==

Algoritmos de Busca Heurística (PARTE 2)

Busca Heurística em Grafos–OU

Prof. Alexandre Direne – DInf-UFPR

==

Recomendação de Bibliografia:

- Artificial Intelligence: A Modern Approach. Stuart Russell e Peter Norvig. Third Edition, Prentice Hall, 2010.
- Programming in Prolog. William F. Clocksin and C. S. Mellish. Springer-Verlag, 1987.
- Inteligência Artificial: Ferramentas e Teorias. Guilherme Bittencourt. Terceira Ed., Editora da UFSC, 2006.
- Artificial Intelligence. Elaine Rich e Kevin Knight. Second Edition, McGraw Hill, 1993.
- Artificial Intelligence. Patrick H. Winston. Second Edition, Addison-Wesley, 1993.
- Introduction to Artificial Intelligence. Eugene Charniak e Drew McDermott. Addison-Wesley, 1989.
- Inteligência Artificial - um curso prático. G. Arariópia. Editora LTC - Livros Técnicos e Científicos, 1989 (do qual a maior parte deste arquivo foi composto ***).

Recomendação de Páginas Web:

`http://www.cs.dartmouth.edu/~brd/Teaching/AI/Lectures/Summaries/search.html`

`http://www.decom.ufop.br/prof/guarda/CIC250/index.htm`

`http://aima.cs.berkeley.edu/`

`http://aima.cs.berkeley.edu/newchap05.pdf`

Recomendação de Software:

- Pacote de software para a Programação em Inteligência Artificial (ambiente Poplog):

  `http://www.cs.bham.ac.uk/research/poplog/freepoplog.html`

- Pacote de software para a Programação em Prolog (SWI–Prolog):

  `http://www.swi-prolog.org`

Busca Heurística em Grafos–OU

Os principais algoritmos de busca desta parte são as variações da busca em Grafos–OU. Tais variações se baseiam em uma série de formalismos que podem ser definidos como abordagens mais específicas de busca sobre o conceito de árvores de estados como espaços de solução de problemas. Estes formalismos estão apresentados nas seções seguintes, logo antes das descrições das variações dos algoritmos propriamente ditos.

1 Métodos de Busca Cega ou Força Bruta

1.1 Busca em Amplitude

Considere a ávore de busca genericamente apresentada na Figura 1. Observe o seguinte:

- O estado inicial da árvore está representado pelo nodo b e o estado final pelo nodo n.
- O percurso da árvore se inicia com a investigação dos nodos $c - d - e$, os quais estão distantes da raiz apenas um ramo de altura.
- Passa-se então para a investigação dos nodos $g - h - i - j - k$, distantes dois ramos de altura da raiz.
- Finalmente, investiga-se os nodos $n - m$.
- Para isso, o processo de busca em largura deve contar com uma $Fila$, a qual contém trajetórias à espera de sua expansão, de sua eliminação ou mesmo para anunciar a suspensão do processo de busca como um todo.
- Cada trajetória pode ser representada por uma lista de estruturas de dados (da linguagem Prolog), sendo que cada estrutura de dado tem o formato $r(R, N)$, onde R é o ramo (ou transformação atômica) que liga o nodo N ao pai dele.
- Por exemplo, na Figura 1, a trajetória que sai de b, passa por c e chega em h é representada por:

```
[ r(5,h), r(1,c), r(raiz,b) ]
```

- Note que, como a raiz não tem pai, a estrutura de dado $r(raiz, b)$ que a simboliza tem a palavra "raiz" no lugar do argumento reservado ao ramo.
- Depois do primeiro ciclo do processo de busca em amplitude, a $Fila$ possui o seguinte valor:

```
Fila = [ [ r(1,c), r(raiz,b) ],
         [ r(2,d), r(raiz,b) ],
         [ r(3,e), r(raiz,b) ] ]
```

- Neste ponto, o processo tem continuidade por meio da verificação da existência do estado final (ou estado meta) na trajetória que está na frete da $Fila$.
- Como ela não inclui o estado final, o processo cria uma lista contendo todas as possíveis maneiras de expandir a trajetória da frente de $Fila$ para conter os filhos de c.
- Essa lista é chamada de lista de extensões e contém o seguinte valor no momento:

```
ListaExtensoes = [ [ r(4,g), r(1,c), r(raiz,b) ],
                   [ r(5,h), r(1,c), r(raiz,b) ],
                   [ r(6,i), r(1,c), r(raiz,b) ] ]
```

