CAPÍTULO 4

Busca com informação e exploração

Em que vemos como as informações sobre o espaço de estados podem evitar que os algoritmos se percam na escuridão

O Capítulo 3 mostrou que estratégias de busca sem informações podem encontrar soluções para problemas gerando sistematicamente novos estados e testando-os por comparação com o objetivo. Infelizmente, essas estratégias são ineficientes demais na maioria dos casos. Este capítulo mostra como uma estratégia de busca com informação – que utiliza o conhecimento específico do problema – pode encontrar soluções de modo mais eficiente. A Seção 4.1 descreve versões com informação dos algoritmos do Capítulo 3, e a Seção 4.2 explica como podem ser obtidas as informações específicas necessárias ao problema. As Seções 4.3 e 4.4 estudam algoritmos que executam unicamente a **busca local** no espaço de estados, avaliando e modificando um ou mais estados correntes, em vez de explorar de forma sistemática caminhos a partir de um estado inicial. Esses algoritmos são apropriados para problemas nos quais o custo do caminho é irrelevante, e tudo o que importa é o próprio estado solução. A família de algoritmos de busca local inclui métodos inspirados pela física estatística (**têmpera simulada**) e pela biologia evolucionária (**algoritmos genéticos**). Finalmente, a Seção 4.5 investiga a **busca on-line**, em que o agente se defronta com um espaço de estados completamente desconhecido.

4.1 Estratégias de busca com informação (heurística)

BUSCA COM INFORMAÇÃO

Esta seção mostra como uma estratégia de **busca com informação** – uma estratégia que utiliza o conhecimento específico do problema, além da definição do próprio problema – pode encontrar soluções de forma mais eficiente que uma estratégia sem informação.

BUSCA PELA MELHOR ESCOLHA (BEST-FIRST)

FUNÇÃO DE AVALIAÇÃO

A abordagem geral que examinaremos é chamada **busca pela melhor escolha**. A busca pela melhor escolha é uma especialização do algoritmo geral BUSCA-EM-ÁRVORE ou BUSCA-EM-GRAFO, no qual um nó é selecionado para expansão com base em uma **função de avaliação** $f(n)$. Tradicionalmente, o nó com a avaliação *mais baixa* é selecionado para expansão, porque a avaliação mede a distância até o objetivo. A busca pela melhor escolha pode ser implementada dentro de nossa estrutura geral de busca por meio de uma fila de prioridades, uma estrutura de dados que manterá a borda em ordem ascendente de valores de f.

O nome "busca pela melhor escolha" é consagrado, mas inexato. Afinal, se pudéssemos *realmente* expandir o melhor nó primeiro, isso não seria de modo algum uma busca; seria uma marcha direta para o objetivo. Tudo o que podemos fazer é escolher o nó que *parece* ser o melhor de acordo com a função de avaliação. Se a função de avaliação for exatamente precisa, esse será de fato o melhor nó; na realidade, a função de avaliação às vezes será inacurada e poderá levar a busca a se perder. Não obstante, continuaremos a utilizar o nome "busca pela melhor escolha", porque "busca pelo aparentemente melhor " é um nome um tanto complicado.

FUNÇÃO HEURÍSTICA

Existe uma família inteira de algoritmos BUSCA-PELA-MELHOR-ESCOLHA com funções de avaliação diferentes.[1] Um componente fundamental desses algoritmos é uma **função heurística**,[2] denotada por $h(n)$:

$h(n)$ = custo estimado do caminho mais econômico do nó n até um nó objetivo

Por exemplo, na Romênia, poderíamos estimar o custo do caminho mais econômico desde Arad até Bucareste pela distância em linha reta de Arad a Bucareste.

As funções heurísticas são a forma mais comum de aplicar conhecimento adicional do problema ao algoritmo de busca. Estudaremos as heurísticas com maior profundidade na Seção 4.2. Por enquanto, vamos considerá-las funções arbitrárias específicas do problema, com uma restrição: se n é um nó objetivo, então $h(n) = 0$. O restante desta seção focaliza duas maneiras de usar informações heurísticas para orientar a busca.

Busca gulosa pela melhor escolha

BUSCA GULOSA PELA MELHOR ESCOLHA

A **busca gulosa pela melhor escolha**[3] tenta expandir o nó mais próximo à meta, na suposição de que isso provavelmente levará a uma solução rápida. Desse modo, ela avalia nós usando apenas a função heurística: $f(n) = h(n)$.

DISTÂNCIA EM LINHA RETA

Vamos ver como isso funciona no caso de problemas de localização de rotas na Romênia, usando a heurística de **distância em linha reta**, que chamaremos h_{DLR}. Se o objetivo é Bucareste, precisaremos conhecer as distâncias em linha reta até Bucareste, mostradas na Figura 4.1. Por exemplo, $h_{DLR}(In(Arad)) = 366$. Note que os valores de h_{DLR} não podem ser calculados a partir da própria descrição do problema. Além disso, é necessária uma certa experiência para saber que h_{DLR} está relacionada com distâncias rodoviárias reais e que, portanto, é uma heurística útil.

A Figura 4.2 mostra o progresso de uma busca gulosa pela melhor escolha usando h_{DLR} para encontrar um caminho de Arad até Bucareste. O primeiro nó a ser expandido a partir de Arad será Sibiu, porque está mais próximo de Bucareste que Zerind ou Timisoara. O próximo nó a ser expandido será Fagaras, porque é o mais próximo. Por sua vez, Fagaras gera Bucareste, que é o objetivo. Para esse problema específico, a busca gulosa pela melhor escolha usando h_{DLR} encontra uma solução sem ex-

1. O Exercício 4.3 lhe pede para mostrar que essa família inclui vários algoritmos sem informação familiares.
2. Uma função heurística $h(n)$ toma um nó como entrada, mas depende apenas do *estado* nesse nó.
3. Nossa primeira edição chamava essa busca de **busca gulosa**; outros autores a chamavam **busca pela melhor es-**

pandir nenhum nó que não esteja no caminho da solução; conseqüentemente, seu custo de busca é mínimo. Porém, ela não é ótima: o caminho até Bucareste passando por Sibiu e Fagaras é 32 quilômetros mais longo que o caminho por Rimnicu Vilcea e Pitesti. Isso mostra por que o algoritmo é chamado "guloso" – em cada passo, ele tenta chegar o mais perto possível do objetivo.

Arad	366	**Mehadia**	241
Bucareste	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figura 4.1 Valores de h_{DLR} – distâncias em linha reta até Bucareste.

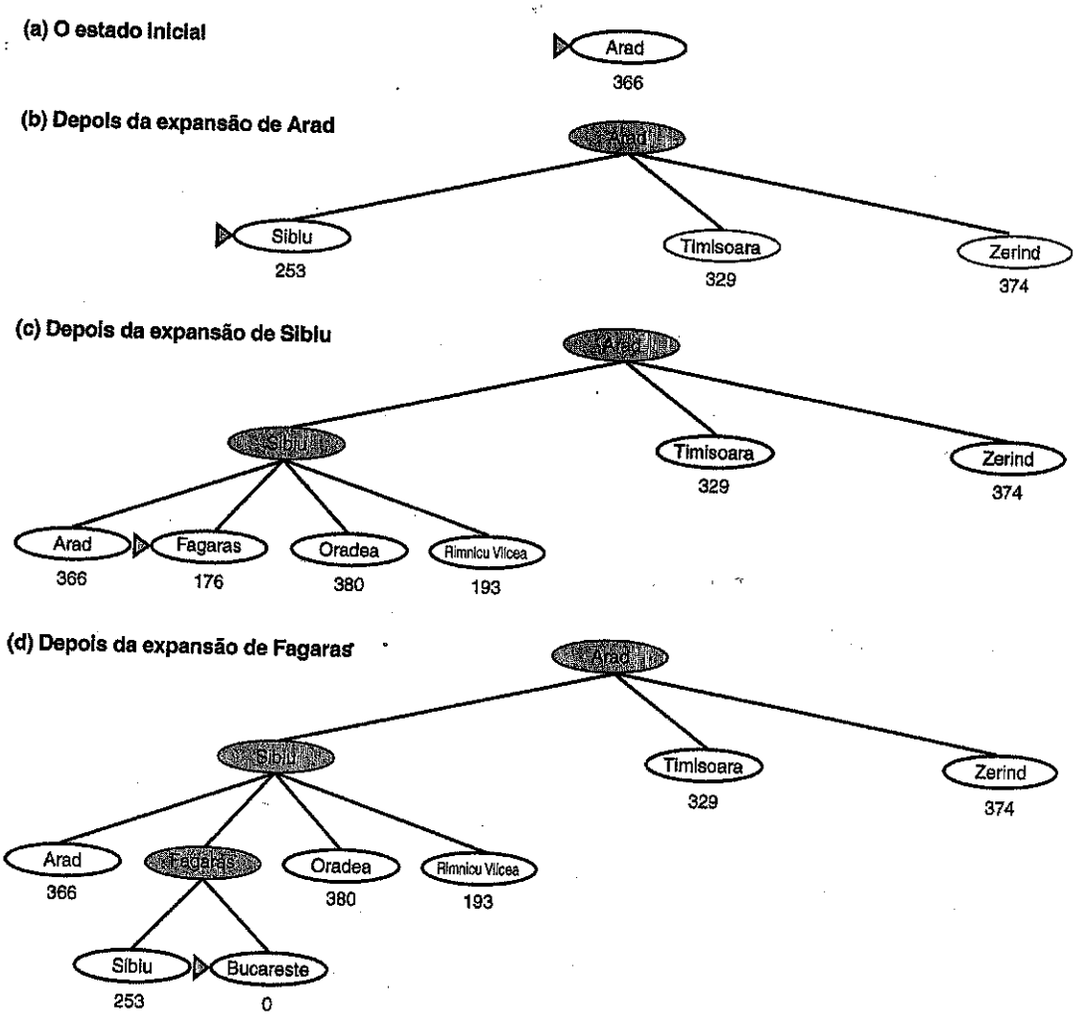


Figura 4.2 Fases de uma busca gulosa pela melhor escolha para Bucareste, usando-se a heurística de distância em linha reta h_{DLR}. Os nós são identificados por seus valores de h.

Minimizar $h(n)$ é uma ação suscetível a falsos inícios. Considere o problema de ir de Iasi até Fagaras. A heurística sugere que Neamt seja expandido primeiro, porque está mais próximo de Fagaras, mas ele é um beco sem saída. A solução é ir primeiro para Vaslui – uma etapa que na realidade é mais distante do objetivo de acordo com a heurística – e depois continuar até Urziceni, Bucareste e Fagaras. Então, nesse caso, a heurística provoca a expansão de nós desnecessários. Além disso, se não tivermos o cuidado de detectar estados repetidos, a solução nunca será encontrada – a busca oscilará entre Neamt e Iasi.

A busca gulosa pela melhor escolha é semelhante à busca em profundidade, pelo fato de preferir seguir um único caminho até o objetivo, mas voltará ao encontrar um beco sem saída. Ela tem os mesmos defeitos da busca em profundidade – não é ótima e é incompleta (porque pode entrar em um caminho infinito e nunca retornar para experimentar outras possibilidades). A complexidade de tempo e espaço do pior caso é $O(b^m)$, onde m é a profundidade máxima do espaço de busca. Porém, com uma boa função heurística, a complexidade pode ter uma redução substancial. A proporção da redução depende do problema específico e da qualidade da heurística.

Busca A*: minimizando o custo total estimado da solução

BUSCA A*

A forma mais amplamente conhecida da busca pela melhor escolha é chamada **busca A***. Ela avalia nós combinando $g(n)$, o custo para alcançar cada nó, e $h(n)$, o custo para ir do nó até o objetivo:

$$f(n) = g(n) + h(n)$$

Tendo em vista que $g(n)$ fornece o custo de caminho desde o nó inicial até o nó n, e que $h(n)$ é o custo estimado do caminho de custo mais baixo desde n até o objetivo, temos:

$f(n)$ = custo estimado da solução de custo mais baixo passando por n

Desse modo, se estivermos tentando encontrar a solução de custo mais baixo, uma opção razoável será experimentar primeiro o nó com o menor valor de $g(n) + h(n)$. Na verdade, essa estratégia é mais que apenas razoável: desde que a função heurística $h(n)$ satisfaça a certas condições, a busca A* será ao mesmo tempo completa e ótima.

HEURÍSTICA ADMISSÍVEL

A análise do caráter ótimo de A* é direta se for usada com BUSCA-EM-ÁRVORE. Nesse caso, A* será ótima se $h(n)$ for uma **heurística admissível** – isto é, desde que $h(n)$ *nunca superestime* o custo para alcançar o objetivo. Heurísticas admissíveis são otimistas por natureza, pois imaginam que o custo da resolução do problema seja menor do que ele é na realidade. Tendo em vista que $g(n)$ é o custo exato para se alcançar n, temos como conseqüência imediata que $f(n)$ nunca irá superestimar o custo verdadeiro de uma solução passando por n.

Um exemplo óbvio de heurística admissível é a distância em linha reta h_{DLR} que usamos para chegar a Bucareste. A distância em linha reta é admissível porque o caminho mais curto entre dois pontos quaisquer é uma linha reta, e assim a linha reta não pode ser uma superestimativa. Na Figura 4.3, mostramos o progresso de uma busca de árvore A* para Bucareste. Os valores de g são calculados a partir dos custos dos passos da Figura 3.2, e os valores de h_{DLR} são dados na Figura 4.1. Em particular, note que Bucareste aparece primeiro na borda do passo (e), mas não está selecionada para expansão, porque seu custo de f (450) é mais alto que o de Pitesti (417). Outra forma de dizer isso é afirmar que *talvez* haja uma solução passando por Pitesti, cujo custo é apenas 417, e assim o algoritmo não admitirá uma solução que custe 450. Desse exemplo, podemos extrair uma prova geral de que *A* usando* BUSCA-EM-ÁRVORE *é ótima se* $h(n)$ *é admissível*. Suponha que um nó objetivo não-ótimo G_2 apareça na borda, e seja C^* o custo da solução ótima. Então, como G_2 não é ótimo e $h(G_2) = 0$ (verdadeiro para qualquer nó objetivo), sabemos que:

$$f(G_2) = g(G_2) + h(G_2) = G(G_2) > C^*.$$

Agora, considere um nó de borda n que está em um caminho de solução ótimo – por exemplo, Pitesti no parágrafo anterior. (Sempre deve haver tal nó se existe uma solução.) Se $h(n)$ não superestimar o custo de completar o caminho de solução, então sabemos que:

$$f(n) = g(n) + h(n) \leq C^*.$$

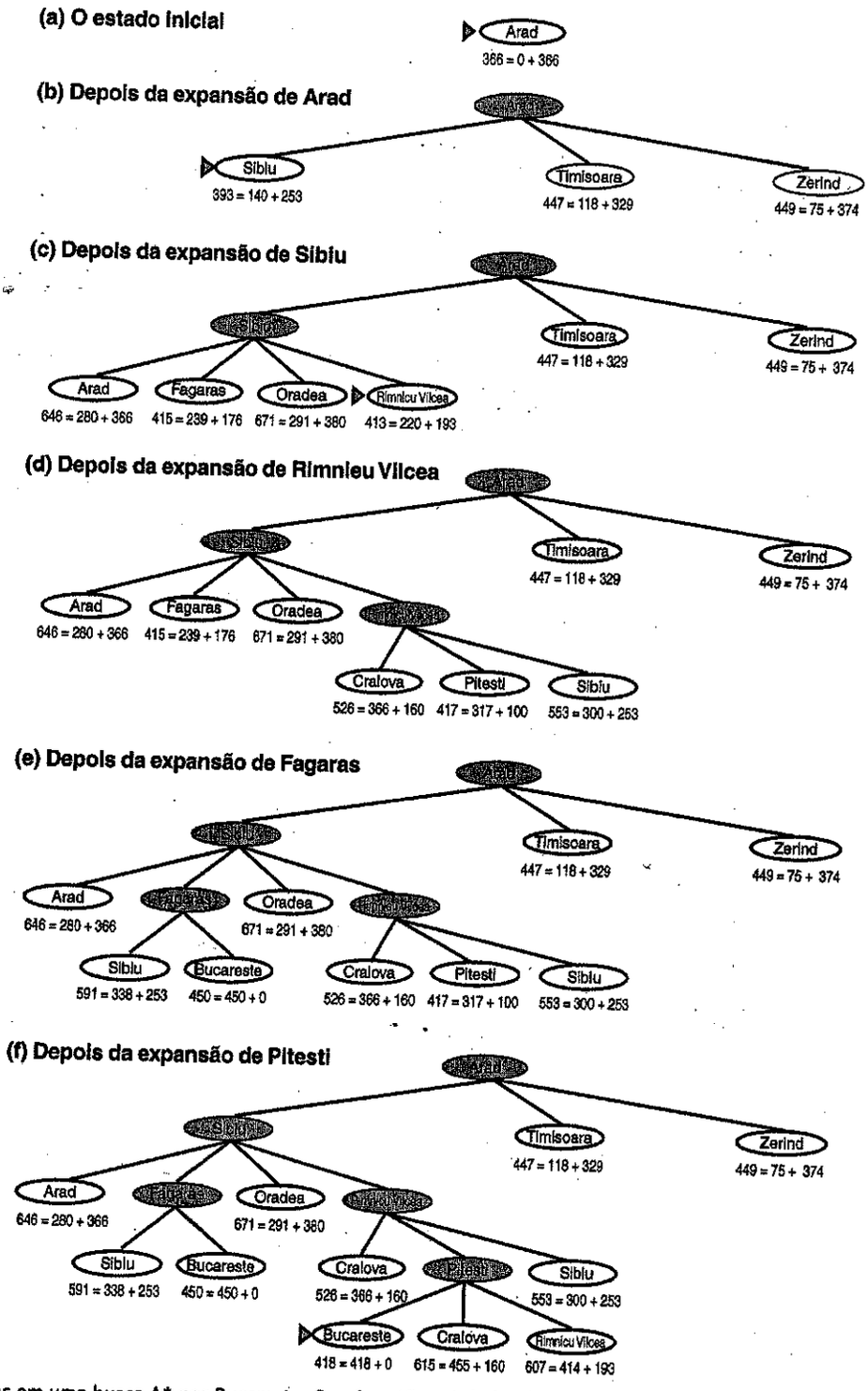


Figura 4.3 Estágios em uma busca A* por Bucareste. Os nós estão rotulados $f = g + h$. Os valores de h são distâncias em linha reta para Bucareste tiradas da Figura 4.1.

Agora, mostramos que $f(n) \leq C^* < f(G_2)$, e assim G_2 não será expandido e A* deve retornar uma solução ótima.

