INTRODUÇÃO À INTELIGÊNCIA ARTIFICIAL

PARTE 1. ESTRATÉGIAS DE BUSCA

1. Sistemas de Produção

1.1. Introdução

Um sistema de produção é um formalismo computacional, onde pode ser identificado claramente três componentes principais: uma base de dados global, um conjunto de regras de produção e um sistema de controle. Os sistemas de produção capturam a essência da operação de muitos sistemas de Inteligência Artificial (IA).

A *base de dados global* é a estrutura de dados central usada por um sistema de produção em IA. Dependendo da aplicação, esta base de dados pode ser uma matriz de números ou uma estrutura de arquivo indexado relacional.

As *regras de produção* operam na base de dados global. Cada regra tem uma pré-condição que ou é satisfeita ou não pela base de dados global. Se a pré-condição for satisfeita, a regra pode ser aplicada. A aplicação de uma regra *altera* a base de dados. O *sistema de controle* escolhe qual regra aplicável deve ser aplicada e termina a computação quando uma condição de terminação na base de dados global for satisfeita.

1.2. Um exemplo: o tabuleiro de 8 pedras

Para o tabuleiro de 8 pedras, a base de dados global seria representada por uma matriz de inteiros (pode-se imaginar o -1 representando o espaço). As regras de produção seriam as regras que aplicadas a uma configuração da base de dados, modificariam esta configuração. Para este exemplo, onde se tem 8 pedras distintas, e cada pedra pode ser movimentada para cima, para baixo, para esquerda e para direita, dever-se-ia ter 8 x 4 = 32 regras de produção. Mas pode-se optar por "movimentar" o branco (vazio), que é único, ao invés de se mover 8 pedras diferentes. Neste caso, o conjunto de regras de produção reduziria a 4 regras, a saber, mover o branco para cima (R1), para baixo (R2), para direita (R3) e para esquerda (R4). esta base de dados, transformam-na. Suponha que se deseje partir de uma configuração inicial *283-164-7b5* e chegar em *123-8b4-765*, como mostra a figura abaixo:

Inicial Meta

1.3. O Procedimento Básico

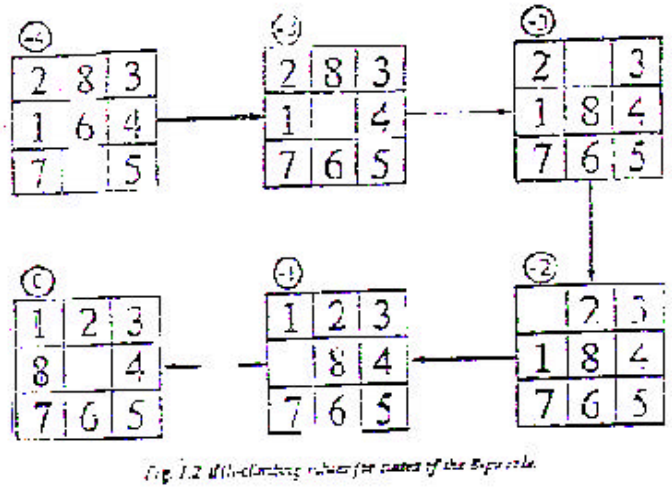
O algoritmo que vai implementar esta busca de aplicação de regras num sistema de produção é o seguinte:

Procedimento *PRODUÇÃO*

1 *DATA* ← base de dados inicial
2 *até DATA* satisfazer a condição de terminação, faça:
3 *início*
4 selecione alguma regra, R, no conjunto de regras que possa ser aplicada a *DATA*
5 *DATA* ← resultado da aplicação de R a *DATA*
6 *fim*

