MÉTODOS DE BUSCA INFORMADA

As estratégias de busca não informadas encontram soluções para problemas a partir de geração sistemática de novos estados, e testando-os quanto ao objetivo.

Infelizmente, essas estratégias são ineficientes para a maioria dos casos. Já a busca informada encontra a solução, utilizando conhecimento específico do problema.

Inicialmente veremos versões informadas para alguns algoritmos vistos anteriormente, seguidos de outros métodos: estratégias de busca informada (heurísticas).

4.1 ESTRATÉGIAS DE BUSCA INFORMADA:

Ao usar um método de busca geral, o único lugar onde o conhecimento pode ser aplicado é na *Queuing Function* (QUEUING_FN), que determina o próximo nó a ser expandido. O conhecimento que faz essa determinação é fornecido pela função de avaliação *f(n)* (*evaluation function*) que retorna o desejo de expandir um determinado nodo, ou seja, o nodo com a menor avaliação é expandido primeiro, porque *f(n)* mede a distância do estado *n* para o objetivo.

Quando os nós são ordenados, o de melhor avaliação é expandido primeiro, e a estratégia resultante é chamada de busca "o melhor primeiro" (*bert-first seach*).

```
function BEST_FIRST_SEARCH (problem, EVAL_FN) returns a solution sequence
inputs problem, a problem
       EVAL_FN, an evaluation function

       QUEUEING_FN ← a function that orders nodes by EVAL_FN
return GENERAL_SEARCH (problem, QUEUEING_FN)
```

O nome *"best-first search"* é respeitável mas incorreto ou impreciso. Se expandirmos o melhor primeiro, esse deveria nos levar ao objetivo e na verdade escolhemos o nodo que PARECE ser o melhor de acordo com a função de avaliação.

Com o objetivo de encontrar a solução de menor custo, esses algoritmos (*best-first search*) usam alguma medida de estimativa de custo da solução e tenta minimizá-lo. Vimos

estratégias que consideram o custo do caminho, no entanto essa medida não direciona a busca para o objetivo. Para poder focar na busca, a medida deve incorporar alguma estimativa de custo do caminho de um estado até um estado próximo ao final.

Existem várias funções de avaliação que podem ser utilizadas, como por exemplo uma função heurística denotada por *h(n)*, onde:

h(n) = custo estimado de menor caminho de um nodo *n* para o objetivo.

Posteriormente veremos mais detalhes, mas agora é importante ressaltar que função heurística é específica por problema e se *n* é o objetivo então *h(n)* = 0.

Abordagens:

- Gulosa: tenta expandir o nó mais perto do objetivo;
- A*: tenta expandir o nó com um custo melhor (menor).

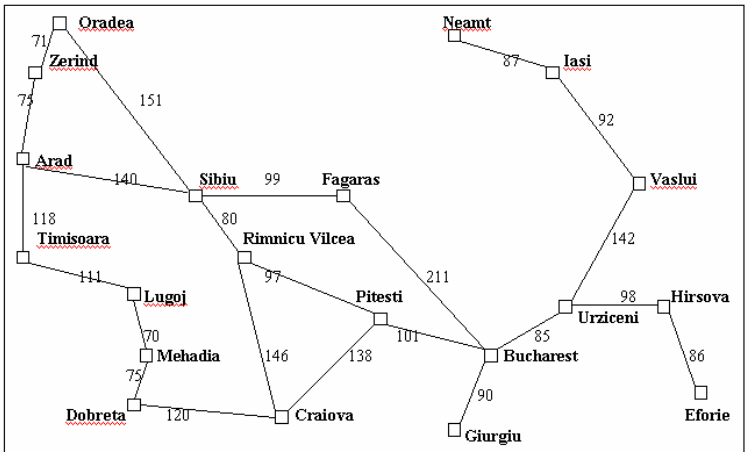
4.2 BUSCA GULOSA: MINIMIZAR O CUSTO ESTIMADO PARA ALCANÇAR O OBJETIVO

É considerada uma das estratégias mais simples de busca *"best-first"*. Sempre expande primeiro o nodo que for julgado mais próximo da meta.