Se usarmos o algoritmo BUSCA-EM-GRAFO da Figura 3.19 em vez de BUSCA-EM-ÁRVORE, essa prova será derrubada. Soluções não-ótimas podem ser retornadas, porque BUSCA-EM-GRAFO pode descartar o caminho ótimo para um estado repetido, se ele não for o primeiro caminho gerado. (Veja o Exercício 4.4.) Existem duas maneiras de corrigir esse problema. A primeira solução é estender BUSCA-EM-GRAFO de tal forma que ele descarte o mais dispendioso entre dois caminhos quaisquer descobertos para o mesmo nó. (Veja a discussão na Seção 3.5.) A anotação extra é confusa, mas garante o caráter ótimo. A segunda solução é assegurar que o caminho ótimo para qualquer estado repetido é sempre o primeiro a ser seguido – como ocorre no caso da busca de custo uniforme. Essa propriedade será válida se impusermos um requisito extra sobre $h(n)$, ou seja, o requisito de **consistência** (também chamada **monotonicidade**). Uma heurística $h(n)$ é consistente se, para todo nó n e todo sucessor n' de n gerado por qualquer ação a, o custo estimado de alcançar o objetivo a partir de n não é maior que o custo do passo de se chegar a n' somado ao custo estimado de alcançar o objetivo a partir de n':

CONSISTÊNCIA
MONOTONICIDADE

$$h(n) \leq c(n, a, n') + h(n').$$

DESIGUALDADE DE TRIÂNGULOS

Essa é uma forma da **desigualdade de triângulos** geral, que estipula que cada lado de um triângulo não pode ser maior que a soma dos outros dois lados. Aqui, o triângulo é formado por n, n' e pelo objetivo mais próximo a n. É bem fácil mostrar (Exercício 4.7) que toda heurística consistente também é admissível. A conseqüência mais importante da consistência é: *A* usando* BUSCA-EM-GRAFO *é ótima se* $h(n)$ *é consistente.*

Embora a consistência seja um requisito mais rígido que a admissibilidade, é necessário muito trabalho para preparar heurísticas admissíveis, mas não consistentes. Todas as heurísticas admissíveis que discutimos neste capítulo também são consistentes. Por exemplo, considere h_{DLR}. Sabemos que a desigualdade de triângulo geral é satisfeita quando cada lado é medido pela distância em linha reta, e que a distância em linha reta entre n e n' não é maior que $c(n, a, n')$. Conseqüentemente, h_{DLR} é uma heurística consistente.

Outra conseqüência importante da consistência é: *se* $h(n)$ *é consistente, então os valores de* $f(n)$ *ao longo de qualquer caminho são não-decrescentes.* A prova decorre diretamente da definição de consistência. Suponha que n' seja um sucessor de n; então $g(n') = g(n) + c(n, a, n')$ para algum a, e temos:

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n).$$

Segue-se que a seqüência de nós expandidos por A* usando BUSCA-EM-GRAFO está em ordem não-decrescente de $f(n)$. Conseqüentemente, o primeiro nó objetivo selecionado para expansão tem de ser uma solução ótima, pois todos os nós posteriores serão pelo menos tão dispendiosos quanto ele.

CONTORNOS

O fato de os custos de f serem não-decrescentes ao longo de qualquer caminho também significa que podemos desenhar **contornos** no espaço de estados, semelhantes aos contornos de um mapa topográfico. A Figura 4.4 mostra um exemplo. No interior de um contorno identificado por 400, todos os nós têm $f(n)$ menor ou igual a 400 e assim por diante. Portanto, considerando que A* expande o nó de borda que tem o menor custo de f, podemos verificar que uma busca A* diverge a partir do nó inicial, acrescentando nós em faixas concêntricas de custo de f crescente.

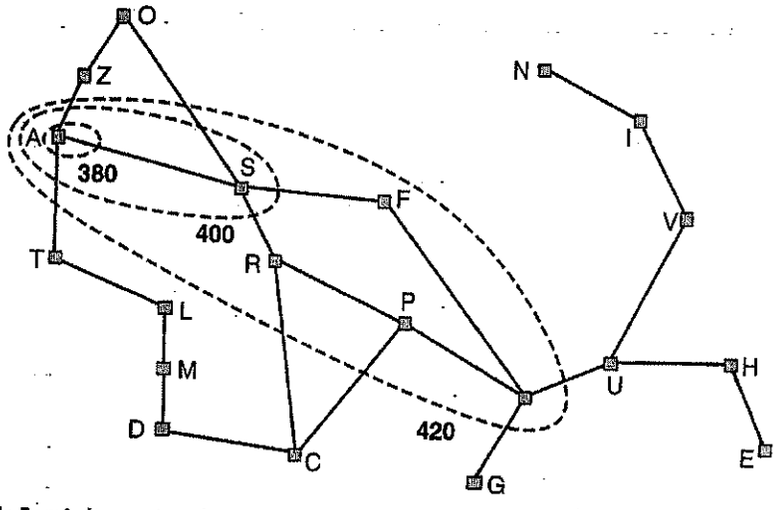


Figura 4.4 Mapa da Romênia, mostrando contornos em $f = 380$, $f = 400$ e $f = 420$, tendo Arad como estado inicial. Nós no interior de um contorno dado têm custos de f menores ou iguais ao valor de contorno.

No caso da busca de custo uniforme (busca A* usando $h(n) = 0$), as faixas serão "circulares" em torno do estado inicial. Com heurísticas mais precisas, as faixas se alongarão em direção ao estado objetivo e se tornarão mais estreitamente concentradas em torno do caminho ótimo. Se C^* for o custo do caminho da solução ótima, podemos dizer que:

- A* expande todos os nós com $f(n) < C^*$.
- A* poderia então expandir alguns nós diretamente no "contorno objetivo" (onde $f(n) = C^*$) antes de selecionar um nó objetivo.

Intuitivamente, é óbvio que a primeira solução encontrada deve ser uma solução ótima, porque nós objetivo em todos os contornos subseqüentes terão custo de f mais alto, e portanto custo de g mais alto (porque todos os nós objetivo têm $h(n) = 0$). Intuitivamente, também é óbvio que a busca A* é completa. À medida que acrescentamos faixas de f crescentes, eventualmente teremos de alcançar uma faixa em que f é igual ao custo do caminho até um estado objetivo.[4]

PODA

Note que A* não expande nenhum nó com $f(n) > C^*$ – por exemplo, Timisoara não é expandido na Figura 4.3, embora seja filho da raiz. Dizemos que a subárvore abaixo de Timisoara foi **podada**; como h_{DLR} é admissível, o algoritmo pode ignorar com segurança essa subárvore, embora ainda garanta o caráter ótimo. O conceito de poda – deixar de considerar certas possibilidades sem ter de examiná-las – é importante para muitas áreas da IA.

ÓTIMAMENTE EFICIENTE

Uma observação final é que, entre algoritmos ótimos desse tipo – algoritmos que estendem os caminhos de busca a partir da raiz –, A* é **otimamente eficiente** para qualquer função heurística dada. Isto é, nenhum outro algoritmo ótimo tem a garantia de expandir um número de nós menor que A* (exceto talvez pelo rompimento de ligações entre nós com $f(n) = C^*$). Isso ocorre porque qualquer algoritmo que *não* expande todos os nós com $f(n) < C^*$ corre o risco de omitir a solução ótima.

O fato de a busca A* ser completa, ótima e otimamente eficiente entre todos esses algoritmos é bastante interessante. Infelizmente, isso não significa que A* seja a resposta para todas as nossas necessidades de busca. Na verdade, para a maioria dos problemas, o número de nós dentro do espaço de busca do contorno de meta ainda é exponencial em relação ao comprimento da solução.

4. A completeza exige que só exista um número finito de nós com custo menor ou igual a C^*, uma condição verdadeira se todos os custos de passos excederem algum ε finito e se b for finito.

Embora a prova do resultado esteja além do escopo deste livro, demonstrou-se que o crescimento exponencial ocorrerá, a menos que o erro na função heurística não cresça com maior rapidez que o logaritmo do custo de caminho real. Em notação matemática, a condição para crescimento subexponencial é:

$$|h(n) - h^*(n)| \leq O(\log h^*(n)),$$

onde $h^*(n)$ é o custo *verdadeiro* para ir de n até o objetivo. Para quase todas as heurísticas em uso na prática, o erro é pelo menos proporcional ao custo do caminho, e o crescimento exponencial resultante no final irá sobrecarregar qualquer computador. Por essa razão, com freqüência é impraticável insistir em descobrir uma solução ótima. É possível usar variantes de A* que encontrem rapidamente soluções não-ótimas ou, às vezes, projetar heurísticas mais precisas, embora não estritamente admissíveis. Em todo caso, o uso de uma boa heurística ainda proporciona enorme economia em comparação com o uso de uma busca sem informação. Na Seção 4.2, examinaremos a questão de projetar boas heurísticas.

Entretanto, o tempo de computação não é a principal desvantagem de A*. Pelo fato de manter todos os nós gerados na memória (como fazem todos os algoritmos BUSCA-EM-GRAFO), em geral A* esgota o espaço bem antes de esgotar o tempo. Por essa razão, A* não é prático para muitos problemas de grande escala. Algoritmos desenvolvidos recentemente superaram o problema do espaço sem sacrificar o caráter ótimo ou a completeza, a um custo pequeno no tempo de execução. Esses algoritmos serão discutidos em seguida.

Busca heurística limitada pela memória

O caminho mais simples para reduzir requisitos de memória de A* é adaptar a idéia de aprofundamento iterativo ao contexto de busca heurística, resultando no algoritmo A* de aprofundamento iterativo (AIA*).* A principal diferença entre AIA* e o aprofundamento iterativo padrão é que o corte usado é o custo de $f\,(g + h)$ em vez da profundidade; a cada iteração, o valor de corte é o menor custo de f de qualquer nó que tenha excedido o corte na iteração anterior. AIA* é prático para muitos problemas com custo de passo unitário e evita a sobrecarga substancial associada à manutenção de uma fila ordenada de nós. Infelizmente, ele apresenta as mesmas dificuldades com os custos de valor real que encontramos na versão iterativa da busca de custo uniforme descrita no Exercício 3.11. Esta seção examinará brevemente dois algoritmos mais recentes limitados pela memória, chamados BRPM e LMA*.

BUSCA RECURSIVA PELO MELHOR

A **busca recursiva pelo melhor** (BRPM) é um algoritmo recursivo simples que tenta imitar a operação da busca pela melhor escolha-padrão, mas utiliza apenas espaço linear. O algoritmo é mostrado na Figura 4.5. Sua estrutura é semelhante à de uma busca recursiva em profundidade; porém, em vez de continuar a descer indefinidamente pelo caminho atual, ela controla o valor de f do melhor caminho alternativo disponível a partir de qualquer ancestral do nó atual. Se o nó atual exceder esse limite, a recursão retornará ao caminho alternativo. À medida que a recursão se desenrola, BRPM repõe o valor de f de cada nó ao longo do caminho com o melhor valor de f de seus filhos. Desse modo, BRPM guarda na memória o valor de f da melhor folha na subárvore esquecida e pode portanto decidir se vale a pena voltar a expandir a subárvore em algum momento posterior. A Figura 4.6 mostra como BRPM alcança Bucareste.

Nota do revisor técnico: IDA – *iterative-deepening A**.

```
função BUSCA-RECURSIVA-PELO-MELHOR(problema) retorna uma solução ou falha
  BRPM(problema, CRIAR-NÓ(ESTADO-INICIAL[problema]), ∞)

função BRPM(problema, nó, f_limite) retorna uma solução ou falha e um novo limite de custo de f
  se TESTAR-OBJETIVO[problema](estado) então retornar nó
  sucessores ← EXPANDIR(nó, problema)
  se sucessores está vazio então retornar falha, ∞
  para cada s em sucessores faça
      f[s] ← max(g(s) + h(s), f[nó])
  repita
      melhor ← o nó de mais baixo valor de f em sucessores
      se f[melhor] > f_limite então retornar falha, f[melhor]
      alternativo ← o segundo mais baixo valor de f entre sucessores
      resultado, f[melhor] ← BRPM(problema, melhor, min(f_limite, alternativa))
      se resultado ≠ falha então retornar resultado
```

Figura 4.5 O algoritmo para busca recursiva pelo melhor.

BRPM é um pouco mais eficiente que AIA*, mas ainda sofre de excessiva geração repetida de nós. No exemplo da Figura 4.6, primeiro BRPM segue o caminho via Rimnicu Vilcea, depois "muda de idéia" e tenta Fagaras, e então volta a mudar de idéia. Essas mudanças de idéia ocorrem porque, toda vez que o melhor caminho atual é estendido, há uma boa chance de que seu valor de f aumente – h em geral é menos otimista para nós mais próximos à meta. Quando isso acontece, em particular em grandes espaços de busca, o segundo melhor caminho pode se tornar o melhor caminho, e assim a busca tem de regressar para segui-lo. Cada mudança de idéia corresponde a uma iteração de AIA*, e pode exigir muitas reexpansões de nós esquecidos, a fim de recriar o melhor caminho e estendê-lo a mais um nó.

Como A*, BRPM é um algoritmo ótimo se a função heurística $h(n)$ é admissível. Sua complexidade de espaço é $O(bd)$, mas sua complexidade de tempo é bastante difícil de caracterizar: ela depende tanto da exatidão da função heurística quanto da freqüência com que o melhor caminho muda à medida que os nós são expandidos. Tanto AIA* quanto BRPM estão sujeitos ao aumento potencialmente exponencial de complexidade associado à busca em grafos (veja a Seção 3.5), porque não podem verificar a presença de estados repetidos além dos que estão no caminho atual. Desse modo, talvez eles tenham de explorar o mesmo estado muitas vezes.

AIA* e BRPM sofrem por utilizar *muito pouca* memória. Entre iterações, AIA* retém apenas um único número: o limite de custo de f atual. BRPM retém mais informações na memória, mas só utiliza a quantidade de memória correspondente a $O(bd)$: mesmo que houvesse mais memória disponível, BRPM não teria como utilizá-la.

Portanto, parece sensato usar toda a memória disponível. Dois algoritmos que fazem isso são **LMA*** (A* limitado pela memória) e **LMSA*** (LMA* simplificado). Descreveremos o LMSA*, que é – digamos – mais simples. O LMSA* prossegue exatamente como A*, expandindo a melhor folha até completar a memória. Nesse ponto, ele não poderá adicionar um novo nó à árvore de busca sem descartar um nó antigo. O LMSA* sempre descarta o *pior* nó de folha – o nó com o mais alto valor de f. Como BRPM, LMSA* copia então o valor do nó esquecido em seu pai. Desse modo, o ancestral de uma subárvore esquecida conhece a qualidade do melhor caminho nessa subárvore. Com essas informações, o LMSA* só regenera a subárvore quando *todos os outros caminhos* se mostram piores que o caminho esquecido. Outro modo de dizer isso é afirmar que, se todos os descendentes de um nó n forem esquecidos, não saberemos que caminho seguir a partir de n, mas ainda teremos uma idéia do quanto vale a pena ir para qualquer lugar a partir de n.

LMA*

LMSA*

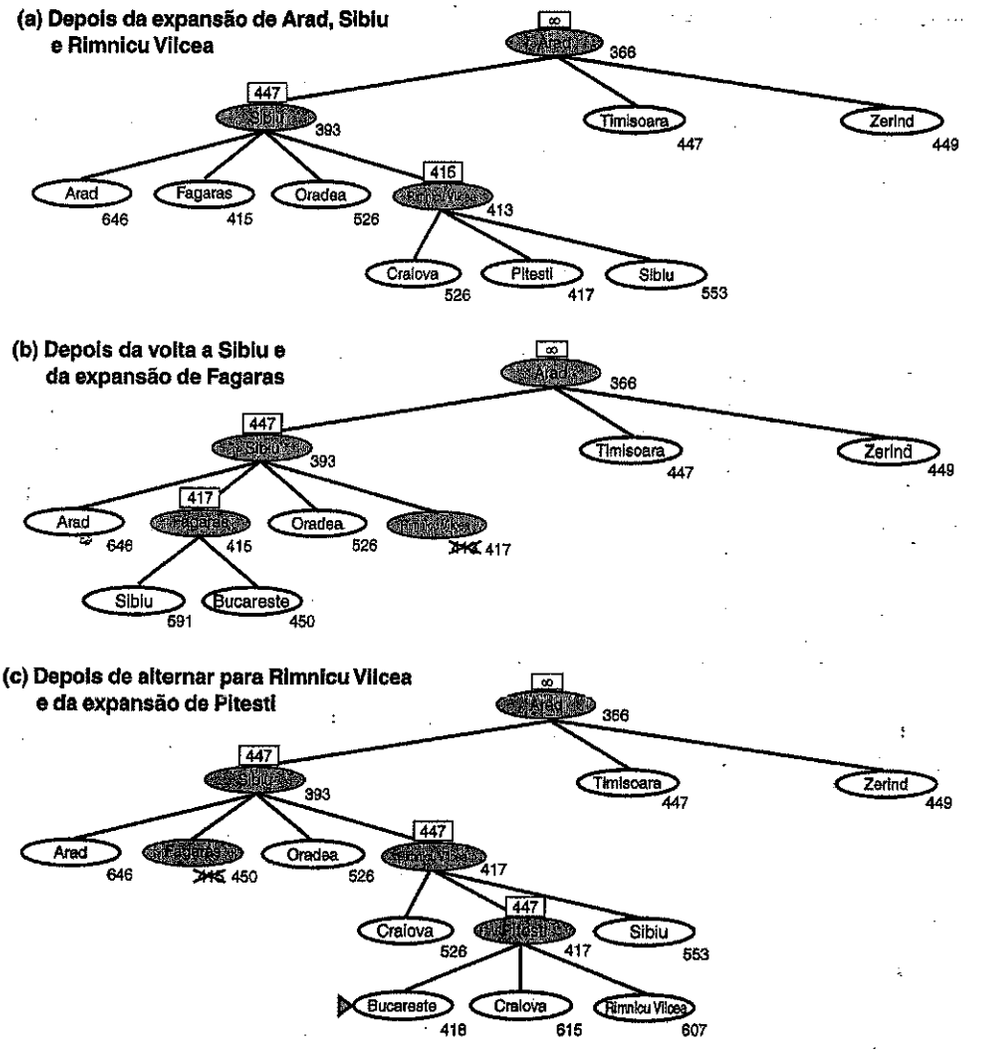


Figura 4.6 Fases em uma busca de BRPM para descobrir a rota mais curta até Bucareste. O valor do limite de *f* para cada chamada recursiva é mostrado sobre cada nó atual. (a) O caminho via Rimnicu Vilcea é seguido até a melhor folha atual (Pitesti) ter um valor pior que o melhor caminho alternativo (Fagaras). (b) A recursão retorna e o valor da melhor folha da subárvore esquecida (417) é copiado em Rimnicu Vilcea; em seguida, Fagaras é expandido, revelando um valor de folha melhor, 450. (c) A recursão retorna e o melhor valor de folha da subárvore esquecida (450) é copiado em Fagaras; depois, Rimnicu Vilcea é expandido. Dessa vez, como o melhor caminho alternativo (passando por Timisoara) custa pelo menos 447, a expansão continua até Bucareste.

O algoritmo completo é muito complicado para ser reproduzir aqui,[5] mas existe uma sutileza que vale a pena mencionar. Dissemos que o LMSA* expande a melhor folha e elimina a pior folha. E se *todos* os nós de folhas tiverem o mesmo valor de *f*? O algoritmo poderia então selecionar o mesmo nó para eliminação e expansão. O LMSA* resolve esse problema expandindo a melhor folha *mais nova* e eliminando a pior folha *mais antiga*. Essas duas folhas só poderão representar o mesmo nó se houver apenas uma folha; nesse caso, a árvore de busca atual deve ser um caminho único desde a raiz até a folha que preenche toda a memória. Se a folha não for um nó objetivo, *mesmo que ela esteja em um caminho de solução ótima*, essa solução não será alcançável com a memória disponível. Assim, o nó poderá ser descartado exatamente como se não tivesse nenhum sucessor.

