente implementar a solução. Por outro lado, em alguns casos, implementar a solução depois que ela tenha sido encontrada é muito simples, situação na qual seria melhor utilizar um método de busca mais rápido e não se preocupar se ele encontrou a solução ótima ou não.

Busca em largura é um método ótimo de busca, mas busca em profundidade não é. A busca em profundidade oferece a primeira solução que encontra, que pode ser a pior solução que existe. Devido à busca em largura examinar todos os nós em uma dada profundidade antes de ir para a próxima profundidade, se for encontrada uma solução, não poderá haver outra solução antes dessa na árvore de busca.

Em alguns casos, o termo ótimo é utilizado para descrever um 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.

4.8.4 IRREVOGABILIDADE

Métodos que retrocedem são descritos como uma tentativa. Métodos que não retrocedem, e que, portanto, examinam apenas um caminho, são descritos como irrevogáveis. Busca em profundidade é um exemplo de busca por tentativa.

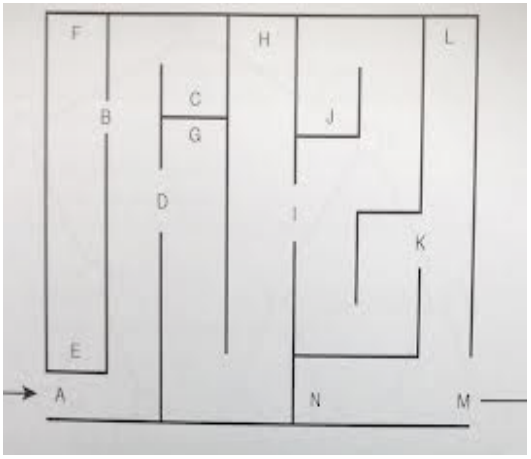
Métodos de busca irrevogáveis frequentemente encontrarão soluções subótimas para problemas, pois tendem a ser enganados por ótimos locais — soluções que parecem localmente boas, porém, são menos favoráveis quando comparadas com outras soluções em outros lugares do espaço de busca.

4.9 POR QUE HUMANOS USAM BUSCA EM PROFUNDIDADE

Tanto busca em profundidade como busca em largura são fáceis de implementar, apesar da busca em profundidade ser um pouco mais fácil. Ela é também um tanto mais fácil para humanos entenderem, pois se parece bem mais com o modo natural que eles procuram por coisas.

4.9.1 EXEMPLO 1: Percorrendo um Labirinto

Quando percorrem um labirinto muitas pessoas irão perambular a esmo, esperando encontrar por fim a saída. Essa abordagem geralmente será bem-sucedida no final, mas não é a mais racional e frequentemente leva ao que chamamos de "andar em círculos". Esse problema, é claro, refere-se à busca em espaços que contenham laços e isso pode ser evitado convertendo o espaço de busca em uma árvore de busca.



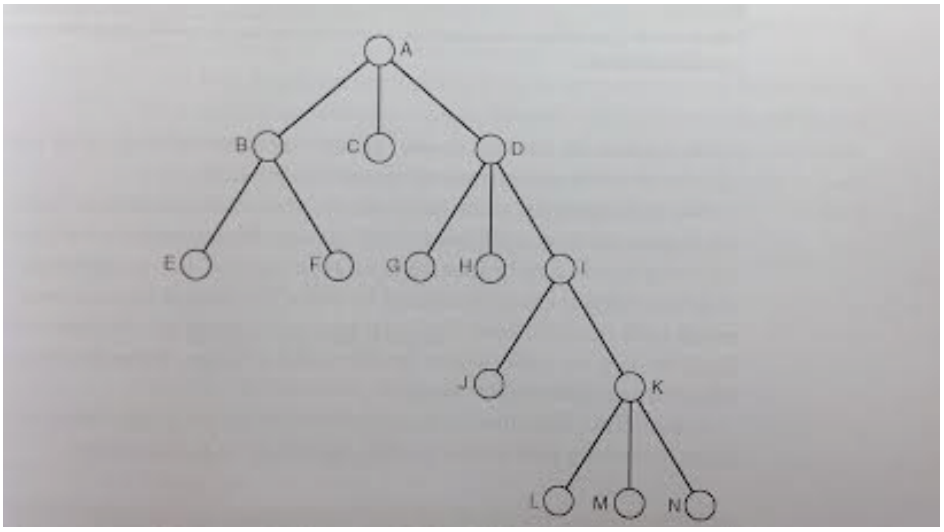
Um método alternativo que muitas pessoas conhecem para percorrer um labirinto é começar com a sua mão pelo lado esquerdo do labirinto (ou pelo lado direito, se preferir) e seguir o labirinto de um lado ao outro, sempre indo com a sua mão esquerda no lado esquerdo da parede do labirinto. Desse modo você terá a garantia de encontrar a saída. Como pode ser visto na figura, isto se deve a essa técnica corresponder exatamente à busca em profundidade.

Na figura, 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 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



Você deve ser capaz de observar que ao seguir a árvore de busca utilizando busca em profundidade faz-se o mesmo caminho. Isso só acontece porque os nós da árvore de busca foram ordenados corretamente. A ordem foi escolhida para que cada nó tenha primeiro o seu filho mais à esquerda e por último o seu filho mais à direita. Utilizar uma ordem diferente, levaria a busca em profundidade a seguir um caminho diferente pelo labirinto.

4.9.2 EXEMPLO 2: Comprando um Presente

Quando procuramos por um presente para uma pessoa querida em várias lojas, cada uma com vários andares e cada andar com vários departamentos, busca em profundidade seria a abordagem natural, se não a mais simples.

Isso envolveria visitar cada andar no primeiro edifício antes de ir para o próximo edifício. Uma abordagem em largura significaria examinar o primeiro departamento em cada loja e depois voltar para examinar o segundo departamento em cada loja, e assim por diante. Esse caminho não faz sentido devido à relação espacial entre os departamentos, andares e lojas. Para um computador, qualquer uma das abordagens trabalharia igualmente bem desde que, na representação utilizada, ir de um edifício ao outro não gastasse tempo de computação.

Nos dois exemplos anteriores pode ser observado que utilizar busca em largura, ainda que seja uma abordagem perfeitamente razoável para um computador, seria bem estranha para um humano. Isso provavelmente deve-se à busca em profundidade ter como abordagem explorar totalmente cada caminho antes de mudar para outro caminho, enquanto na busca em largura, a abordagem envolve voltar a visitar e expandir determinados caminhos muitas vezes.

Apesar disso, implementações em software de ambos os algoritmos são muito parecidas, pelo menos quando expressas em pseudocódigo.

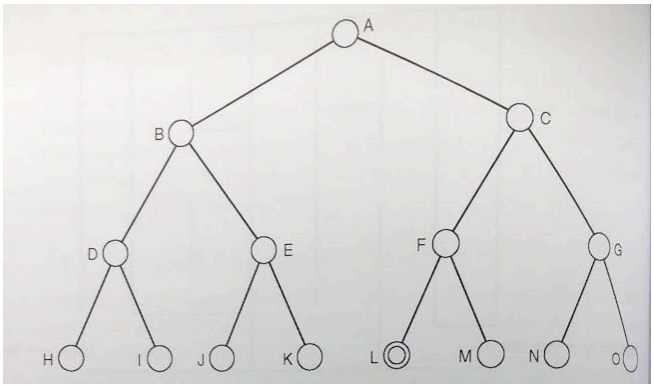
4.10 IMPLEMENTANDO BUSCA EM PROFUNDIDADE E BUSCA EM LARGURA

Uma implementação em pseudocódigo da busca em profundidade é demonstrada a seguir.

