RESOLUÇÃO DE PROBLEMAS

Extraído de notas de aula do prof. Michel Gagnon

École Polytechnique de Montréal

http://www.professeurs.polymtl.ca/michel.gagnon/Disciplinas/Bac/IA/index_ia.html

---

Conteúdo:

Noções preliminares

Características de problemas:

- Possibilidade de decompor (permite resolver por decomposiçao em problemas menores). O problema das torres de Hanoi é um exemplo de problema decomponível.
- Natureza das etapas da solução:
    - Ignoráveis (demonstração de teorema)
    - Recuperáveis (quebra-cabeça de 8)
    - Irrecuperáveis (xadrez)
- Interação com o ambiente:
    - acesso completo --> estado único
    - acesso parcial --> multiplos estados (para representar os todos estados possíveis, sendo que não temos o conhecimento suficiente para identificar qual é o estado real)
    - ambiente previsível ou não --> problema contingente (exemplo: jogos de cartas, como o bridge)
- Avaliação da solução (absoluta ou relativa).
- Tipo de solução (caminho ótimo / estado desejado / satisfação de restrições).
- Papel do conhecimento (em particular, a quantidade de conhecimento requerida).
- Interação com pessoa.

Representação de um problema como sistema de produção

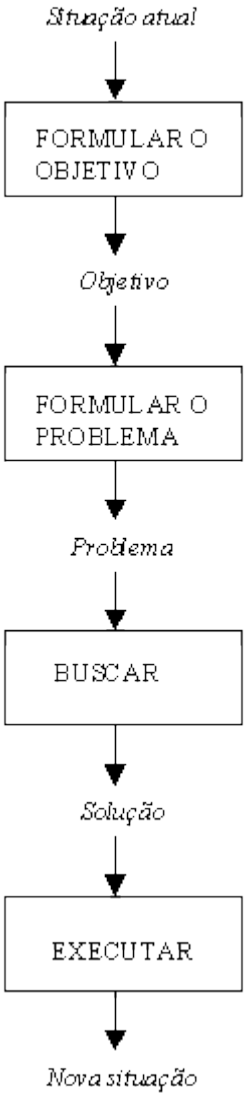
Tipo de problemas:

- Identificação de caminho ótimo
- Estado desejado (objetivo).

Método:

- Avaliação da situação atual
- Formulação do objetivo (conjunto de estados que satisfazem o objetivo)
- Formulação do problema:
    - Decidir quais ações e quais estados têm que ser considerados
    - Abstração (conservar apenas as informações pertinentes)
- Busca:
    - Exame de várias seqüências de ações possíveis
    - Estratégia de contrôle
- Execução da solução escolhida (modifica a situação atual)

Nota: Custo total da resolução do problema = Custo de busca + Custo da execução



Formulação do problema

A formulação do problema contém quatro elementos:

- Estado inicial
- Operadores (definem como passar de um estado a outro estado)
- Teste de sucesso (determina se o estado atual corresponde ao objetivo)
- Função de custo (especifica o custo para cada passo em um caminho)

Nota: O estado inicial e os operadores definem o espaço de busca

Exemplo: quebra-cabeça de 8

Estado inicial:

5	4	
6	1	8
7	3	2

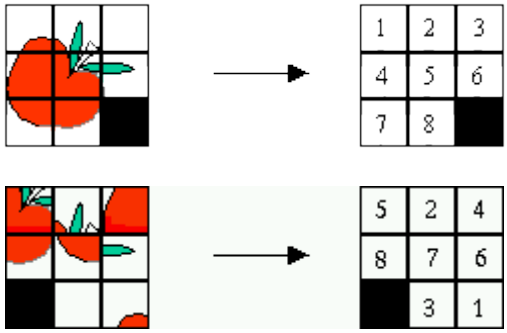
Teste de sucesso: A configuração dos quadrados tem que ser a seguinte:

1	2	3
8		4
7	6	5

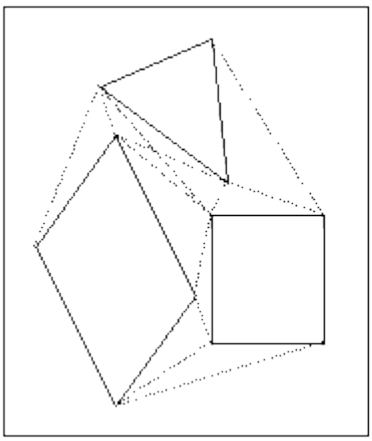
Operadores: Quatro operadores que exprimem as mudanças possíveis do quadrado vazio.

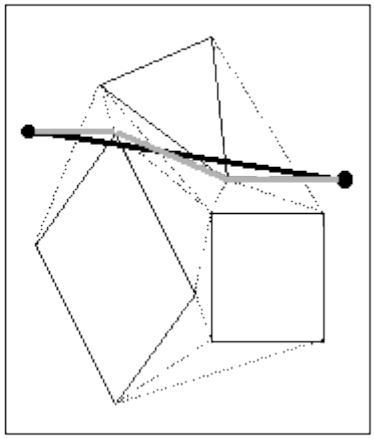
Custo: 1 para cada passo.

Exemplo de abstração : Se o quebra-cabeça representa uma imagem, precisa-se fazer uma abstração. A cada quadrado será associado um número. Com essa abstração, o problema fica equivalente ao outro.



Importância da tarefa de abstração: a abstração pode fazer a diferença entre um problema resolúvel e um problema que não é. Suponhamos o problema que consiste em achar o caminho mais curto entre dois pontos, considerando um conjunto de obstáculos de forma poligonal (veja figura abaixo). Se permitirmos que o caminho possa passar por qualquer lugar (excluindo o espaço dentro dos obstáculos), provavelmente o número de estados a considerar tornará o problema insolúvel. Se na tarefa de abstração aproveitarmos o fato de que o caminho mais curto é necessariamente constituido de linhas diretas que juntam as arestas dos polígonos, o número de estados a considerar será muito menor.





Nota: A formulação do problema depende do estado inicial e do objetivo, pois as possibilidades para os primeiro e último passos no caminho dependem das suas posições relativas com os obstáculos.

Espaço de estados:

O espaço de estado é a árvore de todos os estados que podemos produzir a partir do estado inicial. A busca vai percorrer esse espaço até achar o estado desejado.

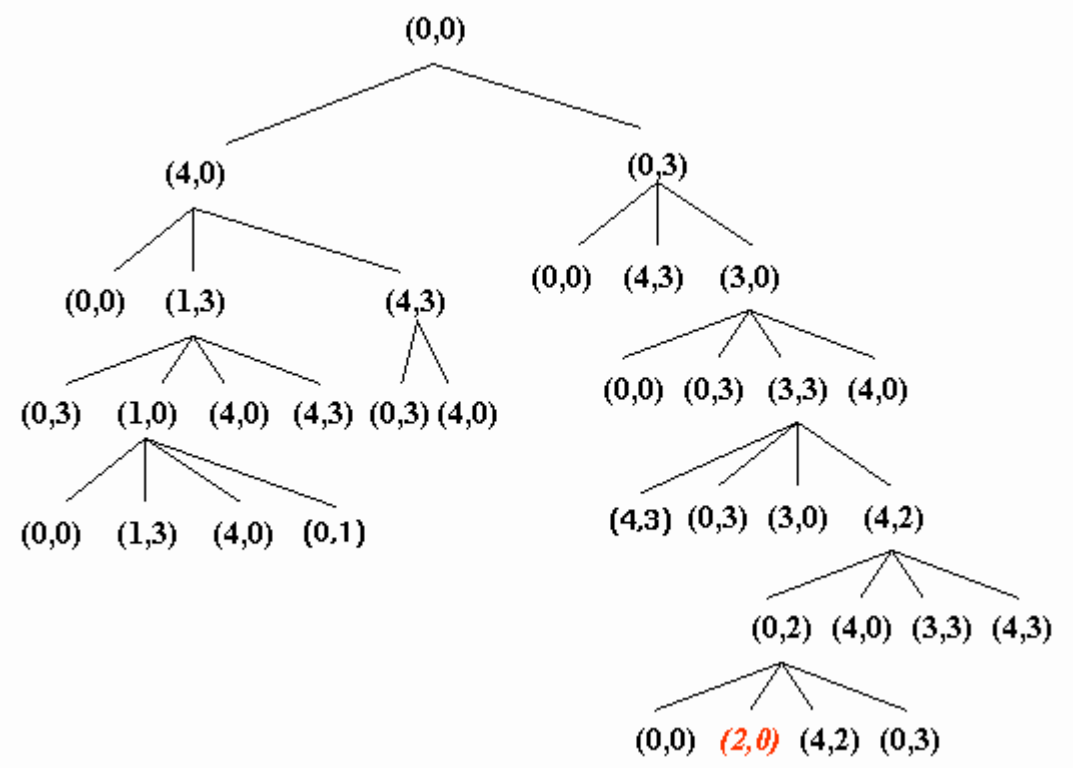
Exemplo: Problema das jarras de água.

Um estado é representado por um par (X,Y), onde X e Y são números que indicam a quantidade de água que contém as jarras de 4 e 3 litros, respetivamente. O estado inicial é (0,0) e o sistema de produção consiste nos seguintes operadores:

1. (X,Y) --> (4,Y) se X < 4
2. (X,Y) --> (X,3) se Y < 3
3. (X,Y) --> (0,Y) se X > 0
4. (X,Y) --> (X,0) se Y > 0
5. (X,Y) --> (X - min(X, 3-Y), min(3, X+Y)) se Y < 3
6. (X,Y) --> (min(4, X+Y), Y - min(4-X, Y)) se X < 4

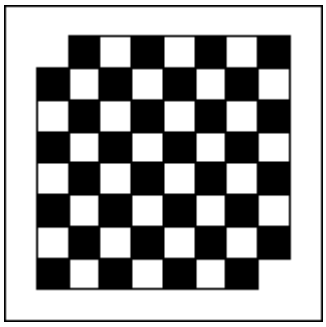
Os dois primeiro operadores representam a ação de encher uma das jarras. Os operadores 3 e 4 representam a ação de esvaziar uma jarra. Finalmente, os dois últimos operadores representam a ação des transvasar (talvez parcialmente) o conteúdo de uma jarra na outra.

O objetivo, nesse problema, é de obter 2 litros de água na jarra de 4 litros. Na nossa representação, isso corresponde ao estado (2,0). Eis uma ilustração de parte do espaço de estados que contém o estado desejado (não mostramos a continuação dos estados repetidos):

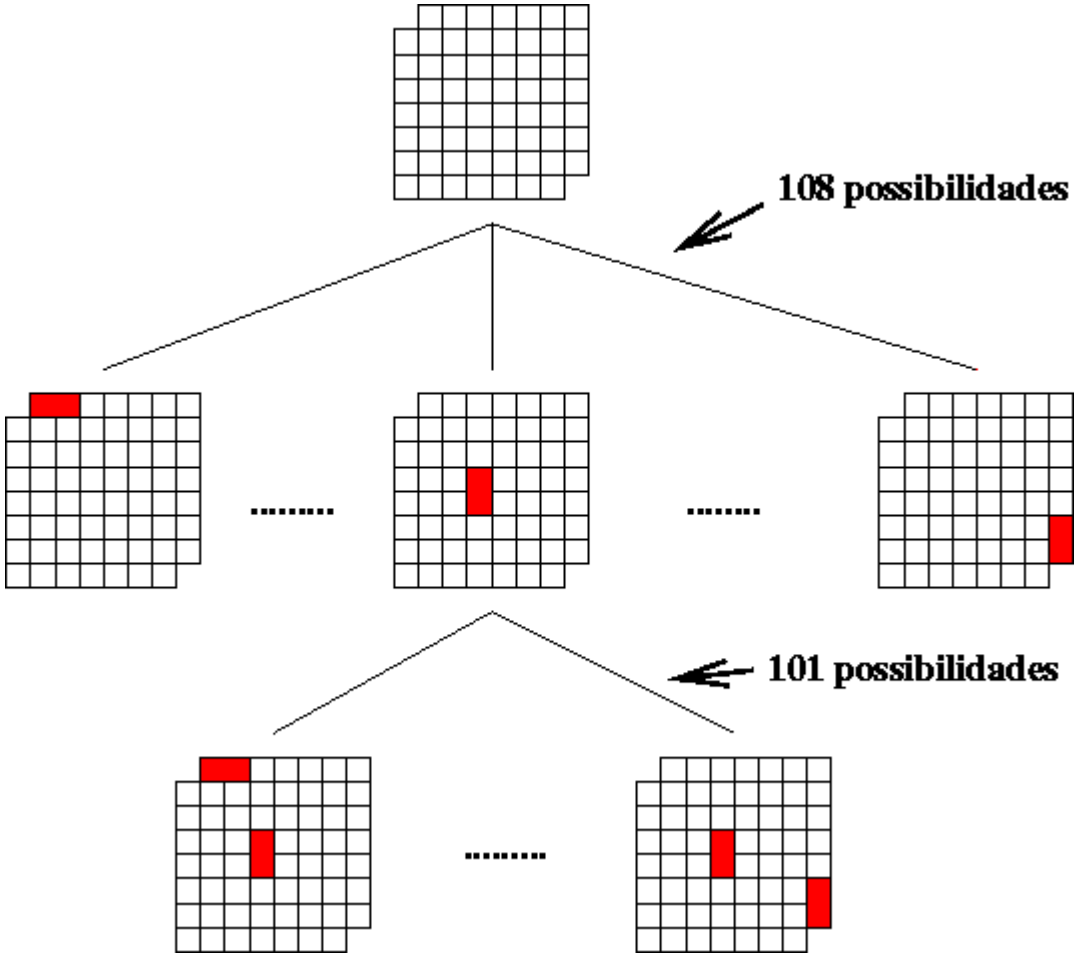


Exemplo: Problema do tabuleiro de xadrez mutilado.

Nesse problema, temos um tabuleiro de jogo de xadrez onde dois quadras em cantos opostos foram cortados:



O problema consiste em cobrir totalmente o tabuleiro com pedras de jogo de dominó, considerando que uma pedra cobre exatamente duas casa contíguas do tabuleiro. A abstração que será feita nesse caso é a de não considerar a cor das casas. Sendo o estado inicial o tabuleiro vazio, tem 108 estados possíveis depois de ter colocado a primeira pedra de dominó. Pode-se verificar que por um tabuleiro de tamanho *n* o número de possibilidades para colocar a primeira pedra é *2n(n-1)-4*. Para colocar a segunda pedra, o número de possibilidade varia entre 101 e 104, dependendo do número de pares de casas contíguas que a pedra impede de usar:



Nesse problema, sabemos que a profundidade da árvore é limitada e podemos identificar o seu valor (não é sempre assim com os outros problemas). A profundidade máxima corresponde a o número de pedras que cabem no tabuleiro, isto é, o número de casas dividido por 2. Nesse caso a profundidade é de 31. Isso pode dar a falsa impressão que o problema será resolvido facilmente. Não é assim por que o número de possibilidades por cada estado é muito alto.

O número de estados atingíveis a partir do nível *n+1* é igual a o número de possibilidades no nível *n* menos as posições que são bloqueadas pela pedra colocada. Sabendo que o número máximo de posições bloqueadas é 7 e chamando P_n e P_{n+1} as possibilidades aos níveis *n* e *n+1*, respectivamente, vamos supor $P_{n+1} = P_n - 7$. Então, no nosso problema, o número total de estados no espaço será: *108 * 101 * 94 ...* = $6{,}84 \times 10^{25}$ = 68400000000000000000000000 estados. Portanto, o problema não é fácil como ele parece.

Mas na abstração que fizemos, perdemos uma informação que torna o problema muito fácil. Considerando o fato que duas casas contíguas são de cores diferentes, podemos ver que uma pedra sembre cobrirá uma casa preta e uma casa branca. Então, para resolver o problema, o número de casas brancas deve ser igual ao número de casas pretas. Estudando bem o tabuleiro mutilado, podemos ver facilmente que as duas casas cortadas são brancas. Como num tabuleiro normal tem o mesmo número de casas brancas e pretas, no nosse tabuleiro tem 2 casas pretas a mais. Portanto, não tem solução para esse problema. Isso é mais um exemplo mostrando que a representação do problema é importante.

Gerar e testar

É a abordagem mais simples para resolver um problema:

1. Gerar uma solução possível.
2. Testar se é realmente uma solução.
3. Se não for uma solução, voltar à etapa 1.

Aplicável só se conseguirmos limitar o espaço de busca.

Busca

Método:

- Geração dinâmica de uma árvore representando os estados alcançáveis a partir de um estado.
- A seleção do estado a ser expandido é determinado pela estratégia de controle.
- O processo pára quando o estado designado por um nodo folha corresponde ao objetivo.

Estruturas necessárias :

Nodo:

- Estado
- Nodo pai
- Operador utilizado para produzir o nodo
- Profundidade
- Custo do caminho até o nodo

Fila:

Contém os nodos produzidos que estão esperando para ser expandidos. Esses nodos constituem a **fronteira** da busca.