Para a maioria dos problemas, o custo de atingir uma meta pode ser estimado mas não calculado exatamente. A função que calcula uma estimativa de custo é chamada de função heurística *h(n),* e estima o custo do caminho de um estado *n* até o objetivo.

A *"best-first search"* que usa uma função heurística *h* para determinar o próximo nó a ser expandido é chamada de busca gulosa (*greedy search*).

Para entender melhor é preciso considerar um problema, pois a função heurística é especifica por problema.



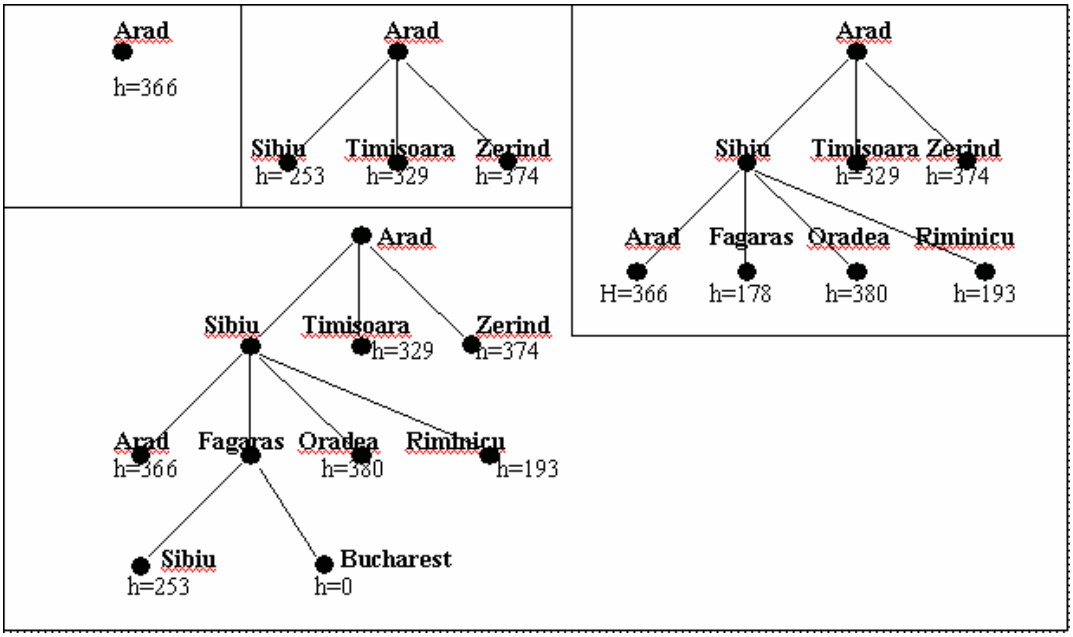
DISTÂNCIA EM LINHA RETA

Arad	366	Fagaras	178	Mehadia	241	Sibiu	253
Bucharest	0	Giurgiu	77	Neamt	234	Timisoara	329
Cariova	160	Hirsova	151	Oradea	380	Urziceni	80
Dobreta	242	Iasi	226	Pitesti	98	Vaslui	199
Eforie	161	Lugoj	244	Rimnicu vilcea	193	Zerind	374

Uma boa função heurística para o problema de encontrar uma rota é considerar a distância em linha reta:

$h_{DLR}(n)$ = distância em linha reta entre n e o objetivo

Note que para calcularmos os valores de h_{DLR} é preciso saber as coordenadas das cidades no mapa. Além disso, h_{DLR} é aplicável porque uma via de A a B normalmente tende a manter a mesma direção.



Vale ressaltar que para o exemplo acima, o caminho Arad, Sibiu, Fagaras e Bucharest é o menor porque não expandiu nenhum nó que não estava no caminho da solução. No entanto, o melhor caminho seria por "Rimnicu Vilcea"(32 km mais perto). Isso por que a busca Gulosa não

analisa a longo prazo, e sim tende a achar uma solução rápida, mas nem sempre ótima. Assim, é necessária uma análise mais cuidadosa das opções a longo prazo e não uma escolha imediata.

