CAPÍTULO 3

Resolução de problemas por meio de busca

Em que vemos como um agente pode encontrar uma seqüência de ações que alcança seus objetivos, quando nenhuma ação isolada é capaz de fazê-lo.

Os agentes mais simples examinados no Capítulo 2 eram os agentes reativos, que baseiam suas ações em um mapeamento direto de estados em ações. Tais agentes não podem operar bem em ambientes para os quais esse mapeamento seria grande demais para armazenar e levaria muito tempo para se aprender. Por outro lado, os agentes baseados em objetivos podem ter sucesso, considerando ações futuras e a conveniência de seus resultados.

AGENTE DE RESOLUÇÃO DE PROBLEMAS

Este capítulo descreve um tipo de agente baseado em objetivos chamado **agente de resolução de problemas**. Os agentes de resolução de problemas decidem o que fazer encontrando seqüências de ações que levam a estados desejáveis. Começamos definindo com precisão os elementos que constituem um "problema" e sua "solução", e apresentamos diversos exemplos para ilustrar essas definições. Em seguida, descrevemos vários algoritmos de busca de uso geral que podem ser usados para resolver esses problemas e comparamos as vantagens de cada algoritmo. Os algoritmos são **sem informação**, no sentido de que não recebem nenhuma informação sobre o problema além de sua definição. O Capítulo 4 lida com algoritmos de busca com informação, aqueles que têm alguma idéia de onde procurar soluções.

Este capítulo utiliza conceitos da análise de algoritmos. Os leitores não-familiarizados com os conceitos de complexidade assintótica (isto é, a notação $O(\,)$) e NP-completeza devem consultar o Apêndice A.

3.1 Agentes de resolução de problemas

Os agentes inteligentes devem maximizar sua medida de desempenho. Como mencionamos no Capítulo 2, esse objetivo é às vezes simplificado se o agente pode adotar um **objetivo** e desejar satisfazê-lo. Primeiro, vamos examinar por que e como um agente poderia realizar essa ação.

Imagine um agente na cidade de Arad, na Romênia, aproveitando uma viagem de férias. A medida de desempenho do agente contém muitos fatores: ele quer melhorar seu bronzeado, melhorar seu conhecimento do idioma romeno, ver as paisagens, apreciar a vida noturna (como ela é), evitar ressacas e assim por diante. O problema de decisão é complexo e envolve muitos compromissos e leitura cuidadosa de guias de viagem. Agora, suponha que o agente tenha uma passagem não-reembolsável para partir de Bucareste na manhã seguinte. Nesse caso, faz sentido para o agente adotar o **objetivo** de chegar a Bucareste. Os cursos de ação que não chegam a Bucareste a tempo podem ser rejeitados sem consideração adicional, e o problema de decisão do agente fica bastante simplificado. Os objetivos ajudam a organizar o comportamento, limitando os objetivos que o agente está tentando alcançar. A **formulação de objetivos**, baseada na situação atual e na medida de desempenho do agente, é o primeiro passo para a resolução de problemas.

FORMULAÇÃO DE OBJETIVOS

Vamos considerar que um objetivo é um conjunto de estados do mundo – exatamente os estados em que o objetivo é satisfeito. A tarefa do agente é descobrir que seqüência de ações o levará a um estado objetivo. Antes de poder fazer isso, ele precisa decidir que espécies de ações e estados deve considerar. Se tentasse considerar ações ao nível de "mover o pé esquerdo para frente uma polegada" ou "girar o volante um grau para a esquerda", o agente provavelmente nunca conseguiria sair do estacionamento, quanto mais chegar a Bucareste, porque nesse nível de detalhe existe muita incerteza no mundo e haveria muitos passos para se chegar a uma solução. A **formulação de problemas** é o processo de decidir que ações e estados devem ser considerados, dado um objetivo. Examinaremos mais adiante os detalhes desse processo. No momento, vamos supor que o agente irá considerar ações no nível de dirigir desde uma cidade importante até outra. Os estados que ele considerará corresponderão portanto a estar em uma determinada cidade.[1]

FORMULAÇÃO DE PROBLEMAS

Nosso agente agora adotou o objetivo de dirigir para Bucareste, e está considerando para onde ir a partir de Arad. Existem três estradas que saem de Arad, uma em direção a Sibiu, uma para Timisoara e uma para Zerind. Nenhuma dessas atinge o objetivo; assim, a menos que o agente esteja muito familiarizado com a geografia da Romênia, ele não saberá que estrada deve seguir.[2] Em outras palavras, o agente não saberá qual das ações possíveis é a melhor, porque não conhece o suficiente sobre o estado que resulta da execução de cada ação. Se o agente não tiver nenhum conhecimento adicional, ele ficará paralisado. O melhor que ele poderá fazer é escolher uma das ações ao acaso.

Porém, suponha que o agente tenha um mapa da Romênia, seja em papel ou em sua memória. A finalidade de um mapa é fornecer ao agente informações sobre os estados em que ele próprio pode entrar, e sobre as ações que ele pode executar. O agente pode usar essas informações para considerar estágios subseqüentes de uma jornada hipotética passando por cada uma das três cidades, procurando descobrir um percurso que eventualmente chegue a Bucareste. Depois de encontrar um caminho no mapa de Arad até Bucareste, ele poderá alcançar seu objetivo executando as ações de dirigir que correspondem aos passos da viagem. Em geral, *um agente com várias opções imediatas de valor desconhecido pode decidir o que fazer examinando primeiro diferentes* seqüências *de ações possíveis que levam a estados de valor conhecido, e depois escolhendo a melhor seqüência.*

BUSCA

SOLUÇÃO

Esse processo de procurar por tal seqüência é chamado **busca**. Um algoritmo de busca recebe um problema como entrada e retorna uma **solução** sob a forma de uma seqüência de ações. Depois que uma so-

1. Observe que cada um desses "estados" realmente corresponde a um grande *conjunto* de estados do mundo, porque um estado real do mundo especifica todos os aspectos da realidade. É importante ter em mente a distinção entre estados em resolução de problemas e estados do mundo.

2. Estamos supondo que a maioria dos leitores está na mesma posição e pode se imaginar com facilidade tão desorientado quanto nosso agente. Nos desculpamos com leitores romenos que são incapazes de tirar proveito desse dispositivo pedagógico.

EXECUÇÃO

lução é encontrada, as ações que ela recomenda podem ser executadas. Isso se chama fase de **execução**. Desse modo, temos um simples projeto de "formular, buscar, executar" para o agente, como mostra a Figura 3.1. Depois de formular um objetivo e um problema a resolver, o agente chama um procedimento de busca para resolvê-lo. Em seguida, ele utiliza a solução para orientar suas ações, fazendo o que a solução recomendar como a próxima ação – em geral, a primeira ação da seqüência – e então removendo esse passo da seqüência. Depois que a solução for executada, o agente formulará um novo objetivo.

```
função AGENTE-DE RESOLUÇÃO-DE-PROBLEMAS-SIMPLES(percepção) retorna uma ação
    entradas: percepção, uma percepção
    variáveis estáticas: seq, uma seqüência de ações, inicialmente vazia
                         estado, alguma descrição do estado atual do mundo
                         objetivo, um objetivo, inicialmente nulo
                         problema, uma formulação de problema

estado ← ATUALIZAR-ESTADO(estado, percepção)
se seq está vazia então faça
    objetivo ← FORMULAR-OBJETIVO(estado)
    problema ← FORMULAR-PROBLEMA(estado, objetivo)
    seq ← BUSCA(problema)
ação ← PRIMEIRO(seq)
seq ← RESTO(seq)
retornar ação
```

Figura 3.1 Um agente simples de resolução de problemas. Primeiro, ele formula um objetivo e um problema, busca uma seqüência de ações que resolveriam o problema e depois executa as ações uma de cada vez. Quando essa seqüência se completa, ele formula outro objetivo e recomeça. Observe que, quando está executando a seqüência, o agente ignora suas percepções; ele supõe que a solução que encontrou sempre funcionará.

Primeiramente, descreveremos o processo de formulação de problemas, e depois dedicaremos a parte principal do capítulo a diversos algoritmos correspondentes à função BUSCA. Não descreveremos mais os detalhes internos das funções ATUALIZAR-ESTADO e FORMULAR-OBJETIVO neste capítulo.

Antes de mergulharmos nos detalhes, vamos fazer uma pequena pausa para ver onde os agentes de resolução de problemas se encaixam na discussão de agentes e ambientes do Capítulo 2. O projeto de agente da Figura 3.1 pressupõe que o ambiente é estático, porque a formulação e resolução do problema é feita sem dedicar atenção a quaisquer mudanças que possam estar ocorrendo no ambiente. O projeto do agente também pressupõe que o estado inicial é conhecido; é mais fácil conhecê-lo se o ambiente é **observável**. A idéia de enumerar "cursos alternativos de ação" pressupõe que o ambiente pode ser visto como **discreto**. Por fim, e mais importante, o projeto de agente pressupõe que o ambiente é **determinístico**. As soluções para problemas são seqüências de ações únicas, e assim elas não podem lidar com quaisquer eventos inesperados; além disso, as soluções são executadas sem atenção às percepções! Um agente que executa seus planos com os olhos fechados, por assim dizer, deve estar bastante seguro do que está acontecendo. (Os adeptos da teoria de controle chamam esse sistema de **laço de repetição aberto**, porque ignorar as percepções rompe o laço de repetição entre agente e ambiente.) Todas essas suposições significam que estamos lidando com os tipos mais fáceis de ambientes, e essa é uma razão pela qual este capítulo se encontra no início do livro. A Seção 3.6 examina rapidamente o que acontece quando relaxamos as suposições de possibilidade de observação e determinismo. Os Capítulos 12 e 17 estudam esse assunto com profundidade muito maior.

LAÇO DE REPETIÇÃO ABERTO

Problemas e soluções bem definidos

PROBLEMA

Um **problema** pode ser definido formalmente por quatro componentes:

ESTADO INICIAL

- O **estado inicial** em que o agente começa. Por exemplo, o estado inicial do nosso agente na Romênia poderia ser descrito como *Em(Arad)*.

FUNÇÃO SUCESSOR

- Uma descrição das **ações** possíveis que estão disponíveis para o agente. A formulação mais comum[3] utiliza uma **função sucessor**. Dado um estado particular x, SUCESSOR (x) retorna um conjunto de pares ordenados $\langle$*ação,sucessor*$\rangle$, em que cada ação é uma das ações válidas no estado x e cada sucessor é um estado que pode ser alcançado a partir de x aplicando-se a ação. Por exemplo, a partir do estado *Em(Arad)*, a função sucessor para o problema da Romênia retornaria:

  $\{\langle Ir(Sibiu), Em(Sibiu)\rangle, \langle Ir(Timisoara), Em(Timisoara)\rangle, \langle Ir(Zerind), Em(Zerind)\rangle\}$

ESPAÇO DE ESTADOS

Juntos, o estado inicial e a função sucessor definem implicitamente o **espaço de estados** do problema – o conjunto de todos os estados acessíveis a partir do estado inicial. O espaço de estados forma um grafo em que os nós são estados e os arcos entre nós são ações. (O mapa da Romênia mostrado na Figura 3.2 pode ser interpretado como um grafo do espaço de estados se visualizarmos cada estrada como duas possíveis ações de dirigir, uma para cada sentido.) Um **caminho** no espaço de estados é uma seqüência de estados conectados por uma seqüência de ações.

CAMINHO

TESTE DE OBJETIVO

- O **teste de objetivo**, que determina se um dado estado é um estado objetivo. Às vezes existe um conjunto explícito de estados objetivo possíveis, e o teste simplesmente verifica se o estado dado é um deles. O objetivo do agente na Romênia é o conjunto unitário {*Em(Bucareste)*}. Algumas vezes, o objetivo é especificado por uma propriedade abstrata, e não por um conjunto de estados explicitamente enumerado. Por exemplo, no xadrez, o objetivo é alcançar um estado chamado "xeque-mate", em que o rei do oponente está sob ataque e não consegue escapar.

CUSTO DE CAMINHO

- Uma função de **custo de caminho** que atribui um custo numérico a cada caminho. O agente de resolução de problemas escolhe uma função de custo que reflete sua própria medida de desempenho. Para o agente que tenta chegar a Bucareste, o tempo é essencial, e assim o custo de um caminho poderia ser seu comprimento em quilômetros. Neste capítulo, supomos que o custo de um caminho pode ser descrito como a soma dos custos das ações individuais ao longo do caminho. O **custo de passo** de adotar a ação a para ir do estado x ao estado y é denotado por $c(x, a, y)$. Os custos dos passos para a Romênia são mostrados na Figura 3.2 como distâncias de rotas. Vamos supor que os custos dos passos são não-negativos.[4]

CUSTO DE PASSO

Os elementos precedentes definem um problema e podem ser reunidos em uma única estrutura de dados que é fornecida como entrada para um algoritmo de resolução de problemas. Uma **solução** para um problema é um caminho desde o estado inicial até um estado objetivo. A qualidade da solução é medida pela função de custo de caminho, e uma **solução ótima** tem o menor custo de caminho entre todas as soluções.

SOLUÇÃO ÓTIMA

Formulação de problemas

Na seção anterior, propusemos uma formulação do problema de chegar a Bucareste em termos do estado inicial, da função sucessor, do teste de objetivo e do custo de caminho. Essa formulação parece razoável, ainda que omita um número muito grande de aspectos do mundo real. Compare a descrição do estado simples que escolhemos, *Em(Arad)*, a uma viagem real cruzando o país, onde o estado

3. Uma formulação alternativa utiliza um conjunto de **operadores** que podem ser aplicados a um estado para gerar sucessores.

4. As implicações de custos negativos são exploradas no Exercício 3.17.

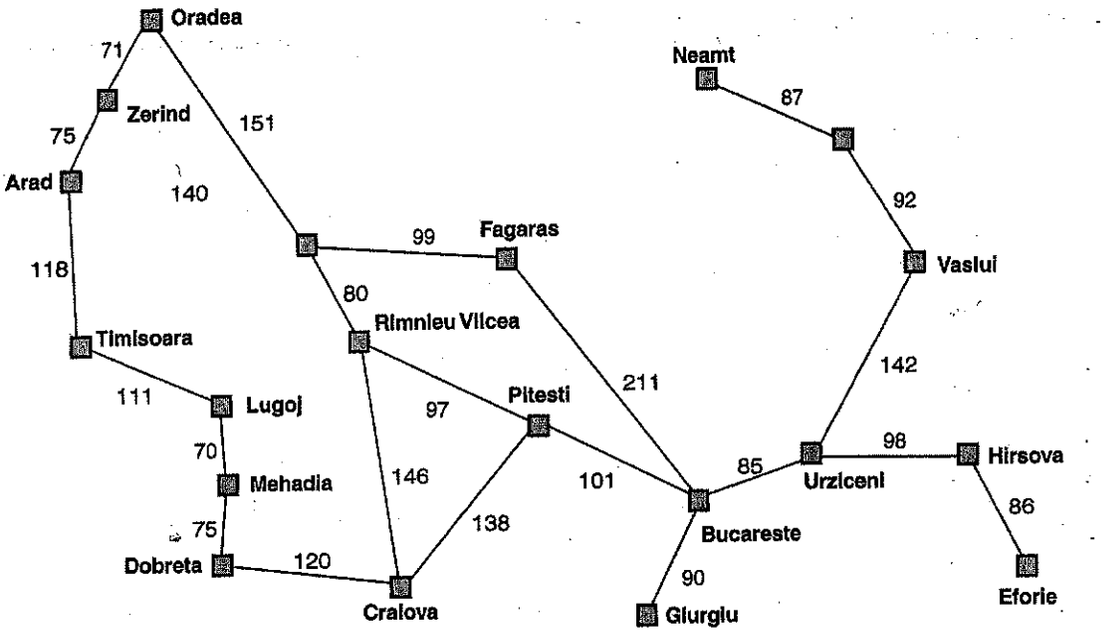


Figura 3.2 Um mapa rodoviário simplificado de parte da Romênia.

do mundo inclui muitos itens: os companheiros de viagem, o que toca no rádio, a paisagem vista da janela, a existência ou não de policiais nas proximidades, a distância até a próxima parada para descanso, as condições da estrada, o tempo e assim por diante. Todas essas considerações são omitidas de nossas descrições de estados, porque são irrelevantes para o problema de encontrar uma rota para Bucareste. O processo de remover detalhes de uma representação é chamado **abstração**.