O LMSA* será completo se existir qualquer solução alcançável – isto é, se *d*, a profundidade do nó objetivo mais raso, for menor que o tamanho da memória (expresso em número de nós). Ele será óti-

5. Um esboço rudimentar foi apresentado na primeira edição deste livro.

mo se qualquer solução ótima for alcançável; caso contrário, ele retornará a melhor solução alcançável. Em termos práticos, o LMSA* poderia bem ser o melhor algoritmo de uso geral para encontrar soluções ótimas, particularmente quando o espaço de estados for um grafo, os custos dos passos não forem uniformes e a geração de nós for dispendiosa em comparação com a sobrecarga adicional de manter as listas abertas e fechadas.

No entanto, em problemas muito difíceis, com freqüência o LMSA* será forçado a alternar continuamente entre um conjunto de caminhos de soluções candidatas, do qual apenas um pequeno subconjunto pode caber na memória. (Isso lembra o problema de **thrashing** em sistemas de paginação de disco.) Portanto, o tempo extra necessário para geração repetida dos mesmos nós significa que problemas que seriam praticamente solúveis por A* com uma memória ilimitada tornam-se intratáveis para o LMSA*. Quer dizer, *limitações de memória podem tornar um problema intratável do ponto de vista do tempo de computação*. Embora não exista nenhuma teoria para explicar relação inversamente proporcional entre tempo e memória, este parece ser um problema inevitável. A única saída é abandonar o requisito de otimização.

THRASHING

Aprendizagem para fazer buscas melhores

Apresentamos várias estratégias fixas – busca em extensão, busca gulosa pela melhor escolha e assim por diante – que foram projetadas por cientistas da computação. Um agente poderia *aprender* como fazer uma busca melhor? A resposta é sim, e o método se baseia em um importante conceito chamado **espaço de meta-estados**. Cada estado em um espaço de meta-estados captura o estado interno (computacional) de um programa que está fazendo busca em um **espaço de estados de nível objeto** como a Romênia. Por exemplo, o estado interno do algoritmo A* consiste na árvore de busca atual. Cada ação no espaço de meta-estados é um passo de computação que altera o estado interno; por exemplo, cada passo de computação em A* expande um nó de folha e adiciona seus sucessores à árvore. Desse modo, a Figura 4.3, que mostra uma seqüência de árvores de busca cada vez maiores, pode ser vista como a representação de um caminho no espaço de meta-estados, onde cada estado no caminho é uma árvore de busca de nível de objeto.

ESPAÇO DE META-ESTADOS

ESPAÇO DE ESTADOS DE NÍVEL OBJETO

Agora, o caminho na Figura 4.3 tem cinco passos, incluindo um passo – a expansão de Fagaras – que não é especialmente útil. Para problemas mais difíceis, haverá muitos desses passos equivocados, e um algoritmo de **meta-aprendizagem** poderá aprender a partir dessas experiências a evitar a exploração de subárvores pouco promissoras. As técnicas usadas para esse tipo de aprendizagem são descritas no Capítulo 21. O objetivo da aprendizagem é minimizar o **custo total** da resolução de problemas, equilibrando o gasto computacional e o custo do caminho.

META-APRENDIZAGEM

4.2 Funções heurísticas

Nesta seção, estudaremos as heurísticas para o quebra-cabeça de 8 peças, a fim de elucidar a natureza das heurísticas em geral.

O quebra-cabeça de 8 peças foi um dos mais antigos problemas de busca heurística. Como mencionamos na Seção 3.2, o objetivo do quebra-cabeça é deslizar os blocos no sentido horizontal ou vertical para o espaço vazio, até a configuração coincidir com a configuração objetivo (Figura 4.7).

O custo da solução média para uma instância do quebra-cabeça de 8 peças gerada ao acaso é cerca de 22 passos. O fator de ramificação é aproximadamente igual a 3. (Quando o espaço vazio está no meio, há quatro movimentos possíveis; quando ele está em um canto, são possíveis dois movimentos e, quando ele está em uma borda, há três movimentos.) Isso significa que uma busca exaustiva para a

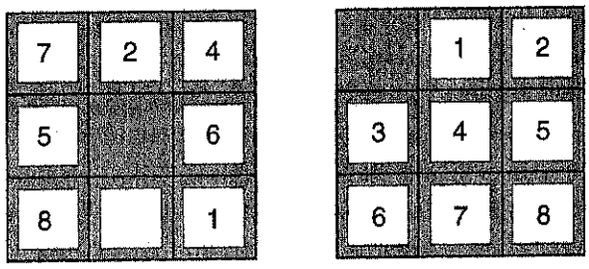


Figura 4.7 Uma instância típica do quebra-cabeça de 8 peças. A solução tem 26 passos.

profundidade 22 examinaria cerca de $3^{22} \approx 3{,}1 \times 10^{10}$ estados. Controlando os estados repetidos, poderíamos reduzir esse valor aproximadamente 170.000 vezes, porque existem apenas $9!/2 = 181.440$ estados distintos acessíveis. (Veja o Exercício 3.4.) Esse é um número gerenciável, mas o número correspondente para o quebra-cabeça de 15 peças é próximo de 10^{13}, e assim é necessário encontrar uma boa função heurística. Se quisermos descobrir as soluções mais curtas usando A*, precisaremos de uma função heurística que nunca superestime o número de passos até o objetivo. Existe uma longa história de tais heurísticas para o quebra-cabeça de 15 peças; aqui estão duas candidatas comumente utilizadas:

- h_1 = O número de blocos em posições erradas. Para a Figura 4.7, todos os oito blocos estão fora de posição, e assim o estado inicial teria $h_1 = 8$. h_1 é uma heurística admissível, porque é claro que qualquer bloco que esteja fora do lugar deve ser movido pelo menos uma vez.
- h_2 = A soma das distâncias dos blocos de suas posições objetivo. Como os blocos não podem se mover em diagonal, a distância que levaremos em conta é a soma das distâncias horizontal e vertical. Às vezes, essa soma é chamada **distância de quadras urbanas** ou **distância Manhattan**. h_2 também é admissível, porque o resultado de qualquer movimento é deslocar um bloco para uma posição mais próxima do objetivo. Os blocos de 1 a 8 no estado inicial fornecem uma distância Manhattan igual a:

DISTÂNCIA MANHATTAN

$$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$$

Como seria de esperar, nenhum desses valores superestima o custo da solução verdadeira, que é 26.

O efeito da exatidão da heurística sobre o desempenho

FATOR DE RAMIFICAÇÃO EFETIVA

Uma maneira de caracterizar a qualidade de uma heurística é o **fator de ramificação efetiva**, b^*. Se o número total de nós gerados por A* para um determinado problema é N, e se a profundidade da solução é d, então b^* é o fator de ramificação que uma árvore uniforme de profundidade d precisaria ter para conter $N + 1$ nós. Desse modo:

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

Por exemplo, se A* encontrar uma solução na profundidade 5 usando 52 nós, o fator de ramificação efetiva será 1,92. O fator de ramificação efetiva pode variar em diversas instâncias do problema mas, em geral, ele é relativamente constante para problemas suficientemente difíceis. Portanto, medidas experimentais de b^* em um pequeno conjunto de problemas podem fornecer uma boa orientação sobre a utilidade geral da heurística. Uma heurística bem projetada teria um valor de b^* próximo de 1, permitindo a resolução de problemas bastante extensos.

Para testar as funções heurísticas h_1 e h_2, geramos 1.200 problemas aleatórios com tamanhos de soluções variando de 2 a 24 (100 para cada número par) e resolvemos esses problemas com a busca por aprofundamento iterativo e a busca em árvore A* utilizando h_1 e h_2. A Figura 4.8 fornece o número médio de nós expandidos por cada estratégia e o fator de ramificação efetivo. Os resultados sugerem que h_2 é melhor que h_1 e muito melhor que a busca por aprofundamento iterativo. Em nossas soluções com o comprimento 14, A* com h_2 é 30.000 vezes mais eficiente que a busca por aprofundamento iterativo sem informação.

	Custo da busca			Fator de ramificação efetiva		
d	BAI	A*(h_1)	A*(h_2)	BAI	A*(h_1)	A*(h_2)
2	10	6	6	2,45	1,79	1,79
4	112	13	12	2,87	1,48	1,45
6	680	20	18	2,73	1,34	1,30
8	6384	39	25	2,80	1,33	1,24
10	47127	93	39	2,79	1,38	1,22
12	3644035	227	73	2,78	1,42	1,24
14	–	539	113	–	1,44	1,23
16	–	1301	211	–	1,45	1,25
18	–	3056	363	–	1,46	1,26
20	–	7276	676	–	1,47	1,27
22	–	18094	1219	–	1,48	1,28
24	–	39135	1641	–	1,48	1,26

Figura 4.8 Comparação entre os custos da busca e entre os fatores de ramificação efetiva para os algoritmos BUSCA-POR-APROFUNDAMENTO-ITERATIVO e A* com h_1, h_2. A média dos dados é calculada sobre 100 instâncias do quebra-cabeça de 8 peças, para diversos comprimentos de solução.

DOMINÂNCIA

Poderíamos perguntar se h_2 é *sempre* melhor que h_1. A resposta é sim. É fácil verificar a partir das definições das duas heurísticas que, para qualquer nó n, $h_2(n) \geq h_1(n)$. Desse modo, dizemos que aquele h_2 **domina** h_1. A dominância se traduz diretamente em eficiência: A* usando h_2 nunca expandirá mais nós que A* usando h_1 (exceto talvez por alguns nós com $f(n) = C^*$). O argumento é simples. Lembre-se da observação da página 97 de que todo nó com $f(n) < C^*$ seguramente será expandido. Isso é o mesmo que dizer que todo nó com $h(n) < C^* - g(n)$ sem dúvida será expandido. Porém, como h_2 é pelo menos tão grande quanto h_1 para todos os nós, todo nó seguramente expandido pela busca a* com h_2 também será seguramente expandido com h_1, e então h_1 também poderia fazer outros nós serem expandidos. Conseqüentemente, sempre é melhor usar uma função heurística com valores mais altos, desde que ela não superestime o custo e que o tempo de computação para a heurística não seja muito grande.

Criação de funções heurísticas admissíveis

Vimos que tanto h_1 (blocos mal posicionados) quanto h_2 (distância Manhattan) são heurísticas bastante boas para o quebra-cabeça de 8 peças, e que h_2 é melhor. Como alguém pôde chegar a criar h_2? Um computador pode criar tal heurística mecanicamente?

h_1 e h_2 são estimativas do comprimento de caminho restante para o quebra-cabeça de 8 peças, mas também são comprimentos de caminho perfeitamente precisos para versões *simplificadas* do quebra-cabeça. Se as regras do quebra-cabeça fossem alteradas de forma que um bloco pudesse se deslo-

car para qualquer lugar, e não apenas para o quadrado vazio adjacente, então h_1 daria o número exato de passos da solução mais curta. De modo semelhante, se um bloco pudesse se mover um quadrado em qualquer direção, mesmo sobre um quadrado ocupado, então h_2 forneceria o número exato de passos na solução mais curta. Um problema com menos restrições sobre as ações é chamado **problema relaxado**. *O custo de uma solução ótima para um problema relaxado é uma heurística admissível para o problema original.* A heurística é admissível porque a solução ótima no problema original também é, por definição, uma solução no problema relaxado e, portanto, tem de ser pelo menos tão dispendiosa quanto a solução ótima no problema relaxado. Como a heurística derivada é um custo exato para o problema relaxado, ela deve obedecer à desigualdade de triângulos e, assim, deve ser **consistente** (veja a página 101).

PROBLEMA RELAXADO

Se uma definição de problema for enunciada em uma linguagem formal, será possível construir automaticamente problemas relaxados.[6] Por exemplo, se as ações do quebra-cabeça de 8 peças forem descritas como:

> Um bloco pode se mover do quadrado A para o quadrado B se
>
> A é horizontal ou verticalmente adjacente a B **e** B é vazio

Podemos gerar três problemas relaxados removendo uma ou ambas as condições:

(a) Um bloco pode se mover do quadrado A para o quadrado B se A é adjacente a B.
(b) Um bloco pode se mover do quadrado A para o quadrado B se B está vazio.
(c) Um bloco pode se mover do quadrado A para o quadrado B.

A partir de (a), podemos derivar h_2 (distância Manhattan). O raciocínio é que h_2 seria a pontuação adequada se movêssemos um bloco por vez até seu destino. A heurística derivada de (b) é discutida no Exercício 4.9. A partir de (c), podemos derivar h_1 (blocos mal posicionados), porque esse valor seria a pontuação adequada se os blocos pudessem se mover até o destino pretendido em um único passo. Note que é crucial que os problemas relaxados gerados por essa técnica possam ser resolvidos essencialmente *sem busca*, porque as regras relaxadas permitem que o problema seja decomposto em oito subproblemas independentes. Se o problema relaxado for difícil de resolver, será dispendioso obter os valores da heurística correspondente.[7]

Um programa chamado ABSOLVER pode gerar heurísticas automaticamente a partir de definições de problemas, usando o método de "problema relaxado" e várias outras técnicas (Prieditis, 1993). O ABSOLVER gerou uma nova heurística para o quebra-cabeça de 8 peças melhor que qualquer heurística preexistente e descobriu a primeira heurística útil para o famoso quebra-cabeça do cubo de Rubik.

Um problema com a geração de novas funções heurísticas é que freqüentemente não se consegue obter uma heurística "claramente melhor". Se uma coleção de heurísticas admissíveis $h_1 \ldots h_m$ estiver disponível para um problema e nenhuma delas dominar qualquer das outras, qual devemos escolher? Na verdade, não precisamos fazer nenhuma escolha. Podemos ter o melhor de todos os mundos, definindo:

$$h(n) = \text{máx}\{h_1(n), \ldots, h_m(n)\}.$$

6. Nos Capítulos 8 e 11, descreveremos linguagens formais apropriadas para essa tarefa; com descrições formais que podem ser manipuladas, a construção de problemas relaxados pode ser automatizada. No momento, usaremos o idioma natural.

7. Observe que uma heurística perfeita pode ser obtida simplesmente permitindo-se que h execute uma busca completa em extensão "às escondidas". Desse modo, existe uma relação inversamente proporcional entre exatidão e tempo de computação para funções heurísticas.

Essa heurística composta utilizará a função que for mais exata sobre o nó em questão. Como às heurísticas componentes são admissíveis, h é admissível; também é fácil provar que h é consistente. Além disso, h domina todas as suas heurísticas componentes.

SUBPROBLEMA

Heurísticas admissíveis também podem ser derivadas do custo da solução de um **subproblema** de um problema específico. Por exemplo, a Figura 4.9 mostra um subproblema da instância do quebra-cabeça de 8 peças da Figura 4.7. O subproblema envolve colocar os blocos 1, 2, 3, 4 em suas posições corretas. É claro que o custo da solução ótima desse subproblema é um limite inferior sobre o custo do problema completo. Na verdade, ela é substancialmente mais precisa que a distância Manhattan em alguns casos.

BANCOS DE DADOS DE PADRÕES

A idéia que rege os **bancos de dados de padrões** é armazenar esses custos de soluções exatas para toda instância possível de subproblema – em nosso exemplo, toda configuração possível dos quatro blocos e do espaço vazio. (Note que as posições dos outros quatro blocos são irrelevantes para os propósitos de resolução do subproblema, mas movimentos desses blocos são importantes para o custo.) Portanto, calculamos uma heurística admissível h_{DB} para cada estado completo encontrado durante uma busca, simplesmente examinando a configuração de subproblema correspondente no banco de dados. O próprio banco de dados é construído por meio de uma busca inversa a partir do estado objetivo, registrando-se o custo de cada novo padrão encontrado; o custo dessa busca é amortizado ao longo das várias instâncias subseqüentes do problema.

A escolha de 1-2-3-4 é arbitrária; também poderíamos construir bancos de dados para 5-6-7-8, e para 2-4-6-8 e assim por diante. Cada banco de dados gera uma heurística admissível, e essas heurísticas podem ser combinadas, conforme explicamos antes, tomando-se o valor máximo. Uma heurística combinada desse tipo é muito mais precisa que a distância Manhattan; o número de nós gerados quando se resolvem quebra-cabeças de 15 peças ao acaso pode ser reduzido até 1.000 vezes.

Poderíamos perguntar se as heurísticas obtidas a partir do banco de dados 1-2-3-4 e do banco de dados 5-6-7-8 poderiam ser *somadas*, considerando-se que os dois subproblemas parecem não se sobrepor. Isso ainda daria uma heurística admissível? A resposta é não, porque as soluções do subproblema 1-2-3-4 e do subproblema 5-6-7-8 para um determinado estado quase certamente irão compartilhar alguns movimentos – é improvável que 1-2-3-4 possa ser colocado no lugar sem tocar 5-6-7-8 e *vice-versa*. Porém, e se não levarmos em conta esses movimentos? Isto é, não registramos o custo total da resolução do subproblema 1-2-3-4, mas apenas do número de movimentos que envolvem 1-2-3-4. Então, é fácil ver que a soma dos dois custos ainda é um limite inferior sobre o custo da resolução do problema inteiro. Essa é a idéia por trás dos **bancos de dados de padrões disjuntos**. Usando tais bancos de dados, é possível resolver quebra-cabeças de 15 peças ao acaso em alguns milissegundos – o número de nós gerados é reduzido 10.000 vezes em comparação com o uso da distância Manhattan. Para quebra-cabeças de 24 peças, é possível obter uma aceleração de aproximadamente um milhão de vezes.

BANCOS DE DADOS DE PADRÕES DISJUNTOS

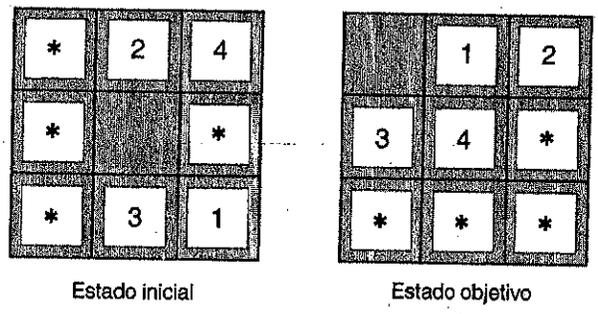


Figura 4.9 Um subproblema do quebra-cabeça de 8 peças dado na Figura 4.7. A tarefa é colocar os blocos 1, 2, 3 e 4 em suas posições corretas, sem se preocupar com que acontecerá aos outros blocos.

Os bancos de dados de padrões disjuntos funcionam no caso de quebra-cabeças de blocos deslizantes, porque o problema pode ser dividido de tal modo que cada movimento afete apenas um subproblema – porque apenas um arquivo é movido de cada vez. Para um problema como o cubo de Rubik, esse tipo de subdivisão não pode ser feito, porque cada movimento afeta 8 ou 9 dos 26 cubos. No momento, não está claro como definir bancos de dados disjuntos para tais problemas.