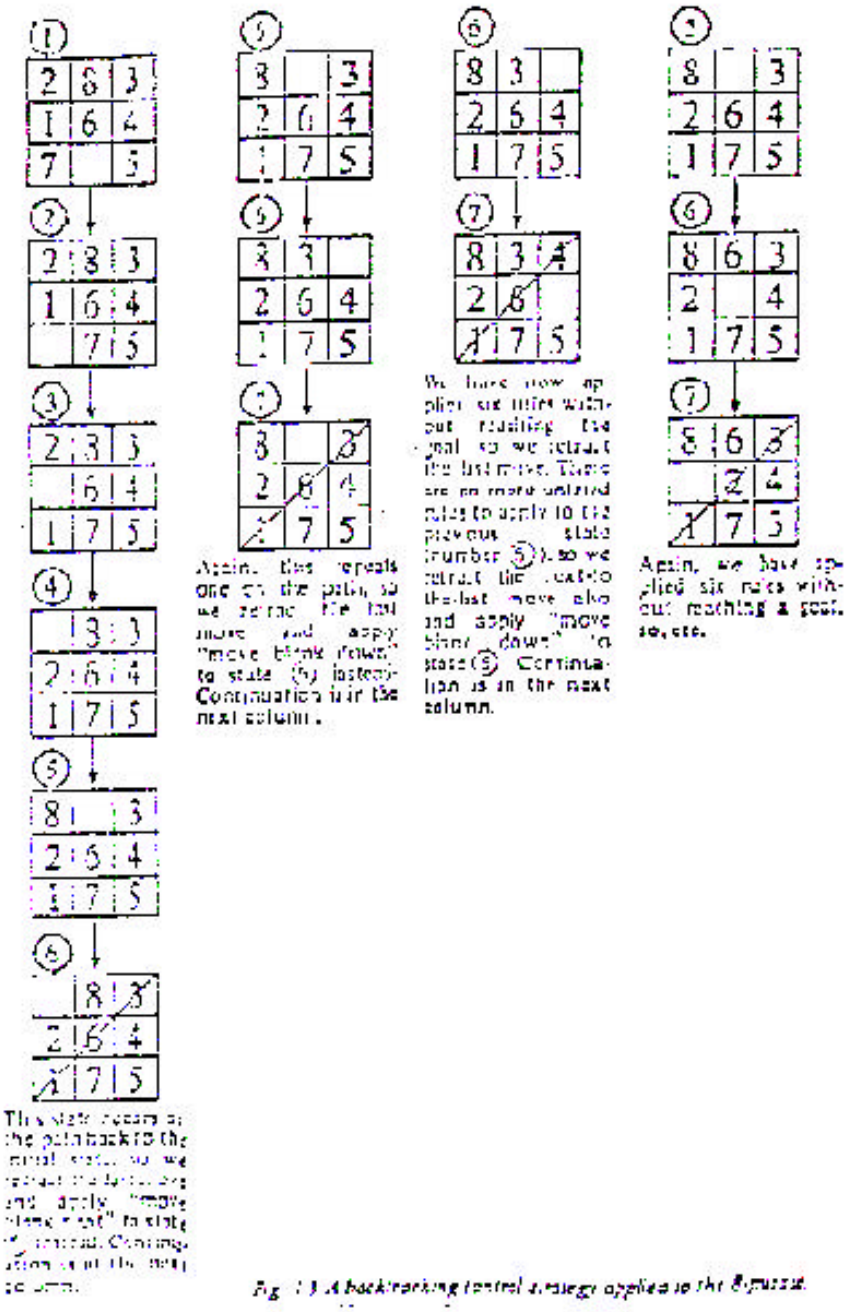
1.4. Controle

O procedimento acima é não-determinístico porque não se especificou precisamente como se selecionará uma regra aplicável na linha 4. A seleção de regras e a manutenção da trilha das seqüências de regras já tentadas e as bases de dados que elas produzem constituem o que se chama de *estratégia de controle para sistemas de produção*. Em muitas aplicações de IA, a informação disponível para a estratégia de controle não é suficiente para permitir a seleção da maioria das regras apropriadas no passo 4. A operação dos sistemas de produção de IA pode então ser caracterizada como um processo de busca nos quais as regras são tentadas até que alguma seqüência delas seja encontrada que produza uma base de dados que satisfaça a condição de terminação. As estratégias de controle eficientes requerem conhecimento suficiente sobre o problema a ser resolvido tal que a regra selecionada no passo 4 tenha uma boa chance de ser a mais apropriada. Uma estratégia boa (informada) para o problema do tabuleiro de 8 pedras é a estratégia *Hill-Climbing*, que consiste em aplicar uma regra que não tire uma pedra que já esteja na sua posição final ((Nilsson, 1982):



Existem dois tipos principais de estratégias de controle: irrevogável e tentativa. Num regime de controle *irrevogável*, uma regra aplicável é selecionada e aplicada irrevogavelmente sem provisão para reconsideração mais tarde. Num regime de controle *tentativo*, uma regra aplicável é selecionada (ou arbitrariamente ou talvez com uma boa razão), a regra é aplicada, mas é feita provisão para retornar mais tarde a este ponto para aplicar alguma outra regra.