ABSTRAÇÃO

Além de abstrair a descrição do estado, devemos abstrair as próprias ações. Uma ação de direção tem muitos efeitos. Além de mudar a posição do veículo e de seus ocupantes, ela gasta tempo, consome combustível, gera poluição e muda o agente (como se costuma dizer, viajar é expandir seus horizontes). Em nossa formulação, levamos em conta apenas a mudança de posição. Além disso, existem muitas ações que omitiremos por completo: ligar o rádio, olhar pela janela, diminuir a velocidade ao se aproximar de policiais e assim por diante. É claro que não especificamos ações no nível de "girar o volante três graus para a esquerda".

Podemos ser mais precisos quanto à definição do nível apropriado de abstração? Pense nos estados abstratos e nas ações que escolhemos como correspondentes a grandes conjuntos de estados detalhados do mundo e seqüências detalhadas de ações. Agora, considere uma solução para o problema abstrato: por exemplo, o caminho de Arad para Sibiu para Rimnicu Vilcea para Pitesti para Bucareste. Essa solução abstrata corresponde a um grande número de caminhos mais detalhados. Como exemplo, poderíamos dirigir com o rádio ligado entre Sibiu e Rimnicu Vilcea, e depois desligá-lo pelo restante da viagem. A abstração será *válida* se pudermos expandir qualquer solução abstrata em uma solução no mundo mais detalhado; uma condição suficiente é que, para cada estado detalhado que seja "em Arad", existe um caminho detalhado para algum estado que seja "em Sibiu" e assim por diante. A abstração é *útil* se a execução de cada uma das ações na solução é mais fácil que o problema original; nesse caso, elas são fáceis o bastante para poderem ser executadas sem busca ou planejamento adicional por um agente de direção médio. A escolha de uma boa abstração envolve portanto a remoção da maior quantidade possível de detalhes, enquanto se preserva a validade e se assegura que as ações abstratas são fáceis de executar. Se não fosse a habilidade de elaborar abstrações úteis, os agentes inteligentes seriam completamente sufocados pelo mundo real.

3.2 Exemplos de problemas

A abordagem de resolução de problemas é aplicada a uma ampla série de ambientes de tarefas. Listamos aqui alguns dos mais conhecidos, fazendo distinção entre *miniproblemas* e *problemas do mundo real*. Um **miniproblema** se destina a ilustrar ou exercitar diversos métodos de resolução de problemas. Ele pode ter uma descrição concisa e exata. Isso significa que ele pode ser usado com facilidade por diferentes buscadores, com a finalidade de comparar o desempenho de algoritmos. Um **problema do mundo real** é aquele cujas soluções de fato preocupam as pessoas. Eles tendem a não apresentar uma única descrição consensual, mas tentaremos dar uma idéia geral de suas formulações.

MINIPROBLEMA

PROBLEMA DO MUNDO REAL

Miniproblemas

O primeiro exemplo que examinaremos é o mundo de aspirador de pó introduzido inicialmente no Capítulo 2. (Veja a Figura 2.2.) Ele pode ser formulado como um problema, assim:

- **Estados:** O agente está em uma entre duas posições, cada uma das quais pode conter sujeira ou não. Desse modo, há $2 \times 2^2 = 8$ estados do mundo possíveis.
- **Estado inicial:** Qualquer estado pode ser designado como o estado inicial.
- **Função sucessor:** Gera os estados válidos que resultam da tentativa de executar as três ações (*Esquerda*, *Direita* e *Aspirar*). O espaço de estados completo é mostrado na Figura 3.3.
- **Teste de objetivo:** Verifica se todos os quadrados estão limpos.
- **Custo de caminho:** Cada passo custa 1, e assim o custo do caminho é o número de passos do caminho.

Comparado com o mundo real, esse miniproblema tem posições discretas, sujeira discreta, limpeza confiável e nunca é desorganizado depois de limpo. (Na Seção 3.6, relaxaremos essas suposições.) Devemos observar que o estado é determinado tanto pela posição do agente quanto pela localização da sujeira. Um ambiente maior com n posições tem $n\,2^n$ estados.

QUEBRA-CABEÇA DE 8 PEÇAS

O **quebra-cabeça de 8 peças** (veja um exemplo na Figura 3.4), consiste em um tabuleiro de 3×3 com oito peças numeradas e um espaço vazio. Uma peça adjacente ao espaço vazio pode deslizar para o espaço. O objetivo é alcançar um estado objetivo especificado, como o do lado direito da figura. A formulação-padrão é:

- **Estados:** Uma descrição de estado especifica a posição de cada uma das oito peças e do espaço vazio em um dos nove quadrados.
- **Estado inicial:** Qualquer estado pode ser designado como o estado inicial. Observe que qualquer objetivo específico pode ser alcançado a partir de exatamente metade dos estados iniciais possíveis (Exercício 3.4).
- **Função sucessor:** Gera os estados válidos que resultam da tentativa de executar as quatro ações (o espaço vazio se desloca para a *Esquerda*, *Direita*, *Acima* ou *Abaixo*).
- **Teste de objetivo:** Verifica se o estado corresponde à configuração de objetivo mostrada na Figura 3.4. (São possíveis outras configurações de objetivos.)
- **Custo de caminho:** Cada passo custa 1, e assim o custo do caminho é o número de passos do caminho.

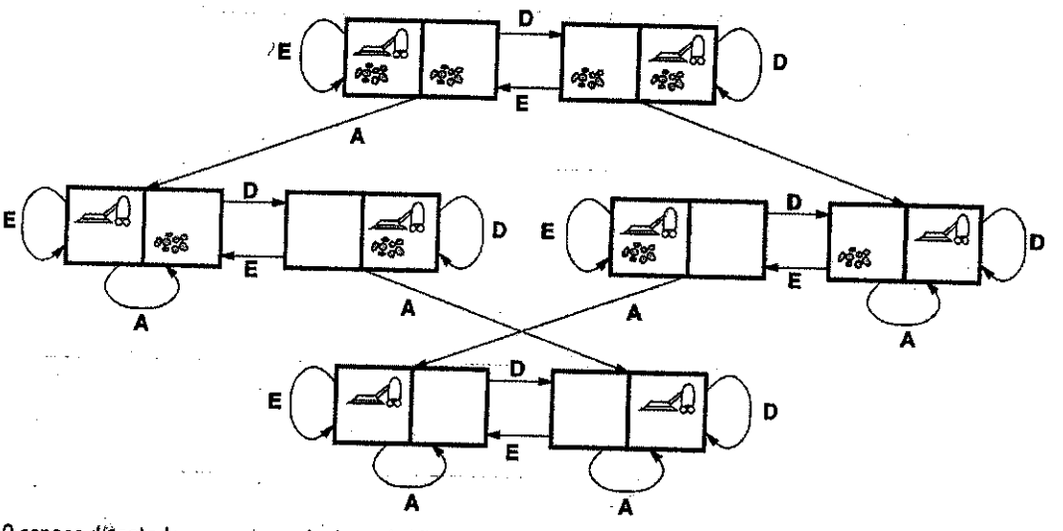


Figura 3.3 O espaço de estados para o mundo de aspirador de pó. Os arcos denotam ações: E = *Esquerda*, D = *Direita*, A = *Aspirar*.

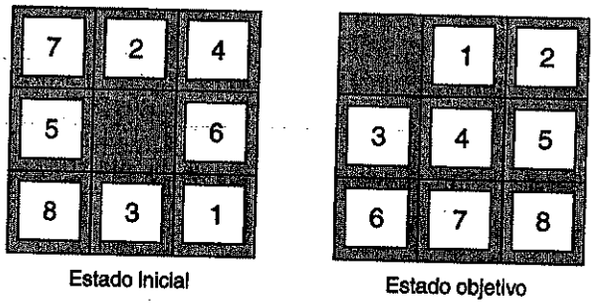


Figura 3.4 Uma instância típica do quebra-cabeça de 8 peças.

Que abstrações incluímos aqui? As ações são reduzidas a seus estados iniciais e finais, ignorando-se as posições intermediárias por onde o bloco está deslizando. Abstraímos ações como sacudir o tabuleiro quando as peças ficam presas ou extrair as peças com uma faca e colocá-las de volta no tabuleiro. Ficamos com uma descrição das regras do quebra-cabeça, evitando todos os detalhes de manipulações físicas.

QUEBRA-CABEÇAS DE BLOCOS DESLIZANTES

O quebra-cabeça de 8 peças pertence à família de **quebra-cabeças de blocos deslizantes**, usados com freqüência como problemas de teste para novos algoritmos de busca em IA. Sabe-se que essa classe geral é NP-completa; assim, ninguém espera encontrar métodos significativamente melhores no pior caso que os algoritmos de busca descritos neste capítulo e no próximo. O quebra-cabeça de 8 peças tem $9!/2 = 181.440$ estados acessíveis e é resolvido com facilidade. O quebra-cabeça de 15 peças (em um tabuleiro de 4×4) tem aproximadamente 1,3 trilhão de estados, e instâncias aleatórias podem ser resolvidas de forma ótima em alguns milissegundos pelos melhores algoritmos de busca. O quebra-cabeça de 24 peças (em um tabuleiro de 5×5) tem cerca de 10^{25} estados, e instâncias aleatórias ainda são bastante difíceis de resolver de forma ótima com as máquinas e os algoritmos atuais.

PROBLEMA DE 8 RAINHAS

O objetivo do **problema de 8 rainhas** é posicionar oito rainhas em um tabuleiro de xadrez de tal forma que nenhuma rainha ataque qualquer outra. (Uma rainha ataca qualquer peça situada na mesma linha, coluna ou diagonal.) A Figura 3.5 mostra uma tentativa de solução que falhou: a rainha na coluna mais à direita é atacada pela rainha do canto superior esquerdo.

Embora existam algoritmos de propósito especial eficientes para esse problema e para toda a família de n rainhas, ele continua a ser um interessante problema de teste para algoritmos de busca. Há dois tipos principais de formulações. Uma **formulação incremental** envolve operadores que *ampliam* a descrição do estados, iniciando com um estado vazio; para o problema de 8 rainhas, isso significa que

FORMULAÇÃO INCREMENTAL

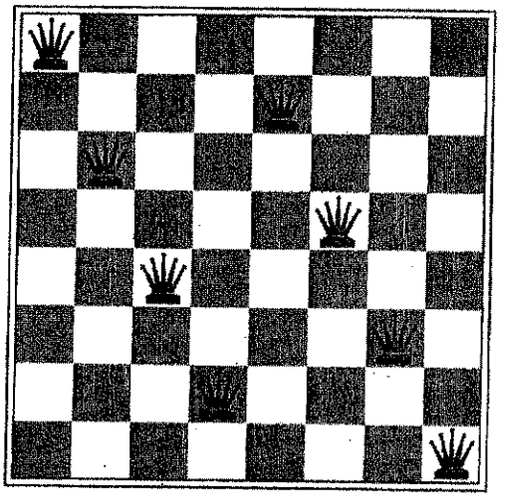


Figura 3.5 Uma quase-solução para o problema de 8 rainhas. (A solução fica como exercício).

FORMULAÇÃO DE ESTADOS COMPLETOS

cada ação acrescenta uma rainha ao estado. Uma **formulação de estados completos** começa com todas as 8 rainhas e as desloca pelo tabuleiro. Em qualquer caso o custo de caminho não tem nenhum interesse, porque apenas o estado final é importante. A primeira formulação incremental que se poderia experimentar é:

- **Estados:** Qualquer disposição de 0 a 8 rainhas no tabuleiro é um estado.
- **Estado inicial:** Nenhuma rainha no tabuleiro.
- **Função sucessor:** Colocar uma rainha em qualquer quadrado vazio.
- **Teste de objetivo:** 8 rainhas estão no tabuleiro e nenhuma é atacada.

Nessa formulação, temos $64 \cdot 63 \cdots 57 \approx 3 \times 10^{14}$ seqüências possíveis para investigar. Uma formulação melhor proibiria a colocação de uma rainha em qualquer quadrado que já estiver sob ataque:

- **Estados:** Os estados são disposições de n rainhas ($0 \leq n \leq 8$), uma por coluna nas n colunas mais à esquerda, sem que nenhuma rainha ataque outra.
- **Função sucessor:** Adicione uma rainha a qualquer quadrado na coluna vazia mais à esquerda, de tal modo que ela não seja atacada por qualquer outra rainha.

Essa formulação reduz o espaço de estados de 8 rainhas de 3×10^{14} para apenas 2.057, e as soluções são fáceis de encontrar. Por outro lado, para 100 rainhas, a formulação inicial tem aproximadamente 10^{400} estados, enquanto a formulação melhorada tem cerca de 10^{52} estados (Exercício 3.5). Essa é uma enorme redução, mas o espaço de estados aperfeiçoado ainda é grande demais para ser tratado pelos algoritmos deste capítulo. O Capítulo 4 descreve a formulação de estados completos, e o Capítulo 5 apresenta um algoritmo simples que facilita a resolução até mesmo do problema de um milhão de rainhas.

Problemas do mundo real

PROBLEMA DE ROTEAMENTO

Já vimos como o **problema de roteamento** é definido em termos de posições especificadas e transições ao longo de ligações entre elas. Os algoritmos de roteamento são utilizados em uma grande variedade de aplicações, como o roteamento em redes de computadores, planejamento de operações militares e sistemas de planejamento de viagens aéreas. Em geral, a especificação desses problemas é complexa.

Considere um exemplo simplificado de um problema de viagens aéreas, especificado como a seguir:

- **Estados:** Cada um é representado por uma posição (por exemplo, um aeroporto) e pela hora atual.
- **Estado inicial:** É especificado pelo problema.
- **Função sucessor:** Retorna os estados resultantes de tomar qualquer vôo programado (talvez especificado com mais detalhes pela classe e pela posição da poltrona) que parte depois da hora atual somada ao tempo de trânsito no aeroporto, desde o aeroporto atual até outro.
- **Teste de objetivo:** Estamos no destino após algum tempo previamente especificado?
- **Custo de caminho:** Depende do custo monetário, do tempo de espera, do tempo de vôo, dos procedimentos alfandegários e de imigração, da qualidade da poltrona, da hora do dia, do tipo de aeronave, dos prêmios por milhagem em vôos freqüentes e assim por diante.

Os sistemas comerciais de informações para viagens utilizam uma formulação de problema desse tipo, com muitas complicações adicionais para manipular as estruturas bizantinas de tarifas que as empresas aéreas impõem. Porém, qualquer viajante experiente sabe que nem toda viagem aérea transcorre de acordo com os planos. Um sistema realmente bom deve incluir planos de contingência – como reservas substitutas em vôos alternativos – até o ponto em que esses planos possam ser justificado sem função do custo e pela probabilidade de fracasso no plano original.

PROBLEMAS DE TOUR

Os **problemas de tour** estão estreitamente relacionados aos problemas de roteamento, mas apresentam uma importante diferença. Por exemplo, considere o problema: "Visitar cada cidade da Figura 3.2 pelo menos uma vez, começando e terminando em Bucareste." Como ocorre com o roteamento, as ações correspondem a viagens entre cidades adjacentes. Porém, o espaço de estados é bastante diferente. Cada estado deve incluir não apenas a posição atual, mas também o *conjunto de cidades que o agente visitou*. Assim, o estado inicial seria "Em Bucareste; Visitado {Bucareste}", um estado intermediário típico seria "Em Vaslui; Visitado {Bucareste,Urziceni,Vaslui}", o teste de objetivo verificaria se o agente está em Bucareste e se todas as 20 cidades foram visitadas.