Avaliação da estratégia de controle&:

- Completude
- Complexidade (tempo e memória)
- Otimalidade

Algoritmo geral de busca :

nodos <-- CRIAR-FILA(*estado-inicial*)
loop
    se *nodos* é vazio **retorna** falha
    nodo <-- TIRAR-PRIMEIRO(*nodos*)
    se TESTE-SUCESSO(*nodo*) tem sucesso
        retorna *nodo*
    novos-nodos <-- EXPANDIR(*nodo*)
    nodos <-- ACRESCENTAR-NA-FILA(*nodos*,*novos-nodos*)
fim

Nota: A função EXPANDIR retorna uma lista de nodos que representam os estados resultando da aplicação de todos os operadores possíveis ao estado representado por *nodo*.

Exercícios

3.1 Eis a definição de uma cláusula para procurar todos os estados sucessores de um estado, no problema das jarras (Solução):

```
sucessores(Estado,ListaSuc):-
      findall(E,sucessor(Estado,E),ListaSuc).

?- sucessores((3,0),S).
S = [(0,0),(0,3),(3,3),(4,0)]
```

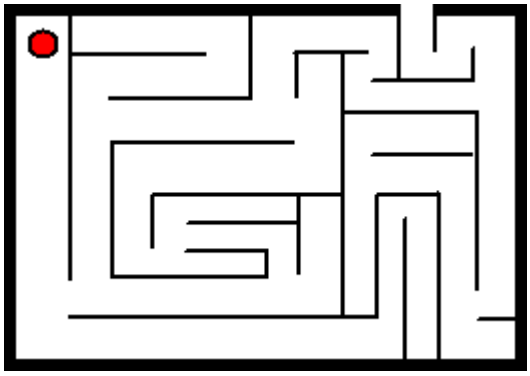
Escreva as cláusulas que definem o predicado `sucessor`.

3.2 Usando o predicado `sucessor` do exercício precedente, escreva um programa Prolog que retorna uma solução ao problema das jarras (Solução):

```
?- solucao((0,0),(2,0),S).
S = [(0,0),(0,3),(4,3),(4,0),(1,3),(1,0),(0,1),(4,1),(2,3),(2,0)]
```

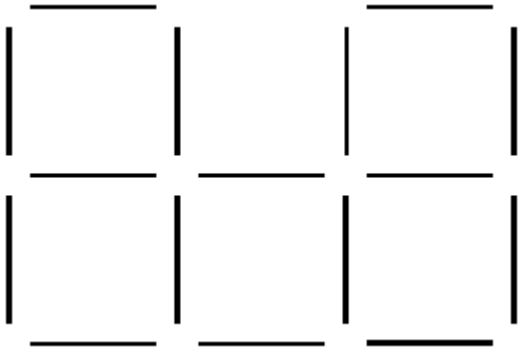
Nota: Como tem mais de uma solução, a sua primeira solução pode ser diferente.

3.3 Queremos achar um caminho para sair desse labirinto (o círculo vermelho representa a posição inicial):

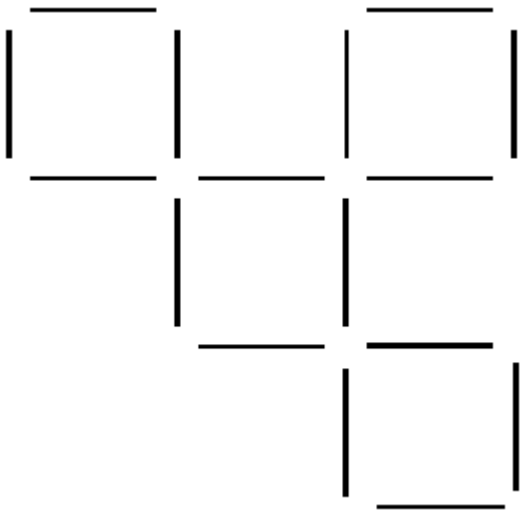


Proponha um sistema de produção que representa o problema. (Tem que identificar como representar os estados, o estado inicial, o teste de sucesso, os operadores para passar de um estado a outro e, finalmente, a função de custo).

3.4 Eis uma figura feita com 16 fósforos:



Queremos, minimizando o número de fósforos deslocados, obter uma figura que contém apenas quatro quadrados. Por exemplo, isso é uma solução com o deslocamento de três fósforos:



Proponha um sistema de produção para representar esse problema.

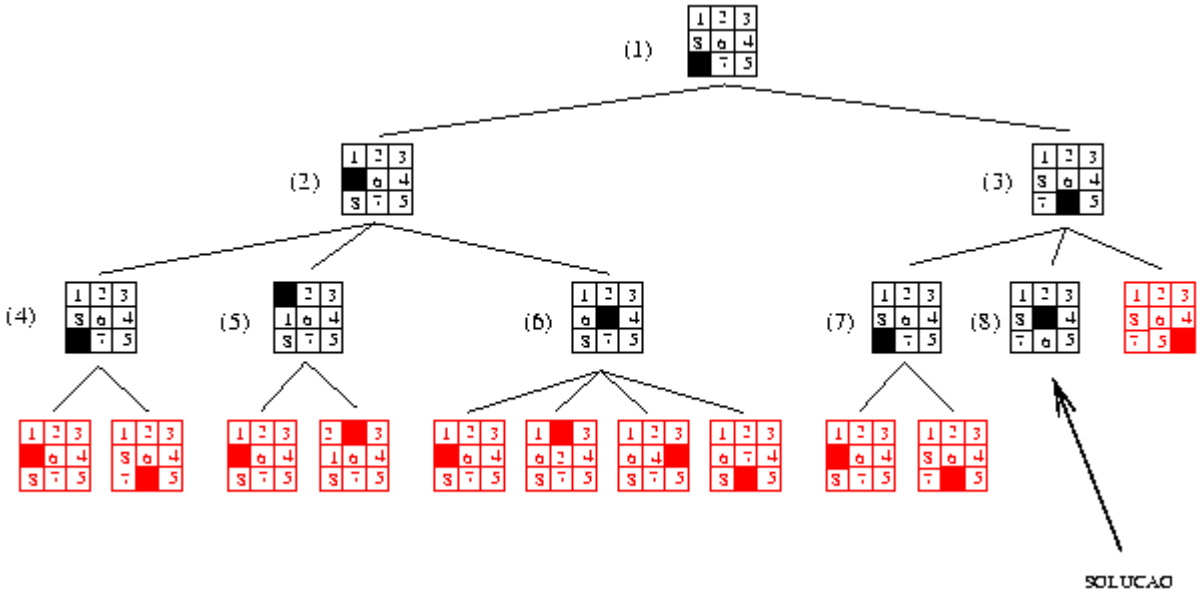
Algoritmos básicos de busca (busca cega)

Busca em largura básica (breadth-first)

Obtemos uma busca em largura se substituimos ACRESCENTAR-NA-FILA por ACRESCENTAR-NO-FIM no algoritmo geral de busca:

nodos <-- CRIAR-FILA(*estado-inicial*)
loop
    se *nodos* é vazio **retorna** falha
    nodo <-- TIRAR-PRIMEIRO(*nodos*)
    se TESTE-SUCESSO(*nodo*) tem sucesso
        retorna *nodo*
    novos-nodos <-- EXPANDIR(*nodo*)
    nodos <-- ACRESCENTAR-NO-FIM(*nodos*,*novos-nodos*)
fim

Exemplo: quebra-cabeça de 8. Eis um exemplo de árvore de busca:



Os estados visitados no momento que se encontra o estado final são indicados em preto. São indicados em vermelho os estados que foram criados mas não visitados ainda. Os números entre parênteses indicam a ordem de percurso dos nodos. É muito importante notar que, por cada estado considerado, se ele falha no teste de sucesso utilizado para identificar o objetivo, os estados sucessores possíveis são criados antes de continuar o percurso.

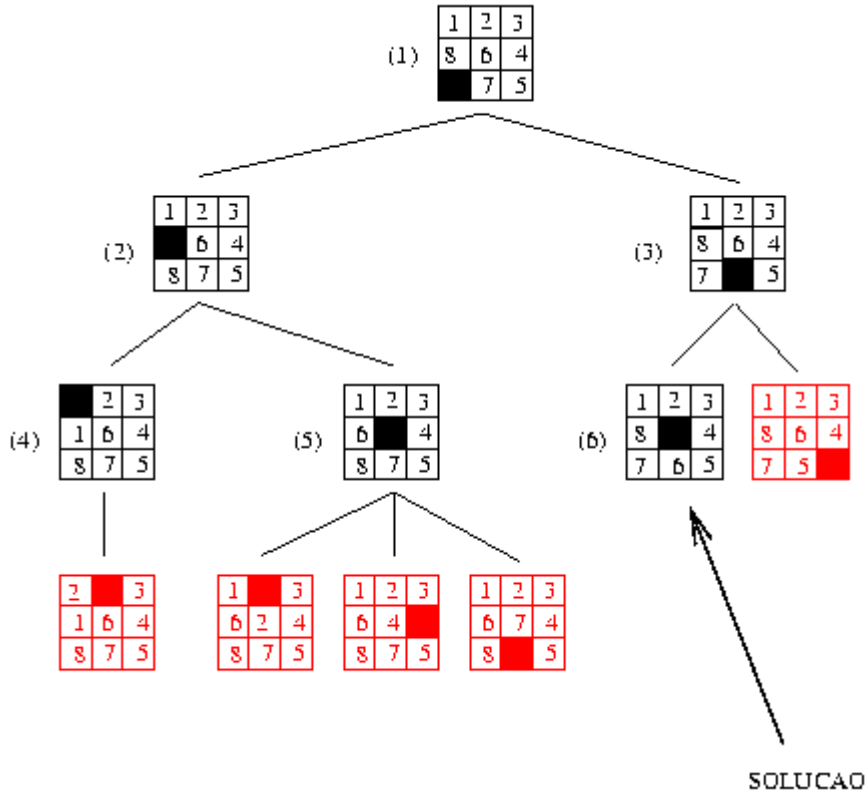
Vantagens desse algoritmo:

Completo
Ótimo, sob certas condições (por exemplo, é ótimo se os operadores sempre têm o mesmo custo).

Desvantagens :

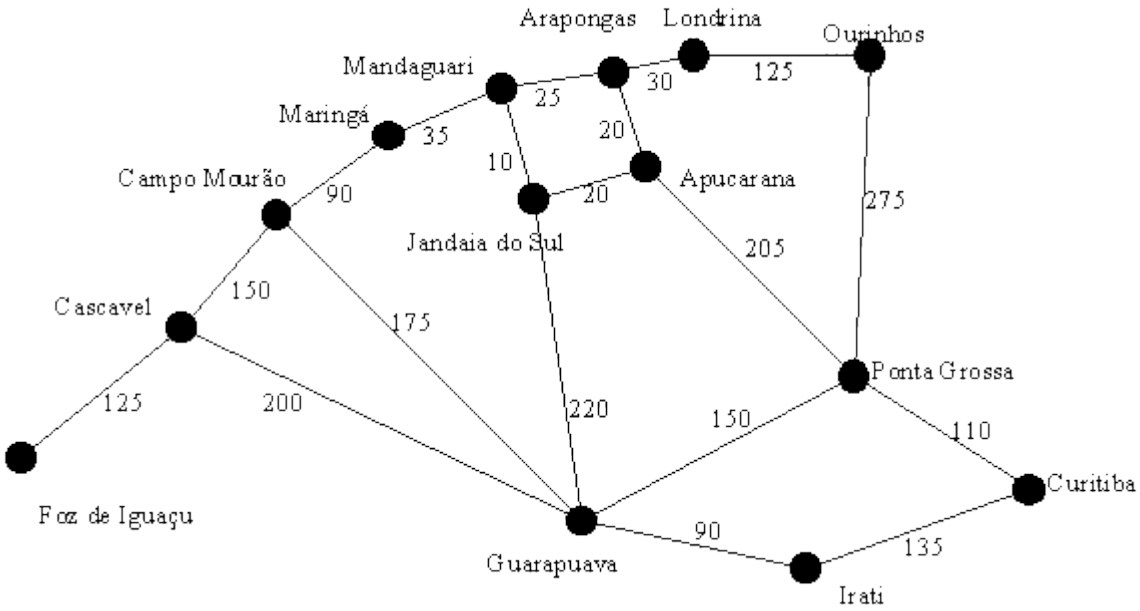
Requer muita memória e tempo (complexidade exponencial): $O(b^p)$ onde b é o fator de ramificação e p a profundidade.

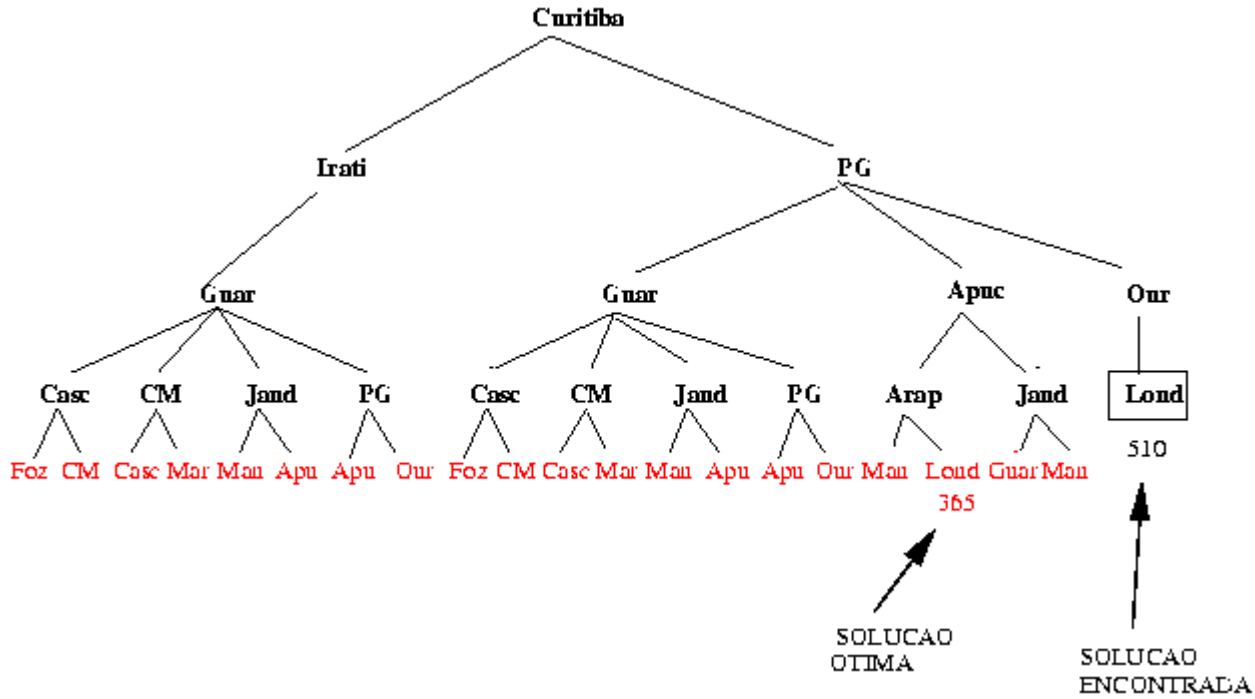
Nota : Se excluirmos os estados que já foram tratados, o algoritmo fica muito mais eficiente:



Nem em todos casos é possível descartar um nodo que aparece de novo no caminho. Mais para diante, veremos uma definição formal das condições que permitem isso.

Exemplo pelo qual a busca em largura não dá uma solução ótima : a busca do caminho mais curto entre Curitiba e Londrina:





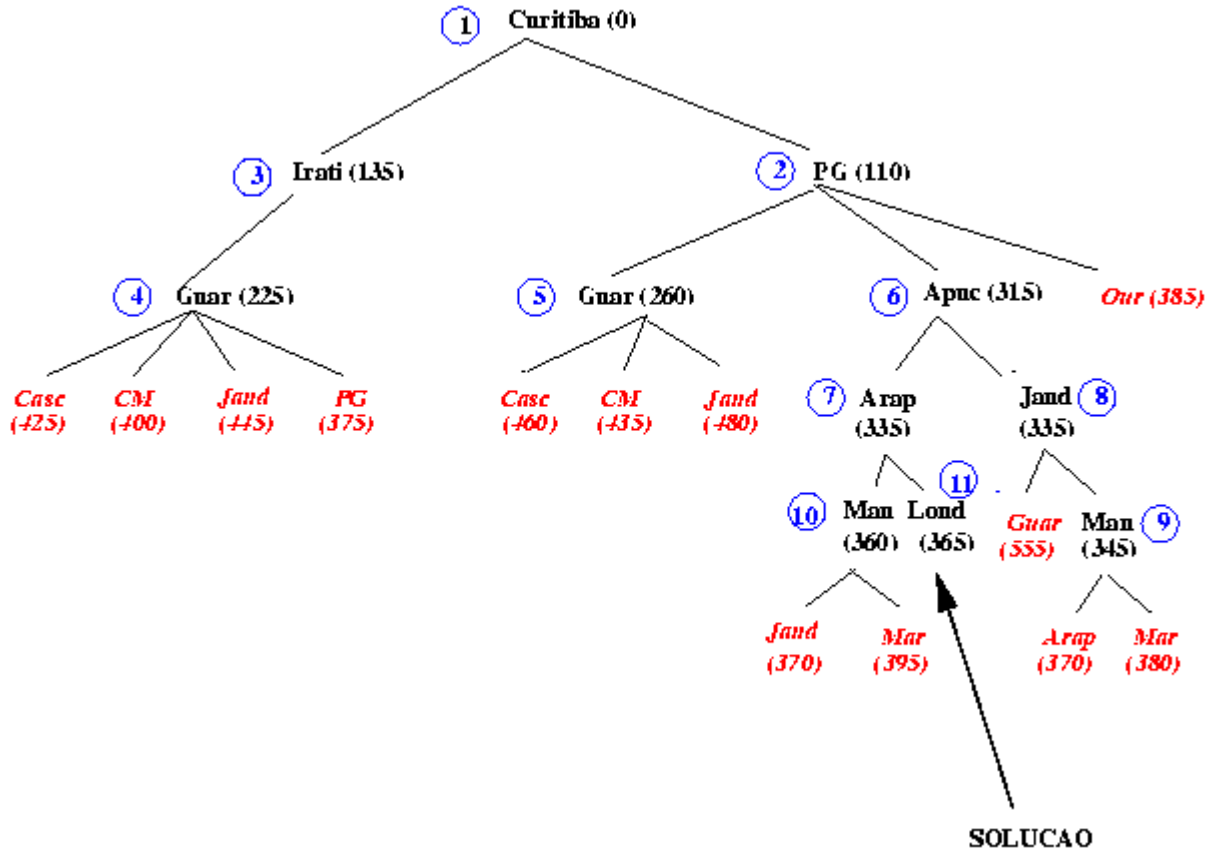
Pode ver que a primeira solução encontrada não é ótima, pois tem, nos nodos esperando para ser visitados, um que tem valor menor. Note que nesse caso eliminamos os nodos redundantes. Podemos fazer isso por que é impossível que o caminho mais curto passe duas vezes pela mesma cidade.

