

# Busca competitiva

*Em que examinamos os problemas que surgem quando tentamos planejar com antecedência em um mundo no qual outros agentes estão fazendo planos contra nós.*

## 5.1 JOGOS

O Capítulo 2 examinou os **ambientes multiagentes**, em que cada agente precisa considerar as ações de outros agentes e o modo como essas ações afetam seu próprio bem-estar. A imprevisibilidade desses outros agentes pode introduzir muitas **contingências** possíveis no processo de resolução de problemas do agente, conforme discutido no Capítulo 4. Neste capítulo, abordaremos ambientes **competitivos**, em que os objetivos dos agentes estão em conflito, dando origem a problemas de **busca competitiva** — frequentemente conhecidos como **jogos**.

A **teoria de jogos** (matemática), um ramo da economia, visualiza qualquer ambiente multiagente como um jogo, desde que o impacto de cada agente sobre os outros seja “significativo”, não importando se os agentes são cooperativos ou competitivos.<sup>1</sup> Em IA, os “jogos” mais comuns são de um tipo bastante especializado — que os teóricos de jogos denominam jogos determinísticos de revezamento de dois jogadores **de soma zero** com **informações perfeitas** (como o xadrez). Em nossa terminologia, isso significa ambientes determinísticos completamente observáveis em que dois agentes agem alternadamente e em que os valores de utilidade no fim do jogo são sempre iguais e opostos (ou simétricos). Por exemplo, se um jogador ganha um jogo de xadrez, o outro jogador necessariamente perde. É essa oposição entre as funções utilidade dos agentes que gera a situação de competição.

Os jogos ocuparam as faculdades intelectuais dos seres humanos — chegando algumas vezes a um grau alarmante — desde que surgiu a civilização. Para pesquisadores de IA, a natureza abstrata dos jogos os torna um assunto atraente para estudo. É fácil representar o estado de um jogo e, em geral, os agentes se restringem a um pequeno número de ações cujos resultados são definidos por regras precisas. Jogos físicos, como críquete e hóquei sobre o gelo, têm descrições muito mais complicadas, uma faixa muito maior de ações possíveis e regras bastante imprecisas definindo a legalidade das ações. Com exceção do futebol de robôs, esses jogos físicos não atraíram muito interesse na comunidade de IA.

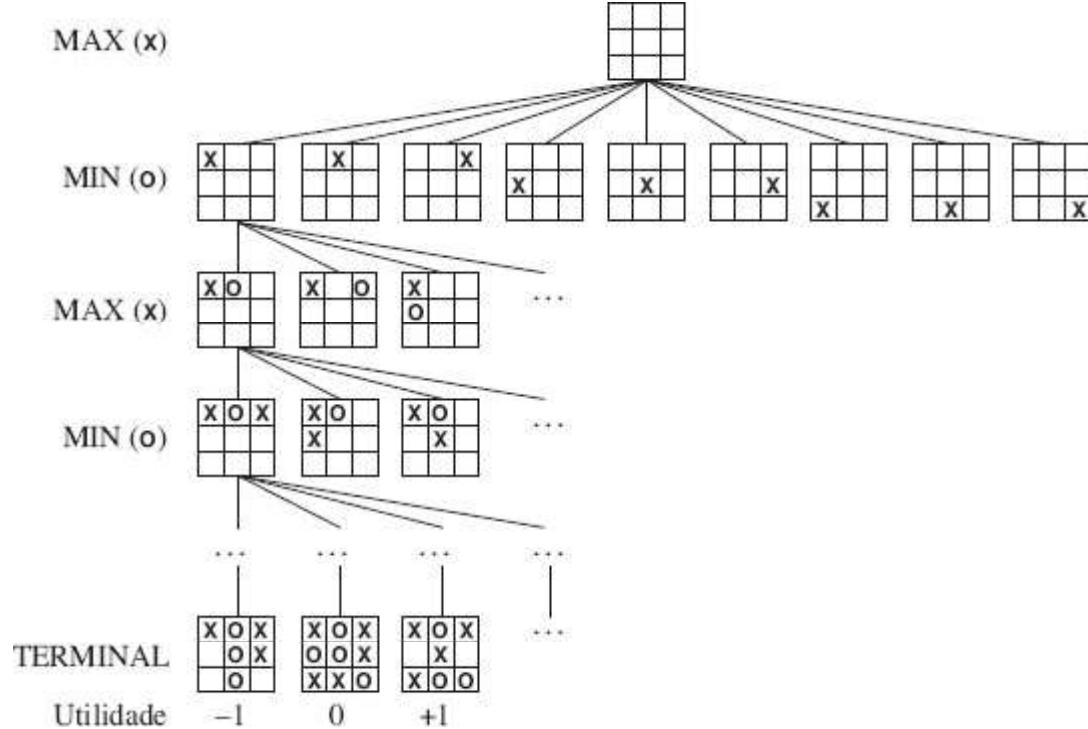
Os jogos, diferentemente da maior parte dos miniproblemas estudados no Capítulo 3, são interessantes *porque* são muito difíceis de resolver. Por exemplo, o xadrez tem um fator médio de ramificação de cerca de 35, e as partidas com frequência chegam até a 50 movimentos por cada jogador; assim, a árvore de busca tem aproximadamente  $35^{100}$  ou  $10^{154}$  nós (embora o grafo de busca tenha “apenas” cerca de  $10^{40}$  nós distintos). Os jogos, como o mundo real, exigem portanto a habilidade de tomar *alguma* decisão, até mesmo quando o cálculo da decisão *ótima* é inviável. Os jogos também penalizam a ineficiência de forma severa. Enquanto uma implementação de busca A\* com a metade da eficiência levará duas vezes mais para executar até a conclusão, um programa de xadrez com metade da eficiência no uso de seu tempo disponível provavelmente será derrubado sem piedade, mantendo-se outros aspectos inalterados. Por essa razão, a busca de jogos elaborou várias ideias interessantes sobre como fazer o melhor uso possível do tempo.

Começamos com uma definição do movimento ótimo e um algoritmo para descobri-lo. Em seguida, examinaremos técnicas para escolher um bom movimento quando o tempo é limitado. A **poda** nos permite ignorar partes da árvore de busca que não fazem diferença para a escolha final, e as **funções de avaliação** de heurísticas nos oferecem a oportunidade de fazer uma aproximação da verdadeira utilidade de um estado sem realizar uma busca completa. A Seção 5.5 discute jogos como gamão, que incluem um elemento de sorte; também discutimos o jogo de *bridge*, que inclui elementos de **informação imperfeita** porque nem todas as cartas estão visíveis para cada jogador. Por fim, veremos como os programas de jogos de última geração se comportam contra a oposição humana e, ainda, orientações para desenvolvimentos futuros.

Primeiro consideraremos jogos com dois jogadores, que chamaremos MAX e MIN por motivos que logo ficarão óbvios. MAX faz o primeiro movimento, e depois eles se revezam até o jogo terminar. No fim do jogo, os pontos são dados ao jogador vencedor e são impostas penalidades ao perdedor. Um jogo pode ser definido formalmente como uma espécie de problema de busca com os seguintes componentes:

- $S_0$ : o **estado inicial**, que especifica como o jogo é criado no início.
- JOGADORES( $s$ ): define qual jogador deve se mover em um estado.
- AÇÕES( $s$ ): retornam o conjunto de movimentos válidos em um estado.
- RESULTADO( $s, a$ ): o **modelo de transição** que define o resultado de um movimento.
- TESTE DE TÉRMINO( $s$ ): um **teste de término**, que é verdadeiro quando o jogo termina e, do contrário, falso. Os estados em que o jogo é encerrado são chamados **estados terminais**.
- UTILIDADE( $s, p$ ): uma **função utilidade** (também chamada função objetivo ou função compensação) define o valor numérico para um jogo que termina no estado terminal  $s$  por um jogador  $p$ . No xadrez, o resultado é uma vitória, uma derrota ou um empate, com valores +1, 0 ou  $\frac{1}{2}$ . Alguns jogos têm uma variedade mais ampla de resultados possíveis; a compensação no gamão varia de 0 até +192. Um **jogo de soma zero** é (confusamente) definido como aquele em que a compensação total para todos os jogadores é a mesma para cada instância do jogo. O xadrez é de soma zero porque cada jogo tem compensação  $0 + 1, 1 + 0$  ou  $\frac{1}{2} + \frac{1}{2}$ . “Soma constante” teria sido um termo melhor, mas soma zero é tradicional e faz sentido se você imaginar que de cada jogador é cobrada uma taxa de entrada de  $\frac{1}{2}$ .

O estado inicial função **AÇÕES** e a função **RESULTADO** definem a **árvore de jogo** correspondente ao jogo — uma árvore onde os nós são estados do jogo e as bordas são movimentos. A Figura 5.1 mostra parte da árvore de jogo para o jogo da velha. A partir do estado inicial, MAX tem nove movimentos possíveis. O jogo se alterna entre a colocação de um X por MAX e a colocação de um O por MIN até alcançarmos nós de folhas correspondentes a estados terminais, tais que um jogador tem três símbolos em uma linha ou todos os quadrados são preenchidos. O número em cada nó de folha indica o valor de utilidade do estado terminal, do ponto de vista de MAX; valores altos são considerados bons para MAX e ruins para MIN (o que explica os nomes dados aos jogadores).



**Figura 5.1** Uma árvore de busca (parcial) para o jogo da velha. O nó superior é o estado inicial, e MAX faz o primeiro movimento colocando um X em um quadrado vazio. Mostramos parte da árvore de busca fornecendo movimentos alternados por MIN (O) e MAX(x), até alcançarmos finalmente os estados terminais, aos quais podem ser atribuídas utilidades de acordo com as regras do jogo.

Para o jogo da velha, a árvore de jogo é relativamente pequena, menos de  $9! = 362.880$  nós terminais. Mas, para o xadrez, há mais de  $10^{40}$  nós, de modo que é melhor pensar na árvore de jogo como sendo uma construção teórica que não podemos perceber no mundo físico. Mas, independentemente do tamanho da árvore de jogo, é trabalho de MAX a busca de uma boa jogada. Usamos o termo **árvore de busca** para uma árvore que está sobreposta à árvore de jogo completa, examinando os nós o suficiente para permitir que um jogador determine que lance fazer.

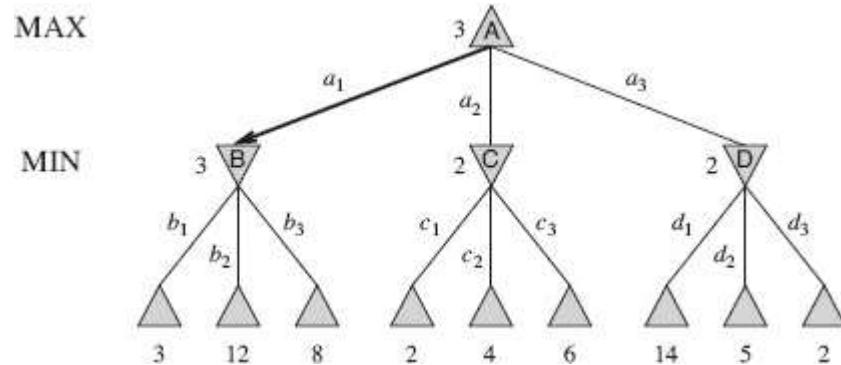
## 5.2 DECISÕES ÓTIMAS EM JOGOS

Em um problema de busca normal, a solução ótima seria uma sequência de ações que levasse a um estado objetivo — um estado terminal que representa uma vitória. Por outro lado, em um jogo, MIN tem alguma relação com esse estado. Portanto, MAX deve encontrar uma **estratégia** de contingência

que especifique o movimento de MAX no estado inicial e depois os movimentos de MAX nos estados resultantes de cada resposta possível de MIN, e depois os movimentos de MAX nos estados resultantes de cada resposta possível de MIN a *esses* movimentos, e assim por diante.

Isso é exatamente análogo ao algoritmo E-OU de busca (Figura 4.11), com MAX no papel de OU e MIN equivalente a E. Grosseiramente falando, uma ótima estratégia leva a resultados pelo menos tão bons como qualquer outra estratégia quando se está jogando com um adversário infalível. Começaremos mostrando como encontrar essa estratégia ótima.

Até mesmo um jogo simples como o jogo da velha é muito complexo para traçarmos a árvore de jogo inteira em uma página e, assim, nos limitaremos ao jogo trivial da Figura 5.2. Os movimentos possíveis para MAX no nó raiz são identificados por  $a_1$ ,  $a_2$  e  $a_3$ . As respostas possíveis para  $a_1$  correspondentes a MIN são  $b_1$ ,  $b_2$  e  $b_3$ , e assim sucessivamente. Esse jogo específico termina depois de um movimento realizado por MAX e por MIN. (No linguajar dos jogos, dizemos que essa árvore tem a profundidade de um único movimento, que consiste em dois meios movimentos, cada um dos quais é chamado **jogada**.) As utilidades dos estados terminais nesse jogo variam de 2 a 14.



**Figura 5.2** Uma árvore de jogo de duas jogadas. Os nós  $\Delta$  são “nós de MAX”, nos quais é a vez de MAX efetuar um movimento, e os nós  $\nabla$  são “nós de MIN”. Os nós terminais mostram os valores de utilidade para MAX; os outros nós estão identificados com seus valores minimax. O melhor movimento de MAX na raiz é  $a_1$  porque leva a um estado com o mais alto valor minimax, e a melhor resposta de MIN é  $b_1$  porque leva a um estado com o mais baixo valor minimax.

Dada uma árvore de jogo, a estratégia ótima pode ser determinada do **valor minimax** de cada nó, que representamos como **VALOR-MINIMAX( $n$ )**. O valor minimax de um nó é a utilidade (para MAX) de se encontrar no estado correspondente, *supondo-se que ambos os jogadores tenham desempenho ótimo* desde esse estado até o fim do jogo. É óbvio que o valor minimax de um estado terminal é simplesmente sua utilidade. Além disso, dada uma escolha, MAX preferirá se mover para um estado de valor máximo, enquanto MIN preferirá um estado de valor mínimo. Assim, temos:

$$\text{VALOR-MINIMAX}(s) = \begin{cases} \text{UTILIDADE}(s) & \text{se TESTE DE TÉRMINO}(s) \\ \max_{a \in A_{\text{ações}}(s)} \text{MINIMAX}(\text{RESULTADO}(s, a)) & \text{se JOGADOR}(s) = \text{MAX} \\ \min_{a \in A_{\text{ações}}(s)} \text{MINIMAX}(\text{RESULTADO}(s, a)) & \text{se JOGADOR}(s) = \text{MIN} \end{cases}$$

Vamos aplicar essas definições à árvore de jogo da Figura 5.2. Os nós terminais no nível inferior obtiverem os valores utilidade da função UTILIDADE do jogo. O primeiro nó de MIN, identificado

por  $B$ , tem três sucessores com valores 3, 12 e 8; portanto, seu valor minimax é 3. De modo semelhante, os outros dois nós de MIN têm valor minimax 2. O nó raiz é um nó de MAX; seus estados sucessores têm valores minimax 3, 2 e 2; logo, ele tem um valor minimax igual a 3. Também podemos identificar a **decisão minimax** na raiz: a ação  $a_1$  é a escolha ótima para MAX porque leva ao estado com o mais alto valor minimax.

Essa definição de jogo ótimo para MAX supõe que MIN também jogue de forma ótima — ela maximiza o resultado para MAX no *pior caso*. E se MIN não jogar de forma ótima? Nesse caso, é fácil mostrar (Exercício 5.7) que MAX terá um desempenho ainda melhor. Pode haver outras estratégias contra oponentes não ótimos que poderão funcionar melhor que a estratégia de minimax; porém, essas estratégias necessariamente têm um desempenho pior contra oponentes ótimos.

## 5.2.1 O algoritmo minimax

O **algoritmo minimax** (Figura 5.3) calcula a decisão minimax a partir do estado corrente. Ela utiliza uma computação recursiva simples dos valores minimax de cada estado sucessor, implementando diretamente as equações da definição. A recursão percorre todo o caminho descendente até as folhas da árvore e, depois, os valores minimax são **propagados de volta** pela árvore, à medida que a recursão retorna. Por exemplo, na Figura 5.2, primeiro o algoritmo efetua uma recursão descendo a árvore até os três nós de folhas inferiores e emprega a função UTILIDADE sobre eles para descobrir que seus valores são 3, 12 e 8, respectivamente. Em seguida, ele toma o mínimo desses valores, 3, e o devolve como valor propagado de volta para o nó  $B$ . Um processo semelhante fornece os valores propagados de volta de 2 para  $C$  e 2 para  $D$ . Por fim, tomamos o valor máximo entre 3, 2 e 2 para obter o valor propagado de volta igual a 3 para o nó raiz.

---

**função DECISÃO-MINIMAX( $estado$ ) retorna uma ação**  
**retornar**  $\arg \max_{a \in \text{Ações}(s)} \text{VALOR-MIN}(\text{RESULTADO}(estado, a))$

---

**função VALOR-MAX( $estado$ ) retorna um valor de utilidade**  
**se** TESTE TERMINAL( $estado$ ) **então retornar** UTILIDADE( $estado$ )  
 $v \leftarrow -\infty$   
**para cada**  $a$  em AÇÕES( $estado$ ) **faça**  
 $v \leftarrow \text{MAX}(v, \text{VALOR-MIN}(\text{RESULTADO}(s, a)))$   
**retornar**  $v$

---

**função VALOR-MIN( $estado$ ) retorna um valor de utilidade**  
**se** TESTE-TERMINAL( $estado$ ) **então retornar** UTILIDADE( $estado$ )  
 $v \leftarrow \infty$   
**para cada**  $a$  em AÇÕES( $estado$ ) **faça**  
 $v \leftarrow \text{MIN}(v, \text{VALOR-MAX}(\text{RESULTADO}(s, a)))$   
**retornar**  $v$

**Figura 5.3** Um algoritmo para calcular decisões minimax. Ele retorna a ação correspondente ao melhor movimento possível, isto é, o movimento que leva ao resultado com a melhor utilidade, sob a suposição de que o oponente joga para minimizar a utilidade. As funções VALOR-MAX e VALOR-MIN passam por toda a árvore de jogo, até chegar às folhas, a fim de determinar o valor de propagação de volta de um estado. A notação  $\operatorname{argmax}_{a \in S} f(a)$  calcula o elemento  $a$  do conjunto  $S$  que possui o valor máximo de  $f(a)$ .

O algoritmo minimax executa uma exploração completa em profundidade da árvore de jogo. Se a profundidade máxima da árvore é  $m$  e existem  $b$  movimentos válidos em cada ponto, a complexidade de tempo do algoritmo minimax é  $O(b^m)$ . A complexidade de espaço é  $O(b^m)$  para um algoritmo que gera todos os sucessores de uma vez ou  $O(m)$  para um algoritmo que gera ações, uma de cada vez. É claro que, em jogos reais, o custo de tempo é totalmente impraticável, mas esse algoritmo serve como base para a análise matemática de jogos e para algoritmos mais práticos.