A variável `estado` representa o estado corrente em qualquer ponto dado no algoritmo e `fila` é uma estrutura de dados que armazena os estados, de forma que permita inserção e remoção em cada extremidade. Neste algoritmo, sempre inserimos pela frente e removemos pela frente, o que, como veremos depois, significa que a busca em profundidade pode ser facilmente implementada utilizando uma pilha.

Nessa implementação, utilizamos a função `sucessores(estado)` que simplesmente retorna todos os sucessores de um dado estado.

```
Function profundidade (){
   fila = [ ]; // inicializa uma fila vazia
   estado = noRaiz; // inicializa o estado
inicial
   while (true){
   if (eh_objetivo(estado)){
      return SUCESSO;
   } else {
     inserirFrenteFila(sucessores(estado));
   }
   if fila == [ ] {
         report FALHA;
   }
   estado = fila[0]; //estado =primeiro fila
   remover_primeiro_item_da(fila);
}
```



A tabela abaixo mostra os estados que as variáveis `fila` e `estado` assumem quando se executa o algoritmo de busca em profundidade em uma simples árvore de busca.

PASSO	ESTADO	LISTA	OBS
1	A	vazia	A fila começa vazia e o estado inicial é o nó raiz
2	A	B, C	Os filhos de A são inseridos na fila
3	B	D, E, C	Os filhos de B são inseridos no início da lista
4	D	H, I, E, C	
5	H	I, E, C	H é um nó folha, não possui filhos
6	I	E, C	Da mesma forma, I não tem sucessores
7	E	J, K, C	Os filhos de E são inseridos na lista
8	J	K, C	
9	K	C	Fomos até a última folha, e retrocedemos a CC
10	C	F, G	
11	F	L, M, G	Inserção dos filhos de F
12	L	M, G	SUCESSO

De fato, busca em profundidade pode ser prontamente implementada na maioria dos sistemas computacionais utilizando-se uma **pilha**, que é simplesmente uma fila "último a entrar, primeiro a sair" (às vezes chamada de LIFO – Last In, First Out). Dessa forma, uma versão recursiva do algoritmo dado anteriormente pode ser utilizada como a seguir. Como esta função é recursiva, ela precisa ser chamada com um argumento:

```
profundidade_recursiva(no_raiz);
```

A função é definida a seguir:

```
Function profundidade_recursiva (estado){
   if (eh_objetivo(estado)){
         return SUCESSO;
   } else {
         remover_da_pilha (estado);
         inserir_na_pilha (sucessores(estado));
   }
   while (pilha != [ ]){
         if (profundidade_recursiva (pilha [0]) == SUCESSO){
               return SUCESSO;
         }
         remover_primeiro_item_da (pilha);
   }
   return FALHA;
}
```

Se você executar este algoritmo no papel (ou em uma linguagem de programação como C), perceberá que ele segue a árvore do mesmo modo que o algoritmo anterior, **profundidade**.

Como disse, busca em profundidade e busca em largura podem ser implementadas de forma bem similar. A seguir temos um pseudocódigo de uma implementação não recursiva de busca em largura, que deve ser comparada com a implementação de busca em profundidade anterior:

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

Observe que a única diferença entre profundidade e largura é que onde **profundidade** insere estados sucessores no **início da fila**, **largura** os insere no **final da fila**. Então, quando aplicado à árvore de busca da figura a seguir, largura seguirá um caminho bem diferente de profundidade, como é demonstrado na próxima tabela.

PASSO	ESTADO	LISTA	OBS
1	A	vazia	A fila começa vazia e o estado inicial é o nó raiz
2	A	B, C	Os filhos de A são inseridos na fila
3	B	C, D, E	Os filhos de B são inseridos no final da lista
4	C	D, E, F, G	Os filhos de C são inseridos
5	D	E, F, G, H, I	Inserção dos filhos de D
6	E	F, G, H, I, J, K	Inserção dos filhos de E
7	F	G, H, I, J, K, L, M	Os filhos de F são inseridos na lista
8	G	H, I, J, K, L, M, N, O	Inserção dos filhos de G
9	H	I, J, K, L, M, N, O	
10	I	J, K, L, M, N, O	
11	J	K, L, M, N, O	
12	K	L, M, N, O	
13	L	M, N, O	SUCESSO

Você observará que, neste caso particular, busca em profundidade encontrou o objetivo em alguns passos a menos que a busca em largura. Como foi sugerido, 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.

Outro fator importante a observar é que a fila fica bem maior quando se utiliza busca em largura. Para grandes árvores e, especialmente para árvores com elevados fatores de ramificação, isso pode fazer uma diferença significativa devido ao algoritmo de busca em profundidade nunca exigir uma fila maior que a profundidade máxima da árvore, ao passo que busca em largura, no pior caso, precisará de uma fila igual ao número de nós no nível da árvore com maior número de nós (oito em uma árvore de profundidade três, com fator de

ramificação de dois, corno na figura anterior). Dai, dizemos que busca em profundidade é geralmente mais eficiente no uso de memória que a busca em largura.

Como vimos, entretanto, **busca em profundidade não é ótima e nem completa**, enquanto **busca em largura é ótima e completa**. Isso quer dizer que busca em profundidade pode não encontrar a melhor solução e, de fato, pode nem mesmo encontrar uma solução. Em contraste, busca em largura sempre encontrará a melhor solução.

4.11 BUSCA EM PROFUNDIDADE COM APROFUNDAMENTO INTERATIVO

Busca em Profundidade com Aprofundamento Iterativo, ou BPAI (também chamada de Busca com Aprofundamento Iterativo ou BAI), é uma técnica de busca exaustiva que combina as buscas em profundidade e em largura. O algoritmo BPAI envolve repetidamente conduzir buscas em profundidade na árvore, começando por uma busca em profundidade limitada a uma profundidade de um, depois uma busca com profundidade dois, e assim em diante, até que um nó objetivo seja encontrado.

Este é um algoritmo que parece ser um pouco pródigo em termos de número de passos que são requeridos para encontrar uma solução. Entretanto, ele tem a vantagem de combinar a eficiência do uso de memória da busca em profundidade com a vantagem de que ramos da árvore de busca que sejam infinitos ou extremamente longos não comprometerão a busca.

E também compartilha a vantagem da busca em largura, que sempre encontrará o caminho que envolva o menor número de passos pela árvore (embora não necessariamente o melhor caminho).

Apesar de parecer que BPAI seria um modo extremamente ineficiente para realizar busca em uma árvore, acontece de essa busca ser quase tão eficiente quanto as buscas em profundidade e em largura. Isto pode ser visto pelo fato de que, para muitas árvores, a maioria dos nós está no nível mais profundo, significando que todas as três abordagens gastam a maior parte do seu tempo examinando esses nós.