Busca em largura a custo uniforme (Branch-and-Bound)

Modificação do algoritmo de busca em largura para aumentar o conjunto de problemas pelos quais o método retorna uma solução ótima. Ao invés de dar prioridade aos nodos que se encontram no nível menos profundo, o algoritmo escolhe o nodo que tem o menor custo. Agora, a condição para obter uma solução ótima é que o custo para passar ao próximo estado nunca seja negativo.

```
nodos <-- CRIAR-FILA(estado-inicial)
loop
        se nodos é vazio retorna falha
        nodo <-- TIRAR-PRIMEIRO(nodos)
        se TESTE-SUCESSO(nodo) tem sucesso
                retorna nodo
        novos-nodos <-- EXPANDIR(nodo)
        nodos <-- ORDENAR(ACRESCENTAR-NA-FILA(nodos, novos-nodos))
fim
```

Agora, com esse algoritmo de busca, vamos encontrar primeiro o caminho mais curto até Londrina (a ordem de processamento dos nodos é indicada com os números azuis que aparecem dentro de círculos):



Vantagens desse algoritmo:

Completo
Ótimo, se o custo até o próximo nodo nunca é negativo.

Desvantagens :

Complexidade em memória e tempo igual à da busca em largura: $O(b^p)$ onde b é o fator de ramificação e p a profundidade.

Busca em profundidade (depth-first)

Obtemos uma busca em profundidade se substituimos ACRESCENTAR-NO-INICIO por ACRESCENTAR-NO-FIM no algoritmo geral du busca:

nodos <-- CRIAR-FILA(*estado-inicial*)
loop
se *nodos* é vazio **retorna** falha
nodo <-- TIRAR-PRIMEIRO(*nodos*)
se TESTE-SUCESSO(*nodo*) tem sucesso
retorna *nodo*
novos-nodos <-- EXPANDIR(*nodo*)
nodos <-- ACRESCENTAR-NO-INICIO(*nodos*,*novos-nodos*)
fim

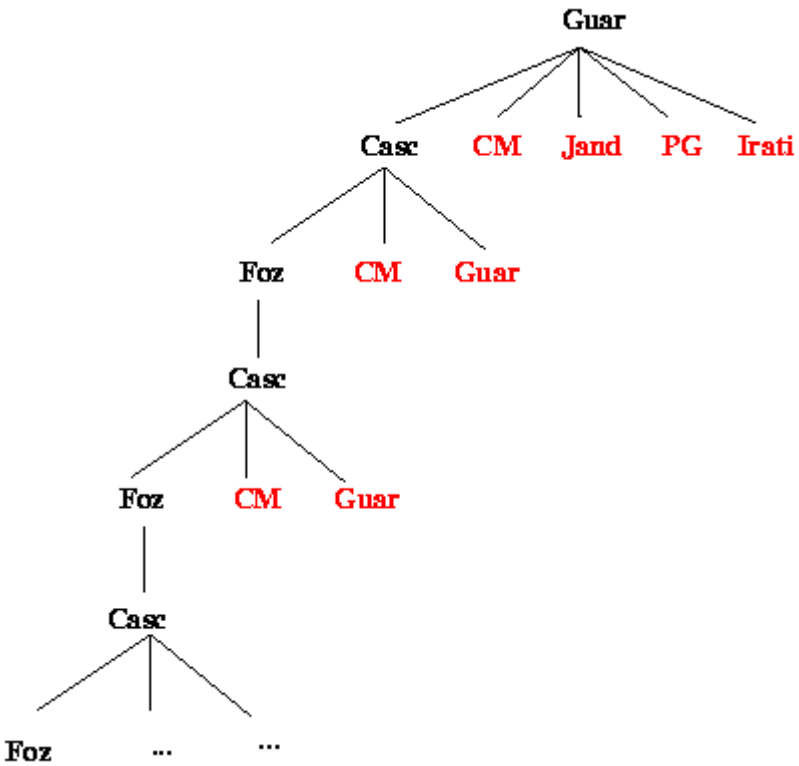
Vantagens :

Econômico em memória.
As vezes, pode achar a solução muito rapidamente (especialmente se tem muitas soluções).

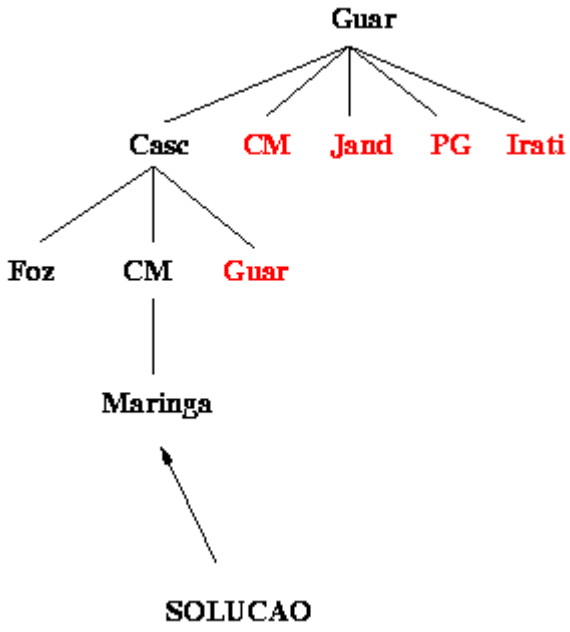
Desvantagens :

Não completo. A busca pode entrar em um loop infinito, especialmente se o domínio dos estados é infinito.
Não tem certeza que vai retornar uma solução ótima. A primeira solução não é necessariamente a melhor.

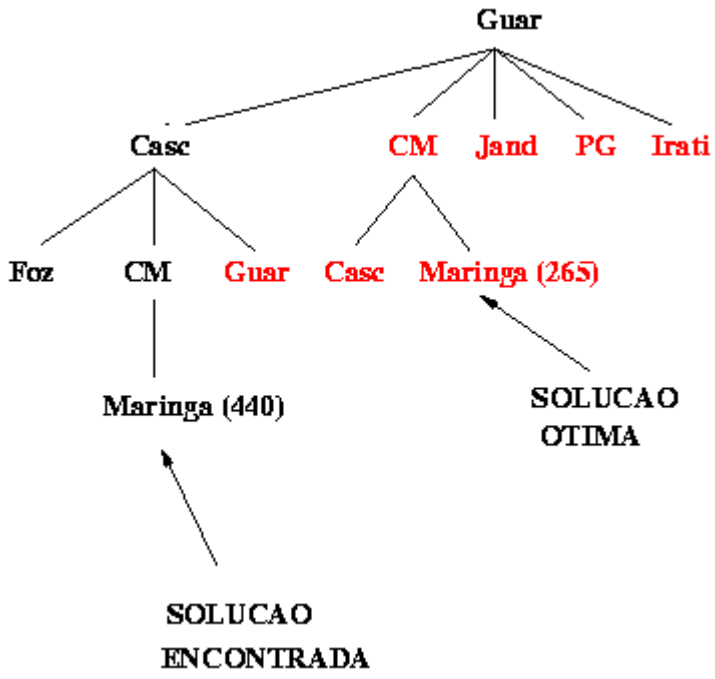
Supondo, por exemplo, estamos procurando um caminho entre Guarapuava e Maringá, a busca pode entrar em loop infinito se não evitamos passar duas vezes pela mesma cidade:



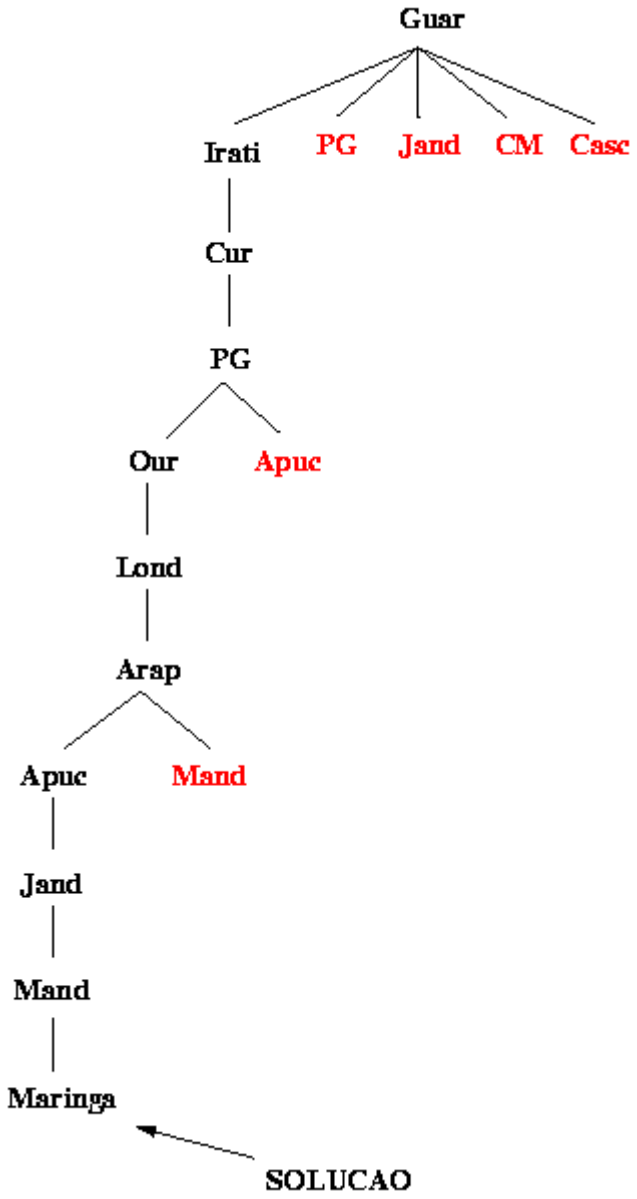
Se, ao contrário, evitamos retornar à mesma cidade, achamos rápidamente uma solução:



Mas essa solução não é ótima. A solução ótima está escondida nos nodos que não foram visitados:



É importante notar que o algoritmo de busca em profundidade, se ele é menos exigente em memória, não é melhor que a busca em largura no que se refere ao tempo de execução. Depois da expansão de um nodo, a ordem da aparição dos nodos filhos na pilha influencia muito a busca. Veja por exemplo como a ordem pode tornar muito ruim a busca do caminho entre Guarapuava e Maringá:



Variações sobre busca em profundidade

Até agora a busca em profundidade é a técnica mais vantajosa em termos de uso da memória. Infelizmente, ela apresenta a desvantagem de não necessáriamente retornar uma solução (ou mergulhar demais antes de voltar a un nível de profundidade mais baixa onde se encontra a solução). Existem duas técnicas baseada na busca em profundidade para contornar esse problema: a busca com profundidade limitada e a busca em profundidade iterativa.

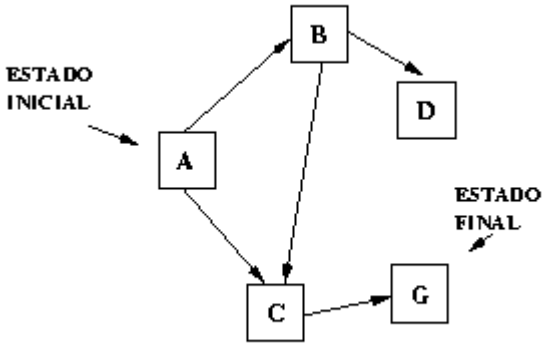
Busca com profundidade limitada:

Escolhe-se um valor limite de profundidade que a busca não pode ultrapassar. Isso vai dar certo somente se pudermos confiar que a solução encontra-se dentro desse limite. Se não escolhermos o bom valor de limite de profundidade, pode não retornar uma solução. Esse tipo de busca resolve o problema da incompletude mas ainda é possível que seja retornada uma solução não ótima.

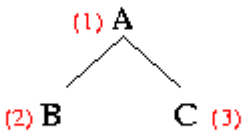
Busca em profundidade iterativa:

Compromisso entre a busca em largura e a busca em profundidade. Nesse caso, tentamos primeiro uma busca em profundidade com limite de profundidade 0. Se não encontramos uma solução, repetimos com limite de profundidade 1, 2, 3 e assim por diante até achar uma solução. Essa solução tem a vantagem de economia em memória, pois é a busca em profundidade que é utilizada, e também vai achar a solução ótima (se a funçao de custo é uniforme), pois não passamos a um nível superior de profundidade antes de ter esgotado o nível precedente. Esse algoritmo é adequado se o espaço de estados é grande e a profundidade da solução não é conhecida. A complexidade em tempo é igual à da busca em largura. A complexidade em memória é igual à da busca em profundidade.

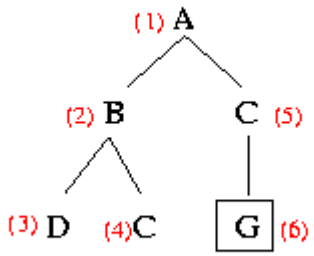
Para ilustrar a busca em profundidade iterativa, considere o seguinte espaço de estado (suponhamos que o custo de um estado para outro é 1):



É fácil ver que uma busca em profundidade pode retornar a solução não ótima ABCG. Com uma busca em profundidade iterativa, uma primeira busca será realizada até o nível 0. Nesse caso somente o estado A será visitado. Como isso não é o estado final, recomeçamos uma busca até o nível 1. Eis uma ilustração dessa busca (usamos números entre parênteses para indicar a ordem de visita dos estados):



De novo não encontramos uma solução. Uma nova busca é realizada, nessa vez até o nível 2, e uma solução (ótima) é retornada:



A busca em profundidade iterativa parece muito ineficiente pois o mesmo nodo pode ser expandido muitas vezes. Quando fazemos mais uma iteração para o nível *n+1*, todos os nodos que foram criados no nível *n* vão ser criados de novo. Mas surpreendentemente, o algoritmo não é muito mais ineficiente que a busca em largura. Por exemplo, com um fator de ramificação de 10, o número de nodos expandidos, em relação à busca em largura, é somente 11% maior.

Com a busca em profundidade iterativa, obtemos um comportamento idêntico à busca em largura no que concerne a completude e a otimalidade, mas com um uso de memória mais viável. Contudo, se a função de custo não é uniforme, ele tem também o mesmo defeito que a busca em largura: é possível que a resposta retornada não seja ótima. Uma solução a esse problema seria uma busca onde a cada iteração, ao invés de incrementar o limite de profundidade, incrementamos o limite de custo. Uma técnica semelhante será apresentada mais para frente.

Busca bidirecional

Duas buscas são realizadas em paralelo. Uma a partir do estado inicial e outra a partir do objetivo. Temos uma solução quando as duas se encontram. Para poder utilizar essa solução, temos que respeitar as seguintes exigências:

- Os operadores devem ser reversíveis. Não é sempre possível calcular o predecessor de um nodo. Por exemplo, se o objetivo é de comprar um máximo de itens com menos dinheiro possível, já é difícil representar o estado final, ainda mais os predecessores desse estado.
- Além de definir qual tipo de busca realizada nas duas direções de maneira a maximizar as chances delas se encontrarem, precisamos de um método para verificar o encontro.