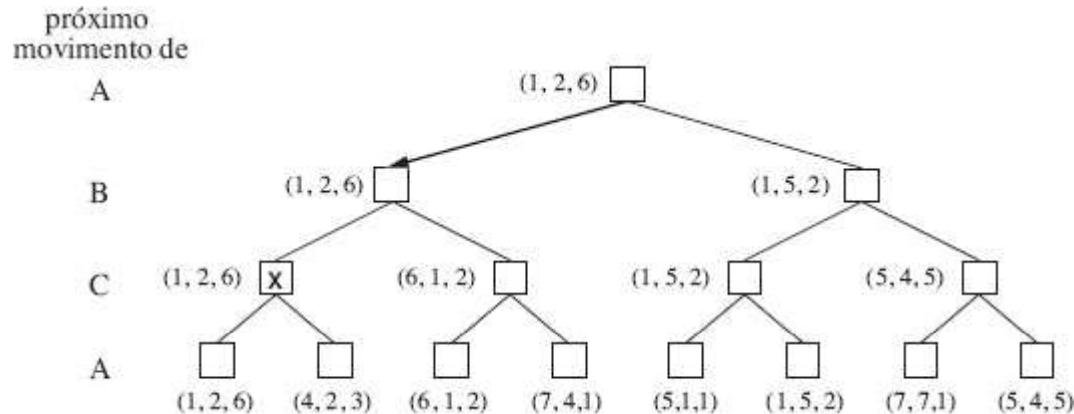
## 5.2.2 Decisões ótimas em jogos com vários participantes

Muitos jogos populares permitem mais de dois jogadores. Vamos examinar a maneira de estender a ideia de minimax a jogos com vários jogadores. Isso é simples do ponto de vista técnico, mas destaca algumas questões conceituais novas e interessantes.

Primeiro, precisamos substituir o único valor para cada nó por um *vetor* de valores. Por exemplo, em um jogo de três jogadores com os participantes  $A$ ,  $B$  e  $C$ , um vetor  $\langle v_A, v_B, v_C \rangle$  está associado a cada nó. Para os estados terminais, esse vetor fornece a utilidade do estado do ponto de vista de cada jogador. (Em jogos de soma zero com dois jogadores, o vetor de dois elementos pode ser reduzido a um único valor porque os valores são sempre opostos.) O caminho mais simples para implementar isso é fazer a função UTILIDADE retornar um vetor de utilidades.

Agora temos de considerar os estados não terminais. Vamos examinar o nó identificado por  $X$  na árvore de jogo da Figura 5.4. Nesse estado, o jogador  $C$  define o que fazer. As duas escolhas levam a estados terminais com vetores de utilidade  $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$  e  $\langle v_A = 4, v_B = 2, v_C = 3 \rangle$ . Tendo em vista que 6 é maior que 3,  $C$  deve-se escolher o primeiro movimento. Isso significa que, se o estado  $X$  for alcançado, a jogada subsequente levará a um estado terminal com utilidades  $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$ . Consequentemente, o valor de  $X$  que foi propagado de volta é esse vetor. Em geral, o valor propagado de volta de um nó  $n$  é sempre o vetor de utilidade do estado do sucessor com o mais alto valor para a escolha do jogador em  $n$ . Qualquer pessoa que participa de jogos com vários jogadores, como Diplomacy™, logo fica ciente de que muito mais acontece do que em jogos de dois jogadores. Os jogos com vários participantes normalmente envolvem **alianças**, sejam elas formais ou informais, entre os jogadores. As alianças são feitas e desfeitas à medida que o jogo se desenrola. Como entender tal comportamento? As alianças constituem uma consequência natural de estratégias ótimas para cada jogador em um jogo com vários participantes? É possível que sim. Por exemplo, vamos supor que  $A$  e  $B$  estejam em posições fracas e que  $C$  esteja em uma posição mais forte. Então, com frequência, é ótimo para  $A$  e  $B$  atacarem  $C$  em vez de atacarem um ao outro, para que  $C$  não destrua cada um deles individualmente. Desse modo, a colaboração emerge de um comportamento

puramente egoísta. É claro que, tão logo  $C$  se enfraqueça sob o violento ataque conjunto, a aliança perderá seu valor, e  $A$  ou  $B$  poderá violar o acordo. Em alguns casos, as alianças explícitas apenas tornam concreto aquilo que teria acontecido de qualquer modo. Em outros casos, um estigma social incorpora-se para romper uma aliança, de forma que os jogadores devem buscar o equilíbrio entre a vantagem imediata de romper uma aliança e a desvantagem a longo prazo de serem considerados pouco confiáveis. Veja a Seção 17.5 para obter mais informações sobre essas complicações.



**Figura 5.4** As três primeiras jogadas de uma árvore de jogo com três jogadores ( $A, B, C$ ). Cada nó é identificado com valores do ponto de vista de cada jogador. O melhor movimento está assinalado na raiz.

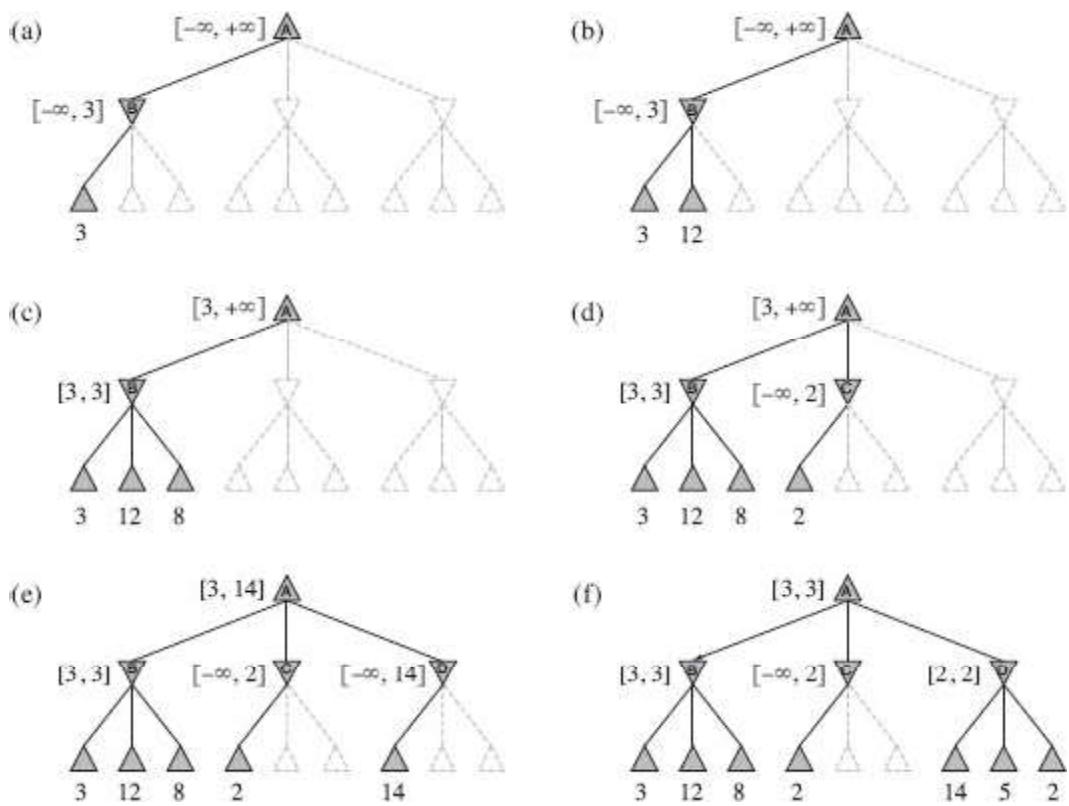
Se o jogo for de soma diferente de zero, a colaboração também poderá ocorrer com apenas dois jogadores. Por exemplo, vamos supor que exista um estado terminal com utilidades  $\langle v_A = 1.000, v_B = 1.000 \rangle$  e que 1.000 seja a mais alta utilidade possível para cada jogador. Então, a estratégia ótima é a de ambos os jogadores fazerem todo o possível para alcançar esse estado, isto é, os jogadores cooperarão de forma automática para atingir uma meta mutuamente desejável.

## 5.3 PODA ALFA-BETA

O problema da busca minimax é que o número de estados de jogo que ela tem de examinar é exponencial em relação ao número de movimentos. Infelizmente, não podemos eliminar o expoente, mas resulta que podemos efetivamente reduzi-lo pela metade. O artifício é a possibilidade de calcular a decisão minimax correta sem examinar todos os nós na árvore de jogo. Ou seja, podemos tomar emprestada a ideia de **poda** do Capítulo 3, a fim de poder deixar de considerar grandes partes da árvore. A técnica específica que examinaremos é chamada **poda alfa-beta**. Quando é aplicada a uma árvore minimax padrão, ela retorna o mesmo movimento que minimax retornaria, mas poda as ramificações que não terão influência possível sobre a decisão final.

Considere novamente a árvore de jogo de duas jogadas da Figura 5.2. Vamos acompanhar mais uma vez o cálculo da decisão ótima, agora prestando bastante atenção ao que conhecemos em cada ponto do processo.

Os passos são explicados na Figura 5.5. O resultado é que podemos identificar a decisão minimax sem jamais avaliar dois dentre os nós de folhas.



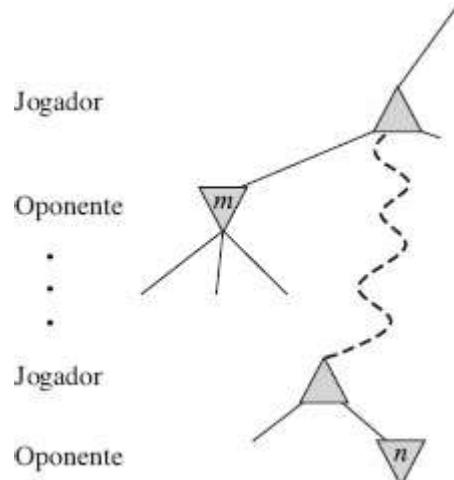
**Figura 5.5** Fases no cálculo da decisão ótima para a árvore de jogo da Figura 5.2. Em cada ponto, mostramos o intervalo de valores possíveis para cada nó. (a) A primeira folha sob  $B$  tem valor 3. Consequentemente,  $B$ , que é um nó de MIN, tem valor *máximo* 3. (b) A segunda folha sob  $B$  tem valor 12; MIN evitaria esse movimento, de forma que o valor de  $B$  ainda é, no máximo, 3. (c) A terceira folha sob  $B$  tem valor 8; vimos todos os estados sucessores de  $B$  e, assim, o valor de  $B$  é exatamente 3. Agora, podemos deduzir que o valor da raiz é *pelo menos* 3, porque MAX tem uma escolha de valor 3 na raiz. (d) A primeira folha abaixo de  $C$  tem o valor 2. Consequentemente,  $C$ , que é um nó de MIN, tem valor *máximo* 2. Porém, sabemos que  $B$  vale 3; portanto, MAX nunca escolheria  $C$ . Desse modo, não há razão para se examinar os outros sucessores de  $C$ . Esse é um exemplo de poda alfabética. (e) A primeira folha abaixo de  $D$  tem o valor 14, e então  $D$  vale *no máximo* 14. Esse valor ainda é mais alto que a melhor alternativa de MAX (isto é, 3) e, portanto, precisamos continuar a explorar sucessores de  $D$ . Note também que agora temos limites para todos os sucessores da raiz e, consequentemente, o valor da raiz também é no máximo 14. (f) O segundo sucessor de  $D$  vale 5 e, assim, novamente precisamos continuar a exploração. O terceiro sucessor vale 2; agora,  $D$  vale exatamente 2. A decisão de MAX na raiz é efetuar o movimento para  $B$ , o que nos dá o valor 3.

Isso também pode ser visto como uma simplificação da fórmula de VALOR-MINIMAX. Sejam  $x$  e  $y$  valores dos dois sucessores não avaliados do nó  $C$  na Figura 5.5 e seja  $z$  o mínimo entre  $x$  e  $y$ . Então, o valor do nó raiz é dado por:

$$\begin{aligned}
 \text{MINIMAX}(raiz) &= \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2)) \\
 &= \max(3, \min(2, x, y), 2) \\
 &= \max(3, z, 2) \quad \text{onde } z = \min(2, x, y) \leq 2 \\
 &= 3.
 \end{aligned}$$

Em outras palavras, o valor da raiz e, consequentemente, a decisão minimax são *independentes* dos valores das folhas podadas  $x$  e  $y$ .

 A poda alfa-beta pode ser aplicada a árvores de qualquer profundidade e frequentemente é possível podar subárvores inteiras em lugar de podar apenas folhas. O princípio geral é este: considere um nó  $n$  em algum lugar na árvore (veja a Figura 5.6), tal que o Jogador tenha a escolha de movimento até esse nó. Se o Jogador tiver uma escolha melhor  $m$  no nó pai de  $n$  ou em qualquer ponto de escolha adicional acima dele, então  $n$  nunca será alcançado em um jogo real. Assim, uma vez que descobrimos o suficiente sobre  $n$  (examinando alguns de seus descendentes) para chegar a essa conclusão, poderemos podá-lo.



**Figura 5.6** O caso geral de poda alfa-beta. Se  $m$  é melhor que  $n$  para o Jogador, nunca chegaremos a  $n$  em um jogo.

Lembre-se de que a busca minimax é do tipo em profundidade; então, em qualquer instante só temos de considerar os nós ao longo de um único caminho na árvore. A poda alfa-beta obtém seu nome a partir dos dois parâmetros a seguir, que descrevem limites sobre os valores propagados de volta que aparecem em qualquer lugar ao longo do caminho:

$\alpha$  = o valor da melhor escolha (isto é, a de valor mais alto) que encontramos até o momento em qualquer ponto de escolha ao longo do caminho para MAX.

$\beta$  = o valor da melhor escolha (isto é, a de valor mais baixo) que encontramos até agora em qualquer ponto de escolha ao longo do caminho para MIN.

A busca alfa-beta atualiza os valores de  $\alpha$  e  $\beta$  à medida que prossegue e poda as ramificações restantes em um nó (isto é, encerra a chamada recursiva) tão logo se sabe que o valor do nó corrente é pior que o valor corrente de  $\alpha$  ou  $\beta$  para MAX ou MIN, respectivamente. O algoritmo completo é mostrado na Figura 5.7. Encorajamos o leitor a acompanhar seu comportamento quando ele é aplicado à árvore da Figura 5.5.

**função** BUSCA-ALFA-BETA(*estado*) **retorna** uma ação

$v \leftarrow \text{VALOR-MAX}(\text{estado}, -\infty, +\infty)$

**retornar** a ação em AÇÕES(*estado*) com valor  $v$

**função** VALOR-MAX(*estado*,  $\alpha$ ,  $\beta$ ) **retorna** um valor de utilidade

**se** TESTE-TERMINAL(*estado*) **então retornar** UTILIDADE(*estado*)

$v \leftarrow -\infty$

**para cada**  $a$ , em AÇÕES(*estado*) **faça**

$v \leftarrow \text{MAX}(v, \text{VALOR-MIN}(\text{RESULTADO}(s, a), \alpha, \beta))$

**se**  $v \geq \beta$  **então retornar**  $v$

$\alpha \leftarrow \text{MAX}(\alpha, v)$

**retornar**  $v$

---

**função** VALOR-MIN(*estado*,  $\alpha$ ,  $\beta$ ) **retorna** um valor de utilidade

**se** TESTE-TERMINAL(*estado*) **então retornar** UTILIDADE(*estado*)

$v \leftarrow +\infty$

**para cada**  $a$ , em AÇÕES(*estado*) **faça**

$v \leftarrow \text{MIN}(v, \text{VALOR-MIN}(\text{RESULTADO}(s, a), \alpha, \beta))$

**se**  $v \leq \alpha$  **então retornar**  $v$

$\beta \leftarrow \text{MIN}(\beta, v)$

**retornar**  $v$

**Figura 5.7** O algoritmo de busca alfa-beta. Note que essas rotinas são idênticas às rotinas de MINIMAX da Figura 5.3, com exceção das duas linhas em cada uma das funções VALOR-MIN e VALOR-MAX que mantêm  $\alpha$  e  $\beta$  (e da necessidade de repassar esses parâmetros).

### 5.3.1 Ordenação de movimentos

A efetividade da poda alfa-beta é altamente dependente da ordem em que os estados são examinados. Por exemplo, na Figura 5.5(e) e (f), não poderíamos podar quaisquer sucessores de  $D$  porque os piores sucessores (do ponto de vista de MIN) foram gerados primeiro. Se o terceiro sucessor tivesse sido gerado primeiro, seríamos capazes de podar os outros dois. Isso sugere que poderia valer a pena tentar examinar primeiro os sucessores que têm probabilidade de serem melhores.

Se supusermos que isso pode ser feito,<sup>2</sup> então o resultado será que alfa-beta precisará examinar apenas  $O(b^{m/2})$  nós para escolher o melhor movimento, em vez de  $O(b^m)$  para minimax. Isso significa que o fator de ramificação efetivo se tornará  $\sqrt{b}$  em vez de  $b$  — no caso do xadrez, 6 em vez de 35. Em outras palavras, alfa-beta poderá resolver uma árvore aproximadamente duas vezes tão profunda como minimax no mesmo período de tempo. Se os sucessores forem examinados em ordem aleatória, em vez de se tomar o melhor em primeiro lugar, o número total de nós examinados será cerca de  $O(b^{3m/4})$  para um valor moderado de  $b$ . No caso do xadrez, uma função de ordenação bastante simples (como experimentar capturas primeiro, depois ameaças, depois movimentos para a frente e, em seguida, movimentos para trás) levará você a uma distância de aproximadamente duas vezes o resultado do melhor caso,  $O(b^{m/2})$ .

Acrescentar esquemas dinâmicos de ordenação de movimentos, como tentar primeiro os movimentos considerados os melhores da última vez, nos levará até bem perto do limite teórico. O passado pode ser o lance anterior — muitas vezes, as mesmas ameaças permanecem — ou poderia

vir da exploração do lance atual. Uma maneira de obter informações do lance atual é com busca de aprofundamento iterativo da pesquisa. Primeiro, pesquise uma jogada profunda e registre o melhor caminho de movimentos. Em seguida, busque uma jogada mais profunda, mas utilize o caminho registrado para informar a ordenação do movimento. Como vimos no Capítulo 3, o aprofundamento iterativo em uma árvore de jogo exponencial acrescenta apenas uma fração constante para o tempo total de busca, que pode ser mais do que compensado por uma melhor ordenação de movimento. As melhores jogadas são muitas vezes chamadas de **lances mortais** e tentá-los de primeira é chamado de heurística de lance mortal.

No Capítulo 3, observamos que estados repetidos na árvore de busca podem causar um aumento exponencial no custo da busca. Em muitos jogos, estados repetidos ocorrem com frequência devido a **transposições** — permutações diferentes da mesma sequência que terminam na mesma posição. Por exemplo, se as brancas têm um movimento  $a_1$  que pode ser respondido pelas pretas com  $b_1$  e um movimento não relacionado  $a_2$  no outro lado do tabuleiro que pode ser respondidos por  $b_2$ , as sequências  $[a_1, b_1, a_2, b_2]$  e  $[a_1, b_2, a_2, b_1]$  terminarão na mesma posição. Vale a pena armazenar a avaliação dessa posição resultante em uma tabela de hash na primeira vez em que ela for encontrada, de forma que não tenhamos de recalcular-a em ocorrências subsequentes.

A tabela de hash de posições já vistas é tradicionalmente chamada **tabela de transposição**; em essência, ela é idêntica à lista *explorada* em BUSCA-EM-GRAFO (Seção 3.3). O uso de uma tabela de transposição pode ter um efeito drástico, chegando às vezes a duplicar a profundidade de busca acessível no xadrez. Por outro lado, se estivermos avaliando um milhão de nós por segundo, não será prático manter *todos* eles na tabela de transposição. São usadas diversas estratégias para escolher os nós que devem ser mantidos e os que devem ser descartados.