Vai-se distinguir dois tipos de regimes de controle tentativo. Num, chamado *backtracking* (veja aplicação para o problema do tabuleiro de 8 pedras na figura abaixo (Nilsson, 1982)), um ponto de backtracking é estabelecido quando uma regra é selecionada. Caso a computação subseqüente encontre dificuldade de produzir uma solução, o estado da computação reverte ao ponto de backtracking prévio, onde uma outra regra é aplicada, e o processo continua.



No segundo tipo de regime de controle tentativo, chamado de controle por *busca em grafo*, a provisão é feita para manter a trilha dos efeitos de muitas seqüências de regras simultaneamente. Vários tipos de estrutura de grafos e procedimentos de busca em grafos são usados neste tipo de controle.

2. Estratégias de Busca para sistemas de produção de IA

2.1. Backtracking

Em muitos problemas de interesse, a aplicação de uma regra imprópria pode atrasar substancialmente ou inibir uma terminação bem sucedida. Nestes casos, deseja-se uma estratégia de controle que possa tentar uma regra e, se mais tarde descobrir que esta regra é imprópria, voltar e tentar uma outra regra ao invés daquela.

O processo de backtracking é uma forma na qual a estratégia de controle pode ser tentativa. Uma regra é selecionada, e se não levar a uma solução, os passos intermediários são "esquecidos" e uma outra regra é selecionada. Formalmente, a estratégia de backtracking pode ser usada a despeito de quanto conhecimento está disponível na hora de selecionar a regra. Se não houver conhecimento nenhum disponível, as regras podem ser selecionadas de acordo com algum esquema arbitrário. Por último, o controle retornará para selecionar a regra apropriada. Obviamente, se puder ser usado conhecimento confiável para seleção de regras, o backtracking para considerar regras alternativas ocorrerá com menor freqüência, e o processo como um todo será mais eficiente.

Um procedimento recursivo simples captura a essência da operação de um sistema de produção sob controle de backtracking. Este procedimento, chamado BACKTRACK, pega um único argumento, DATA, inicialmente com o valor igual a base de dados global do sistema de produção. Até a terminação bem sucedida, o procedimento retorna uma lista de regras, que, se aplicada em seqüência à base de dados inicial, produz uma base de dados satisfazendo a condição de terminação. Se o procedimento parar sem achar tal lista de regras, ele retorna FAIL.

2.1.1. Procedimento Recursivo BACKTRACK (DATA)

1 se TERMO(DATA), retorne □; TERMO é um predicado verdadeiro para argumentos que satisfazem a condição de terminação do sistema de produção. Até a terminação bem sucedida, □, a lista vazia, é retornada.

2 se DEADEND(DATA), retorne FAIL; DEADEND é um predicado verdadeiro para argumentos que são sabidos não pertencerem ao caminho para uma solução. Neste caso, o procedimento retorna o símbolo FAIL (falha).

3 REGRAS ← REGRASAPL(DATA); REGRASAPL é uma função que computa as regras aplicáveis a seu argumento e as ordena (ou arbitrariamente, ou de acordo com mérito heurístico).

4 LOOP: se NULL(REGRAS), retorne FAIL; se não houver (mais) regras a aplicar, o procedimento falha.

5 R ← PRIMEIRA(REGRAS); a melhor das regras aplicáveis é selecionada.

6 REGRAS ← CAUDA(REGRAS); a lista de regras aplicáveis é diminuída removendo a recém selecionada.

7 RDATA ← R(DATA); a regra R é aplicada para produzir uma nova base de dados.

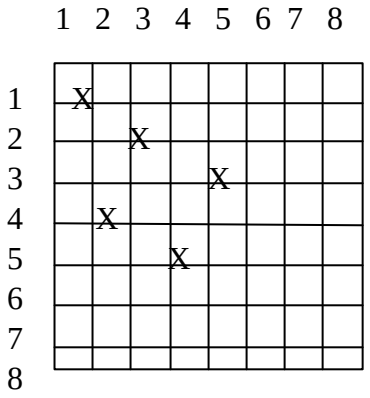
8 CAMINHO ← BACKTRACK(RDATA); BACKTRACK é chamada recursivamente na nova base de dados.