Para uma árvore de profundidade ***d*** e com fator de ramificação de ***b***, o número total de nós é
1 nó raiz
b nós da primeira camada
$\mathbf{b^2}$ nós da segunda camada
...
$\mathbf{b^n}$ nós na camada n

Assim, o número total de nós é $\mathbf{1 + b + b^2 + b^3 + ... + b^d}$ que é uma progressão geométrica igual a
$\mathbf{(1 - b^{d+1}) / (1 - b)}$

Por exemplo, para uma árvore de profundidade de 2, com fator de ramificação de 2, temos
(1 — 8) / (1 – 2) = 7 nós. Utilizando busca em profundidade ou em largura, isso significa que o número total de nós a serem examinados é de sete.

Utilizando BPAI, os nós devem ser examinados mais de uma vez, resultando na seguinte progressão:
(d + 1) + b(d) + b2 (d —1) + b3 (d — 2) + ... + bd

Em consequência, BPAI tem complexidade de tempo de $O(b^d)$. Ele tem a eficiência de memória da busca em profundidade, pois só precisa armazenar informação sobre o atual caminho. Portanto, sua complexidade de espaço é de O(bd).

No caso da árvore com profundidade de 2 e fator de ramificação de 2 isso significa examinar o seguinte número de nós: (3 + 1) + 3 X 2 + 4 X 2 = 18.

Portanto, para uma árvore pequena, BPAI é bem mais ineficiente em tempo que as buscas em profundidade e em largura. Entretanto, se compararmos o tempo necessário para uma árvore maior, com profundidade de 4 e fator de ramificação de 10, a árvore tem o seguinte número de nós: $1 — 10^5 / 1 — 10 = 11.111$ nós.

BPAI examinará o seguinte número de nós: (4 + 1) + 10 X 4 + 100 X 3 + 1.000 X 2 + 10.000 = 12.345 nós. Então, conforme a árvore cresce, observamos que a maioria dos nós a serem examinados (neste caso, 10.000 em 12.345) está na última linha, que precisa ser examinada apenas uma vez em qualquer caso.

Assim como busca em largura, BPAI é ótima e completa. Devido a possuir também boa eficiência de espaço, é um método de busca muito bom para ser utilizado quando o espaço de busca pode ser muito grande e a profundidade do nó objetivo não for conhecida.

4.12 USANDO HEURÍSTICAS NA BUSCA

Busca em profundidade e em largura foram descritas como métodos de busca de força bruta. Isso porque eles não empregam qualquer conhecimento especial sobre as árvores de busca que estão examinando, mas simplesmente examinam cada nó, em ordem, até que aconteça de encontrar o objetivo. Isso pode ser parecido com o ser humano que está percorrendo um labirinto, seguindo com a mão pelo lado esquerdo da parede do labirinto.

Em alguns casos, isto é o melhor que pode ser feito, pois não há informação adicional disponível que possa ser utilizada para direcionar a busca de maneira melhor.

Frequentemente, entretanto, essa informação existe e pode ser utilizada. Observe o exemplo de procura por um presente adequado. Bem, poucas pessoas simplesmente entrariam de loja em loja, indo em cada departamento sucessivamente, até topar com um presente. A maioria das pessoas seguiria diretamente para a loja que considerasse a mais provável de ter um presente adequado. Se nenhum presente fosse encontrado naquela loja, seguiria para a próxima loja que considerasse ser a mais provável de ter um presente adequado.

Esse tipo de informação é chamado de **heurística** e humanos a utilizam o tempo todo para solucionar todo tipo de problema. Computadores também podem utilizar heurísticas e, em muitos problemas, heurísticas podem tornar relativamente simples um problema, que de outra forma seria impossível.

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. Para dois nós, ***m*** e ***n***, e uma função heurística ***f***, se ***f(m)*** < ***f(n)***, então deve ser o caso que ***m*** é mais provável de estar em um caminho ótimo para o nó objetivo do que ***n***. Em outras palavras, quanto mais baixo o valor heurístico de um nó, mais provável é que ele esteja em um ótimo caminho para um objetivo e é mais sensato para um método de busca examinar aquele nó.

A seguir veremos detalhes de vários métodos de busca que utilizam heurística e, desse modo, são vistos como **métodos de busca heurística** ou **métodos de busca heuristicamente informados**.

Tipicamente, a heurística utilizada em busca é aquela que forneça uma estimativa de distância de qualquer nó dado até um nó objetivo. Essa estimativa pode ser ou não exata, mas deve ao menos fornecer resultados melhores que mera suposição.

4.12.1 MÉTODOS INFORMADOS E NÃO INFORMADOS

Um método de busca ou uma heurística 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 é **não informado** ou **cego**. Em outras palavras, métodos de busca que usam heurísticas são informados e aqueles que não usam são cegos.

A busca pelo primeiro melhor é um exemplo de busca informada, ao passo que busca em largura e busca em profundidade são não informadas ou cegas.

Uma heurística ***h*** é dita ser **mais informada** que outra heurística, ***j***, se ***h(nó) >= j(nó)*** para todos os nós no espaço de busca. (Na verdade, de modo a que ***h*** seja mais informada que ***j***, deve haver algum nó onde ***h(nó)*** > ***j(nó)***. Caso contrário, elas são igualmente informadas.)

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

4.12.2 ESCOLHENDO UMA BOA HEURÍSTICA

Algumas heurísticas são melhores que outras e, quanto melhor (mais informada) a heurística for, menos nós ela precisará examinar na árvore de busca para encontrar uma solução. Assim, da mesma forma que optar por uma representação correta, escolher a heurística correta Pode fazer uma diferença significativa na nossa capacidade de solucionar um problema.

Ao escolher heurísticas, geralmente consideramos que uma heurística que reduza o número de nós que precisam ser examinados na árvore de busca seja uma boa heurística. É também importante considerar a efici-

ência em executar a heurística propriamente dita. Em as palavras, se é gasta uma hora para computar um valor heurístico para um dado estado, o fato de que, assim, poupam-se uns poucos minutos do tempo total de busca, é irrelevante. Assumiremos que as funções heurísticas escolhidas sejam extremamente simples de calcular e, portanto, não interfiram na eficiência geral do algoritmo de busca.

4.12.3 EXEMPLO: O Quebra-Cabeça 8

Para ilustrar como heurísticas são desenvolvidas, utilizaremos o quebra-cabeça 8, como ilustrado na figura a seguir.

O quebra-cabeça consiste em uma grade 3 X 3, com os números de 1 a 8 em peças dentro da grade e um espaço vazio. Peças podem ser deslizadas dentro da grade, mas uma peça só pode ser deslocada para o espaço vazio se for adjacente a ele. O estado inicial do quebra-cabeça é uma configuração aleatória e o estado objetivo é como mostrado na segunda imagem na figura, na qual os números vão de 1 a 8, em torno do espaço vazio, no sentido horário, com 1 no canto superior esquerdo.

7	6	
4	3	1
2	5	8

1	2	3
8		4
7	6	5

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, o fator de ramificação será de 3, e se ele estiver em uma quina, o fator de ramificação será de 2. Assim, o fator médio de ramificação da árvore de busca é de 3.