## 5.4 DECISÕES IMPERFEITAS EM TEMPO REAL

---

O algoritmo minimax gera o espaço de busca do jogo inteiro, enquanto o algoritmo alfa-beta nos permite podar grandes partes desse espaço. Porém, alfa-beta ainda tem de fazer a busca em toda a distância até os estados terminais, pelo menos para uma parte do espaço de busca. Em geral, essa profundidade não é prática porque os movimentos devem ser realizados em um período de tempo razoável — normalmente por alguns minutos, no máximo. O artigo de Claude Shannon, *Programming a computer for playing chess* (1950), propunha em vez disso que os programas cortassem a busca mais cedo e aplicassem uma **função de avaliação** heurística aos estados da busca, transformando efetivamente nós não terminais em folhas terminais. Em outras palavras, a sugestão é alterar minimax ou alfa-beta de duas maneiras: substituir a função utilidade por uma função de avaliação de heurística AVAL, que fornece uma estimativa da utilidade da posição, e o teste de término por um **teste de corte** que decide quando aplicar AVAL. Isso nos dá o seguinte para minimax heurística para o estado  $s$  e profundidade máxima  $d$ :

$$\text{H-MINIMAX}(s, d) = \begin{cases} \text{AVAL}(s) & \text{se TESTE DE CORTE}(s, d) \\ \max_{a \in \text{Ações}(s)} \text{H-MINIMAX}(\text{RESULTADO}(s, a), d + 1) & \text{se JOGADOR}(s) = \text{MAX} \\ \min_{a \in \text{Ações}(s)} \text{H-MINIMAX}(\text{RESULTADO}(s, a), d + 1) & \text{se JOGADOR}(s) = \text{MIN.} \end{cases}$$

## 5.4.1 Funções de avaliação

Uma função de avaliação retorna uma *estimativa* da utilidade esperada do jogo, a partir de uma dada posição, da mesma forma que as funções de heurísticas do Capítulo 3 retornam uma estimativa da distância até a meta.

A ideia de um avaliador não era nova quando Shannon a propôs. Durante séculos, os jogadores de xadrez (e os aficionados por outros jogos) desenvolveram meios de julgar o valor de uma posição porque os seres humanos são ainda mais limitados que os programas de computador no volume de busca que podem realizar. Deve ficar claro que o desempenho de um programa de jogos depende fortemente da qualidade de sua função de avaliação. Uma função de avaliação inexata guiará um agente em direção a posições que acabarão por ser perdidas. Qual é a maneira exata de projetarmos boas funções de avaliação?

Primeiro, a função de avaliação deve ordenar os estados *terminais* do mesmo modo que a verdadeira função utilidade: estados que são vitórias que devem avaliar empates, que por sua vez devem ser melhores que perdas. Caso contrário, um agente que utilizasse a função de avaliação poderia errar, mesmo que pudesse antecipar todos os movimentos até o fim do jogo. Em segundo lugar, a computação não deve demorar tempo demais! (O ponto fundamental é a busca mais rápida.) Em terceiro lugar, no caso de estados não terminais, a função de avaliação deve estar fortemente relacionada com as chances reais de vitória.

O leitor deve ter se surpreendido com a expressão “chances de vitória”. Afinal, o xadrez não é um jogo de azar: conhecemos o estado corrente com certeza e não há dados envolvidos no processo. Contudo, se a busca tiver de ser cortada em estados não terminais, o algoritmo será necessariamente *incerto* sobre os resultados finais desses estados. Esse tipo de incerteza é induzido por limitações computacionais, não informativas.

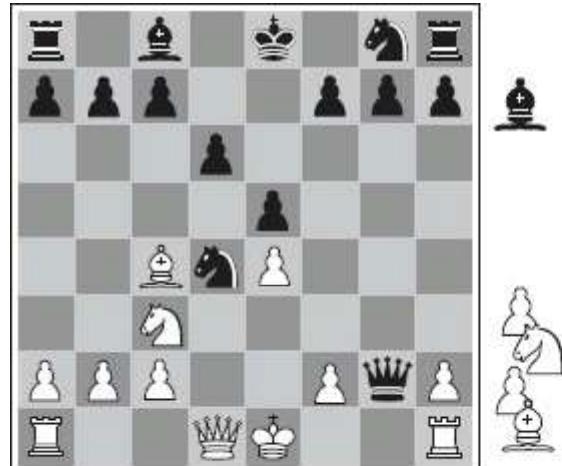
Dado o volume limitado de computação que a função de avaliação pode realizar para determinado estado, o melhor que ela pode fazer é arriscar um palpite sobre o resultado final.

Vamos tornar essa ideia mais concreta. A maioria das funções de avaliação atua calculando diversas **características** do estado — por exemplo, no xadrez, teríamos características para o número de peões brancos, pretos, rainhas brancas, pretas, e assim por diante. Consideradas em conjunto, as características definem diversas *categorias* ou *classes de equivalência* de estados: os estados de cada categoria têm os mesmos valores para todas as características. Por exemplo, ao final do jogo, uma categoria contém ao todo dois peões *versus* um peão. Qualquer categoria específica, em termos gerais, conterá alguns estados que levam a vitórias, alguns que levam a empates e alguns que levam a derrotas. A função de avaliação não tem como saber quais são os estados de cada grupo, mas pode retornar um único valor capaz de refletir a *proporção* de estados que conduzem a cada resultado. Por exemplo, vamos supor que nossa experiência sugira que 72% dos estados encontrados na categoria dois peões *versus* um peão levem a uma vitória (com utilidade +1); 20% levem a uma derrota (0) e 8% a um empate (1/2). Então, uma avaliação razoável dos estados na categoria é a média ponderada ou o **valor esperado**:  $(0,72 \times +1) + (0,20 \times 0) + (0,08 \times 1/2) = 0,76$ . Em princípio, o valor esperado pode ser determinado para cada categoria, o que resulta em uma função de avaliação que funciona para qualquer estado. Como ocorre com estados terminais, a função de avaliação não precisa retornar valores esperados reais, desde que a *ordenação* dos estados seja a

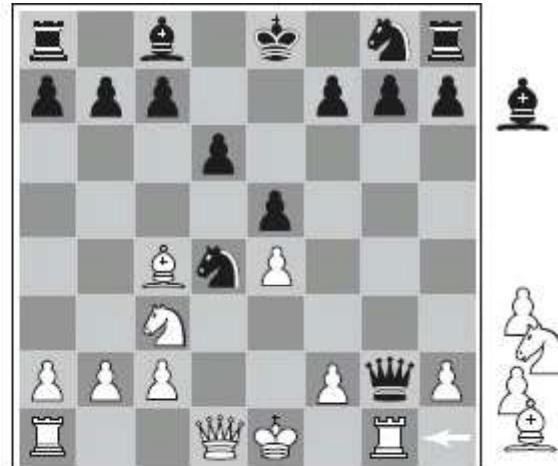
mesma.

Na prática, essa espécie de análise exige muitas categorias e, consequentemente, muita experiência para estimar todas as probabilidades de vitória. Em vez disso, a maioria das funções de avaliação calcula contribuições numéricas separadas de cada característica e depois as *combina* para encontrar o valor total. Por exemplo, os livros introdutórios de xadrez fornecem um **valor material** aproximado para cada peça: cada peão vale 1, um cavalo ou um bispo pena 3, uma torre 5 e a rainha 9. Outras características, como “boa estrutura de peões” e “segurança do rei” poderiam valer, digamos, metade de um peão. Esses valores de características são então simplesmente somados para se obter a avaliação da posição.

Uma vantagem segura equivalente a um peão fornece uma probabilidade substancial de vitória, e uma vantagem segura equivalente a três peões deve proporcionar uma vitória quase certa, como ilustra a Figura 5.8(a). Matematicamente, essa espécie de função de avaliação é chamada **função linear ponderada** porque pode ser expressa como:



(a) As brancas jogam



(b) As brancas jogam

**Figura 5.8** Duas posições de xadrez, que diferem apenas na posição da torre na parte inferior direita. Em (a), as pretas têm uma vantagem de um cavalo e dois peões, que deveria ser o suficiente para vencer o jogo. Em (b), a branca vai capturar a rainha, dando-lhe uma vantagem que deveria ser forte o suficiente para vencer.

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^n w_i f_i(s) ,$$

onde cada  $w_i$  é um peso e cada  $f_i$  é uma característica da posição. No caso do xadrez,  $f_i$  poderia representar os números de cada tipo de peça no tabuleiro,  $w_i$  poderia corresponder aos valores das peças (1 para peão, 3 para bispo etc.).

Somar os valores de características parece algo razoável, mas, na verdade, envolve uma suposição muito forte: que a contribuição de cada característica é *independente* dos valores das outras características. Por exemplo, a atribuição do valor 3 a um bispo ignora o fato de que os bispos são mais poderosos no fim do jogo, quando têm bastante espaço de manobra.

Por essa razão, os programas atuais de xadrez e de outros jogos também utilizam combinações *não lineares* de características. Por exemplo, um par de bispos poderia valer um pouco mais que o dobro

do valor de um único bispo, e um bispo poderia valer mais no fim do jogo que no início (isto é, quando a característica do *número de lances* for alta ou a característica do *número de peças restantes* for baixa).

O leitor atento notará que as características e os pesos *não* fazem parte das regras do xadrez! Eles vêm de séculos de experiência humana no jogo de xadrez. Em jogos nos quais esse tipo de experiência não está disponível, os pesos da função de avaliação podem ser estimados pelas técnicas de aprendizado de máquina do Capítulo 18. Vale a pena reafirmar que a aplicação dessas técnicas ao xadrez confirma que um bispo vale realmente cerca de três peões.

## 5.4.2 Busca com corte

A próxima etapa é modificar BUSCA-ALFA-BETA, de modo que ela chame a função heurística AVAL quando for apropriado cortar a busca. Em termos de implementação, substituímos as duas linhas da Figura 5.7 que mencionam TESTE-TERMINAL pela linha a seguir:

**se TESTE-DE-CORTE(*estado, profundidade*) então retornar AVAL(*estado*).**

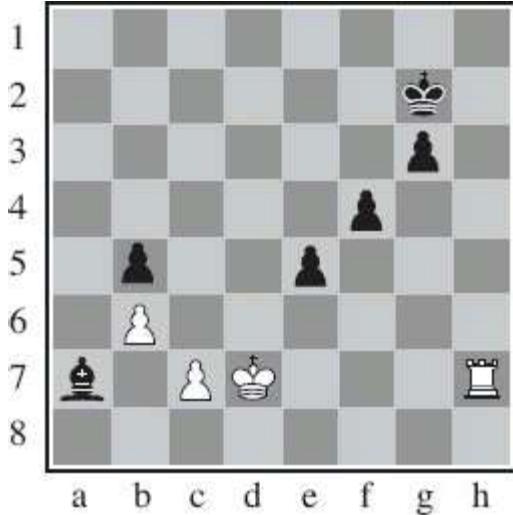
Também devemos providenciar alguma notação para que a *profundidade* corrente seja incrementada em cada chamada recursiva. A abordagem mais direta para controlar a quantidade de busca é definir um limite de profundidade fixo, a fim de que TESTE-DE-CORTE (*estado, profundidade*) retorne *verdadeiro* para toda *profundidade* maior que alguma profundidade fixa *d* (ela também deve retornar *verdadeiro* para todos os estados terminais, como fazia TESTE-TERMINAL). A profundidade *d* é escolhida de modo que um movimento seja selecionado dentro do tempo previsto. Uma abordagem mais resistente é aplicar o aprofundamento iterativo (veja o Capítulo 3). Quando o tempo se esgota, o programa retorna o movimento selecionado pela busca mais profunda concluída. Como bônus, o aprofundamento iterativo também ajuda com a ordenação do movimento.

No entanto, essas simples abordagens podem levar a erros, devido à natureza aproximada da função de avaliação. Considere mais uma vez a função de avaliação simples para xadrez, baseada na vantagem material. Suponha que o programa pesquise até a profundidade limite, alcançando a posição da Figura 5.8(b), onde as pretas têm a vantagem de um cavalo e dois peões. Isso seria reportado como o valor heurístico do estado, declarando-se assim que o estado será uma vitória provável das peças pretas. Porém, o próximo movimento das brancas captura a rainha preta sem qualquer compensação. Portanto, a posição resulta na realidade em uma vitória das brancas, mas isso só pode ser visto observando-se mais uma jogada à frente.

É óbvio que é necessário um teste de corte mais sofisticado. A função de avaliação deve ser aplicada apenas a posições **quiescentes** — isto é, posições em que é improvável haver grandes mudanças de valores no futuro próximo. Por exemplo, no xadrez, as posições em que podem ser feitas capturas favoráveis não são quiescentes para uma função de avaliação que simplesmente efetua a contagem material. Posições não quiescentes podem ser expandidas adiante, até serem alcançadas posições quiescentes. Essa busca extra é chamada de **busca de quiescência**; às vezes, ela se restringe a considerar apenas certos tipos de movimentos, como movimentos de captura, que resolvem com

rapidez as incertezas da posição.

O **efeito de horizonte** é mais difícil de eliminar. Ele surge quando o programa está enfrentando um movimento feito pelo oponente que causa sérios danos e, em última instância, inevitável, mas poderia ser evitado temporariamente, através de táticas de adiamento. Considere o jogo de xadrez na Figura 5.9. É claro que não há caminho para o bispo preto fugir. Por exemplo, a torre branca pode capturá-lo movendo-se para h1, depois a1, depois a2; em seguida, uma captura à profundidade da jogada 6. Mas as pretas têm uma sequência de movimentos que empurra a captura do bispo “além do horizonte”. Suponha que os pretas busquem a profundidade da jogada 8. A maioria dos movimentos das pretas vai levar à captura eventual do bispo e, portanto, serão marcadas como lances “ruins”. Mas as pretas considerarão o xeque no rei branco com o peão em e4. Isso fará com que o rei capture o peão. Agora as pretas vão considerar o xeque novamente, com o peão em f5, levando a outra captura de peão. Isso leva quatro jogadas, e as quatro jogadas restantes não são suficientes para capturar o bispo. As pretas pensam que a linha de jogo poupará o bispo, ao preço de dois peões, quando na verdade tudo o que foi feito é empurrar a inevitável captura do bispo para além do horizonte que as pretas podiam visualizar.



**Figura 5.9** O efeito de horizonte. Com as pretas se movendo, o bispo preto está certamente condenado. Mas as pretas podem evitar esse evento marcando o rei branco com seus peões, obrigando o rei a capturar os peões. Isso empurra a perda inevitável do bispo sobre o horizonte e, portanto, o sacrifício dos peões é visto pelo algoritmo de busca como boas jogadas e não como más.

Uma estratégia para mitigar o efeito horizonte é a **extensão singular**, um movimento que é “claramente melhor” do que todos os outros movimentos em determinada posição. Uma vez descoberto em qualquer lugar da árvore no curso de uma busca, esse movimento singular é lembrado. Quando a busca atinge o limite de profundidade normal, o algoritmo verifica se a extensão singular é um movimento legal; se for, permite que o lance seja considerado. Isso faz com que a árvore fique mais profunda, mas, havendo poucas extensões singulares, não adicionará muitos nós totais à árvore.

### 5.4.3 Poda adiantada

Até agora, mencionamos a busca com corte em certo nível e dissemos que a realização da busca

alfa-beta não tem nenhum efeito comprovado sobre o resultado (pelo menos com respeito aos valores de avaliação heurística). Também é possível efetuar a **poda adiantada**, significando que alguns movimentos em dado nó serão podados de imediato, sem consideração adicional. É claro que a maioria dos seres humanos que jogam xadrez só considera alguns movimentos a partir de cada posição (pelo menos de forma consciente). Uma abordagem à poda adiantada é a **busca em feixe**: em cada jogada, considera-se apenas um “feixe” das  $n$  melhores jogadas (de acordo com a função de avaliação) em vez de considerar todos os movimentos possíveis. Infelizmente, a abordagem é bastante perigosa porque não há nenhuma garantia de que o melhor movimento não será podado.

O algoritmo PROBCUT, ou corte probabilístico (Buro, 1995), é uma versão da poda adiantada de busca alfa-beta que utiliza a estatística adquirida com a experiência prévia para diminuir a chance de a melhor jogada ser podada. A busca alfa-beta poda qualquer nó que esteja *provavelmente* fora da janela  $(\alpha, \beta)$  atual. O PROBCUT também poda nós que *provavelmente* estão fora da janela. Ela calcula essa probabilidade fazendo uma busca superficial para calcular o valor propagado  $v$  de um nó e, em seguida, usa a experiência passada para estimar como é que a pontuação de  $v$  à profundidade  $d$  na árvore ficaria fora de  $(\alpha, \beta)$ . Buro aplicou essa técnica ao programa Othello, LOGISTELLO, e descobriu que uma versão de seu programa com PROBCUT bateu a versão regular em 64% do tempo, mesmo quando se dá o dobro de tempo à versão regular.

A combinação de todas as técnicas descritas aqui resulta em um programa que pode jogar xadrez (ou outros jogos) de modo respeitável. Vamos supor que implementamos uma função de avaliação para xadrez, um teste de corte razoável com uma busca de quiescência, e ainda uma grande tabela de transposição. Vamos supor também que, depois de meses de tediosa escovação de bits, podemos gerar e avaliar cerca de um milhão de nós por segundo no PC mais atual, o que nos permite buscar aproximadamente 200 milhões de nós por movimento sob controles de tempo-padrão (três minutos por movimento). O fator de ramificação para xadrez é cerca de 35 em média, e  $35^5$  é aproximadamente igual a 50 milhões; assim, se usássemos a busca minimax, só poderíamos examinar cerca de cinco jogadas à frente. Embora não seja incompetente, tal programa pode ser enganado com facilidade por um jogador de xadrez humano médio, que ocasionalmente pode planejar seis ou oito jogadas à frente. Com a busca alfa-beta, chegamos a cerca de 10 jogadas, o que resulta em um nível de desempenho de especialista. A Seção 5.8 descreve técnicas adicionais de poda que podem estender a profundidade de busca efetiva a aproximadamente 14 jogadas. Para alcançar o *status* de grande mestre, precisaríamos de uma função de avaliação extensivamente ajustada e de um grande banco de dados de movimentos ótimos de abertura e de fim de jogo

#### 5.4.4 Busca *versus* acesso

De alguma forma parece um exagero que um programa de xadrez inicie um jogo, considerando uma árvore de um bilhão de estados de jogo, apenas para concluir que moverá o seu peão para e4. Por cerca de um século, existem livros disponíveis sobre xadrez que descrevem boas jogadas na abertura e encerramento (Tattersall, 1911). Não é de estranhar, portanto, que muitos programas de jogo utilizem a *tabela de acesso* em vez de busca para abertura e encerramento dos jogos.

Para as aberturas, o computador conta principalmente com a perícia dos seres humanos. O melhor

conselho dos peritos humanos de como jogar cada abertura é copiado de livros e introduzido em tabelas para uso do computador. No entanto, os computadores também podem coletar estatísticas de um banco de dados de partidas previamente jogadas para ver que sequências de abertura conduzem a uma vitória na maioria das vezes. Nos movimentos iniciais há poucas opções e, desse modo, comentários de peritos e jogos passados dos quais extrair. Normalmente, depois de 10 movimentos termina-se em uma posição raramente vista, e o programa deve mudar da tabela de acesso para busca.