PROBLEMA DO CAIXEIRO-VIAJANTE

O **problema do caixeiro-viajante** (PCV) é um problema de tour em que cada cidade deve ser visitada exatamente uma vez. O objetivo é encontrar o percurso *mais curto*. O problema é conhecido por ser NP-difícil, mas um grande esforço tem sido despendido para melhorar os recursos de algoritmos de PCV. Além de planejar viagens para caixeiros-viajantes, esses algoritmos são usados para tarefas como planejar movimentos de máquinas automáticas para perfuração de placas de circuitos e de máquinas industriais em fábricas.

LAYOUT DE VLSI

Um problema de **layout de VLSI** exige o posicionamento de milhões de componentes e conexões em um chip para minimizar a área, minimizar retardos de circuitos, minimizar capacitâncias de fuga e maximizar o rendimento industrial. O problema de layout vem depois da fase de projeto lógico, e normalmente se divide em duas partes: **layout de células** e **roteamento de canais**. No layout de células, os componentes primitivos do circuito são agrupados em células, cada uma das quais executa alguma função reconhecida. Cada célula tem uma área ocupada fixa (tamanho e forma) e exige um certo número de conexões para cada uma das outras células. O objetivo é dispor as células no chip de tal forma que elas não se sobreponham e exista espaço para que os fios de conexão sejam colocados entre as células. O roteamento de canais encontra uma rota específica para cada fio passando pelos espaços vazios entre as células. Esses problemas de busca são extremamente complexos, mas sem dúvida vale a pena resolvê-los. No Capítulo 4, veremos alguns algoritmos capazes de solucioná-los.

NAVEGAÇÃO DE ROBÔS

A **navegação de robôs** é uma generalização do problema de roteamento descrito antes. Em vez de um conjunto discreto de rotas, um robô pode se mover em um espaço contínuo com (em princípio) um conjunto infinito de ações e estados possíveis. No caso de um robô em movimento circular sobre uma superfície plana, o espaço é essencialmente bidimensional. Quando o robô tem braços e pernas ou ro-

das que também devem ser controlados, o espaço de busca passa a ter várias dimensões. São exigidas técnicas avançadas apenas para tornar finito o espaço de busca. Examinaremos alguns desses métodos no Capítulo 25. Além da complexidade do problema, robôs reais também devem lidar com erros nas leituras de seus sensores e nos controles do motor.

SEQÜÊNCIA AUTOMÁTICA DE MONTAGEM

A **seqüência automática de montagem** de objetos complexos por um robô foi demonstrada primeiramente por FREDDY (Michie, 1972). Desde então, o progresso tem sido lento mas seguro, até chegar ao ponto em que a montagem de objetos complexos como motores elétricos se torna economicamente viável. Em problemas de montagem, o objetivo é encontrar uma ordem na qual devem ser montadas as peças de algum objeto. Se for escolhida a ordem errada, não haverá como acrescentar alguma peça mais adiante na seqüência sem desfazer uma parte do trabalho já realizado. A verificação da viabilidade de um passo na seqüência é um problema geométrico de busca difícil, intimamente relacionado à navegação de robôs. Desse modo, a geração de sucessores válidos é a parte dispendiosa da seqüência de montagem. Qualquer algoritmo prático deve evitar explorar mais do que uma fração minúscula desse espaço de estados. Outro problema de montagem importante é o **projeto de proteínas**, em que o objetivo é encontrar uma seqüência de aminoácidos que serão incorporados em uma proteína tridimensional com as propriedades adequadas para curar alguma doença.

PROJETO DE PROTEÍNAS

PESQUISAS NA INTERNET

Recentemente, houve um aumento da demanda por robôs de software que executam **pesquisas na Internet**, procurando respostas para perguntas, informações inter-relacionadas ou oportunidades de compras. Essa é uma boa aplicação para técnicas de busca, porque é fácil conceituar a Internet como um grafo de nós (páginas) conectados por links. Uma descrição completa da busca na Internet será apresentada no Capítulo 10.

3.3 Em busca de soluções

ÁRVORE DE BUSCA

Depois de formular alguns problemas, agora precisamos resolvê-los. Isso é feito por meio de uma busca em todo o espaço de estados. Este capítulo lida com técnicas de busca que utilizam uma **árvore de busca** explícita, gerada pelo estado inicial e pela função sucessor que, juntos, definem o espaço de estados. Em geral, podemos ter um *grafo* de busca em lugar de uma *árvore* de busca, quando o mesmo estado pode ser alcançado a partir de vários caminhos. Adiaremos a consideração dessa importante complicação até a Seção 3.5.

NÓ DE BUSCA

EXPANSÃO

GERAÇÃO

A Figura 3.6 mostra algumas das expansões na árvore de busca para encontrar uma rota de Arad até Bucareste. A raiz da árvore de busca é um **nó de busca** correspondente ao estado inicial, *Em(Arad)*. O primeiro passo é testar se esse é um estado objetivo. É claro que não é, mas é importante verificar isso, a fim de podermos resolver problemas complicados como "Partindo de Arad, chegar a Arad". Como esse não é um estado objetivo, precisamos considerar alguns outros estados. Isso é feito por **expansão** do estado atual; ou seja, aplicando-se a função sucessor ao estado atual, **gerando** assim um novo conjunto de estados. Nesse caso, alcançamos três novos estados: *Em(Sibiu)*, *Em(Timisoara)* e *Em(Zerind)*. Agora temos de escolher qual dessas três possibilidades merece consideração adicional.

Essa é a essência da busca – seguir uma opção agora e deixar as outras reservadas para mais tarde, no caso de a primeira escolha não levar a uma solução. Vamos supor que escolhemos primeiro Sibiu. Verificamos se ela é um estado objetivo (não é) e depois a expandimos para obter *Em(Arad)*, *Em(Fagaras)*, *Em(Oradea)* e *Em(RimnicuVilcea)*. Portanto, podemos escolher qualquer dessas quatro opções ou então voltar e escolher Timisoara ou Zerind. Continuamos a escolher, testar e expandir até ser encontrada uma solução ou não existirem mais estados a serem expandidos. A escolha de qual estado expandir é determinada pela **estratégia de busca**. O algoritmo geral de busca em árvore é descrito informalmente na Figura 3.7.

ESTRATÉGIA DE BUSCA

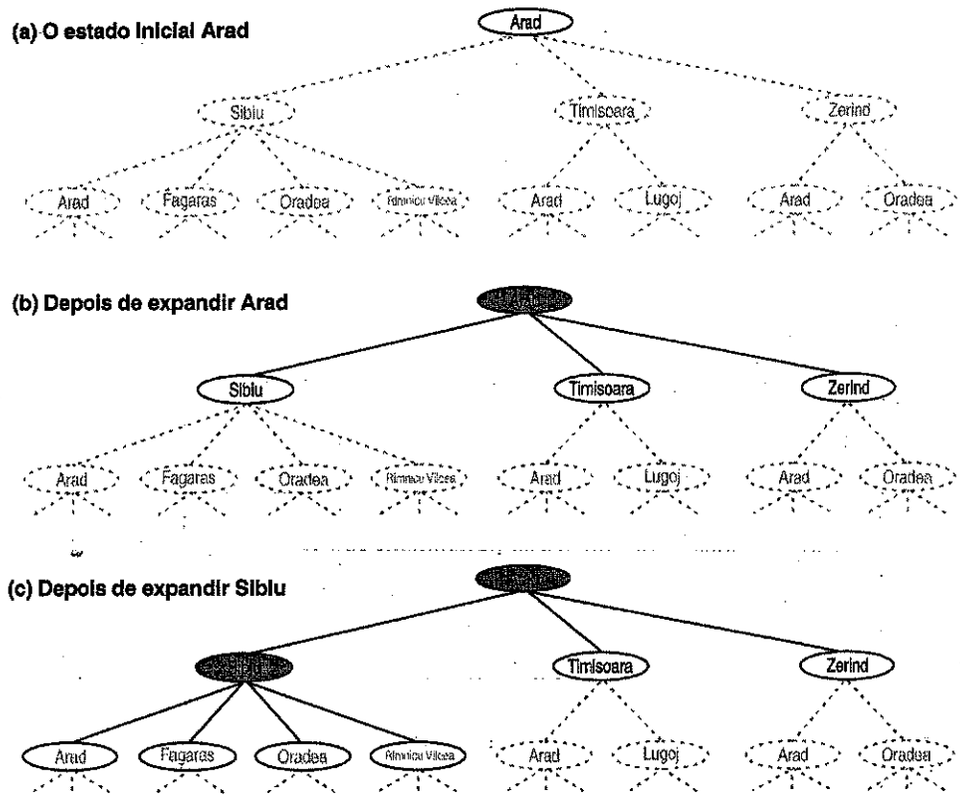


Figura 3.6 Árvores de busca parciais para localização de uma rota desde Arad até Bucareste. Nós que foram expandidos estão sombreados; nós que foram gerados mas ainda não foram expandidos têm um contorno em negrito; nós que ainda não foram gerados são mostrados em linhas leves tracejadas.

```
função BUSCA-EM-ÁRVORE(problema, estratégia) retorna uma solução ou falha
    inicializar a árvore de busca usando o estado inicial de problema
    repita
        se não existe nenhum candidato para expansão então retornar falha
        escolher um nó folha para expansão de acordo com estratégia
        se o nó contém um estado objetivo então retornar a solução correspondente
        senão expandir o nó e adicionar os nós resultantes à árvore de busca
```

Figura 3.7 Uma descrição informal do algoritmo geral de busca em árvore.

É importante distinguir entre o espaço de estados e a árvore de busca. No caso do problema de localização de rotas, existem apenas 20 estados no espaço de estados, um para cada cidade. No entanto, há um número infinito de caminhos nesse espaço de estados, e assim a árvore de busca tem um número infinito de nós. Por exemplo, os três caminhos Arad-Sibiu, Arad-Sibiu-Arad, Arad-Sibiu-Arad-Sibiu são os três primeiros de uma seqüência infinita de caminhos. (É óbvio que um bom algoritmo de busca evita seguir tais caminhos repetidos; a Seção 3.5 mostra como fazê-lo.)

Existem muitas maneiras de representar nós, mas partiremos do princípio de que um nó é uma estrutura de dados com cinco componentes:

- ESTADO: O estado no espaço de estados a que o nó corresponde.
- NÓ-PAI: O nó na árvore de busca que gerou esse nó.
- AÇÃO: A ação que foi aplicada ao pai para gerar o nó.
- CUSTO-DO-CAMINHO: O custo, tradicionalmente denotado por $g(n)$, do caminho desde o estado inicial até o nó, indicado pelos ponteiros do pai.

- PROFUNDIDADE: O número de passos ao longo do caminho, desde o estado inicial.

É importante lembrar a distinção entre nós e estados. Um nó é uma estrutura de dados de anotação usada para representar a árvore de busca. Um estado corresponde a uma configuração do mundo. Desse modo, os nós estão em caminhos específicos, definidos por ponteiros NÓ-PAI, enquanto os estados não estão. Além disso, dois nós diferentes podem conter o mesmo estado do mundo, se esse estado for gerado por meio de dois caminhos de busca diferentes. A estrutura de dados nó é representada na Figura 3.8.

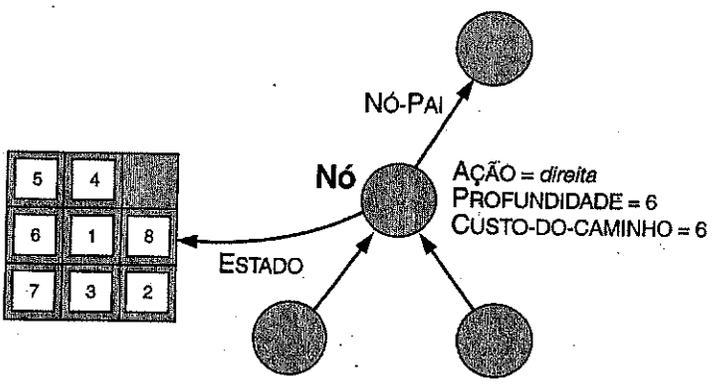


Figura 3.8 Nós são as estruturas de dados a partir das quais a árvore de busca é construída. Cada nó tem um pai, um estado e diversos campos de anotação. Setas apontam do filho para o pai.

BORDA
NÓ FOLHA

Também precisamos representar a coleção de nós que foram gerados, mas ainda não expandidos — essa coleção é chamada **borda**. Cada elemento da borda é **um nó folha**, isto é, um nó sem sucessores na árvore. Na Figura 3.6, a borda de cada árvore consiste nesses nós com contornos em negrito. A representação mais simples da borda seria um conjunto de nós. Então, a estratégia de busca seria uma função que selecionasse o próximo nó a ser expandido desse conjunto. Embora seja conceitualmente simples, isso poderia ser dispendioso em termos computacionais, porque a função de estratégia poderia ser obrigada a examinar todo elemento do conjunto para escolher o melhor. Portanto, vamos supor que a coleção de nós é implementada como uma **fila**. As operações sobre uma fila são:

FILA

- CRIAR-FILA(*elemento, ...*) cria uma fila com o(s) elemento(s) dado(s).
- VAZIA?(*fila*) retorna verdadeiro somente se não existir mais nenhum elemento na fila.
- PRIMEIRO(*fila*) retorna o primeiro elemento da fila.
- REMOVER-PRIMEIRO(*fila*) retorna PRIMEIRO(*fila*) e o remove da fila.
- INSERIR(*elemento, fila*) insere um elemento na fila e retorna a fila resultante. (Veremos que diferentes tipos de filas inserem elementos em ordem distinta.)
- INSERIR-TODOS(*elementos, fila*) insere um conjunto de elementos na fila e retorna a fila resultante.

Com essas definições, podemos escrever a versão mais formal do algoritmo geral de busca em árvore, mostrado na Figura 3.9.

Medição do desempenho de resolução de problemas

A saída de um algoritmo de resolução de problemas consiste em *falha* ou em uma solução. (Alguns algoritmos podem ficar paralisados em um laço de repetição infinito e nunca retornar uma saída.) Avaliaremos o desempenho do algoritmo em quatro aspectos:

```
função BUSCA-EM-ÁRVORE(problema,borda) retorna uma solução ou falha

  borda ← INSERIR(CRIAR-NÓ(ESTADO-INICIAL[problema]), borda)
  repita
     se VAZIA?(borda) então retornar falha
      nó ← REMOVER-PRIMEIRO(borda)
     se TESTAR-OBJETIVO[problema] aplicado a ESTADO[nó] tem sucesso
        então retornar SOLUÇÃO(nó)
     borda ← INSERIR-TODOS(EXPANDIR(nó, problema), borda)

função EXPANDIR(nó, problema) retorna um conjunto de nós

  sucessores ← o conjunto vazio
  para cada ⟨ação, resultado⟩ em SUCESSOR [problema](ESTADO[nó]) faça
     s ← um novo NÓ
     ESTADO[s] ← resultado
     NÓ-PAI[s] ← nó
     AÇÃO[s] ← ação
     CUSTO-DO-CAMINHO[s] ← CUSTO-DO-CAMINHO [nó] + CUSTO-DO-PASSO(nó, ação, s)
     PROFUNDIDADE[s] ← PROFUNDIDADE[nó] + 1
     adicionar s a sucessores
  retornar sucessores
```

Figura 3.9 O algoritmo geral de busca em árvore. (Observe que o argumento *borda* deve ser uma fila vazia, e o tipo da fila afetará a ordem da busca.) A função SOLUÇÃO retorna a seqüência de ações obtida seguindo-se os ponteiros para o nó pai de volta até a raiz.