Complexidade dos algoritmos básicos de busca

Eis uma tabela que resume a complexidade dos algorimtos básicos de busca:

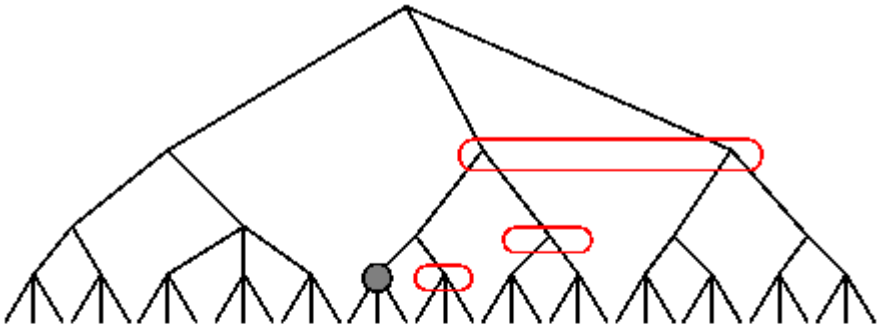
	Prof.	*Ampl.*	*Custo unif.*	*Prof. limit.*	*Prof. iterat.*
Tempo	b^m	b^d	b^d	b^l	b^d
Memória	bm	b^d	b^d	bl	bd
Sol. ótima	Não	Sim*	Sim**	Não	Sim*
Completude	Não	Sim	Sim	Sim, se l >= d	Sim

b = fator de ramificação
d = profundidade da solução
l = limite de profundidade especificado
m = profundidade máxima atingida na busca

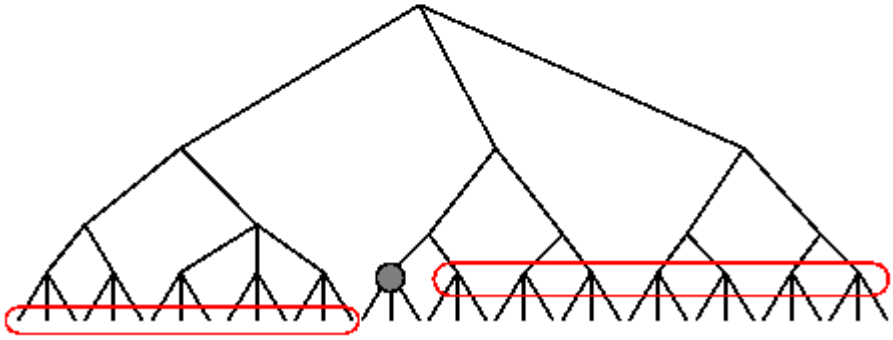
* Somente se o custo para de um estado ao próximo é sempre o mesmo (função de custo uniforme).

** Somente se o custo não diminua quando o caminho aumenta.

A seguinte figura mostra intuitivamente, para as buscas em profundidade e em largura, quais são os nodos esperando quando o nodo indicado por o ponto cinza está visitado:



Busca em profundidade

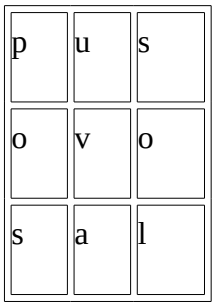


Busca em amplitude

Pode ver que o algoritmo de busca em largura tem muito mais nodos esperando, então é muito mais "guloso" em memória.

Exercícios

3.5 Considere o seguinte problema. Temos uma matriz de 3x3 e devemos colocar em cada posição uma letra do alfabeto de tal maneira que todas linhas e colunas formem uma palavra que existe em português. Eis um exemplo de solução para esse problema:

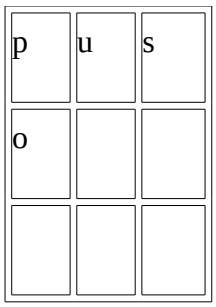


Suponha que uma posição sem letra é representada pela constante `vazio` e que existe uma base de fatos que representa as palavras do português:

```
palavra([a,l,o]).
palavra([p,u,s]).
...
palavra([s,a,l).
```

Considere agora as duas seguintes maneiras de representar um estado no espaço de busca:

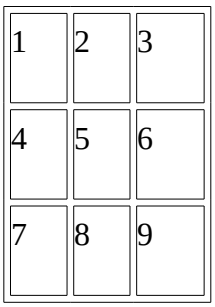
- *Primeira abordagem:* Representamos um estado utilizando uma matriz exatamente como na questão 4. Por exemplo, o estado ilustrado na figura 1a teria a representação ilustrada em 1b.
- *Segunda abordagem:* Utilizamos uma lista de 9 elementos, onde cada posição na lista corresponde à ordem ilustrada na figura 1c. O estado da figura 1a seria representado como ilustrado em 1d.



(a)

```
[[p,u,s],
 [o,vazio,vazio],
 [vazio,vazio,vazio]]
```

(b)



(c)

```
[p,u,s,o,vazio,vazio,vazio,vazio,vazio]
```

(d)

a) Sabendo que o problema será implementado como um sistema de produção (estado inicial, operadores para produzir os estados sucessores, teste de sucesso, função de custo), identifique as vantagens e desvantagens dessas duas representações.

b) Supondo que você tem a possibilidade de utilizar o algoritmo `breadth-first` ou o algoritmo `depth-first`, qual será o melhor, considerando o espaço de busca desse problema (justifique).

Busca heurística

Uma busca heurística é uma busca que utiliza uma função *h(n)* que, por cada nodo *n* do espaço de busca, dá uma avaliação do custo para atingir o estado final. A função *h(n)* é chamada **função heurística**.

Melhor escolha(best-first)

O algoritmo de busca pela melhor escolha é o seguinte:

nodos <-- CRIAR-FILA(*estado-inicial*)
loop
    se *nodos* é vazio **retorna** falha
    nodo <-- TIRAR-MELHOR-NODO(*nodos*)
    se TESTE-SUCESSO(*nodo*) tem sucesso
        retorna *nodo*
    novos-nodos <-- EXPANDIR(*nodo*)
    nodos <-- ACRESCENTAR(*nodos*,*novos-nodos*)
fim

A diferença entre ese algoritmo e o algoritmo geral é que ele usa uma função *f(n)*, que dá um valor a cada nodo *n* da lista dos nodos abertos (os nodos que estão esperando para ser expandidos). O nodo que tem o menor valor é escolhido para continuar a busca.

Vamos ver agora os algoritmos que resultam de diferentes definições da função *f(n)*.

Busca em largura com custo uniforme

Se utilizamos a função *f(n) = g(n)*, onde *g(n)* dá o custo de estado inicial até o nodo *n*, obtemos o algoritmo de busca em largura com custo uniforme. Já sabemos que esse algoritmo não garante uma solução ótima.

Busca gulosa

Se *f(n) = h(n)*, onde *h(n)* é uma estimação do custo do caminho mais curto do nodo n até o objetivo, o resultado é uma busca "gulosa" , porque sempre vai escolher o maior passo possível, sem se preocupar se no final vai se obter a melhor solução. Por exemplo, para achar o caminho de Guarapuava até Curitiba, a escolha inicial é entre as seguintes cidades: Cascavel, Campo Mourão, Jandaia do sul, Ponta Grossa e Irati. O algoritmo vai escolher Ponta Grossa, pois é o mais perto de Curitiba. Então, o algoritmo vai dar uma resposta não ótima, pois o caminho que passa por Irati é mais curto.

Algoritmo A*

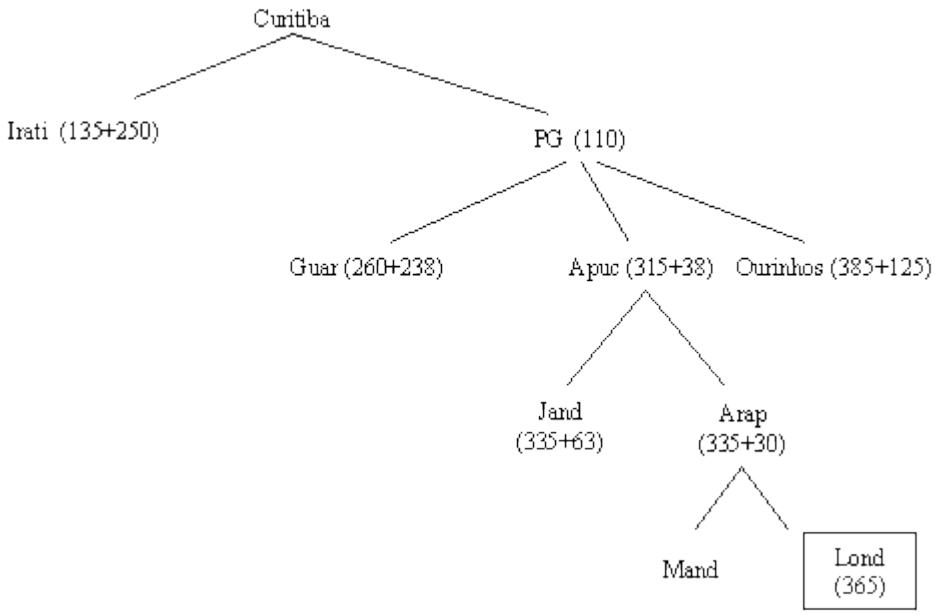
Uma maneira de melhorar os dois algoritmos precedentes é de combinar as duas funções: *f(n) = g(n) + h(n)*. Obtemos assim o algoritmo A. Essa função estima o custo total do estado inicial até o objetivo.

A obtenção da solução ótima depende da função heurística. Se *h(n)* não faz uma boa avaliação do custo até o objetivo, tem chance de "perder" a solução ótima.

Algoritmo A* e heurística admissível

Obtemos o algoritmo A* quando o algoritmo A usa uma heurística *h(n)* **admissível**. Uma heurística é admissível se para cada nodo, o valor retornado por esta heurística nunca ultrapassa o custo real do melhor caminho desse nodo até o objetivo. No exemplo do caminho de Curitiba até Londrina, o algoritmo acha a solução diretamente, se utilizarmos a seguinte heurística:

h(n) = distância em linha reta até a destinação



*Completude do algoritmo A**:

Como o algoritmo sempre expande o nodo que minimiza a função *f(n)* e que *f(n)* sempre retorna um valor que não ultrapassa o custo real para atingir o objetivo, necessariamente o algoritmo vai achar uma solução, se ela existe. Mas tem duas condições:

1. Não existe nodo com fator de ramificação infinito.
2. Não existe caminho com custo finito mas com número infinito de nodos.

*Prova da otimalidade do algoritmo A**:

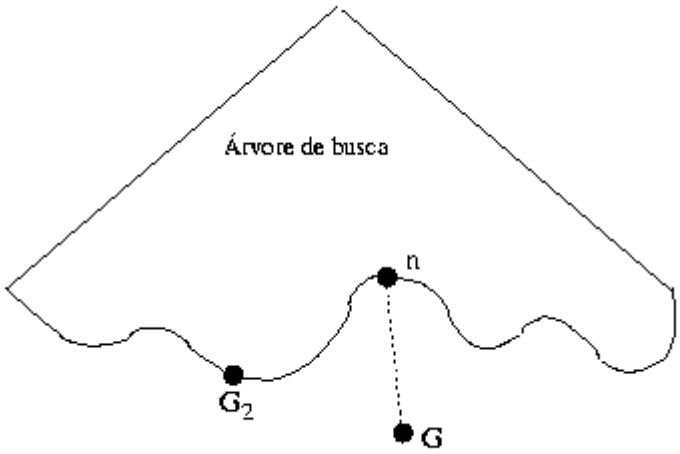
Seja *G* o estado final de uma solução ótima, *f** o custo dessa solução ótima, e G_2 o estado final de uma solução não ótima com custo $g(G_2) > f^*$ (veja ilustração abaixo).

Suponhamos que o algoritmo A* visita G_2 primeiro. Então existe, nas lista dos nodos abertos, um nodo *n* que faz parte do caminho até *G*. Como a função *h(n)* é admissível, temos $f^* >= f(n)$.

Visto que G_2 foi selecionado antes de *n*, temos também $f(n) >= f(G_2)$. Como $f^* >= f(n)$, podemos deduzir $f^* >= f(G_2)$.

G_2 sendo o estado final, $f(G_2) = g(G_2) + 0$.

Então, $f^* >= g(G_2)$, o que contradiz a hipótese.



A escolha da função heurística

Não é sempre fácil identificar uma boa função heurística para resolver um problema. Depende muito das características do problema.

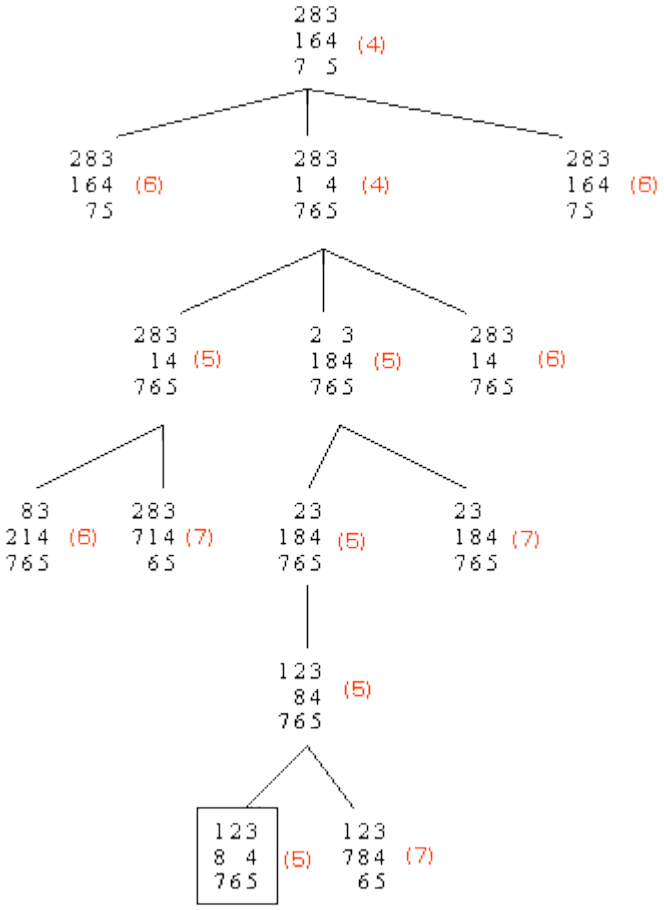
Exemplo do quebra-cabeça de 8:

$h_1(n)$ = número de quadrados em uma posição errada.

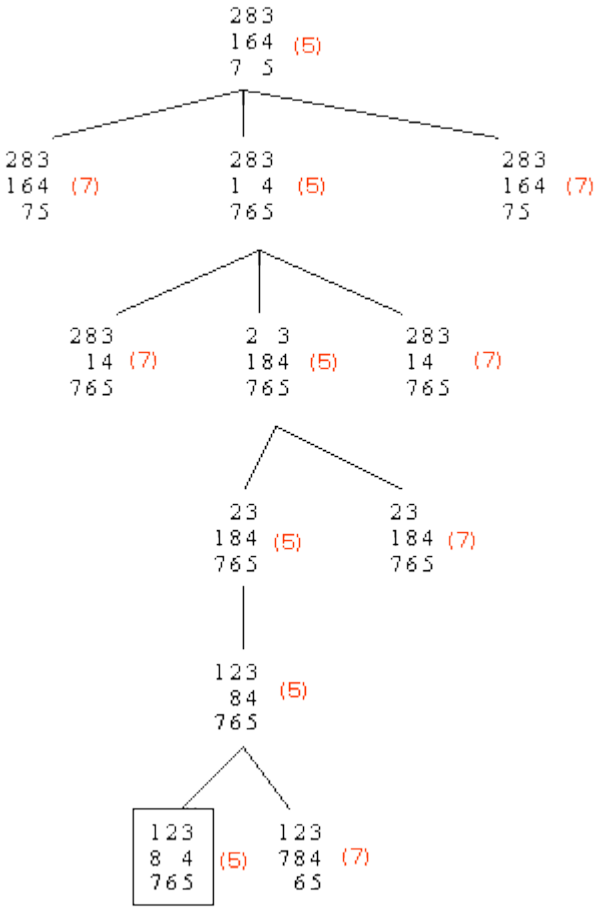
$h_2(n)$ = soma das distâncias que separam os quadrados das posições finais.

Com a função $h_2(n)$, o espaço de estados gerado é menor. Então, o algoritmo acha mais rapidamente a solução.

Espaço de estados gerado com $h_1(n)$ (para cada estado indicamos entre parênteses o valor da função heurística):



Espaço de estados gerado com $h_2(n)$:



Eliminação das redundâncias e monotonicidade:

No exemplo do quebra-cabeça de 8, não consideramos os nodos que aparecem por mais de uma vez. Por exemplo, a expansão do segundo nodo pode reproduzir o estado inicial. Não conservamos esse nodo porque, nesse caso, temos certeza que cada nova aparição de um estado não pode ter um valor *f(n)* menor. Mas não é sempre assim. Normalmente, quando un nodo aparece de novo, temos que compará-lo com o nodo já existente, e conservá-lo só se tiver um valor menor. No exemplo do quebra-cabeça de 8, a simplificação é possível porque a função heurística *h(n)* é **monotônica**.