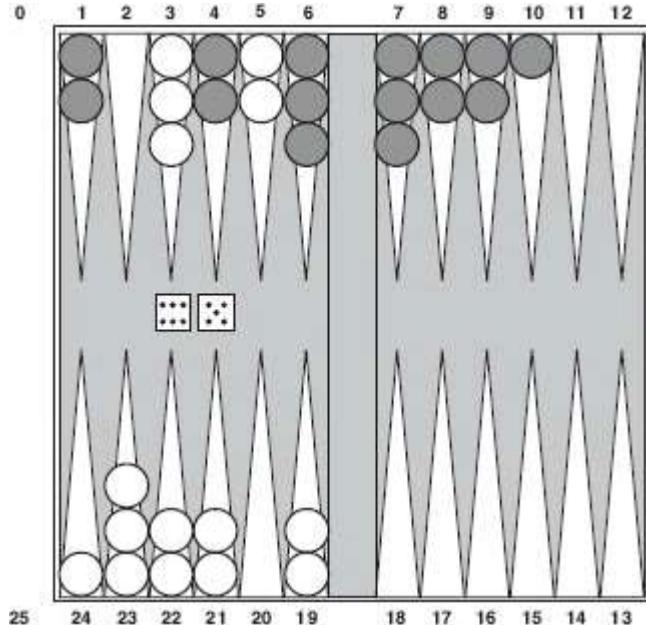
Perto do final do jogo há novamente poucas posições possíveis e, assim, mais chance de fazer acesso. Mas aqui é o computador que tem a experiência: a análise de computador de finais de jogos vai muito além de qualquer coisa alcançada pelos seres humanos. Um ser humano pode dizer-lhe a estratégia geral para jogar um final de rei e torre *versus* rei (RTR): reduzir a mobilidade do rei oposto, encorralando-o em um canto do tabuleiro, utilizando o seu rei para evitar que o adversário escape do aperto. Outros finais, como rei, bispo e cavalo *versus* rei (RBCR), são difíceis de dominar e não têm uma descrição sucinta da estratégia. Um computador, por outro lado, pode *resolver* completamente o fim do jogo, produzindo um **programa de ação**, que é um mapeamento de todos os estados possíveis para a melhor jogada nesse estado. Então podemos apenas procurar pela melhor jogada em vez de recalcular-a novamente. Qual será o tamanho da tabela de acesso RBCR? Acontece que há 462 maneiras como dois reis podem ser colocados no tabuleiro sem estar adjacentes. Depois de os reis serem colocados, haverá 62 quadrados vazios para o bispo, 61 para o cavalo e dois jogadores possíveis para o próximo movimento; portanto, haverá apenas  $462 \times 62 \times 61 \times 2 = 3.494.568$  posições possíveis. Algumas dessas são xeque-mates; marque-as como tal em uma tabela. Em seguida, faça uma busca minimax **retrógrada**: reverte as regras do xadrez para retroceder em vez de mover. Qualquer movimento das brancas que, não importa com qual movimento as pretas respondam, termina em uma posição marcada como vitória, devendo ser também uma vitória. Continue essa busca até que todas as 3.494.568 posições estejam resolvidas como vitória, perda ou empate e você terá uma tabela de acesso infalível para todos os finais de jogos RBCR.

Utilizando essa técnica e com grande esforço de truques de otimização, Ken Thompson (1986, 1996) e Lewis Stiller (1992, 1996) resolveram todos os finais de jogos de xadrez com até cinco peças e alguns com seis peças, tornando-os disponíveis na Internet. Stiller descobriu um caso em que existia um mate forçado, mas exigia 262 movimentos, o que causou alguma consternação porque as regras do xadrez exigem que haja uma captura ou um movimento de um peão dentro de 50 lances. Mais tarde, o trabalho de Marc Bourzutschky e Yakov Konoval (Bourzutschky, 2006) resolveu todos os finais de jogo de seis peças sem peão e alguns de sete peças, havendo um final de jogo RTCRTBC (rei, torre, cavalo, rei, torre, bispo, cavalo) que com o melhor jogo requer 517 movimentos até a captura, que então conduz a um mate.

Se pudéssemos estender o tabuleiro de xadrez de final de jogo de seis para 32 peças, as brancas saberiam no movimento de abertura se seria uma vitória, perda ou empate. Isso não aconteceu até agora para o xadrez, mas tem acontecido para damas, como é explicado na seção de notas históricas.

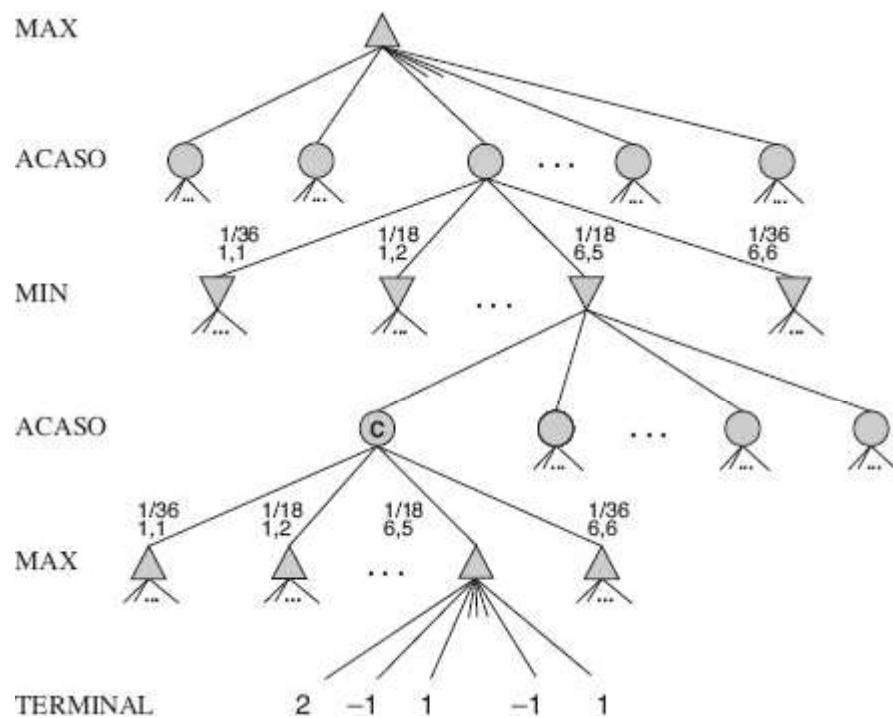
## 5.5 JOGOS ESTOCÁSTICOS

Na vida real, existem muitos eventos externos imprevisíveis que podem nos colocar em situações inesperadas. Muitos jogos refletem essa imprevisibilidade, incluindo um elemento aleatório, como o lançamento de dados. Nós os chamamos de **jogos estocásticos**. O gamão é um jogo típico que combina sorte e habilidade. Os dados são rolados no início da ação de cada jogador para determinar os movimentos válidos. Por exemplo, na posição do jogo de gamão representada na Figura 5.10, as peças brancas tiveram uma rolagem de dados com seis e cinco pontos e têm quatro movimentos possíveis.



**Figura 5.10** Uma posição típica em gamão. O objetivo do jogo é mover todas as peças para fora do tabuleiro. As brancas se movimentam no sentido horário (para a direita) até a posição 25, e as pretas se movimentam no sentido anti-horário (para a esquerda) até 0. Uma peça pode se mover para qualquer posição, a menos que existam várias peças oponentes nessa posição; se houver um oponente, ele será capturado e terá de recomeçar. Na posição mostrada, as brancas obtiveram 6 e 5 nos dados e devem escolher entre quatro movimentos válidos: (5–10, 5–11), (5–11, 19–24), (5–10, 10–16) e (5–11, 11–16), onde a notação (5–11, 11–16) significa mover uma peça da posição 5 para a 11 e depois mover uma peça da 11 para a 16.

Embora o jogador com as brancas saiba quais são seus próprios movimentos válidos, ele não sabe qual será a jogada das pretas e, portanto, não sabe quais serão os movimentos válidos das pretas. Isso significa que o jogador com as brancas não pode construir uma árvore de jogo-padrão do tipo que vimos em xadrez e no jogo da velha. Uma árvore de jogo em gamão deve incluir **nós de acaso** além de nós MAX e MIN. Os nós de acaso são mostrados como circunferências na Figura 5.11. As ramificações que levam a cada nó de acaso denotam as jogadas de dados possíveis, e cada uma é identificada com a jogada e a chance de que ela ocorra. Existem 36 maneiras de rolar dois dados, todas igualmente prováveis; porém, como 6–5 é igual a 5–6, existem apenas 21 lançamentos distintos. Os seis duplos (1–1 a 6–6) têm uma chance de 1/36; dizemos então que  $P(1-1) = 1/36$ . Os outros 15 lançamentos distintos têm a probabilidade de 1/18 cada.



**Figura 5.11** Árvore de jogo esquemática para uma posição de gamão.

A próxima etapa é entender como tomar decisões corretas. É óbvio que desejaremos escolher o movimento que leve à melhor posição. Porém, as posições resultantes não têm valores minimax definidos. Em vez disso, só podemos calcular o **valor esperado** de uma posição: a média sobre todos os resultados possíveis dos nós de acaso.

Isso nos leva a generalizar o **valor minimax** para jogos determinísticos até um **valor de expectiminimax** para jogos com nós de acaso. Nós terminais e nós de MAX e MIN (para os quais o lançamento de dados é conhecido) funcionam exatamente do mesmo modo que antes.

Para os nós de acaso calculamos o valor esperado, que é a soma do valor de todos os resultados, ponderada pela probabilidade de cada ação do acaso:

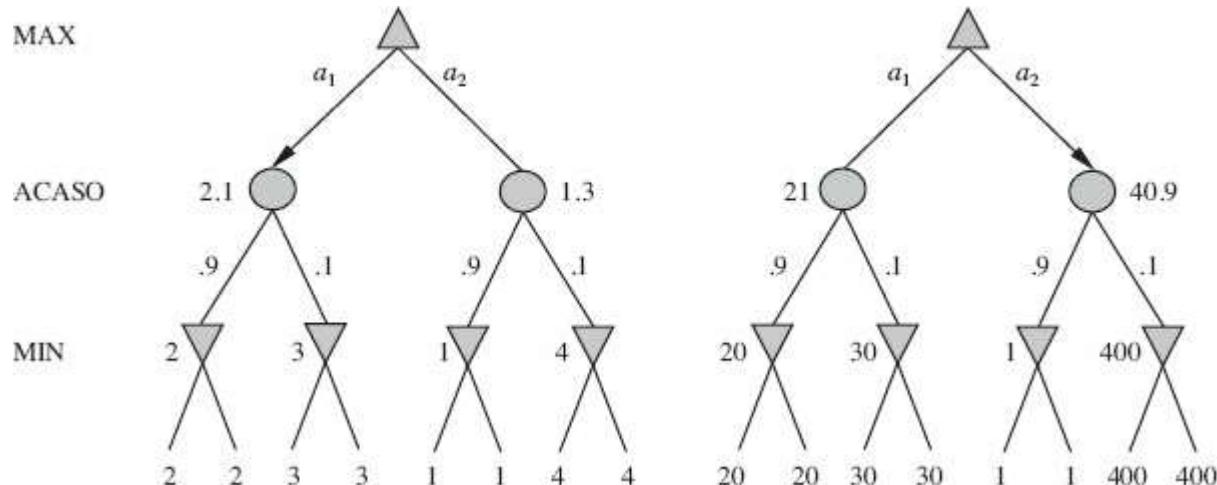
$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILIDADE}(s) & \text{se TESTE DE TÉRMINO}(s) \\ \max_a \text{EXPECTIMINIMAX}(\text{RESULTADO}(s, a)) & \text{se JOGADOR}(s) = \text{MAX} \\ \min_a \text{EXPECTIMINIMAX}(\text{RESULTADO}(s, a)) & \text{se JOGADOR}(s) = \text{MIN} \\ \sum_r P(r) \text{EXPECTIMINIMAX}(\text{RESULTADO}(s, r)) & \text{se JOGADOR}(s) = \text{ACASO} \end{cases}$$

onde  $r$  representa um possível lançamento de dados (ou outro evento ao acaso) e **RESULTADO** ( $s, r$ ) é o mesmo estado que  $s$ , com o fato adicional de que o resultado do lançamento de dados é  $r$ .

### 5.5.1 Funções de avaliação para os jogos de azar

Como ocorre no caso de minimax, a aproximação óbvia a fazer com expectiminimax é cortar a busca em certo ponto e aplicar uma função de avaliação a cada folha. Poderíamos pensar que as funções de avaliação de jogos como gamão devem ser exatamente como as funções de avaliação para xadrez — elas só precisam fornecer pontuações mais altas para posições melhores. Porém, de fato, a presença de nós de acaso significa que temos de ser mais cuidadosos sobre o significado dos valores

de avaliação. A Figura 5.12 mostra o que acontece: com uma função de avaliação que atribui valores  $[1, 2, 3, 4]$  às folhas, o movimento  $a_1$  é melhor; com valores  $[1, 20, 30, 400]$ , o movimento  $a_2$  é melhor. Consequentemente, o programa se comportará de forma bastante diferente se fizermos uma mudança na escala de alguns valores de avaliação! Ocorre que, para evitar essa sensibilidade, a função de avaliação deve ser uma transformação linear positiva da probabilidade de vencer a partir de uma posição (ou, de modo mais geral, da utilidade esperada da posição). Essa é uma propriedade importante e geral de situações em que a incerteza está envolvida, como veremos com mais detalhes no Capítulo 16.



**Figura 5.12** Uma transformação com preservação da ordem em valores de folhas altera o melhor movimento.

Se o programa conhecesse com antecedência todos os lançamentos de dados que ocorreriam no restante do jogo, a resolução de um jogo com dados seria muito semelhante à resolução de um jogo sem dados, o que minimax faz no tempo  $O(b^m)$ , onde  $b$  é o fator de ramificação e  $m$  é a profundidade máxima da árvore de jogo. Como expectiminimax também está considerando todas as sequências de lançamentos de dados possíveis, ele levará o tempo  $O(b^m n^m)$ , onde  $n$  é o número de lançamentos distintos.

Ainda que a profundidade da busca se limitasse a alguma profundidade pequena  $d$ , o custo extra comparado com o de minimax tornaria pouco realista considerar a possibilidade de examinar uma distância muito grande à frente na maioria dos jogos de azar. Em gamão,  $n$  é 21 e  $b$  em geral é cerca de 20, mas, em algumas situações, ele pode chegar a 4.000 em lançamentos de dados que resultam em valores duplos. Talvez essas jogadas sejam tudo o que poderíamos administrar.

Outro modo de pensar no problema é: a vantagem de alfa-beta é que ela ignora desenvolvimentos futuros que simplesmente não irão acontecer, dada a melhor jogada. Desse modo, ela se concentra em ocorrências prováveis. Em jogos com dados, não há *nenhuma* sequência provável de movimentos porque, para que esses movimentos ocorressem, os dados primeiro teriam de cair da maneira correta para torná-los válidos. Esse é um problema geral sempre que a incerteza entra em cena: as possibilidades são enormemente multiplicadas, e a formação de planos de ação detalhados se torna inútil porque o mundo talvez não acompanhe o jogo.

Sem dúvida deve ter ocorrido ao leitor que talvez algo como a poda alfa-beta poderia ser aplicada a árvores de jogos com nós de acaso. Na verdade, isso é possível. A análise para nós de MIN e

MAX fica inalterada, mas também podemos podar nós de acaso, usando um pouco de engenhosidade. Considere o nó de acaso  $C$  da Figura 5.11 e o que acontece ao seu valor à medida que examinamos e avaliamos seus filhos. É possível encontrar um limite superior sobre o valor de  $C$  antes de examinarmos todos os seus filhos? (Lembre-se de que alfa-beta precisa disso para podar um nó e sua subárvore.) À primeira vista, pode parecer impossível porque o valor de  $C$  é a *média* dos valores de seus filhos e, a fim de calcular a média de um conjunto de números, temos que verificar todos os números. No entanto, se impusermos limites sobre os valores possíveis da função utilidade, poderemos chegar a limites para a média sem verificar todos os números. Por exemplo, se dissermos que todos os valores de utilidade estão entre  $-2$  e  $+2$ , o valor de nós folhas será limitado e, nesse caso, *poderemos* impor um limite superior sobre o valor de um nó de acaso sem examinar todos os seus filhos.

Uma alternativa é fazer a **simulação de Monte Carlo** para avaliar uma posição. Comece com um algoritmo de busca alfa-beta (ou outro).

A partir de uma posição inicial, faça com que o algoritmo jogue milhares de jogos contra si mesmo, usando arremessos de dados aleatórios. No caso do gamão, o percentual de vitórias resultante tem se mostrado uma boa aproximação do valor da posição, mesmo que o algoritmo tenha uma heurística imperfeita e esteja em busca apenas de algumas jogadas (Tesauro, 1995). Para jogos com dados, esse tipo de simulação chama-se **lançamento**.

## 5.6 JOGOS PARCIALMENTE OBSERVÁVEIS

---

Muitas vezes, o xadrez tem sido descrito como guerra em miniatura, mas carece de pelo menos uma característica importante de guerras reais, chamada **observabilidade parcial**. Na “névoa da guerra”, a existência e a disposição das unidades inimigas é muitas vezes desconhecida até ser revelada por contato direto. Como resultado, a guerra inclui o uso de observadores e espiões para colher informações e uso de dissimulação e blefe para confundir o inimigo. Os jogos parcialmente observáveis compartilham essas características e são, portanto, qualitativamente diferentes dos jogos descritos nas seções anteriores.

### 5.6.1 Kriegspiel: xadrez parcialmente observável

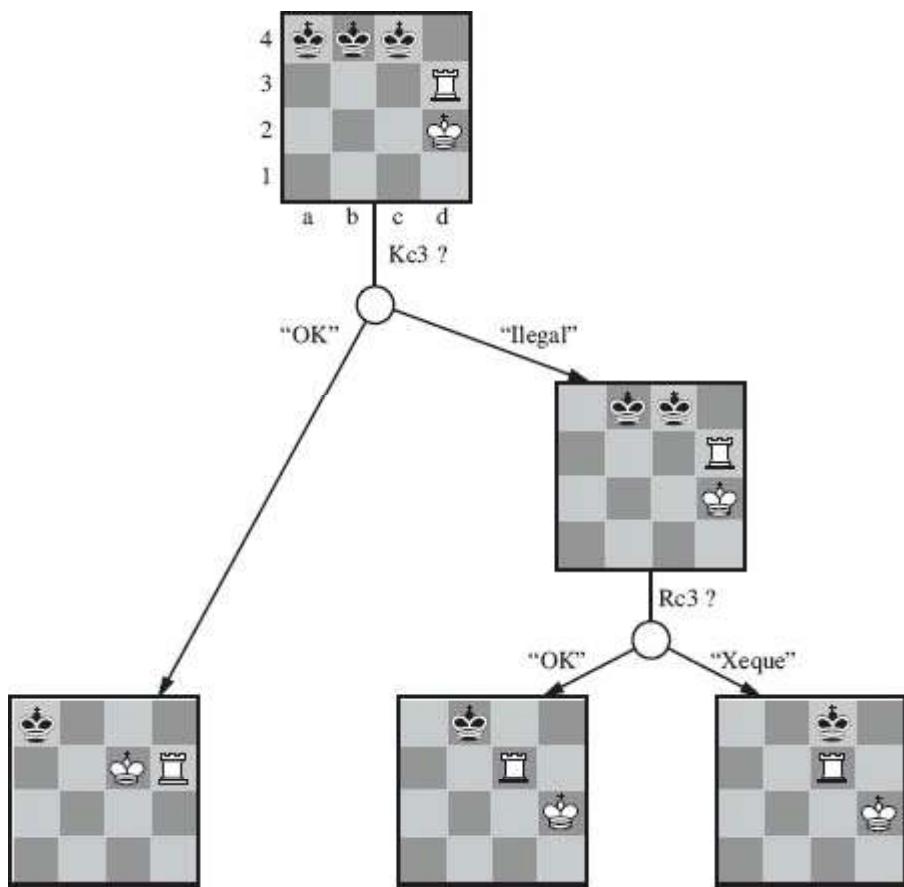
Em jogos *determinísticos* parcialmente observáveis, a incerteza sobre o estado do tabuleiro resulta inteiramente da falta de acesso às escolhas feitas pelo adversário. Essa categoria inclui os jogos infantis como batalha naval (em que cada jogador coloca os navios em locais escondidos do adversário e não se movem) e Stratego (no qual a localização das peças é conhecida, mas os tipos das peças permanecem ocultos). Vamos examinar o jogo de **Kriegspiel**, uma variante parcialmente observável de xadrez em que as partes podem se mover, mas são completamente invisíveis para os adversários.