COMPLETEZA

- **Completeza:** O algoritmo oferece a garantia de encontrar uma solução quando ela existir?

OTIMIZAÇÃO

- **Otimização:** A estratégia encontra a solução ótima, como definida na página 64?

COMPLEXIDADE DE TEMPO

- **Complexidade de tempo:** Quanto tempo ele leva para encontrar uma solução?

COMPLEXIDADE DE ESPAÇO

- **Complexidade de espaço:** Quanta memória é necessária para executar a busca?

A complexidade de tempo e a complexidade de espaço são sempre consideradas em relação a alguma medida da dificuldade do problema. Em ciência da computação teórica, a medida típica é o tamanho do grafo do espaço de estados, porque o grafo é visualizado como uma estrutura de dados explícita inserida no programa de busca. (O mapa da Romênia é um exemplo dessa estrutura.) Em IA, na qual o grafo é representado implicitamente pelo estado inicial e pela função sucessor e com freqüência é infinito, a complexidade é expressa em termos de três quantidades: b, o **fator de ramificação** ou número de máximo de sucessores de qualquer nó; d, a profundidade do nó objetivo menos profundo; e m, o comprimento máximo de qualquer caminho no espaço de estados.

FATOR DE RAMIFICAÇÃO

Com freqüência, o tempo é medido em termos do número de nós gerados[5] durante a busca, e o espaço é medido em termos do número de máximo de nós armazenados na memória.

CUSTO DE BUSCA

CUSTO TOTAL

Para avaliar a efetividade de um algoritmo de busca, podemos considerar apenas **o custo de busca** – que em geral depende da complexidade de tempo, mas também pode incluir um termo para uso da memória – ou podemos usar o **custo total**, que combina o custo de busca e o custo de caminho da so-

5. Alguns textos medem o tempo em termos do número de *expansões* de nós. As duas medidas diferem por no máximo um fator b. Parece que o tempo de execução de uma expansão de nó aumenta com o número de nós gerados nessa expansão.

lução encontrada. Para o problema de localizar uma rota desde Arad até Bucareste, o custo de busca é o período de tempo exigido pela busca, e o custo de solução é o comprimento total do caminho em quilômetros. Desse modo, para calcular o custo total, temos de somar quilômetros e milissegundos. Não existe nenhuma "taxa de câmbio oficial" entre essas duas unidades de medida mas, nesse caso, talvez fosse razoável converter quilômetros em milissegundos usando uma estimativa da velocidade média do carro (porque o tempo é o que importa ao agente). Isso permite ao agente encontrar um ponto de equilíbrio ótimo, no qual se torna contraproducente realizar computação adicional para encontrar um caminho mais curto. O problema mais geral de compensação entre diferentes recursos será examinado no Capítulo 16.

3.4 Estratégias de busca sem informação

BUSCA SEM INFORMAÇÃO

BUSCA COM INFORMAÇÃO

BUSCA HEURÍSTICA

Esta seção focaliza cinco estratégias de busca reunidas sob o título de **busca sem informação** (também chamada **busca cega**). A expressão significa que elas não têm nenhuma informação adicional sobre estados, além daquelas fornecidas na definição do problema. Tudo o que elas podem fazer é gerar sucessores e distinguir um estado objetivo de um estado não-objetivo. As estratégias que sabem se um estado não-objetivo é "mais promissor" que outro são chamadas estratégias de **busca com informação** ou **busca heurística**; elas serão estudadas no Capítulo 4. Todas as estratégias de busca se distinguem pela *ordem* em que os nós são expandidos.

Busca em extensão

BUSCA EM EXTENSÃO

A **busca em extensão** é uma estratégia simples em que o nó raiz é expandido primeiro, em seguida todos os sucessores do nó raiz são expandidos, depois os sucessores *desses nós* e assim por diante. Em geral, todos os nós em uma dada profundidade na árvore de busca são expandidos, antes que todos os nós no nível seguinte sejam expandidos.

A busca em extensão pode ser implementada chamando-se BUSCA-EM-ÁRVORE com uma borda vazia que seja uma fila do tipo first-in-first-out (FIFO), assegurando-se que os nós visitados primeiro serão expandidos primeiro. Em outras palavras, a chamada de BUSCA-EM-ÁRVORE(*problema*,FILA-FIFO( )) resulta em uma busca em extensão. A fila FIFO coloca todos os sucessores recém-gerados no final da fila, o que significa que os nós de baixa profundidade serão expandidos antes de nós mais profundos. A Figura 3.10 mostra o progresso da busca em uma árvore binária simples.

Avaliaremos a busca em extensão usando os quatro critérios da seção anterior. Podemos ver facilmente que ela é *completa* – se o nó objetivo mais raso estiver em alguma profundidade finita d, a busca em extensão eventualmente o encontrará após expandir todos os nós mais rasos (desde que o fator de ramificação b seja finito). O nó objetivo *mais raso* não é necessariamente o nó *ótimo*; tecnicamente, a busca em extensão será ótima se o custo de caminho for uma função não-decrescente da profundidade do nó. (Por exemplo, quando todas as ações tiverem o mesmo custo.)

Até aqui, o que se falou sobre busca em extensão têm sido boas notícias. Para ver por que nem sempre essa estratégia é a preferida, temos de considerar a quantidade de tempo e memória que ela emprega para completar uma busca. Para isso, consideramos um espaço de estados hipotético no qual cada estado tem b sucessores. A raiz da árvore de busca gera b nós no primeiro nível, cada um dos quais gera b outros nós, totalizando b^2 no segundo nível. Cada um *desses* outros nós gera b outros nós, totalizando b^3 nós no terceiro nível e assim por diante. Agora, suponha que a solução esteja na profundidade d. No pior caso, expandiríamos todos os nós exceto o último do nível d (pois o objetivo propriamente dito não é expandido), gerando $b^{d+1} - b$ nós no nível $d + 1$. Então, o número total de nós gerados é:

$$b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

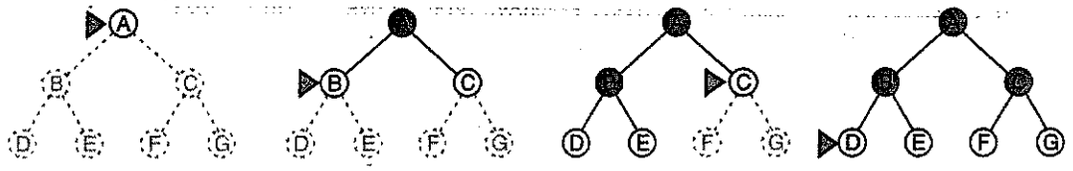


Figura 3.10 Busca em extensão em uma árvore binária simples. Em cada fase, o nó a ser expandido em seguida é indicado por um marcador.

Todo nó gerado deve permanecer na memória, porque faz parte da borda ou é um ancestral de um nó de borda. Portanto, a complexidade de espaço é igual à complexidade de tempo (mais um nó para a raiz).

Os leitores que realizam a análise da complexidade estão preocupados (ou excitados, se gostam de desafios) pelo fato de a complexidade exponencial ter limites como $O(b^{d+1})$. A Figura 3.11 mostra por que isso acontece. Ela lista o tempo e a memória exigidos para uma busca em extensão com fator de ramificação $b = 10$, para diversos valores da profundidade de solução d. A tabela pressupõe que 10.000 nós podem ser gerados por segundo, e que um nó exige 1.000 bytes de espaço de armazenamento. Muitos problemas de busca se enquadram aproximadamente nessas suposições (dentro de um intervalo de 100 vezes) quando são executados em um computador pessoal moderno.

Profundidade	Nós	Tempo	Memória
2	1100	0,11 segundo	1 megabyte
4	111.100	11 segundos	106 megabytes
6	10^7	19 minutos	10 gigabytes
8	10^9	31 horas	1 terabyte
10	10^{11}	129 dias	101 terabytes
12	10^{13}	35 anos	10 petabytes
14	10^{15}	3.523 anos	1 exabyte

Figura 3.11 Requisitos de tempo e memória para a busca em extensão. Os números mostrados pressupõem um fator de ramificação $b = 10$, 10.000 nós/segundo, 1.000 bytes/nó.

Existem duas lições a serem aprendidas a partir da Figura 3.11. Em primeiro lugar, *os requisitos de memória são um problema maior para a busca em extensão do que o tempo de execução*. Trinta e uma horas não seria tempo demais para se esperar pela solução de um problema importante de profundidade 8, mas poucos computadores têm a memória principal da ordem de terabytes que ele exigiria. Felizmente, existem outras estratégias de busca que exigem menos memória.

A segunda lição é que os requisitos de tempo ainda constituem um fator importante. Se seu problema tem uma solução na profundidade 12, então (dadas nossas suposições) ele demorará 35 anos para que a busca em extensão a encontre (ou, na realidade, qualquer busca sem informação). Em geral, *os problemas de busca de complexidade exponencial não podem ser resolvidos por métodos sem informação, para qualquer instância, exceto as menores.*

Busca de custo uniforme

BUSCA DE CUSTO UNIFORME

A busca em extensão é ótima quando os custos de todos os passos são iguais, porque sempre expande o nó *mais raso* não-expandido. Através de uma simples extensão, podemos encontrar um algoritmo que é ótimo para qualquer função de custo de passo. Em vez de expandir o nó mais raso, a **busca de custo uniforme** expande o nó n com o *caminho de custo mais baixo*. Observe que, se todos os custos de passos forem iguais, essa busca será idêntica à busca em extensão.

A busca de custo uniforme não se importa com o *número* de passos que um caminho tem, mas apenas com o seu custo total. Por isso, ela ficará paralisada em um laço de repetição infinito se expandir um nó que tenha uma ação de custo zero levando de volta ao mesmo estado (por exemplo, uma ação *Nula*). Podemos garantir a completeza, desde que o custo de cada passo seja maior ou igual a alguma constante positiva pequena ε. Essa condição também é suficiente para assegurar o *caráter ótimo*. Ele significa que o custo de um caminho sempre aumenta à medida que percorremos o caminho. A partir dessa propriedade, é fácil ver que o algoritmo expande os nós em ordem de custo de caminho crescente. Portanto, o primeiro nó objetivo selecionado para expansão é a solução ótima. (Lembre-se de que BUSCA-EM-ÁRVORE aplica o teste de objetivo apenas aos nós selecionados para expansão.) Recomendamos experimentar o algoritmo para encontrar o caminho mais curto até Bucareste.

A busca de custo uniforme é orientada por custos de caminhos em vez de profundidades; assim, sua complexidade não pode ser caracterizada com facilidade em termos de b e d. Em vez disso, seja C^* o custo da solução ótima, e suponha que toda ação custe pelo menos ε. Então, a complexidade de tempo e espaço do pior caso do algoritmo é $O(b^{\lceil C^*/\varepsilon \rceil})$, que pode ser muito maior que b^d. Essa é a razão por que a busca de custo uniforme pode explorar, e freqüentemente explora, grandes árvores de pequenos passos antes de explorar caminhos envolvendo passos grandes e talvez úteis. Quando todos os custos de passos forem iguais, é claro que $b^{\lceil C^*/\varepsilon \rceil}$ será simplesmente b^d.

Busca em profundidade

BUSCA EM PROFUNDIDADE

A **busca em profundidade** sempre expande o nó *mais profundo* na borda atual da árvore de busca. O progresso da busca é ilustrado na Figura 3.12. A busca prossegue imediatamente até o nível mais profundo da árvore de busca, onde os nós não têm sucessores. À medida que esses nós são expandidos, eles são retirados da borda, e então a busca "retorna" ao nó seguinte mais raso que ainda tem sucessores inexplorados.

Essa estratégia pode ser implementada por BUSCA-EM-ÁRVORE com uma fila last-in-first-out (LIFO), também conhecida como pilha. Como uma alternativa à implementação de BUSCA-EM-ÁRVORE, é comum implementar a busca em profundidade com uma função recursiva que chama a si mesma sucessivamente em cada um de seus filhos. (Um algoritmo recursivo em profundidade incorporando um limite de profundidade é ilustrado na Figura 3.13.)

A busca em profundidade tem requisitos de memória muito modestos. Ela só precisa armazenar um único caminho da raiz até um nó de folha, juntamente com os nós irmãos não-expandidos restantes de cada nó no caminho. Uma vez que um nó é expandido, ele pode ser removido da memória, tão logo todos os seus descendentes tenham sido completamente explorados. (Veja a Figura 3.12.) Para um espaço de estados com fator de ramificação b e profundidade máxima m, a busca em profundidade exige o armazenamento de apenas $bm + 1$ nós. Usando as mesmas suposições da Figura 3.11 e supondo que nós na mesma profundidade do nó objetivo não têm sucessores, descobrimos que a busca em profundidade exigiria 118 kilobytes, em vez de 10 petabytes na profundidade $d = 12$, um espaço 10 bilhões de vezes menor.

BUSCA COM RETROCESSO

Uma variante de busca em profundidade chamada **busca com retrocesso** utiliza ainda menos memória. No retrocesso, apenas um sucessor é gerado de cada vez, em lugar de todos os sucessores; cada nó parcialmente expandido memoriza o sucessor que deve gerar em seguida. Desse modo, é necessária apenas a memória $O(m)$, em vez de $O(bm)$. A busca com retrocesso permite ainda um outro truque de economia de memória (e de economia de tempo): a idéia de gerar um sucessor pela modificação direta da descrição do estado atual, em vez de copiá-lo primeiro. Isso reduz os requisitos de memória a apenas uma descrição de estado e a $O(m)$ ações. Para que isso funcione, devemos ser capazes de desfazer cada modificação quando voltarmos para gerar o próximo sucessor.

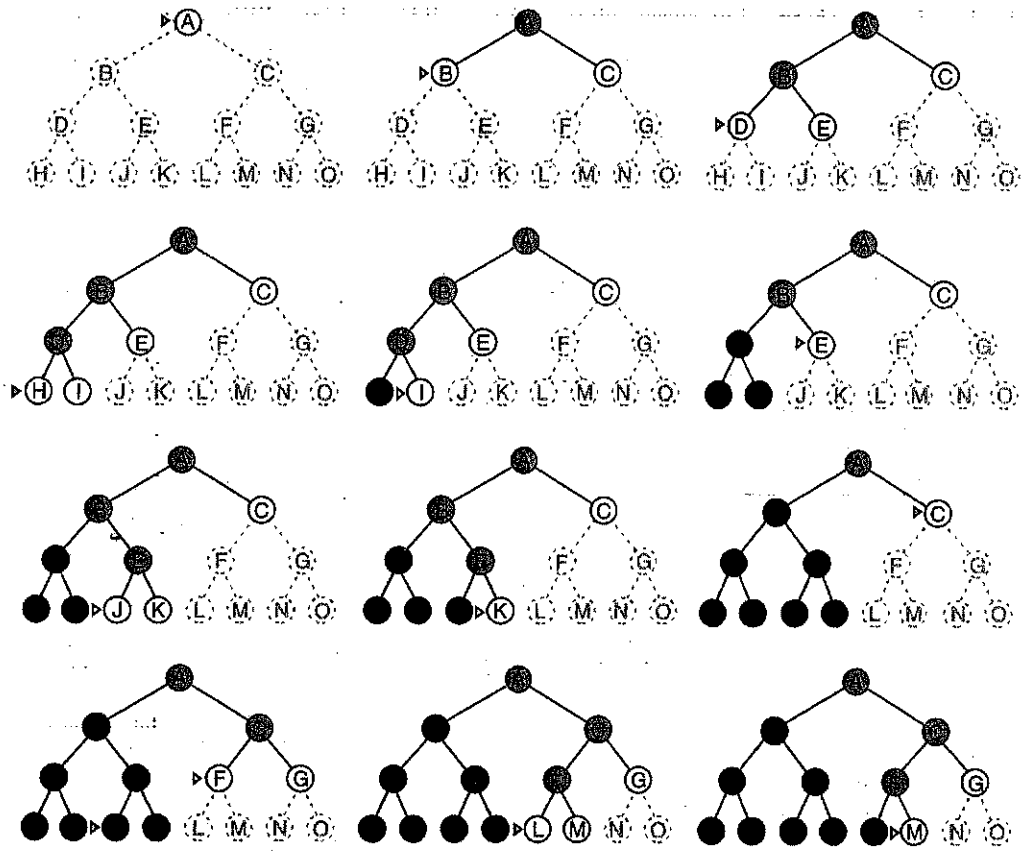


Figura 3.12 Busca em profundidade em uma árvore binária. Os nós que foram expandidos e não têm descendentes na borda podem ser removidos da memória; eles são mostrados em cor preta. Supomos que os nós na profundidade 3 não têm sucessores e que *M* é o único nó objetivo.

```
função BUSCA-EM-PROFUNDIDADE-LIMITADA(problema, limite) retorna uma solução ou falha/corte
  retornar BPL-RECURSIVA(CRIAR-NÓ(ESTADO-INICIAL[problema]), problema, limite)

função BPL-RECURSIVA(nó, problema, limite) retorna uma solução ou falha/corte
  corte_ocorreu? ← falso
  se TESTAR-OBJETIVO[problema](ESTADO[nó]) então retornar SOLUÇÃO(nó)
  senão se PROFUNDIDADE[nó] = limite então retornar corte
  senão para cada sucessor em EXPANDIR(nó, problema) faça
    resultado ← BPL-RECURSIVA(sucessor, problema, limite)
    se resultado = corte então corte_ocorreu? ← verdadeiro
    senão se resultado ≠ falha então retornar resultado
  se corte_ocorreu? então retornar corte senão retornar falha
```

Figura 3.13 Uma implementação recursiva da busca em profundidade limitada.

