Busca no Espaço de Estados

Prof. Dr. Silvio do Lago Pereira

slago@ime.usp.br

1 Introdução

Busca no espaço de estados é uma das técnicas mais utilizadas para resolução de problemas em Inteligência Artificial [1,3,4]. A idéia consiste em supor a existência de um *agente* capaz de executar *ações* que modificam o *estado* corrente de seu mundo. Assim, dados um *estado inicial* representando a configuração corrente do mundo do agente, um *conjunto de ações* que o agente é capaz de executar e uma descrição do *estado meta* que se deseja atingir, a *solução* do problema consiste numa *seqüência de ações* que, quando executada pelo agente, transforma o estado inicial num estado meta.

Nesse artigo, introduzimos o conceito de espaço de estados e o utilizamos para especificar formalmente um problema de busca, bem como um algoritmo não-determinístico que resolve esse tipo de problema. Em seguida, apresentamos as estratégias de busca cega (*em largura* e *em profundidade*) como formas de tornar o algoritmo determinístico. Finalmente, apresentamos as estratégias de busca heurística (*menor-custo*, *melhor-estimativa* e A^*) como formas de garantir a qualidade das soluções e melhorar a eficiência da busca determinística.

2 Espaço de estados

Um *espaço de estados* é definido por um conjunto $\mathcal{S}$ de estados e por um conjunto $\mathcal{A}$ de ações que mapeiam um estado em outro [3].

Como exemplo, vamos considerar o *Mundo do Aspirador* [4]. Nesse mundo, o agente é um aspirador cuja função é limpar as salas de um edifício. Numa versão bastante simplificada, vamos supor que o mundo desse agente seja composto de apenas duas salas, cada uma delas podendo estar limpa ou suja, e que o aspirador seja capaz de executar apenas três ações: $entrarSala1$, $entrarSala2$ e $aspirar$.

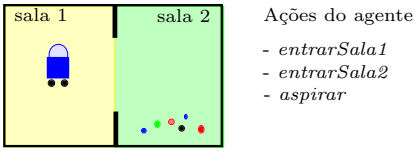


Figura 1. Mundo do Aspirador Simplificado

2.1 Representação de estados

Estados são representados por estruturas, onde cada componente denota um atributo do estado representado. Por exemplo, no Mundo do Aspirador, cada estado pode ser representado por uma estrutura da forma $[X, Y, Z]$, onde $X \in \{1, 2\}$ indica a posição do aspirador, $Y \in \{0, 1\}$ indica se a primeira sala está suja e $Z \in \{0, 1\}$ indica se a segunda sala está suja. Dessa forma, o estado em que o aspirador encontra-se na segunda sala e apenas essa sala está suja é representado por $[2, 0, 1]$. O conjunto de estados para o Mundo do Aspirador é:

$\mathcal{S} = \{[1,0,0], [1,0,1], [1,1,0], [1,1,1], [2,0,0], [2,0,1], [2,1,0], [2,1,1]\}$.

2.2 Representação de ações

As ações são representadas por operadores da forma $oper(\alpha, s, s') \leftarrow \beta$, onde α é uma ação que transforma o estado s no estado s', dado que a condição β esteja satisfeita. Por exemplo, a ação `aspirar` pode ser representada pelos seguintes operadores:

$oper(aspirar, [1, Y, Z], [1, 0, Z]) \leftarrow Y = 1$
$oper(aspirar, [2, Y, Z], [2, Y, 0]) \leftarrow Z = 1$

Geralmente, condições que envolvem apenas teste de igualdade podem ser estabelecidas de forma implícita. Por exemplo, os operadores acima também podem ser codificados como

$oper(aspirar, [1, 1, Z], [1, 0, Z])$
$oper(aspirar, [2, Y, 1], [2, Y, 0])$

Assim, o conjunto de ações para o Mundo do Aspirador é:

$\mathcal{A} = \{oper(entrarSala1, [2, Y, Z], [1, Y, Z]),$
$oper(entrarSala2, [1, Y, Z], [2, Y, Z]),$
$oper(aspirar, [1, 1, Z], [1, 0, Z]),$
$oper(aspirar, [2, Y, 1], [2, Y, 0])\}$

2.3 Estados sucessores

Dado um estado $s \in \mathcal{S}$, seus *estados sucessores* são todos aqueles que podem ser atingidos, a partir de s, pela aplicação de um dos operadores do domínio. Por exemplo, *expandindo* o estado $[2, 0, 1]$, obtemos como estados sucessores $[1, 0, 1]$ e $[2, 0, 0]$. Esses estados são gerados pela aplicação dos operadores $entrarSala1$ e $aspirar$, respectivamente. Note, por exemplo, que o operador $entrarSala2$ não pode ser usado na expansão do estado $[2, 0, 1]$; já que, nesse estado, a condição implícita do operador (*i.e.* $Y = 1$) não está satisfeita.

Exercício 1 *Desenhe um grafo representando o espaço de estados para o Mundo do Aspirador. Nesse grafo, cada nó será um estado do mundo e cada arco (rotulado com uma ação) será uma transição entre dois estados. Os arcos devem ser direcionados do estado para seu estado sucessor.* □

Exercício 2 *Considere uma versão do Mundo do Aspirador onde há um prédio com dois pisos, cada um deles com duas salas (1 e 2) e um saguão (0). Não há passagem direta de uma sala para outra, de modo que o aspirador tem que estar no saguão para entrar numa sala ou para mudar de piso. Para representar os estados nessa versão do problema, podemos usar uma estrutura da forma* $[Pos, Piso_1, Piso_2]$*, onde* $Pos = [Piso, Sala]$*, sendo* $Piso \in \{1, 2\}$ *e* $Sala \in \{0, 1, 2\}$*, indica a posição corrente do aspirador e* $Piso_i = [X, Y]$*, sendo* $X, Y \in \{0, 1\}$*, representa as salas do piso* i*, como na versão simplificada do problema. Por exemplo, se o aspirador estiver no saguão do primeiro piso e houver lixo apenas na sala 1 do segundo piso, o estado correspondente será* $[[1, 0], [0, 0][1, 0]]$*. Com base nessa representação, codifique os operadores para as ações subir, descer,* $entrarSala1$*,* $entrarSala2$*, aspirar e sair.* □

3 O problema de busca

Um *problema de busca* [2,4] é especificado através de três componentes:

- um *espaço de estados* (denotado pelos conjuntos $\mathcal{S}$ e $\mathcal{A}$);
- um *estado inicial* (denotado por um estado particular $s_0 \in S$) e
- um conjunto de *estados meta* (denotado por um conjunto $\mathcal{G} \subseteq S$).