As regras do Kriegspiel são as seguintes: brancas e pretas veem um tabuleiro que contém apenas suas próprias peças. Um árbitro, que pode ver todas as peças, julga o jogo e faz anúncios periódicos

que os dois jogadores escutam. Por sua vez, as brancas propõem ao árbitro qualquer movimento que seria legal se não houvesse peças pretas. Se o movimento de fato não for legal (por causa das peças pretas), o árbitro anuncia “ilegal”. Nesse caso, as brancas podem manter os movimentos propostos até que seja encontrado um legal — e aprendem mais sobre a localização das peças pretas no processo. Uma vez que seja proposto um lance legal, o árbitro anuncia uma ou mais das seguintes opções: “Capture no quadrado  $X$ ” se houver uma captura, e “Xeque em  $D$ ” se o rei preto estiver em xeque, onde  $D$  é a direção do xeque e pode ser um dos “cavalos”, “linha”, “coluna”, “diagonal longa” ou “diagonal curta” (no caso de xeque a descoberto, o árbitro pode fazer dois anúncios de “xeque”). Se houver um xeque-mate ou afogamento nas pretas, o árbitro avisa; caso contrário, é a vez do lance do preto.

O Kriegspiel pode parecer terrivelmente impossível, mas os humanos o administram muito bem e os programas de computador estão começando a alcançá-lo. Ajuda lembrar a noção de **estado de crença**, como definido na Seção 4.4 e ilustrado na Figura 4.14 — o conjunto de todos os estados do tabuleiro *logicamente possível* dá o histórico completo das percepções até o momento. Inicialmente, o estado de crença branco é uma peça avulsa porque as peças pretas ainda não se moveram. Após um movimento das brancas e uma resposta das pretas, o estado de crença das brancas contém 20 posições porque as pretas têm 20 respostas a qualquer movimento das brancas. Manter o controle do estado de crença no decorrer do jogo é exatamente o problema da **estimativa de estado**, para o qual a etapa de atualização é dada na Equação 4.6. Podemos mapear a estimativa de estado do Kriegspiel diretamente para o quadro parcialmente observável, não determinístico da Seção 4.4, se considerarmos o adversário como fonte de não determinismo, ou seja, os **RESULTADOS** do movimento das brancas são compostos do resultado (previsível) do próprio movimento das brancas e pelo resultado imprevisível dado pela resposta das pretas.<sup>3</sup>

Dado um estado de crença atual, as brancas podem perguntar: “Posso ganhar o jogo?” Para um jogo parcialmente observável, a noção de **estratégia** é alterada; em vez de especificar um movimento a ser feito para cada *movimento* possível que o adversário possa fazer, precisamos de um movimento para cada *sequência de percepção* possível que possa ser recebida. Para o Kriegspiel, uma estratégia vencedora, ou **xeque-mate garantido**, é aquela que, para cada sequência de percepção possível, leva a um xeque-mate real para cada estado do tabuleiro possível no estado de crença atual, independentemente da forma como o adversário se move. Com essa definição, o estado de crença do adversário é irrelevante — a estratégia tem que funcionar, mesmo que o adversário possa ver todas as peças. Isso simplifica muito cálculo. A Figura 5.13 mostra parte de um xeque-mate garantido para um final de jogo RTR (rei, torre contra rei). Nesse caso, as pretas têm apenas uma peça (o rei); assim, um estado de crença para as brancas pode ser mostrado em um tabuleiro simples marcando cada posição possível do rei preto.



**Figura 5.13** Parte de um xeque-mate garantido RTR, mostrado ao final do jogo, em um tabuleiro reduzido. No estado de crença inicial, o rei preto está em uma das três localizações possíveis. Através de uma combinação de movimentos de sondagem, a estratégia se reduz para um. A conclusão do xeque-mate é deixada como exercício.

O algoritmo genérico de busca E-OU pode ser aplicado no espaço de estado de crença para encontrar os xeque-mates garantidos, como na Seção 4.4. O algoritmo de estado de crença incremental mencionado na seção muitas vezes encontra xeque-mates no meio do jogo até uma profundidade 9 — provavelmente muito além das habilidades dos jogadores humanos.

Além dos xeque-mates garantidos, o Kriegspiel admite um conceito inteiramente novo que não faz sentido em jogos totalmente observáveis: **xeque-mate probabilístico**. Ainda é requerido que tais xeque-mates funcionem em cada estado do tabuleiro em estado de crença; são probabilísticos com respeito à randomização dos movimentos do jogador vencedor. Para obter a ideia básica, considere o problema de encontrar um rei preto solitário utilizando apenas o rei branco. Simplesmente movendo de forma aleatória, o rei branco *eventualmente* dará de cara com o rei preto, mesmo que este último tente evitar esse destino, desde que o preto não pode se manter imaginando movimentos evasivos corretos indefinidamente. Na terminologia da teoria da probabilidade, a detecção ocorre *com probabilidade* 1. O final de jogo RBCR — rei, bispo e cavalo, contra o rei — está ganha nesse sentido; a branca apresenta à preta uma sequência infinita aleatória de escolhas, para uma das quais a preta vai imaginar incorretamente e revelar sua posição, levando ao xeque-mate. O fim do jogo RBBR, por outro lado, se ganha com probabilidade  $1 - \epsilon$ .

A branca pode forçar uma vitória apenas, deixando um de seus bispos sem proteção em um movimento. Se acontecer de a preta estar no lugar certo e capturar o bispo (um movimento que iria perder se o bispo estivesse protegido), a partida estará empatada. A branca pode optar por um

movimento arriscado em algum ponto escolhido randomicamente no meio de uma distância muito longa, reduzindo assim  $\epsilon$  para uma constante arbitrariamente pequena, mas não pode reduzir  $\epsilon$  para zero.

É muito raro que um xeque-mate garantido ou probabilístico possa ser encontrado em qualquer profundidade razoável, exceto ao final do jogo. Às vezes, uma estratégia de xeque-mate funciona para *alguns* dos estados do tabuleiro no estado de crença atual, mas não para outros. Tentar tal estratégia pode ter sucesso, levando a um **xeque-mate acidental** — accidental no sentido de que a branca não poderia *saber* que seria xeque-mate — se acontecer de as peças pretas estarem nos lugares certos (a maioria dos xeque-mates em jogos entre os seres humanos é dessa natureza acidental). Essa ideia leva naturalmente à questão de *o quanto provável* é que determinada estratégia irá vencer, o que leva, por sua vez, à questão do *quanto é provável* que cada estado do tabuleiro, no estado de crença atual, é o verdadeiro estado do tabuleiro.

 Uma primeira inclinação deverá ser propor que todas as posições do tabuleiro, no estado de crença atual, sejam igualmente prováveis, mas isso pode não estar certo. Considere, por exemplo, o estado de crença da branca após o primeiro lance do jogo da preta. Por definição (assumindo que a preta desempenha otimamente), a preta deve ter jogado um ótimo lance, e assim deveria ser atribuída uma probabilidade zero a todas as posições do tabuleiro resultantes de lances subótimos. Esse argumento não é muito certo também porque o objetivo de *cada jogador não é apenas mover as peças para os quadrados à direita, mas também minimizar a informação que o adversário tem sobre sua localização*. Jogar qualquer estratégia “ótima” previsível fornece informações ao adversário. Por isso, o jogo ideal em jogos parcialmente observáveis requer uma vontade de jogar de alguma forma *ao acaso* (é por isso que os inspetores de higiene de restaurante fazem visitas de inspeção aleatórias). Isso significa selecionar ocasionalmente lances que podem parecer “intrinsecamente” fracos, mas ganham força a partir de sua forte imprevisibilidade porque é improvável que o adversário tenha preparado qualquer defesa contra eles.

A partir dessas considerações, parece que as probabilidades associadas com as posições do tabuleiro no estado de crença atual só podem ser calculadas a partir de uma estratégia ótima randomizada; por sua vez, o cálculo dessa estratégia parece exigir conhecimento das probabilidades das diversas posições que o tabuleiro possa ter. Esse enigma pode ser resolvido adotando a noção da teoria dos jogos de uma solução de **equilíbrio**, que será abrangida no Capítulo 17. Um equilíbrio especifica uma estratégia ótima randomizada para cada jogador. Porém, calcular o equilíbrio é proibitivamente caro, mesmo para pequenos jogos, e está fora de questão para o Kriegspiel. Atualmente, o projeto de algoritmos eficientes para o jogo geral de Kriegspiel é um tópico de busca aberta. A maioria dos sistemas realiza a perspectiva de profundidade limitada em seu próprio espaço de estado de crença, ignorando o estado de crença do adversário. As funções de avaliação são semelhantes às do jogo observável, mas incluem um componente para o tamanho do estado de crença — quanto menor, melhor!

## 5.6.2 Jogos de cartas

Os jogos de cartas dão muitos exemplos de observabilidade parcial *estocástica*, onde a falta de

informação é gerada aleatoriamente. Por exemplo, em muitos jogos, no início do jogo as cartas são distribuídas aleatoriamente, com cada jogador recebendo uma mão que não é visível para os outros jogadores. Tais jogos incluem *bridge*, *whist*, *hearts* e algumas formas de pôquer.

À primeira vista, pode parecer que esses jogos de cartas são como jogos de dados: as cartas são distribuídas de forma aleatória e determinam as jogadas disponíveis para cada jogador, mas no início todos os “dados” são jogados! Mesmo que essa analogia acabe sendo incorreta, ela sugere um algoritmo efetivo: considerar todas as distribuições possíveis das cartas invisíveis, resolver cada uma como se fosse um jogo totalmente observável, e então escolher o lance que tem a melhor média de resultado sobre todos os lances. Suponha que cada mão ocorra com a probabilidade  $P(s)$ , então o lance que queremos é

$$\operatorname{argmax}_a \sum_s P(s) \operatorname{MINIMAX}(\operatorname{RESULTADO}(s, a)). \quad (5.1)$$

Aqui, executaremos o MINIMAX exato se computacionalmente viável; caso contrário, executaremos o H-MINIMAX.

Agora, na maioria dos jogos de carta, o número de mãos possíveis é bastante grande. Por exemplo, no jogo de *bridge*, cada jogador vê apenas duas das quatro mãos; existem duas mãos invisíveis, de 13 cartas cada, então o número de mãos é  $(26/13) = 10.400.600$ . A resolução de uma mão é muito difícil, por isso resolver 10 milhões está fora de questão. Em vez disso, recorreremos a uma aproximação de Monte Carlo: em vez de somar *todas* as rodadas, tomamos uma *amostra aleatória* de  $N$  rodadas, onde a probabilidade de a rodada  $s$  aparecer na amostra é proporcional a  $P(s)$ :

$$\operatorname{argmax}_a \frac{1}{N} \sum_{i=1}^N \operatorname{MINIMAX}(\operatorname{RESULT}(s_i, a)). \quad (5.2)$$

[Observe que  $P(s)$  não aparece explicitamente no somatório porque as amostras já estão extraídas de acordo com  $P(s)$ .] À medida que  $N$  aumenta, a soma sobre a amostra aleatória tende ao valor exato, mas mesmo para  $N$  relativamente pequeno — digamos, de 100 a 1.000 — o método dá uma boa aproximação. Pode também ser aplicado a jogos determinísticos, como o de Kriegspiel, a partir de algumas estimativas razoáveis de  $P(s)$ .

Para jogos como *whist* e *hearts*, nos quais não há fase de lance ou de apostas antes de o jogo começar, cada rodada será igualmente provável e, assim, os valores de  $P(s)$  são todos iguais. Para o *bridge*, o jogo é precedido por uma fase de leilão, em que cada equipe indica quantos *tricks* espera ganhar. Como os jogadores fazem seus lances com base em suas cartas, os outros jogadores aprendem mais sobre a probabilidade de cada lance. Levar isso em conta para decidir como jogar a mão é complicado, pelas razões mencionadas em nossa descrição do Kriegspiel: os jogadores podem fazer os lances de forma a minimizar a informação transmitida aos seus adversários. Mesmo assim, a abordagem é bastante eficaz para o *bridge*, como mostraremos na Seção 5.7.

A estratégia descrita nas Equações 5.1 e 5.2 chama-se às vezes *média sobre clarividência* porque assume que o jogo vai se tornar observável para ambos os jogadores imediatamente após o primeiro lance. Apesar de seu apelo intuitivo, a estratégia pode conduzir a um extravio. Considere a seguinte história:

Dia 1: A estrada *A* leva a um monte de moedas de ouro; a estrada *B* leva a uma bifurcação. Tome a estrada da esquerda e você encontrará uma grande pilha de ouro; tome a estrada da direita e você será atropelado por um ônibus.

Dia 2: A estrada *A* leva a um monte de moedas de ouro; a estrada *B* leva a uma bifurcação. Tome a estrada da direita e você encontrará uma grande pilha de ouro; tome a estrada da esquerda e você será atropelado por um ônibus.

Dia 3: A estrada *A* leva a um monte de moedas de ouro; a estrada *B* leva a uma bifurcação. Um ramo da bifurcação leva a uma grande pilha de ouro, mas, se tomar o caminho errado será atropelado por um ônibus. Infelizmente, você não sabe qual é o correto.

A média sobre clarividência leva ao seguinte raciocínio: no dia 1, *B* é a escolha certa; no dia 2, *B* é a escolha certa; no dia 3, a situação é a mesma que a do dia 1 ou 2, então *B* deverá ser ainda a escolha certa.

Agora podemos ver como uma média sobre a clarividência falha: não considera o *estado de crença* em que o agente estará depois da ação. Um estado de crença de total ignorância não é desejável, especialmente quando uma possibilidade certa é a morte. Por assumir que cada estado futuro será automaticamente de perfeito conhecimento, a abordagem nunca seleciona ações que *reúnem informações* (como o primeiro lance na Figura 5.13); nem vai escolher as ações que escondem informação do adversário ou fornece informação a um parceiro porque se assume que eles já conhecem as informações e ela nunca vai **blefar** no pôquer,<sup>4</sup> pois assume que o adversário pode ver as suas cartas. No Capítulo 17, vamos mostrar como construir algoritmos que fazem todas essas coisas pela virtude de resolver o verdadeiro problema de decisão parcialmente observável.

## 5.7 PROGRAMAS DE JOGOS DE ÚLTIMA GERAÇÃO

---

Em 1965, o matemático russo Alexander Kronrod chamou o xadrez de “*Drosophila* inteligência artificial”. John McCarthy discorda: enquanto os geneticistas usam moscas de frutas para fazer descobertas que se aplicam à biologia de forma mais ampla, a IA usou o xadrez para fazer o equivalente à criação de moscas de fruta muito rápidas. Talvez a melhor analogia seja de que o xadrez é para a IA o mesmo que a corrida automobilística de Grand Prix é para a indústria do automóvel: programas de última geração de jogo são incrivelmente rápidos, máquinas altamente otimizadas, que incorporam os últimos avanços da engenharia, mas eles não são muito úteis para fazer as compras ou dirigir fora da estrada. No entanto, corridas e jogos geram excitação e um fluxo constante de inovações que são adotadas por uma comunidade maior. Nesta seção, verificaremos o que é preciso para se sair bem em vários jogos.

**Xadrez:** o programa de xadrez da IBM, Deep Blue, agora aposentado, foi muito conhecido por derrotar o campeão mundial Garry Kasparov em um jogo de exibição amplamente divulgado. O Deep Blue executou em um computador paralelo com 30 processadores IBM RS/6000 fazendo busca alfa-beta. A única parte era uma configuração personalizada de 480 processadores de xadrez VLSI que realizava a geração de movimento e de ordenação para os últimos poucos níveis da árvore e avaliava os nós folha. O Deep Blue buscava até 30 bilhões de posições por movimento, alcançando

rotineiramente uma profundidade igual a 14. O coração da máquina é uma busca alfa-beta de aprofundamento iterativo padrão com uma tabela de transposição, mas a chave de seu sucesso parece ter sido sua habilidade de gerar extensões além do limite de profundidade para linhas suficientemente interessantes de movimentos forçados. Em alguns casos, a busca alcançou uma profundidade de 40 jogadas. A função de avaliação tinha mais de 8.000 características, muitas delas descrevendo padrões de peças altamente específicos. Foi usado um “livro de aberturas” com aproximadamente 4.000 posições, bem como um banco de dados de 700.000 jogos de grandes mestres a partir do qual podiam ser extraídas recomendações consensuais. O sistema também utilizava um grande banco de dados de finais de jogos com posições resolvidas, contendo todas as posições com cinco peças e muitas com seis peças. Esse banco de dados tem o efeito de estender de forma significativa a profundidade efetiva da busca, permitindo ao Deep Blue jogar com perfeição em alguns casos, até mesmo quando está a muitos movimentos de distância do xeque-mate.

O sucesso do Deep Blue reforçou a ampla convicção de que o progresso dos jogos de computadores vinha principalmente de um hardware cada vez mais poderoso — uma visão encorajada pela IBM. Mas as melhorias algorítmicas têm permitido aos programas executar em PCs-padrão, para ganhar campeonatos mundiais de xadrez por computador. Diversas heurísticas de poda são usadas para reduzir o fator de ramificação efetivo a menos de 3 (comparado ao fator de ramificação real de aproximadamente 35). A mais importante delas é a heurística de **movimento nulo**, que gera um bom limite inferior sobre o valor de uma posição, usando uma busca rasa na qual o oponente chega ao dobro de movimentos no início. Esse limite inferior frequentemente permite a poda alfa-beta sem o custo de uma busca em profundidade total. Também é importante a **poda de futilidades**, que ajuda a decidir com antecedência que movimentos causarão um corte beta nos nós sucessores.

O HYDRA pode ser visto como o sucessor do Deep Blue. O HYDRA executa em um cluster de processador de 64 com 1 gigabyte por processador e com hardware personalizado em forma de chips FPGA (*field programmable gate array*: arranjo de portas programável em campo). O HYDRA atinge 200 milhões de avaliações por segundo, o mesmo que o Deep Blue, mas alcança 18 camadas de profundidade, em vez de apenas 14, por causa do uso agressivo da heurística de movimento nulo e poda adiantada.

O RYBKA, vencedor do Campeonato Mundial de Xadrez de Computador de 2008 e 2009, é considerado o computador jogador mais forte do momento. Ele utiliza um processador Xeon de 8-core e 3,2 GHz Intel imediatamente disponível, mas pouco se sabe sobre a concepção do programa. Sua principal vantagem parece ser a sua função de avaliação, que foi ajustada pelo seu desenvolvedor principal, o mestre internacional Vasik Rajlich, e pelo menos três outros grandes mestres.

Os jogos mais recentes sugerem que os programas de computador mais avançados de xadrez impulsionaram todos os competidores humanos (veja as observações históricas para detalhes).

**Jogo de damas:** Jonathan Schaeffer e seus colegas desenvolveram o CHINOOK, que executa em PCs e utiliza busca alfa-beta. O Chinook derrotou o campeão humano de longa data em uma partida abreviada em 1990, e desde 2007 tem sido capaz de jogar perfeitamente utilizando busca alfa-beta combinada com uma base de dados de 39 trilhões de posições finais.

O jogo **Othello**, também chamado Reversi, provavelmente é mais popular como jogo de computador do que como jogo de tabuleiro. Ele tem um espaço de busca menor que o do xadrez, em geral de 5-15 movimentos válidos, mas a experiência de avaliação teve de ser desenvolvida desde o início. Em 1997, o programa Logistello (Buro, 2002) derrotou o campeão mundial humano, Takeshi Murakami, por seis jogos a zero. De modo geral, todos reconhecem que os seres humanos não são capazes de superar os computadores no Othello.

**Gamão:** A Seção 6.5 explicou por que a inclusão da incerteza dos lançamentos de dados torna uma busca profunda um luxo dispendioso. A maior parte do trabalho em gamão se dedicou a melhorar a função de avaliação. Gerry Tesauro (1992) combinou o método de aprendizado de reforço de Samuel com as técnicas de redes neurais para desenvolver um avaliador notavelmente preciso que é utilizado com uma busca até a profundidade 2 ou 3. Depois de disputar mais de um milhão de jogos de treinamento contra si mesmo, o programa de Tesauro, TD-GAMMON ficou competitivo com os melhores jogadores humanos. Em alguns casos, as opiniões do programa nos movimentos de abertura do jogo alteraram de forma radical a sabedoria adquirida.