Além disso, a busca Gulosa é passível de início errado: de Iasi para Fagaras, se não tomarmos cuidado para detectar estados repetidos, a solução nunca será encontrada (Iasi ↔ Neamt). Neste caso, a solução seria via Vaslui que não é o nó escolhido pela heurística ***(Retrocesso ou Backtraking)***

Propriedades da busca Gulosa:

Não é completa: pode gerar busca infinita;

Não é ótima: nem sempre encontra a melhor solução

Tempo = espaço = $O(b^m)$ com m = profundidade máxima do espaço de busca (pior caso)

4.3 BUSCA A*: MINIMIZAR O CUSTO TOTAL DO CAMINHO

Busca Gulosa: minimiza a estimativa do custo e por meio disso minimiza o custo de busca, mas infelizmente pode não ser nem ótimo nem completo.

Busca custo uniforme: minimiza o custo do caminho *g(n)*, é ótimo e completo, mas pode ser muito ineficiente.

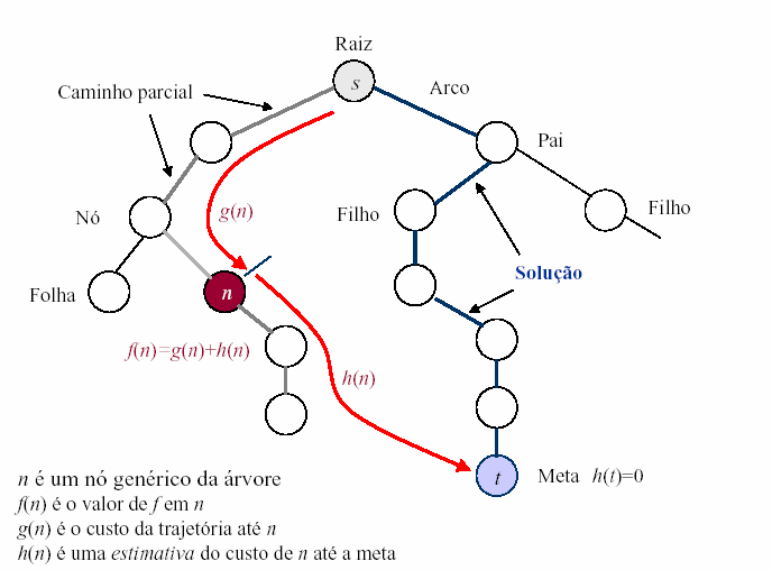
⇒ combinar as duas estratégias *h(n)* e *g(n): f(n) = h(n) + g(n),* onde

h(n) é a estimativa de custo de *n* até o objetivo;

g(n) é o custo acumulado do estado inicial até *n*;

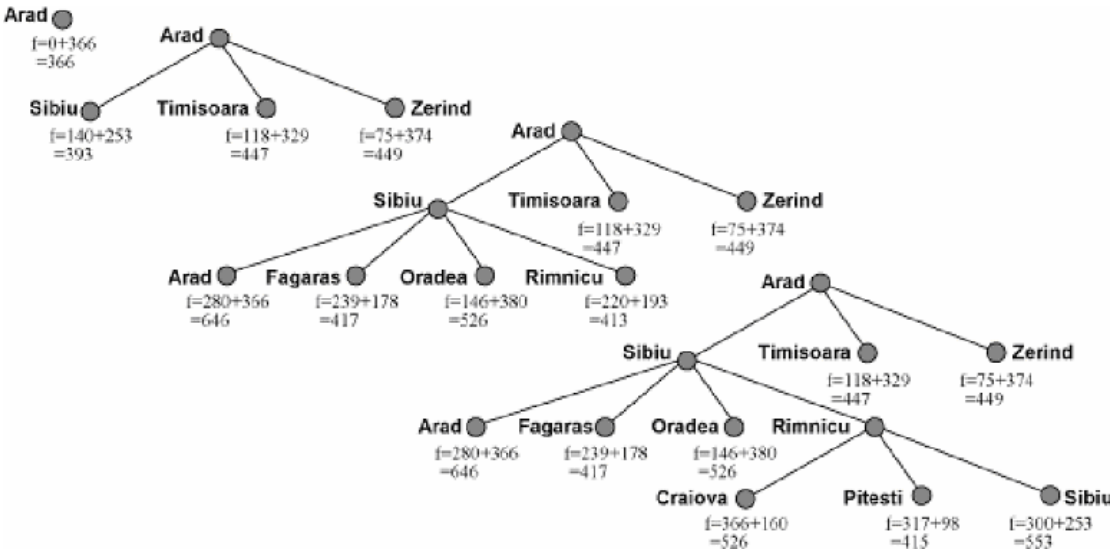
f(n) é o custo estimado da solução que passa por *n.*