No caso de problemas com grandes descrições de estados, como a montagem robótica, essas técnicas são críticas para o sucesso.

A desvantagem da busca em profundidade é que ela pode fazer uma escolha errada e ficar paralisada ao descer um caminho muito longo (ou mesmo infinito), quando uma opção diferente levaria a uma solução próxima à raiz da árvore de busca. Por exemplo, na Figura 3.12, a busca em profundidade explorará toda a subárvore da esquerda, mesmo que o nó *C* seja um nó objetivo. Se o nó *J* também fosse um nó objetivo, a busca em profundidade o retornaria como uma solução; conseqüentemente, a

busca em profundidade não é ótima. Se a subárvore da esquerda tivesse uma profundidade ilimitada mas não contivesse nenhuma solução, a busca em profundidade nunca terminaria; desse modo, ela não é completa. No pior caso, a busca em profundidade irá gerar todos os $O(b^m)$ nós na árvore de busca, onde m é a profundidade máxima de qualquer nó. Observe que m pode ser muito maior que d (a profundidade da solução mais rasa) e será infinita se a árvore for ilimitada.

Busca em profundidade limitada

BUSCA EM PROFUNDIDADE LIMITADA

O problema de árvores ilimitadas pode ser atenuado pela busca em profundidade com um limite de profundidade predeterminado ℓ. Isto é, nós na profundidade ℓ são tratados como se não tivessem sucessores. Essa abordagem é chamada **busca em profundidade limitada**. O limite de profundidade resolve o problema de caminhos infinitos. Infelizmente, ele também introduz uma fonte adicional de incompleteza, se escolhermos $\ell < d$, ou seja, o objetivo mais raso está além do limite de profundidade. (Isso não é improvável quando d é desconhecido.) A busca em profundidade limitada também não será ótima se escolhermos $\ell > d$. Sua complexidade de tempo é $O(b^\ell)$ e sua complexidade de espaço é $O(b\ell)$. A busca em profundidade pode ser visualizada como um caso especial da busca em profundidade limitada com $\ell = \infty$.

Às vezes, limites de profundidade podem se basear no conhecimento que se tem sobre o problema. Por exemplo, no mapa da Romênia há 20 cidades. Portanto sabemos que, se existe uma solução, ela deve ter o comprimento 19 no caso mais longo, e então $\ell = 19$ é uma escolha possível. Porém, de fato, se estudássemos cuidadosamente o mapa, descobriríamos que qualquer cidade pode ser alcançada a partir de qualquer outra cidade em no máximo 9 passos. Esse número, conhecido como **diâmetro** do espaço de estados, nos dá um limite de profundidade melhor, o que leva a uma busca em profundidade limitada mais eficiente. No entanto, na maioria dos problemas não conhecemos um bom limite de profundidade antes de resolvermos o problema.

DIÂMETRO

A busca em profundidade limitada pode ser implementada como uma modificação simples do algoritmo geral de busca em árvore ou do algoritmo de busca recursiva em profundidade. Mostramos o pseudocódigo para a busca recursiva em profundidade limitada na Figura 3.13. Observe que a busca em profundidade limitada pode terminar com dois tipos de falhas: o valor padrão *falha* indica nenhuma solução; o valor *corte* indica nenhuma solução dentro do limite de profundidade.

Busca de aprofundamento iterativo em profundidade

BUSCA POR APROFUNDAMENTO ITERATIVO

A **busca por aprofundamento iterativo** (ou busca em profundidade por aprofundamento iterativo) é uma estratégia geral, usada com freqüência em combinação com a busca em profundidade, que encontra o melhor limite de profundidade. Ela faz isso aumentando gradualmente o limite – primeiro 0, depois 1, depois 2 e assim por diante – até encontrar um objetivo. Isso ocorrerá quando o limite de profundidade alcançar d, a profundidade do nó objetivo mais raso. O algoritmo é mostrado na Figura 3.14. O aprofundamento iterativo combina os benefícios da busca em profundidade e da busca em extensão. Como na busca em profundidade, seus requisitos de memória são muito modestos: $O(bd)$, para sermos precisos. Como na busca em extensão, ele é completo quando o fator de ramificação é finito, e ótimo quando o custo de caminho é uma função não-decrescente da profundidade do nó. A Figura 3.15 mostra quatro iterações de BUSCA-POR-APROFUNDAMENTO-ITERATIVO em uma árvore de busca binária, onde a solução é encontrada na quarta iteração.

A busca por aprofundamento iterativo pode parecer um desperdício, porque os estados são gerados várias vezes. Na verdade, esse custo não é muito alto porque, em uma árvore de busca com o mesmo (ou quase o mesmo) fator de ramificação em cada nível, a maior parte dos nós estará no nível infe-

```
função BUSCA-POR-APROFUNDAMENTO-ITERATIVO(problema) retorna uma solução ou falha
  entradas: problema, um problema

  para profundidade ← 0 até ∞ faça
    resultado ← BUSCA-EM-PROFUNDIDADE-LIMITADA(problema, profundidade)
    se resultado ≠ corte então retornar resultado
```

Figura 3.14 Algoritmo de busca por aprofundamento iterativo que aplica repetidamente a busca em profundidade limitada com limites crescentes. Ele termina quando uma solução é encontrada ou se a busca em profundidade limitada retorna *falha*, indicando que não existe nenhuma solução.

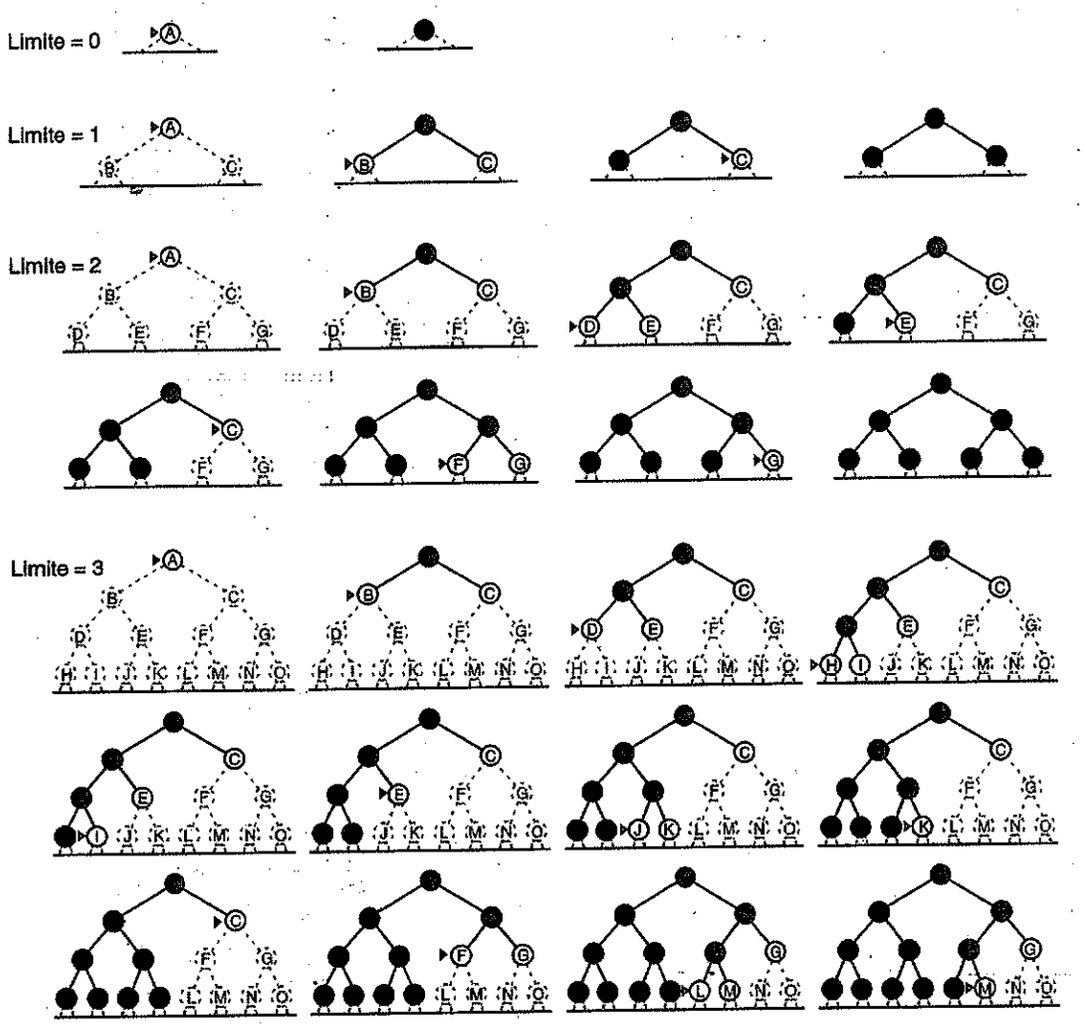


Figura 3.15 Quatro iterações de busca por aprofundamento iterativo em uma árvore binária.

rior, e assim não importa muito se os níveis superiores são gerados várias vezes. Em uma busca por aprofundamento iterativo, os nós no nível inferior (profundidade d) são geradas uma vez, os do penúltimo nível inferior são gerados duas vezes e assim por diante, até os filhos da raiz, que são gerados d vezes. Portanto, o número total de nós gerados é:

$$N(\text{BAI}) = (d)b + (d-1)b^2 + \ldots + (1)b^d$$

O que dá uma complexidade de tempo igual a $O(b^d)$. Podemos comparar esse valor aos nós gerados por uma busca em extensão:

$$N(\text{BE}) = b + b^2 + \ldots + b^d + (b^{d+1} - \text{b}).$$

Note que a busca em extensão gera alguns nós na profundidade $d + 1$, enquanto o aprofundamento iterativo não gera esses nós. Como resultado, o aprofundamento iterativo é de fato *mais rápido* que a busca em extensão, apesar da geração repetida de estados. Por exemplo, se $b = 10$ e $d = 5$, os números são:

$$N(\text{BAI}) = 50 + 400 + 3.000 + 20.000 + 100.000 = 123.450$$
$$N(\text{BE}) = 10 + 100 + 1.000 + 10.000 + 100.000 + 999.990 = 1.111.100.$$

Em geral, o aprofundamento iterativo é o método de busca sem informação preferido quando existe um espaço de busca grande e a profundidade da solução não é conhecida.

A busca por aprofundamento iterativo é análoga à busca em extensão, pelo fato de explorar uma camada completa de novos nós em cada iteração, antes de passar para a próxima camada. Seria interessante desenvolver um algoritmo iterativo análogo à busca de custo uniforme, herdando as garantias de caráter ótimo do algoritmo anterior, ao mesmo tempo em que se reduz suas necessidades de memória. A idéia é usar limites crescentes de custo de caminho, em vez de limites crescentes de profundidade. O algoritmo resultante, chamado **busca por alongamento iterativo**, é explorado no Exercício 3.11. Infelizmente, o alongamento iterativo incorre em uma sobrecarga substancial, em comparação com a busca de custo uniforme.

BUSCA POR ALONGAMENTO ITERATIVO

Busca bidirecional

A idéia que rege a busca bidirecional é executar duas buscas simultâneas – uma direta a partir do estado inicial, e a outra inversa a partir do objetivo, parando quando as duas buscas se encontram em um ponto intermediário (Figura 3.16). A motivação é que $b^{d/2} + b^{d/2}$ é muito menor que b^d ou, na figura, a área dos dois círculos pequenos é menor que a área do único círculo grande com centro no início e que chega até o objetivo.

A busca bidirecional é implementada fazendo-se uma ou ambas as buscas verificarem cada nó antes de ele ser expandido, para ver se o nó está na borda da outra árvore de busca; nesse caso, é encontrada uma solução. Por exemplo, se um problema tem profundidade de solução $d = 6$, e se cada sentido executa a busca em extensão em um nó de cada vez, no pior caso, as duas buscas se encontrarão quando cada uma delas tiver expandido todos os nós da profundidade 3, exceto um. Para $b = 10$, isso significa um total de 22.200 gerações de nós, em comparação com 11,111.100 para uma busca em extensão padrão. A verificação da pertinência de um nó à outra árvore de busca pode ser feita em tem-

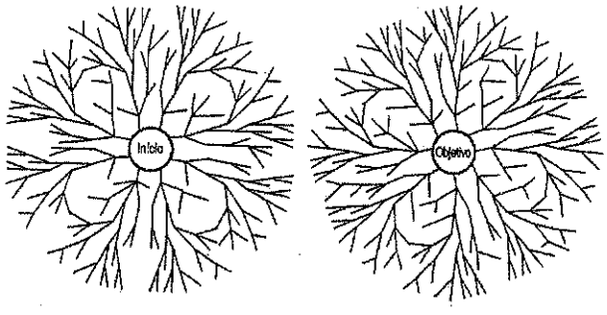


Figura 3.16 Uma visão esquemática de uma busca bidirecional prestes a ter sucesso, quando uma ramificação a partir do nó inicial encontra uma ramificação a partir do nó objetivo.

po constante com uma tabela de hash, e então a complexidade de tempo da busca bidirecional é $O(b^{d/2})$. Pelo menos uma das árvores de busca deve ser mantida na memória, de forma que a verificação da pertinência possa ser realizada; conseqüentemente, a complexidade de espaço também é $O(b^{d/2})$. Esse requisito de espaço é a deficiência mais significativa da busca bidirecional. O algoritmo é completo e ótimo (para passos de custo uniforme) se ambas as buscas forem em extensão; outras combinações podem sacrificar a completeza, o caráter ótimo ou ambos.

PREDECESSORES

A redução da complexidade de tempo torna a busca bidirecional atraente, mas como realizaremos a busca inversa? Isso não é tão fácil quanto parece. Sejam $Pred(n)$ os **predecessores** de um nó n, isto é, todos os nós que têm n como sucessor. A busca bidirecional exige que $Pred(n)$ seja computável de forma eficiente. O caso mais fácil ocorre quando todas as ações no espaço de estados são reversíveis, de forma que $Pred(n) = Succ(n)$. Outros casos podem exigir uma substancial engenhosidade.