Continuando com o Mundo do Aspirador como exemplo, um possível problema de busca seria o seguinte: *dado que inicialmente o aspirador esteja na primeira sala e que ambas as salas estejam sujas, encontre um estado onde ambas as salas estejam limpas.* Nesse caso, temos

- *espaço de estados*: conjuntos $\mathcal{S}$ e $\mathcal{A}$, conforme descritos na seção 2;
- *estado inicial*: $[1, 1, 1]$ e
- *estados meta*: $\mathcal{G} = \{[1, 0, 0], [2, 0, 0]\}$.

A *solução* para um problema de busca consiste numa seqüência de ações que rotulam o caminho que leva do estado inicial a um dos estados meta no espaço de estados do problema.

Exercício 3 *Encontre duas soluções possíveis para o problema de busca especificado acima, identificando-as no grafo que representa o espaço de estados para a versão simplificada do Mundo do Aspirador (veja exercício 1).* □

3.1 Algoritmo de busca não-determinístico

Geralmente, os algoritmos de busca não especificam explicitamente o conjunto de estados de um problema. Isso acontece porque esses estados podem ser gerados sob demanda, à medida em que forem sendo encontrados durante a busca.

Dados um conjunto $\mathcal{A}$ de ações, um estado inicial $s_0 \in \mathcal{S}$ e um conjunto de estados meta $\mathcal{G} \subseteq \mathcal{S}$, um algoritmo não-determinístico[1] de busca pode ser especificado sa seguinte maneira:

```
BUSCA(A, s_0, G)
1  Σ ← {s_0}
2  enquanto Σ ≠ ∅ faça
3    s ← remove(Σ)
4    se s ∈ G então devolva caminho(s)
5    Σ ← Σ ∪ sucessores(s, A)
6  devolva fracasso
```

Nesse algoritmo, a função $remove(\Sigma)$ remove e devolve um dos estados no conjunto Σ, escolhido de forma aleatória; a função $caminho(s)$ devolve a seqüência de ações que rotulam o caminho que vai de s_0 até s, no espaço de estados do problema; e, finalmente, a função $sucessores(s, \mathcal{A})$ devolve o conjunto dos estados sucessores de s, obtidos a partir da aplicação dos operadores em $\mathcal{A}$.

O rastreamento da execução de um algoritmo de busca, produz uma estrutura denominada *árvore de busca*.

Exemplo 1. Sejam $\mathcal{A}$ o conjunto de ações para o Mundo do Aspirador, $s_0 = [1, 1, 1]$ e $\mathcal{G} = \{[1, 0, 0], [2, 0, 0]\}$. O rastreamento da chamada $\textsc{Busca}(\mathcal{A}, s_0, \mathcal{G})$ pode produzir, por exemplo, a árvore de busca apresentada na Figura 2.

Na primeira iteração do laço no algoritmo Busca, temos $\Sigma = \{[1, 1, 1]\}$. Então, quando a função *remove* é chamada, a única escolha possível é $s = [1, 1, 1]$. Entretanto, como esse estado não é meta ($s \notin \mathcal{G}$), seus sucessores ($[2, 1, 1]$ e $[1, 0, 1]$) são adicionados ao conjunto Σ e a execução prossegue. Assim, na segunda iteração do laço, teremos $\Sigma = \{[2, 1, 1], [1, 0, 1]\}$. Como o algoritmo é não-determinístico, qualquer um desses estados poderá ser escolhido. Vamos supor que a escolha seja $s = [2, 1, 1]$. Como esse estado não é meta, na terceira iteração teremos $\Sigma = \{[1, 1, 1], [2, 1, 0], [1, 0, 1]\}$. O algoritmo escolhe $s = [2, 1, 0]$ e adiciona seu único sucessor $[1, 1, 0]$ a Σ. Dessa forma, iniciamos a quarta iteração do laço com $\Sigma = \{[1, 1, 1], [1, 1, 0], [1, 0, 1]\}$. Supondo que nessa iteração a escolha seja $s = [1, 1, 0]$, Σ passa a ser o conjunto $\{[1, 1, 1], [1, 0, 0], [1, 0, 1]\}$. Finalmente, se na quinta iteração a escolha for $s = [1, 0, 0]$, como esse é um estado meta, a busca termina e o caminho $[entrarSala2, aspirar, entrarSala1, aspirar]$ é devolvida como solução do problema. □

[1] Algoritmos não-determinístico são aqueles que fazem algum tipo de escolha aleatória.

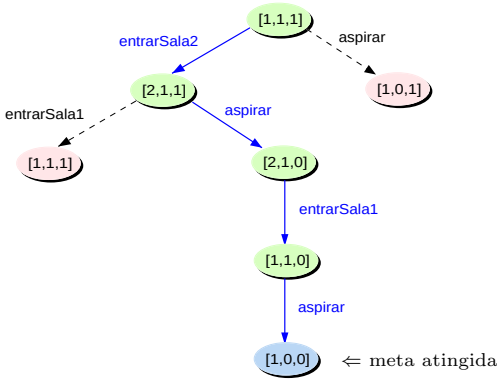


Figura 2. Árvore de busca gerada pelo algoritmo BUSCA

Exercício 4 *Mostre que, fazendo outras escolhas, o algoritmo de busca não-determinística poderia ter encontrado uma solução mais curta (i.e. com menos ações) para o problema do exemplo 1.* □

Exercício 5 *Mostre que, fazendo outras escolhas, o algoritmo de busca não-determinística poderia ficar executando infinitamente, sem nunca encontrar uma solução para o problema do exemplo 1.* □

3.2 Detecção de ciclos no espaço de estados

Um problema com o algoritmo de busca não-determinística é que, caso o espaço de estados do problema contenha ciclos, ele pode executar infinitamente. Para evitar que isso aconteça, podemos guardar os estados já expandidos e impedir que esses estados sejam expandidos novamente durante a busca.

BUSCA$'(\mathcal{A}, s_0, \mathcal{G})$
1 $\Gamma \leftarrow \emptyset$
2 $\Sigma \leftarrow \{s_0\}$
3 **enquanto** $\Sigma \neq \emptyset$ **faça**
4 $s \leftarrow remove(\Sigma)$
5 **se** $s \in \mathcal{G}$ **então devolva** $caminho(s)$
6 $\Gamma \leftarrow \Gamma \cup \{s\}$
7 $\Sigma \leftarrow \Sigma \cup (sucessores(s, \mathcal{A}) - \Gamma)$
8 **devolva fracasso**

Nessa nova versão do algoritmo, o conjunto Γ é utilizado para guardar os estados expandidos durante a busca. Quando um estado é expandido, todos seus sucessores já expandidos anteriormente são descartados, de modo que apenas

estados ainda não explorados são adicionados ao conjunto Σ. Com essa modificação, garantimos que o algoritmo BUSCA$'$ termine, sempre que o espaço de estados do problema de busca for finito.