A figura abaixo[1] ilustra a estratégia usada pela busca A*:



Busca A* usa *f* como função de avaliação e função heurística *h* admissível. Ex.: a função h_{DLR} é admissível porque qualquer outro caminho será maior que a linha reta entre dois pontos.

⇒ Estratégia é escolher o nó com menor valor de *f(n)*como mostrado na figura a seguir:



[1] Figura extraída do material do Prof. Fernando Gomide da Faculdade de Engenharia Elétrica e Computação da Unicamp (http://www.dca.fee.unicamp.br/~gomide/)

Propriedades da busca A*:

Será completa e ótima se a função h nunca superestimar o custo de atingir o objetivo: heurística admissível ⇒ visão otimista (custo estimado para resolver um problema é sempre menor que ele realmente é). Se *h* é admissível então *f(n)* nunca superestima o custo real da melhor solução a partir de n ⇒ BUSCA A*.

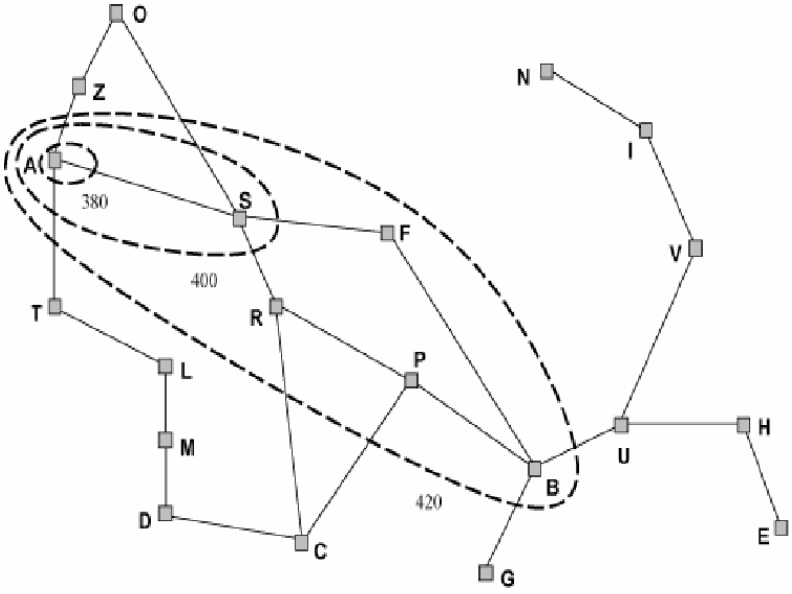
O algoritmo é monotônico, ou seja, o custo de cada nodo gerado no mesmo caminho nunca diminui. No entanto, pode haver problemas onde $f(n') < f(n)$, sendo *n* é o pai de *n'*. Considere o seguinte exemplo:

$$f(n) = g(n) + h(n) = 5 + 4 = 9$$

$$f(n') = g(n') + h(n') = 6 + 2 = 8$$

Para garantir a monotocidade de *f*, todo momento que um nodo for gerado, usa-se a equação *PathMax*: $f(n') = max(f(n), g(n') + h(n'))$, baseando na premissa de que todo caminho que passa por *n'*, passa também pelo pai *n*.

Se *f* nunca diminui ao longo de qualquer caminho, é possível desenhar contornos no espaço de estados, conforme mostrado na figura a seguir, onde é ilustrado contornos com $f = 380$, $f = 400$ e $f = 420$.



Semelhante a busca em largura, mas ao invés de geração de nodos da árvore por profundidade, o que é considerado é o contorno gerado por *f*. Para a busca com custo uniforme (A* usando $h(n) = 0$) os contornos são circulares ao redor do estado inicial.