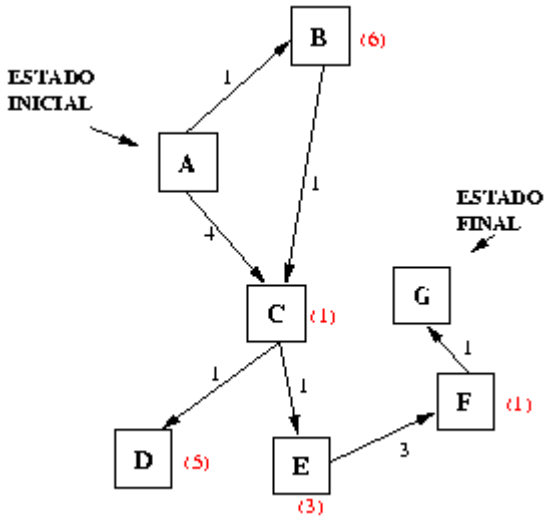
Uma heurística monotônica é uma função que respeita a seguinte condição:

Para todo estado n_i e n_j onde n_j é um descendente de n_i, temos

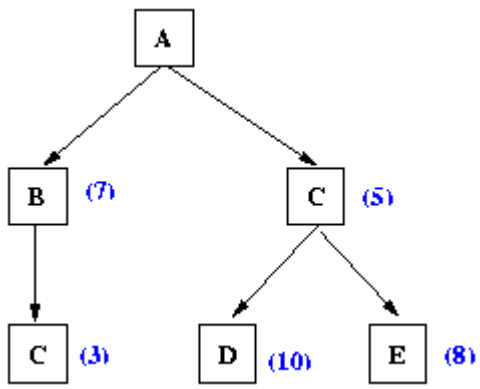
$h(n_i) - h(n_j) =<$ custo real de n_i até n_j.

Com uma heurística monotônica, f(n)=g(n)+h(n) tem a característica de ser uma função não decrescente. Uma consequência importante disso é que na primeira vêz que um nodo é visitado (cuidado: isso não inclui os nodos abertos) para ser expandido, *temos certeza que o caminho até esse nodo é ótimo* . Isso significa que cada vez que esse nodo será visitado novamente, não será necessário expandi-lo.

Aqui é um exemplo onde a função heurística não é monotônica (ao lado de cada estado, indicamos entre parênteses o valor retornado pela função heurística):



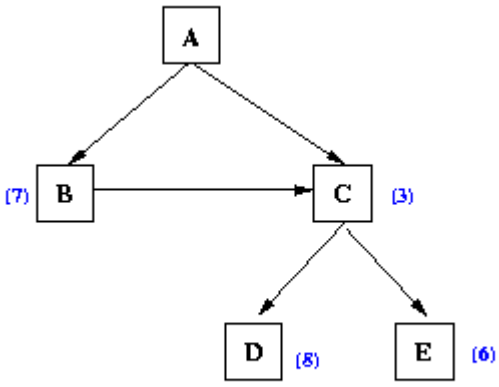
Nesse exemplo, o estado C vai aparecer uma segunda vez na busca, mas com valor menor, como mostra a seguinte figura (a cada nodo indicamos o valor retornado por f(n):



Nesse caso, podemos mostrar que a função h(n) não é monotônica: a diferença de custo entre B e C é 1, enquanto a diferença entre os valores h(n) é 5. A consequência disso é que o valor de f(n) diminuiu de B para C.

O uso de uma função monotônica pode fazer uma grande diferença no desempenho da busca. Considere o que acontece quando um nodo é visitado de novo. Se a função é monotônica, a única coisa que devemos fazer é verificar, na lista dos nodos fechados, se ele não existe. Se ele existe, podemos descartar essa nova ocorrência do nodo. Se a função não é monotônica, deveremos também comparar os valores dos dois nodos, caso ele já existe na lista dos fechados. Se a nova ocorrência do nodo tem um valor menor, deveremos aplicar um desses procedimentos:

- Eliminar da memória o antigo nodo e todos os seus descendentes.
- Ao invés de apontar no novo nodo, apontar no antigo, e atualizar o valor de todos os descendentes desse nodo. No exemplo acima, o resultado seria o seguinte:

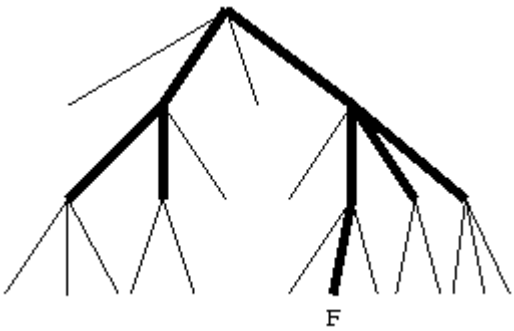


Essas duas abordagens têm um custo computacional não desprezível que é evitado com uma função monotônica.

Avaliação de uma função heurística:

Para avaliar uma função heurística, é bom entender bem o efeito dela na busca. Suponhamos que, no problema considerado, a expansão de um nodo resulta na criação de *b* novos nodos, em média (em outros termos, o fator de ramificação médio é *b*). A conseqüência da função heurística é que apenas alguns desses nodos vão ser visitados na busca da solução.

Para entender melhor, consideramos a seguinte ilustração de um espaço de busca, onde o estado final é representado por F, e onde as linhas grossas representam os caminhos até os nodos que foram visitados pelo algoritmo:



Com esse exemplo, podemos perceber que o fator de ramificação real é aproximativamente 2. Se considerarmos só os nodos visitados pelo algoritmo, o fator de ramificação seria um valor perto de 2. Esse último valor é o **fator de ramificação efetivo** e pode ser usado para avaliar uma função heurística.

Seja *N* o número de nodos criados para obter a solução e *d* a profundidade da solução. O fator de ramificação é o valor b* que respeita essa equação

$$N = 1 + b^* + (b^*)^2 + ... + (b^*)^d$$

Uma boa heurística tem um valor b* mais próximo possível de 1.

No caso do quebra-cabeça de 8, o fator de ramificação médio é 2,67. Então, temos $1 < b^* < 2,67$. Os resultados com as heurística $h_1(n)$ = *número de peças mal colocadas* e $h_2(n)$ = *soma das distâncias até posição final* são os seguintes:

d	$h_1(n)$	$h_2(n)$
2	1,79	1,79
10	1,38	1,22
20	1,47	1,27

Uma outra maneira de avaliar uma heurística é compará-la com uma outra heurística. Se $h_1(n)$ e $h_2(n)$ são duas heurísticas admissíveis, a melhor é a que sempre dá os valores maiores. Diz-se que a heurística que dá o maior valor é mais **informada**.

Prova: Seja $h_2(n)$ a função de maior valor, quer dizer, $h_2(n)$ domina $h_1(n)$, e f* o custo da solução ótima. O algoritmo A* vai visitar todos os nodos que tem valor f(n) =< f*. Toda vez que um nodo será expandido com $h_2(n)$, ele será também com $h_1(n)$ Pode existir um *n* por o qual o valor de $h_1(n)$ permitirá a expansão de um nodo e o valor de $h_2(n)$ faria a função f(n) ultrapassar o custo do caminho ótimo. Nesse caso, o nodo nunca seria expandido com a função $h_2(n)$. Então, menos nodos serão expandidos com a

função h_2

Como achar uma função heurística:

Não tem receita para identificar uma boa função heurística. Temos que analizar as características do problema e usar a nossa intuição para achar a boa heurística. Mas, podemos usar algumas dicas:

- Imaginar o mesmo problema com menos restrições
- Se não tiver jeito de escolher entre duas heurísticas h1(n) e h2(n), usar h(n)=max(h1(n),h2(n)), que também é admissível e domina as duas.
- Usar informações estatísticas.
- Identificar as características de um estado que são úteis para definir a heurística.

Notas adicionais:

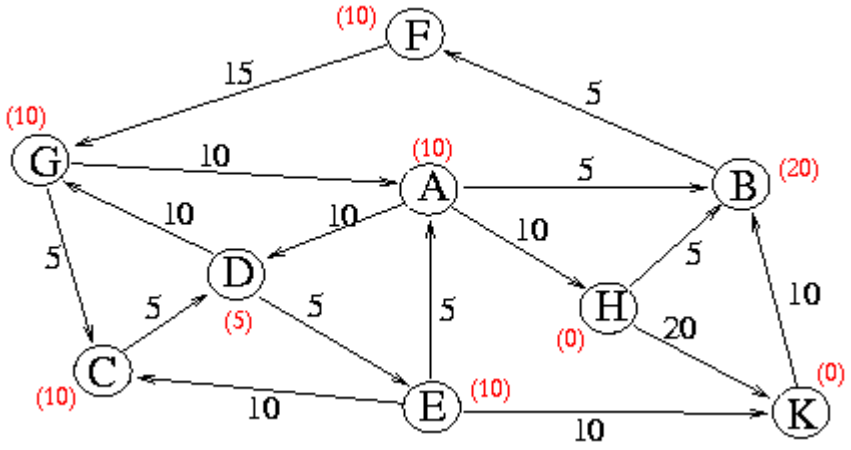
- O custo do cálculo da função heurística tem que ser bem menor que o custo para expandir um nodo. Senão, não vale a pena calcular o valor heurística, pois seria mais barato fazer a expansão. Então, temos que achar um equilíbrio entre os dois custos.
- Pode existir outras razões que a redundância para eliminar um nodo. Por exemplo, usando o algoritmo A*, podemos eliminar um nodo cujo custo ultrapassa o custo da solução ótima. É claro que isso funciona só se tem jeito de ter essa informação. Mas em alguns tipos de problema, isso é possível. Por exemplo, se quisermos achar o melhor jeito da fazer compras gastando menos dinheiro possível, podemos cortar um nodo que representa um estado onde não temos mais dinheiro.

- O custo de A* em memória e tempo é e exponencial, assim como a busca em largura, apesar de ser mais viável que essa última em muitos casos reais. Não existe algoritmo que expande menos nodos que A*. Mas mesmo assim, em problemas muito difíceis o algoritmo A* será ineficiente.

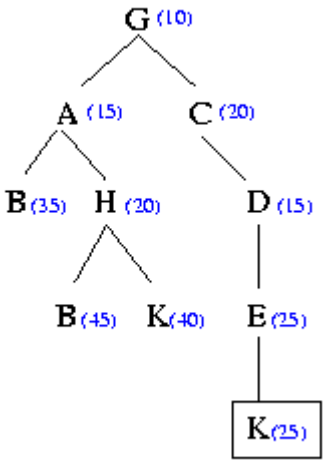
Algoritmo IDA* (A* iterativo)

O maior problema com A* é o uso da memória. Para contornar esse problema, podemos utilizar a mesma técnica que foi utilizada para a busca cega: uma versão iterativa do algoritmo. A idéia é o seguinte. Calculamos o valor f(n) do estado inicial. Como a solução ótima não pode ter um custo menor, realizaremos uma busca em profundidade na qual serão expandidos somente os nodos que não ultrapassam esse limite. Se, de todos os nodos visitados, nenhum é a solução, recomeçamos uma nova busca, aumentando o limite. Qual será o melhor valor para o novo limite? Considere os nodos que não foram expandidos. Um deles minimiza o valor f(n). Como a solução ótima não pode ter um valor menor, recomeçaremos com esse novo valor. Assim por diante até que o estado final seja visitado.

Para ilustrar o algoritmo, considere o seguinte espaço de estado (entre parênteses é indicado o valor h(n) para cada estado):

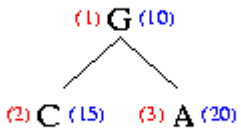


Suponhamos que G é o estado inicial e K o estado final. Eis a árvore de busca no momento que A* encontra a solução:

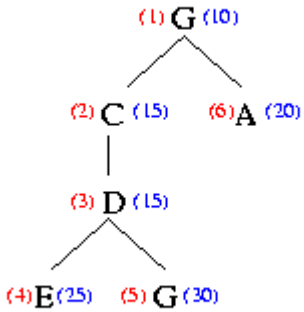


Nesse exemplo, aproveitamos da monotonicidade da função: os nodos A e D foram descartados na segunda ocorrência. Note que no final da busca, tem 10 nodos na memória: 6 nodos fechados (incluindo o estado final) e 4 nodos abertos. Vamos ver agora como o mesmo problemas será resolvido com o algoritmo IDA*.

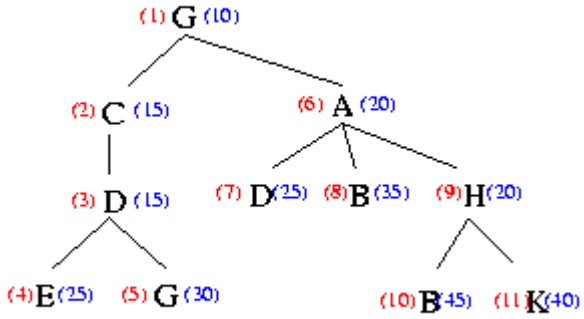
Como f(G) = 0 + 10 = 10, a primeira busca expande todos os nodos que tem um valor que não ultrapassa 10. Somente G será expandido (os números em azul na esquerda indicam a ordem de visita e os número em vermelho na direita indicam o valor f(n)):



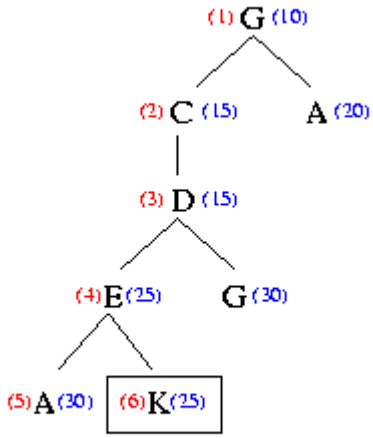
Nos nodos abertos, o com menor valor é o nodo C, com o valor 15. Então, uma nova busca em profundidade é realizada, como limite de 15:



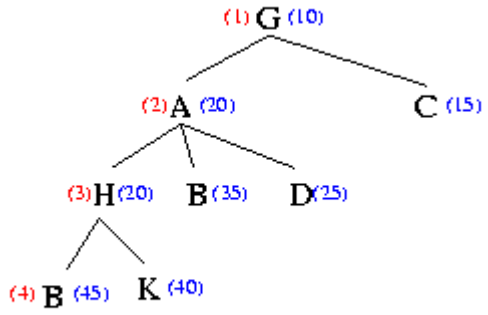
Agora recomeçamos uma busca com limite de 20:



Note que apareceu uma solução, mas ela não é retornada porque o seu valor ultrapassa o limite. Finalmente, recomençando com o menor valor dos nodos abertos, 25, obtemos a solução ótima:



Nesse exemplo, o maior espaço de memória usada foi na última iteração, que tem dois nodos esperando na pilha. O pior caso, nesse exemplo, seria uma busca que visite, nas duas últimas iterações, o nodo A antes do nodo C, e depois o nodo H antes dos nodos D e B. Nesse caso, teríamos quatro nodos esperando na pilha como ilustrado na seguinte figura:



Conncluíndo, o algortimo IDA* reduziu de 10 para 4 (no pior caso) o número de nodos ocupando o espaço de memória. É uma melhoria importante. Passamos de um uso exponencial de memória a um uso proporcional à profundidade máxima atingida.

Algoritmo RBFS (A* recursivo)

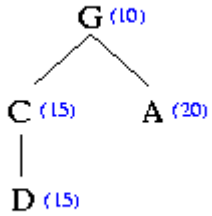
O algoritmo RBSF (Recursive Best-First Search) também apresenta um uso de memória semelhante ao da busca em profundidade. A idéia consiste em memorizar para cada nodo *n* o menor valor entre o seu valor *f(n)* e os valores de todos os seus descendentes. Para realizar essa atualização, aplicamos o seguinte algoritmo:

1. Seja *n* o último nodo expandido, e n_1, n_2, ..., n_i, os nodos resultando dessa expansão.
2. Se o menor dos valores $f(n_1)$, $f(n_2)$, ..., $f(n_i)$ é maior que *f(n)*, substituimos o valor *f(n)* por esse valor.
3. Repetimos esse processo, comparando os valores de *n* e seus irmãos com o valor do pai de *n*. Essa propagação pára quando encontramos um pai que não tem um valor inferior ao menor de seus filhos.

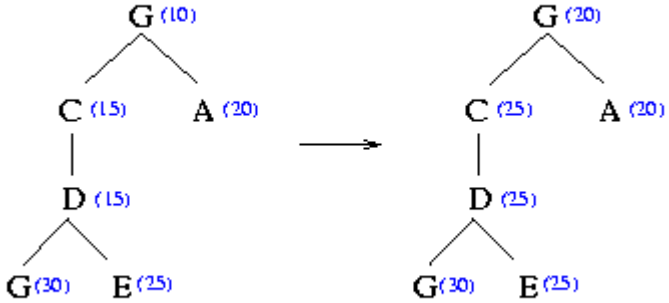
Depois dessa atualização, todo nodo temo um limite inferior do custo total. A busca tenta efetuar a expansão em profundidade. Quando existe um nodo que apresenta um valor mínimo e que não é no nível mais profundo da busca, apagamos todos os nodos que estão em um nível de profundidade maior que esse nodo mínimo, e recomeçamos a expansão a partir desse nodo.