Exercício 6 *Desenhe uma das árvores de buscas que poderiam ser produzidas pelo rastreamento da chamada* BUSCA$'(\mathcal{A}, s_0, \mathcal{G})$*, sendo* $\mathcal{A}$ *o conjunto de ações para o Mundo do Aspirador,* $s_0 = [1, 1, 1]$ *e* $\mathcal{G} = \{[2, 0, 0]\}$. □

4 Estratégias de busca cega

As estratégias de *busca cega* ou *busca não-informada* consistem em políticas que sistematizam o comportamento de um algoritmo de busca, sem levar em conta a qualidade da solução encontrada[2] [2,4]. Há duas estratégias de busca cega que são bastante utilizadas na prática: *busca em largura* (*breadth-first search*) e *busca em profundidade* (*depth-first search*).

4.1 Busca em largura

Na busca em largura, o estado inicial (nível 0) é expandido primeiro, sendo seus sucessores posicionados no nível 1 da árvore de busca. Em seguida, cada um dos estados do nível 1 são expandidos, sendo seus sucessores posicionados no nível 2, e assim por diante, de tal forma que todos os estados num nível n sejam expandidos antes daqueles no nível $n + 1$.

O algoritmo de busca em largura é obtido simplesmente sistematizando a ordem de inserção e remoção de estados no conjunto Σ, que agora passa a se comportar como uma *fila*[3]:

```
BuscaLargura(A, s0, G)
1  Γ ← ∅
2  Σ ← {s0}
3  enquanto Σ ≠ ∅ faça
4    s ← removePrimeiro(Σ)
5    se s ∈ G então devolva caminho(s)
6    Γ ← Γ ∪ {s}
7    insereNoFinal(sucessores(s, A) − Γ, Σ)
8  devolva fracasso
```

Nesse algoritmo, a função $removePrimeiro$ e o procedimento $insereNoFinal$ servem para tratar a variável Σ como uma fila. Agora, o algoritmo de busca não tem escolha: o estado a ser expandido é sempre aquele que encontra-se no início da fila Σ.

[2] Como veremos, as estratégias de busca heurística tentam minimizar o custo da solução, seja em termos do número de ações ou em termos dos custos dessas ações.

[3] Numa fila, também chamada lista FIFO, o primeiro que entra é o primeiro que sai.

Exemplo 2. Sejam $\mathcal{A}$ o conjunto de ações para o Mundo do Aspirador, $s_0 = [1,1,1]$ e $\mathcal{G} = \{[1,0,0],[2,0,0]\}$. O rastreamento de BuscaLargura$(\mathcal{A}, s_0, \mathcal{G})$ produz a arvore de busca da Figura 3, onde os estados estão numerados na ordem em que eles são selecionados para expansão durante a busca.

O algoritmo BuscaLargura sempre encontra uma solução com o menor número de ações[4]. Particularmente, nesse exemplo, como todos os estados do nível 3 são examinados antes daqueles no nível 4, a busca terminará assim que o estado $[2,0,0]$ for atingido; devolvendo o caminho $[aspirar, entrarSala2, aspirar]$ como solução do problema. □

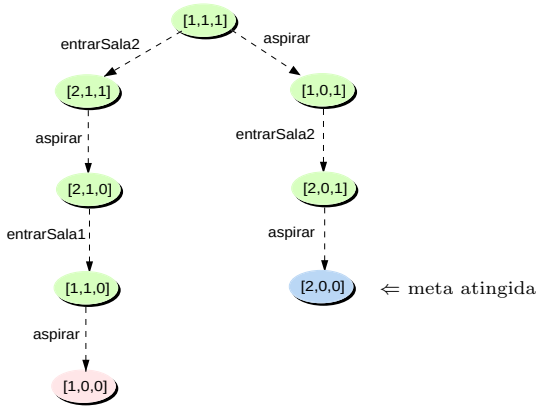


Figura 3. Árvore de busca gerada pelo algoritmo BuscaLargura

Exercício 7 *O Problema dos Jarros [3] consiste no seguinte:*

> *Há dois jarros com capacidades de 3 e 4 litros, respectivamente. Nenhum dos jarros contém qualquer medida ou escala, de modo que só se pode saber o conteúdo exato quando eles estão cheios. Sabendo-se que podemos encher ou esvaziar um jarro, bem como trasferir água de um jarro para outro, encontre uma seqüência de passos que deixe o jarro de 4 litros com exatamente 2 litros de água.*

Para representar os estados desse problema, podemos usar um par $[X,Y]$, onde $X \in \{0,1,2,3\}$ representa o conteúdo do primeiro jarro e $Y \in \{0,1,2,3,4\}$ representa o conteúdo do segundo jarro.

As ações podem ser representadas pelos seguintes operadores:

[4] Como veremos, uma solução com o menor número de ações não é, necessariamente, a solução de menor custo.

$oper(enche1, [X, Y], [3, Y]) \leftarrow X < 3$
$oper(enche2, [X, Y], [X, 4]) \leftarrow Y < 4$
$oper(esvazia1, [X, Y], [0, Y]) \leftarrow X > 0$
$oper(esvazia2, [X, Y], [X, 0]) \leftarrow Y > 0$
$oper(despeja1em2, [X, Y], [0, X + Y]) \leftarrow X > 0, Y < 4, X + Y \leq 4$
$oper(despeja1em2, [X, Y], [X + Y - 4, 4]) \leftarrow X > 0, Y < 4, X + Y > 4$
$oper(despeja2em1, [X, Y], [X + Y, 0]) \leftarrow X < 3, Y > 0, X + Y \leq 3$
$oper(despeja2em1, [X, Y], [3, X + Y - 3]) \leftarrow X < 3, Y > 0, X + Y > 3$

O estado inicial é $s_0 = [0, 0]$ *e o conjunto de estados meta é* $\mathcal{G} = \{[X, 2]\}$.