Então, uma busca exaustiva na árvore de busca precisaria examinar cerca de 320 estados, o que dá em torno de 3,5 bilhões. Devido a existirem apenas 9! ou 362.880 possíveis estados, a árvore de busca pode certamente ser muito reduzida evitando-se estados repetidos.

É útil encontrar modos de reduzir ainda mais a árvore de busca, a fim de delinear um modo de solucionar o problema eficientemente. Uma heurística ajudaria a fazer isso, nos informando aproximadamente quantos deslocamentos um dado estado dista do estado objetivo. Examinaremos diversas heurísticas possíveis aplicáveis ao quebra-cabeça 8.

Para ser útil, nossa heurística nunca deve superestimar o custo de ir um dado estado até um estado objetivo. Uma heurística assim é definida como **admissível**. Como veremos, para muitos métodos de busca é essencial que as heurísticas utilizadas sejam admissíveis.

A primeira heurística a considerar é contar quantas peças estão no lugar errado. Chamaremos esta heurística de h_1(nó). No caso do primeiro estado exibido na figura, h_1(nó) = 8, pois todas as peças estão fora de posição. Entretanto, isto confunde, já que seria possível imaginar um estado com valor heurístico de 8, mas no qual cada peça fosse para sua posição correta em um deslocamento. Esta heurística é certamente admissível, já que, se uma peça estiver na posição errada, ela deverá ser deslocada pelo menos uma vez.

Uma heurística aprimorada, h_2 leva em conta quão longe cada peça deve ser deslocada para ir para o seu estado correto. Isso é conseguido 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, assim chamada por causa das vias utilizadas em Manhattan).

Para o primeiro estado, esta heurística forneceria um valor de h_2(nó) = 2 + 2 + 2 + 2 + 3 + 3 + 1 + 3 = 18. Obviamente, esta ainda é uma heurística admissível, pois, de modo a solucionar o quebra-cabeça, cada peça deve ser deslocada um quadrado por vez, a partir de sua posição inicial até sua posição no estado objetivo.

Vale notar que $h_2(nó) >= h_1(nó)$ para qualquer nó. Isto significa que h_2 **domina** h_1 o que quer dizer que um método de busca utilizando a heurística h_2 sempre terá desempenho mais eficiente que o mesmo método de busca utilizando h_1. Isto é devido a h_2, ser mais informada que h_1. Apesar de uma heurística nunca dever superestimar o custo, é sempre melhor escolher a heurística que forneça a maior subestimativa de custo possível. A heurística ideal seria, assim, aquela que fornecesse exatamente custos corretos a cada vez.

Esta eficiência é mais bem entendida em termos de **fator efetivo de ramificação**, b*, de uma busca.

Se um método de busca expandir ***n*** nós ao solucionar um problema específico e o nó objetivo estiver a uma profundidade ***d***, então ***b**** será o fator de ramificação de uma **árvore unifo**rme que contenha ***n*** nós. Heurísticas que forneçam um menor fator efetivo de ramificação tem melhor desempenho. Ao solucionar o quebra-cabeça 8, a execução de um método de busca com h_2 terá um menor fator efetivo de ramificação que a execução do mesmo método com h_1.

Outro modo de encontrar uma heurística é tirar proveito de características do problema que está sendo modelado pela árvore de busca. Por exemplo, no caso do jogo de damas, computadores são capazes de usar heurísticas tais como o fato de um jogador ter mais damas no tabuleiro que dá a ele mais chances de ganhar de um jogador que tenha menos damas.

4.12.4 MONOTONICIDADE

Um método de busca é descrito como **monótono** se ele sempre chega a um dado nó pelo caminho mais curto possível.

Então, um método de busca que chegue a um dado nó em diferentes profundidades de uma árvore de busca não é monótono. Um método de busca monótono deve ser admissível, desde que haja apenas um estado objetivo.

Uma heurística **monotônica** é uma heurística que tenha esta propriedade.

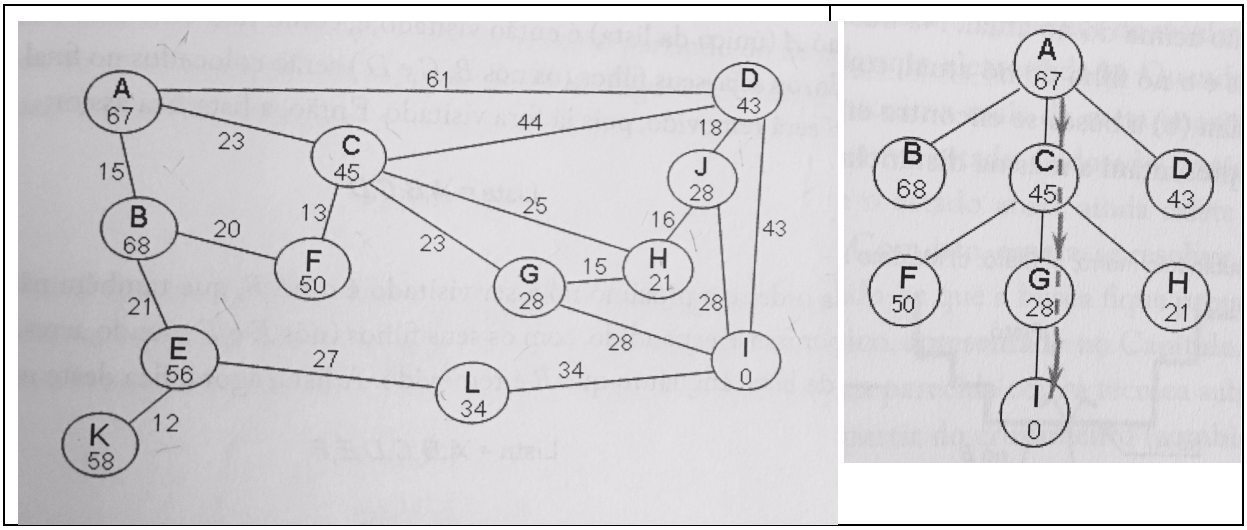
Uma **heurística admissível** é uma heurística que nunca superestime a distância verdadeira entre um nó e o objetivo. Uma heurística monotônica é também admissível, assumindo-se que haja apenas um estado objetivo.

4.13 BUSCA MELHOR ESCOLHA

A **Busca Melhor Escolha** depende do conhecimento que permite estimar um valor relativo à meta (por exemplo, a distância em linha reta de um nó até a meta) e move-se na direção do nó que apresenta um valor melhor que o nó atual. Com isso tenta-se minimizar o número de nós visitados.

Para cada dó gerado, a função heurística de avaliação retorna o valor estimado até a meta, e a busca segue para o primeiro nó filho com valor melhor que o do nó atual.

No caso do grafo ao lado, partindo do nó A, que não é a meta, são colocados na lista os seus nós filhos (B, C e D), juntamente com seus valores de distância (estimada) até a meta.