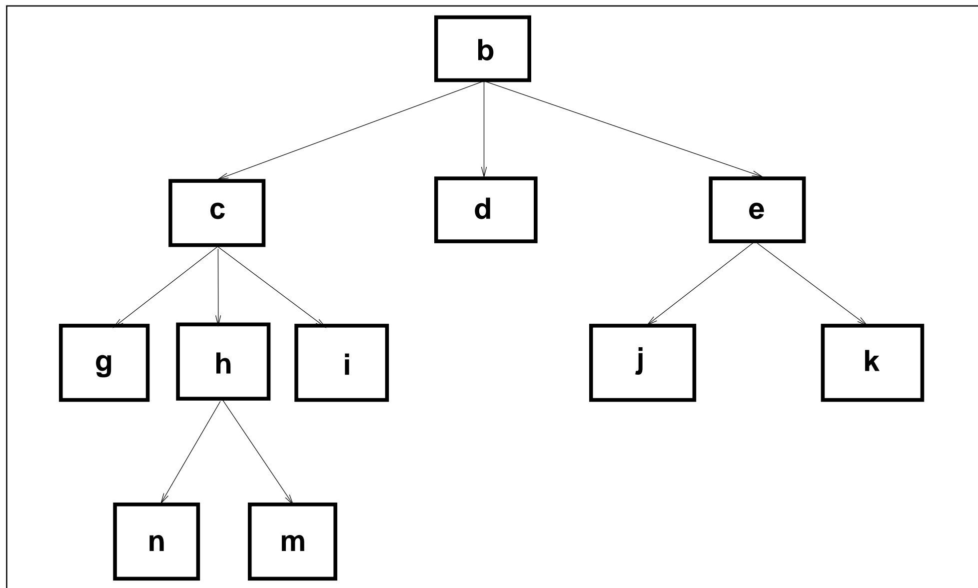

Figura 1: Exemplo de árvore com ordem de busca em aplitude

- As trajetórias da lista de extensões levam a dois ramos de altura em distância da raiz, fazendo com que sejam investigados apenas depois das que estão na $Fila$.

- Isto significa que a lista de extensões deve ser concatenada atrás de todas as trajetórias que estão na $Fila$, a qual ficará assim:

```
FilaEstendida = [ [ r(2,d), r(raiz,b) ],
                  [ r(3,e), r(raiz,b) ],
                  [ r(4,g), r(1,c), r(raiz,b) ],
                  [ r(5,h), r(1,c), r(raiz,b) ],
                  [ r(6,i), r(1,c), r(raiz,b) ] ]
```

- Note que a trajetória $[r(1,c), r(raiz,b)]$ foi retirada da $Fila$ porque já se verificou que ela não possui o estado final.

- A próxima trajetória a ser investigada é $[r(2,d), r(raiz,b)]$, a qual também não possui o estado final, sendo igualmente retirada da $Fila$.

- Neste caso, nenhuma lista de extensões é criada pois o nodo d não tem filhos.

- A $Fila$ se torna então:

```
Fila= [ [ r(3,e), r(raiz,b) ],
        [ r(4,g), r(1,c), r(raiz,b) ],
        [ r(5,h), r(1,c), r(raiz,b) ],
        [ r(6,i), r(1,c), r(raiz,b) ] ]
```

- A próxima trajetória a ser investigada é $[r(3,e), r(raiz,b)]$, a qual também não possui o estado final, sendo igualmente retirada da $Fila$.