Com base nessa especificação, desenhe a árvore de busca criada pelo algoritmo BuscaLargura*, ao procurar a solução do problema.* □

4.2 Busca em profundidade

Na busca em profundidade, expandimos sempre o estado mais à esquerda, no nível mais profundo da árvore de busca; até que uma solução seja encontrada, ou até que um *beco*[5] seja atingido. Nesse último caso, retrocedemos e reiniciamos a busca no próximo estado ainda não expandido, posicionado mais à esqueda, no nível mais profundo da árvore[6].

Assim como no caso da busca em largura, o algoritmo de busca em profundidade é obtido pela sistematização da ordem de inserção e remoção de estados no conjunto Σ. No caso da busca em profundidade, entretanto, em vez de uma fila usamos uma *pilha*[7]. Essa é a única mudança necessária para transformar a busca em largura numa busca em profundidade.

```
BuscaProfundidade(A, s0, G)
1  Γ ← ∅
2  Σ ← {s0}
3  enquanto Σ ≠ ∅ faça
4    s ← removePrimeiro(Σ)
5    se s ∈ G então devolva caminho(s)
6    Γ ← Γ ∪ {s}
7    insereNoInicio(sucessores(s, A) − Γ, Σ)
8  devolva fracasso
```

Nesse algoritmo, a função $removePrimeiro$ e o procedimento $insereNoInicio$ garantem que a variável Σ seja tratada como uma pilha.

[5] Um *beco* é um estado que não pode ser expandido, seja porque ele já foi visitado anteriormente, ou porque não há nenhum operador aplicável para a sua expansão.
[6] Essa política de retrocesso é também conhecida como *backtracking*.
[7] Numa pilha, também chamada lista Lifo, o último que entra é o primeiro que sai.

Exemplo 3. Sejam $\mathcal{A}$ o conjunto de ações para o Mundo do Aspirador, $s_0 = [1,1,1]$ e $\mathcal{G} = \{[2,0,1]\}$. O rastreamento de BuscaProfundidade$(\mathcal{A}, s_0, \mathcal{G})$ produz a árvore de busca da Figura 4, onde os estados estão numerados na ordem em que eles são selecionados para expansão durante a busca.

Note que o estado 5-$[2,0,0]$ é um beco. O único operador que pode ser aplicado a esse estado é $entrarSala1$, mas isso resultaria no estado sucessor $[1,0,0]$, que é um estado já visitado anteriormente. Sendo assim, a busca retrocede e toma outro caminho, que produz a solução $[aspirar, entrarSala2]$. □

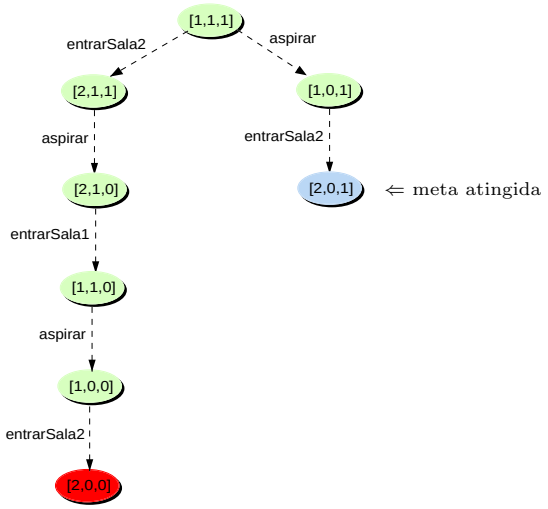


Figura 4. Árvore de busca gerada pelo algoritmo BuscaProfundidade

Exercício 8 *Sejam $\mathcal{A}$ o conjunto de ações para o Mundo do Aspirador, $s_0 = [1,1,1]$ e $\mathcal{G} = \{[1,0,0],[2,0,0]\}$. Mostre que, para esse problema, a busca em profundidade encontra uma solução com uma ação a mais que aquela encontrada pela busca em largura, para o mesmo problema.* □

Exercício 9 *O Problema do Fazendeiro [5] consiste no seguinte:*

> *Um fazendeiro encontra-se na margem esquerda de um rio, levando consigo um lobo, uma ovelha e um repolho. O fazendeiro precisa atingir a outra margem do rio com toda a sua carga intacta, mas para isso dispõe somente de um pequeno bote com capacidade para levar apenas ele mesmo e mais uma de suas cargas. O fazendeiro poderia cruzar o rio quantas vezes fossem necessárias para trasportar seus pertences, mas o problema é que, na ausência do fazendeiro, o lobo pode comer a ovelha e essa, por sua vez, pode comer o repolho. Encontre uma seqüência de passos que resolva esse problema.*

Para representar os estados desse problema, podemos usar uma estrutura da forma $[F, L, O, R]$*, cujas variáveis denotam, respectivamente, as posições do fazendeiro, do lobo, da ovelha e do repolho. Cada variável pode assumir os valores* e *ou* d*, dependendo da margem do rio onde o objeto se encontra.*

As ações podem ser representadas pelos seguintes operadores:

$oper(vai, [e, L, O, R], [d, L, O, R]) \leftarrow L \neq O, O \neq R$
$oper(levaLobo, [e, e, O, R], [d, d, O, R]) \leftarrow O \neq R$
$oper(levaOvelha, [e, L, e, R], [d, L, d, R])$
$oper(levaRepolho, [e, L, O, e], [d, L, O, d]) \leftarrow L \neq O$
$oper(volta, [d, L, O, R], [e, L, O, R]) \leftarrow L \neq O, O \neq R$
$oper(trazLobo, [d, d, O, R], [e, e, O, R]) \leftarrow O \neq R$
$oper(trazOvelha, [d, L, d, R], [e, L, e, R])$
$oper(trazRepolho, [d, L, O, d], [e, L, O, e]) \leftarrow L \neq O$

O estado inicial é $s_0 = [e, e, e, e]$ *e o conjunto de estados meta é* $\mathcal{G} = \{[d, d, d, d]\}$.

Com base nessa especificação, desenhe a árvore de busca criada pelo algoritmo BuscaProfundidade*, ao procurar a solução do problema.* □

Exercício 10 *Considere os seguintes operadores, que descrevem os vôos existentes entre cidades de um país:*

$oper(1, a, b)$, $oper(2, a, b)$, $oper(3, a, d)$, $oper(4, b, e)$, $oper(5, b, f)$, $oper(6, c, g)$, $oper(7, c, h)$, $oper(8, c, i)$, $oper(9, d, j)$, $oper(10, e, k)$, $oper(11, e, l)$, $oper(12, g, m)$, $oper(13, j, n)$, $oper(14, j, o)$, $oper(15, k, f)$, $oper(16, l, h)$, $oper(17, m, d)$, $oper(18, o, a)$, $oper(19, n, b)$

Por exemplo, o operador $oper(1, a, b)$ *indica que o vôo* 1 *parte da cidade* A *e chega à cidade* B*. Com base nesses operadores, e supondo que eles sejam usados na ordem em que eles foram declarados, desenhe as árvores de busca em largura e em profundidade que são criadas durante a busca por uma seqüência de vôos que levem da cidade* A *até a cidade* J*.* □

5 Estratégias de busca heurística

Como vimos na seção anterior, as estratégias de busca cega encontram soluções testando e expandindo estados, sistematicamente. Infelizmente, além de serem ineficientes, essas estratégias não garantem encontrar soluções de custo mínimo.