PASSO	ESTADO	LISTA	OBS
1	A	vazia	A fila começa vazia e o estado inicial é o nó raiz
2	**A67**	B68, C45, D43	Os filhos de A são inseridos na fila
3	B68	C45, D43	B68 é pior que o nó anterior A67

4	**C45**	F50, G28, H21, D43	C45 é melhor que A67. Os filhos de C são inseridos
5	F50	G28, H21, D43	F50 é pior que o nó anterior AC45
6	**G28**	I0, H21, D43	G28 é melhor que C45. Os filhos de G são inseridos
7	**I0**	H21, D43	SUCESSO

Quando a busca se descola sempre para o melhor de todos os filhos de um certo nó, é chamado **Busca Gulosa da melhor escolha**.

4.13 SUBIDA DA COLINA

Subida da colina é um exemplo de um método de busca **informado**, pois ele utiliza informação sobre o espaço de busca de uma maneira razoavelmente eficiente. Se tentar escalar uma montanha, em dia de neblina, com um altímetro, mas sem mapa, você utilizaria uma abordagem de subida da colina do tipo Gerar e Testar:

Verifique a altura a alguns centímetros da sua posição corrente em cada direção: norte, sul, oeste e leste.

Tão logo encontre uma posição que tenha altura maior que a altura de onde você está, vá para lá e recomece o algoritmo.

Se todas as direções levam para mais abaixo de onde você está, então pare e assuma que chegou ao topo. Como veremos mais tarde, isto nem sempre é verdadeiro.

Ao examinar uma árvore de busca, a 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 àquele do nó corrente.

Esta busca tem um processamento parecido com a técnica da melhor escolha, entretanto, não utiliza uma lista para manter o controle sobre os nós visitados, objetivando diminuir a quantidade de memória necessária.

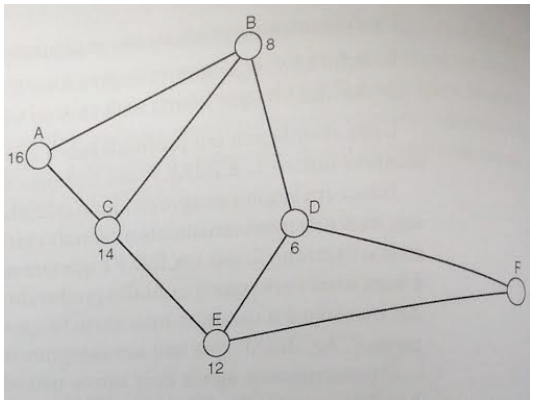
4.13.1 SUBIDA DA COLINA PELA ENCOSTA DE MAIOR ACLIVE

Subida da colina pela encosta de maior aclive é análoga à subida da colina, exceto que, em vez de se mover para a primeira posição encontrada que seja mais alta que a posição corrente, você sempre verifica em volta em todas as quatro direções e escolhe a posição que seja mais alta.

Subida da colina pela encosta de maior aclive também pode ser vista 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 dado.

Para este método, aplicamos uma heurística à árvore de busca mostrada na figura anterior, que é a distância em linha direta de cada cidade à cidade objetivo. Estamos utilizando esta heurística para aproximar a verdadeira distância de cada cidade ao objetivo, que será obviamente maior que a distância em linha direta.

Na figura abaixo, podemos ver o mesmo problema de busca como apresentado anteriormente, mas em vez de registrar os comprimentos das arestas, registramos quão longe (utilizando uma medida em linha direta) cada cidade está do objetivo, a cidade F.




```
Function colina( ){
   fila = [ ]; // inicializa uma fila vazia
   estado = no_raiz; // inicializa o estado
   while(true){
     if (eh_objetivo(estado)){
        return(SUCESSO)
     } else {
        ordenar(sucessores(estado));
        inserirNaFrente(sucessores(estado));
     }
     if fila == [ ]{
        report FALHA;
     }
     estado = fila[0]; // estado= primeiro da fila
     remover_primeiro_item_da(fila);
}
```

Agora a subida da colina segue como a busca em profundidade, mas, a cada passo, os novos nós a serem inseridos na fila são ordenados pela distância ao objetivo. Observe que a única diferença entre essa implementação e aquela dada para a busca em profundidade é que, na subida da colina, os sucessores de um estado são ordenados de acordo com a distância ao objetivo antes de serem inseridos na fila:

PASSO	ESTADO	LISTA	OBS
1	A	vazia	A fila começa vazia e o estado inicial é o nó raiz
2	A16	B8, C14	Os filhos de A são inseridos na fila. B antes de C
3	B8	D6, C14, C14	Os filhos de B são inseridos em ordem
4	D6	F0, E12, C14, C14	C45 é melhor que A67. Os filhos de C são inseridos
5	F0	E12, C14, C14	SUCESSO

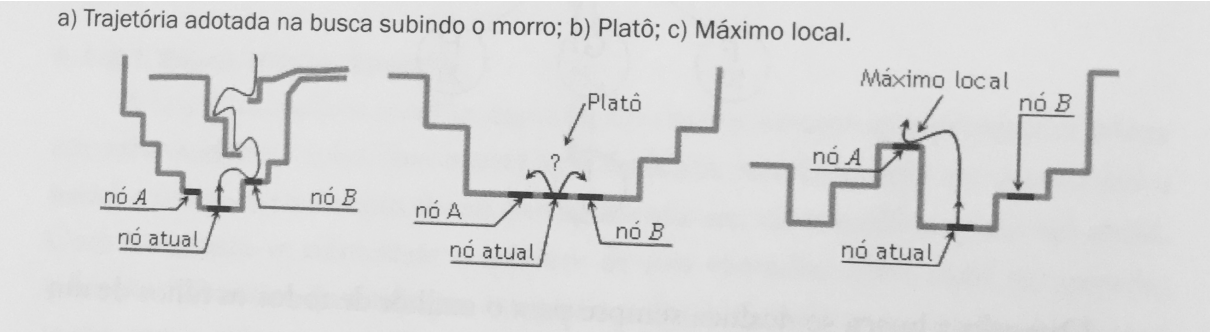
Neste caso, a subida da colina produziu o mesmo caminho que a busca em largura, que é o caminho com menos passos, mas não o menor caminho. Entretanto, em muitos casos, utilizar esta heurística permite à subida da colina identificar caminhos mais curtos do que aqueles que seriam identificados pelas buscas em profundidade ou em largura. A subida da colina utiliza heurísticas para identificar caminhos eficientemente, mas não necessariamente identifica o melhor caminho.

Se executássemos as buscas da direita para a esquerda, em vez de da esquerda para a direita (ou ordenando a árvore de modo oposto), então verificaríamos que a busca em largura produziria um caminho diferente: A,C,E.F (que é de fato o caminho mais curto), mas a subida da colina ainda produziria o mesmo caminho, A,B,D,F. Em outras palavras, a ordenação específica dos nós utilizada afeta o resultado produzido pelas buscas em largura e em profundidade, mas não afeta a subida da colina do mesmo modo. Isto claramente pode ser uma propriedade útil.