9 se CAMINHO = FAIL, vá para LOOP; se a chamada recursiva falha, tente uma outra regra.

10 retorne CONS(R,CAMINHO); de outra forma, passe a lista de regras bem sucedidas, adicionando R à frente da lista.

2.1.2. Exemplo: o problema das 8 rainhas

Imagine um tabuleiro de xadrez (matriz 8 por 8). O problema das 8 rainhas consiste em colocar uma rainha em cada linha de tal forma que uma rainha não "coma" a outra.



Suponha a configuração acima. O algoritmo deve agora tentar colocar uma rainha na linha 6. Não é possível. *Backtrack* para realocar a rainha da linha 5, movendo-a para a coluna 8. Ainda assim não é possível realocar a 6. Logo, *backtrack* para a linha 4 e assim por diante.

2.2. Busca em Grafo

Os grafos (ou mais especialmente, as árvores) são estruturas extremamente úteis para manter a trilha dos efeitos de muitas seqüências de regras.

Nas estratégias de backtracking, o sistema de controle efetivamente esquece qualquer caminho que resulta em falha. Apenas o caminho correntemente sendo estendido é armazenado explicitamente. Um procedimento mais flexível envolve o armazenamento explícito de todos os caminhos de tal forma que qualquer um deles pode ser candidato a futura extensão.

2.2.1. Notação de Grafos

Um *grafo* consiste de um conjunto (não necessariamente finito) de *nós*. Certos pares de nós são conectados por *arcos*, e estes arcos são *direcionados* de um membro do par ao outro. Tal

grafo é chamado de *grafo direcionado*. Para este modelo, os nós são rotulados por base de dados e os arcos são rotulados por regras. Se um arco é direcionado do nó n_i para o nó n_j, então o nó n_j é o *sucessor* do nó n_i, e o nó n_i é o *pai* do nó n_j. Nos grafos de interesse, um nó pode ter apenas um número finito de sucessores. (Os sistemas de produção têm apenas um número finito de regras aplicáveis.) Cada nó de um par pode ser sucessor do outro; neste caso o par de arcos direcionados é algumas vezes substituído por uma *margem*.

Uma *árvore* é um caso especial de um grafo no qual cada nó tem, no máximo, um pai. Um nó na árvore que não tem pai é chamado de *nó raiz*. Um nó na árvore que não tem sucessores é chamado de *nó folha*. Diz-se que o nó raiz é de *profundidade zero*. A profundidade de qualquer outro nó na árvore é, por definição, a profundidade de seus pais mais 1.

Uma seqüência de nós $(n_{i1},n_{i2},...,n_{ik})$, com cada n_{ij} um sucessor de n_{ij-1} para $j = 2,...,k$, é chamada de um *caminho de comprimento k* do nó n_{i1} para o nó n_{ik}. Se um caminho existe entre o nó n_i para o nó n_j, então o nó n_j é acessível a partir do nó n_i. O nó n_j é então um *descendente* do nó n_i, e o nó n_i é um *ancestral* do nó n_j. Note que o problema de achar uma seqüência de regras para transformar uma base de dados em outra é equivalente ao problema de achar um caminho num grafo.

Freqüentemente é conveniente atribuir custos positivos aos arcos, para representar o custo da aplicação da regra correspondente. Usa-se a notação $c(n_i, n_j)$ para denotar o custo de um arco direcionado do nó n_i para o nó n_j. O custo de um caminho entre dois nós é a soma dos custos de todos os arcos que conectam os nós no caminho. Em alguns problemas, deseja-se achar o caminho que tenha custo *mínimo* entre dois nós.