Nessa seção, vamos apresentar mais três estratégias de busca: a primeira, baseada no conceito de *custo de ação*, é ineficiente, mas garante encontrar uma solução de custo mínimo; a segunda, baseada do conceito de *função heurística*[8], é muito eficiente, mas não garante encontrar uma solução de custo mínimo; e, finalmente, a terceira, que combina as duas estratégias anteriores, não só é muito eficiente, mas também garante encontrar uma solução de custo mínimo [2,3,4].

[8] Essa palavra tem a mesma raiz da palavra *eureka*, do grego, e significa *descoberta*.

5.1 Custo de ação

Em muitos problemas, as ações podem ter custos associados, dependendo da dificuldade que o agente tem em executá-las. Para levar essa informação em conta durante a busca, precisamos adicionar mais um campo na especificação dos operadores: $oper(\alpha, s, s', g(\alpha)) \leftarrow \beta$, onde $g(\alpha)$ é o custo da ação α, que transforma o estado s num estado s', dado que a condição β esteja satisfeita.

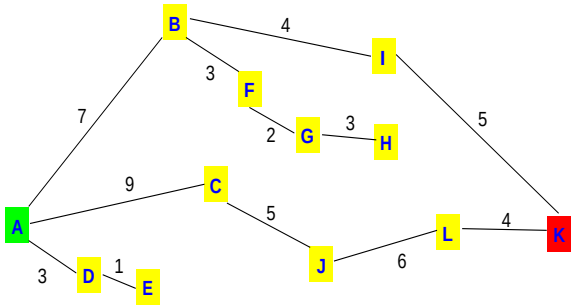


Figura 5. Mapa de vias entre cidades

Como exemplo, vamos considerar o *Problema das Rotas* [4]. Esse problema consiste em, dado um mapa de vias (veja a Figura 5), uma cidade de origem e uma cidade de destino, encontrar uma rota de distância mínima entre essas duas cidades. Nesse domínio, o agente é capaz de viajar de uma cidade à outra e, portanto, suas ações podem ser especificadas conforme segue[9]:

$via(a, b, 7)$
$via(a, c, 9)$
$via(a, d, 3)$
$via(b, f, 3)$
$via(b, i, 4)$
$via(c, j, 5)$
$via(d, e, 1)$
$via(f, g, 2)$
$via(g, h, 3)$
$via(i, k, 5)$
$via(j, l, 6)$
$via(k, l, 4)$
$oper(vai(A, B), A, B, C) \leftarrow via(A, B, C)$
$oper(vai(A, B), A, B, C) \leftarrow via(B, A, C)$

[9] Declarar as vias em separado é necessário porque as elas são bidirecionais.

Custo de caminho. Quando as ações têm custo, o custo de um caminho $[a_1, a_2, \ldots, a_n]$ é $\sum_{i=1}^{n} g(a_i)$, onde $g(a_i)$ é o custo da ação a_i. Por exemplo, de acordo com a Figura 5, o custo do caminho $[vai(a,d), vai(d,e)]$ é $3 + 1 = 4$.

Numa árvore de busca, todo estado s está associado a um caminho que leva do estado inicial s_0 até s. Então, podemos definir o *custo de um estado numa árvore de busca* como sendo o custo do caminho que leva até ele.

5.2 Busca pelo menor custo

O algoritmo de *busca pelo menor custo* combina os comportamentos da busca em largura e da busca em profundidade. Nesse tipo de busca, cada vez que um estado é expandido, um custo é associado a cada um de seus sucessores. O algoritmo progride expandindo sempre o estado de menor custo, até que um estado meta seja encontrado.

O algoritmo de busca pelo menor custo é obtido através do uso de uma *fila de prioridades ascendente*[10] para guardar os estados a serem expandidos.

```
BuscaMenorCusto(A, s_0, G)
1  Γ ← ∅
2  Σ ← {s_0}
3  enquanto Σ ≠ ∅ faça
4    s ← removePrimeiro(Σ)
5    se s ∈ G então devolva caminho(s)
6    Γ ← Γ ∪ {s}
7    insereEmOrdem(sucessoresG(s, A) − Γ, Σ)
8  devolva fracasso
```

No algoritmo BuscaMenorCusto, a função *removePrimeiro* e o procedimento *insereEmOrdem* servem para tratar o conjunto Σ como uma fila de prioridades. Além disso, a função *sucessoresG* devolve uma lista de estados sucessores e seus respectivos custos (necessários para estabelecer a ordem dos estados na fila de prioridades ascendente), que são dados pela função $g(s)$.

Exemplo 4. Sejam $\mathcal{A}$ o conjunto de ações para o Problema das Rotas, $s_0 = a$ e $\mathcal{G} = \{[k]\}$. O rastreamento de BuscaMenorCusto$(\mathcal{A}, s_0, \mathcal{G})$ produz a árvore de busca da Figura 6, onde os estados estão numerados na ordem em que são expandidos durante a busca e cada um deles tem um custo associado.

Observe que o algoritmo pula de uma ramo para outro da árvore, misturando busca em largura com busca em profundidade. Observe também que, quando o estado $i : 11$ é expandido, o estado meta $k : 16$ é gerado. Entretanto, em vez de parar a busca, o algoritmo prossegue expandindo os estados $g : 12$, $j : 14$ e $h : 15$, que têm custos menores. Isso é necessário porque, eventualmente, a expansão de um desses estados poderia gerar um estado k com custo menor que 16. □

[10] Nesse tipo de fila, os itens são mantidos em ordem crescente.