- Neste ponto, monta-se a lista de extensões, a qual é colocada no fim da $Fila$. Seu valor agora é o seguinte:

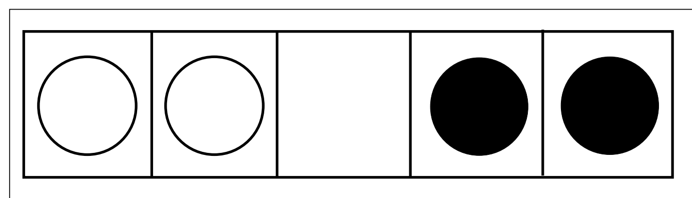

Figura 2: Um estado inicial para o problema de movimentação das fichas

```
FilaEstendida = [ [ r(4,g), r(1,c), r(raiz,b) ],
                  [ r(5,h), r(1,c), r(raiz,b) ],
                  [ r(6,i), r(1,c), r(raiz,b) ],
                  [ r(7,j), r(3,e), r(raiz,b) ],
                  [ r(8,k), r(3,e), r(raiz,b) ] ]
```

- A trajetória $[r(4,g), r(1,c), r(raiz,b)]$ não possui o estado final e é retirada da $Fila$, fazendo o processo seguir de maneira semelhante.

- Depois de vários ciclos, finalmente, temos a $Fila$ no seguinte estado:

```
Fila = [ [ r(9, n), r(5,h), r(1,c), r(raiz,b) ],
         [ r(10,m), r(5,h), r(1,c), r(raiz,b) ] ]
```

- Agora a primeira trajetória da $Fila$ possui o estado final n e, desta forma, pode ser considerada como a solução do problema de busca.

- A resposta que deverá ser impressa pela máquina é algo como:

$$b \xrightarrow{1} c \xrightarrow{5} h \xrightarrow{9} n$$

Em resumo, o algoritmo de busca em amplitude pode ser resumidamente descrito assim:

- retira-se a trajetória que está na cabeça da $Fila$;

- se a trajetória da cabeça não inclui o estado final, uma lista (possivelmente vazia) é composta com as trajetórias expandidas (sem repetição de estados) a partir da trajetória da cabeça da $Fila$ (a qual é totalmente descartada em seguida);

- a lista de trajetórias expandidas é então concatenada no final da $Fila$;

- se a trajetória da cabeça inclui o estado final, o conteúdo da trajetória é impresso e o algoritmo pára.

Para ilustrar a implementação do algoritmo de busca em amplitude, será considerado o problema de como posicionar duas fichas pretas entre duas fichas brancas. Para especificar melhor tal problema, tome como objeto de apoio uma bandeja que possui cinco lacunas de encaixe. Em seu estado inicial, as quatro fichas podem ocupar qualquer posição relativa das lacunas, ficando sempre uma lacuna vazia. A Figura 2 mostra uma das possíveis configurações de Estados Iniciais para o problema referido.

As duas operações atômicas possíveis são as seguintes:

- fazer com que uma ficha deslize para a lacuna vazia, se a lacuna vazia estiver imediatamente ao lado (esquerdo ou direito) da ficha;

- fazer com que uma ficha salte para a lacuna vazia, se o salto ocorrer por cima de uma única ficha.