4.4 INVENTANDO FUNÇÕES HEURÍSTICAS

Alguns questionamentos:

- Como definir uma boa função heurística *h*?
- É possível um processo de computador definir uma heurística automaticamente?

Propriedades:

- Função heurística depende de cada problema específico.
- Deve ser admissível, ou seja não superestimar o custo real da solução.

Existem algumas estratégias para definir *h*:

1. Relaxar restrições do problema
2. Usar informações estatísticas - executar a busca sobre um número aleatório de problemas (configurações aleatórias).
3. Identificar os atributos mais relevantes do problema (exige aprendizagem).

RELAXANDO O PROBLEMA

Problemas com menos restrições nas ações/operadores são chamados de ***Problemas relaxados*** (*"relaxed problem"*), em outras palavras, problema relaxado é uma versão simplificada do problema original, onde os operadores são menos restritivos.

O custo de uma solução do problema relaxado é uma boa heurística para o problema original.

Considere o Jogo de 8 peças: se as regras do jogo fossem diferentes e uma peça pudesse ser movida para qualquer lugar e não mais para um espaço vazio adjacente $\Rightarrow$ o número de posições erradas (h_1) daria o número de passos para a solução mais curta. Semelhantemente, se uma peça pudesse ser movida em qualquer direção $\Rightarrow$ somatório da distância de cada peça (h_2) daria o número de passos para a solução mais curta.

Três problemas relaxados podem ser gerados removendo uma ou duas condições das regras do jogo de 8 peças:

Um número pode mover-se de A para B se A é adjacente a B (h_2).

Um número pode mover-se de A para B se B está vazio.

Um número pode mover-se de A para B (h_1).

Obs: ABSOLVER foi criado para gerar, automaticamente, heurísticas a partir da definição de um problema usando problemas relaxados e outras técnicas.

Resumidamente, as seguintes heurísticas são possíveis para o Jogo de 8 Peças:

h_1 = número de peças que estão em posição errada

h_2 = distância de *Manhattan* = soma das distâncias de cada peça a posição correta.

USANDO INFORMAÇÃO ESTATÍSTICA

As funções heurísticas podem ser "melhoradas" com informações estatísticas:

- Executar a busca com um conjunto de treinamento (por exemplo, 100 configurações diferentes do jogo), e computar os resultados
- Se em 90% dos casos, quando $h(n) = 14$, a distância real da solução é 18, então quando o algoritmo encontrar 14 para o resultado da função, substitui-se esse valor por 18.

A utilização de informações estatísticas elimina admissibilidade, mas expande menos nós.

ESCOLHENDO FUNÇÕES HEURÍSTICAS

É sempre melhor usar uma função heurística com valores mais altos, contando que ela seja admissível.

Considera-se que h_i domina h_k, se $h_i(n) \geq h_k(n)$, para todo n no espaço de estados. No jogo de 8 peças, h_2 domina h_1.

Caso existam muitas funções heurísticas para o mesmo problema, e nenhuma delas domine as demais, deve-se usar uma heurística composta $h(n) = max(h_1(n), h_2(n), ..., h_m(n)) \Rightarrow$ considerando que todas as h_i são admissíveis, h será admissível e dominará cada função h_i individualmente.

Qualidade da Função Heurística é medida através do fator de expansão efetivo (b^*):

$$N = 1 + b^* + (b^*)^2 + ... + (b^*)^d \text{ , onde}$$

N é o total de nodos expandidos para uma instância do problema e d é a profundidade da solução.

$\Rightarrow$ Uma boa heurística terá b^* muito próximo de 1.

4.5 TÉCNICAS DE BUSCA COM MEMÓRIA LIMITADA

Vimos que um dos problemas da busca A* é a utilização de memória. Diante disso, veremos dois algoritmos com limitação de memória:

IDA* *(Iterative Deepening A*):* Igual ao aprofundamento iterativo, porém com limite na função de avaliação *f* ao invés da profundidade *d*.

SMA* *(Simplified Memory Bounded A*):* Similar ao A*, mas restringe o tamanho da fila para que caiba na memória. O número de nós guardados em memória é fixado previamente.