4.13.2 CONTRAFORTES, PLATÔS E CRISTAS

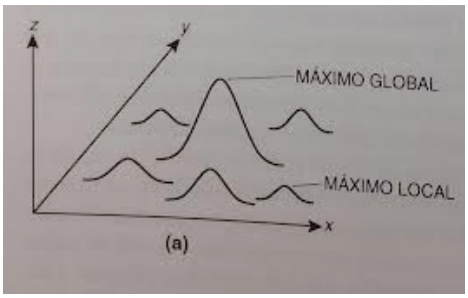
Embora tenhamos falado sobre utilizar técnicas de busca para percorrer árvores de busca, elas também podem ser utilizadas para solucionar problemas de busca que são representados de diferentes modos. Em especial, frequentemente representamos um problema de busca como um espaço tridimensional, no qual os eixos x e y são utilizados para representarem variáveis e o eixo z (ou altura) é utilizado para representar o resultado.

O objetivo é geralmente maximizar o resultado e, então, métodos de busca nestes casos procuram encontrar o ponto mais alto no espaço.

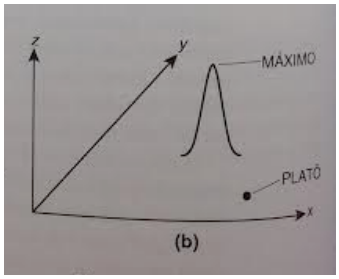


Muitos destes espaços de busca podem ser percorridos com sucesso utilizando a subida da colina e outros métodos de busca heuristicamente informados. Alguns espaços de busca, entretanto, apresentarão dificuldades especificas para estas técnicas.

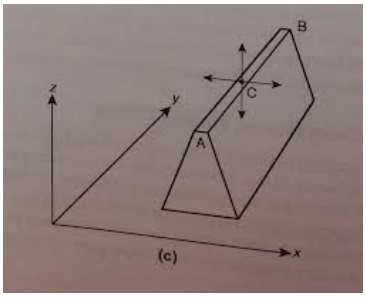
Em especial, a subida da colina pode ser induzida a erro por **contrafortes**, **platôs** e **cristas**. A figura a seguir tem três ilustrações mostrando contrafortes, um platô e uma crista. Esta figura mostra o espaço de busca representado como um terreno tridimensional. Neste tipo de terreno, o objetivo da busca é encontrar os valores de x e y que levam ao maior valor possível de z — em outras palavras, o ponto mais alto do terreno. Esta é uma outra forma de ver a busca tradicional: busca normalmente tem o objetivo de maximizar uma função, que, neste caso, é mostrada como a altura do terreno, mas é tradicionalmente uma função que detalha a distância de um nó ao nó objetivo.



Contrafortes são geralmente chamados de **máximos locais** pelos matemáticos. Um máximo local é uma parte de um espaço de busca que parece ser preferível às partes em torno dele, mas que é na verdade apenas um contraforte ou uma colina maior. Técnicas de subida da colina atingirão este pico, no qual uma técnica mais sofisticada sairia dali à procura do **máximo global**. A figura mostra um espaço de busca que tem um único máximo global cercado por diversos contrafortes ou máximos locais. Muitos métodos de busca atingiriam o topo de um desses contrafortes e, como não há nada mais alto por perto, concluiria que esta seria a melhor solução para o problema.



Um platô é uma região em um espaço de busca na qual todos os valores são os mesmos. Neste caso, embora possa muito bem existir um valor máximo apropriado nas vizinhanças, não há indicação a partir do terreno local de qual direção seguir para encontrá-lo. A subida da colina não tem bom desempenho nesta situação. A figura mostra um espaço de busca que consiste em apenas um pico cercado por um platô. Um método de busca de subida da colina poderia muito bem ficar preso ao platô sem uma clara indicação de para onde seguir para encontrar uma boa solução.



O último problema para a subida da colina é apresentado pelas cristas. Uma crista é uma região longa e estreita de terras altas com terras baixas em ambos os lados. Ao procurar em uma das quatro direções, norte, sul, leste e oeste, a partir da crista, um algoritmo subida da colina determinaria que qualquer ponto no topo da crista fosse um máximo, pois a colina desaparece naquelas quatro direções. A direção correta, que leva ao topo da crista, é muito estreita e identificar esta direção utilizando subida da colina poderia ser muito sutil.

A figura mostra uma crista. O ponto marcado A é inferior ao ponto marcado B, que é o máximo global. Quando um método subida da colina estiver no ponto C, seria difícil para ele encontrar como ir dali para B. As setas no ponto C mostram que ao ir para norte, sul, leste ou oeste o método encontrar-se-ia em um ponto inferior. A direção correta é acima da crista.

4.14 BUSCA TÊMPERA SIMULADA

Outra busca que também não usa listas para encontrar a solução para um problema é a Têmpera Simulada. Diferente da Busca Subida da Colina, que sempre se desloca em direção ao topo e, consequentemente, apresente dificuldades na presença de máximos locais, a Têmpera Simulada gera estados com alguma aleatoriedade.

Quando o novo estado é melhor que o estado atual (se aproxima mais da meta, maximizando ou minimizando a função de avaliação), o novo estado é adotado, porém, mesmo quando o novo estado não é melhor que o estado atual, ainda existe uma probabilidade deste novo estado ser adotado.

4.15 BUSCA MENOR CUSTO OU A*

Explora os nós com o menor custo total, obtido somando o custo exato para ir do **nó inicial até o nó atual** (**função custo**) com o custo estimado para continuar até a meta (**função avaliação**), através de cada nó filho.

Esta busca também sofre o problema dos máximos locais. Assim, os nós que inicialmente parecem bons, podem se revelar ruins; da mesma maneira que nós não muito promissores, em seguida, podem se tornar interessantes.

Durante a busca, os nós são inseridos em uma lista que deve ser ordenada de acordo com o custo total dos nós. Usando o exemplo da figura, tem-se: partindo do no A, que não é a meta, são estimados os custos totais pelos caminhos indo por seus filhos (B, C e D):

Custo de B = 15 + 68 = 83 Custo de C = 23 + 45 = 68 Custo de D = 61 + 43 = 104

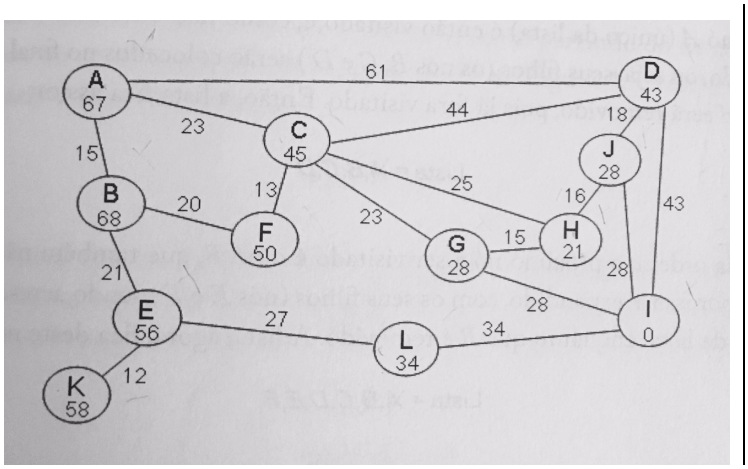
Então vai por C, que tem o menor custo e como não é a meta, vai ser eliminado e expandido. Assim, calcula-se os custos indo pelos filhos de C

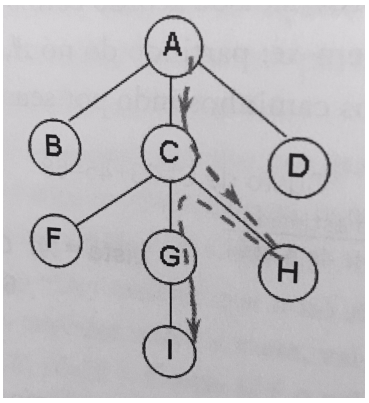
Custo de F = 23 + 13 + 50 = 86 Custo de G = 23 + 23 + 28 = 74 Custo de H = 23 + 25 + 21 = 69

Então, agora segue por H, que tem o menor custo, porém H é um máximo local, pois, se fosse por G, chegaria mais rapidamente à meta. Visitando H, constata-se que não é a meta e, portanto é eliminado e expandido. O filho de H é o nó J.