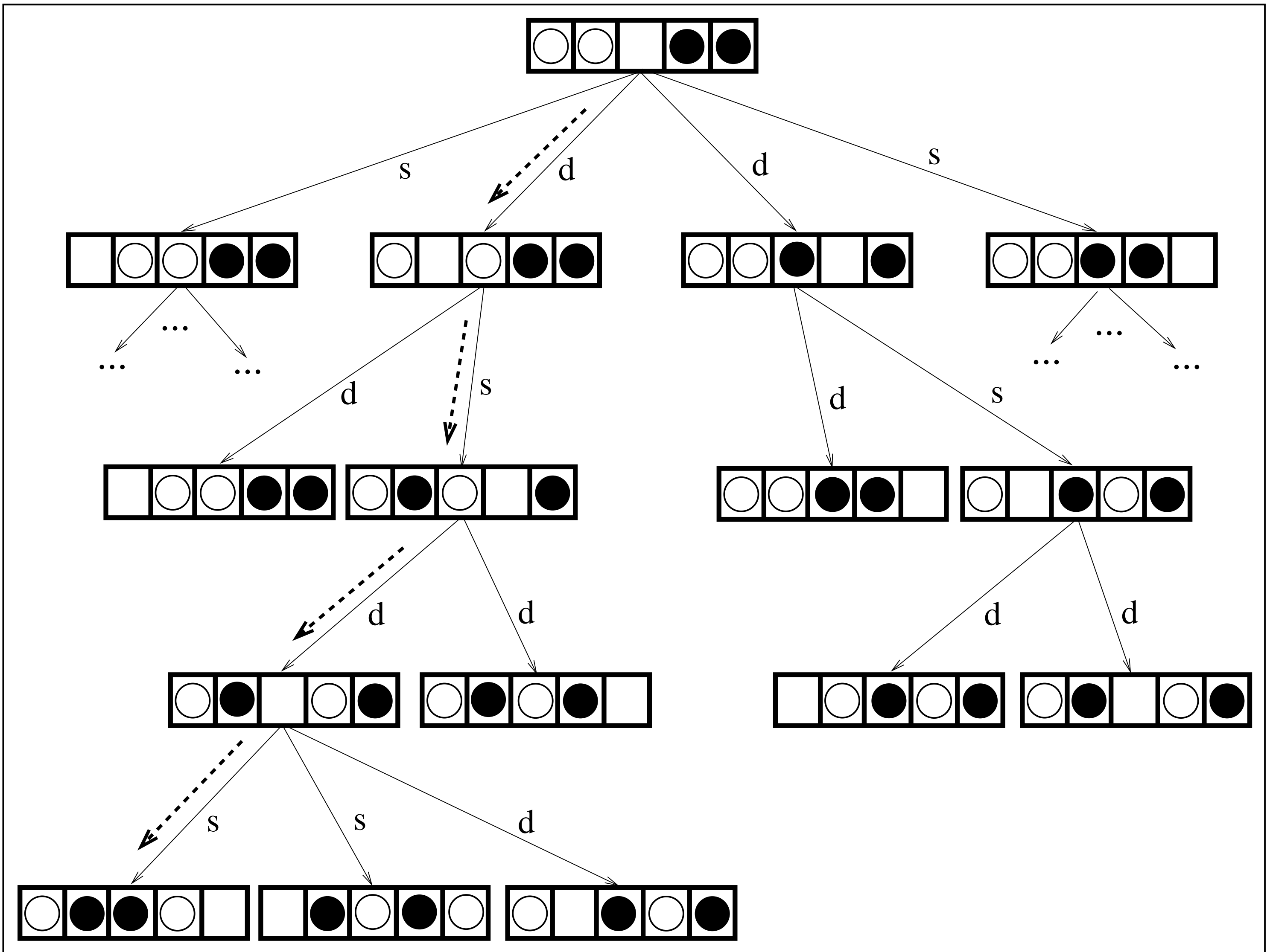

Figura 3: Fragmento da árvore de busca para o problema de movimentação das fichas

Por enquanto, para simplificar a apresentação dos conceitos de implementação, é importante notar que os custos de ambas as operação atômicas são os mesmos.

A Figura 3 mostra um fragmento da árvore de busca para o problema de movimentação das fichas o qual inclui a ocorrência em altura mínima de um dos possíveis Estados Finais.