Heurísticas de aprendizagem a partir da experiência

Uma função heurística $h(n)$ deve estimar o custo de uma solução que começa a partir do estado no nó n. Como um agente poderia construir tal função? Uma solução foi dada na seção anterior – ou seja, criar problemas relaxados para os quais uma solução ótima possa ser encontrada com facilidade. Outra solução é aprender a partir de experiência. Nesse caso, "experiência" significa, por exemplo, resolver uma grande quantidade de quebra-cabeças de 8 peças. Cada solução ótima para um problema de quebra-cabeça de 8 peças fornece exemplos a partir dos quais $h(n)$ pode ser aprendido. Cada exemplo consiste em um estado do caminho de solução e no custo real da solução a partir desse ponto. Com base nesses exemplos, um algoritmo de **aprendizado indutivo** pode ser usado para construir uma função $h(n)$ que pode (com sorte) prognosticar custos de soluções para outros estados que surgem durante a busca. As técnicas para fazer exatamente isso usando redes neurais, árvores de decisão e outros métodos são demonstradas no Capítulo 18. (Os métodos de aprendizagem por reforço descritos no Capítulo 21 também são aplicáveis.)

CARACTERÍSTICAS

Os métodos de aprendizagem indutiva funcionam melhor quando são alimentados com **características** de um estado que são relevantes para sua avaliação, e não apenas com a descrição bruta do estado. Por exemplo, a característica "número de blocos mal posicionados" poderia ser útil na previsão da distância real entre um estado e o objetivo. Vamos chamar essa característica $x_1(n)$. Poderíamos tomar 100 configurações de quebra-cabeça de 8 peças geradas ao acaso e obter estatísticas sobre os custos reais de sua solução. Talvez descobríssemos que, quando $x_1(n)$ é 5, o custo da solução média é aproximadamente 14 e assim por diante. Considerando-se esses dados, o valor de x_1 pode ser usado para prever $h(n)$. É claro que podemos utilizar várias características. Uma segunda característica $x_2(n)$ poderia ser o "número de pares de blocos adjacentes que também são adjacentes no estado objetivo". Como $x_1(n)$ e $x_2(n)$ devem ser combinados para prognosticar $h(n)$? Uma abordagem comum é usar uma combinação linear:

$$h(n) = c_1x_1(n) + c_2x_2(n)$$

As constantes c_1 e c_2 são ajustadas para proporcionar a melhor adaptação aos dados reais em custos de soluções. Pressupõem que c_1 deve ser positiva e c_2 deve ser negativa.

4.3 Algoritmos de busca local e problemas de otimização

Os algoritmos de busca que vimos até agora foram projetados para explorar sistematicamente espaços de busca. Esse caráter sistemático é alcançado mantendo-se um ou mais caminhos na memória e registrando-se as alternativas que foram exploradas em cada ponto ao longo do caminho e quais delas não foram exploradas. Quando um objetivo é encontrado, o *caminho* até esse objetivo também constitui uma *solução* para o problema.

No entanto, em muitos problemas, o caminho até o objetivo é irrelevante. Por exemplo, no problema de 8 rainhas (veja a página 69), o que importa é a configuração final das rainhas, e não a ordem em

que elas são acrescentadas. Essa classe de problemas inclui muitas aplicações importantes, como projeto de circuitos integrados, layout de instalações industriais, escalonamento de jornadas de trabalho, programação automática, otimização de rede de telecomunicações, roteamento de veículos e gerenciamento de carteiras.

BUSCA LOCAL
ESTADO CORRENTE

Se o caminho até o objetivo não importa, podemos considerar uma classe diferente de algoritmos, aqueles que não se preocupam de forma alguma com os caminhos. Os algoritmos de **busca local** operam usando um único **estado corrente** (em vez de vários caminhos) e em geral se movem apenas para os vizinhos desse estado. Normalmente, os caminhos seguidos pela busca não são guardados. Embora os algoritmos de busca local não sejam sistemáticos, eles têm duas vantagens: (1) usam pouquíssima memória – quase sempre um valor constante; e (2) freqüentemente podem encontrar soluções razoáveis em grandes ou infinitos espaços (contínuos) de estados para os quais os algoritmos sistemáticos são inadequados.

PROBLEMAS DE OTIMIZAÇÃO
FUNÇÃO OBJETIVO

Além de encontrar objetivos, os algoritmos de busca local são úteis para resolver **problemas de otimização** puros, nos quais o objetivo é encontrar o melhor estado de acordo com uma **função objetivo**. Muitos problemas de otimização não se adaptam ao modelo de busca "padrão" introduzido no Capítulo 3. Por exemplo, a natureza fornece uma função objetivo – aptidão para a reprodução – que a evolução de Darwin poderia estar tentando otimizar, mas não existe nenhum "teste de objetivo" e nenhum "custo de caminho" para esse problema.

TOPOLOGIA DE ESPAÇO DE ESTADOS
MÍNIMO GLOBAL
MÁXIMO GLOBAL

Para entender a busca local, descobriremos que é muito útil considerar a **topologia de espaço de estados** (como na Figura 4.10). Uma topologia tem ao mesmo tempo "posição" (definida pelo estado) e "elevação" (definida pelo valor da função de custo da heurística ou da função objetivo). Se a elevação corresponder ao custo, o objetivo será encontrar o vale mais baixo – um **mínimo global**; se a elevação corresponder a uma função objetivo, então o objetivo será encontrar o pico mais alto – um **máximo global**. (Você pode fazer a conversão de um para o outro apenas inserindo um sinal de subtração.) Os algoritmos de busca local exploram essa topologia. Um algoritmo de busca local **completo** sempre encontra umo objetivo, caso ele exista; um algoritmo **ótimo** sempre encontra um mínimo/máximo global.

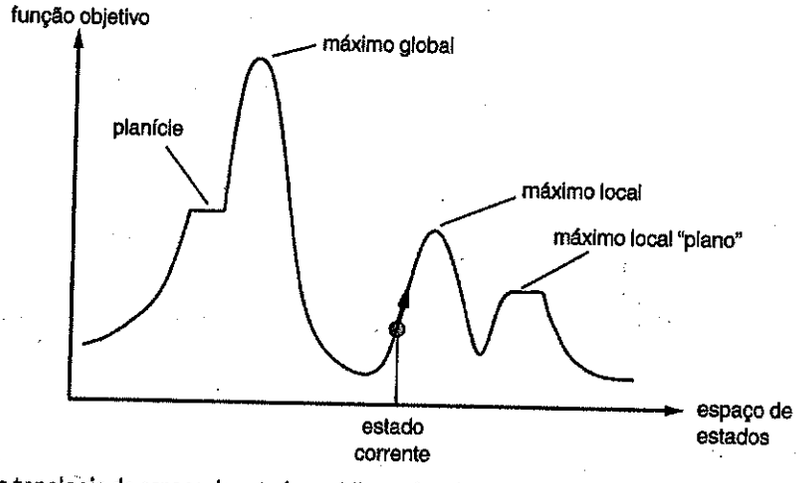


Figura 4.10 Uma topologia de espaço de estados unidimensional, no qual a elevação corresponde à função objetivo. O objetivo é encontrar o máximo global. A busca de subida de encosta modifica o estado corrente para tentar melhorá-lo, como mostra a seta. As diversas características topográficas são definidas no texto.

Busca de subida de encosta

SUBIDA DE ENCOSTA

O algoritmo de busca de **subida de encosta** é mostrado na Figura 4.11. Ele é simplesmente um laço repetitivo que se move de forma contínua no sentido do valor crescente – isto é, encosta acima. O al-

```
função SUBIDA-DE-ENCOSTA(problema) retorna um estado que é um máximo local
  entradas: problema, um problema
  variáveis locais: corrente, um nó
                    vizinho, um nó

  corrente ← CRIAR-NÓ(ESTADO-INICIAL[problema])
  repita
    vizinho ← um sucessor de corrente com valor mais alto
    se VALOR[vizinho] ≤ VALOR[corrente] então retornar ESTADO[corrente]
    corrente ← vizinho
```

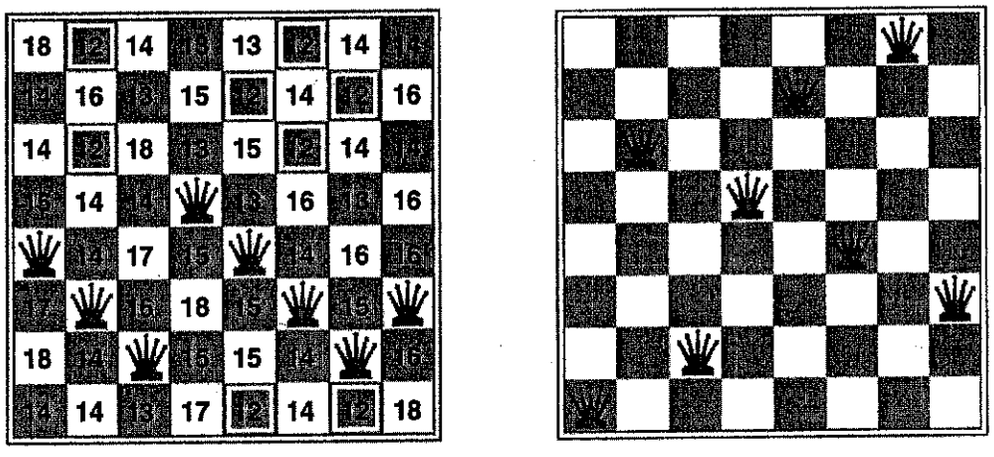
Figura 4.11 O algoritmo de busca de subida de encosta (versão **pela trilha mais íngreme**) é a técnica de busca local mais básica. Em cada passo, o nó corrente é substituído pelo melhor vizinho; nessa versão, esse é o vizinho com o VALOR mais alto; porém, se fosse usada uma estimativa de custo de heurística h, encontraríamos o vizinho com o h mais baixo.

goritmo termina quando alcança um "pico" em que nenhum vizinho tem valor mais alto. O algoritmo não mantém uma árvore de busca, e assim a estrutura de dados do nó atual só precisa registrar o estado e o valor de sua função objetivo. A busca de subida de encosta não examina antecipadamente valores de estados além dos vizinhos imediatos do estado corrente. É como tentar alcançar o cume do Monte Everest em meio a um nevoeiro denso durante uma crise de amnésia.

Para ilustrar a subida de encosta, usaremos o **problema de 8 rainhas** introduzido na página 69. Em geral, os algoritmos de busca local utilizam uma **formulação de estados completos**, onde cada estado tem 8 rainhas no tabuleiro, uma por coluna. A função sucessora retorna todos os estados possíveis gerados pela movimentação de uma única rainha para outro quadrado na mesma coluna (de forma que cada estado tenha $8 \times 7 = 56$ sucessores). A função de custo da heurística h é o número de pares de rainhas que estão atacando umas às outras, seja direta ou indiretamente. O mínimo global dessa função é zero, que só ocorre em soluções perfeitas. A Figura 4.12(a) mostra um estado com $h = 17$. A figura também mostra os valores de todos os seus sucessores, na qual os melhores sucessores têm $h = 12$. Os algoritmos de subida de encosta normalmente fazem uma escolha aleatória entre o conjunto de melhores sucessores, caso exista mais de um.

BUSCA GULOSA LOCAL

A subida de encosta às vezes é chamada **busca gulosa local**, porque captura um bom estado vizinho sem decidir com antecedência para onde irá em seguida. Embora a gula seja considerada um dos



(a) (b)

Figura 4.12 (a) Um estado de 8 rainhas com estimativa de custo de heurística $h = 17$, mostrando o valor de h para cada sucessor possível obtido pela movimentação de uma rainha dentro de sua coluna. Os melhores movimentos estão marcados. (b) Um mínimo local no espaço de estados de 8 rainhas; o estado tem $h = 1$, mas todo sucessor tem um custo mais alto.

sete pecados capitais, na verdade os algoritmos gulosos freqüentemente funcionam muito bem. Muitas vezes, a subida de encosta progride com grande rapidez em direção a uma solução, porque normalmente é bem fácil melhorar um estado ruim. Por exemplo, a partir do estado da Figura 4.12(a), bastam cinco passos para alcançar o estado da Figura 4.12(b), que tem $h = 1$ e está muito próxima de uma solução. Infelizmente, a subida de encosta com freqüência fica paralisada, pelas seguintes razões:

- **Máximos locais:** Um máximo local é um pico mais alto que cada um de seus estados vizinhos, embora seja mais baixo que o máximo global. Os algoritmos de subida de encosta que alcançarem a vizinhança de um máximo local serão deslocados para cima em direção ao pico, mas depois ficarão paralisados, sem ter para onde ir. A Figura 4.10 ilustra esquematicamente o problema. Em termos mais concretos, o estado da Figura 4.12(b) é de fato um máximo local (isto é, um mínimo local para o custo h); todo movimento de uma única rainha piora a situação.
- **Picos:** Um pico é mostrado na Figura 4.13. Os picos resultam em uma seqüência de máximos locais que torna muito difícil a navegação para algoritmos gulosos.
- **Platôs:** Um platô é uma área da topologia de espaço de estados em que a função de avaliação é plana. Ele pode ser um máximo local plano, a partir do qual não existe nenhuma saída encosta acima, ou uma **planície**, a partir da qual é possível progredir. (Veja a Figura 4.10.) Uma busca de subida de encosta talvez seja incapaz de encontrar a saída do platô.

PLANÍCIE

Em cada caso, o algoritmo alcança um ponto em que não há nenhum progresso. A partir de um estado do problema de 8 rainhas gerado aleatoriamente, a subida de encosta pela trilha mais íngreme ficará paralisada 86% do tempo, resolvendo apenas 14% de instâncias de problemas. Ela funciona com rapidez, demorando apenas 4 passos em média quando tem sucesso e 3 quando fica paralisada – nada mal para um espaço de estados com $8^8 \approx 17$ milhões de estados.

O algoritmo da Figura 4.11 pára ao alcançar um platô em que o melhor sucessor tem o mesmo valor do estado corrente. É possível que não seja boa idéia prosseguir – permitir um **movimento lateral**, na esperança de que o platô seja na realidade uma planície, como mostra a Figura 4.10? Normalmente a resposta é sim, mas devemos ter cuidado. Se sempre permitirmos movimento laterais quando não houver nenhum movimento encosta acima, ocorrerá uma repetição infinita sempre que o algoritmo alcançar um máximo local plano que não seja uma planície. Uma solução comum é impor um limite sobre o número de movimentos laterais consecutivos permitidos. Por exemplo, poderíamos permitir

MOVIMENTO LATERAL

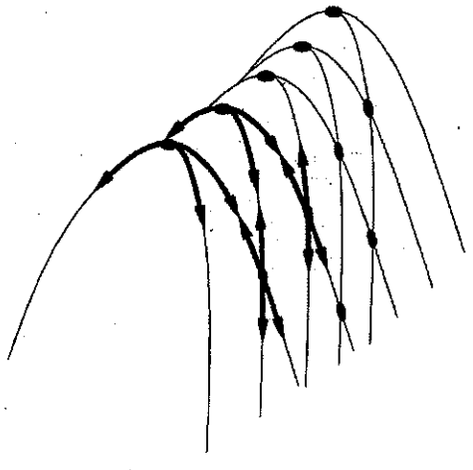


Figura 4.13 Ilustração do motivo pelo qual os picos causam dificuldades na subida de encosta. A malha de estados (círculos escuros) está sobreposta em um pico que se eleva da esquerda para a direita, criando uma seqüência de máximos locais que não estão diretamente conectados uns aos outros. A partir de cada máximo local, todas as ações disponíveis apontam encosta abaixo.

até, digamos, 100 movimentos laterais consecutivos no problema de 8 rainhas. Isso aumenta a porcentagem de instâncias de problemas resolvidos por subida de encosta de 14% para 94%. O sucesso tem um custo: o algoritmo demora em média 21 passos aproximadamente para cada instância bem-sucedida, e 64 passos para cada falha.

SUBIDA DE ENCOSTA ESTOCÁSTICA

SUBIDA DE ENCOSTA PELA PRIMEIRA ESCOLHA

Foram criadas muitas variantes de subida de encosta. A **subida de encosta estocástica** escolhe ao acaso entre os movimentos encosta acima; a probabilidade de seleção pode variar com a declividade do movimento encosta acima. Em geral, isso converge mais lentamente que a subida mais íngreme, mas em algumas topologias de estados encontra soluções melhores. A **subida de encosta pela primeira escolha** implementa a subida de encosta estocástica gerando sucessores ao acaso, até ser gerado um sucessor melhor que o estado corrente. Essa é uma boa estratégia quando um estado tem muitos sucessores (por exemplo, milhares). O Exercício 4.16 pede que você investigue.

SUBIDA DE ENCOSTA COM REINÍCIO ALEATÓRIO

Os algoritmos de subida de encosta descritos até agora são incompletos – com freqüência, eles deixam de encontrar um objetivo que existe, porque ficam paralisados em máximos locais. A **subida de encosta com reinício aleatório** adota o conhecido adágio: "Se não tiver sucesso na primeira vez, continue tentando." Ela conduz uma série de buscas de subida de encosta a partir de estados iniciais gerados ao acaso,[8] parando ao encontrar um objetivo. Ela é completa, com a probabilidade se aproximando de 1, pela simples razão de que irá eventualmente gerar um estado objetivo como estado inicial. Se cada busca de subida de encosta tiver uma probabilidade de sucesso p, o número esperado de reinícios exigidos será $1/p$. Para instâncias de 8 rainhas sem a permissão de movimentos laterais, $p \approx 0{,}14$; assim, precisamos de aproximadamente 7 iterações para encontrar umo objetivo (6 falhas e 1 sucesso). O número esperado de passos é o custo de uma iteração bem-sucedida somado a $(1-p)/p$ vezes o custo da falha ou cerca de 22 passos. Quando permitimos movimentos laterais, são necessárias $1/0{,}94 \approx 1{,}06$ iteração em média e $(1 \times 21) + (0{,}06/0{,}94) \times 64 \approx 25$ passos. Então, no caso de 8 rainhas, a subida de encosta com reinício aleatório é de fato muito eficiente. Mesmo para três milhões de rainhas, a abordagem pode encontrar soluções em menos de um minuto.[9]

O sucesso da subida de encosta depende muito da forma da topologia do espaço de estados: se houver poucos máximos locais e platôs, a subida de encosta com reinício aleatório encontrará uma boa solução com muita rapidez. Por outro lado, muitos problemas reais têm uma topologia mais parecida com uma família de ouriços em um piso plano, com ouriços em miniatura vivendo na ponta de cada espinho de um ouriço, *ad infinitum*. Em geral, os problemas NP-difíceis têm um número exponencial de máximos locais em que ficam paralisados. Apesar disso, um máximo local razoavelmente bom pode ser encontrado com freqüência depois de um pequeno número de reinícios.

Busca de têmpera simulada

Um algoritmo de subida de encosta que *nunca* faz movimentos "encosta abaixo" em direção a estados com valor mais baixo (ou de custo mais alto) sem dúvida é incompleto, porque pode ficar paralisado em um máximo local. Em contraste, um percurso puramente aleatório – isto é, a movimentação até um sucessor escolhido uniformemente ao acaso a partir do conjunto de sucessores – é completo, mas extremamente ineficiente. Então, parece razoável tentar combinar a subida de encosta com um percurso aleatório que resulte de algum modo em eficiência e completeza. A **têmpera simulada** é esse

TÊMPERA SIMULADA

8. Gerar um estado *aleatório* a partir de um espaço de estados implicitamente especificado pode ser um problema difícil por si só.