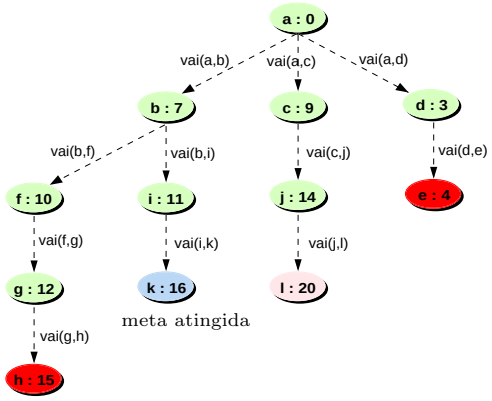


Figura 6. Árvore de busca gerada pelo algoritmo BuscaMenorCusto

Exercício 11 *Desenhe a árvore de busca produzida, para o Problema das Rotas, quando o algoritmo* BuscaMenorCusto *é chamado com* $s_0 = k$ *e* $\mathcal{G} = [a]$. □

Exercício 12 *Explique o que acontece quando o algoritmo* BuscaMenorCusto *trabalha com um conjunto de ações que tenham todas o mesmo custo.* □

Exercício 13 *Considere o mapa de vôos da Figura 7, representado pelos operadores a seguir:*

$voo(a, b, 1)$
$voo(a, c, 9)$
$voo(a, d, 4)$
$voo(b, c, 7)$
$voo(b, e, 6)$
$voo(b, f, 1)$
$voo(c, f, 7)$
$voo(d, f, 4)$
$voo(d, g, 5)$
$voo(e, h, 9)$
$voo(f, h, 4)$
$voo(g, h, 1)$

$oper(voa(A, B), A, B, C) \leftarrow voo(A, B, C)$
$oper(voa(A, B), A, B, C) \leftarrow voo(B, A, C)$

Sejam $\mathcal{A}$ *o conjunto de ações acima,* $s_0 = a$ *e* $\mathcal{G} = [h]$*:*

(a) Desenhe a árvore de busca produzida por BuscaMenorCusto$(\mathcal{A}, s_0, \mathcal{G})$.

(b) Mostre que, usando os operadores na ordem declarada acima, os algoritmos de busca em largura e em profundidade encontram soluções de custo superior àquele encontrado pela busca de menor custo, quando $s_0 = a$ *e* $\mathcal{G} = [h]$. □

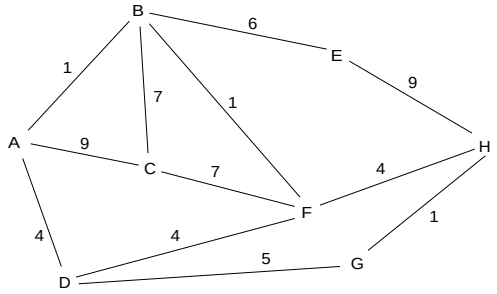


Figura 7. Mapa de vôos entre cidades

5.3 Função heurística

Uma *função heurística* é uma função que *estima* o custo mínimo de um caminho (desconhecido) que leva de um determinado estado a um estado meta.

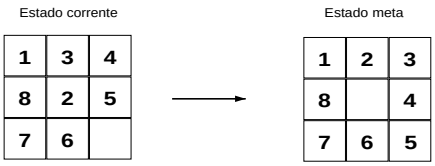
Seja h^* uma função que calcula o custo mínimo exato[11] de um caminho que leva de um determinado estado a um estado meta. Formalmente, uma função heurística pode ser qualquer função h que apresente as seguintes propriedades:

- $h(s) = 0$ se e somente se $s \in \mathcal{G}$ e
- para todo $s \in \mathcal{S}$, $h(s) \leq h^*(s)$.

Essas propriedades garantem que a estimativa dada pela função heurística seja *admissível*, ou seja, que nunca superestime o custo real de uma solução.

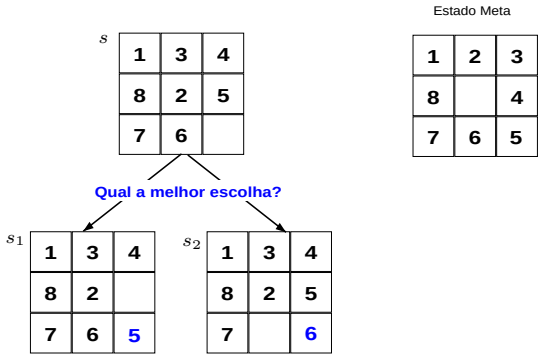
Note que as funções heurísticas dependem do domínio de aplicação, bem como da criatividade e experiência de quem a projeta.

Exemplo 5. O problema do *Quebra-Cabeça de 8* [4] consiste em movimentar as peças do quebra-cabeça horizontal ou verticalmente (para ocupar a posição vazia adjacente à peça) de modo que a configuração final seja alcançada:



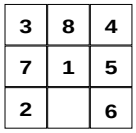
Por exemplo, expandindo o estado corrente acima, temos:

[11] Evidentemente, se tivéssemos essa função, na prática, não precisaríamos de algoritmos de busca. Assim, a suposição da existência de h^* é apenas teórica



Agora, usando uma função heurística, o algoritmo de busca deveria expandir o melhor entre esses dois estados sucessores. Mas como decidir qual deles é o melhor? Uma possibilidade é verificar o quão longe cada peça encontra-se de sua posição na configuração final e apontar como melhor estado aquele cuja soma das distâncias é mínima. Por exemplo, no estado s_1, as peças 1, 5, 6, 7 e 8 já estão em suas posições finais. Para as peças 2, 3 e 4, a distância é 1. Portanto, $h(s_1) = 3$. Analogamente, temos $h(s_2) = 5$. Esses valores indicam que uma solução a partir do estado s_1 pode ser obtida com no mínimo mais três expansões, enquanto que uma solução a partir de s_2 requer no mínimo mais cinco expansões. Então, evidentemente, o algoritmo de busca deve expandir o estado s_1. □

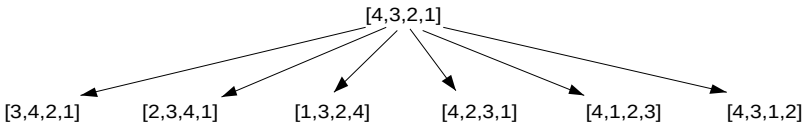
Exercício 14 *Usando soma de distâncias, calcule h para o estado a seguir:*



□

Exercício 15 *Explique porque a soma das distâncias das peças é uma heurística admissível para o Problema do Quebra-Cabeça de 8.* □

Exercício 16 *Considere um domínio onde o agente é capaz de permutar dois itens quaisquer de uma lista e o problema consiste em, dada uma lista com os itens numa ordem arbitrária, encontrar uma seqüência de permutações que colocam os itens da lista em ordem crescente. Por exemplo, suponha que o estado inicial seja a lista* $[4, 3, 2, 1]$. *Expandindo esse estado temos:*