Considere a questão do que queremos dizer com a palavra "objetivo" na frase "busca inversa a partir do objetivo". No caso do quebra-cabeça de 8 peças e da localização de uma rota na Romênia, existe apenas um estado objetivo; assim, a busca inversa é muito semelhante à busca direta. Se houver vários estados objetivo *explicitamente listados* – por exemplo, os dois estados objetivo livres de sujeira da Figura 3.3 – poderemos construir um novo estado objetivo fictício cujos predecessores imediatos serão todos os estados objetivo reais. Como alternativa, algumas gerações de nós redundantes podem ser evitadas visualizando-se o conjunto de estados objetivo como um único estado, e também cada um de seus predecessores como um conjunto de estados – especificamente, o conjunto de estados que tem um sucessor correspondente no conjunto de estados objetivo. (Veja também a Seção 3.6.)

O caso mais difícil da busca bidirecional ocorre quando o teste de objetivo só fornece uma descrição implícita de algum conjunto possivelmente extenso de estados objetivo – por exemplo, todos os estados que satisfazem ao teste de objetivo "xeque-mate" no xadrez. Uma busca inversa precisaria construir descrições compactas de "todos os estados que levam ao xeque-mate pelo movimento m_1" e assim por diante, e essas descrições teriam de ser testadas por comparação com os estados gerados pela busca direta. Não existe nenhum método geral para fazer isso de modo eficiente.

Comparação entre estratégias de busca sem informação

A Figura 3.17 compara estratégias de busca em termos dos quatro critérios de avaliação definidos na Seção 3.4.

Critério	Em extensão	Custo uniforme	Em profundidade	Em profundidade limitada	Aprofundamento iterativo	Bidirecional (se aplicável)
Completa?	Sim[a]	Sim[a,b]	Não	Não	Sim[a]	Sim[a,d]
Tempo	$O(b^{d+1})$	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Espaço	$O(b^{d+1})$	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Ótima?	Sim[c]	Sim	Não	Não	Sim[c]	Sim[c,d]

Figura 3.17 Avaliação de estratégias de busca. b é o fator de ramificação; d é a profundidade da solução mais rasa; m é a profundidade máxima da árvore de busca; l é o limite de profundidade. As anotações sobrescritas são: [a] completa se b é finito; [b] completa se o custo do passo é $\geq \epsilon$ para positivo ϵ; [c] ótima se os custos dos passos são todos idênticos; [d] se ambos os sentidos utilizam busca em extensão.

3.5 Como evitar estados repetidos

Até aqui fizemos tudo, mas ignoramos uma das complicações mais importantes do processo de busca: a possibilidade de desperdiçar tempo expandindo estados que já foram encontrados e expandidos antes. Para alguns problemas, essa possibilidade nunca surge; o espaço de estados é uma árvore e só existe um caminho até cada estado. A formulação eficiente do problema de 8 rainhas (onde cada nova rainha é colocada na coluna vazia mais à esquerda) é eficiente em grande parte devido a isso – cada estado pode ser alcançado apenas através de um caminho. Se formularmos o problema de 8 rainhas de forma que uma rainha possa ser inserida em qualquer coluna, cada estado com n rainhas poderá ser acessado por $n!$ caminhos diferentes.

Para alguns problemas, os estados repetidos são inevitáveis. Isso inclui todos os problemas em que as ações são reversíveis, como os problemas de localização de rotas e os quebra-cabeças de blocos deslizantes. As árvores de busca para esses problemas são infinitas mas, se podarmos alguns dos estados repetidos, poderemos reduzir a árvore de busca a um tamanho finito, gerando apenas a parte da árvore que inclui o grafo de espaço de estados. Considerando-se a árvore de busca apenas até uma profundidade fixa, é fácil encontrar casos em que a eliminação de estados repetidos resulta em uma redução exponencial do custo da busca. No caso extremo, um espaço de estados de tamanho $d + 1$ (Figura 3.18(a)) se torna uma árvore com 2^d folhas (Figura 3.18(b)). Um exemplo mais realista é a **malha retangular** ilustrada na Figura 3.18(c). Em uma malha, cada estado tem quatro sucessores, e assim a árvore de busca que inclui estados repetidos tem 4^d folhas; porém, existem apenas cerca de $2d^2$ estados distintos dentro de d passos de qualquer estado dado. Para $d = 20$, isso significa aproximadamente um trilhão de nós, mas apenas cerca de 800 estados distintos.

MALHA RETANGULAR

Portanto, os estados repetidos podem fazer um problema solúvel se tornar insolúvel, se o algoritmo não os detectar. Em geral, a detecção significa comparar o nó prestes a ser expandido aos nós que já foram expandidos; se for encontrada uma correspondência, isso significa que o algoritmo descobriu dois caminhos para o mesmo estado e pode descartar um deles.

Para a busca em profundidade, os únicos nós na memória são aqueles no caminho a partir da raiz até o nó atual. A comparação desses nós ao nó atual permite que o algoritmo detecte caminhos em ciclos que podem ser descartados de imediato. Isso é bom para assegurar que os espaços de estados finitos não se tornarão árvores de busca infinitas devido a ciclos; infelizmente, ela não evita a proliferação exponencial de caminhos sem ciclos em problemas como os da Figura 3.18. O único caminho para evitar isso é manter mais nós na memória. Existe uma relação inversamente proporcional fundamental entre espaço e tempo. *Algoritmos que esquecem sua história estão condenados a repeti-la.*

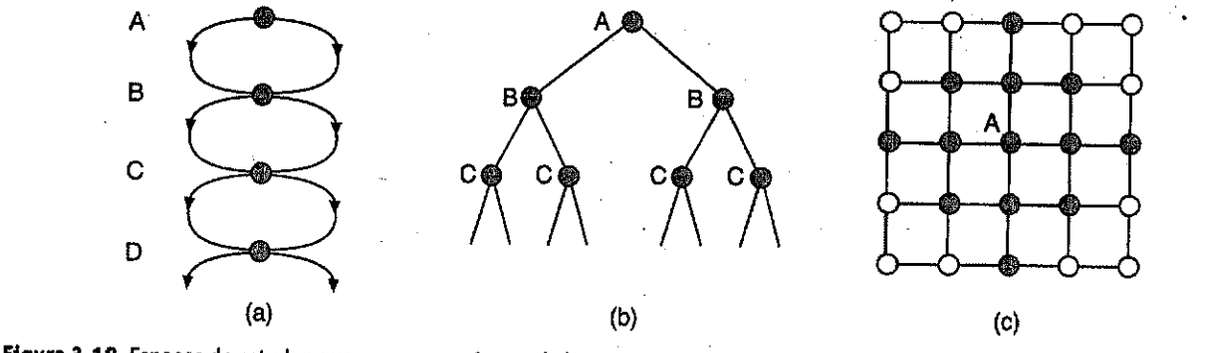


Figura 3.18 Espaços de estados que geram uma árvore de busca exponencialmente maior. (a) O espaço de estados em que existem duas ações possíveis que levam de A até B, duas de B até C e assim por diante. O espaço de estados contém $d + 1$ estados, onde d é a profundidade máxima. (b) A árvore de busca correspondente, que tem 2^d ramificações correspondentes aos 2^d caminhos pelo espaço. (c) Um espaço de malha retangular. Estados a 2 passos do estado inicial (A) são mostrados em cor cinza.

Se um algoritmo se lembrar de todo estado que visitou, ele poderá ser visualizado como a exploração direta do grafo de espaço de estados. Podemos modificar o algoritmo BUSCA-EM-ÁRVORE geral para incluir uma estrutura de dados chamada **lista fechada**, que armazena cada nó expandido. (A borda de nós não-expandidos às vezes é chamada **lista aberta**.) Se o nó atual corresponder a um nó na lista fechada, ele será descartado em vez de ser expandido. O novo algoritmo é chamado BUSCA-EM-GRAFO (Figura 3.19). Em problemas com muitos estados repetidos, BUSCA-EM-GRAFO é muito mais eficiente que BUSCA-EM-ÁRVORE. Seus requisitos de tempo e espaço do pior caso são proporcionais ao tamanho do espaço de estados. Isso pode ser muito menor que $O(b^d)$.

LISTA FECHADA
LISTA ABERTA

```
função BUSCA-EM-GRAFO(problema,borda) retorna uma solução ou falha

  fechado ← um conjunto vazio
  borda ← INSERIR(CRIAR-NÓ(ESTADO-INICIAL[problema]), borda)
  repita
    se VAZIA?(borda) então retornar falha
    nó ← REMOVER-PRIMEIRO(borda)
    se TESTAR-OBJETIVO[problema](ESTADO[nó]) então retornar SOLUÇÃO(nó)
    se ESTADO[nó] não está em fechado então
        adicionar ESTADO[nó] a fechado
        borda ← INSERIR-TODOS(EXPANDIR(nó,problema), borda)
```

Figura 3.19 O algoritmo geral de busca em grafos. O conjunto *fechado* pode ser implementado com uma tabela de hash para permitir a verificação eficiente de estados repetidos. Esse algoritmo pressupõe que o primeiro caminho até um estado *s* é o mais econômico (veja o texto).

O caráter ótimo da busca em grafos é uma questão complicada. Dissemos antes que, quando um estado repetido é detectado, isso significa que o algoritmo encontrou dois caminhos para o mesmo estado. O algoritmo BUSCA-EM-GRAFO da Figura 3.19 sempre descarta o caminho *recém-descoberto*; é óbvio que, se o caminho recém-descoberto fosse mais curto que o caminho original, BUSCA-EM-GRAFO poderia perder uma solução ótima. Felizmente, podemos mostrar (Exercício 3.12) que isso não pode acontecer quando se utiliza a busca de custo uniforme ou a busca em extensão com passos de custo constante; conseqüentemente, essas duas estratégias ótimas de busca em árvores também são estratégias ótimas de busca em grafos. Por outro lado, a busca por aprofundamento iterativo utiliza a expansão em profundidade e pode facilmente seguir um caminho não-ótimo para um nó antes de encontrar o caminho ótimo. Portanto, a busca em grafos por aprofundamento iterativo precisa verificar se um caminho recém-descoberto para um nó é melhor que o original e, se for, talvez tenha de revisar as profundidades e os custos dos caminhos dos descendentes desse nó.

Observe que o uso de uma lista fechada significa que a busca em profundidade e a busca por aprofundamento iterativo não têm mais requisito de espaço linear. Como o algoritmo BUSCA-EM-GRAFO mantém cada nó na memória, algumas buscas são inviáveis devido a limitações de memória.

3.6 Busca com informações parciais

Na Seção 3.3, supomos que o ambiente seja completamente observável e determinístico, e que o agente sabe quais são os efeitos de cada ação. Portanto, o agente pode calcular exatamente o estado que resulta de qualquer seqüência de ações e sempre sabe em que estado se encontra. Suas percepções não fornecem nenhuma informação nova depois de cada ação. O que acontece quando o conhecimento dos estados ou ações é incompleto? Descobrimos que diferentes tipos de incompleteza levam a três tipos distintos de problemas:

1. **Problemas sem sensores** (também chamados **problemas de conformidade**): Se o agente não tem nenhum sensor, então (até onde sabe) ele poderia estar em um dentre vários estados iniciais possíveis, e cada ação poderia portanto levar a um dentre vários estados sucessores possíveis.
2. **Problemas de contingência:** Se o ambiente for parcialmente observável ou se as ações forem incertas, as percepções do agente fornecerão *novas* informações depois de cada ação. Cada percepção possível define uma contingência que deve ser planejada. Um problema é chamado **adverso** se a incerteza é causada pelas ações de outro agente.
3. **Problemas de exploração**: Quando os estados e as ações do ambiente são desconhecidos, o agente deve atuar para descobri-los. Os problemas de exploração podem ser visualizados como um caso extremo de problemas de contingência.

Como um exemplo, usaremos o ambiente do mundo de aspirador de pó. Lembre-se de que o espaço de estados tem oito estados, como mostra a Figura 3.20. Existem três ações – *Esquerda*, *Direita* e *Aspirar* – e o objetivo é limpar toda a sujeira (estados 7 e 8). Se o ambiente é observável, determinístico e completamente conhecido, então o problema é solucionável de forma trivial por qualquer dos algoritmos que descrevemos. Por exemplo, se o estado inicial é 5, então a seqüência de ações [*Direita,Aspirar*] alcançará um estado objetivo, o estado 8. O restante desta seção lida com as versões sem sensores e de contingência do problema. Os problemas de exploração são abordados na Seção 4.5, e os problemas adversos no Capítulo 6.

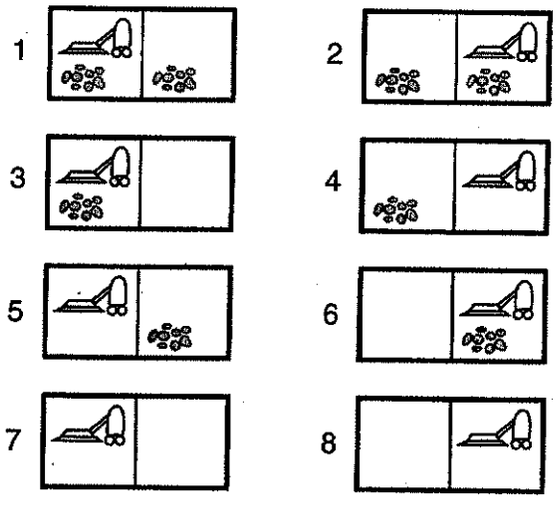


Figura 3.20 Os oito estados possíveis do mundo de aspirador de pó.

Problemas sem sensores

Suponha que o agente aspirador de pó conheça todos os efeitos de suas ações, mas não tenha nenhum sensor. Assim, ele só sabe que seu estado inicial pertence ao conjunto {1, 2, 3, 4, 5, 6, 7, 8}. Poderíamos supor que a situação do agente é sem esperança mas, de fato, ele pode se sair muito bem. Tendo em vista que sabe o que suas ações fazem, ele pode – por exemplo – calcular que a ação *Direita* fará com que ele esteja em um dos estados {2, 4, 6, 8}, e que a seqüência de ações [*Direita,Aspirar*] sempre terminará em um dos estados {4, 8}. Finalmente, a seqüência [*Direita,Aspirar,Esquerda,Aspirar*] garante que ele irá alcançar o estado objetivo 7, independente do estado inicial. Dizemos que o agente pode fazer a **coerção** do mundo para o estado 7, mesmo que ele não saiba onde começou. Em suma, quando o mundo não é completamente observável, o agente deve raciocinar sobre *conjuntos* de estados que poderia alcançar, em vez de estados isolados. Chamamos cada conjunto de

COERÇÃO

ESTADO DE CRENÇA

estados de **estado de crença**, representando a crença atual do agente sobre os estados físicos possíveis em que ele poderia se encontrar. (Em um ambiente completamente observável, cada estado de crença contém um único estado físico.)

Para resolver problemas sem sensores, realizamos a busca no espaço de estados de crença, e não nos estados físicos. O estado inicial é um estado de crença, e cada ação faz o mapeamento de um estado de crença para outro estado de crença. Uma ação é aplicada a um estado de crença efetuando-se a união dos resultados da aplicação da ação a cada estado físico no estado de crença. Um caminho agora conecta diversos estados de crença e uma solução é um caminho que leva a um estado de crença, *cujos membros são todos* estados objetivos. A Figura 3.21 mostra o espaço de estados de crença acessível para o mundo de aspirador de pó determinístico sem sensores. Existem apenas 12 estados de crença acessíveis, mas o espaço de estados de crença inteiro contém cada conjunto possível de estados físicos, isto é, $2^8 = 256$ estados de crença. Em geral, se o espaço de estados físicos tem S estados, o espaço de estados de crença tem 2^S estados de crença.