Custo de J = 23 + 25 + 16 + 28 = 92

Em seguida o nó G é visitado. Como não é a meta, será eliminado e expandido com o seu filho I inserido na lista. O custo de I = 23 + 25 + 16 + 28 + 0 = 74.





PASSO	ESTADO	LISTA	OBS
1	A	vazia	A fila começa vazia e o estado inicial é o nó raiz
2	A67	C68, B83, D104	Os filhos de A são inseridos na fila em ordem
3	C68	H69, G74, B83, F86, D104	Os filhos de C são inseridos em ordem
4	H69	G74, B83, F86, J92, D104	J é o único filho de H e é inserido em ordem
5	G74	I74, B83, F86, J92, D104	O nó I é inserido na fila
6	I74	B83, F86, J92, D104	SUCESSO

Além das distâncias percorridas entre os nós, usadas em todos os exemplos anteriores, o número de conexões também é outra medida que pode ser interessante para o cálculo dos custos. Dependendo do tipo de problema que está sendo resolvido, por exemplo, um problema de voos, pode ser mais interessante diminuir uma conexão do que percorrer uma distância maior por causa dos elevados custos decorrentes do pouso da decolagem necessários para se fazer uma conexão (tempo, combustível, problemas com passageiros, etc.).

Neste caso, devem ser dados valores ao custo de uma conexão e, usando o custo igual a 1 para os arcos:

custo(A) = 67 + 0	custo(D) = 43 + 1	custo(G) = 28 + 0	custo(J) = 28 + 2
custo(B) = 68 + 1	custo(E) = 56 + 2	custo(H) = 21 + 2	
custo(C) = 45 + 1	custo(F) = 50 + 2	custo(I) = 0 + 3	

O primeiro valor é a distância estimada do nó até a meta e o segundo é a quantidade mínima de conexões para chegar ao nó. O segundo valor deve ser multiplicado pelo custo da conexão.

4.16 IDENTIFICANDO CAMINHOS ÓTIMOS

Existem diversos métodos que identificam o **caminho ótimo** em uma árvore de busca. O caminho ótimo é aquele que tem o menor **custo** ou envolve percorrer a distância mais curta do nó inicial ao nó objetivo. As técnicas descritas anteriormente podem encontrar o caminho ótimo por acidente, mas nenhuma delas garante encontrá-lo.

4.16.1 EXEMPLO: O Problema da Mochila

O **problema da mochila** é uma ilustração interessante do uso dos algoritmos de busca gulosa e das suas falhas. O **problema fracional da mochila** pode ser expresso assim:

Um homem está colocando itens na sua mochila. Ele quer levar os itens mais valiosos que ele puder, mas há um limite de quanto peso ele pode colocar na mochila. Cada item tem um peso w_i e vale v_i. Ele pode apenas colocar um peso total de W na sua mochila. Os itens que ele quer levar são coisas que podem ser fragmentadas e ainda reter o valor delas (como farinha e leite) e ele é capaz de levar frações de itens. Assim, o problema é chamado de problema *fracional* da mochila.

Ao solucionar este problema, um algoritmo de busca gulosa fornece a melhor solução. O problema é solucionado calculando-se o valor por unidade de peso de cada item: v_i / w_i e, então, levando tanto quanto for possível do item de maior valor por unidade de peso. Se ainda sobrar espaço, ele vai para o item com o próximo maior valor por unidade de peso e assim por diante.

O **problema da mochila 0-1** é o mesmo que o problema fracionai da mochila, exceto que ele não pode levar partes de itens. Cada item é, assim, algo como um aparelho de televisão ou um notebook, que deve ser levado no todo. Ao solucionar este problema, uma abordagem de busca gulosa não funciona, como pode ser visto pelo seguinte exemplo:

Nosso homem tem uma mochila que pode transportar um total de 100 quilos. Os itens são:
1 barra de ouro de valor $1800 e peso 50 Kg
1 barra de platina de valor $1500 e peso 30 Kg
1 notebook de valor $2000 e peso 50 Kg

Assim, temos quatro itens cujos valores de v e w são como se segue:

v1 = 1800	w1 = 50	v1 / w1 = 36
v2 = 1500	w2 = 30	v2 / w2 = 50
v3 = 2000	w3 = 50	v3 / w3 = 40

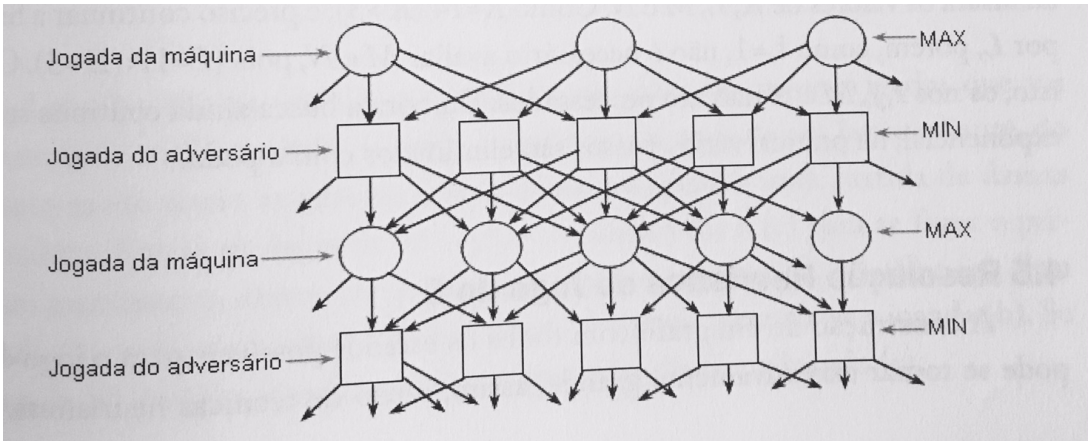
Neste caso, uma estratégia de busca gulosa pegaria primeiramente o item 2 e então pegaria o item 3, dando um peso total de 80 kg e um valor total de $3500. Na verdade, a melhor solução é pegar os itens 1 e 3, deixando o item 2, dando um peso total de 100 Kg e um valor total de $3800.