O programa em Prolog que segue é capaz de realizar a busca em amplitude. Um dos possíveis Estados Iniciais está listado (como comentário) no final do código e coincide com o Estado Inicial da Figura 2.

```
% ------------------------------------------------------------
%   PREDICADOS UTILITARIOS PARA A BUSCA (PROBLEMA DAS FICHAS)
% ------------------------------------------------------------

concatenadas([], L, L) :-
  !.
concatenadas([X|C1], L2, [X|C3]) :-
  concatenadas(C1, L2, C3).

pertence_a(X, [X|_]) :-
  !.
```

```
pertence_a(X, [_|C]) :-
  pertence_a(X, C).

trajetoria_impressa([ r(raiz,Raiz) ]) :-
  write('Estado Inicial: '),
  write(Raiz),
  write('. \n'),
  !.
trajetoria_impressa([ r(Operacao,Nodo) | Resto ]) :-
  trajetoria_impressa(Resto),
  write(Operacao),
  write(' para obter o estado: '),
  write(Nodo),
  write('. \n').

transformado([v, A, B, C, D], operacao(deslisa), [A, v, B, C, D]).
transformado([A, v, B, C, D], operacao(deslisa), [v, A, B, C, D]).
transformado([A, v, B, C, D], operacao(deslisa), [A, B, v, C, D]).
transformado([A, B, v, C, D], operacao(deslisa), [A, v, B, C, D]).
transformado([A, B, v, C, D], operacao(deslisa), [A, B, C, v, D]).
transformado([A, B, C, v, D], operacao(deslisa), [A, B, v, C, D]).
transformado([A, B, C, v, D], operacao(deslisa), [A, B, C, D, v]).
transformado([A, B, C, D, v], operacao(deslisa), [A, B, C, v, D]).
transformado([v, A, B, C, D], operacao(salta), [B, A, v, C, D]).
transformado([A, v, B, C, D], operacao(salta), [A, C, B, v, D]).
transformado([A, B, v, C, D], operacao(salta), [v, B, A, C, D]).
transformado([A, B, v, C, D], operacao(salta), [A, B, D, C, v]).
transformado([A, B, C, v, D], operacao(salta), [A, v, C, B, D]).
transformado([A, B, C, D, v], operacao(salta), [A, B, v, D, C]).

e_estado_final([v, o, *, *, o]).
e_estado_final([o, v, *, *, o]).
e_estado_final([o, *, v, *, o]).
e_estado_final([o, *, *, v, o]).
e_estado_final([o, *, *, o, v]).

trajetoria_expandida([r(Ramo,Nodo) | Resto], [r(Op,Filho), r(Ramo,Nodo) | Resto]) :-
  transformado(Nodo, operacao(Op), Filho),
  not(produz_ciclo(Filho, [r(Ramo,Nodo) | Resto])).

produz_ciclo(Estado, Trajetoria) :-
  pertence_a(r(_,Estado), Trajetoria).

adjacentes(Trajetoria, Adjacentes) :-
  findall(Ti, trajetoria_expandida(Trajetoria, Ti), Adjacentes),
  !.
adjacentes( _ , [] ).

possui_estado_final([ r(_,Nodo) | _ ]) :-
  e_estado_final(Nodo).
```

```
% ------------------------------------------------------------
%                   BUSCA EM AMPLITUDE
% ------------------------------------------------------------

solucao_computada_em_amplitude([ Solucao | _ ], Solucao) :-
  possui_estado_final(Solucao),
  !.
solucao_computada_em_amplitude([ Trajetoria | Fila ], Solucao) :-
  adjacentes(Trajetoria, Lista_de_Trajetorias_Expandidas),
  concatenadas(Fila, Lista_de_Trajetorias_Expandidas, Fila_Expandida),
  solucao_computada_em_amplitude(Fila_Expandida, Solucao).

% *** Exemplo de Estado Inicial: [o, o, v, *, *]

solucao_da_busca_em_amplitude_a_partir_do(Estado_Inicial) :-
  solucao_computada_em_amplitude([ [ r(raiz,Estado_Inicial) ] ], Trajetoria),
  trajetoria_impressa(Trajetoria).
```

Para ilustrar a aplicação do predicado `solucao_da_busca_em_amplitude_a_partir_do` basta compilar seu código e executar a seguinte consulta:

```
?- solucao_da_busca_em_amplitude_a_partir_do([o, o, v, *, *]).
Estado Inicial: [o, o, v, *, *].
deslisa para obter o estado: [o, v, o, *, *].
salta para obter o estado: [o, *, o, v, *].
deslisa para obter o estado: [o, *, v, o, *].
salta para obter o estado: [o, *, *, o, v].
yes
```

1.2 Busca em Profundidade

Na busca em profundidade, as trajetórias recém produzidas são incluídas em uma *Pilha* e não em uma fila. Na ávore de busca genericamente apresentada na Figura 1, o Estado Final n pode ser encontrado efetuando-se a busca em profundidade. Suponha que as seguintes trajetórias estejam na *Pilha* para serem examinadas ao final do primeiro ciclo de busca:

```
Pilha = [ [ r(1,c), r(raiz,b) ],
          [ r(2,d), r(raiz,b) ],
          [ r(3,e), r(raiz,b) ] ]
```

Retirando-se a trajetória do topo da *Pilha*, verifica-se que ela não inclui o Estado Final. Empilha-se então as trajetórias extandidas com os filhos do nodo c, as quais deixam a *Pilha* no seguinte estado:

```
NovaPilha = [ [ r(4,g), r(1,c), r(raiz,b) ],
              [ r(5,h), r(1,c), r(raiz,b) ],
              [ r(6,i), r(1,c), r(raiz,b) ],
              [ r(2,d), r(raiz,b) ],
              [ r(3,e), r(raiz,b) ] ]
```