Agora, o agente deveria escolher o melhor desses seis estados para continuar a busca. Intuitivamente, não é difícil de se perceber que os estados $[1, 3, 2, 4]$ *e* $[4, 2, 3, 1]$ *são os melhores; pois, a partir desses estados, podemos atingir a meta permutando apenas mais um par de itens.*

Considere a função heurística $h(s) = |\{(a_i, a_j) : a_i, a_j \in s \wedge i \leq j\}|$*, que contabiliza o número de inversões na lista representada por um estado* s*:*

(a) Calcule o valor de h *para cada um dos seis estados sucessores acima.*

(b) Explique porquê a função h *não é admissível para o Problema de Ordenação.*

(c) Tente encontrar uma outra heurística que seja admissível. □

5.4 Busca pela melhor estimativa

O algoritmo de *busca pela melhor estimativa* é semelhante ao algoritmo de busca pelo menor custo. A diferença é que, em vez de se basear no custo $g(s)$, do caminho já percorrido até s, para selecionar os estados a serem expandidos durante a busca, a busca pela melhor estimativa se baseia no custo estimado $h(s)$ do caminho que ainda precisa ser percorrido, a partir de s, até um estado meta.

BuscaMelhorEstimativa($\mathcal{A}, s_0, \mathcal{G}$)
1 $\Gamma \leftarrow \emptyset$
2 $\Sigma \leftarrow \{s_0\}$
3 `enquanto` $\Sigma \neq \emptyset$ `faça`
4 $s \leftarrow removePrimeiro(\Sigma)$
5 `se` $s \in \mathcal{G}$ `então devolva` $caminho(s)$
6 $\Gamma \leftarrow \Gamma \cup \{s\}$
7 $insereEmOrdem(sucessoresH(s, \mathcal{A}) - \Gamma, \Sigma)$
8 `devolva fracasso`

Nesse algoritmo, a função *sucessoresH* devolve uma lista de estados sucessores e seus respectivos custos estimados (necessários para estabelecer a ordem dos estados na fila de prioridades ascendente), que são dados pela função $h(s)$.

Exemplo 6. Vamos considerar novamente o *Problema das Rotas*, especificado na subseção 5.1. Como heurística, usaremos a distância em linha reta entre a cidade corrente e a cidade que se deseja atingir[12]. Vamos encontrar uma rota que leve da cidade a à cidade k e, para facilitar a exposição, vamos definir a função heurística da seguinte forma:

$h(a) = 15$
$h(b) = 7$
$h(c) = 6$
$h(d) = 14$

[12] Essa heurística pode ser calculada a partir das coordenadas de cada cidade no mapa.

$h(e) = 15$
$h(f) = 7$
$h(g) = 8$
$h(h) = 5$
$h(i) = 5$
$h(j) = 3$
$h(k) = 0$
$h(l) = 4$

O rastreamento de BuscaMelhorEstimativa$(\mathcal{A}, a, [k])$ produz a árvore de busca da Figura 8, onde os estados estão numerados na ordem em que são expandidos durante a busca e cada um deles tem uma heurística associada.

Observe que o uso da heurística orienta a busca de tal forma que algoritmo, praticamente, segue direto em direção à meta. Evidentemente, a busca pela melhor estimativa nem sempre é tão eficiente assim. Tudo vai depender do problema sendo resolvido e da heurística utilizada. Entretanto, apesar de eficiente, a busca pela melhor estimativa não pode garantir que uma solução de custo mínimo será encontrada. Nesse exemplo, a solução encontrada tem custo 24, um custo bem superior ao custo daquela solução encontrada pelo algoritmo de busca pelo menor custo (veja o exemplo 4). □

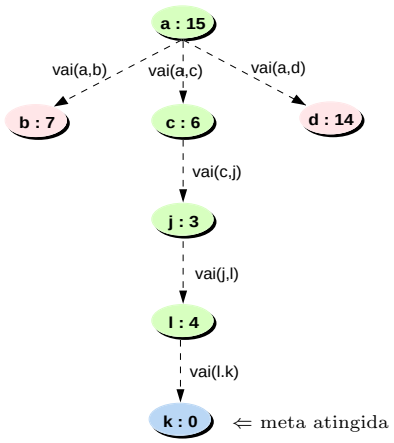


Figura 8. Árvore de busca gerada pelo algoritmo BuscaMelhorEstimativa

Exercício 17 *Considere o Problema do Quebra-cabeça de 8, apresentado no exemplo 5. Para esse problema, qual algoritmo seria mais apropriado: busca pelo menor custo ou busca pela melhor estimativa? Justifique.* □

Exercício 18 *Desenhe a árvore de busca, para o Problema das Rotas, quando o algoritmo* BuscaMelhorEstimativa *é chamado com* $s_0 = k$ *e* $\mathcal{G} = \{[a]\}$. □

5.5 Busca A^*

Conforme vimos, a busca pelo menor custo minimiza o custo do caminho percorrido até o estado corrente, enquanto a busca pela melhor estimativa tenta minimizar o custo estimado do caminho a ser percorrido do estado corrente até um estado meta. Para garantir a solução de custo mínimo, a busca pelo menor custo acaba tendo que examinar uma grande quantidade de estados no espaço de estados do problema, podendo ser muito ineficiente. Por outro lado, a busca pela melhor estimativa reduz o espaço de busca consideravelmente; porém, não consegue garantir uma solução de custo mínimo. Felizmente, é possível combinar essas duas estratégias de busca num mesmo algoritmo, denominado A^* [3,4].

O algoritmo de *busca* A^* minimiza o custo total do caminho percorrido, desde o estado inicial até um estado meta, usando uma função da forma $f(s) = g(s) + h(s)$, onde $g(s)$ é uma função de custo e $h(s)$ é uma função heurística.

```
BuscaA*(A, s_0, G)
1  Γ ← ∅
2  Σ ← {s_0}
3  enquanto Σ ≠ ∅ faça
4    s ← removePrimeiro(Σ)
5    se s ∈ G então devolva caminho(s)
6    Γ ← Γ ∪ {s}
7    insereEmOrdem(sucessoresF(s, A) − Γ, Σ)
8  devolva fracasso
```

Nesse algoritmo, a função *sucessoresF* devolve uma lista de estados sucessores e seus respectivos custos $g(s) + h(s)$ (necessários para estabelecer a ordem dos estados na fila de prioridades ascendente), que são dados pela função $f(s)$.

Exemplo 7. Vamos continuar com o *Problema das Rotas*, cujos custos das ações foram especificados na subseção 5.1 e cujos valores da função heurística foram definidos no exemplo 6.