4.17 BUSCAS COM ADVERSÁRIOS: Min-Max

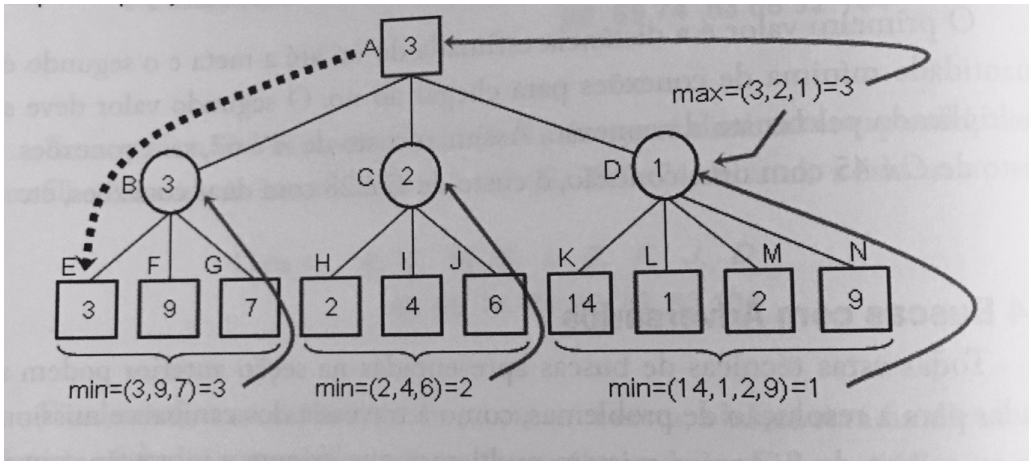
Todas estas técnicas de buscas apresentadas podem ser usadas para a resolução de problemas, como a travessia dos canibais e missionários ou o jogo do 8, porém existem problemas que exigem a interação com um adversário, como é o caso dos jogos de damas e xadrez.

Neste caso, a solução também pode ser representada por um grafo de estados, entretanto, uma nova estratégia precisa ser adotada, pois não basta a máquina encontrar o melhor caminho para as suas jogadas, é preciso que o caminho escolhido atenue os estragos gerados pelas jogadas do seu adversário.

A estratégia mais conhecida neste caso é a técnica **Min-Max**, que procura um caminho pelo grafo que maximiza as jogadas da máquina e ao mesmo tempo minimiza os danos causados pelas jogadas de seu adversário.



A figura a seguir apresenta os estados de um jogo interativo, em que são apresentados os valores associados aos caminhos percorridos na árvore min-max. Os valores obtidos para B = min(E, F, G), C = min(H, I, J), e D = min(K, L, M, N), são, respectivamente, 3, 2 e 1. Em seguida, o valor obtido em A = max(B, C, D), é 3, assim o caminho escolhido fica ABE.



4.18 PODA ALFA-BETA

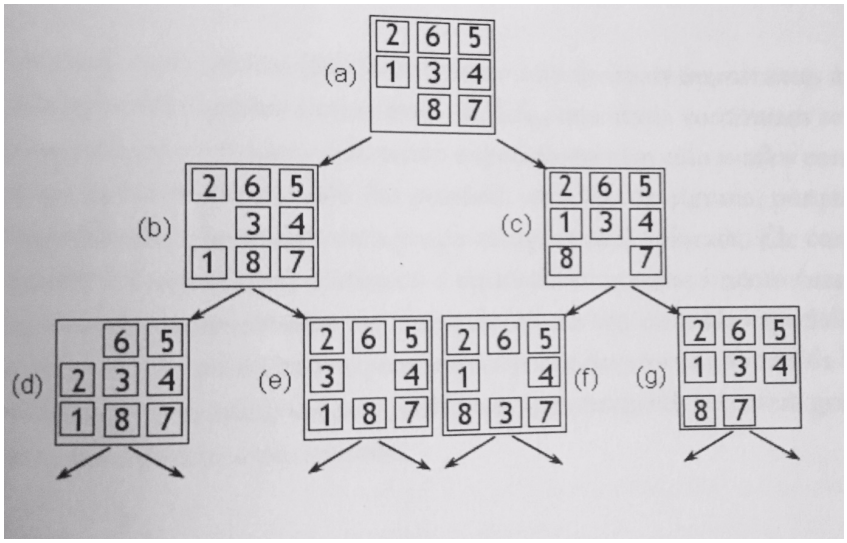
O problema como método min-max é que a quantidade de buscas pode se tornar muito grande. A **poda Alfa-Beta** sugere então diminuir o número de caminhos a serem investigados, descartando caminhos ruins. Usando o mesmo exemplo da figura anterior, a poda alfa-beta realiza o seguinte processamento:

Inicialmente avalia os valores de E, F e G para encontrar o valor mínimo para B, que no caso é 3.

Em seguida, avalia os valores de H, I e J para encontrar o valor mínimo para C, porém quando avalia H, já encontra o valor 2 e como ele é menor do que 3, encontrado para B, não há mais necessidade de avaliar I e J, pois, no próximo passo, a busca irá escolher o valor máximo entre B, C e D.

Logo a seguir, a busca analisará os valores de K, L, M e N. Como K = 14 (k > 3), é preciso continuar a busca por L, porém como L = 1, não é necessário avaliar M e N, pois (L = 1) < (E = 3). Com isto, os nós I, J, M e N não são processados. Embora a busca ainda continue sendo exponencial, na prática, vários passos são eliminados com a poda.

4.19 RESOLUÇÃO HEURÍSTICA DO JOGO DOS 8



A construção de um grafo com todos os estados possíveis para o jogo do 8, pode se tornar excessivamente grande, assim, o uso de técnicas heurísticas que não usam listas e exploram apenas os nós com maior probabilidade de sucesso é o mais indicado.

Existem várias possibilidades para a função de avaliação que vai ser usada para estimar a distância entre um estado e o estado final (solução procurada).

SUGESTÃO 1: A função de avaliação é dada pela soma da quantidade de pedras fora de suas respectivas posições: para (b) tem-se 8, enquanto que para (c) tem-se 9; desta forma, partindo de (a), é mais interessante fazer a jogada (b) do que a jogada (c).

SUGESTÃO 2: A função de avaliação é dada pela soma das diferenças entre o número da pedra e a casa que ela ocupa elevadas ao quadrado; de maneira que para (b) tem-se

$(1-2)^2 + (2-6)^2 + (4-9)^2 + (5-3)^2 + (6-4)^2 + (7-1)^2 + (8-8)^2 + (9-7)^2 = 94$

Enquanto que para (c) tem-se 44. Portanto, partindo de (a), é mais interessante fazer a jogada (c) do que a jogada (b)

Uma heurística mais apurada ainda consiste em mover a pedra que vai apresentar o melhor resultado, após a avaliação de mais de um lance adiante, do mesmo modo como as pessoas fazem quando disputam uma partida de damas ou xadrez.

Então, no lugar de se avaliar os estados (b) e (c) para se fazer o primeiro movimento, devem ser avaliados os quadro estados (d), (e), (f) e (g). Se a jogada (d) ou (e) é a melhor entre as quatro possíveis, executa-se a jogada (b). Se a jogada (f) ou (g) é a melhor entre as quatro, realiza-se a jogada (c).