Como a trajetória do topo de *Pilha* não inclui o Estado Final e não pode ser expandida, a *Pilha* fica no seguinte estado:

```
Pilha = [ [ r(5,h), r(1,c), r(raiz,b) ],
          [ r(6,i), r(1,c), r(raiz,b) ],
          [ r(2,d), r(raiz,b) ],
          [ r(3,e), r(raiz,b) ] ]
```

Agora, apesar da trajetória do topo de *Pilha* não incluir o Estado Final, uma de suas formas expandidas possui. No final do processo de busca, a *Pilha* está assim:

```
Pilha = [ [ r(9,n),  r(5,h), r(1,c), r(raiz,b) ],
          [ r(10,m), r(5,h), r(1,c), r(raiz,b) ],
          [ r(6,i), r(1,c), r(raiz,b) ],
          [ r(2,d), r(raiz,b) ],
          [ r(3,e), r(raiz,b) ] ]
```

Uma das variações de implementação de programa em Prolog que é capaz de realizar a busca em profundidade mais completa, adotando a eliminação de trajetórias que possuem estados repetidos, está listado abaixo. Note que os predicados <u>utilitários</u> são exatamente os mesmos definidos no programa que implementa a buca em amplitude (por isso não foram repetidos aqui).

```
% --------------------------------------------------------------
% BUSCA EM PROFUNDIDADE (COM PILHA EXPLICITA DE TRAJETORIAS)
% --------------------------------------------------------------

solucao_computada_em_profundidade([ Solucao | _ ], Solucao) :-
  possui_estado_final(Solucao),
  !.
solucao_computada_em_profundidade([ Trajetoria | Pilha ], Solucao) :-
  adjacentes(Trajetoria, Lista_de_Trajetorias_Expandidas),
  concatenadas(Lista_de_Trajetorias_Expandidas, Pilha, Pilha_Expandida),
  solucao_computada_em_profundidade(Pilha_Expandida, Solucao).

% *** Exemplo de Estado Inicial: [o, o, v, *, *]

solucao_da_busca_em_profundidade_a_partir_do(Estado_Inicial) :-
  solucao_computada_em_profundidade([ [ r(raiz,Estado_Inicial) ] ], Trajetoria),
  trajetoria_impressa(Trajetoria).
```

Para ilustrar a aplicação do predicado `solucao_da_busca_em_profundidade_a_partir_do` basta compilar seu código e executar a seguinte consulta:

```
?- solucao_da_busca_em_profundidade_a_partir_do([o, o, v, *, *]).
Estado Inicial: [o, o, v, *, *].
deslisa para obter o estado: [o, v, o, *, *].
salta para obter o estado: [o, *, o, v, *].
deslisa para obter o estado: [o, *, v, o, *].
deslisa para obter o estado: [o, v, *, o, *].
deslisa para obter o estado: [v, o, *, o, *].
```

```
salta para obter o estado: [*, o, v, o, *].
deslisa para obter o estado: [*, v, o, o, *].
salta para obter o estado: [*, o, o, v, *].
deslisa para obter o estado: [*, o, o, *, v].
salta para obter o estado: [*, o, v, *, o].
deslisa para obter o estado: [*, v, o, *, o].
deslisa para obter o estado: [v, *, o, *, o].
salta para obter o estado: [o, *, v, *, o].
yes
```

Note os seguites pontos:

- a solução do mesmo problema é bem mais longa do que a encontrada pela busca em amplitude;
- isso ocorre pois as trajetórias recém-expandidas são sempre mais longas (possuem um ramo a mais) e são elas as inseridas no topo da *Pilha* para serem investigadas primeiro do que as mais curtas. Na prática, este procedimento leva, em geral, o algoritmo a encontrar soluções mais longas;
- tal desvantagem é compensada pelo fato de "pilhas" serem mais eficientemente implementadas em computadores do que "filas", tanto por gastarem menos tempo de manipulação de sua estrutura de dado quanto por economizarem memória (apesar do programa anterior não economizar memória como deveria);
- isso significa que a linguagem Prolog pode ser utilizada de forma mais adequada ainda para a implementação da busca em profundidade do que mostrou o programa acima (economizando memória ao ponto de alocar apenas o espaço de uma trajetória).