9. Luby *et al.* (1993) provam que é melhor, em alguns casos, reiniciar um algoritmo de busca aleatória depois de um período de tempo fixo e específico, e isso pode ser *muito* mais eficiente que deixar cada busca continuar indefinidamente. Proibir ou limitar o número de movimentos laterais é um exemplo dessa estratégia.

algoritmo. Em metalurgia, **a têmpera** é o processo usado para temperar ou endurecer metais e vidro aquecendo-os a alta temperatura e depois esfriando-os gradualmente, permitindo assim que o material seja misturado em um estado cristalino de baixa energia. Para entender a têmpera simulada, vamos mudar nosso ponto de vista de subida de encosta para **descida gradiente** (isto é, minimização do custo) e imaginar a tarefa de colocar uma bola de pingue-pongue na fenda mais profunda em uma superfície acidentada. Se simplesmente deixarmos a bola rolar, ela acabará em um mínimo local. Se agitarmos a superfície, poderemos fazer a bola quicar para fora do mínimo local. O artifício é agitar com força suficiente para fazer a bola sair dos mínimos locais, mas não o bastante para desalojá-la do mínimo global. A solução de têmpera simulada é começar a agitar com força (isto é, em alta temperatura) e depois reduzir gradualmente a intensidade da agitação (ou seja, baixar a temperatura).

DESCIDA GRADIENTE

O laço de repetição mais interno do algoritmo de têmpera simulada (Figura 4.14) é muito semelhante à subida de encosta. Porém, em vez de escolher o *melhor* movimento, ele escolhe um movimento *aleatório*. Se o movimento melhorar a situação, ele sempre será aceito. Caso contrário, o algoritmo aceitará o movimento com alguma probabilidade menor que 1. A probabilidade diminui exponencialmente com a "má qualidade" do movimento – o valor ΔE segundo o qual a avaliação piora. A probabilidade também diminui à medida que a "temperatura" T se reduz: movimentos "ruins" têm maior probabilidade de serem permitidos no início, quando a temperatura é alta, e depois se tornam mais improváveis conforme T diminui. Pode-se provar que, se o *escalonamento* diminuir T com lentidão suficiente, o algoritmo encontrará um valor ótimo global com probabilidade próxima de 1.

A têmpera simulada foi usada inicialmente de forma extensiva para resolver problemas de layout de VLSI no começo dos anos 80. Ela foi amplamente aplicada ao escalonamento industrial e a outras tarefas de otimização em grande escala. No Exercício 4.16, você será convidado a comparar seu desempenho ao da subida de encosta com reinício aleatório no quebra-cabeça de n rainhas.

Busca em feixe local

BUSCA EM FEIXE LOCAL

A manutenção de apenas um nó na memória pode parecer uma reação extrema ao problema de limitação de memória. O algoritmo de **busca em feixe local**[10] mantém o controle de k estados, em vez de somente um. Ela começa com k estados gerados aleatoriamente. Em cada passo, são gerados todos os sucessores de todos os k estados. Se qualquer um deles for um objetivo, o algoritmo irá parar. Caso contrário, ele selecionará os k melhores sucessores a partir da lista completa e repetirá a ação.

À primeira vista, uma busca em feixe local com k estados talvez pareça não ser nada mais que a execução de k reinícios aleatórios em paralelo, e não em seqüência. De fato, os dois algoritmos são bastante diferentes. Em uma busca com reinício aleatório, cada processo de busca funciona de forma independente dos outros. *Em uma busca em feixe local, são repassadas informações úteis entre os k processos paralelos da busca*. Por exemplo, se um estado gerar vários sucessores bons e todos os outros $k-1$ estados gerarem sucessores ruins, o efeito será como se o primeiro estado dissesse aos outros: "Venha para cá, aqui está melhor!" O algoritmo logo abandonará as buscas infrutíferas e deslocará seus recursos para o processo em que estiver sendo realizado maior progresso.

Em sua forma mais simples, a busca em feixe local pode se ressentir de uma falta de diversidade entre os k estados – eles podem ficar rapidamente concentrados em uma pequena região do espaço de estados, tornando a busca pouca mais que uma versão dispendiosa de subida de encosta. Uma variante chamada **busca em feixe estocástica**, análoga à subida de encosta estocástica, ajuda a atenuar esse problema. Em vez de escolher o melhor k a partir do conjunto de sucessores candidatos, a busca

BUSCA EM FEIXE ESTOCÁSTICA

10. A busca em feixe local é uma adaptação da **busca em feixe (beam search)**, um algoritmo baseado em caminhos.

```
função TÊMPERA-SIMULADA(problema, escalonamento) retorna um estado solução
  entradas: problema, um problema
            escalonamento, um mapeamento de tempo para "temperatura"
  variáveis locais: corrente, um nó
                    próximo, um nó
                    T, uma "temperatura" que controla a probabilidade de passos descendentes

  corrente ← CRIAR-NÓ(ESTADO-INICIAL[problema])
  para t ← 1 até ∞ faça
      T ← escalonamento[t]
      se T = 0 então retornar corrente
      próximo ← um sucessor de corrente selecionado ao acaso
      ΔE ← VALOR[próximo] – VALOR[corrente]
      se ΔE > 0 então corrente ← próximo
      senão corrente ← próximo somente com probabilidade e^(ΔE/T)
```

Figura 4.14 O algoritmo de têmpera simulada, uma versão de subida de encosta estocástica, onde alguns movimentos encosta abaixo são permitidos. Movimentos encosta abaixo são prontamente aceitos no início do escalonamento da têmpera, e depois com menor freqüência no decorrer do tempo. A entrada *escalonamento* define o valor de T como uma função do tempo.

em feixe estocástica escolhe k sucessores ao acaso, com a probabilidade de escolher um determinado sucessor que seja uma função crescente de seu valor. A busca em feixe estocástica guarda alguma semelhança com o processo de seleção natural, pelo qual os "sucessores" (descendência) de um "estado" (organismo) ocupam a próxima geração de acordo com seu "valor" (adaptação ou fitness).

Algoritmos genéticos

ALGORITMO GENÉTICO

Um **algoritmo genético** (ou AG) é uma variante de busca em feixe estocástica, na qual os estados sucessores são gerados pela combinação de *dois* estados pais, em vez de serem gerados pela modificação de um único estado. A analogia em relação à seleção natural é a mesma que se dá na busca em feixe estocástica, exceto pelo fato de agora estarmos lidando com a reprodução sexuada, e não com a reprodução assexuada.

POPULAÇÃO
INDIVÍDUO

Como ocorre com a busca em feixe, os AGs começam com um conjunto de k estados gerados aleatoriamente, chamado **população**. Cada estado, ou **indivíduo**, é representado como um cadeia sobre um alfabeto finito – muito freqüentemente um cadeia de valores 0 e 1. Por exemplo, um estado de 8 rainhas deve especificar as posições das 8 rainhas, cada uma em coluna de 8 quadrados, e portanto exige $8 \times \log_2 8 = 24$ bits. Como outra alternativa, o estado poderia ser representado como 8 dígitos, cada um no intervalo de 1 a 8. (Veremos mais adiante que as duas codificações têm comportamento diferente.) A Figura 4.15(a) mostra uma população de quatro cadeias de 8 dígitos que representam estados de 8 rainhas.

FUNÇÃO DE FITNESS

A produção da próxima geração de estados é mostrada na Figura 4.15(b)-(e). Em (b), cada estado é avaliado pela função de avaliação ou (na terminologia do AG) pela **função de fitness**. Uma função de fitness deve retornar valores mais altos para estados melhores; assim, para o problema de 8 rainhas, usamos o número de pares de rainhas *não-atacantes*, que têm o valor 28 para uma solução. Os valores dos quatro estados são 24, 23, 20 e 11. Nessa variante específica do algoritmo genético, a probabilidade de um indivíduo de ser escolhido para reprodução é diretamente proporcional à sua pontuação de fitness, e as porcentagens são mostradas ao lado das pontuações brutas.

Em (c), dois pares escolhidos aleatoriamente são selecionados para reprodução, de acordo com as probabilidades mostradas em (b). Note que um indivíduo é selecionado duas vezes, e um indivíduo

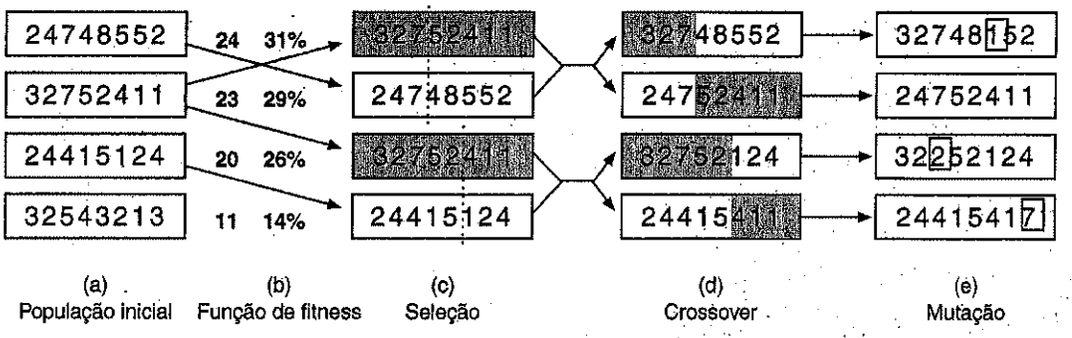


Figura 4.15 O algoritmo genético. A população inicial em (a) é classificada pela função de fitness em (b), resultando em pares de correspondência em (c). Eles produzem descendentes em (d), sujeitos à mutação em (e).

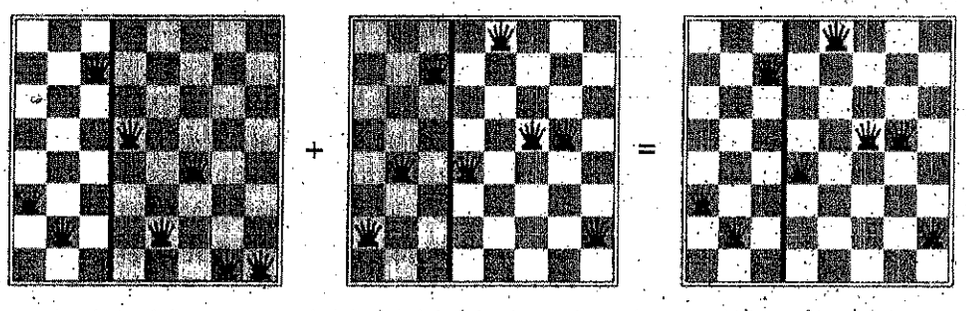


Figura 4.16 Os estados de 8 rainhas correspondentes aos dois primeiros pais na Figura 4.15(c) e à primeira descendência da Figura 4.15(d). As colunas sombreadas foram perdidas na etapa de crossover, e as colunas não-sombreadas foram preservadas.

não é selecionado de modo algum.[11] Para cada par a ser cruzado é escolhido ao acaso um ponto de

CROSSOVER **crossover** dentre as posições na cadeia. Na Figura 4.15, os pontos de crossover estão depois do terceiro dígito no primeiro par e depois do quinto dígito no segundo par.[12]

Em (d), os próprios descendentes são criados por crossover das cadeias pais no ponto de crossover. Por exemplo, o primeiro filho do primeiro par recebe os três primeiros dígitos do primeiro pai e os dígitos restantes do segundo pai, enquanto o segundo filho recebe os três primeiros dígitos do segundo pai e o restante do primeiro pai. Os estados de 8 rainhas envolvidos nessa etapa de reprodução são mostrados na Figura 4.16. O exemplo ilustra o fato de que, quando dois estados pais são bastante diferentes, a operação de crossover pode produzir um estado que está longe do estado de qualquer pai. Em geral, a população é bastante diversa no início do processo, e assim o crossover (como a têmpera simulada) freqüentemente executa grandes passos no espaço de estados bem no início do processo de busca e passos menores mais adiante, quando a maioria dos indivíduos é bastante semelhante.

MUTAÇÃO Finalmente, em (e), cada posição está sujeita à **mutação** aleatória com uma pequena probabilidade independente. Um dígito sofreu mutação no primeiro, no terceiro e no quarto descendente. No problema de 8 rainhas, isso corresponde à escolha de uma rainha ao acaso e à movimentação da rainha para um quadrado aleatório em sua coluna. A Figura 4.17 descreve um algoritmo que implementa todas essas etapas.

Como a busca em feixe estocástico, os algoritmos genéticos combinam uma propensão para subir a encosta com a exploração aleatória e com a troca de informações entre processos de busca paralelos. A

11. Existem muitas variantes desta regra de seleção. Pode-se mostrar que o método de **culling**, no qual todos os indivíduos abaixo de um determinado limiar são descartados, converge com maior rapidez que a versão aleatória (Baum *et al.*, 1995).

12. É nessa situação que a codificação é importante. Se for usada uma codificação de 24 bits em vez de 8 dígitos, o ponto de crossover terá uma chance de 2/3 de estar no meio de um dígito, o que resulta em uma mutação essencialmente arbitrária desse dígito.

```
função ALGORITMO-GENÉTICO(população, FN-FITNESS) retorna um indivíduo
  entradas: população, um conjunto de indivíduos
            FN-FITNESS, uma função que mede a adaptação de um indivíduo

  repita
    nova_população ← conjunto vazio
    para i ← 1 até TAMANHO(população) faça
        x ← SELEÇÃO-ALEATÓRIA(população, FN-FITNESS)
        y ← SELEÇÃO-ALEATÓRIA(população, FN-FITNESS)
        filho ← REPRODUZ(x, y)
        se (pequena probabilidade aleatória) então filho ← MUTAÇÃO(filho)
        adicionar filho a nova_população
    população ← nova_população
  até algum indivíduo estar adaptado o suficiente ou até ter decorrido tempo suficiente
  retornar o melhor indivíduo em população, de acordo com FN-FITNESS
```

```
função REPRODUZ(x, y) retorna um indivíduo
  entradas: x, y, indivíduos pais

  n ← COMPRIMENTO(x)
  c ← número aleatório de 1 a n
  retornar CONCATENA(SUBCADEIA(x, 1, c), SUBCADEIA(y, c + 1, n))
```

Figura 4.17 Um algoritmo genético. O algoritmo é igual ao que foi representado na Figura 4.15, com uma variação: em sua versão mais popular, cada união de dois pais produz apenas um descendente, e não dois.

principal vantagem dos algoritmos genéticos, se houver, vem da operação de crossover. Pode ser demonstrado matematicamente que, se as posições do código genético forem permutadas inicialmente em ordem aleatória, o crossover não trará nenhuma vantagem. Intuitivamente, a vantagem vem da habilidade do crossover para combinar grandes blocos de genes que evoluem de forma independente para executar funções úteis, elevando assim o nível de granularidade em que a busca opera. Por exemplo, a colocação das três primeiras rainhas nas posições 2, 4 e 6 (em que elas não atacam as outras) constitui um bloco útil que pode ser combinado com outros blocos para elaborar uma solução.

ESQUEMA

A teoria de algoritmos genéticos explica como isso funciona usando a idéia de **esquema**, uma subcadeia na qual algumas posições podem ser deixadas sem especificação. Por exemplo, o esquema 246***** descreve todos os estados de 8 rainhas em que as três primeiras rainhas estão nas posições 2, 4 e 6, respectivamente. As cadeias que correspondem ao esquema (como 24613578) são chamadas **instâncias** do esquema. É possível mostrar que, se o valor de fitness médio das instâncias de um esquema estiver acima da média, então o número de instâncias do esquema dentro da população crescerá com o passar do tempo. É claro que é improvável que esse efeito seja significativo, caso bits adjacentes estejam totalmente não-relacionados uns com os outros porque, nesse caso, haverá poucos blocos contíguos que proporcionem um benefício consistente. Os algoritmos genéticos funcionam melhor quando os esquemas correspondem a componentes significativos de uma solução. Por exemplo, se a cadeia for uma representação de uma antena, os esquemas poderão representar componentes da antena, como refletores e defletores. É provável que um bom componente seja bom em uma grande variedade de projetos diferentes. Isso sugere que o uso bem-sucedido de algoritmos genéticos exige uma cuidadosa engenharia na representação.

Na prática, os algoritmos genéticos tiveram um amplo impacto sobre problemas de otimização, como layout de circuitos e escalonamento de prestação de serviços. No momento, não está claro se a atração de algoritmos genéticos surge de seu desempenho ou de suas origens esteticamente agradáveis na teoria da evolução. Ainda há muito trabalho a ser feito para identificar as condições sob as quais os algoritmos genéticos funcionam bem.

4.4 Busca local em espaço contínuo

No Capítulo 2, explicamos a distinção entre ambientes discretos e contínuos, assinalando que a maioria dos ambientes reais é contínua. Ainda assim, nenhum dos algoritmos que descrevemos pode manipular espaços de estados contínuos – na maior parte dos casos, a função sucessora retornaria infinitamente muitos estados! Esta seção fornece uma introdução *muito breve* a algumas técnicas de busca local para encontrar soluções ótimas em espaços contínuos. A literatura sobre esse tópico é vasta; muitas técnicas básicas tiveram origem no século XVII, depois do desenvolvimento do cálculo por Newton e Leibniz.[13] Descobriremos usos para essas técnicas em diversos lugares no livro, incluindo os capítulos sobre aprendizado, visão e robótica. Em suma, qualquer atividade ligada ao mundo real.

Evolução e busca

A teoria da **evolução** foi desenvolvida por Charles Darwin em *On the Origin of Species by Means of Natural Selection* (1859). A idéia central é simples: variações (conhecidas como **mutações**) ocorrem na reprodução e serão preservadas em gerações sucessivas, em proporção aproximada a seu efeito sobre a adaptação reprodutiva.

A teoria de Darwin foi desenvolvida sem qualquer conhecimento de como as características dos organismos podem ser herdadas e modificadas. As leis probabilísticas que governam esses processos foram identificadas primeiro por Gregor Mendel (1866), um monge que fez experiências com ervilhas usando o que ele denominou fertilização artificial. Muito mais tarde, Watson e Crick (1953) identificaram a estrutura da molécula do DNA e seu alfabeto, AGTC (adenina, guanina, timina, citosina). No modelo-padrão, a variação ocorre por mutações localizadas na seqüência de genes e por "crossover" (no qual o DNA de um descendente é gerado pela combinação de longas seções de DNA de cada pai).

A analogia com os algoritmos de busca local já foi descrita; a principal diferença entre a busca em feixe estocástico e a evolução é o uso da reprodução *sexuada*, na qual os sucessores são gerados a partir de *vários* organismos, em vez de apenas um. Porém, os mecanismos reais da evolução são muito mais ricos do que permite a maioria dos algoritmos genéticos. Por exemplo, as mutações podem envolver reversões, duplicações e movimentação de grandes blocos de DNA; alguns vírus tomam emprestado o DNA de um organismo e o inserem em outro; e ainda existem genes de transposição que nada fazem além de copiar a si mesmos muitos milhares de vezes dentro do genoma. Existem até mesmo genes que envenenam células de companheiros potenciais que não transportam o gene, aumentando assim suas chances de replicação. O mais importante é o fato de que *os próprios genes codificam os mecanismos* pelos quais o genoma é reproduzido e convertido em um organismo. Em algoritmos genéticos, esses mecanismos constituem um programa separado que não está representado dentro das cadeias que estão sendo manipuladas.