IDA* (ITERATIVE DEEPENING A*):

Usa o aprofundamento iterativo para poupar memória. A busca em profundidade é modificada para usar um limite em *f* ao invés de profundidade. Assim, cada iteração expande todos os nós dentro do contorno para o limite *f* atual, estendendo gradualmente a linha para descobrir o próximo contorno (completo e ótimo).

Como não armazena seu passado é condenado a repeti-lo $\Rightarrow$ os únicos estados armazenados são os do limite em *f* atual.

O algoritmo IDA* tem dificuldade em domínio mais complexo: problema em que o valor da heurística é diferente para cada estado $\Rightarrow$ cada contorno poderá incluir um nó. Assim, se a busca A* expande *n* nós, a busca IDA* terá *n* iterações com *n* contornos, que irá expandir $1 + 2 + ... + n^2 \Rightarrow O(n^2)$.

A dificuldade da busca IDA* pode ser contornada utilizando um limitador de memória

SMA* (SIMPLIFIED MEMORY BOUNDED A*)

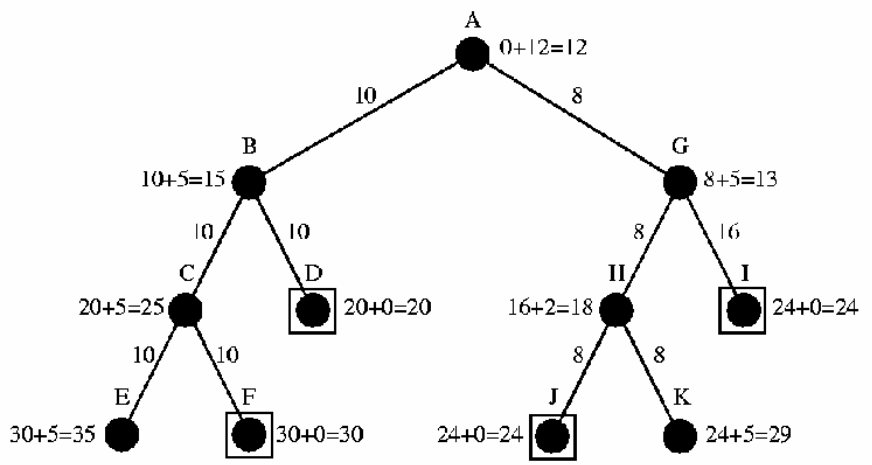
Similar ao A*, mas restringe o tamanho da fila para que caiba na memória. O número de nós guardados em memória é fixado previamente. Durante a busca, utiliza toda a memória disponível, evita estados repetidos enquanto a memória permitir, e é completa e ótima se a memória disponível for suficiente para guardar o caminho da solução mais rasa. Se não puder ser ótima, acha a melhor possível dada a memória disponível.

Tendo memória suficiente para realizar a busca, armazena a árvore completa baseado na premissa de que é mais fácil manter o nó do que ter que gerá-lo novamente.

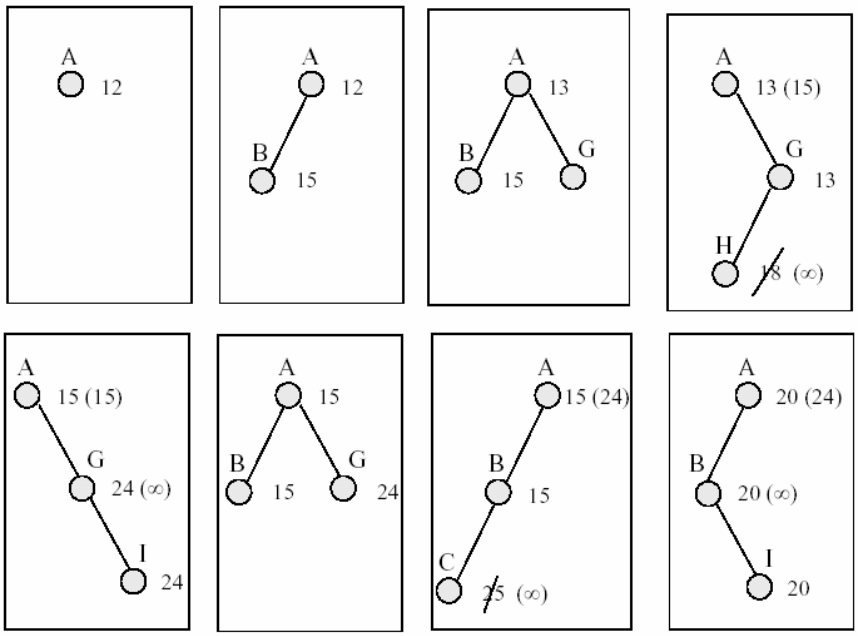
Funcionamento:

Antes de gerar um sucessor, o algoritmo verifica se existe memória suficiente. Caso não exista o algoritmo retira um dos nós da fila para poder gerar o sucessor. Os nós retirados são os menos promissores e são chamados de "nós esquecidos". Para evitar re-explorar nós, é guardada informação do melhor caminho na sub-árvore esquecida.

Considere o seguinte exemplo:



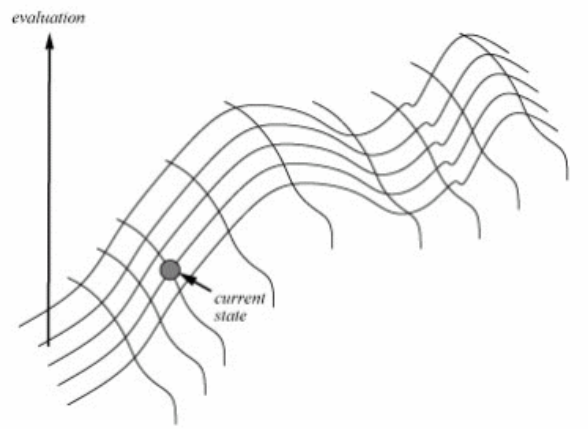
As soluções estão marcadas com um quadrado, os custos reais estão ao lado das arestas e a função de avaliação (custo acumulado mais estimativa) ao lado dos nodos. O funcionamento do algoritmo SMA* para esse problema é mostrado na figura a seguir:



4.6 ALGORITMO DE MELHORIAS ITERATIVAS (AMI) - ITERATIVE IMPROVEMENT ALGORITHMS

A idéia é começar com o estado inicial (configuração completa), e melhorá-lo iterativamente. Os estados estão representados sobre uma superfície (função de avaliação) onde a altura de qualquer ponto na superfície corresponde a sua avaliação do estado naquele ponto.

O algoritmo se "move" pela superfície em busca de pontos mais altos/baixos (dependendo do objetivo): o ponto mais alto/baixo (máximo/mínimo global) corresponde a solução ótima (nodo onde a função de avaliação atinge seu valor máximo/mínimo)



Esses algoritmos guardam apenas o estado atual, e não vêem além dos vizinhos imediatos do estado, contudo muitas vezes são os melhores métodos para tratar problemas reais muito complexos.

Aplicações: Problemas de Otimização:

Localização de facilidades: distribuir facilidades maximizando a satisfação dos demandantes:

- escolher a localização de torres de celular de forma a minimizar o risco de algum ponto ficar sem cobertura.
- escolher a localização de armazéns de forma a minimizar o deslocamento dos caminhões de entrega até os clientes.

Coloração de grafos: colorir mapas usando o mínimo de cores e sem usar mesmas cores em países vizinhos. Exemplo: distribuir frequências em ERBs de forma a aumenta a possibilidade de conexões entre telefones celulares sem permitir interferências.

Técnicas Aplicáveis:

- Matemáticas: Programação Linear, Indução Matemática, ...
- Heurísticas: Hill-Climbing, Simulated Annealing, Algoritmos Genéticos, Redes Neurais Artificiais, ...

Dentro dos métodos heurísticos, veremos duas classes de algoritmos:

- Hill-Climbing, gradiente ascendente ou subida da encosta: só faz modificações que melhoram o estado atual.
- Simulated Annealing: pode fazer modificações que pioram o estado temporariamente para possivelmente melhorá-lo no futuro (ideal para problema que encontram muitas planícies e muitos máximos locais).