Nossa discussão sobre problemas sem sensores pressupôs até agora ações determinísticas, mas a análise fica essencialmente inalterada se o ambiente for não-determinístico – isto é, se as ações puderem ter vários resultados possíveis. Isso ocorre porque, na ausência de sensores, o agente não tem como saber qual foi o resultado real, e assim os diversos resultados possíveis são apenas estados físicos adicionais no estado de crença do sucessor. Por exemplo, suponha que o ambiente obedeça à Lei de Murphy: a chamada ação *Aspirar às vezes* deposita sujeira no carpete, *mas só se ainda não houver*

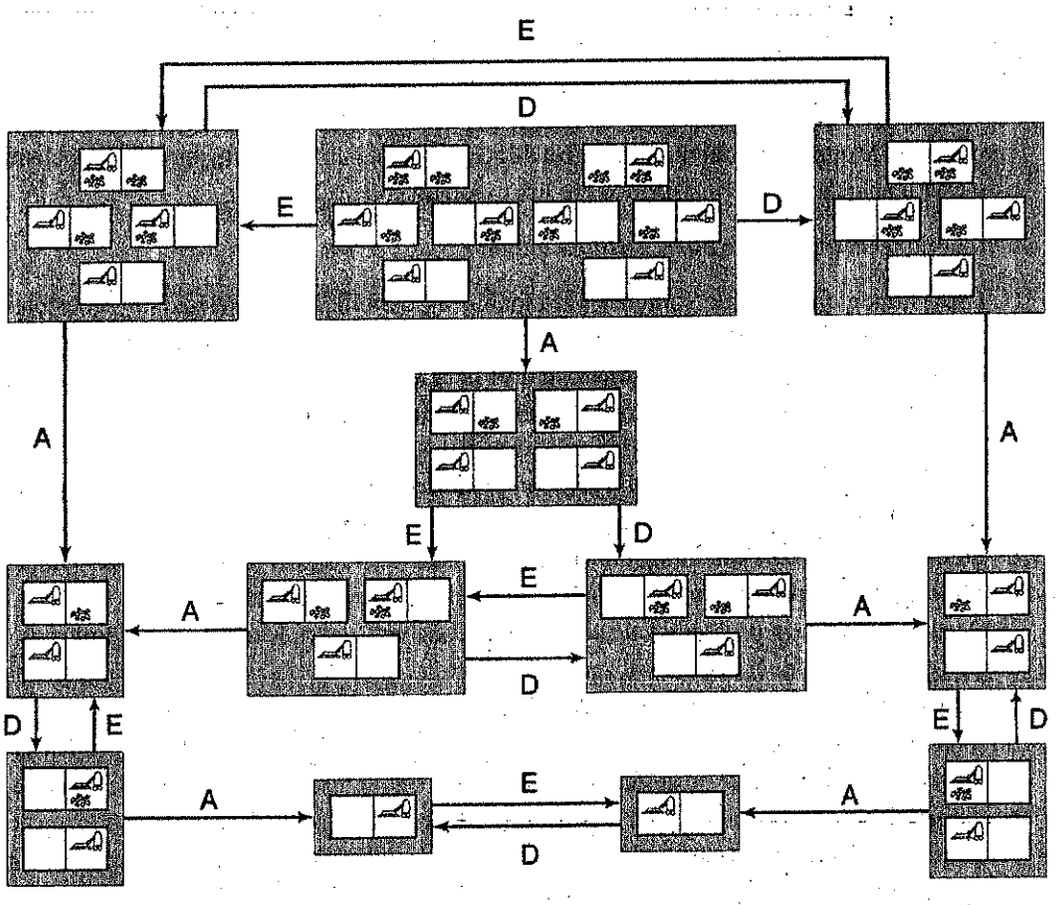


Figura 3.21 A porção acessível do espaço de estados de crença para o mundo de aspirador de pó determinístico e sem sensores. Cada retângulo sombreado corresponde a um único estado de crença. Em qualquer ponto dado, o agente se encontra em um estado de crença específico, mas não sabe em que estado físico se encontra. O estado de crença inicial (ignorância completa) é o retângulo central superior. As ações são representadas por arcos rotulados. Os laços de um estado para ele próprio foram omitidos para manter a clareza.

sujeira.[6] Então, se *Aspirar* for aplicada ao estado físico 4 (veja a Figura 3.20), haverá dois resultados possíveis: os estados 2 e 4. Aplicada ao estado de crença inicial, {1, 2, 3, 4, 5, 6, 7, 8}, *Aspirar* leva agora ao estado de crença que é a união dos conjuntos de resultados para os oito estados físicos. Calculando essa união, descobrimos que o novo estado de crença é {1, 2, 3, 4, 5, 6, 7, 8}. Assim, no caso de um agente sem sensores no mundo da Lei de Murphy, a ação *Aspirar* deixa o estado de crença inalterado! De fato, o problema é insolúvel. (Veja o Exercício 3.18.) Intuitivamente, isso acontece porque o agente não tem como saber se o quadrado atual está sujo e, por conseguinte, não tem como saber se a ação *Aspirar* o limpará ou criará mais sujeira.

Problemas de contingência

PROBLEMA DE CONTINGÊNCIA

Quando o ambiente é tal que o agente pode obter novas informações de seus sensores depois de agir, o agente enfrenta um **problema de contingência**. A solução para um problema de contingência freqüentemente toma a forma de uma *árvore*, onde cada ramo pode ser selecionado, dependendo das percepções recebidas até esse ponto na árvore. Por exemplo, suponha que o agente esteja no mundo da Lei de Murphy e tenha um sensor de posição e um sensor de sujeira local, mas nenhum sensor capaz de detectar sujeira em outros quadrados. Desse modo, a percepção [*E, Sujo*] significa que o agente está em um dos estados {1, 3}. O agente poderia formular a seqüência de ações [*Aspirar, Direita, Aspirar*]. A aspiração mudaria o estado para um dos estados {5, 7} e a movimentação para a direita alteraria o estado para um dos estados {6, 8}. A execução da última ação *Aspirar* no estado 6 nos leva ao estado 8, um objetivo, mas a execução dessa ação no estado 8 pode nos levar de volta ao estado 6 (pela Lei de Murphy) e, nesse caso, o plano falhará.

Examinando-se o espaço de estados de crença para essa versão do problema, é possível determinar com facilidade que nenhuma seqüência fixa de ações garante uma solução para esse problema. Porém, existe uma solução se não insistirmos em uma seqüência de ações *fixa*:

[*Aspirar, Direita,* ***se*** [*D,Sujo*] ***então*** *Aspirar*]

Isso estende o espaço de soluções para incluir a possibilidade de selecionar ações baseadas em contingências que surgem durante a execução. Muitos problemas no mundo físico real são problemas de contingência, porque é impossível um prognóstico exato. Por essa razão, muitas pessoas mantêm os olhos abertos enquanto estão caminhando ou dirigindo.

Às vezes, os problemas de contingência permitem soluções puramente seqüenciais. Por exemplo, considere um mundo de Lei de Murphy *completamente observável*. As contingências surgem se o agente executar uma ação *Aspirar* em um quadrado limpo, porque a sujeira pode ou não ser depositada no quadrado. Desde que o agente nunca faça isso, não surgirá nenhuma contingência e haverá uma solução seqüencial a partir de cada estado inicial (Exercício 3.18).

Os algoritmos para problemas de contingência são mais complexos que os algoritmos de busca-padrão deste capítulo; eles são focalizados no Capítulo 12. Os problemas de contingência também se prestam a um projeto de agente um pouco diferente, no qual o agente pode atuar *antes* de encontrar um plano garantido. Isso é útil porque, em vez de considerar com antecedência cada contingência possível que *poderia* surgir durante a execução, com freqüência é melhor começar a agir e ver quais contingências *surgem*. O agente pode então continuar a resolver o problema, levando em conta as informações adicionais. Esse tipo de **intercalação** de busca e execução também é útil em problemas de exploração (veja a Seção 4.5) e para jogos (veja o Capítulo 6).

INTERCALAÇÃO

6. Supomos que a maioria dos leitores enfrenta problemas semelhantes que podem simpatizar com nosso agente. Pedimos desculpas aos proprietários de eletrodomésticos modernos e eficientes que não podem tirar proveito desse recurso pedagógico.

3.7 Resumo

Este capítulo introduziu métodos que um agente pode usar para selecionar ações em ambientes determinísticos, observáveis, estáticos e completamente conhecidos. Em tais casos, o agente pode construir seqüências de ações que alcançam seus objetivos; esse processo é chamado **busca**.

- Antes de um agente poder começar a procurar soluções, ele deve formular um **objetivo**, e depois usar o objetivo para formular um **problema**.
- Um problema consiste em quatro partes: o **estado inicial**, um conjunto de **ações**, uma função **teste de objetivo** e uma função **custo de caminho**. O ambiente do problema é representado por um **espaço de estados**. Um **caminho** pelo espaço de estados a partir do estado inicial até um estado objetivo é uma **solução**.
- Um único algoritmo BUSCA-EM-ÁRVORE geral pode ser usado para resolver qualquer problema; variantes específicas do algoritmo incorporam diferentes estratégias.
- Os algoritmos de busca são julgados de acordo com **completeza**, **otimização**, **complexidade de tempo** e **complexidade de espaço**. A complexidade depende de b, o fator de ramificação no espaço de estados, e de d, a profundidade da solução mais rasa.
- A **busca em extensão** seleciona para expansão o nó mais raso não-expandido na árvore de busca. Ela é completa, ótima para passos de custo unitário e tem complexidade de tempo e espaço igual a $O(b^d)$. A complexidade de espaço a torna impraticável na maioria dos casos. A **busca de custo uniforme** é semelhante à busca em extensão, mas expande o nó com caminho de custo mais baixo, $g(n)$. Ela é completa e ótima se o custo de cada passo excede algum limite positivo ε.
- A **busca em profundidade** seleciona para expansão o nó não-expandido mais profundo na árvore de busca. Ela não é completa nem ótima, e tem complexidade de tempo igual a $O(b^m)$ e complexidade de espaço igual a $O(bm)$, onde m é a profundidade máxima de qualquer caminho no espaço de estados.
- A **busca em profundidade limitada** impõe um limite fixo de profundidade em uma busca em profundidade.
- A **busca por aprofundamento iterativo** chama a busca em profundidade limitada com limites crescentes até encontrar um objetivo. Ela é completa, ótima para passos de custo unitário e tem complexidade de tempo igual a $O(b^d)$ e complexidade de espaço igual a $O(bd)$.
- A **busca bidirecional** pode reduzir enormemente a complexidade de tempo, mas nem sempre é aplicável e pode exigir muito espaço.
- Quando o espaço de estados é um grafo em vez de uma árvore, pode ser compensador procurar estados repetidos na árvore de busca. O algoritmo BUSCA-EM-GRAFO elimina todos os estados duplicados.
- Quando o ambiente é parcialmente observável, o agente pode aplicar algoritmos de busca no espaço de **estados de crença** ou conjuntos de estados possíveis em que o agente poderia estar. Em alguns casos, uma única seqüência de solução pode ser construída; em outros casos, o agente precisa de um **plano de contingência** para lidar com circunstâncias desconhecidas que podem surgir.

Notas bibliográficas e históricas

A maior parte do problemas de busca em espaço de estados analisados neste capítulo tem uma longa história na literatura e são menos triviais do que poderia parecer. O problema dos missionários e canibais usado no Exercício 3.9 foi analisado em detalhes por Amarel (1968). Ele foi considerado anteriormente em IA por Simon e Newell (1961) e em pesquisa operacional por Bellman e Dreyfus (1962). Estudos como esses e como o trabalho de Newell e Simon no Logic Theorist (1957) e no GPS (1961) levaram ao estabelecimento de algoritmos de busca como as principais armas do arsenal dos pesquisadores da IA na década de 1960 e ao estabelecimento da resolução de problemas como a tarefa canônica da IA. Infelizmente, pouco trabalho foi desenvolvido na automação da etapa de formulação do problema. Um tratamento mais recente da representação e abstração de problemas, incluindo programas de IA que executam eles próprios essas tarefas (em parte), encontra-se em Knoblock (1990).

O quebra-cabeças de 8 peças é um irmão menor do quebra-cabeça de 15 peças, criado pelo famoso projetista de jogos americano Sam Loyd (1959) na década de 1870. O quebra-cabeça de 15 peças logo alcançou imensa popularidade nos Estados Unidos, comparável à sensação mais recente causada pelo cubo de Rubik. Ele também atraiu rapidamente a atenção de matemáticos (Johnson e Story, 1879; Tait, 1880). Os editores do *American Journal of Mathematics* declararam: "Nas últimas semanas, o quebra-cabeça de 15 peças chegou ao público americano, e pode-se dizer com segurança que ele atraiu a atenção de nove entre dez pessoas de ambos os sexos e de todas as idades e condições sociais na comunidade. Porém, isso não teria influenciado os editores a ponto de induzi-los a inserir artigos sobre tal assunto no *American Journal of Mathematics*, mas sim o fato de que ..." (segue-se um resumo do interesse matemático do quebra-cabeça de 15 peças). Uma análise exaustiva do quebra-cabeça de 8 peças foi feita com a ajuda de computadores por Schofield (1967). Ratner e Warmuth (1986) mostraram que a versão geral de $n \times n$ do quebra-cabeça de 15 peças pertence à classe de problemas NP-completos.

O problema de 8 rainhas foi publicado primeiro anonimamente na revista alemã de xadrez *Schach* em 1848; mais tarde, ele foi atribuído a um certo Max Bezzel. O problema foi republicado em 1850 e, nessa época, atraiu a atenção do eminente matemático Carl Friedrich Gauss, que tentou enumerar todas as soluções possíveis, mas encontrou apenas 72. Mais tarde, Nauck publicou todas as 92 soluções, em 1850. Netto (1901) generalizou o problema para n rainhas, e Abramson e Yung (1989) encontraram um algoritmo $O(n)$.

Cada um dos problemas de busca do mundo real listado no capítulo foi o assunto de um grande esforço de pesquisa. Os métodos para selecionar vôos ótimos de linhas aéreas permanecem patenteados em sua maior parte, mas Carl de Marcken (em uma comunicação pessoal) mostrou que a cotação e as restrições de passagens de linhas aéreas se tornaram tão complicadas que o problema de selecionar um vôo ótimo é formalmente *indecidível*. O problema do caixeiro-viajante é um problema combinatório padrão em ciência da computação teórica (Lawler, 1985; Lawler *et al.*, 1992). Karp (1972) provou que o PCV é NP-difícil, mas foram desenvolvidos métodos efetivos de aproximação heurística (Lin e Kernighan, 1973). Arora (1998) criou um esquema de aproximação completamente polinomial para PCVs euclidianos. Os métodos de layout VLSI foram pesquisados por Shahookar e Mazumder (1991), e surgiram muitos documentos de otimização de layout em periódicos de VLSI. Os problemas de navegação e montagem de robôs são discutidos no Capítulo 25.

Os algoritmos de busca sem informação para resolução de problemas constituem um tópico central da ciência de computação clássica (Horowitz e Sahni, 1978) e da pesquisa operacional (Dreyfus, 1969); Deo e Pang (1984) e Gallo e Pallottino (1988) apresentam pesquisas mais recentes. A busca em extensão foi formulada para resolver labirintos por Moore (1959). O método de **programação dinâmica** (Bellman e Dreyfus, 1962), que registra sistematicamente soluções para todos os subproble-

mas de comprimentos crescentes, pode ser visto como uma forma de busca em extensão sobre grafos. O algoritmo de caminhos mais curtos de dois pontos de Dijkstra (1959) é a origem da busca de custo uniforme.