A evolução de Darwin pode parecer um mecanismo ineficiente, tendo gerado cegamente cerca de 10^{45} organismos sem melhorar uma vírgula sequer suas heurísticas de busca. Contudo, cinqüenta anos antes de Darwin, outro grande naturalista francês chamado Jean Lamarck (1809) propôs uma teoria da evolução pela qual as características *adquiridas por adaptação durante a vida de um organismo* seriam transmitidas a seus descendentes. Tal processo seria efetivo, mas não parece ocorrer na natureza. Muito mais tarde, James Baldwin (1896) propôs uma teoria similar em suas características superficiais: que o comportamento aprendido durante a vida de um organismo poderia acelerar a velocidade da evolução. Diferente da teoria de Lamarck, a teoria de Baldwin é inteiramente consistente com a evolução de Darwin, porque se baseia em pressões de seleção operando sobre indivíduos que encontraram pontos ótimos locais no conjunto de comportamentos possíveis permitidos por sua constituição genética. Simulações em modernos computadores confirmam que o "efeito de Baldwin" é real, desde que a evolução "comum" possa criar organismos cuja medida interna de desempenho esteja de alguma forma relacionada à adaptação real.

13. Um conhecimento básico de cálculo multivariado e aritmética vetorial será útil durante a leitura desta seção.

Começaremos com um exemplo. Vamos supor que queremos instalar três novos aeroportos em qualquer lugar na Romênia, de tal forma que a soma dos quadrados das distâncias de cada cidade no mapa (Figura 3.2) até o aeroporto mais próximo seja minimizada. Então, o espaço de estados é definido pelas coordenadas dos aeroportos: (x_1, y_1), (x_2, y_2) e (x_3, y_3). Esse é um espaço *hexadimensional*; também dizemos que os estados são definidos por seis **variáveis**. (Em geral, os estados são definidos por um vetor n-dimensional de variáveis, $\mathbf{x}$.) A movimentação nesse espaço corresponde a mover um ou mais dos aeroportos no mapa. A função objetivo $f(x_1, y_1, x_2, y_2, x_3, y_3)$ é relativamente fácil de calcular para qualquer estado específico, uma vez que sejam calculadas as cidades mais próximas, mas é bastante complicada para se descrever no caso geral.

Uma maneira de evitar problemas contínuos é simplesmente tornar discreta a vizinhança de cada estado. Por exemplo, podemos mover apenas um aeroporto de cada vez na direção x ou y por um valor fixo $\pm\delta$. Com 6 variáveis, isso nos dá 12 sucessores para cada estado. Podemos então aplicar qualquer dos algoritmos de busca local descritos anteriormente. Também é possível aplicar diretamente a subida de encosta estocástica e a têmpera simulada, sem tornar o espaço discreto. Esses algoritmos escolhem sucessores ao acaso, o que pode ser feito pela geração de vetores aleatórios de comprimento δ.

GRADIENTE

Existem muitos métodos que tentam usar o **gradiente** da topologia para encontrar um máximo. O gradiente da função objetivo é um vetor ∇f que fornece a magnitude e a direção da inclinação mais íngreme. Em nosso problema, temos:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right).$$

Em alguns casos, podemos encontrar um máximo resolvendo a equação $\nabla f = 0$. (Por exemplo, isso poderia ser feito se estivéssemos instalando apenas um aeroporto; a solução é a média aritmética das coordenadas de todas as cidades.) Porém, em muitos casos, essa equação não pode ser resolvida de forma fechada. Por exemplo, com três aeroportos, a expressão para o gradiente depende das cidades que estão mais próximas a cada aeroporto no estado corrente. Isso significa que podemos calcular o gradiente *local*, mas não *global*. Mesmo assim, ainda podemos executar a subida de encosta pela trilha mais íngreme atualizando o estado corrente com a fórmula:

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x}),$$

onde α é uma constante pequena. Em outros casos, a função objetivo pode não estar disponível de modo algum em uma forma diferenciável – por exemplo, o valor de um conjunto específico de posições de aeroportos pode ser determinado pela execução de algum pacote de simulação econômica em grande escala. Nesses casos, um **gradiente empírico** pode ser determinado pela avaliação da resposta a pequenos incrementos e decrementos em cada coordenada. A busca de gradiente empírico é igual à subida de encosta pela trilha mais íngreme em uma versão do espaço de estados dividida em unidades discretas.

GRADIENTE EMPÍRICO

Oculta sob a frase "α é uma constante pequena" reside uma enorme variedade de métodos para ajuste de α. O problema básico é que, se α é pequena demais, são necessários muitos passos; se α é grande demais, a busca pode ultrapassar o limite máximo. A técnica de **busca linear** tenta superar esse dilema estendendo a direção de gradiente atual – em geral, pela duplicação repetida de α – até f começar a diminuir novamente. O ponto em que isso ocorrer se torna o novo estado corrente. Existem diversas escolas de pensamento relacionadas ao modo como a nova direção deve ser escolhida nesse ponto.

BUSCA LINEAR

NEWTON-RAPHSON

Para muitos problemas, o algoritmo mais eficiente é o venerável método de **Newton-Raphson** (Newton, 1671; Raphson, 1690). Essa é uma técnica geral para encontrar as raízes de funções – isto

é, resolver equações da forma $g(x) = 0$. Ela funciona calculando uma nova estimativa para a raiz x de acordo com a fórmula de Newton:

$$x \leftarrow x - g(x)/g'(x).$$

Para encontrar um máximo ou um mínimo de f, precisamos encontrar $\mathbf{x}$ tal que o *gradiente* seja zero (isto é, $\nabla f(\mathbf{x}) = \mathbf{0}$). Desse modo $g(x)$ na fórmula de Newton se torna $\nabla f(\mathbf{x})$, e a equação de atualização pode ser escrita em forma de vetor de matriz como:

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x})\, \nabla f(\mathbf{x}),$$

HESSIANA

onde $\mathbf{H}f(\mathbf{x})$ é a matriz **hessiana** de segundas derivadas, cujos elementos H_{ij} são dados por $\partial^2 f/\partial x_i \partial x_j$. Tendo em vista que a matriz hessiana tem n^2 entradas, Newton-Raphson se torna dispendioso em espaços de dimensões elevadas, e foram desenvolvidas muitas aproximações.

Os métodos de busca local se ressentem de máximos locais, picos e platôs em espaços de estados contínuos, de forma muito semelhante ao que ocorre em espaços discretos. Reinícios aleatórios e têmpera simulada são recursos que podem ser usados e freqüentemente são úteis. Porém, os espaços contínuos de dimensões elevadas são lugares grandes em que é fácil se perder.

OTIMIZAÇÃO RESTRITA

Um último tópico sobre o qual seria útil alguma familiarização é a **otimização restrita**. Um problema de otimização é restrito se as soluções devem satisfazer a algumas restrições rígidas sobre os valores de cada variável. Por exemplo, em nosso problema de localização de aeroportos, poderíamos restringir os locais ao interior da Romênia e a áreas de terra firme (e não no meio de lagos). A dificuldade dos problemas de otimização restrita depende da natureza das restrições e da função objetivo. A categoria mais conhecida é a dos problemas de **programação linear**, em que as restrições devem ser desigualdades lineares formando uma região *convexa* e a função objetivo também é linear. Os problemas de programação linear podem ser resolvidos em tempo polinomial no número de variáveis. Problemas com diferentes tipos de restrições e funções objetivo também foram estudados – programação quadrática, programação cônica de segunda ordem e assim por diante.

PROGRAMAÇÃO LINEAR

4.5 Agentes de busca on-line e ambientes desconhecidos

BUSCA OFF-LINE

Até agora nos concentramos em agentes que utilizam algoritmos de **busca off-line**. Eles calculam uma solução completa antes de entrar no mundo real (veja a Figura 3.1) e depois executam a solução sem recorrer a suas percepções. Em contraste, um agente de **busca on-line**[14] opera pela **intercalação** de computação e ação: primeiro, ele executa uma ação, depois observa o ambiente e calcula a próxima ação. A busca on-line é uma boa idéia em domínios dinâmicos ou semidinâmicos – domínios em que existe uma penalidade por continuar calculando durante muito tempo. A busca on-line é uma idéia muito melhor no caso de domínios estocásticos. Em geral, uma busca off-line teria de apresentar um plano de contingência exponencialmente grande que considerasse todos os acontecimentos possíveis, enquanto uma busca on-line só precisa considerar o que realmente acontece. Por exemplo, é aconselhável que um agente de jogo de xadrez faça seu primeiro movimento bem antes de ter calculado o curso completo do jogo.

BUSCA ON-LINE

14. O termo "on-line" é de uso comum em ciência da computação para fazer referência a algoritmos que devem processar dados de entrada à medida que eles são recebidos, em vez de esperar que o conjunto de dados de entrada inteiro se torne disponível.

PROBLEMA DE EXPLORAÇÃO

A busca on-line é uma idéia *necessária* para um **problema de exploração**, em que os estados e as ações são desconhecidos para o agente. Um agente nesse estado de ignorância deve usar suas ações como experimentos para determinar o que fazer em seguida e, conseqüentemente, deve intercalar computação e ação.

O exemplo canônico de busca on-line é um robô colocado em um novo edifício e que tem de explorá-lo para elaborar um mapa que possa ser usado com a finalidade de ir de A até B. Os métodos para escapar de labirintos – um conhecimento exigido dos ambiciosos aspirantes a heróis da Antigüidade – também são exemplos de algoritmos de busca on-line. No entanto, a exploração espacial não é a única forma de exploração. Considere um bebê recém-nascido: ele tem muitas ações possíveis, mas não conhece os resultados de nenhuma delas, e só experimentou alguns dos estados que tem possibilidade de alcançar. A descoberta gradual do bebê de como o mundo funciona é, em parte, um processo de busca on-line.

Problemas de busca on-line

Um problema de busca on-line só pode ser resolvido por um agente que executa ações, e não por um processo puramente computacional. Iremos supor que o agente sabe apenas o seguinte:

- AÇÕES(s), que retorna uma lista de ações permitidas no estado s.
- A função de custo de passo $c(s, a, s')$ – observe que isso não pode ser usado enquanto o agente não souber que s' é o resultado.
- TESTAR-OBJETIVO(s).

Em particular, observe que o agente *não pode* acessar os sucessores de um estado, exceto experimentando realmente todas as ações nesse estado. Por exemplo, no problema de labirinto mostrado na Figura 4.18, o agente não sabe que ir *Para cima* a partir de (1,1) leva a (1,2); nem sabe, tendo feito isso, que ir *Para baixo* o levará de volta a (1,1). Esse grau de ignorância pode ser reduzido em algumas aplicações – por exemplo, um robô explorador poderia saber como suas ações de movimentação funcionam e ser ignorante apenas sobre as posições dos obstáculos.

Vamos supor que o agente sempre possa reconhecer um estado que visitou antes, e também que as ações são determinísticas. (Essas duas últimas suposições serão relaxadas no Capítulo 17.) Finalmente, o agente poderia ter acesso a uma função de heurística admissível $h(s)$ que avalia a distância desde o estado corrente até um estado objetivo. Por exemplo, na Figura 4.18, o agente talvez conheça a posição do objetivo e seja capaz de usar a heurística da distância Manhattan.

Em geral, o objetivo do agente é alcançar um estado objetivo ao mesmo tempo que minimiza o custo. (Outro objetivo possível é simplesmente explorar o ambiente inteiro.) O custo é o custo total de caminho correspondente ao caminho que o agente de fato percorre. É comum comparar esse custo

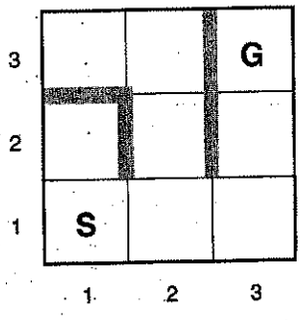
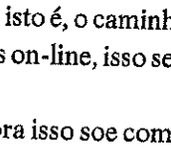


Figura 4.18 Um problema de labirinto simples. O agente começa em S e tem de alcançar G, mas não sabe nada do ambiente.

ao custo de caminho do caminho que o agente seguiria *se conhecesse o espaço de busca com antecedência* – isto é, o caminho real mais curto (ou a exploração completa mais curta). Na linguagem de algoritmos on-line, isso se denomina **razão competitiva**; gostaríamos que ela fosse tão pequena quanto possível.

RAZÃO COMPETITIVA

Embora isso soe como uma solicitação razoável, é fácil ver que a melhor razão competitiva que se pode alcançar é infinita em alguns casos. Por exemplo, se algumas ações forem irreversíveis, a busca on-line poderá chegar acidentalmente a um estado de beco sem saída, a partir do qual nenhum estado objetivo será alcançável. Talvez você considere o termo "acidentalmente" pouco convincente – afinal, poderia existir um algoritmo que não tomasse o caminho do beco sem saída em sua exploração. Nossa afirmativa, para sermos mais precisos, é que *nenhum algoritmo pode evitar becos sem saída em todos os espaços de estados*. Considere os dois espaços de estados de becos sem saída da Figura 4.19(a). Para um algoritmo de busca on-line que visitasse os estados S e A, os dois espaços de estados pareceriam *idênticos*, e assim ele teria de tomar a mesma decisão em ambos. Por essa razão, ele falhará em um deles. Esse é um exemplo de uma **disputa antagônica** – podemos imaginar um oponente que constrói o espaço de estados enquanto o agente o explora e que pode posicionar as metas e os becos sem saída onde desejar.



DISPUTA ANTAGÔNICA

Os becos sem saída constituem uma dificuldade real para a exploração de robôs – escadarias, rampas, precipícios e todos os tipos de terrenos naturais apresentam oportunidades para ações irreversíveis. Para progredir, simplesmente iremos supor que o espaço de estados é **explorável com segurança** – isto é, algum estado objetivo é alcançável a partir de todo estado alcançável. Os espaços de estados com ações reversíveis, como labirintos e quebra-cabeças de 8 peças, podem ser vistos como grafos não-orientados e sem dúvida são exploráveis com segurança.

EXPLORÁVEL COM SEGURANÇA

Mesmo em ambientes exploráveis com segurança, nenhuma razão competitiva limitada poderá ser garantida, se houver caminhos de custo ilimitado. É fácil mostrar isso em ambientes com ações irreversíveis, mas essa afirmativa também permanece verdadeira para o caso reversível, como mostra a Figura 4.19(b). Por essa razão, é comum descrever o desempenho de algoritmos de busca on-line em termos do tamanho do espaço de estados inteiro, e não apenas da profundidade do objetivo mais raso.

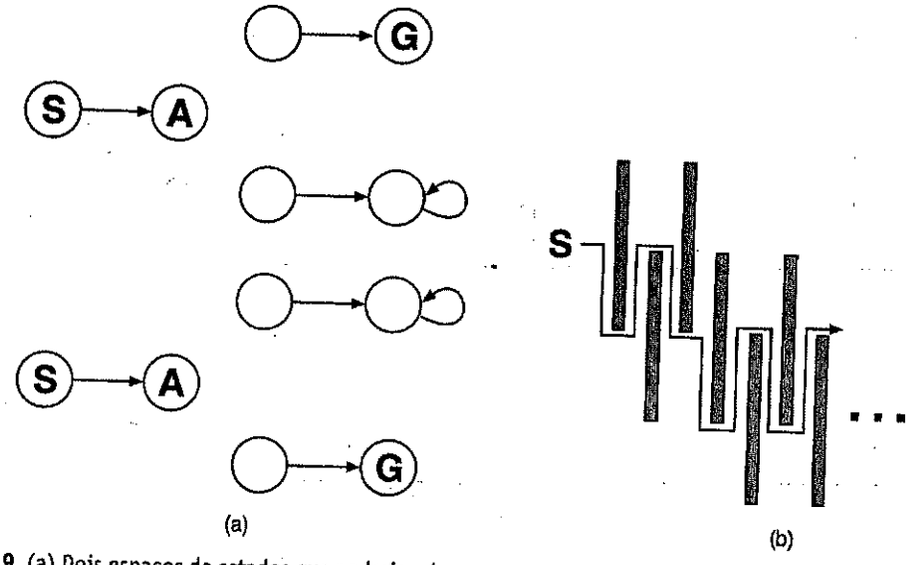


Figura 4.19 (a) Dois espaços de estados que poderiam levar um agente de busca on-line a um beco sem saída. Qualquer agente específico falhará em pelo menos um desses espaços. (b) Um ambiente bidimensional que pode fazer um agente de busca on-line seguir uma rota arbitrariamente ineficiente até o objetivo. Seja qual for a escolha do agente, o oponente bloqueará essa rota com outra parede longa e estreita, para que o caminho seguido seja muito mais longo que o melhor caminho possível.

Agentes de busca on-line

Depois de cada ação, um agente on-line recebe uma percepção informando-o de qual estado ele alcançou; a partir dessa informação, ele pode ampliar seu mapa do ambiente. O mapa atual é usado para decidir aonde ir em seguida. Essa intercalação de planejamento e ação significa que os algoritmos de busca on-line são bastante diferentes dos algoritmos de busca off-line que vimos antes. Por exemplo, algoritmos off-line como A* têm a habilidade de expandir um nó em uma parte do espaço, e depois expandir imediatamente um nó em outra parte do espaço, porque a expansão de nós envolve ações simuladas, em vez de ações reais. Por outro lado, um algoritmo on-line só pode expandir um nó que ele ocupa fisicamente. Para evitar percorrer todos os caminhos da árvore para expandir o próximo nó, parece melhor expandir nós em uma ordem *local*. A busca em profundidade tem exatamente essa propriedade, porque (exceto quando ocorre retrocesso) o próximo nó expandido é um filho do nó expandido anterior.

Um agente de busca on-line em profundidade é mostrado na Figura 4.20. Esse agente armazena seu mapa em uma tabela, *resultado*[*a*, *s*], que registra o estado resultante da execução da ação *a* no estado *s*. Sempre que uma ação do estado corrente não é explorada, o agente experimente essa ação. A dificuldade surge quando o agente tenta todas as ações em um estado. Na busca off-line em profundidade, o estado é simplesmente retirado da fila; em uma busca on-line, o agente tem de regressar fisicamente. Na busca em profundidade, isso significa voltar até o estado a partir do qual o agente entrou no estado corrente mais recentemente. Isso é conseguido mantendo-se uma tabela que lista, para cada estado, os estados predecessores aos quais o agente ainda não regressou. Se o agente esgotar os estados aos quais ele pode regressar, sua busca estará completa.

Recomendamos que o leitor acompanhe o progresso do AGENTE-BP-ON-LINE quando aplicado ao labirinto da Figura 4.18. É bastante fácil verificar que o agente acabará, no pior caso, percorrendo toda ligação entre estados no espaço de estados exatamente duas vezes. Para a exploração, isso é ótimo; por outro lado, para encontrar um objetivo, a razão competitiva do agente poderia ser arbitrariamente ruim se resultasse em uma longa excursão quando houvesse um objetivo bem próximo ao estado inicial. Uma variante on-line do aprofundamento iterativo resolve esse problema; no caso de um ambiente que seja uma árvore uniforme, a razão competitiva de tal agente será uma constante pequena.

```
função AGENTE-BP-ON-LINE(s') retorna uma ação
  entradas: s', uma percepção que identifica o estado corrente
  variáveis estáticas: resultado, uma tabela, indexada por ação e estado, inicialmente vazia
        inexplorado, uma tabela que lista, para cada estado visitado, as ações ainda não-tentadas
        sem_retrocesso, uma tabela que lista, para cada estado visitado, os retrocessos ainda não-tentados
        s, a, o estado e a ação anteriores, inicialmente nulos

  se TESTAR-OBJETIVO(s ) então retornar parar
  se s  é um novo estado então inexplorado[s ] ← AÇÕES(s )
  se s é não-nulo então faça
    resultado [a, s] ← s
    somar s ao início de sem_retrocesso[s ]
  se inexplorado[s ] é vazio então
    se sem_retrocesso[s ] é vazio então retornar parar
    senão a ← uma ação b tal que resultado[b, s ] = DESEMPILHA(sem_retrocesso[s ])
  senão a ← DESEMPILHA(inexplorado[s ])
  s ← s'
  retornar a
```

Figura 4.20 Um agente de busca on-line que utiliza exploração em profundidade. O agente só é aplicável em espaços de busca bidirecionais.