O rastreamento de Busca$A^*(\mathcal{A}, a, [k])$ produz a árvore de busca da Figura 9, onde os estados estão numerados na ordem em que são expandidos durante a busca e cada um deles tem associada a soma de seu custo com a sua heurística.

Observe que o uso da função custo $g(s)$ garante que o algoritmo encontre uma solução de custo mínimo (como na busca pelo menor custo) e, ao mesmo tempo, o uso da heurística $h(s)$ reduz o espaço de busca explorado, melhorando a eficiência do algoritmo (como na busca pela melhor estimativa). □

Exercício 19 *Desenhe a árvore de busca, para o Problema das Rotas, quando o algoritmo* BuscaA^* *é chamado com* $s_0 = k$ *e* $\mathcal{G} = \{[a]\}$. □

Exercício 20 *No Quebra-Cabeça de 8, cada ação tem custo 1. Usando a heurística da soma das distâncias, desenhe a árvore de busca gerada pelo algoritmo* A^**, quando o estado inicial do quebra-cabeça é* $[[1, 2, 3], [b, 6, 4], [8, 7, 5]]$. □

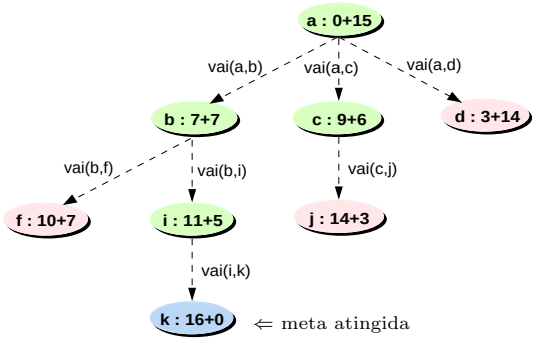


Figura 9. Árvore de busca gerada pelo algoritmo BUSCAA^*

A Implementação dos algoritmos de busca em PROLOG

O código a seguir implementa em SWI-PROLOG, versão 5.2.7, um algoritmo de busca genérico que deve ser parametrizado com o tipo de busca que se deseja realizar, conforme as instruções dadas no comentário.

```
/*-------------------------------------------------------------------------------------+
| Algoritmo de busca genérico (selecione uma estratégia)                               |
| Para executar, especifique o problema (veja anexo B) e digite:                       |
| ?- busca. <enter>                                                                    |
+-------------------------------------------------------------------------------------*/

% selecione a estratégia de busca desejada
% 1 - largura
% 2 - profundidade
% 3 - menor custo
% 4 - melhor estimativa
% 5 - A*
% modificando a linha a seguir

estratégia(5).

busca :- inicial(So), busca([0-0-0-So-[]],[]).

busca([_-G-_-Estado-Caminho|_],_) :-
   meta(M), member(Estado,M), !,
   reverse(Caminho,Solução),
   estratégia(T),
   (T=1 -> N = 'busca em largura' ;
    T=2 -> N = 'busca em profundidade' ;
    T=3 -> N = 'busca pelo menor custo' ;
    T=4 -> N = 'busca pela melhor estimativa' ;
    T=5 -> N = 'busca A'),
   format('~nEstratégia........: ~w', [N]),
   format('~nCusto da solução..: ~w',[G]),
   format('~nSeqüência de ações: ~w~n',[Solução]).

busca([_-G-_-Estado-Caminho|ListaEstados],Expandidos) :-
   sucessores(Estado,Sucessores),
   union([Estado],Expandidos,NovosExpandidos),
   subtract(Sucessores,NovosExpandidos,SucessoresNaoExpandidos),
   estende(G,Estado,Caminho,SucessoresNaoExpandidos,NovosEstados),
   insere(NovosEstados,ListaEstados,NovaListaEstados),
```

```
   busca(NovaListaEstados,NovosExpandidos).

sucessores(Estado,Sucessores) :-
   findall(X,oper(_,Estado,X,_),Sucessores).


estende(_,_,_,[],[]).

estende(X,E,C,[S|Ss],[F-G-H-S-[A|C]|Ps]) :-
   oper(A,E,S,Y),
   G is X+Y,
   h(S,H),
   estratégia(Tipo),
   (Tipo=1 -> F is 0 ;
    Tipo=2 -> F is 0 ;
    Tipo=3 -> F is G ;
    Tipo=4 -> F is H ;
    Tipo=5 -> F is G+H),
   estende(X,E,C,Ss,Ps).


insere(NovosEstados,ListaEstados,NovaListaEstados) :-
   estratégia(Tipo),
   (Tipo=1 -> append(ListaEstados,NovosEstados,NovaListaEstados) ;
    Tipo=2 -> append(NovosEstados,ListaEstados,NovaListaEstados) ;
              append(ListaEstados,NovosEstados,Lista), sort(Lista,NovaListaEstados)).
```

B Exemplo de especificação de problema em PROLOG

```
/*-----------------------------------------------------------------------------+
| Especificação do Problema das Rotas                                          |
+-----------------------------------------------------------------------------*/

% defina as ações do problema
% atribua custo 1 se as ações não tiverem custo

  via(a,b,7).  via(a,c,9).  via(a,d,3).  via(b,f,3).  via(b,i,4).  via(c,j,5).
  via(d,e,1).  via(f,g,2).  via(g,h,3).  via(i,k,5).  via(j,l,6).  via(k,l,4).

  oper(vai(A,B),A,B,C) :- via(A,B,C).
  oper(vai(A,B),A,B,C) :- via(B,A,C).

% defina o estado inicial e a meta

  inicial(a).
  meta([k]).

% defina a função heurística
% declare h(_,0), se não houver heurística a ser utilizada

  h(a,15).  h(b,7).  h(c,6).   h(d,14).  h(e,15).  h(f,7).
  h(g,8).   h(h,5).  h(i,5).   h(j,3).   h(k,0).   h(l,4).
```

Referências

1. AMBLE, T. *Logic Programming and Knowledge Engineering*, Addison-Wesley, 1987.
2. BRATKO, I. *Prolog - Programming for Artificial Intelligence*, Addison-Wesley, 1990.
3. RICH, E. AND KNIGHT, K. *Inteligência Artificial*, 2^a ed., Makron Books, 1995.
4. RUSSELL, S. AND NORVIG, P. *Artificial Intelligence - A Modern Approach*, Prentice-Hall, 1995.
5. STERLING, L. AND SHAPIRO, E. *The Art of Prolog*, MIT Press, 1986.