No tipo mais simples de problema, deseja-se achar um caminho (talvez tendo custo mínimo) entre um nó dado *s*, representando a base de dados inicial e um outro nó dado *t*, representando alguma outra base de dados. A situação mais usual envolve achar um caminho entre um nó s e *qualquer* membro do conjunto de nós $\{t_i\}$ que representa as bases de dados que satisfazem a condição terminal. Chama-se o conjunto $\{t_i\}$ o conjunto meta e cada nó *t* em $\{t_i\}$ é um *nó meta.*

2.2.2. Um procedimento geral de busca em grafo

Procedimento GRAPHSEARCH

1 Crie um grafo de busca, G, consistindo somente do nó inicial, *s*. Ponha *s* numa lista chamada OPEN.

2 Crie uma lista chamada CLOSED que está inicialmente vazia.

3 LOOP: se OPEN está vazia, saia com falha.
4 Selecione o primeiro nó em OPEN, remova-o de OPEN, o ponha-o em CLOSED. Chame este nó de *n*.

5 Se *n* é um nó meta, saia com sucesso com a solução obtida traçando um caminho entre os ponteiros de *n* para *s* em G. (Os ponteiros são estabelecidos no passo 7.)

6 Expanda o nó *n*, gerando o conjunto, M, de seus sucessores que não são ancestrais de *n*. Instale estes membros de M como sucessores de *n* em G.

7 Estabeleça um ponteiro para *n* a partir daqueles membros de M que ainda não estejam nem em OPEN nem em CLOSED. Adicione estes membros de M a OPEN. Para cada membro de M que já esteja em OPEN ou CLOSED, decida se direciona ou não seu ponteiro para *n*. Para cada membro de M já em CLOSED, decida para cada um de seus descendentes em G se redireciona ou não o seu ponteiro.

8 Reordene a lista OPEN, de acordo com alguma esquema arbitrário ou de acordo com mérito heurístico.

9 Vá para LOOP.

2.2.3. Procedimentos não informados de Busca em Grafos

Se nenhuma informação heurística do domínio do problema é usada para ordenar os nós em OPEN, algum esquema arbitrário deve ser usado no passo 8 da algoritmo. O procedimento de busca resultante é chamado *não informado*. Em IA, normalmente não se interessa por procedimentos não informados. Os procedimentos não informados mais comuns são: busca em profundidade e busca em largura.

O primeiro tipo de busca não informada ordena os nós em OPEN na ordem descendente de sua profundidade na árvore de busca. Os nós mais profundos são colocados primeiro na lista. Nós de profundidade igual são ordenados arbitrariamente. A busca que resulta de tal ordenação é chamada de busca *por profundidade* (“depth-first”) (veja aplicação para o tabuleiro de 8 pedras na figura abaixo (Nilsson, 1982)) porque o nó mais profundo na árvore de busca é sempre selecionado para expansão. Para prevenir o processo de busca de ficar “rodando” em algum caminho inútil para sempre, um limite de profundidade é necessário. Nenhum nó cuja profundidade na árvore de busca excede este limite é gerado.