Em conseqüência de seu método de retrocesso, AGENTE-BP-ON-LINE só funcionará em espaços de estados nos quais as ações são reversíveis. Existem algoritmos um pouco mais complexos que funcionam em espaços de estados gerais, mas nenhum desses algoritmos tem uma razão competitiva limitada.

Busca local on-line

Assim como a busca em profundidade, a **busca de subida de encosta** tem a propriedade de localidade em suas expansões de nós. De fato, como ela mantém apenas um estado corrente na memória, a busca de subida de encosta *já* é um algoritmo de busca on-line! Infelizmente, não é muito útil em sua forma mais simples, porque deixa o agente parado em máximos locais, sem ter para onde ir. Além disso, os reinícios aleatórios não podem ser usados, porque o agente não tem como se transportar para um novo estado.

PERCURSO ALEATÓRIO

Em vez de reinícios aleatórios, poderíamos considerar o uso de um **percurso aleatório** para explorar o ambiente. Um percurso aleatório simplesmente seleciona ao acaso uma das ações disponíveis do estado corrente; a preferência pode ser dada a ações que ainda não foram tentadas. É fácil provar que um percurso aleatório irá *eventualmente* encontrar um objetivo ou completar sua exploração, desde que o espaço seja finito.[15] Por outro lado, o processo pode ser muito lento. A Figura 4.21 mostra um ambiente em que um percurso aleatório levará exponencialmente muitos passos para encontrar o objetivo porque, em cada passo, o progresso no sentido inverso é duas vezes mais provável que o progresso no sentido direto. É claro que o exemplo é fictício, mas existem muitos espaços de estados reais cuja topologia resulta nesses tipos de "armadilhas" para percursos aleatórios.

A ampliação da subida de encosta com *memória* em vez de aleatoriedade acaba sendo uma abordagem mais efetiva. A idéia básica é armazenar uma "melhor estimativa atual" $H(s)$ do custo para alcançar o objetivo a partir de cada estado que tenha sido visitado. $H(s)$ começa sendo apenas a estimativa heurística $h(s)$ e é atualizada à medida que o agente ganha experiência no espaço de estados. A Figura 4.22 mostra um exemplo simples em um espaço de estados unidimensional. Em (a), o agente parece estar paralisado em um mínimo local plano no estado sombreado. Em vez de permanecer onde está, o agente deve seguir o que parece ser o melhor caminho até o objetivo, com base nas estimativas de custo atuais para seus vizinhos. O custo estimado para alcançar o objetivo através de um vizinho s' é o custo para chegar a s' somado ao custo estimado para ir de lá até um objetivo – isto é, $c(s, a, s') + H(s')$. No exemplo, existem duas ações com custos estimados $1 + 9$ e $1 + 2$, e assim parece melhor mover-se para a direita. Agora, é claro que a estimativa de custo 2 para o estado sombreado foi exageradamente otimista. Tendo em vista que o melhor movimento custa 1 e levou a um estado distante pelo menos 2 passos de um objetivo, o estado sombreado deve estar pelo menos 3 passos distante de

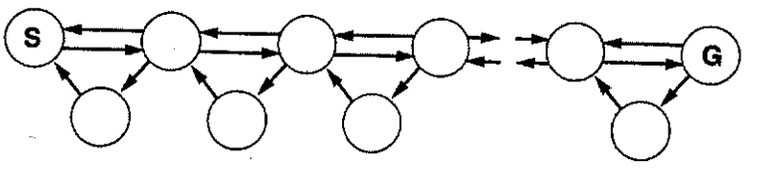


Figura 4.21 Um ambiente em que um percurso aleatório levará exponencialmente muitos passos para encontrar o objetivo.

15. O caso infinito é muito mais complicado. Os percursos aleatórios são completos em grades infinitas unidimensionais e bidimensionais, mas não em grades tridimensionais! Nesse último caso, a probabilidade de o percurso resultar no retorno ao ponto de partida é de apenas 0,3405, aproximadamente. (Veja em Hughes, 1995, uma introdução geral.)

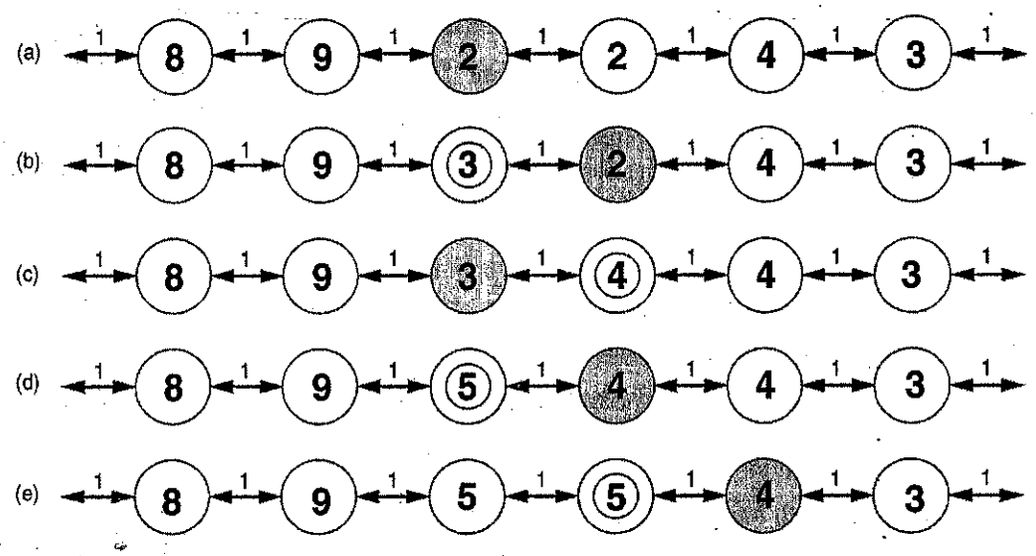


Figura 4.22 Cinco iterações de ATRA* em um espaço de estados unidimensional. Cada estado é identificado com $H(s)$, a estimativa de custo atual para alcançar um objetivo, e cada arco é identificado com seu custo de passo. O estado sombreado marca a posição do agente, e os valores atualizados em cada iteração estão dentro de círculos.

um objetivo, e portanto seu H deve ser atualizado de acordo, como mostra a Figura 4.22(b). Continuando esse processo, o agente irá recuar e avançar mais duas vezes, atualizando H em cada vez e "aplainando" o mínimo local até escapar para a direita.

ATRA*

Um agente que implementa esse esquema, chamado aprendizado em tempo real A* (**ATRA***), é mostrado na Figura 4.23. Da mesma forma que AGENTE-BP-ON-LINE, ele constrói um mapa do ambiente usando a tabela *resultado*. Ele atualiza a estimativa de custo para o estado que acabou de deixar, e depois escolhe o "aparentemente melhor" movimento de acordo com suas estimativas de custo atuais. Um detalhe importante é que sempre se supõe que ações ainda não-tentadas em um estado s levam imediatamente ao objetivo com o menor custo possível, ou seja, $h(s)$. Esse **otimismo sob incerteza** encoraja o agente a explorar novos caminhos, possivelmente promissores.

OTIMISMO SOB INCERTEZA

Um agente ATRA* oferece a garantia de encontrar um objetivo em qualquer ambiente finito explorável com segurança. Porém, diferente de A*, ele não é completo para espaços de estados infinitos – há casos em que ele pode ficar indefinidamente perdido. O agente pode explorar um ambiente de n estados em $O(n^2)$ passos no pior caso, mas com freqüência funciona muito melhor. O agente ATRA* é apenas um em uma grande família de agentes on-line que podem ser definidos pela especificação da regra de seleção de ação e a regra de atualização de diferentes modos. Discutiremos essa família, desenvolvida originalmente para ambientes estocásticos, no Capítulo 21.

Aprendizado em busca on-line

A ignorância inicial dos agentes de busca on-line oferece várias oportunidades para aprendizado. Primeiro, os agentes aprendem um "mapa" do ambiente – mais precisamente, o resultado de cada ação em cada estado – apenas registrando cada uma de suas experiências. (Note que a suposição de ambientes determinísticos significa que uma experiência é suficiente para cada ação.) Em segundo lugar, os agentes de busca local adquirem estimativas mais precisas do valor de cada estado usando regras de atualização local, como no ATRA*. No Capítulo 21, veremos que essas atualizações convergem eventualmente para valores *exatos* em todo estado, desde que o agente explore o espaço de estados da maneira correta. Uma vez conhecidos valores exatos podem ser tomadas decisões ótimas pela simples movimentação para o sucessor de valor mais alto – isto é, a subida de encosta pura é então uma estratégia ótima.

```
função AGENTE-ATRA*(s') retorna uma ação
  entradas: s', uma percepção que identifica o estado corrente
  variáveis estáticas: resultado, uma tabela, indexada por ação e estado, inicialmente vazia
        H, uma tabela de estimativas de custo indexada pelo estado, inicialmente vazia
        s, a, o estado e a ação anteriores, inicialmente nulos

  se TESTAR-OBJETIVO(s') então retornar parar
  se s' é um novo estado (não em H) então H[s'] ← h(s')
  a menos que s seja nulo
    resultado[a, s] ← s
    H[s] ← min_{b∈Ações(s)} CUSTO-ATRA*(s, b, resultado[b, s], H)
  a ← uma ação b em AÇÕES(s') que minimiza CUSTO-ATRA*(s', b, resultado[b, s'], H)
  s ← s'
  retornar a

função CUSTO-ATRA*(s, a, s', H) retorna uma estimativa de custo
  se s' é indefinido então retornar h(s)
  senão retornar c(s, a, s') + H[s']
```

Figura 4.23 AGENTE-ATRA* seleciona uma ação de acordo com os valores de estados vizinhos, que são atualizados à medida que o agente se move no espaço de estados.

Se você seguiu nossa sugestão para acompanhar o comportamento de AGENTE-BP-ON-LINE no ambiente da Figura 4.18, terá notado que o agente não é muito brilhante. Por exemplo, depois de ver que a ação *Para cima* vai de (1,1) para (1,2), o agente ainda não tem idéia de que a ação *Para baixo* volta a (1,1) ou de que a ação *Para cima* também vai de (2,1) para (2,2), de (2,2) para (2,3) e assim por diante. Em geral, gostaríamos que o agente aprendesse que *Para cima* aumenta a coordenada *y*, a menos que exista uma parede no caminho, que *Para baixo* a reduz e assim por diante. Para que isso aconteça, primeiro precisamos de uma representação formal e explicitamente manipulável para esses tipos de regras gerais; até agora, ocultamos a informação contida na caixa-preta chamada função sucessora. A Parte III é dedicada a essa questão. Em segundo lugar, precisamos de algoritmos que possam construir regras gerais adequadas a partir das observações específicas feitas pelo agente. Esses assuntos serão estudados no Capítulo 18.

4.6 Resumo

Este capítulo examinou a aplicação de **heurísticas** para reduzir custos de busca. Observamos uma série de algoritmos que usam heurísticas e descobrimos que o caráter ótimo tem um preço alto em termos de custo de busca, mesmo com boas heurísticas.

- A **busca pela melhor escolha** é simplesmente BUSCA-EM-GRAFO, onde os nós não-expandidos de custo mínimo (de acordo com alguma medida) são selecionados para expansão. Em geral, os algoritmos de busca pela melhor escolha usam uma função **heurística** $h(n)$ que estima o custo de uma solução a partir de n.
- A **busca gulosa pela melhor escolha** expande nós com $h(n)$ mínimo. Ela não é ótima, mas com freqüência é eficiente.
- A **busca A*** expande nós com valor mínimo para $f(n) = g(n) + h(n)$. A* é completa e ótima, desde que possamos garantir que $h(n)$ é admissível (para BUSCA-EM-ÁRVORE) ou consistente (para BUSCA-EM-GRAFO). A complexidade de espaço de A* ainda é proibitiva.

- O desempenho de algoritmos de busca heurística depende da qualidade da função heurística. Às vezes, boas heurísticas podem ser construídas relaxando-se a definição do problema, calculando-se previamente os custos de solução para subproblemas em um banco de dados de padrões ou aprendendo-se a partir da experiência com a classe do problema.
- **BRPM** e **LMSA*** são algoritmos de busca robustos e ótimos que utilizam quantidades limitadas de memória; sendo dado tempo suficiente, eles podem resolver problemas que A* não consegue resolver porque esgota a memória.
- Métodos de *busca local* como **subida de encosta** operam sobre formulações de estados completos, mantendo na memória apenas um pequeno número de nós. Foram desenvolvidos vários algoritmos estocásticos, incluindo a **têmpera simulada**, que retorna soluções ótimas quando recebe um cronograma de resfriamento apropriado. Muitos métodos de busca local também podem ser usados para resolver problemas em espaços contínuos.
- Um **algoritmo genético** é uma busca de subida de encosta estocástica em que é mantida uma grande população de estados. Novos estados são gerados por **mutação** e por **crossover**, que combina pares de estados da população.
- Os **problemas de exploração** surgem quando o agente não tem nenhuma idéia sobre os estados e ações de seu ambiente. No caso de ambientes exploráveis com segurança, agentes de **busca on-line** podem construir um mapa e encontrar um objetivo, se existir algum. A atualização de estimativas heurísticas a partir da experiência fornece um método efetivo para escapar de mínimos locais.

Notas bibliográficas e históricas

O uso de informações heurísticas em resolução de problemas aparece em um primeiro ensaio de Simon e Newell (1958), mas a expressão "busca heurística" e o uso de funções heurísticas que estimam a distância até o objetivo veio um pouco mais tarde (Newell e Ernst, 1965; Lin, 1965). Doran e Michie (1966) conduziram extensos estudos experimentais de busca heurística aplicada a vários problemas, especialmente o quebra-cabeça de 8 peças e o quebra-cabeça de 15 peças. Embora tenham realizado análises teóricas de comprimento de caminho e "penetração" (a razão entre o comprimento de caminho e o número total de nós examinados até o momento) em busca heurística, Doran e Michie parecem ter ignorado as informações fornecidas pelo comprimento de caminho atual. O algoritmo A*, que incorpora o comprimento de caminho atual na busca heurística, foi desenvolvido por Hart, Nilsson e Raphael (1968), com algumas correções posteriores (Hart *et al.*, 1972). Dechter e Pearl (1985) demonstraram a eficiência ótima de A*.

O artigo original sobre A* introduziu a condição de consistência em funções heurísticas. A condição de monotonicidade foi introduzida por Pohl (1977) como uma alternativa mais simples, mas Pearl (1984) mostrou que as duas eram equivalentes. Diversos algoritmos anteriores a A* usaram o equivalente a listas abertas e fechadas; esses algoritmos incluem a busca em extensão, em profundidade e de custo uniforme (Bellman, 1957; Dijkstra, 1959). Em particular, o trabalho de Bellman mostrou a importância dos custos aditivos de caminhos na simplificação de algoritmos de otimização.

Pohl (1970, 1977) foi pioneiro no estudo do relacionamento entre o erro em funções heurísticas e a complexidade de tempo de A*. A prova de que A* funciona em tempo linear se o erro na função heurística é limitado por uma constante pode ser encontrada em Pohl (1977) e em Gaschnig (1979). Pearl (1984) fortaleceu esse resultado para permitir um crescimento logarítmico no erro. O "fator de ramificação efetiva" como medida da eficiência da busca heurística foi proposto por Nilsson (1971).

Existem muitas variações do algoritmo A*. Pohl (1973) propôs o uso de *ponderação dinâmica*, que utiliza uma soma ponderada $f_w(n) = w_g g(n) + w_h h(n)$ do comprimento de caminho atual e da função heurística como uma função de avaliação, em vez da soma simples $f(n) = g(n) + h(n)$ usada em A*. Os pesos w_g e w_h são ajustados dinamicamente à medida que a busca progride. Pode-se mostrar que o algoritmo de Pohl é ε admissível – isto é, tem a garantia de encontrar soluções dentro de um fator 1 + ε da solução ótima – onde ε é um parâmetro fornecido ao algoritmo. A mesma propriedade é exibida pelo algoritmo A^*_ε (Pearl, 1984), que pode selecionar qualquer nó a partir da borda, desde que seu custo de f esteja dentro de um fator 1 + ε do nó de borda de custo f mais baixo. A seleção pode ser feita para minimizar o custo da busca.

A* e outros algoritmos de busca em espaço de estados estão intimamente relacionados às técnicas de *desvio e limite* bastante usadas em pesquisa operacional (Lawler e Wood, 1966). Os relacionamentos entre a busca em espaço de estados e desvio e limite foram investigados em profundidade (Kumar e Kanal, 1983; Nau *et al.*, 1984; Kumar *et al.*, 1988). Martelli e Montanari (1978) demonstraram uma conexão entre programação dinâmica (veja o Capítulo 17) e certos tipos de busca em espaço de estados. Kumar e Kanal (1988) tentaram uma "unificação geral" da busca heurística, da programação dinâmica e das técnicas de desvio e limite sob o nome PDC – o "processo de decisão composto".

Como os computadores do final da década de 1950 e do início dos anos 70 tinham no máximo alguns milhares de bytes de memória principal, a busca heurística limitada pela memória foi um antigo tópico de pesquisa. O Graph Traverser (Doran e Michie, 1966), um dos mais antigos programas de busca, entrega o resultado a um operador depois de realizar uma busca pela melhor escolha até o limite da memória. O AIA* (Korf, 1985a, 1985b) foi o primeiro algoritmo de busca heurística ótimo e limitado pela memória amplamente utilizado, e foi desenvolvido um grande número de variantes. Uma análise da eficiência do AIA* e de suas dificuldades com heurísticas de valores reais aparece em Patrick *et al.*, (1992).

EXPANSÃO ITERATIVA

O BRPM (Korf, 1991, 1993) é na realidade um pouco mais complicado que o algoritmo mostrado na Figura 4.5, o qual é mais parecido com o algoritmo desenvolvido independentemente, chamado **expansão iterativa** ou EI (Russell, 1992). O BRPM utiliza um limite inferior, bem como o limite superior; os dois algoritmos se comportam de forma idêntica com heurísticas admissíveis, mas o BRPM expande nós ordenados pelo melhor, mesmo com uma heurística inadmissível. A idéia de controlar o melhor caminho alternativo apareceu antes na elegante implementação de A* em Prolog feita por Bratko (1986) e no algoritmo DTA* (Russell e Wefald, 1991). Este último trabalho também discute espaços de meta-estados e meta-aprendizagem.

O algoritmo LMA* apareceu em Chakrabarti *et al.* (1989). LMSA*, ou LMA* simplificado, emergiu de uma tentativa de implementar o LMA* como um algoritmo de comparação para o EI (Russell, 1992). Kaindl e Khorsand (1994) aplicaram o LMSA* para produzir um algoritmo de busca bidirecional substancialmente mais rápido que algoritmos anteriores. Korf e Zhang (2000) descrevem uma abordagem de dividir e conquistar, e Zhou e Hansen (2002) introduzem a busca em grafos de A* limitada pela memória. Korf (1995) estuda técnicas de busca limitada pela memória.