Uma segunda maneira de implementação da busca em profundidade está listada no programa em Prolog que segue. Note que ele não manipula explicitamente nenhuma *Pilha*.

```
% ----------------------------------------------------------------
% BUSCA EM PROFUNDIDADE (SEM A PILHA DE TRAJETORIAS)
% ----------------------------------------------------------------

solucao_computada_em_profundidade(Solucao, Solucao) :-
  possui_estado_final(Solucao),
  !.
solucao_computada_em_profundidade(Trajetoria, Solucao) :-
  trajetoria_expandida(Trajetoria, T2),
  solucao_computada_em_profundidade(T2, Solucao).

% *** Exemplo de Estado Inicial: [o, o, v, *, *]

solucao_da_busca_em_profundidade_sem_pilha_a_partir_do(Estado_Inicial) :-
  solucao_computada_em_profundidade([ r(raiz,Estado_Inicial) ], Trajetoria),
  trajetoria_impressa(Trajetoria).
```

Observe duas coisas importantes:

- o primeiro termo do predicado `solucao_computada_em_profundidade` agora é apenas uma trajetória e não mais uma *Pilha*;

- a segunda cláusula do predicado `solucao_computada_em_profundidade` permite a retroação (backtracking) intra-cláusula.

Uma terceira variação de implementação da busca em profundidade ainda pode ser atingida por meio do uso explícito do predicado $fail$. Ele irá forçar a produção de todos os estados diretamente atingíveis a partir do estado atual (ou estado corrente).

2 Métodos de Busca Heurística

Os métodos de busca heurística fazem uso de conhecimento específico sobre o domínio do problema para guiar o processo de busca.

Muitos dos referidos métodos procuram atribuir uma nota a cada nodo da árvore de busca para representar uma estimativa (ou função heurística) do custo de transformação do Estado Atual no Estado Final.

Os conceitos básicos que são tipicamente necessários ao entendimento estão apresentados a seguir.

- Método de Busca Admissível: Um método de busca é admissível se ele sempre encontra a solução de menor custo (quando há ao menos uma solução).

- Função Heurística Admissível: função que nunca superestima o que chamamos genericamente de "distância conceitual" (custo) entre o Estado Atual e o Estado Final.

- Função Monotônica do Custo de Caminhos: função que nunca decresce (pode ser estável) em seu valor de custo real acumulado (custo realizado) para cobrir um caminho exclusivo qualquer entre o Estado Inicial e o Estado Atual (*i.e.*, não há caminho com custo negativo).

- Desigualdade Triangular da Função de Custo de Caminhos: uma função de custo real acumulado (custo realizado) obedece à desigualdade triangular se o custo do caminho direto entre dois nodos for sempre menor que a soma dos custos de dois ou mais caminhos indiretos que ligam os mesmos dois nodos.

- Função Heurística Mais Informada: h_2 é uma função mais informada do que h_1 se ambas forem admissíveis e ainda ocorrer que

$$\boxed{\forall E \notin ConjuntoEstadosFinais(h_2(E) \geq h_1(E))}$$

2.1 Algoritmo A^*

O Algoritmo A^* adota como base de orientação da busca a estimativa de transformação do Estado Inicial (E_i) até o no Estado Final (E_f), passando pelo Estado Atual (E_a). Tal estimativa pode ser expressa pela seguinte fórmula:

$$\boxed{f(E_a) = g(E_a) + h(E_a)}$$

Nesta fórmula, temos os seguintes detalhes:

- a função $g(E_a)$ expressa o custo (já conhecido) de transformação do E_i em E_a;

- a função $h(E_a)$ sintetiza a estimativa do custo (desconhecido) de transformação do E_a em E_f.

O processo de busca executado por meio do algoritmo A^* reorganiza, a cada ciclo, as trajetórias da Lista de Prioridades de modo que aquelas com menor $f(E_a)$ venham para a frente e sejam investigadas primeiro.