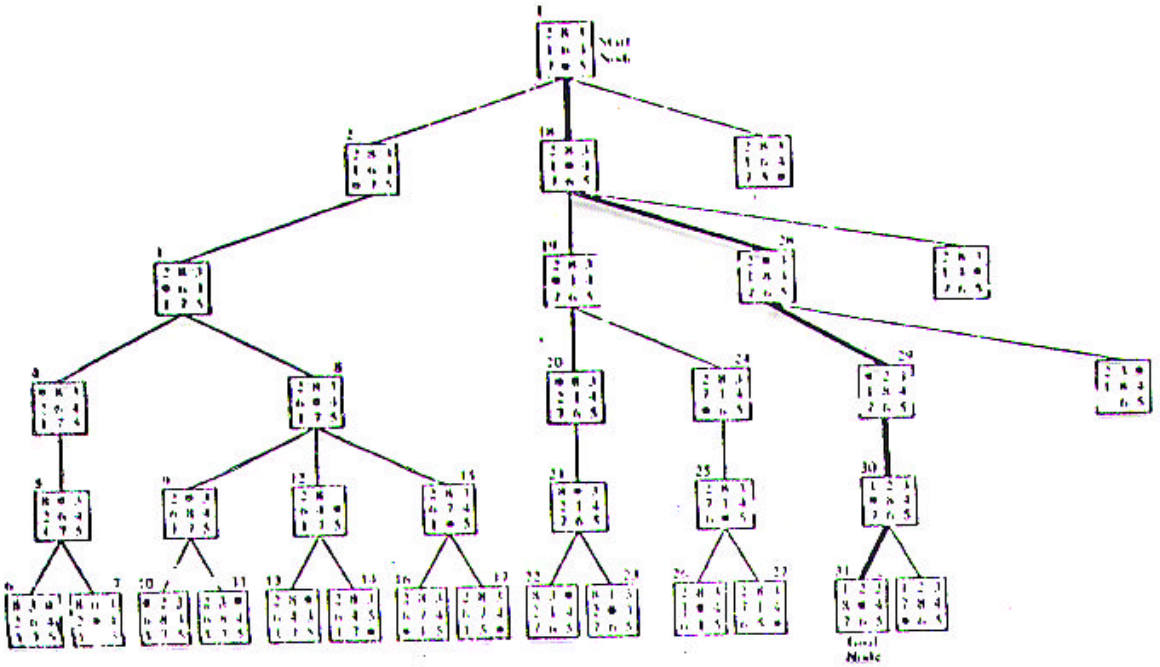


Fig. 2.6 A search tree produced by a depth-first search.

O procedimento de busca em profundidade gera novas bases de dados em uma ordem similar à gerada por uma estratégia de controle backtracking não informada. A correspondência seria exata se o processo de busca em grafo gerasse apenas um sucessor de cada vez. Usualmente, a implementação backtracking é preferida à versão busca em profundidade do GRAPHSEARCH porque o backtracking é mais simples de implementar e envolve menos armazenamento. (As estratégias de backtracking salvam apenas um caminho ao nó meta; elas não salvam o registro inteiro da busca como fazem as estratégias de busca em grafo por profundidade.)

Algoritmo: Busca em Profundidade

1. Se o estado inicial é um estado meta, saia e retorne com sucesso.
2. Caso contrário, faça o seguinte até que sucesso ou falha seja assinalado:
   (a) Gere um sucessor, *E*, do estado inicial. Se não houver mais sucessores, assinale falha.
   (b) Chame Busca por Profundidade com *E* como o estado inicial.
   (c) Se sucesso é retornado, assinale sucesso. Caso contrário, continue neste loop.

O segundo tipo de procedimento de busca não informada ordena os nós em OPEN na ordem crescente de sua profundidade na árvore de busca. (Novamente, para promover o término antecipado, nós metas devem ser postos imediatamente no ínicio de OPEN.) A busca que resulta de tal ordenação é chamada de busca *em largura* (“breadth-first”) (veja aplicação para o tabuleiro de 8 pedras na figura abaixo (Nilsson, 1982)) porque a expansão dos nós na árvore de busca acontece ao longo do “contorno” de profundidade igual.