A idéia de que heurísticas admissíveis podem ser derivadas por relaxação de problemas aparece no importante artigo de Held e Karp (1970), que usaram a heurística de árvore de amplitude mínima para resolver o PCV. (Veja o Exercício 4.8.)

A automatização do processo de relaxamento foi implementada com sucesso por Prieditis (1993), fundamentada em um trabalho anterior com Mostow (Mostow e Prieditis, 1989). O uso de bancos de dados de padrões para derivar heurísticas admissíveis se deve a Gasser (1995) e a Culberson e Schaeffer (1998); os bancos de dados de padrões disjuntos são descritos por Korf e Felner (2002). A interpretação probabilística de heurísticas foi investigada em profundidade por Pearl (1984) e por Hansson e Mayer (1989).

Sem dúvida, a fonte mais completa sobre heurísticas e algoritmos de busca heurísticas é o texto de Pearl (1984), *Heuristics*. Esse livro fornece uma cobertura especialmente boa da ampla variedade de ramificações e variações de A*, inclusive rigorosas provas de suas propriedades formais. Kanal e Kumar (1988) apresentam uma antologia de artigos importantes sobre busca heurística. Novos resultados sobre algoritmos de busca aparecem regularmente no periódico *Artificial Intelligence*.

As técnicas de busca local têm uma longa história em matemática e ciência da computação. Na realidade, o método de Newton-Raphson (Newton, 1671; Raphson, 1690) pode ser visto como um método de busca local muito eficiente para espaços contínuos em que as informações de gradiente estão disponíveis. Brent (1973) é uma referência clássica para algoritmos de otimização que não exigem tais informações. A busca em feixe, que apresentamos como um algoritmo de busca local, teve origem como uma variante de largura limitada da programação dinâmica para reconhecimento de voz no sistema HARPY (Lowerre, 1976). Um algoritmo relacionado é analisado em profundidade por Pearl (1984, Capítulo 5).

O tópico de busca local foi revigorado nos últimos anos por resultados surpreendentemente bons para problemas de satisfação de restrições como o de n rainhas (Minton *et al.*, 1992) e raciocínio lógico (Selman *et al.*, 1992) e pela incorporação da aleatoriedade, de várias buscas simultâneas e de outros aperfeiçoamentos. Esse renascimento do que Christos Papadimitriou chamou algoritmos da "Nova Era" também despertou interesse crescente entre os cientistas da computação teórica (Koutsoupias e Papadimitriou, 1992; Aldous e Vazirani, 1994). No campo da pesquisa operacional, uma variante da subida de encosta chamada **busca tabu** ganhou popularidade (Glover, 1989; Glover e Laguna, 1997). Baseado em modelos de memória limitada de curto prazo em seres humanos, esse algoritmo mantém uma lista de tabus de k estados visitados anteriormente que não podem ser visitados outra vez; além de melhorar a eficiência durante a busca em grafos, isso pode permitir que o algoritmo escape de alguns mínimos locais. Outra melhoria útil em relação à subida de encosta é o algoritmo STAGE (Boyan e Moore, 1998). A idéia é usar os máximos locais encontrados pela subida de encosta com reinício aleatório para ter uma idéia da forma geral da topologia. O algoritmo ajusta uma superfície suave ao conjunto de máximos locais, e depois calcula analiticamente o máximo global dessa superfície. Esse se torna o novo ponto de reinício. Demonstrou-se que o algoritmo funciona na prática em problemas difíceis. Gomes *et al.* (1998) mostraram que as distribuições de tempo de execução de algoritmos de retrocesso sistemático com freqüência têm uma **distribuição de cauda pesada**; isso significa que a probabilidade de um tempo de execução muito longo é maior do que seria previsto se os tempos de execução estivessem distribuídos de maneira normal. Isso fornece uma justificativa teórica para reinícios aleatórios.

BUSCA TABU

DISTRIBUIÇÃO DE CAUDA PESADA

A têmpera simulada foi descrita primeiro por Kirkpatrick *et al.* (1983), que a tomou emprestada diretamente do **algoritmo de Metropolis** (usado para simular sistemas complexos em física (Metropolis *et al.*, 1953) e foi criado supostamente durante um jantar festivo em Los Alamos). A têmpera simulada agora é um campo em si mesmo, com centenas de artigos publicados a cada ano.

Encontrar soluções ótimas em espaços contínuos é o principal assunto de diversos campos, incluindo a **teoria de otimização**, a **teoria de controle ótimo** e o **cálculo de variações**. Introduções adequadas (e práticas) são oferecidas por Press *et al.* (2002) e Bishop (1995). A **programação linear** (PL) foi uma das primeiras aplicações de computadores; o **algoritmo simplex** (Wood e Dantzig, 1949; Dantzig, 1949) ainda é usado, apesar da complexidade exponencial do pior caso. Karmarkar (1984) desenvolveu um algoritmo prático de tempo polinomial para PL.

O trabalho de Sewall Wright (1931) sobre o conceito de uma **topologia de fitness** foi um importante precursor para o desenvolvimento de algoritmos genéticos. Na década de 1950, diversos estatísticos, incluindo Box (1957) e Friedman (1959), utilizaram técnicas evolucionárias em problemas de otimização, mas somente quando Rechenberg (1965, 1973) introduziu as **estratégias de evolução** para resolver problemas de otimização de aerofólios a abordagem ganhou popularidade. Nas dé-

ESTRATÉGIAS DE EVOLUÇÃO

cadas de 1960 e 1970, John Holland (1975) defendeu os algoritmos genéticos, não só como uma ferramenta útil, mas também como um método para expandir nossa compreensão da adaptação, biológica ou não (Holland, 1995). O movimento de **vida artificial** (Langton, 1995) leva essa idéia um passo adiante, visualizando os produtos de algoritmos genéticos como *organismos*, em vez de soluções para problemas. O trabalho nesse campo desenvolvido por Hinton e Nowlan (1987) e por Ackley e Littman (1991) foi realizado principalmente para esclarecer as implicações do efeito de Baldwin. Para um conhecimento geral sobre os fundamentos da evolução, recomendamos Smith e Szathmáry (1999).

VIDA ARTIFICIAL

A maioria das comparações de algoritmos genéticos com outras abordagens (em especial a subida de encosta estocástica) descobriu que os algoritmos genéticos convergem mais lentamente (O'Reilly e Oppacher, 1994; Mitchell *et al.*, 1996; Juels e Wattenberg, 1996; Baluja, 1997). Tais descobertas não são universalmente populares dentro da comunidade de AG, mas tentativas recentes dentro dessa comunidade para entender a busca baseada na população como uma forma aproximada de aprendizado bayesiano (veja o Capítulo 20) talvez ajudem a reduzir o abismo entre o campo e seus críticos (Pelikan *et al.*, 1999). A teoria de **sistemas quadráticos dinâmicos** também pode explicar o desempenho dos AGs (Rabani *et al.*, 1998). Veja em Lohn *et al.* (2001) um exemplo de AGs aplicado ao projeto de antenas, e em Larrañaga *et al.* (1999) uma aplicação ao problema do caixeiro-viajante.

PROGRAMAÇÃO GENÉTICA

O campo de **programação genética** está intimamente relacionado aos algoritmos genéticos. A principal diferença é que as representações que sofrem mutações e combinações são programas, em vez de cadeias de bits. Os programas são representados sob a forma de árvores de expressões; as expressões podem estar em uma linguagem-padrão como Lisp ou podem ser projetadas especificamente para representar circuitos, controladores de robôs e assim por diante. O crossover envolve a união de subárvores, e não de subcadeias. Essa forma de mutação garante que os descendentes serão expressões bem-formadas, o que não ocorreria se os programas fossem manipulados como cadeias.

O recente interesse em programação genética foi incentivado pelo trabalho de John Koza (Koza, 1992, 1994), mas remonta pelo menos aos primeiros experimentos com código de máquina realizados por Friedberg (1958) e com autômatos de estados finitos, desenvolvidos por Fogel *et al.* (1966). Como no caso de algoritmos genéticos, existe um debate sobre a eficácia da técnica. Koza *et al.* (1999) descrevem uma variedade de experimentos no projeto automatizado dos dispositivos de circuitos utilizando programação genética.

Os periódicos *Evolutionary Computation* e *IEEE Transactions on Evolutionary Computation* estudam algoritmos genéticos e programação genética; também são encontrados artigos em *Complex Systems*, *Adaptive Behavior* e *Artificial Life*. As principais conferências são a International Conference on Genetic Algorithms e a Conference on Genetic Programming, fundidas recentemente para formar a Genetic and Evolutionary Computation Conference. Os textos de Melanie Mitchell (1996) e David Fogel (2000) oferecem boas visões gerais do campo.

Os algoritmos para explorar espaços de estados desconhecidos têm despertado interesse por muitos séculos. A busca em profundidade em um labirinto pode ser implementada mantendo-se a mão esquerda na parede; os ciclos podem ser evitados marcando-se cada junção. A busca em profundidade falha com ações irreversíveis; o problema mais geral de exploração de **grafos eulerianos** (isto é, grafos em que cada nó tem números iguais de arestas de entrada e saída) foi resolvido por um algoritmo devido a Hierholzer (1873). O primeiro estudo algorítmico completo do problema de exploração de grafos arbitrários foi empreendido por Deng e Papadimitriou (1990), que desenvolveram um algoritmo completamente geral, mas mostraram que não é possível nenhuma razão competitiva limitada para explorar um grafo geral. Papadimitriou e Yannakakis (1991) examinaram a questão de encontrar caminhos até um objetivo em ambientes de planejamento de caminhos geométricos (em que todas as ações são reversíveis). Eles mostraram que uma pequena razão competitiva pode ser alcançada com obstáculos quadrados, mas que não é possível alcançar nenhuma razão limitada com obstáculos retangulares em geral. (Veja a Figura 4.19.)

GRAFOS EULERIANOS

BUSCA EM TEMPO REAL

O algoritmo ATRA* foi desenvolvido por Korf (1990) como parte de uma investigação da **busca em tempo real** para ambientes em que o agente deve atuar depois de buscar apenas durante um período de tempo fixo (uma situação muito mais comum em jogos com dois participantes). O ATRA* é de fato um caso especial de algoritmos de aprendizado de reforço para ambientes estocásticos (Barto *et al.*, 1995). Sua política de otimismo sob incerteza – sempre se dirigir para o estado não-visitado mais próximo – pode resultar em um padrão de exploração menos eficiente no caso não-informado do que a busca simples em profundidade (Koenig, 2000). Dasgupta *et al.* (1994) mostram que a busca de aprofundamento iterativo on-line é otimamente eficiente para encontrar umo objetivo em uma árvore uniforme sem informações heurísticas. Diversas variantes informadas sobre o tema do ATRA* foram desenvolvidas com diferentes métodos de busca e atualização dentro da porção conhecida do grafo (Pemberton e Korf, 1992). Até agora, não existe uma boa compreensão de como encontrar metas com eficiência ótima quando se utilizam informações heurísticas.

BUSCA PARALELA

O tópico de algoritmos de **busca paralela** não foi abordado no capítulo porque em parte exige uma longa discussão das arquiteturas de computadores paralelos. A busca paralela está se tornando um tópico importante, tanto em IA quanto em ciência da computação teórica. Uma breve introdução à literatura da IA pode ser encontrada em Mahanti e Daniels (1993).

Exercícios

4.1 Represente a operação da busca A* aplicada ao problema de ir até Bucareste a partir de Lugoj usando a heurística de distância em linha reta. Isto é, mostre a seqüência de nós que o algoritmo irá considerar e a pontuação de f, g e h para cada nó.

4.2 O **algoritmo de caminho heurístico** é uma busca pela melhor escolha na qual a função objetivo é $f(n) = (2-w)g(n) + wh(n)$. Para que valores de w esse algoritmo oferece a garantia de ser ótimo? Que espécie de busca ele executa quando $w = 0$? E quando $w = 1$? E quando $w = 2$?

4.3 Prove cada uma das afirmações a seguir:

a. A busca em extensão é um caso especial de busca de custo uniforme.

b. A busca em extensão, a busca em profundidade e a busca de custo uniforme são casos especiais de busca pela melhor escolha.

c. A busca de custo uniforme é um caso especial de busca A*.

4.4 Crie um espaço de estados em que A* usando BUSCA-EM-GRAFO retorne uma solução não-ótima com uma função $h(n)$ que seja admissível, mas inconsistente.

4.5 Vimos na página 95 que a heurística de distância em linha reta leva a busca gulosa pela melhor escolha a se perder no problema de ir de Iasi até Fagaras. Porém, a heurística é perfeita no problema oposto: ir de Fagaras até Iasi. Existem problemas para os quais a heurística é falha em ambos os sentidos?

4.6 Crie uma função heurística para o quebra-cabeça de 8 peças que algumas vezes realize estimativas exageradas, e mostre como ela pode levar a uma solução não-ótima em um problema específico. (Use um computador para ajudá-lo, se desejar.) Prove que, se h nunca superestimar por um valor maior que c, A* usando h retornará uma solução cujo custo excede o da solução ótima por não mais que c.

4.7 Prove que, se uma heurística é consistente, ela tem de ser admissível. Construa uma heurística admissível que não seja consistente.

4.8 O problema do caixeiro-viajante (PCV) pode ser resolvido por meio da heurística de árvore de amplitude mínima (AAM), usada para avaliar o custo da conclusão de um tour, considerando-se que

um tour parcial já tenha sido construído. O custo da AAM de um conjunto de cidades é a menor soma dos custos de vínculos de qualquer árvore que conecta todas as cidades.

a. Mostre como essa heurística pode ser derivada de uma versão relaxada do PCV.
b. Mostre que a heurística de AAM domina a distância em linha reta.
c. Escreva um gerador de problemas para instâncias do PCV em que as cidades sejam representadas por pontos aleatórios no quadrado unitário.
d. Encontre um algoritmo eficiente na literatura para construir a AAM, e use esse algoritmo com um algoritmo de busca admissível para resolver instâncias do PCV.

4.9 Na página 111, definimos o relaxamento do quebra-cabeça de 8 peças em que um bloco pode se mover do quadrado A para o quadrado B, se B estiver vazio. A solução exata desse problema define a **heurística de Gaschnig** (Gaschnig, 1979). Explique por que a heurística de Gaschnig é pelo menos tão precisa quanto h_1 (blocos mal posicionados) e mostre casos em que ela é mais precisa que h_1 e h_2 (distância Manhattan). Você poderia sugerir um modo de calcular a heurística de Gaschnig com eficiência?

4.10 Apresentamos duas heurísticas simples para o quebra-cabeça de 8 peças: a distância Manhattan e blocos mal posicionados. Várias heurísticas na literatura pretendem aperfeiçoar essas heurísticas – por exemplo, veja Nilsson (1971), Mostow e Prieditis (1989) e Hansson *et al.* (1992). Teste essas afirmações implementando as heurísticas e comparando o desempenho dos algoritmos resultantes.

4.11 Forneça o nome do algoritmo que resulta de cada um dos seguintes casos especiais:

a. Busca em feixe local com $k = 1$.
b. Busca em feixe local com $k = \infty$.
c. Têmpera simulada com $T = 0$ em todos os momentos.
d. Algoritmo genético com tamanho de população $N = 1$.

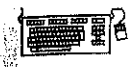
4.12 Às vezes, não existe nenhuma boa função de avaliação para um problema, mas existe um bom método de comparação: um modo de saber se um nó é melhor que outro sem atribuir valores numéricos a qualquer deles. Mostre que isso é suficiente para realizar uma busca pela melhor escolha. Existe um análogo de A*?

4.13 Relacione a complexidade de tempo de ATRA* à sua complexidade de espaço.

4.14 Suponha que um agente esteja em um ambiente de labirinto 3×3 como o da Figura 4.18. O agente sabe que sua posição inicial é (1,1), que o objetivo está em (3,3) e que as quatro ações *Para cima*, *Para baixo*, *Esquerda*, *Direita* têm seus efeitos habituais, a menos que sejam bloqueadas por uma parede. O agente *não* sabe onde estão as paredes internas. Em qualquer estado específico, o agente percebe o conjunto de ações válidas; ele também pode saber se o estado já foi visitado antes ou é um novo estado.

a. Explique como esse problema de busca on-line pode ser visualizado como uma busca off-line no espaço de estados de convicção, onde o estado de convicção inicial inclui todas as configurações possíveis de ambiente. Qual é o tamanho do estado de convicção inicial? Qual é o tamanho do espaço de estados de convicção?
b. Quantas percepções distintas são possíveis no estado inicial?
c. Descreva as primeiras ramificações de um plano de contingência para esse problema. Qual é o tamanho (aproximado) do plano completo?

Note que esse plano de contingência é uma solução para *todo ambiente possível* que se ajusta à descrição dada. Portanto, a intercalação de busca e execução não é estritamente necessária, nem mesmo em ambientes desconhecidos.

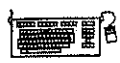


4.15 Neste exercício, exploraremos o uso de métodos de busca local para resolver PCVs do tipo definido no Exercício 4.8.

a. Crie uma abordagem de subida de encosta para resolver PCVs. Compare os resultados com soluções ótimas obtidas por meio do algoritmo A* com a heurística de AAM (Exercício 4.8).

b. Crie uma abordagem de algoritmo genético para o problema do caixeiro-viajante. Compare os resultados com os das outras abordagens. Talvez você queira consultar Larrañaga *et al.* (1999) para obter algumas sugestões de representações.

4.16 Gere um grande número de instâncias do quebra-cabeça de 8 peças e de 8 rainhas e resolva-as (quando possível) por subida de encosta (variantes de subida mais íngreme e primeira escolha), por subida de encosta com reinício aleatório e por têmpera simulada. Meça o custo da busca e a porcentagem de problemas resolvidos e elabore um grafo desses valores contra o custo da solução ótima. Comente seus resultados.



4.17 Neste exercício, examinaremos a subida de encosta no contexto da navegação de robôs, usando o ambiente da Figura 3.22 como exemplo.

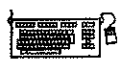
a. Repita o Exercício 3.16 usando subida de encosta. Seu agente ficará paralisado em um mínimo local? É *possível* que ele fique paralisado com obstáculos convexos?

b. Construa um ambiente poligonal não-convexo no qual o agente fique paralisado.

c. Modifique o algoritmo de subida de encosta de forma que, em vez de realizar uma busca de profundidade 1 a fim de decidir para onde ir em seguida, ele realize uma busca de profundidade k. Ele deve encontrar o melhor caminho de k passos e percorrer uma passo ao longo dele, e depois repetir o processo.

d. Existe algum k para o qual o novo algoritmo oferece a garantia de escapar de mínimos locais?

e. Explique como o ATRA* permite ao agente escapar de mínimos locais nesse caso.



4.18 Compare o desempenho de A* e BRPM em um conjunto de problemas gerados aleatoriamente nos domínios do quebra-cabeça de 8 peças (com distância Manhattan) e de PCV (com AAM – veja o Exercício 4.8). Discuta seus resultados. O que acontece ao desempenho do BRPM quando um pequeno número aleatório é adicionado aos valores heurísticos no domínio do quebra-cabeça de 8 peças?