HILL-CLIMBING

O algoritmo não mantém uma árvore de busca, guarda apenas o estado atual e sua avaliação. É simplesmente um "loop" que fica se movendo na direção crescente (para maximizar) ou decrescente (para minimizar) da função de avaliação. Quando existe mais de um "melhor" sucessor para o nó atual, o algoritmo faz uma escolha aleatória. O algoritmo move-se sempre na direção que apresenta maior taxa de variação de *f*.

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
inputs: problems, a problema
local variables: current, a node
                 neighbor, a node

current ← MAKE-NODE(INITIAL-NODE[problem])
loop do
       neighbor ← a highest-valued successor of current
       if VALUE[neighbor] ≤ VALUE[current] then
              return STATE[current]
       current ← neighbor
```

Pode gerar três problemas, que podem levar ao desvio do caminho a solução: Máximo local, Planície (Platôs) e Encosta (Picos):

Máximos Locais: são picos mais baixos do que o pico mais alto no espaço de estados (solução ótima) ⇒ o algoritmo pára no máximo local porque a função de avaliação é menor

para todos os estados filhos do estado atual, apesar do objetivo estar em um ponto mais alto. Só é permitido movimento com taxa crescente de avaliação.

Planícies/Platôs: região do espaço de estados onde a função de avaliação dá o mesmo resultado, ou seja, $f(n) = f(filhos(n))$. O algoritmo pára depois de algumas tentativas.

Picos: encostas podem ter lados muito íngremes e o algoritmo chega ao topo com facilidade.

Nos casos acima, o algoritmo chega a um ponto de onde não faz mais progresso. ⇒ Solução: reinício aleatório (algoritmo que realiza uma série de buscas a partir de estados iniciais aleatórios).

Cada busca é executada:

- até que um número máximo de iterações estipulado seja atingido, ou;
- até que os resultados encontrados não apresentem melhora significativa.

Propriedades do algoritmo:

O algoritmo pode chegar a uma solução ótima quando iterações suficientes forem permitidas. O sucesso desse método depende muito do formato da superfície do espaço de estados: se existem poucos estados máximos locais, o reinício aleatório encontra uma boa solução rapidamente. Caso contrário, o custo de tempo é exponencial.

SIMULATED ANNEALING

Semelhante a subida da encosta, porém oferece meios para se escapar de máximos locais: ao invés de recomeçar em um local aleatório, retrocede para situações piores, no entanto não garante que a solução encontrada seja a melhor possível.

Nas iterações iniciais, não escolhe necessariamente o "melhor" passo, e sim um movimento aleatório: se a situação melhorar, esse movimento é escolhido. Caso contrário, associa a esse movimento uma probabilidade de escolha menor que 1.

Em resumo, o que o algoritmo faz é atribuir uma certa "energia" inicial ao processo de busca, permitindo que além de subir encostas, o algoritmo seja capaz de descer encostas e percorrer planícies, se a energia for suficiente. A energia é diminuída ao longo do tempo, o que faz com que o processo vá se estabilizando em algum máximo que tenha maior chance de ser a solução do problema.

CONCLUSÃO

- Solução de problemas usando técnicas de busca heurística: dificuldade em definir e usar a função de avaliação ⇒ compromisso entre tempo gasto na seleção de um nó e redução do espaço de busca.

- Achar o melhor nó a ser expandido a cada passo pode ser tão difícil quanto o problema de busca em geral.

Aplicável em situações onde o caminho do estado inicial até o atual é irrelevante. O estado atual contém toda informação necessária para orientar a busca. Alguns problemas conhecidos podem ser resolvidos de forma gradativa como o de 8 rainhas: começar com todas as 8 rainhas no tabuleiro e movê-las tentando reduzir o número de ataques entre elas.

Busca Online e ambientes desconhecidos

Até agora nos concentramos em algoritmos de busca offline, onde uma solução completa era computada antes de executar um passo no ambiente real, e então executa a sem obter novos estímulos. Diferentemente, a busca online intercala planejamento e ação: primeiro escolhe e executa a ação escolhida, então observa o ambiente e escolhe a próxima