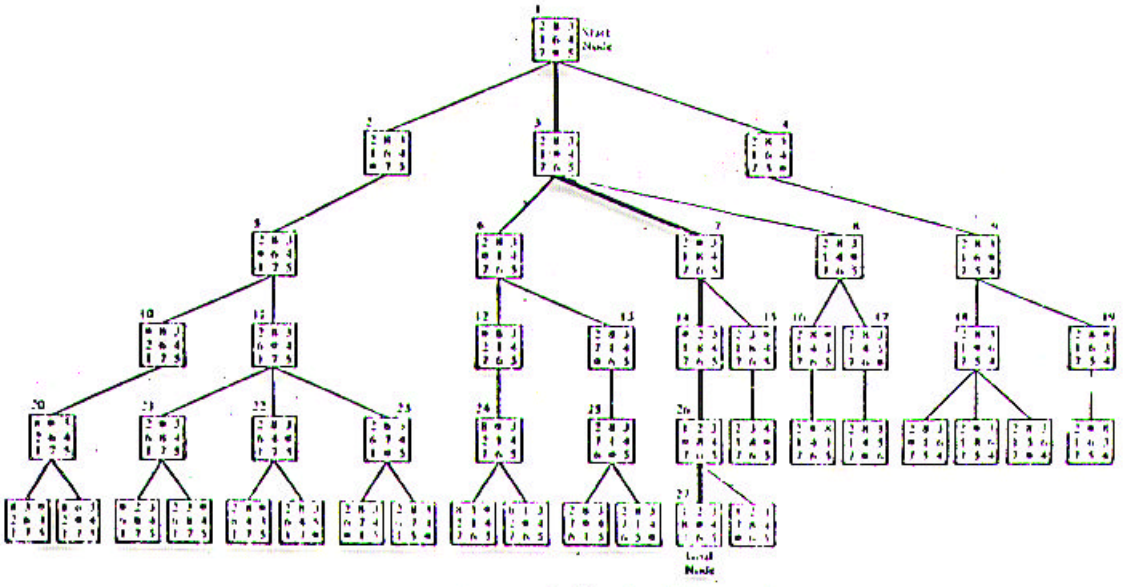


Fig. 2.7 A search tree produced by a breadth first search.

A busca por largura garante achar um caminho de comprimento mais curto a um nó meta, se este caminho existir. (Se nenhum caminho existe, o método falhará para grafos finitos ou nunca terminará para grafos infinitos.)

Algoritmo: Busca em Largura

1. Crie uma variável chamada *LISTA-DE-NÓS* e faça-a igual ao estado inicial.
2. Até que um estado meta seja achado ou *LISTA-DE-NÓS* esteja vazia faça:
   (a) Remova o primeiro elemento da *LISTA-DE-NÓS* e chame-o de *E*. Se *LISTA-DE-NÓS* estiver vazia, saia.
   (b) Para cada forma que cada regra possa casar com o estado descrito em *E* faça:
      i. Aplique a regra para gerar um estado novo.
      ii. Se o estado novo é um estado meta, saia e retorne este estado.
      iii. Caso contrário, adicione o estado novo ao final de *LISTA-DE-NÓS*.

2.2.4. Procedimentos Heurísticos de Busca em Grafos

Os métodos de busca não informados, seja a busca em largura ou em profundidade, são métodos exaustivos para achar caminhos para o nó meta. Em princípio, estes métodos provêem uma solução ao problema de achar um caminho, mas eles são freqüentemente inviáveis para usar o controle dos sistemas de produção porque a busca expande muitos nós antes que um caminho seja encontrado. Como sempre existem limites práticos à quantidade de tempo e armazenamento disponíveis para usar na busca, alternativas mais eficientes à busca não informada devem ser encontradas.

Para muitas tarefas é possível usar a informação dependente da tarefa para ajudar a reduzir a busca. Informação deste tipo é usualmente chamada de *informação heurística* [1], e os procedimentos de busca que a usa são chamados de *métodos heurísticos de busca*. É possível especificar a heurística que reduza o esforço de busca (abaixo daquele esforço gasto, por exemplo, pela busca em largura) sem sacrificar a garantia de achar um caminho de comprimento mínimo. Algumas heurísticas reduzem bastante o esforço de busca mas não garantem achar caminhos de custo mínimo. Em muitos problemas práticos, estamos interessado em minimizar alguma *combinação* do custo do caminho e o custo da busca necessário para obter o caminho. Além disso, nós estamos interessados em métodos de busca que minimizem esta combinação *média* sobre todos os problemas a serem encontrados. Se o custo de combinação média do método de busca 1 é menor que o custo de combinação média do método de busca 2 então o método de busca 1 tem mais *potência heurística* que o método de busca 2. Note que de acordo com esta definição, não é necessário que um método de busca com mais potência heurística garanta achar um caminho de custo mínimo.

Custos de combinação média nunca são realmente computados, porque é difícil decidir combinar custo de caminho e custo de esforço de busca e porque seria difícil definir uma distribuição de probabilidade sobre o conjunto de problemas a ser encontrado. Além disso, o problema de decidir se um método de busca tem mais potência heurística do que outro é normalmente deixado para a intuição informada, obtida das experiências reais com os métodos.

Considere o seguinte problema.

> **O problema do caixeiro viajante:** Um vendedor tem uma lista de cidades, que ele deve visitar apenas uma vez. Existem estradas diretas ligando cada par de cidades da lista. Ache a rota que o vendedor deve seguir para a viagem mais curta possível que começa e termina em uma das cidades.