Uma versão de aprofundamento iterativo projetada para fazer uso eficiente do relógio no xadrez foi usada primeiro por Slate e Atkin (1977) no programa de jogo de xadrez CHESS 4.5, mas a aplicação à busca do caminho mais curto em grafos se deve a Korf (1985a). A busca bidirecional, introduzida por Pohl (1969, 1971), também pode ser muito eficiente em alguns casos.

Os ambientes parcialmente observáveis e não-determinísticos não foram estudados em grande profundidade segundo a abordagem de resolução de problemas. Algumas questões sobre eficiência na busca em estados de crença foram investigadas por Genesereth e Nourbakhsh (1993). Koenig e Simmons (1998) estudaram a navegação de robôs a partir de uma posição inicial desconhecida, e Erdmann e Mason (1988) estudaram o problema de manipulação robótica sem sensores, utilizando uma forma contínua de busca em estados de crença. A busca com contingência foi estudada dentro do subcampo de planejamento. (Consulte o Capítulo 12.) Em sua maior parte, o planejamento e a atuação com informações incertas foram tratados usando-se as ferramentas de probabilidade e teoria da decisão (veja o Capítulo 17).

Os livros didáticos de Nilsson (1971, 1980) são boas fontes gerais de informações sobre algoritmos de busca clássicos. Um estudo completo e mais atualizado pode ser encontrado em Korf (1988). Documentos sobre novos algoritmos de busca – que, notavelmente, continuam a ser descobertos – aparecem em periódicos como *Artificial Intelligence*.

Exercícios

3.1 Defina com suas próprias palavras os termos a seguir: estado, espaço de estados, árvore de busca, nó de busca, objetivo, ação, função sucessor e fator de ramificação.

3.2 Explique por que a formulação do problema deve seguir a formulação do objetivo.

3.3 Suponha que AÇÕES-VÁLIDAS(s) denote o conjunto de ações válidas no estado s, e que RESULTADO(a, s) denote o estado que resulta da execução de uma ação válida a no estado s. Defina SUCESSOR em termos de AÇÕES-VÁLIDAS e RESULTADO, e *vice-versa*.

3.4 Mostre que os estados do quebra-cabeça de 8 peças se dividem em dois conjuntos disjuntos, tais que nenhum estado pertencente a um conjunto pode ser transformado em um estado pertencente ao outro conjunto por meio de qualquer número de movimentos. (*Sugestão*: Veja Berlekamp *et al.* (1982).) Crie um procedimento que informe em que classe encontra-se um determinado estado e explique por que isso é bom no caso da geração de estados aleatórios.

3.5 Considere o problema de n rainhas usando a formulação incremental "eficiente" dada na página 68. Explique por que o tamanho do espaço de estados é pelo menos $\sqrt[3]{n!}$ e faça uma estimativa do maior n para o qual a exploração exaustiva é possível. (*Sugestão:* Derive um limite inferior sobre o fator de ramificação, considerando o número máximo de quadrados que uma rainha pode atacar em qualquer coluna.)

3.6 Um espaço de estados finito conduz a uma árvore de busca finita? E no caso de um espaço de estados finito que é uma árvore? Você poderia ser mais preciso em definir que tipos de espaços de estados sempre levam a árvores de busca finitas? (Adaptado de Bender, 1996.)

3.7 Forneça o estado inicial, o teste de objetivo, a função sucessor e a função de custo para cada um dos itens a seguir. Escolha uma formulação que seja precisa o suficiente para ser implementada.

a. Você tem de colorir um mapa plano utilizando apenas quatro cores, de tal modo que não haja duas regiões adjacentes com a mesma cor.

b. Um macaco com um metro de altura está em uma sala em que algumas bananas estão suspensas no teto, a 2,5 metros de altura. Ele gostaria de alcançar as bananas. A sala contém dois engradados empilháveis, móveis e escaláveis, com um metro de altura cada.

c. Você tem um programa que emite a mensagem de saída "registro de entrada inválido" quando alimentado com um certo arquivo de registros de entrada. Você sabe que o processamento de cada registro é independente dos outros registros e deseja descobrir qual registro é inválido.

d. Você tem três jarros, com capacidade para 12 litros, 8 litros e 3 litros, respectivamente, e ainda uma torneira de água. É possível encher os jarros ou esvaziá-los passando a água de um para outro ou derramando-a no chão. Você precisa medir exatamente um litro.

3.8 Considere um espaço de estados onde o estado inicial é o número 1 e a função sucessor para o estado n retorna dois estados, com os números $2n$ e $2n + 1$.

a. Desenhe a porção do espaço de estados correspondente aos estados 1 a 15.

b. Suponha que o estado objetivo seja 11. Liste a ordem em que os nós serão visitados no caso da busca em extensão, da busca em profundidade limitada com limite 3 e da busca por aprofundamento iterativo.

c. A busca bidirecional seria apropriada para esse problema? Em caso afirmativo, descreva em detalhes como ela funcionaria.

d. Qual é o fator de ramificação em cada sentido da busca bidirecional?

e. A resposta para (c) sugere uma reformulação do problema que lhe permitiria resolver o problema de ir do estado 1 até um determinado estado objetivo com praticamente nenhuma busca?

3.9 O problema de **missionários e canibais** em geral é enunciado como a seguir. Três missionários e três canibais estão em um lado de um rio, juntamente com um barco que pode conter uma ou duas pessoas. Descubra um meio de fazer todos atravessarem o rio, sem deixar que um grupo de missionários de um lado fique em número menor que o número de canibais nesse lado do rio. Esse problema é famoso em IA, porque foi assunto do primeiro artigo que abordou a formulação de problemas a partir de um ponto de vista analítico (Amarel, 1968).

a. Formule o problema precisamente, fazendo apenas as distinções necessárias para assegurar uma solução válida. Trace um diagrama do espaço de estados completo.

b. Implemente e resolva o problema de forma ótima, utilizando um algoritmo de busca apropriado. É boa idéia verificar a existência de estados repetidos?

c. Por que você imagina que as pessoas têm dificuldades para resolver esse quebra-cabeça, considerando-se que o espaço de estados é tão simples?

3.10 Implemente duas versões da função sucessor para o quebra-cabeça de 8 peças: uma que gere todos os sucessores de uma vez copiando e editando a estrutura de dados do quebra-cabeça de 8 peças, e uma que gere um único novo sucessor cada vez que é chamado e funcione modificando diretamente o estado pai (desfazendo as modificações conforme necessário). Escreva versões de busca em profundidade por aprofundamento iterativo que utilizem essas funções e compare seu desempenho.

3.11 Na página 80, mencionamos a **busca por alongamento iterativo**, um análogo iterativo da busca de custo uniforme. A idéia é usar limites crescentes sobre o custo do caminho. Se for gerado um nó cujo custo de caminho exceder o limite atual, ele será imediatamente descartado. Para cada nova iteração, o limite é definido como o custo do caminho mais baixo de qualquer nó descartado na iteração anterior.

a. Mostre que esse algoritmo é ótimo para custos de caminhos gerais.
b. Considere uma árvore uniforme com fator de ramificação b, profundidade de solução d e passos de custo unitário. Quantas iterações exigirá o alongamento iterativo?
c. Agora, considere passos de custos obtidos no intervalo contínuo [0, 1] com um custo positivo mínimo ε. Quantas iterações são exigidas no pior caso?
d. Implemente o algoritmo e aplique-o a instâncias dos problemas de quebra-cabeça de 8 peças e caixeiro-viajante. Compare o desempenho do algoritmo ao desempenho da busca de custo uniforme e comente seus resultados.

3.12 Prove que a busca de custo uniforme e a busca em extensão com passos de custos constantes são ótimas quando utilizadas com o algoritmo BUSCA-EM-GRAFO. Mostre um espaço de estados com passos de custos constantes, no qual BUSCA-EM-GRAFO usando o aprofundamento iterativo encontra uma solução não-ótima.

3.13 Descreva um espaço de estados em que a busca por aprofundamento iterativo tem desempenho muito pior que o da busca em profundidade (por exemplo, $O(n^2)$ *versus* $O(n)$).

3.14 Escreva um programa que receba como entrada duas URLs de páginas da Web e encontre um caminho de links de um até o outro. Apresente uma estratégia de busca apropriada. A busca bidirecional é uma boa idéia? Um mecanismo de busca na Internet poderia ser usado para implementar uma função predecessora?

3.15 Considere o problema de encontrar o caminho mais curto entre dois pontos em um plano que tem obstáculos poligonais convexos, como mostra a Figura 3.22. Esta é uma idealização do problema que um robô tem de resolver para navegar em um ambiente congestionado.

a. Suponha que o espaço de estados consista em todas as posições (x, y) do plano. Quantos estados existem? Quantos caminhos existem até o objetivo?
b. Explique brevemente por que o caminho mais curto de um vértice de um polígono até qualquer outro vértice na cena deve consistir em segmentos de reta que unem alguns vértices dos polígonos. Agora defina um bom espaço de estados. Qual é o tamanho desse espaço de estados?
c. Defina as funções necessárias para implementar o problema de busca, incluindo uma função sucessor que receba um vértice como entrada e retorne o conjunto de vértices que podem ser alcançados em linha reta a partir do vértice dado. (Não se esqueça dos vizinhos no mesmo polígono.) Utilize a distância em linha reta como função heurística.
d. Aplique um ou mais algoritmos deste capítulo para resolver alguns problemas neste domínio, e comente seu desempenho.

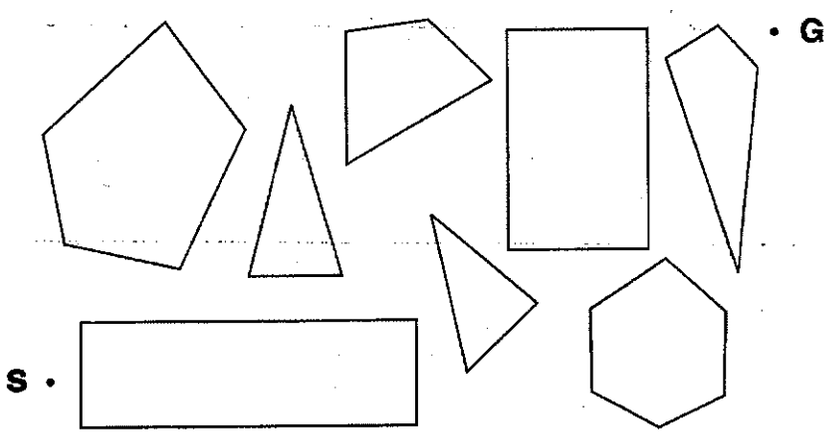


Figura 3.22 Uma cena com obstáculos poligonais.

3.16 Podemos transformar o problema de navegação do Exercício 3.15 em um ambiente como este:

- A percepção será uma lista das posições, *relativas ao agente*, dos vértices visíveis. A percepção *não* inclui a posição do robô! O robô deve deduzir sua própria posição a partir do mapa; por enquanto, você pode assumir que cada posição tem uma "visão" diferente.
- Cada ação será um vetor que descreve um caminho em linha reta a ser seguido. Se o caminho não estiver obstruído, a ação terá sucesso; caso contrário, o robô irá parar no ponto em que seu caminho encontrar um obstáculo. Se o agente retornar um vetor de movimentação zero e estiver no objetivo (fixo e conhecido), o ambiente deve teletransportar o agente para uma *posição aleatória* (que não seja interior a um obstáculo).
- A medida de desempenho debita um ponto do agente por cada unidade de distância percorrida e o recompensa com 1.000 pontos toda vez que o objetivo é alcançado.

a. Implemente esse ambiente e um agente de resolução de problemas para ele. O agente precisará formular um novo problema após cada teletransporte, o que envolverá descobrir sua posição atual.

b. Documente o desempenho do seu agente (fazendo o agente gerar comentários adequados à medida que se movimenta) e informe seu desempenho depois de 100 episódios.

c. Modifique o ambiente de forma que, durante 30% do tempo, o agente termine em um destino não-pretendido (escolhido ao acaso a partir dos outros vértices visíveis, se houver; caso contrário, não haverá nenhum movimento). Esse é um modelo grosseiro dos erros de movimentação de um robô real. Modifique o agente de forma que, ao ser detectado um erro desse tipo, ele descubra onde está e depois elabore um plano para voltar até onde estava e prosseguir com o plano antigo. Lembre-se de que às vezes a volta até o lugar em que ele estava também pode falhar! Mostre um exemplo em que o agente supera com sucesso dois erros de movimentação consecutivos e ainda assim alcança o objetivo.

d. Agora tente dois esquemas de recuperação diferentes após um erro: (1) dirija-se ao vértice mais próximo na rota original; e (2) recrie uma rota até o objetivo a partir da nova posição. Compare o desempenho dos três esquemas de recuperação. A inclusão dos custos da busca afetaria a comparação?

e. Agora, suponha que existam posições a partir das quais a visão seja idêntica. (Por exemplo, suponha que o mundo seja uma malha com obstáculos quadrados.) Que tipo de problema o agente enfrenta agora? Quais seriam as soluções?

3.17 Na página 62, dissemos que não consideraríamos os problemas com custos de caminhos negativos. Neste exercício, vamos explorar esse tema com maior profundidade.

a. Suponha que as ações possam ter custos negativos arbitrariamente grandes; explique por que essa possibilidade forçaria qualquer algoritmo ótimo a explorar todo o espaço de estados.

b. Ajudaria se insistíssemos no fato de que os custos dos passos devem ser maiores ou iguais a alguma constante negativa c? Considere árvores e grafos.

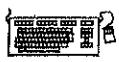
c. Suponha que exista um conjunto de operadores formando um ciclo, de modo que cada execução do conjunto em alguma ordem não resulte em mudança líquida no estado. Se todos esses operadores tiverem custo negativo, qual será a implicação desse fato sobre o comportamento ótimo de um agente em tal ambiente?

d. É possível imaginar facilmente operadores com alto custo negativo, até mesmo em domínios como a localização de rotas. Por exemplo, algumas extensões da estrada poderiam ter belas paisagens que superem de longe os custos normais em termos de tempo e combustível. Explique,

em termos precisos, dentro do contexto de busca de espaços de estados, por que os seres humanos não dirigem indefinidamente em ciclos em belos cenários, e explique como definir o espaço de estados e os operadores para localização de rotas, de forma que agentes artificiais também possam evitar ciclos repetitivos.

e. Você poderia imaginar um domínio real em que os custos dos passos sejam de tal ordem que provoquem a entrada em ciclos repetitivos?

3.18 Considere o mundo de aspirador de pó sem sensores e de duas posições sob a Lei de Murphy. Trace o espaço de estados de crença acessíveis a partir do estado de crença inicial {1, 2, 3, 4, 5, 6, 7, 8) e explique por que o problema é insolúvel. Mostre também que, se o mundo for completamente observável, existirá uma seqüência de solução para cada estado inicial possível.



3.19 Considere o problema do mundo de aspirador de pó definido na Figura 2.2.

a. Qual dos algoritmos definidos neste capítulo seria apropriado para esse problema? O algoritmo deve verificar estados repetidos?

b. Aplique o algoritmo escolhido para calcular uma seqüência ótima de ações para um mundo 3 × 3 cujo estado inicial tem sujeira nos três quadrados superiores e o agente está no centro.

c. Construa um agente de busca para o mundo de aspirador de pó e avalie seu desempenho em um conjunto de mundos 3 × 3 com probabilidade de sujeira igual a 0,2 em cada quadrado. Inclua o custo de busca, bem como o custo de caminho na medida de desempenho, utilizando uma taxa de ponderação razoável.

d. Compare seu melhor agente de busca com um agente reativo aleatório simples que aspira se há sujeira e se move aleatoriamente em caso contrário.

e. Considere o que aconteceria se o mundo fosse ampliado para $n \times n$. Como o desempenho do agente de busca e do agente reativo variará com n?