O **Go** é o jogo de tabuleiro mais popular na Ásia, exigindo de seus profissionais, no mínimo, tanta disciplina quanto o xadrez. Como o tabuleiro tem  $19 \times 19$  e os lances são permitidos na entrada de (quase) todos os quadrados em branco, o fator de ramificação começa em 361, um valor assustador para os métodos de busca alfa-beta comum. Além disso, é difícil escrever uma função de avaliação porque o controle do território é muitas vezes imprevisível até o fim do jogo. Portanto, os programas de topo, como o MoGo, evitam a busca alfa-beta e, em vez disso, utilizam os lançamentos de Monte Carlo. A malícia é decidir que lances fazer no curso do lançamento. Não há poda agressiva; todos os movimentos são possíveis. O método de limites de confiança superior em árvores funciona ao fazer movimentos aleatórios nas primeiras poucas iterações e, ao longo do tempo, guiando o processo de amostragem para selecionar os lances que levaram a vitórias nas amostras anteriores. São adicionados alguns truques, incluindo *regras baseadas em conhecimento* que sugerem lances em particular sempre que determinado padrão foi detectado e *limitou a busca local* para decidir questões táticas. Alguns programas também incluem técnicas especiais da **teoria combinatória dos jogos** para analisar finais de jogos. Essas técnicas decompõem uma posição em suposições que podem ser analisadas separadamente e depois combinadas (Berlekamp e Wolfe, 1994; Muller, 2003). As soluções ótimas obtidas dessa forma têm surpreendido muitos jogadores de Go profissional, que pensavam que estavam jogando otimamente o tempo todo. Os programas atuais para Go funcionam em nível proprietário em tabuleiro  $9 \times 9$  reduzido, mas ainda estão em nível amador avançado em um tabuleiro completo.

O **bridge** é um jogo de cartas de informações imperfeitas: as cartas de um jogador ficam ocultas dos outros jogadores. O *bridge* também é um jogo de *vários participantes* com quatro jogadores em vez de dois, embora os jogadores formem duas equipes. Como vimos na Seção 5.6, o jogo ótimo em jogos parcialmente observáveis como o *bridge* pode incluir elementos de aquisição de informações, comunicação, blefe e cuidadosa ponderação de probabilidades. Muitas dessas técnicas são usadas no programa Bridge Baron<sup>TM</sup> (Smith *et al.*, 1998), que venceu o campeonato de *bridge* por computador de 1997. Embora não jogue muito bem, o Bridge Baron é um dos poucos sistemas de jogos bem-sucedidos a usar planos hierárquicos complexos (veja o Capítulo 11) que envolvem ideias de alto

nível como **trapacear e forçar o descarte de trunfo**, familiares aos jogadores de *bridge*.

O programa GIB (Ginsberg, 1999) venceu o campeonato de *bridge* de 2000 decisivamente usando o método de Monte Carlo. Desde então, outros programas vencedores seguiram a conduta do GIB. A maior inovação do GIB é utilizar **generalização baseada em explanação** para calcular e armazenar na cache regras gerais para desempenho ótimo em várias classes-padrão de situações, em vez de avaliar cada situação individualmente. Por exemplo, em uma situação em que um jogador tem as cartas A-K-Q-J-4-3-2 de um naipe e outro jogador tem 10-9-8-7-6-5, existem  $7 \times 6 = 42$  maneiras para o primeiro jogador encabeçar e o segundo jogador seguir. Mas o GIB trata essas situações apenas como duas: o primeiro jogador pode tirar uma carta alta ou uma baixa; as cartas exatas não importam. Com essa otimização (e algumas outras), o GIB pode resolver as 52 cartas, uma mão plenamente observável *exatamente* em cerca de um segundo. A precisão tática do GIB contribui para sua inabilidade de raciocinar sobre as informações. Ele terminou em 12º lugar em um grupo de 35 no concurso de pares (envolvendo apenas o jogo da mão, não os lances) no campeonato mundial de 1998 para jogadores humanos, superando de longe as expectativas de muitos especialistas.

Existem várias razões para o desempenho do GIB em nível especialista com a simulação de Monte Carlo, considerando que os programas do Kriegspiel não o fazem. Primeiro, a avaliação de GIB da versão totalmente observável do jogo é exata, buscando da árvore de jogo completa, enquanto os programas Kriegspiel dependem da heurística inexata. Mas muito mais importante é o fato de que, no *bridge*, a maior parte da incerteza nas informações parcialmente observáveis vem da aleatoriedade do lance, não do jogo do adversário. A simulação de Monte Carlo trata bem a aleatoriedade, mas nem sempre lida bem com a estratégia, especialmente quando a estratégia envolve o valor da informação.

**Palavras cruzadas:** A maioria das pessoas acha que a parte mais difícil das palavras cruzadas é vir à baila boas palavras, mas, dado o dicionário oficial, acaba por ser bastante fácil programar um gerador de lance para encontrar o lance de pontuação mais alta (Gordon, 1994). No entanto, isso não significa que o jogo seja resolvido: tomar apenas o lance de melhor pontuação a cada vez é resultado de um jogador bom, mas não especialista. O problema é que a palavra cruzada é ao mesmo tempo parcialmente observável e estocástica: você não sabe que letras o outro jogador tem ou quais as próximas letras que você vai puxar. Portanto, jogar palavras cruzadas bem combina as dificuldades do gamão com o *bridge*. No entanto, em 2006, o programa QUACKLE derrotou o ex-campeão mundial, David Boys, por 3×2.

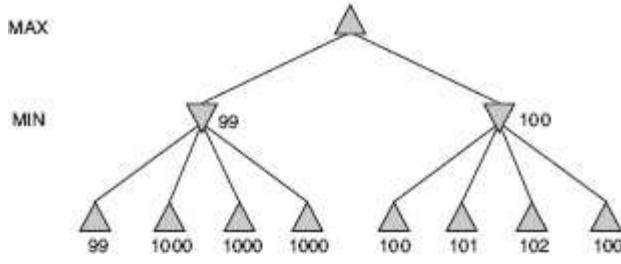
## 5.8 ABORDAGENS ALTERNATIVAS

---

Tendo em vista que o cálculo de decisões ótimas em jogos é intratável na maioria dos casos, todos os algoritmos devem fazer algumas suposições e aproximações. A abordagem-padrão, baseada em minimax, funções de avaliação e alfa-beta, é apenas uma maneira de fazer isso. Talvez porque tenha funcionado por tanto tempo, a abordagem-padrão foi desenvolvida intensivamente e domina outros métodos em jogos de torneios. Alguns especialistas no campo acreditam que isso tenha feito os jogos se divorciarem da parte principal da busca de IA: porque a abordagem-padrão não oferece mais

tanto espaço para novas ideias sobre questões gerais de tomada de decisões. Nesta seção, veremos as alternativas.

Primeiro, vamos considerar a heurística minimax. Ela seleciona um movimento ótimo em dada árvore de busca *desde que as avaliações de nós de folhas sejam exatamente corretas*. Na realidade, as avaliações são normalmente estimativas brutas do valor de uma posição e podemos considerar que há grandes erros associados a elas. A Figura 5.14 mostra uma árvore de jogo de duas jogadas para a qual o minimax sugere tomar o ramo da direita, porque  $100 > 99$ . Esse é o lance correto se as avaliações estiverem todas corretas. Mas é claro que a função de avaliação é apenas aproximada. Suponha que a avaliação de cada nó tenha um erro que é independente de outros nós e aleatoriamente distribuído com média zero e desvio padrão. Então, quando  $\sigma = 5$ , o ramo esquerdo é realmente melhor 71% do tempo e 58% do tempo quando  $\sigma = 2$ . A percepção por trás disso é que o ramo do lado direito tem quatro nós que estão próximos de 99; se um erro na avaliação de qualquer um dos quatro fizer com que o ramo da direita deslize abaixo de 99, então o ramo da esquerda é o melhor.



**Figure 5.14** Uma árvore de jogo de duas jogadas para a qual o minimax pode ser inadequado.

Na realidade, as circunstâncias são realmente piores do que isso porque o erro na função de avaliação *não* é independente. Se obtivermos um nó errado, as chances são altas de que os próximos nós na árvore também estarão errados. Porém, o fato de o nó identificado com 99 ter irmãos identificados com 1.000 sugere que, de fato, ele pode ter um valor verdadeiro mais alto. Podemos usar uma função de avaliação que retorna uma *distribuição de probabilidades* sobre valores possíveis, mas é difícil combinar essas distribuições corretamente porque não vamos ter um bom modelo das dependências muito fortes que existem entre os valores dos nós irmãos.

Em seguida, consideramos o algoritmo de busca que gera a árvore. O objetivo do projetista de algoritmos é especificar uma computação que funcione com rapidez e que gere um bom movimento. Os algoritmos alfa–beta foram projetados não apenas para selecionar um bom movimento, mas também para calcular limites sobre os valores de todos os movimentos válidos. Para ver por que essas informações extras são desnecessárias, considere uma posição em que só existe um movimento válido. A busca alfa–beta ainda irá gerar e avaliar uma árvore de busca grande, dizendo que o único lance é o melhor lance e atribuindo-lhe um valor. Mas, como temos que fazer o movimento de qualquer forma, conhecer o valor do movimento é inútil. Da mesma forma, se houver um lance obviamente bom e vários lances que são legais, mas levam a uma perda rápida, não vamos querer que o alfa-beta desperdice tempo determinando um valor preciso para apenas um bom lance. Melhor fazer apenas o movimento rapidamente e economizar tempo para mais tarde. Isso leva à ideia da *utilidade de uma expansão de nó*. Um bom algoritmo de busca deve selecionar expansões de nós de utilidade elevada, ou seja, aquelas que deverão levar à descoberta de um movimento significativamente melhor. Se não houver nenhuma expansão de nó cuja utilidade seja mais alta que

seu custo (em termos de tempo), o algoritmo deve interromper a busca e efetuar um movimento. Note que isso funciona não apenas para situações de claro favoritismo, mas também no caso de *movimentos simétricos*, para os quais nenhuma quantidade de busca mostrará que um movimento é melhor que outro.

Esse tipo de raciocínio que trata dos resultados obtidos com a computação é chamado **metarraciocínio** (raciocínio sobre o raciocínio). Ele se aplica não apenas aos jogos, mas a qualquer espécie de raciocínio. Todas as computações são feitas com a finalidade de tentar alcançar decisões melhores, todas têm custos e todas têm alguma probabilidade de resultar em certa melhoria na qualidade da decisão. A alfa-beta incorpora o tipo mais simples de metarraciocínio, ou seja, um teorema para o efeito de que certas ramificações da árvore podem ser ignoradas sem perda. É possível fazer muito melhor. No Capítulo 16, veremos como essas ideias podem se tornar exatas e implementáveis.

Finalmente, vamos reexaminar a natureza da própria busca. Os algoritmos para busca heurística e para jogos funcionam gerando sequências de estados concretos, começando pelo estado inicial e depois aplicando uma função de avaliação. É claro que não é assim que os seres humanos jogam. No xadrez, com frequência se tem em mente um objetivo específico — por exemplo, preparar uma armadilha para a rainha do oponente — e se pode usar esse objetivo para gerar *seletivamente* planos plausíveis para alcançá-lo. Esse tipo de raciocínio orientado para objetivos ou planejamento às vezes elimina por completo a busca combinatória. O Paradise de David Wilkins (1980) é o único programa a usar com sucesso o raciocínio orientado para objetivos no xadrez: ele foi capaz de resolver alguns problemas de xadrez que exigiam uma combinação de 18 movimentos. Até agora não existe nenhuma compreensão razoável de como *combinar* os dois tipos de algoritmos para formar um sistema eficiente e robusto, embora o Bridge Baron possa ser um passo na direção correta. Um sistema totalmente integrado seria uma realização significativa não apenas para a pesquisa na área de jogos, mas também para a pesquisa em IA em geral porque seria uma boa base para um agente inteligente geral.

## 5.9 RESUMO

---

Examinamos uma variedade de jogos para entender o que significa um jogo ótimo e para compreender como jogar bem na prática. As ideias mais importantes são:

- Um jogo pode ser definido pelo **estado inicial** (a forma como o tabuleiro é configurado), pelas **ações** válidas em cada estado, o **resultado** de cada ação, um **teste de término** (que informa quando o jogo é encerrado) e por uma **função utilidade** que se aplica a estados terminais.
- Em jogos de dois jogadores com soma zero e **informações perfeitas**, o algoritmo **minimax** pode selecionar movimentos ótimos usando uma enumeração da árvore de jogo em profundidade.
- O algoritmo de busca **alfa-beta** calcula o mesmo movimento ótimo que o minimax, mas alcança uma eficiência muito maior pela eliminação de subárvores comprovadamente irrelevantes.
- Em geral, não é possível considerar a árvore de jogo inteira (mesmo com alfa-beta) e, assim, precisamos cortar a busca em algum ponto e aplicar uma **função de avaliação** que fornece uma

estimativa da utilidade de um estado.

- Muitos programas de jogos pré-calculam tabelas das melhores jogadas no início e no final do jogo para que possam consultar uma jogada em vez de buscar.
- Os jogos de azar podem ser tratados por uma extensão do algoritmo minimax que avalia um **nó de acaso** tomando a utilidade média de todos os seus nós filhos, ponderada pela probabilidade de cada filho.
- O desempenho ótimo em jogos de **informações imperfeitas**, como o Kriegspiel e o *bridge*, exige raciocínio sobre os **estados de crença** corrente e futura de cada jogador. Uma aproximação simples pode ser obtida calculando-se a média dos valores de uma ação sobre cada configuração possível de informações omitidas.
- Os programas têm superado até mesmo jogadores humanos que são campeões em jogos, como xadrez, damas e Othello. Os seres humanos mantêm vantagem em vários jogos de informação imperfeita, como *bridge*, pôquer e Kriegspiel, e em jogos com fatores muito grandes de ramificação e pouco conhecimento heurístico bom, como o Go.

## NOTAS BIBLIOGRÁFICAS E HISTÓRICAS

---

A história inicial dos jogos mecânicos foi marcada por numerosas fraudes. A mais notória dessas fraudes foi a do barão Wolfgang von Kempelen (1734-1804, “o turco”), um suposto autômato jogador de xadrez que derrotou Napoleão antes de ser exposto como um armário de truques de mágica que alojava um especialista humano em xadrez (consulte Levitt, 2000). Ele jogou de 1769 até 1854. Em 1846, Charles Babbage (que tinha ficado fascinado com o turco) parece ter colaborado para a primeira discussão séria sobre a viabilidade dos jogos de xadrez e damas por computador (Morrison e Morrison, 1961). Ele não entendeu a complexidade exponencial das árvores de busca, alegando que “as combinações envolvidas na máquina analítica superavam enormemente qualquer uma requerida, até mesmo pelo jogo de xadrez”. Babbage também projetou, mas não construiu, uma máquina de uso especial para jogar o jogo da velha. A primeira máquina para jogos verdadeira foi construída por volta de 1890 pelo engenheiro espanhol Leonardo Torres y Quevedo. Ele se especializou no final do jogo de xadrez “KRK” (rei e torre contra rei), garantindo uma vitória com rei e torre a partir de qualquer posição.

Em geral, as origens do algoritmo minimax se localizam em um artigo publicado em 1912 por Ernst Zermelo, o desenvolvedor da moderna teoria de conjuntos. Infelizmente, o artigo continha vários erros e não descrevia o minimax de forma correta. Por outro lado, ele delineou as ideias da análise retrógrada e propôs (mas não provou) o que ficou conhecido como teorema de Zermelo: que o xadrez é determinado — as brancas ou as pretas podem forçar uma vitória ou é um empate, só não sabemos qual das opções. Zermelo diz que, se eventualmente soubéssemos, “o xadrez certamente perderia completamente o caráter de jogo”. Uma sólida base para a teoria de jogos foi desenvolvida no original trabalho *Theory of Games and Economic Behavior* (von Neumann e Morgenstern, 1944), que incluía uma análise mostrando que alguns jogos exigem estratégias aleatórias (ou, pelo menos, imprevisíveis). Consulte o Capítulo 17 para obter mais informações.

John McCarthy concebeu a ideia de busca alfa-beta em 1956, embora não a tivesse publicado. O

programa de xadrez NSS (Newell *et al.*, 1958) usava uma versão simplificada de alfa-beta; ele foi o primeiro programa de xadrez a usá-la. A poda alfa-beta foi descrita por Hart e Edwards (1961) e Hart *et al.* (1972). Alfa-beta também foi usada pelo programa de xadrez “Kotok-McCarthy”, escrito por um aluno de John McCarthy (Kotok, 1962). Knuth e Moore (1975) provaram a exatidão de alfa-beta e analisaram sua complexidade de tempo. Pearl (1982b) demonstra que alfa-beta é assintoticamente ótima entre todos os algoritmos de busca de árvores de jogos de profundidade fixa.

Várias tentativas foram feitas para superar os problemas com a “abordagem-padrão” esboçados na Seção 5.8. O primeiro algoritmo de busca heurística não exaustiva com algum embasamento teórico provavelmente foi o B\* (Berliner, 1979), que tentava manter limites intervalares sobre o valor possível de um nó na árvore de jogo, em vez de dar a ele uma única estimativa de valor pontual. Nós folhas são selecionados para expansão em uma tentativa de aprimorar os limites de nível superior até um único movimento se mostrar “claramente melhor”. Palay (1985) estende a ideia de B\*, utilizando distribuições de probabilidades sobre valores no lugar de intervalos. A busca de número de conspiração de David McAllester (1988) expande nós de folhas que, pela alteração de seus valores, poderiam fazer o programa preferir um novo movimento na raiz. O MGSS\* (Russell e Wefald, 1989) usa as técnicas de teoria da decisão do Capítulo 16 para estimar o valor da expansão de cada folha em termos da melhoria esperada na qualidade da decisão na raiz. Ele superou um algoritmo alfa-beta em Othello, apesar de buscar um número de nós uma ordem de magnitude menor. Em princípio, a abordagem do MGSS\* é aplicável ao controle de qualquer forma de deliberação.

Em muitos aspectos, a busca alfa-beta é a análoga para dois jogadores da busca em profundidade ramificada e limitada, dominada por A\* no caso de um único agente. O algoritmo SSS\* (Stockman, 1979) pode ser visualizado como um A\* de dois jogadores e nunca expande mais nós que alfa-beta para chegar à mesma decisão. Os requisitos de memória e a sobrecarga computacional da fila tornam impraticável o SSS\* em sua forma original, mas foi desenvolvida uma versão que necessita de espaço linear a partir do algoritmo RBFS (Korf e Chickering, 1996). Plaat *et al.* (1996) desenvolveram uma nova visão do SSS\* como uma combinação de alfa-beta e tabelas de transposição, mostrando como superar as desvantagens do algoritmo original e desenvolvendo uma nova variante chamada MTD(f) que foi adotada por vários programas importantes.

D. F. Beal (1980) e Dana Nau (1980, 1983) estudaram as deficiências do minimax aplicado a avaliações aproximadas. Eles mostraram que, sob certas suposições de independência sobre a distribuição de valores de folhas na árvore, o uso do minimax pode gerar valores na raiz que na realidade são *menos* confiáveis que o uso direto da própria função de avaliação. O livro de Pearl, *Heuristics* (1984), explica parcialmente esse paradoxo aparente e analisa muitos algoritmos de jogos. Baum e Smith (1997) propõem um substituto para o minimax baseado em probabilidades, mostrando que ele resulta em escolhas melhores em certos jogos. O algoritmo expectiminimax foi proposto por Donald Michie (1966). Bruce Ballard (1983) estendeu a poda alfa-beta para cobrir árvores com nós de acaso e Hauk (2004) reexaminou esse trabalho e proporcionou resultados empíricos.