Vamos ver agora a aplicação desse algoritmo ao nosso exemplo. O estado inicial tem o valor 10. Depois da expansão, aparecem o nodos A e C, com custo 20 e 15 respectivamente. Já sabemos que, qualquer seja o caminho da solução ótima, ele passa por A ou C. Então, o estimativo inicial, que era 10, é otimista demais, pois sabemos agora que não pode ser menor que 15, passando por C. Então, devemos diparar o mecanismo de atualização:

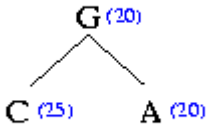
O próximo nodo expandido é o nodo C:



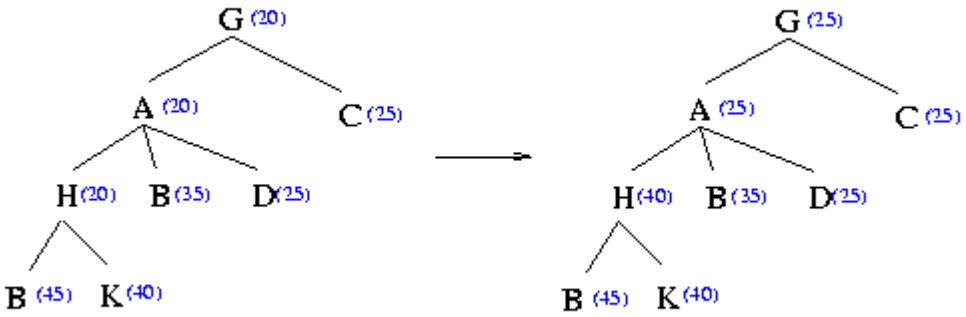
Nesse caso, o algoritmo de atualização não altera os valores, pois o novo nodo criado não tem um valor maior que seu pai. Como esse novo nodo tem o menor valor, ele é expandido, produzindo uma situação que deve ser atualizada:



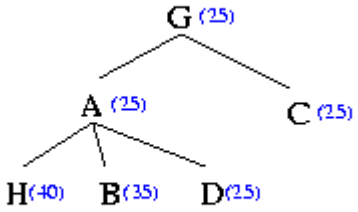
Depois da atualização, é o nodo A que tem o menor valor. Então, devemos considerar esse nodo e esquecer a busca abaixo do nodo C. Note que não perdemos tudo, pois memorizamos o custo da busca passando por C:



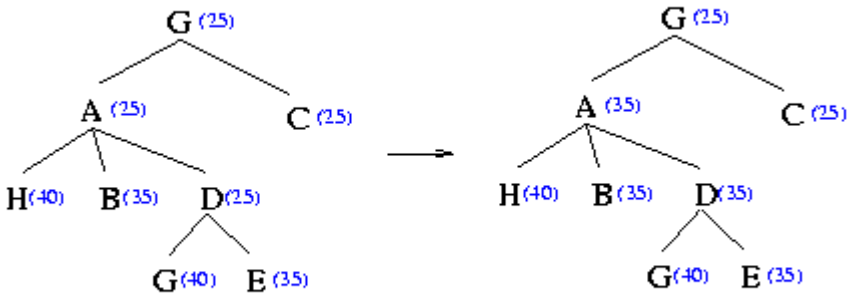
Recomeçando a expansão a partir de A chegamos, em duas etapas, a uma nova situação que exige atualização dos valores:



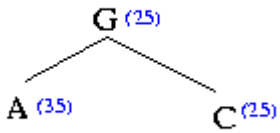
Agora, temos dois nodos candidatos que apresentam o menor valor (C e D), e nenhum deles se encontra no nível mais profundo. Qualquer seja o nodo escolhido, será preciso apagar alguns nodos. Suponhamos que o mais profundo é escolhido, isto é, o nodo D. O apagamento resulta na seguinte situação:



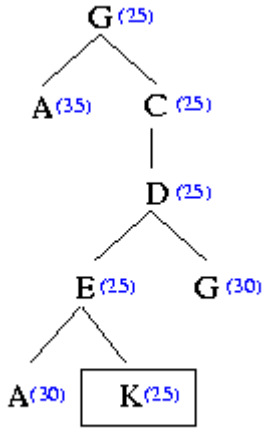
Expandindo o nodo D, obtemos a seguinte situação:



De novo, o nodo de menor valor não está no nível mais profundo. Nese caso, devemos apagar tudo até o nível do nodo C:



A partir desse nodo, chegaremos finalmente à solução ótima:



Subida de encosta (hill-climbing)

A idéia é a seguinte. Propor uma configuração e modificá-la até a obtenção de uma solução. É importante notar que não se tem certeza de achar uma solução ótima. É um método local, no sentido de que a cada momento o algoritmo considera somente os estados imediatamente acessíveis a partir do estado atual. É como se fosse realizada uma busca em profundidade, esquecendo todos os nodos que não foram escolhidos a cada nível da árvore.

Eis os algoritmo, supondo que temos uma função f(n) que queremos maximizar:

1. Identificar o estado atual com o estado inicial: n_{atual} = estado inicial. Em certas aplicações, o estado inicial pode ser gerado aleatoriamente.
2. Identificar todos os estados sucessores possíveis de n_{atual} e calcular, para cada estado sucessor *n*, o valor *f(n)*. Seja n_i o estado sucessor que tem o maior valor.
3. Se $f(n_i) < n_{atual}$, retornar n_{atual}. Isso significa que o algoritmo encontrou um estado que maximiza a função *f(n)*.
4. Senão, $n_{atual} = n_i$ e voltar à etapa 2.

É claro que, se o objetivo é de **minimizar** a função f(n), é o mesmo algoritmo, só que conservamos o estado que tem o menor valor e o algoritmo retorna o estado atual quando $f(n_i) > n_{atual}$ Exemplo: Palavras cruzadas

Suponhamos uma grade de *3x3* que devemos preencher com letras que formam palavras vertical e horizontalmente. Eis uma solução:

p	u	s
o	v	o
s	a	l

Para usar o hill-climbing com esse problema, poderiamos preencher cada posição com uma letra escolhida aleatoriamente para obter o estado final. Uma função de avaliação possível é o número de linhas ou colunas que formam uma palavra. Para passar de um estado a outro estado, uma letra é trocada por outra.

Problemas com o algoritmo hill-climbing:

- Ele pode se encontrar em um máximo local. Como ele sempre "sobe" e pára quando ele se encontra em um cume, se tiver outro cume mais alto no espaço de estados, ele não vai retornar essa solução. Pior ainda, se no cume que ele encontrou o estado não corresponde à solução esperada, ele não retorna uma solução.
- Planaltos. Isso acontece quando todos o estados sucessores tem mais ou menos o mesmo valor. Nesse caso, o algoritmo vai "andar" aleatoriamente nessa região. Esse problema vai acontecer no nosso exemplo das palavras cruzadas.

Soluções

- Gerar uma outra configuração aleatoriamente e recomeçar a partir desse novo estado.
- Têmpera simulada. A idéia é de permitir ao algoritmo escolher, de vez em quando, um estado que tem um valor menor. A probabilidade de fazer tal escolha diminui à medida que a busca progride.

Não é sempre fácil achar uma heurística para tornar eficiente o algoritmo de subida de encosta. Considere por exemplo uma situação onde o objetivo é de inverter os blocos que formam uma pilha:

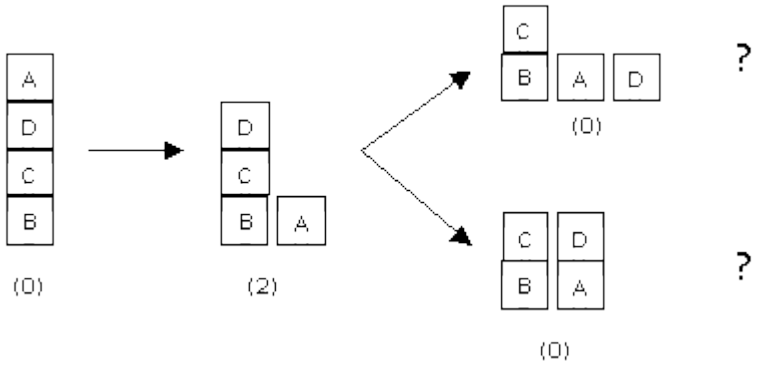
Exemplo do mundo de blocos:

Suponhamos que queremos obter uma configuração específica de blocos. Poderiamos usar a seguinte heurística:

+1 para cada cubo sobre o cubo certo

-1 para cada cubo não posicionado sobre o cubo certo

Suponhamos que o objetivo é de colocar o bloco D sobre C, C sobre B, B sobre A e A sobre a mesa. A seguinte figura ilustra que pode acontecer que a heurística, por retornar o mesmo valor, não ajuda para escolher o próximo estado. Nesse caso, devemos escolher aleatoriamente, o que pode nos afastar do bom caminho:

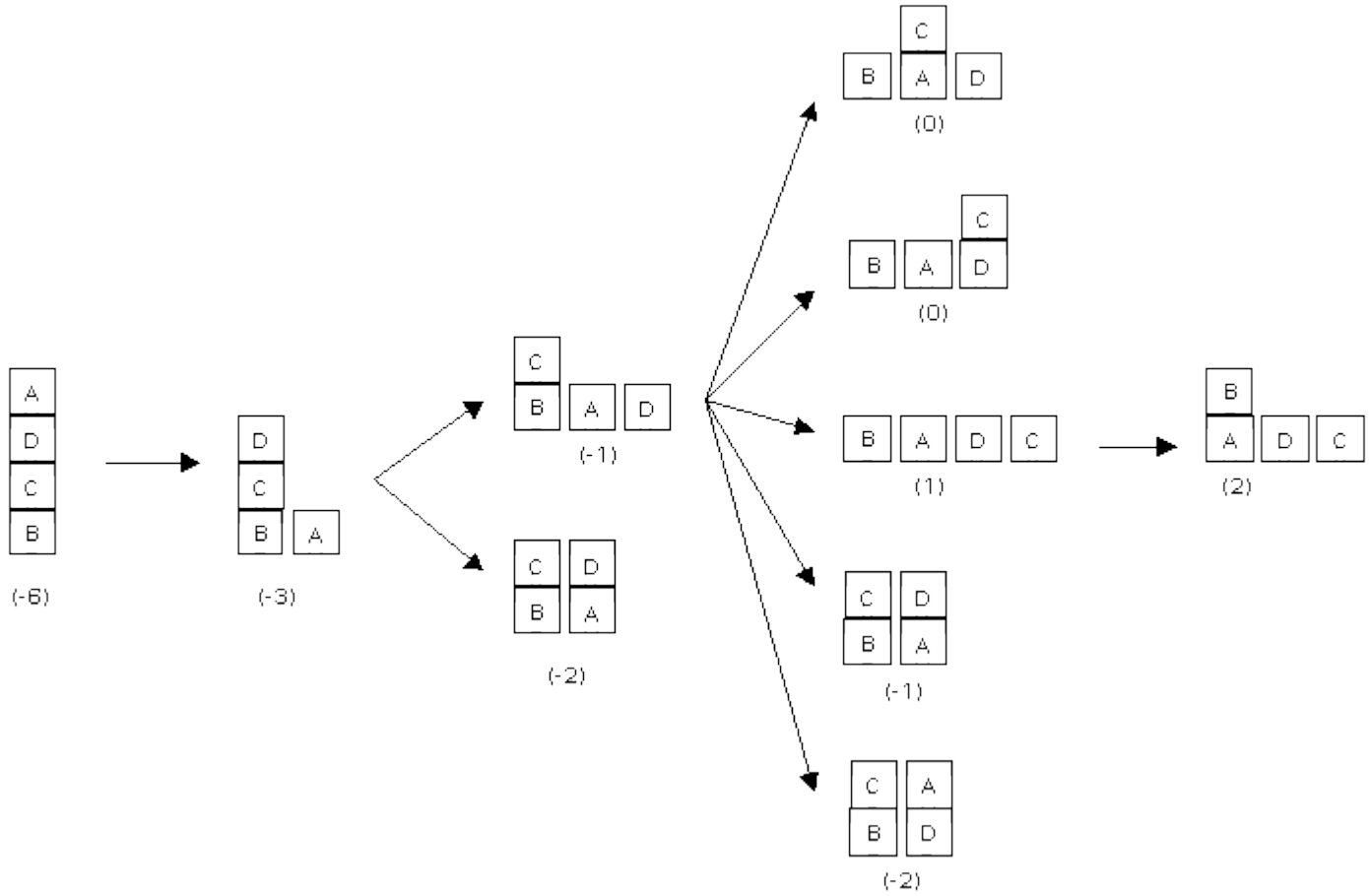


Com o mesmo problema, o algoritmo funcionará melhor com a seguinte eurística:

+n para cada cubo cuja estrutura de n cubo que o suporta é correta

-n para cada cubo cuja estrutura de n cubo que o suporta não é correta

Nesse caso, podemos verificar que o algoritmo vai chegar rapidamente ao estado final.



O método de subida de encosta é equivalente à busca em profundidade sem a memória das alternativas, ou a um gerar-e-testar inteligente. O algoritmo funciona bem quando as etapas do problema são recuperáveis, isto é, um problema no qual nunca chegamos em um estado de onde é impossível sair.

Satisfação de restrições:

Nessa seção é apresentada uma técnica de resolução onde o problema é representado por um conjunto de variáveis a serem instanciadas e um conjunto de restrições sobre essas variáveis. Nesse tipo de problema, a sequência de passos para atingir o estado final é na maioria dos casos irrelevante. O que queremos é identificar um valor para cada variável de tal maneira que todas as restrições sejam respeitadas.

Vamos nos limitarmos a um caso específico, onde as variáveis têm um domínio finito. Os domínios das variáveis não são limitados a conjuntos de valores. Podem ser de qualquer tipo. As restrições são grupadas em dois tipos: as equações e os procedimentos. Eis alguns exemplos do primeiro tipo: X < 4, Y = X+Z, X 0. O segundo tipo se refere às restrições que são definidas por um procedimento. Por exemplo, se X e Y representam pessoas, e queremos que elas sejam de sexos diferentes, poderiamos representar isso por uma restrição *sexo_dif(X,Y)* definida da seguinte maneira em Prolog:

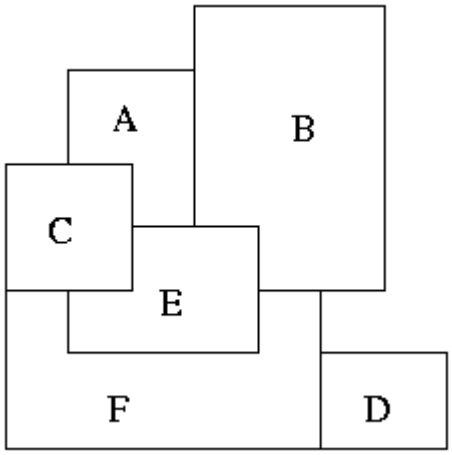
```
sexo_dif(X,Y):-
     mulher(X), homem(Y).
sexo_dif(X,Y):-
     mulher(Y), homem(X).
```

Uma maneira simples de resolver esse tipo de problema é utilizar a técnica Gerar e Testar:

1. Escolher um valor para cada variável do problema.
2. Verificar se as restrições são respeitadas.
3. Se elas são, terminar. Senão voltar à etapa 1.

É interessante aqui discutir como será escolhida cada nova proposta de solução. O método mais natural consiste em usar o retrocesso cronológico (backtrack). Sejam X_1, X_2, ..., X_n as *n* variáveis que compõem o problema. Primeiro, escolhemos um valor para cada variável nessa ordem. Se é preciso voltar à etapa 1, voltamos à última variável que aínda contém valores no seu domínio que não foram escolhidos. Seja X_i essa variável. Selecionamos o próximo valor do domínio que não foi selecionado até agora. Para todas as variáveis que vêm depois na sequência, recomeçamos a seleção a partir do início do domínio.