Um exemplo de uma boa heurística de propósito geral que é útil para vários problemas combinatoriais é a *heurística do vizinho mais próximo*, que trabalha selecionando a alternativa localmente superior a cada passo. Aplicando-a ao problema do caixeiro viajante, produzimos o seguinte procedimento:

1. Arbitrariamente selecione uma cidade inicial.
2. Para selecionar a próxima cidade, olhe todas as cidades ainda não visitadas e selecione a mais próxima a cidade corrente. Vá a ela.
3. Repita o passo 2 até que todas as cidades tenham sido visitadas.

Este procedimento é executado em tempo proporcional a N^2, uma melhora significante sobre $N!$, que seria o tempo gasto caso fosse usada uma estratégia não informada (que geraria uma *explosão combinatorial*).

[1] A palavra *heurística* vem da palavra grega *heuriskein*, que significa "descobrir", que é também a origem de *eureka*, a famosa exclamação de Arquimedes ("Eu achei").

Existem muitas heurísticas que, ainda que não sejam tão gerais quanto a heurística do vizinho mais próximo, são úteis em uma ampla variedade de domínios. Por exemplo, considere a tarefa de descobrir idéias interessantes em alguma área específica. A seguinte heurística é freqüentemente útil:

> Se há uma função interessante de dois argumentos $f(x, y)$, olhe para o que acontece se os dois argumentos são idênticos.

No domínio da matemática, esta heurística leva à descoberta do *quadrado* se f for a função de multiplicação e leva à descoberta da função *identidade* se f for a função da união de conjuntos. Em domínios menos formais, esta mesma heurística leva à descoberta da *introspeção* se f for a função contemplar ou leva à noção de *suicida* se f for a função matar.

Sem a heurística estaríamos desesperançosamente envolvidos em uma explosão combinatorial. Só isto já poderia ser um argumento suficiente a favor de seu uso. Mas existem outros argumentos também:

- Raramente precisamos da solução ótima; uma boa aproximação usualmente serve muito bem. De fato, existe alguma evidência de que as pessoas, quando resolvem problemas, ao invés de serem otimizadoras, elas procuram a satisfação. Em outras palavras, elas procuram qualquer solução que satisfaça algum conjunto de requisitos, e assim que elas encontram um elas terminam. Um bom exemplo disto é a procura de um espaço no estacionamento. A maioria das pessoas param assim que acham um bom espaço, mesmo que haja um espaço melhor mais a frente.
- Ainda que as aproximações produzidas por heurísticas possam não ser muito boas no pior caso, os piores casos raramente acontecem na vida real. Por exemplo, ainda que muitos grafos não sejam separáveis e portanto não possam ser considerados como um conjunto de problemas menores ao invés de um único grande problema, muitos grafos que descrevem o mundo real são separáveis.
- A tentativa de entender por que uma heurística funciona, ou por que ela não funciona, freqüentemente leva a um entendimento mais aprofundado do problema.

EXERCÍCIOS PROPOSTOS

1.1. Usando busca em profundidade, concluir o problema iniciado em sala de aula, para o algoritmo GRAPHSEARCH, para o tabuleiro de 8 pedras:
- escrever o desenvolvimento completo das listas OPEN e CLOSED e dos ponteiros para o pai;
- fazer a recuperação da trajetória ótima.

1.2. Discutir vantagens e desvantagens dos métodos heurísticos e não-informados de busca em grafos.

1.3. Descrever todas as seqüências de regras que são aplicadas ao problema do tabuleiro de 8 pedras, para o procedimento "Hill-Climbing", para qualquer configuração inicial.

1.4. Representar os elementos de um sistema de produção para o exemplo do caixeiro viajante usando a notação de grafos. Considerar as possibilidades para a solução deste problema, levando em consideração um caminho ótimo (custo mínimo).

BIBLIOGRAFIA

- Nilsson, N. J. (1982). "Principles of Artificial Intelligence". Springer-Verlag.
- Rich, E. & Knight, K. (1994). Inteligência Artificial. 2ª. Edição. Makron Books/McGraw-Hill.