Koller e Pfeffer (1997) descreveram um sistema para resolver completamente jogos parcialmente observáveis. O sistema é bastante geral, manipulação de jogos cuja estratégia ótima exige movimentos aleatórios e jogos que são mais complexos do que aqueles manipulados por qualquer sistema anterior. Ainda assim, não pode manipular jogos tão complexos como pôquer, *bridge* e

Kriegspiel. Franco *et al.* (1998) descreveram diversas variantes da busca de Monte Carlo, incluindo uma em que o MIN tem informação completa, mas o MAX, não. Entre os jogos determinísticos, parcialmente observáveis, o Kriegspiel recebeu mais atenção. Ferguson demonstrou estratégias randomizadas deduzidas à mão (*hand-derived*) para ganhar do Kriegspiel com um bispo e o cavalo (1992) ou dois bispos (1995) contra um rei. Os primeiros programas Kriegspiel concentravam-se em encontrar xeque-mates de final de jogo e executavam a busca E-OU no espaço do estado de crença (Sakuta e Iida, 2002; Bolognesi e Ciancarini, 2003). Algoritmos de estado de crença incremental habilitaram xeque-mates de meio de jogo muito mais complexos de ser encontrados (Russell e Wolfe, 2005; Wolfe e Russell, 2007), mas a estimativa de estado eficiente continua a ser o principal obstáculo para jogo geral efetivo (Parker *et al.*, 2005).

O **xadrez** foi uma das primeiras tarefas realizadas em IA, com esforços iniciais por muitos dos pioneiros da computação, incluindo Konrad Zuse em 1945, Norbert Wiener em seu livro *Cybernetics*(1948) e Alan Turing em 1950 (veja Turing *et al.*, 1953). Mas foi o artigo de Claude Shannon, *Programing a Computer for Playing Chess* (1950), que teve o mais completo conjunto de ideias, descrevendo uma representação de posições do tabuleiro, uma função de avaliação, a busca de quiescência e algumas ideias de seletiva (não exaustiva) busca de jogo de árvore. Slater (1950) e os comentaristas de seu artigo também exploraram as possibilidades para o jogo de xadrez de computador.

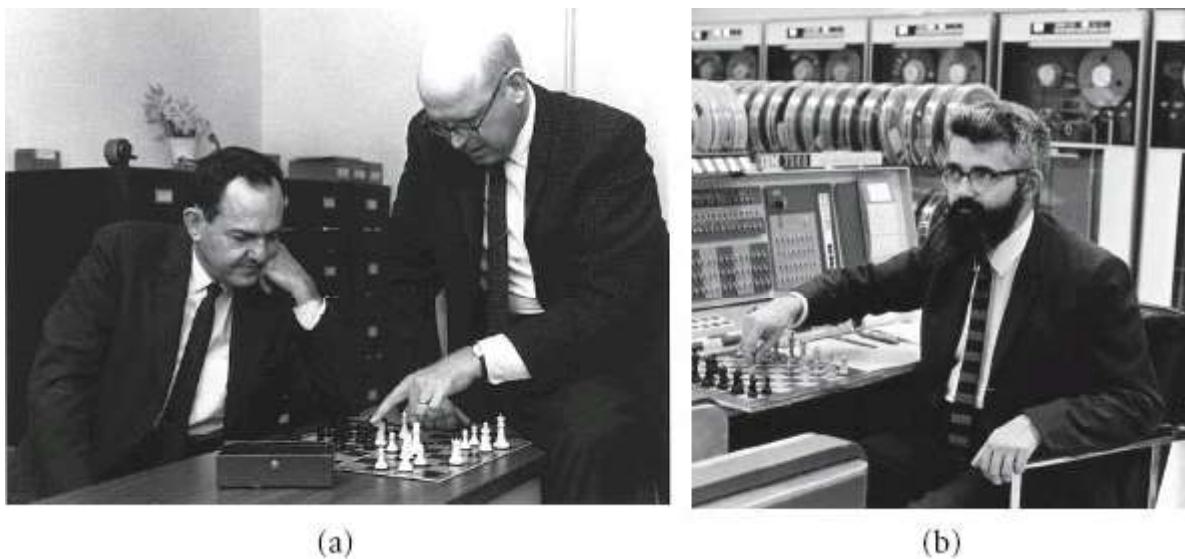
D.G. Prinz (1952) concluiu um programa que resolia problemas de final de jogo de xadrez, mas que não jogou um jogo completo. Stan Ulam e um grupo em Los Alamos National Lab produziram um programa que jogava xadrez em um tabuleiro de  $6 \times 6$  sem bispos (Kister *et al.*, 1957). Podia buscar quatro jogadas profundas em cerca de 12 minutos. Alex Bernstein escreveu o primeiro programa documentado para jogar um jogo completo de xadrez-padrão (Bernstein e Roberts, 1958).<sup>5</sup>

A primeira partida de xadrez em computador apresentou o programa de Kotok-McCarthy do MIT (Kotok, 1962), e o programa ITEP escrito em meados dos anos 1960 no Institute of Theoretical and Experimental Physics em Moscou (Adelson-Velsky *et al.*, 1970). Essa partida intercontinental foi realizada por telégrafo. Terminou com uma vitória de 3×1 para o programa ITEP, em 1967. O primeiro programa de xadrez a competir com sucesso com os seres humanos foi o MacHack-6 do MIT (Greenblatt *et al.*, 1967). Sua taxa Elo de cerca de 1.400 foi bem acima do nível iniciante de 1.000.

O Prêmio Fredkin, criado em 1980, ofereceu prêmios por metas progressivas no jogo de xadrez. O Belle, que atingiu uma classificação de 2.250 (Condon e Thompson, 1982), ganhou um prêmio de US\$5.000,00 pelo primeiro programa a conseguir uma classificação *master*. O prêmio de US\$10.000 para o primeiro programa a conseguir uma classificação USCF (United States Chess Federation) de 2.500 (perto do nível *grandmaster*) foi atribuído ao DEEP THOUGHT (Hsu *et al.*, 1990), em 1989. O Deep Blue (Campbell *et al.*, 2002; Hsu, 2004) recebeu o grande prêmio de US\$100.000,00, por sua vitória referencial sobre o campeão mundial Garry Kasparov em um jogo de exibição de 1997. Kasparov escreveu:

O jogo decisivo da partida foi o jogo 2, que deixou uma cicatriz em minha memória [...] vimos algo que foi bem além de nossas expectativas mais otimistas do quanto um computador seria capaz de prever as consequências posicionais em longo prazo das suas decisões. A máquina recusou-se a

se mover para uma posição que tinha uma vantagem decisiva no curto prazo — mostrando um senso de perigo muito humano. (Kasparov, 1997)



(a)

(b)

**Figura 5.15** Pioneiros no xadrez de computador: (a) Herbert Simon e Allen Newell, os desenvolvedores do programa NSS (1958); (b) John McCarthy e o programa Kotok-McCarthy em uma IBM 7090 (1967).

Provavelmente, a descrição mais completa de um programa de xadrez moderno foi fornecida por Ernst Heinz (2000), cujo programa DARKTHOUGHT foi o programa de PC não comercial mais bem classificado durante o campeonato mundial em 1999.

Nos últimos anos, os programas de xadrez estão se sobrepondo até mesmo aos melhores seres humanos do mundo. Entre 2004 e 2005, o HYDRA derrotou o grande mestre Eigen Vladimirov por  $3,5 \times 0,5$ , o campeão mundial Ruflam Ponomariov por  $2 \times 0$ , e o sétimo do ranking Michael Adams por  $5,5 \times 0,5$ . Em 2006, o DEEP FRITZ bateu o campeão mundial Vladimir Kramnik por  $4 \times 2$  e, em 2007, o RYBKA venceu diversos grandes mestres em jogos em que ele deu chances (como um peão) para os jogadores humanos. A partir de 2009, a mais alta classificação Elo já registrada foi de 2.851 por Kasparov. O HYDRA (Donninger e Lorenz, 2004) foi classificado em algum lugar entre 2.850 e 3.000, principalmente com base em sua derrota por Michael Adams. O programa RYBKA foi classificado entre 2.900 e 3.100, mas isso foi baseado em um pequeno número de jogos e não é considerado confiável. Ross (2004) mostrou como os seres humanos aprenderam a explorar algumas das fraquezas dos programas de computador.

O **jogo de damas**, em vez do xadrez, foi o primeiro dos jogos clássicos disputado inteiramente por um computador. Christopher Strachey (1952) escreveu o primeiro programa funcional para jogo de damas. Com início em 1952, Arthur Samuel, da IBM, trabalhando em seu tempo livre, desenvolveu um programa de damas que aprendeu sua própria função de avaliação, jogando consigo mesmo milhares de vezes (Samuel, 1959, 1967). Descreveremos essa ideia com mais detalhes no Capítulo 21. O programa de Samuel começou como um novato, mas depois de apenas alguns dias o autojogo tinha melhorado além do próprio nível de Samuel. Em 1962, derrotou Robert Nealy, campeão em “damas cegas”, através de um erro de sua parte. Quando se considera que os equipamentos de computação de Samuel (um IBM 704) tinham 10.000 palavras na memória principal, uma fita magnética para armazenamento de longo prazo e um processador de 0,000001 GHz, a vitória é

considerada uma grande realização.

O desafio iniciado por Samuel foi retomado por Jonathan Schaeffer, da Universidade de Alberta. Seu programa CHINOOK ficou em segundo lugar no Aberto dos Estados Unidos de 1990 e recebeu o direito de desafio para o campeonato mundial. Em seguida, enfrentou um problema chamado Marion Tinsley. O Dr. Tinsley tinha sido campeão do mundo há mais de 40 anos, perdendo apenas três jogos em todo esse tempo. Na primeira partida contra o Chinook, Tinsley sofreu suas quarta e quinta derrotas, mas venceu a partida por  $20,5 \times 18,5$ . Uma revanche no campeonato mundial de 1994 terminou prematuramente quando Tinsley teve que se retirar por motivos de saúde. O CHINOOK tornou-se oficialmente o campeão do mundo. Schaeffer continuou construindo seu banco de dados de finais de jogos e, em 2000, “resolveu” o jogo de damas (Schaeffer *et al.*, 2007; Schaeffer, 2008). Isso tinha sido previsto por Richard Bellman (1965). No documento que introduziu a abordagem de programação dinâmica para análise retrógrada, ele escreveu: “No jogo de damas, o número de movimentos possíveis em qualquer situação dada é tão pequeno que podemos esperar com confiança uma solução digital computacional completa para o problema de jogada ótima.” No entanto, Bellman não estimou plenamente o tamanho da árvore do jogo de damas. Existem cerca de 500 quatrilhões de posições. Depois de 18 anos de computação em um *cluster* de 50 ou mais máquinas, a equipe de Jonathan Schaeffer completou uma tabela de final de jogo para todas as posições do jogo de damas com 10 ou menos peças: mais de 39 trilhões de entradas. A partir daí, foram capazes de fazer busca alfa-beta adiantada para derivar uma política que prova que o jogo de damas é de fato um empate com o melhor jogo de ambos os lados. Observe que esta é uma aplicação de busca bidirecional (Seção 3.4.6). A construção de uma tabela de final de jogo para todos os jogos de damas seria impraticável: seria necessário um bilhão de gigabytes de armazenamento. A busca sem qualquer tabela também seria impraticável: a árvore de busca tem cerca de  $8^{47}$  posições, e levaria milhares de anos de pesquisa com a tecnologia atual. Só uma combinação de busca inteligente, dados de final de jogo e uma queda no preço dos processadores e da memória poderia resolver o jogo de damas. Assim, o jogo de damas juntou-se com o Qubic (Patashnik, 1980), o Connect Four (Allis, 1988) e o Nine-Men’s Morris (Gasser, 1998) como jogos que foram resolvidos por análise de computador.

O **gamão**, um jogo de azar, foi analisado matematicamente por Gerolamo Cardano (1663), mas foi tido como jogo de computador apenas no final de 1970, primeiro com o programa BKG (Berliner, 1980b); ele usou uma função de avaliação complexa, construída manualmente e pesquisada apenas à profundidade 1. Foi o primeiro programa a derrotar um campeão mundial humano em um jogo clássico maior (Berliner, 1980a). Berliner reconheceu prontamente que o BKG tinha muita sorte com os dados. O TD-Gammon de Gerry Tesauro (1995) jogou consistentemente em campeonato em nível mundial. O programa BGBLITZ foi o vencedor da Computer Olympiad de 2008.

O **Go** é um jogo determinístico, mas o grande fator de ramificação o torna um desafio. As questões-chave e a literatura prévia em computação com o Go foram resumidas por Bouzy e Cazenave (2001) e Muller (2002). Até 1997 não existiam programas de Go competentes. Hoje, os melhores programas jogam a *maioria* dos seus lances em nível de mestre; o único problema é que no decorrer de um jogo eles costumam fazer pelo menos um erro grave que permite que um oponente forte vença. Considerando que a busca alfa-beta reina na maioria dos jogos, muitos programas Go recentes têm adotado os métodos de Monte Carlo com base em esquema UCT (limite de confiança superior em árvores) (Kocsis e Szepesvari, 2006). O programa Go mais potente como o de 2009 é o

MOGO de Gelly e Silver (Wang e Gelly, 2007; Gelly e Silver, 2008). Em agosto de 2008, o MOGO marcou uma vitória surpreendente contra o profissional de alto nível Myungwan Kim, embora com o MoGo recebendo uma vantagem de nove pedras (o equivalente a uma vantagem da rainha no xadrez). Kim estimou a resistência de MoGo em 2-3 dan, o nível baixo de amador avançado. Para essa partida, o Mogo foi executado em um supercomputador com processador 800 de 15 teraflop (1.000 vezes o Deep Blue). Algumas semanas mais tarde, o Mogo, com apenas uma desvantagem de cinco pedras, ganhou contra um profissional por 6 dan. Na forma  $9 \times 9$  do Go, o MoGo está aproximadamente no nível profissional 1 dan. Os avanços rápidos, provavelmente como experimentação, continuam com novas formas da busca de Monte Carlo. O *Boletim Go Computer*, publicado pela Computer Go Association, descreve a evolução atual.

**Bridge:** Smith *et al.* (1998) relataram sobre como o seu programa baseado em planejamento ganhou o campeonato de *bridge* por computador de 1998 e descreveram (Ginsberg, 2001) como o seu programa GIB, com base na simulação de Monte Carlo, venceu o campeonato seguinte de computador, e surpreendentemente bem, contra jogadores humanos e conjuntos de problemas-padrão de livro. Em 2001-2007, o campeonato de *bridge* por computador foi vencido cinco vezes por JACK e duas vezes por WBRIDGE. Não existem artigos acadêmicos de nenhum dos dois explicando sua estrutura, mas os rumores é que ambos usaram a técnica de Monte Carlo, que foi proposta pela primeira vez para o jogo de *bridge* por Levy (1989).

**Palavras cruzadas:** Brian Sheppard (2002), seu criador, forneceu uma boa descrição de um programa de topo, o Maven. A geração de movimentos de pontuação mais alta é descrita por Gordon (1994), e a modelagem dos adversários foi coberta por Richards e Amir (2007).

**Futebol** (Kitano *et al.*, 1997b; Visser *et al.*, 2008) e **bilhar** (Lam e Greenspan, 2008; Archibald *et al.*, 2009) e outros jogos estocásticos com espaço contínuo de ações estão começando a atrair a atenção em IA, tanto na simulação como com jogadores robôs físicos.

Competições de jogos de computador ocorrem anualmente, e os artigos aparecem em uma variedade de locais. Os anais da conferência de nome bastante ilusório, *Heuristic Programming in Artificial Intelligence*, relatam as Computer Olympiads, que incluem grande variedade de jogos. O General Game Competition (Love *et al.*, 2006) testa programas que devem aprender a jogar determinado jogo desconhecido apenas com uma descrição lógica das suas regras. Também existem diversas coleções editadas de artigos importantes sobre pesquisa na área de jogos (Levy, 1988a, 1988b; Marsland e Schaeffer, 1990). A International Computer Chess Association (ICCA), fundada em 1977, publica o periódico trimestral *ICGA Journal* (antigamente denominado *ICCA Journal*). Foram publicados artigos importantes na antologia em série *Advances in Computer Chess*, começando com o artigo de Clarke (1977). O volume 134 do periódico *Artificial Intelligence* (2002) contém descrições de programas de última geração para xadrez, Othello, Hex, shogi, Go, gamão, pôquer, Scrabble<sup>TM</sup> e outros jogos. Desde 1998 tem sido realizada uma conferência bienal, *Computers and Games*.

## EXERCÍCIOS

---

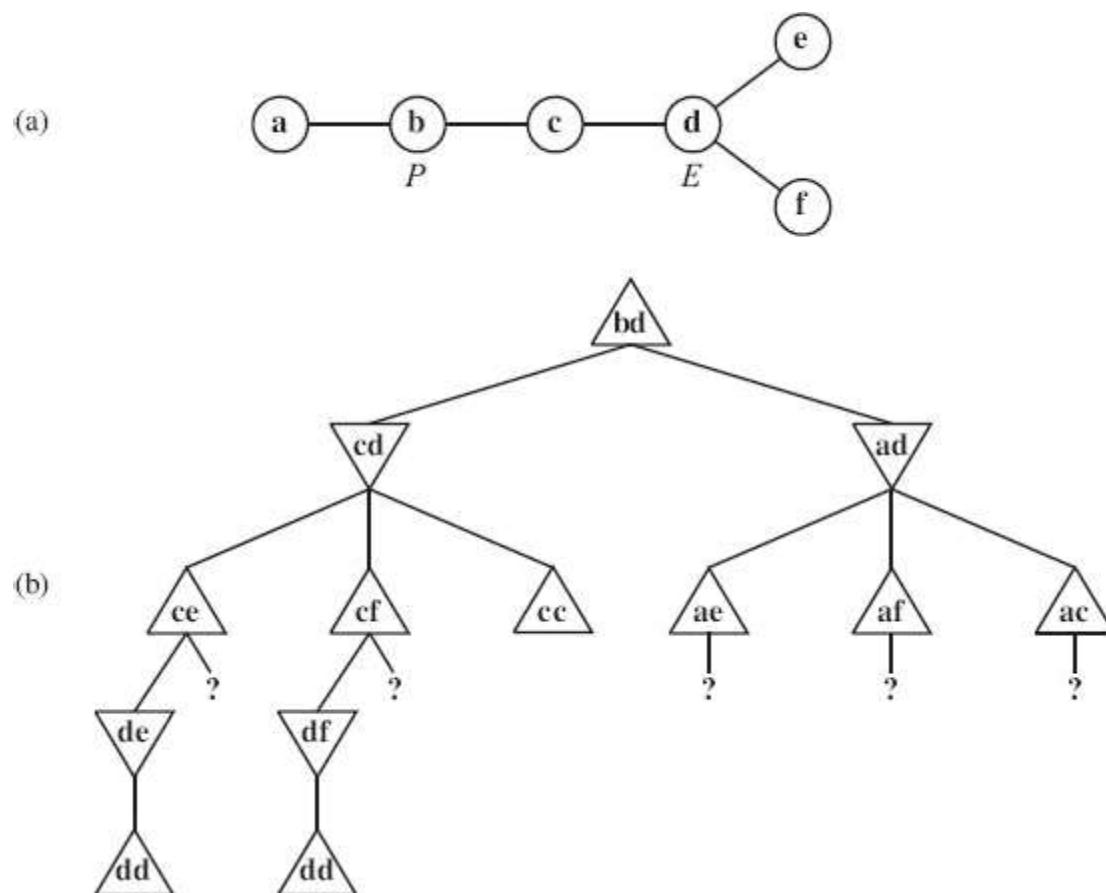
**5.1** Suponha que você tenha um oráculo,  $OM(s)$ , que prevê corretamente o lance do oponente em

qualquer estado. Utilizando isso, formule a definição de um jogo como um problema de busca (agente único). Descreva um algoritmo para encontrar o lance ótimo.