Para entender melhor, considere o exemplo ilustrado na seguinte figura, onde temos um mapa a ser colorido minimizando o número de cores utilizadas e de tal maneira que não tenha duas áreas contíguas coloridas com a mesma cor:



Esse problema pode ser representado pelo conjunto de variáveis {A, B, C, D, E, F} e as seguintes restrições:

A B
A C
A E
B E
B F
C E
C F
E F
D F

Supondo que toda variável tem por domínio o conjunto de cores {azul, vermelho, roxo} e que as instanciações são feitas na sequência A, B, C, D, E e F, a primeira resposta retornada será a seguinte:

	A	B	C	D	E	F
1	azul	azul	azul	azul	azul	azul

Como essa proposta é rejeitada pelas restrições, voltamos à última variável que tem mais valores a propor, que é a variável F. Obtemos assim uma nova proposta onde F tem o valor *vermelho*, que é também rejeitada. Mesma coisa com o último valor possível para F:

	A	B	C	D	E	F
2	azul	azul	azul	azul	azul	vermelho
3	azul	azul	azul	azul	azul	roxo

No próximo backtrack, voltaremos à variável E, pois todas as possibilidades foram esgotadas com a variável F. O próximo valor selecionada para E é a cor *vermelho*. Agora voltamos a selecionar um valor para F, recomeçando no início do domínio. Obteremos assim mais três possibilidades:

	A	B	C	D	E	F
4	azul	azul	azul	azul	vermelho	azul
5	azul	azul	azul	azul	vermelho	vermelho
6	azul	azul	azul	azul	vermelho	roxo

Continuando, obteremos uma solução depois de 124 tentativas:

	A	B	C	D	E	F
7	azul	azul	azul	azul	roxo	azul
8	azul	azul	azul	azul	roxo	vermelho
9	azul	azul	azul	azul	roxo	roxo
10	azul	azul	azul	vermelho	azul	azul
...	...	...	...	...	...	...
123	azul	vermelho	vermelho	vermelho	vermelho	roxo
124	azul	vermelho	vermelho	vermelho	roxo	azul

Nesse exemplo, a busca não atingiu uma largura dramática, mas em casos reais a situação rapidamente fica inviável. O principal problema aqui é que verificamos as restrições somente quando todas as variáveis são instanciadas. Mas na maioria dos casos, podemos detectar conflitos antes mesmo de terminar as instanciações. Na linha 1, por exemplo, não é preciso continuar depois de ter atribuído a cor *azul* a A e B, pois já temos uma restrição violada. Já poderíamos retroceder. Eis as etapas até a solução, usando essa técnica (note que nessa lista, cada etapa corresponde à instanciação de uma variável):

	A	B	C	D	E	F	
1	azul						
2	azul	azul	**Backtrack**				
3	azul	vermelho					
4	azul	vermelho	azul	**Backtrack**			
5	azul	vermelho	vermelho				
6	azul	vermelho	vermelho	azul			
7	azul	vermelho	vermelho	azul	azul	**Backtrack**	
8	azul	vermelho	vermelho	azul	vermelho	**Backtrack**	
9	azul	vermelho	vermelho	azul	roxo		
10	azul	vermelho	vermelho	azul	roxo	azul	**Backtrack**

11	azul	vermelho	vermelho	azul	roxo	vermelho	**Backtrack**
12	azul	vermelho	vermelho	azul	roxo	roxo	**Backtrack**
13	azul	vermelho	vermelho	vermelho			
14	azul	vermelho	vermelho	vermelho	azul	**Backtrack**	
15	azul	vermelho	vermelho	vermelho	vermelho	**Backtrack**	
16	azul	vermelho	vermelho	vermelho	roxo		
17	azul	vermelho	vermelho	vermelho	roxo	azul	**Solução**

A primeira tentativa é bloqueada antes de instanciar a variável C porque já nesse momento a restrição A B está violada. Então, a busca volta para B, lhe atribuindo um novo valor, *vermelho*. Agora é possível instanciar C, mas não poderemos ir além, porque o valor atribuído a C viola a restrição A C. Continuando assim, encontramos uma solução que necessitou a geração de somente quatro instanciações completas, e nove pedidos de backtrack. É um progresso impressionante, comparada à abordagem anterior.

É importante notar aqui que o custo para obter uma instanciação completa é maior, pois devemos testar as restrições a cada instanciação. Esse custo é compensado pelo fato de obter uma busca muito mais limitada. Mas mesmo assim, é possível reduzir o custo. A cada instanciação, não é preciso testar TODAS as restrições. É suficiente testar unicamente as que contêm a variável instanciada. Portanto, se tiver um armazenamento dar restrições indexado pelas variáveis, será possível testar somente as restrições necessárias a cada instanciação.

Uma outra maneira de melhorar ainda mais a busca é usar o **forward-checking**. A idéia é de olhar, considerando os valores já atribuídos e as restrições, se é possível reduzir o domínio das outras variáveis não instanciadas. Por exemplo, assim que a cor azul for escolhida para A, podemos excluir essa cor dos conjuntos de B, C e E. Além de limitar as possibilidades na busca, essa técnica tem o efeito desejável de disparar mais rapidamente o backtrack. Diminuindo assim os domínios das variáveis não instanciadas enquanto avançamos na busca, pode acontecer que uma dela se encontre com um domínio vazio. Assim poderemos efetuar um backtrack antes mesmo de terminar as instanciações. Eis as etapas na resolução de nosso problema (para cada variável não instanciada indicamos o conjunto de valores possíveis):

	A	B	C	D	E	F	
0	{a, v,r}	{a, v,r}	{a, v,r}	{a, v,r}	{a, v,r}	{a, v,r}	
1	a	{v, r}	{v, r}	{a, v,r}	{v, r}	{a, v,r}	
2	a	v	{v, r}	{a, v,r}	{r}	{a, r}	
3	a	v	v	{a, v,r}	{r}	{a, r}	
4	a	v	v	a	{r}	{r}	
5	a	v	v	a	r	{}	**Backtrack**
6	a	v	v	v	{r}	{a, r}	
7	a	v	v	v	r	{a, r}	
8	a	v	v	v	r	a	

Nessa vez, a resposta veio mais rápida ainda. Mas isso também teve um custo. Não somente verificamos as restrições a cada instanciação de variável mas também devemos fazer um cálculo para atualizar o domínios das variáveis não instanciadas. Esse custo poder ser não desprezível. Em vários casos, o forward-checking fica vantajoso apesar do custo.

No nosso exemplo, há um aspecto que não foi considerado e que tem uma importância considerável na obtenção da soluçao: a ordem de instanciação das variáveis. Considere por exemplo a ordem D, A, B, C, E, F. Nesse caso, usando o forward-checking, a busca fica mais demorada:

	D	A	B	C	E	F	
0	{a, v,r }	{a, v,r }	{a, v,r }	{a, v,r }	{a, v,r }	{a, v,r }	
1	a	{a, v,r }	{a, v,r }	{a, v,r }	{a, v,r }	{v, r}	
2	a	a	{v, r}	{v, r}	{v, r}	{v, r}	
3	a	a	v	{v, r}	{r}	{r}	
4	a	a	v	v	{r}	{r}	
5	a	a	v	v	r	{}	**Backtrack**
6	a	a	v	r	{}	{}	**Backtrack**
7	a	a	r	{v, r}	{v }	{v }	
8	a	a	r	v	{}	{}	**Backtrack**
9	a	a	r	r	{v }	{v }	
10	a	a	r	r	v	{}	**Backtrack**
11	a	v	{a, r}	{a, r}	{a, r}	{v, r}	
12	a	v	a	{a, r}	{r}	{v, r}	
13	a	v	a	a	{r}	{v, r}	
14	a	v	a	a	r	{v }	

15	a	v	a	a	r	v	**Solução**

A busca é quase o dobro com essa ordem. A questão agora é a seguinte: existe uma maneira de determinar a ordem de instanciação das variáveis para otimizar a busca? A resposta é sim. Eis as heurísticas para escolha da ordem de instanciação:

- Escolher a variável mais restringida (a que contém o menor domínio).
- Escolher a variável implicada em mais restrições.

Se tentamos resolver nosso problema com essas heurísticas, obtemos imediatamente a solução (note que normalmente, em problemas reais, não é assim, ainda tem que buscar um pouco!):

Variável mais restringida

	A	B	C	D	E	F
0	{a, v,r}	{a, v,r}	{a, v,r}	{a, v,r}	{a, v,r}	{a, v,r}
1	a	{v, r}	{v, r}	{a, v,r}	{v, r}	{a, v,r}
2	a	v	{v, r}	{a, v,r}	{r}	{a, r}
3	a	v	{v}	{a, v,r}	r	{a}
4	a	v	v	{a, v,r}	r	{a}
5	a	v	v	{v, r}	r	a
6	a	v	v	v	r	a	**Solução**

Variável envolvida em maior número de restrições

	A	B	C	D	E	F
0	{a, v,r}	{a, v,r}	{a, v,r}	{a, v,r}	{a, v,r}	{a, v,r}

1	{v, r}	{v, r}	{v, r}	{a, v,r}	a	{v, r}	
2	{v, r}	{r}	{r}	{a, r}	a	v	
3	v	{r}	{r}	{a, r}	a	v	
4	v	r	{r}	{a, r}	a	v	
5	v	r	r	{a, r}	a	v	
6	v	r	r	a	a	v	**Solução**

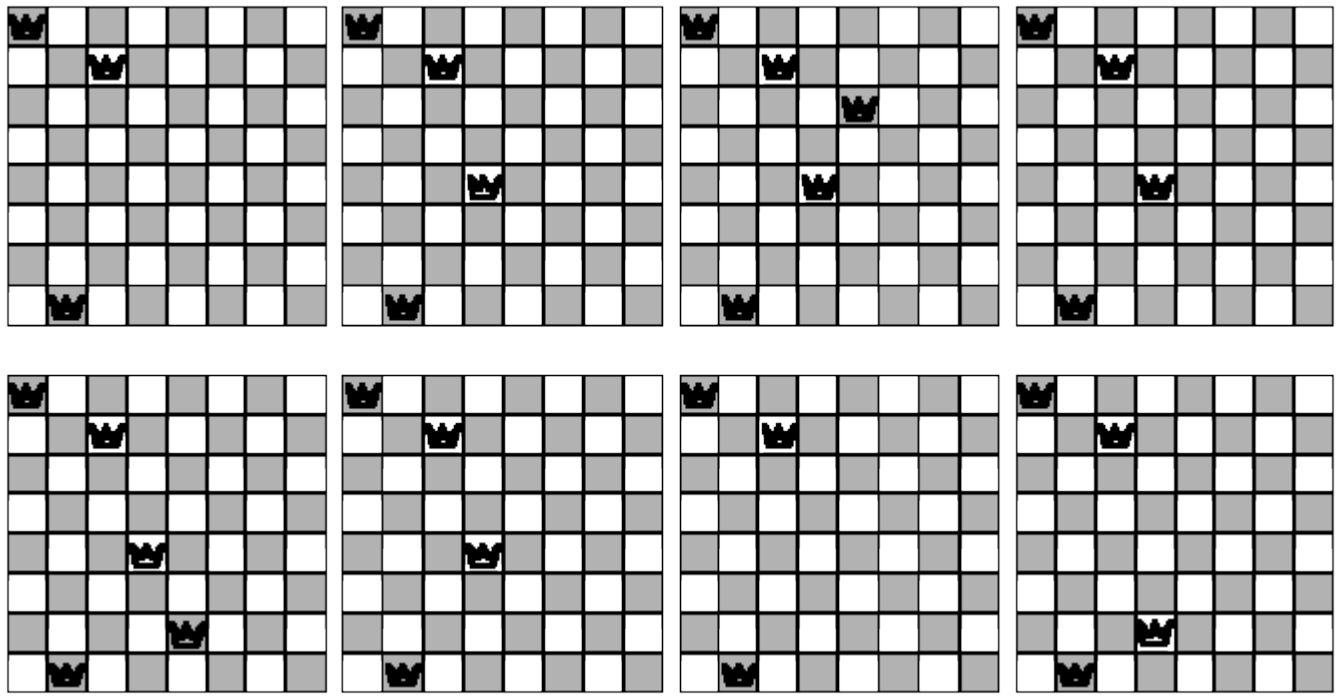
Também a escolha do valor para instanciar a variável pode influenciar muito a busca da solução. Uma heurística é escolher a menos restritiva, isto é, o valor que minimiza a redução dos domínios das outras variáveis após aplicação do forward-checking.

As heurísticas apresentadas acima para resolução de problemas de satisfação de restrições são gerais. En um problema específico, podem existir outras heurísticas para determinar a ordem de instanciação que dependem das características do problema.

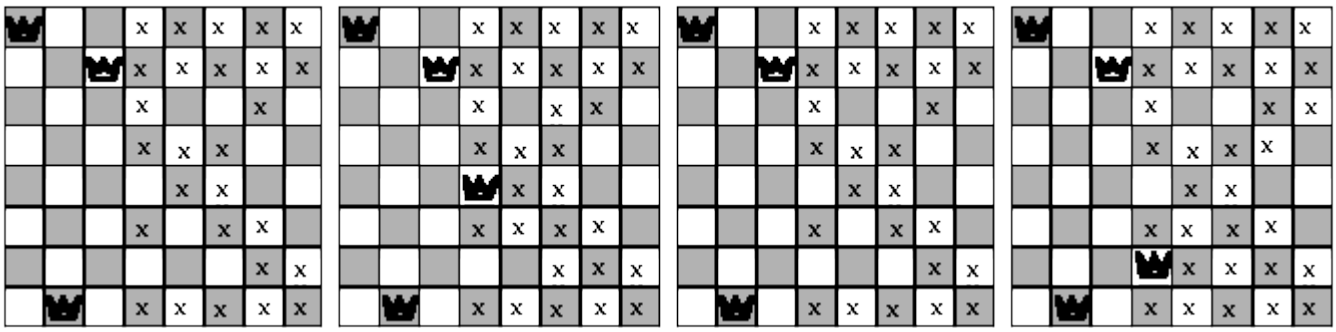
Outro exemplo: problema das 8 rainhas.

O problema das oito rainhas consiste em colocar oito rainhas sobre um tabuleiro de xadrez de tal maneira que nenhuma rainha seja ameaçada por outra. Para passar para o seguinte estado, a idéia é a seguinte: escolhemos uma das rainhas ameaçadas e nós a deslocamos na mesma coluna, no lugar que está ameaçado por o menor número de rainhas.

Sem forward-checking:



Com forward-checking:



Com o forward-checking, assim que colocamos a quarta rainha, podemos ver que será impossível colocar uma rainha na sexta coluna. Portante, é inútil tentar colocar rainha nos dois lugares disponíveis na quinta coluna e o algoritmo já sabe que ele tem que efetuar um backtrack.

Minimização de conflitos

Essa técnica de resolução é uma combinação do Hill-Climbing com satisfação de restrições. Do Hill-Climbing ela usa a idéia de escolher o melhor estado sucessor. Das técnicas de resolução por satisfação de restrições ela usa o princípio de representar o problema por um conjunto de variáveis e restrições sobre essas variáveis. Um estado é representado por uma instanciação completa das variáveis. Como no Hill-Climbing, se o estado não é o estado final (nesse caso, se existem restrições violadas), identificamos o melhor estado sucessor. O que muda em relação ao Hill-Climbing original é a maneira de identificar o estado sucessor.

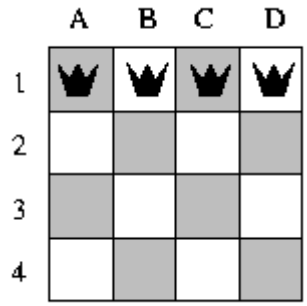
Primeiro, identificamos as variáveis envolvidas em conflitos, isto é, variáveis cujo valor viola uma restrição. Para cada uma delas, consideramos os outros valores que pode receber. E para cada valor possível calculamos o número de conflitos que ele causaria. Escolhemos a variável e o valor que causam o menor número de conflitos. Um exemplo ajudará a esclarecer a técnica.

Exemplo: Problema das oito rainhas:

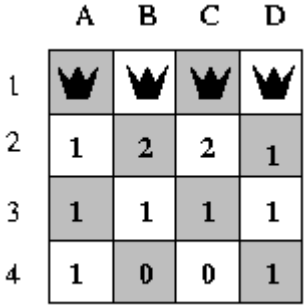
Para simplificar, utilizaremos o problema simplicado de quatro rainhas a colocar em um tabuleiro de 4x4. Cada coluna seria representada por uma variável. Então teríamos quatro variáveis A, B, C e D. O domínio de cada uma é um valor de 1 a 4 para representar a linha onde se encontra a rainha. As restrições seriam as seguintes:

(1) A B
(2) A C
(3) A D
(4) B C
(5) B D
(6) C D
(7) A B + 1
(8) A B - 1
(9) A C + 2
(10) A C - 2
(11) A D + 3
(12) A D - 3
(13) B C + 1
(14) B C - 1
(15) B D + 2
(16) B D - 2
(17) C D + 1
(18) C D - 1

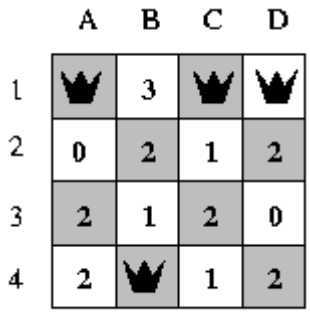
Suponha o seguinte estado inicial, onde A = B = C = D = 1:



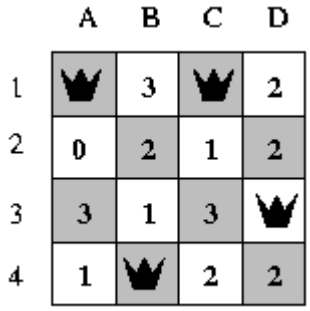
Veja o número de conflitos para cada posição possível:



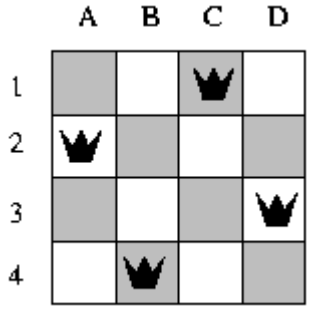
Se colocarmos a rainha B na segunda linha, ela estaria envolvida em dois conflitos (as restrições (8) e (13) acima seriam violadas). Se colocarmos uma das rainhas B ou C na última linha, nenhuma restrição será violada. Portanto é uma dessas possibilidades que será escolhida. Supondo que a rainha B é deslocada para a posição 4, obtemos a seguinte situação:



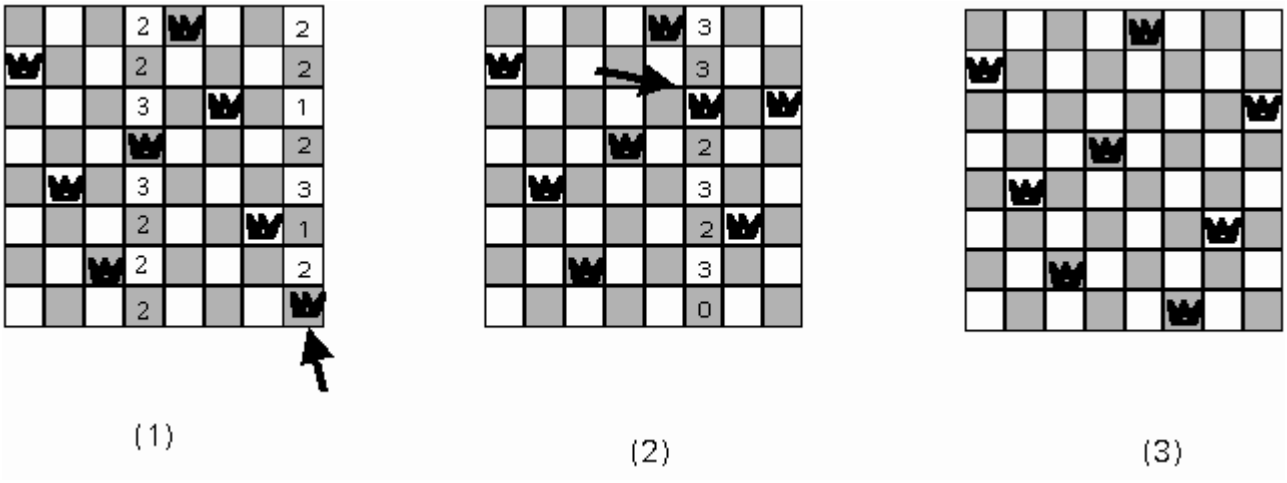
De novo temos duas possibilidades. Deslocando a rainha D na terceira linha, obtemos a seguinte configuração:



Finalmente, colocando a rainha A na segunda coluna, obtemos uma solução:



Eis um exemplo do problema original de oito rainhas onde a solução é encontrada em duas etapas :



Note que em (1) tem duas rainhas ameaçadas. Então, tentamos deslocar uma delas. Escolhemos a última, porque na sua coluna tem duas posições que minimizam a ameaça. Deslocando a rainha na terceira casa de sua coluna, obtemos uma situação onde sobra somente uma rainha ameaçada que podemos deslocar em uma casa segura, resolvendo assim o problema. Se tivéssemos escolhido a outra casa com valor 1, demoraria mais para encontrar a solução: dez etapas a mais, que ainda é rasoável.

Limites das técnicas de busca

As técnicas de busca apresentadas aqui sofrem todas de alguns limites que podem torná-las inviáveis:

- *Ambiente dinâmico*: se o ambiente no qual se encontra o agente é dinâmico, pode acontecer que a situação mudou enquanto ele estava construindo o seu plano. Nesse caso, pode acontecer que o plano não valha mais. Um ambiente pode se tornar dinâmico pela presência de efeitos externos (imagine por exemplo um agente confrontado a um ambiente sensível às condições meterológicas) ou outros agentes (futebol de robô por exemplo).
- *Imprecisão:* A representação que o agente se faz sobre o mundo pode ser imprecisa. Supondo um agente que depende de uma câmera pada se fazer uma representação do mundo, pode acontecer que ele tenha informações parciais. Por exemplo, ele pode ter dificuldade para deduzir o que é escondido por obstáculos na cena que ele está analisando. Se ele tem que se deslocar, a presença de um objeto não previsto pode atrapalhar a execução do plano. Outra fonte de imprecisão são os efeitos das ações. Se por exemplo o agente deve inserir um objeto em um buraco, usando um braço articulado, vai ser difícil ativar a mecânica do braço de maneira a respeitar exatamente os cálculos que foram feitos para determinar o plano.
- *Limite de tempo e de memória* Os recursos de memória são limitados. Também em certas situações, como por exemplo, sistemas de emergência ou o futebol de robô, o agente tem um tempo limitado para efetuar a busca.

Para superar esses limites, existem soluções. Eis alguamas:

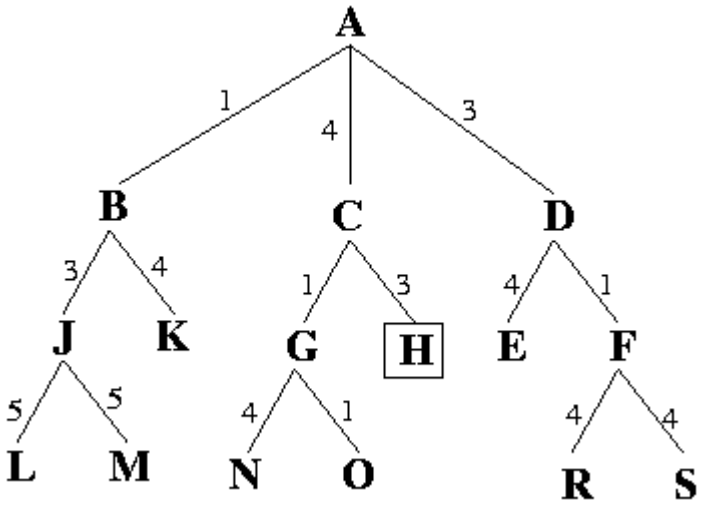
- *Planejamento condicional:* O agente produz um plano onde as açoes escolhidas dependem de condições. No momento da execução, o agente consulta o ambiente para determinar quais condições são respeitadas, a fim de escolher a ação.
- *Monitoramento:* Enquanto executando o plano, o agente verifica se o ambiente mudou de maneira a impossibilitar o plano. Se for o caso, ele recomeça a busca. Uma implementação simples dessa técnica é um agente que executa somente a primeira ação do seu plano. Depois ele analisa o

mundo e se a situaçao mudou, ele recalcula o seu estado inicial (o estado onde ele se encontra agora, depois de ter executado o primeiro passo) e recomeça o processo de busca. Assim por diante até chegar ao estado final.

- *Busca aproximativa:* Se não dá para usar um algoritmo, como A*, que retorna uma solução ótima, podemos usar uma outra técnica mais rápida, mas que pode retornar uma solução não ótima. Isso pode ser feito usando uma heurística não adimissível, ou terminando a busca antes de ter identificado o estado final. Eis exemplos de busca aproximativas:
    - *Busca em ilhas:* Nessa técnica, identificamos alguns estados $n_1 ... n_k$ que devem fazer parte do caminho. Supondo n_0 e n_f os estados iniciais e finais, realizamos uma busca para determinar um plano para passar do estado n_0 até o estado n_1. Depois recomeçamos para identificar um plano de n_1 até n_2, e assim por diante até n_f. O plano final é a concatenação desses sub-planos. O soma dos tempos para efetuar a busca em cada um desses sub-problemas deveria ser menor que o tempo de busca para passar de n_0 a n_f. Como exemplo de aplicação, imagine alguém que está na zona sul de uma cidade e quer se deslocar para a um lugar na zona norte. Ao invés de identificar um plano só, podemos buscar um caminho da origem para um lugar do centro da cidade, e depois do centro da cidade para o destino.
    - *Busca hierárquica:* Nesse caso, existem macro-operadores que usamos em uma primeira busca para identificar as "grande etapas" do plano. Disso resulta um plano não muito detalhado. Recomeçamos o processos para detalhar cada uma das etapas desse plano de alto nível. Isse até achar um plano onde cada etapa é uma ação que pode ser executada. Suponhamos por exemplo que o agente deve empurrar um móvel de um lugar para outro lugar. Ele pode fazer uma busca para identificar um plano de alto nível: (1) Ir até o móvel, (2) Se posicionar para empurrar, (3) empurrar até o destino. Depois, cada uma dessas três etapas é refinada.
    - *Busca com horizonte limitado:* O agente tem um limite de profundidade estabelecido. Se o limite for um limite de tempo, podemos transformar isso em limite de profundidade, estimando a profundidade que a busca não deve ultrapassar para respeitar o limite de tempo. Nesse caso, usamos uma função de avaliação *f(n)* para cada estado *n*. Realizamos uma busca em profundidade, que memoriza o nodo que minimiza o valor *f(n)*. Quando a busca efetua um backtrack, ela expandirá somente os nodos que não têm um valor inferior ao valor mínimo memorizado. No final, a busca retorna o plano até o nodo que tem o menor valor *f(n)*.

Exercícios

3.6 Eis um espaço de estados, com indicação do custo para passar de um nodo ao seguinte. Queremos achar o caminho ótimo até o estado representado pelo nodo H. Suponhamos que h(n) retorna 2 para todo nodo, com a exceção do nodo H, que retornará 0.



Seja os três seguintes algoritmos de busca:

- Busca em profundidade
- Busca em largura
- Busca "melhor escolha", usando o algoritmo A.

a) Para cada um desses algoritmos, indique em que ordem são visitados os nodos até encontrar a solução.

b) Para cada um desses algoritmos, indique quais são os nodos que estão esperando quando ele visita o nodo H pela primeira vez. Dê a lista do nodos na ordem que eles seriam visitados se a busca continuar.

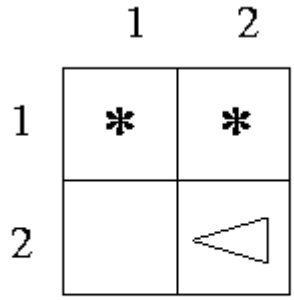
c) A função heurística h(n) é admissível? (Justifique)

d) A função heurística h(n) é monotônica? (Justifique)

3.7 Imagine um robô aspirador que se encontra em uma sala quadrada dividida em quatro posições. A todo momento o robô ocupa um desses quadrados e tem uma das seguintes orientações: norte, sul, leste, oeste. Em certos quadrados, pode ter sujeira. Existem três ações que o robô pode realizar, todas com um custo de 10:

- Avançar para a posição que está na sua frente. No estado resultante, o robô se encontra na posição que está na sua frente e aspira a sujeira, se tiver sujeira nessa nova posição.
- Efetuar uma rotação de 90 graus para a direita.
- Efetuar uma rotação de 90 graus para a esquerda.

O estado final é atingido quando não existe nenhuma posição com sujeira. Suponhamos um estado inicial com sujeira nas posições (1,1) e (2,1), e o robô na posição (2,2) orientado na direção oeste:



ESTADO INICIAL

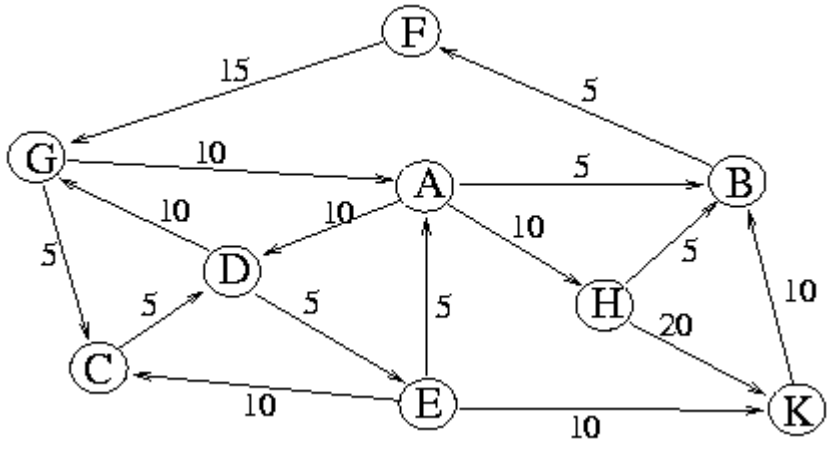
a) Seja a seguinte heurística admissível: $h_1(n) = 10 \times N_s$, onde N_s é o número de quadrados que contêm sujeira. Mostre como está a árvore de busca no momento em que o algoritmo A* encontra a solução e enumere os nodos na ordem em que eles foram visitados. Para cada nodo, indique também qual é o seu valor f(n).

b) Mostre que $h_1(n)$ é admissível.

c) Proponha uma heurística $h_2(n)$, admissível, que domina $h_1(n)$. Mostre como está a árvore de busca no momento em que o algoritmo A* encontra a solução e enumere os nodos na ordem em que eles foram visitados. Para cada nodo, indique também qual é o seu valor f(n).

d) Mostre, comparando as duas árvores de busca obtidas em a) e b), que $h_2(n)$ é melhor que $h_1(n)$.

3.8 Suponha um problema que pode ser representado como um espaço de 9 estados (designados pelas letras A até K). Considere o seguinte grafo, que representa os sucessores possíveis para cada estado:



Os nodos representam os estados e as arestas o custo para passar de um estado a outro estado. A direção da flecha indica o estado resultante. Por exemplo, é possível atingir o estado A a partir do estado G com um custo de 10.

Suponha agora que G é o estado inicial e K o estado final. Mostre a a seqüência dos nodos visitados (ordem de visita) e os nodos que estão esperando para ser visitados no momento em que se encontra uma solução, em cada uma das seguintes buscas:

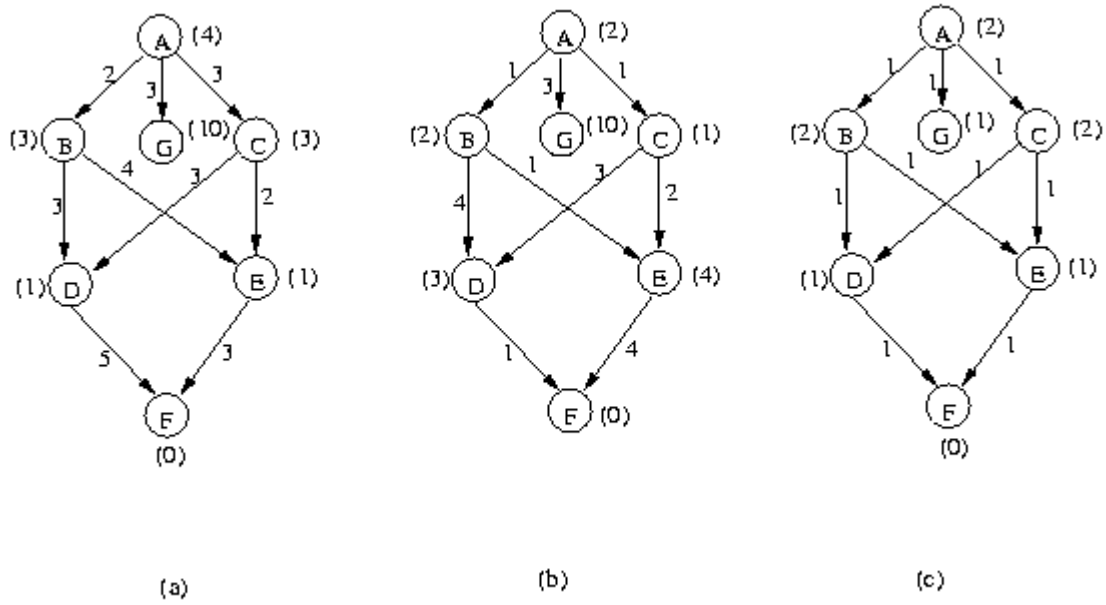
a) Busca em largura.

b) Busca em profundidade.

c) Busca com o algoritmo A*, utilizando a seguinte função heurística: h(A) = h(C) = h(E) = h(F) = h(G) = 10, h(B) = 20, h(D) = 5 e h(H) = h(K) = 0. Para cada nodo da árvore, indique o seu valor f(n).

d) Dessas três buscas, qual é a melhor para esse problema específico? Justifique.

3.9 Suponha um problema que pode ser representado como um espaço de 7 estados, designados pelas letras A até G, onde A é o estado inicial e F o estado final. Considere o seguintes espaços de estados:



(a) (b) (c)

Os nodos representam os estados e as arestas o custo para passar de um estado a outro estado. A cada estado está indicado entre parênteses uma estimativa do custo deste estado até o estado final.

Suponha que você possa escolher entre os seguintes algoritmos de busca: busca em profundidade, busca em largura, A* e Hill-Climbing. Para cada um dos três espaços de estados, indique qual algoritmo será o melhor, considerando que queremos uma solução ótima o mais rápido possível. Justifique as suas respostas e mostre qual será a ordem dos nodos visitados com o método escolhido.

3.10 Eu quero pagar a minha passagem de ônibus, que custa 90 centavos. Para pagá-la, eu quero utilizar ao menos 5 moedas. O cobrador quer que eu lhe dê uma moeda de 25 centavos ou duas de 10 centavos. Represente isso como um problema de satisfação de restrições e mostre como as heurísticas de forward-checking, variável mais restrita e/ou variável mais restringente agilizam a resolução. Para resolver o problema, fixe o número de moedas de 1, 5, 10, 25 e 50 centavos que eu tenho.

3.11 Resolva o problema de coloração acima usando a técnica de minimização de conflitos. Comece com os seguintes valores iniciais: A = D = azul, B = E = vermelho e C = F = roxo.

3.12 Considere o seguinte quebra-cabeça constituído de 3 peças brancas de um lado, 3 peças pretas do outro lado, e uma posição vazia:

O objetivo é inverter a ordem das peças, isto é, ter todas as peças brancas do lado esquerda e as pretas à direita. E isso ao menor custo possível. Para obter esse estado existem três movimentos possíveis:

1. Deslizar uma peça no lugar vazio se ele é contíguo a essa peça (custo = 1).
2. Uma peça pode pular encima de outra peça para alcançar o lugar vazio (custo = 1).
3. Uma peça pode pular encima de duas outras peças para alcançar o lugar vazio (custo = 2).

Proponha uma heurística admissível para resolver esse problema com A*.