## 5.2 Considere o problema de resolver o quebra-cabeça de oito peças.

- Dê uma formulação completa do problema no estilo do Capítulo 3.
- Qual o tamanho do espaço de estados alcançável? Forneça uma expressão numérica exata.
- Suponha que desenvolvemos um problema adversário da seguinte forma: os dois jogadores se revezam em lance; uma moeda é arremessada para determinar o quebra-cabeça no qual o movimento deve ser feito nessa vez; e o vencedor será o primeiro a resolver um quebra-cabeça. Qual algoritmo pode ser usado para a escolha de um lance nesse cenário?
- Dê uma prova informal de que eventualmente alguém vai vencer se ambos jogarem perfeitamente.

**5.3** Imagine que, no Exercício 3.3, um dos amigos queira evitar o outro. O problema então se torna um jogo de dois jogadores de **perseguição e evasão**. Assumamos agora que os jogadores se movem por vez. O jogo só termina quando os jogadores estão no mesmo nó; o resultado final para o perseguidor é menos o tempo total necessário (o evasor “ganha” por nunca perder). A Figura 5.16 mostra um exemplo.



**Figura 5.16** (a) Um mapa onde o custo de cada aresta é 1. Inicialmente, o perseguidor  $P$  está no nó **b** e o evasor  $E$  está no nó **d**. (b) Uma árvore de jogo parcial para esse mapa. Cada nó é rotulado com as posições  $P$  e  $E$ .  $P$  move-se em primeiro lugar. Ainda não foram explorados os ramos marcados com “?”.

- Copie a árvore de jogo e marque os valores dos nós terminais.

- b. Ao lado de cada nó interno, escreva o fato mais forte que você pode inferir sobre o seu valor (um número, uma ou mais desigualdades, tais como “ $\geq 14$ ” ou um “?”).
- c. Abaixo de cada ponto de interrogação, escreva o nome do nó atingido por esse ramo.
- d. Explique como um limite sobre o valor dos nós em (c) pode ser derivado da consideração do comprimento do caminho mais curto no mapa e deduza tais limites para esses nós. Lembre-se do custo para chegar a cada folha, bem como do custo para resolvê-la.
- e. Suponha agora que a árvore como dada, com os limites de folha de (d), seja avaliada da esquerda para a direita. Circule os nós “?” que *não* precisam mais ser expandidos, dados os limites da parte (d), e risque aqueles que não precisam ser considerados.
- f. Você pode provar algo genérico sobre quem vai ganhar o jogo em um mapa que é uma árvore?

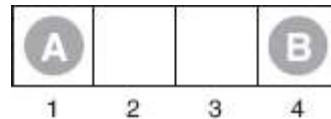
**5.4** Descreva ou implemente descrições de estados, geradores de movimentos, testes de término, funções utilidade e funções de avaliação para um ou mais dos seguintes jogos estocásticos: Monopoly, Scrabble, *bridge* com determinado acordo ou *Texas hold'em poker* (variante do pôquer).

 **5.5** Descreva e implemente um ambiente de jogos de *tempo real*, de *vários participantes*, em que o tempo faça parte do estado do ambiente e no qual os jogadores recebam alocações de tempo fixas.

**5.6** Descreva o quanto a abordagem-padrão para jogos se aplicaria bem a jogos como tênis, bilhar e *croquet*, que ocorrem em um espaço de estados físicos contínuo.

**5.7** Prove a seguinte afirmativa: para toda árvore de jogo, a utilidade obtida por MAX usando decisões de minimax contra um MIN não ótimo nunca será mais baixa que a utilidade obtida no jogo contra um MIN ótimo. Você poderia apresentar uma árvore de jogo em que MAX pudesse atuar ainda melhor usando uma estratégia *ótima* contra um MIN não ótimo?

**5.8** Considere o jogo de dois jogadores descrito na Figura 5.17.



**Figura 5.17** Posição inicial de um jogo simples. O jogador *A* joga primeiro. Os dois jogadores se revezam na movimentação e cada jogador deve mover sua ficha para um espaço adjacente aberto em qualquer sentido. Se o oponente ocupar um espaço adjacente, um jogador pode saltar sobre um oponente até o próximo espaço aberto, se houver (por exemplo, se *A* estiver em 3 e *B* estiver em 2, *A* poderá voltar a 1). O jogo termina quando um jogador chegar à extremidade oposta do tabuleiro. Se o jogador *A* alcançar o espaço 4 primeiro, o valor do jogo para *A* será +1; se o jogador *B* alcançar o espaço 1 primeiro, o valor do jogo para *A* será -1.

- a. Desenhe a árvore de jogo completa, usando as convenções a seguir:
  - Escreva cada estado como  $(s_A, s_B)$ , onde  $s_A$  e  $s_B$  denotam as posições das fichas.
  - Coloque cada estado terminal em um quadrado e escreva o valor de seu jogo em um círculo.
  - Insira os *estados de ciclo* (estados que já aparecem no caminho para a raiz) em quadrados duplos. Tendo em vista que o valor não está claro, identifique cada um com um símbolo “?”

dentro de um círculo.

- b. Agora marque cada nó com seu valor minimax propagado de volta (também em um círculo). Explique como você tratou os valores “?” e por quê.
- c. Explique por que o algoritmo minimax-padrão falharia nessa árvore de jogo e faça um resumo de como você poderia corrigi-lo, baseando-se em sua resposta ao item (b). Seu algoritmo modificado oferece decisões ótimas para todos os jogos com ciclos?
- d. Esse jogo de quatro quadrados pode ser generalizado para  $n$  quadrados, para qualquer  $n > 2$ . Prove que  $A$  vence se  $n$  é par e perde se  $n$  é ímpar.

**5.9** Este problema exercita os conceitos básicos do jogo usando o jogo da velha (zeros e cruzes) como exemplo. Nós definimos  $X_n$  como o número de linhas, colunas ou diagonais com exatamente  $n$   $X$  e não  $O$ . Da mesma forma,  $On$  é o número de linhas, colunas ou diagonais com apenas  $n$   $O$ . A função utilidade atribui +1 a qualquer posição com  $X_3 = 1$  e -1 em qualquer posição com  $O_3 = 1$ . Todas as outras posições terminais têm utilidade 0. Para posições não terminais, utilizamos uma função linear de avaliação definida como  $Eval(s) = 3X_2(s) + X_1(s) - (3O_2(s) + O_1(s))$ .

- a. Existem cerca de quantos jogos da velha possíveis?
- b. Mostre toda a árvore de jogo começando de um tabuleiro vazio até a profundidade 2 (isto é, um  $X$  e um  $O$  na tabuleiro), levando a simetria em conta.
- c. Marque em sua árvore as avaliações de todas as posições com profundidade 2.
- d. Utilizando o algoritmo minimax, marque em sua árvore os valores propagados para as posições à profundidade 1 e 0, e utilize esses valores para escolher a melhor jogada da partida.
- e. Circule os nós à profundidade 2 que *não* seriam avaliados se a poda alfa-beta fosse aplicada, assumindo que os nós são gerados na ordem ótima para a poda alfa-beta.

**5.10** Considere a família de jogos da velha generalizada, definida da seguinte forma. Cada jogo em particular é especificado por um conjunto  $S$  de *quadrados* e uma coleção  $W$  de *posições vencedoras*. Cada posição vencedora é um subconjunto de  $S$ . Por exemplo, no jogo da velha padrão,  $S$  é um conjunto de nove quadrados e  $W$  é uma coleção de oito subconjuntos de  $W$ : as três linhas, as três colunas e as duas diagonais. Em outros aspectos, o jogo é idêntico ao jogo da velha-padrão. A partir de um tabuleiro vazio, os jogadores alternam-se colocando suas marcas em um quadrado vazio. Um jogador que marcar cada quadrado em uma posição vencedora ganhará o jogo. Será empate se todos os quadrados forem marcados e ninguém ganhar o jogo.

- a. Seja  $N = |S|$  o número de quadrados. Dê um limite superior do número de nós em uma árvore de jogo completa para o jogo da velha comum em função de  $N$ .
- b. Dê um limite inferior para o tamanho da árvore de jogo para o pior caso, onde  $W = \{ \}$ .
- c. Proponha uma função de avaliação plausível que possa ser utilizada para qualquer exemplo de jogo da velha comum. A função pode depender de  $S$  e  $W$ .
- d. Assuma que seja possível gerar um novo tabuleiro e verifique se a instrução de máquina  $100N$  é uma posição vencedora, assumindo um processador de 2 gigahertz. Ignore as limitações de memória. Utilizando a sua estimativa em (a), aproximadamente com que tamanho uma árvore de jogo pode ser completamente resolvida por alfa-beta em um segundo de tempo de CPU, em um

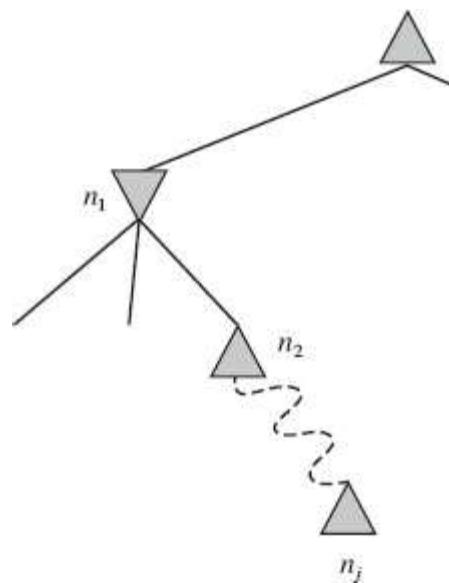
minuto, em uma hora?

### 5.11 Desenvolva um programa de jogo genérico capaz de jogar uma variedade de jogos.

- Implemente geradores de movimentação e funções de avaliação para um ou mais dos seguintes jogos: Kalah, Otelo, damas e xadrez.
- Construa um agente de jogo genérico alfa-beta.
- Compare o efeito de aumentar a profundidade da busca melhorando a ordem dos lances e melhorando a função de avaliação. O quão próximo chegará o seu fator efetivo de ramificação para o caso ideal de ordenação de movimentação perfeita?
- Implemente um algoritmo de busca seletiva, tal como B\* (Berliner, 1979), busca de número de conspiração (McAllester, 1988) ou MGSS \* (Russell e Wefald, 1989) e compare seu desempenho para A\*.

**5.12** Descreva como os algoritmos minimax e alfa-beta se alteram para **jogos de soma diferente de zero** com dois jogadores, em que cada jogador tem sua própria função utilidade e as funções utilidades são conhecidas pelos dois jogadores. Suponha que cada jogador conheça a função utilidade do outro. Se não houver restrições sobre as duas utilidades terminais, é possível qualquer nó ser podado por alfa-beta? O que acontecerá se as funções utilidade do jogador em qualquer estado for a soma de um número entre as constantes  $-k$  e  $k$ , tornando o jogo quase de soma zero?

**5.13** Desenvolva uma prova formal da correção da poda alfa-beta. Para isso, considere a situação mostrada na Figura 5.18. A questão é se devemos podar ou não o nó  $n_j$ , um nó de máximo descendente do nó  $n_1$ . A ideia básica é podá-lo se e somente se for possível mostrar que o valor minimax de  $n_1$  é independente do valor de  $n_j$ .



**Figura 5.18** Situação quando se considera se o nó  $n_j$  deve ser podado.

- A Modalidade  $n_1$  assume o valor mínimo entre seus filhos:  $n_1 = \min(n_2, n_{21}, \dots, n_{2b2})$ . Encontre uma expressão semelhante para  $n_2$  e, consequentemente, uma expressão para  $n_1$  em termos de  $n_j$ .
- Seja  $l_i$  o valor mínimo (ou máximo) dos nós à esquerda do nó  $n_i$  na profundidade  $i$ , cujo valor

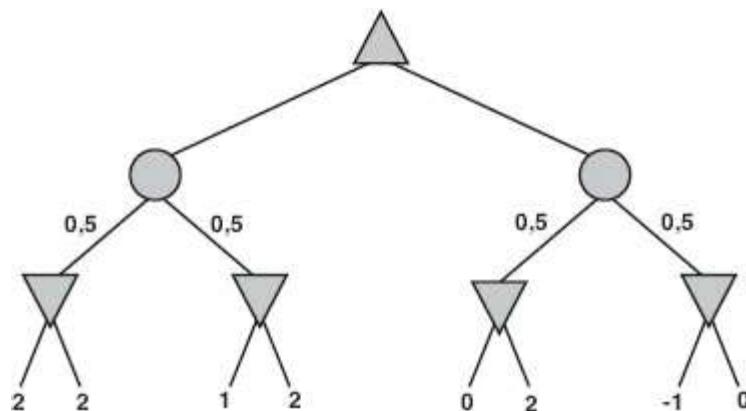
minimax já é conhecido. De modo semelhante, seja  $r_i$  o valor mínimo (ou máximo) dos nós não explorados à direita de  $n_i$  na profundidade  $i$ . Reescreva sua expressão para  $n_1$  em termos dos valores  $l_i$  e  $r_i$ .

- c. Agora, reformule a expressão com a finalidade de mostrar que, para afetar  $n_1$ , o valor de  $n_j$  não deve exceder certo limite derivado dos valores de  $l_i$ .
- d. Repita o processo para o caso em que  $n_j$  é um nó de MIN.

**5.14** Prove que a poda alfa-beta leva o tempo de  $O(2^{m/2})$  com ordenação de lance ótimo, onde  $m$  é a profundidade máxima da árvore de jogo.

**5.15** Vamos supor que você tenha um programa de xadrez capaz de avaliar 10 milhões de nós por segundo. Decida-se por uma representação compacta de um estado de jogo que será armazenada em uma tabela de transposição. Quantas entradas aproximadamente você poderá colocar em uma tabela de 2 gigabytes na memória? Isso será suficiente para os três minutos de busca alocados a um único movimento? Quantos acessos à tabela você poderá efetuar no tempo que levaria para realizar uma única avaliação? Agora, suponha que a tabela de transposição seja maior do que é possível caber na memória. Aproximadamente quantas avaliações você poderia efetuar no tempo necessário para realizar um único acesso a disco com hardware de disco-padrão?

**5.16** Esta questão considera a poda em jogos com nós de acaso. A Figura 5.19 mostra a árvore de jogo completa para um jogo trivial. Suponha que os nós folha estejam para ser avaliados na ordem da esquerda para a direita e que, antes de um nó folha ser avaliado, não sabemos nada sobre o seu valor — a faixa de valores possíveis é  $-\infty$  até  $\infty$ .



**Figura 5.19** A árvore de jogo completa para um jogo trivial com nós de acaso.

- a. Copie a figura, marque o valor de todos os nós internos e indique a melhor jogada na raiz com uma seta.
- b. Dados os valores das primeiras seis folhas, precisamos avaliar a sétima e oitava folhas? Dados os valores das sete primeiras folhas, precisamos avaliar a oitava folha? Explique suas respostas.
- c. Suponha que os valores do nó folha estejam entre  $-2$  e  $2$ , inclusive. Após as duas primeiras folhas serem avaliadas, qual é o intervalo de valor para o nó de acaso da esquerda?
- d. Circule todas as folhas que não precisam ser avaliadas sob o pressuposto em (c).



**5.17** Implemente o algoritmo expectimax e o algoritmo \*-alfa-beta, descrito por Ballard (1983), para podar árvores de jogo com nós de acaso. Experimente-os em um jogo como o gamão e meça a eficiência de poda do algoritmo \*-alfa-beta.

**5.18** Prove que, com uma transformação linear positiva de valores de folha (isto é, a transformação de um valor  $x$  em  $ax + b$  onde  $a > 0$ ), a escolha do movimento permanece inalterada em uma árvore de jogo, mesmo quando existem nós de acaso.

**5.19** Considere o procedimento a seguir para escolher movimentos em jogos com nós de acaso:

- Gere algumas sequências de lançamentos de dados (digamos, 50) descendo até uma profundidade apropriada (digamos, 8).
- Com lançamentos de dados conhecidos, a árvore de jogo se torna determinística. Para cada sequência de lançamentos de dados, resolva a árvore de jogo determinística resultante usando alfa-beta.
- Utilize os resultados para estimar o valor de cada movimento e escolher o melhor.
- Esse procedimento funcionará bem? Por quê (ou por que não)?

**5.20** A seguir, uma árvore “max” consiste apenas em nós max, enquanto uma árvore “expectimax” consiste em um nó max na raiz com camadas alternadas ao acaso e nós max. Nos nós de acaso, todas as probabilidades resultantes são diferentes de zero. O objetivo é *encontrar o valor da raiz* com uma busca de profundidade limitada.

- a. Assumindo que os valores da folha são finitos mas ilimitados, será possível a poda (como em alfa-beta) em uma árvore max? Dê um exemplo ou explique por que não.
- b. A poda é sempre possível em uma árvore expectimax nas mesmas condições? Dê um exemplo ou explique por que não.
- c. Se os valores da folha são não negativos, a poda é sempre possível em uma árvore max? Dê um exemplo ou explique por que não.
- d. Se os valores da folha são não negativos, a poda é sempre possível em uma árvore expectimax? Dê um exemplo ou explique por que não.
- e. Se os valores da folha são restritos na faixa  $[0, 1]$ , a poda é sempre possível em uma árvore max? Dê um exemplo ou explique por que não.
- f. Se os valores da folha são restritos na faixa  $[0, 1]$ , a poda é sempre possível em uma árvore expectimax?
- g. Considere os resultados de um nó de acaso em uma árvore expectimax. Qual das seguintes ordens de avaliação é mais provável de produzir oportunidades de poda?
  - (i) Menor probabilidade em primeiro lugar
  - (ii) Maior probabilidade em primeiro lugar
  - (iii) Não faz qualquer diferença

**5.21** Quais das seguintes alternativas são verdadeiras e quais são falsas? Dê breves explicações.

- a. Em um jogo totalmente observável, de revezamento, de soma zero, entre dois jogadores perfeitamente racionais, não ajuda o primeiro jogador saber que estratégia o segundo jogador está usando, isto é, o lance do segundo jogador é baseado no lance do primeiro jogador.
- b. Em um jogo totalmente observável, de revezamento, de soma zero, entre dois jogadores perfeitamente racionais, não ajuda o primeiro jogador saber que lance o segundo jogador fará, dado o lance do primeiro jogador.
- c. Um agente de gamão perfeitamente racional nunca perde.

**5.22** Considere cuidadosamente a interação de eventos de acaso e a informação parcial em cada um dos jogos do Exercício 5.4.

- a. Para qual deles o modelo expectiminimax padrão é apropriado? Implemente o algoritmo e execute-o em seu agente de jogos, com modificações apropriadas para o ambiente de jogos.
- b. Para quais deles o esquema descrito no Exercício 5.19 seria apropriado?
- c. Descreva como você poderia lidar com o fato de que, em alguns jogos, os jogadores não têm o mesmo conhecimento do estado corrente.

---

<sup>1</sup> Ambientes com muitos agentes são mais bem visualizados como **economias**, em vez de jogos.

<sup>2</sup> É óbvio que isso não pode ser realizado com perfeição; caso contrário, a função de ordenação poderia ser utilizada para se jogar um jogo perfeito!

<sup>3</sup> Às vezes, o estado de crença vai se tornar muito grande para representar apenas uma lista de estados de tabuleiro, mas vamos ignorar essa questão por ora; os Capítulos 7 e 8 sugerem um método para representar compactamente os estados de crença muito grandes.

<sup>4</sup> Blefar — apostar que uma mão é boa, mesmo quando não é — é uma parte essencial da estratégia do pôquer.

<sup>5</sup> Um programa russo, o BESM, pode ter antecipado o programa Bernstein.