Um exemplo concreto de aplicação do A^* se destina a encontrar um caminho que leva, por exemplo, da cidade b à cidade s no mapa da Figura 4. Sobre o mapa, pode-se dizer que:

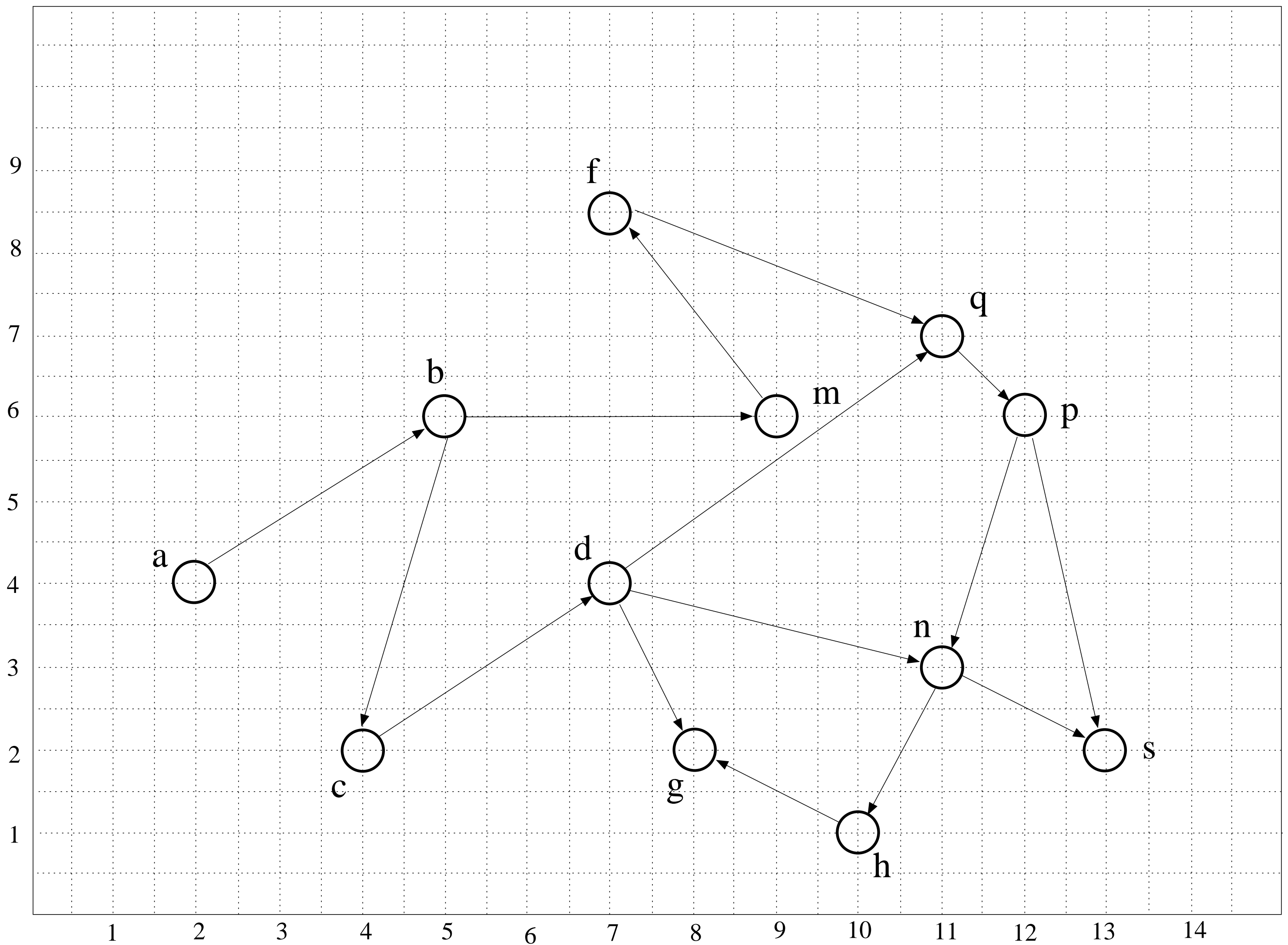

Figura 4: Mapa com ligações entre cidades e suas distâncias euclidianas

- as cidades são localizadas pelas coordenadas X e Y;
- o custo necessário para se realizar o percurso entre duas cidades que possuem ligação direta uma com a outra e que estão localizadas respectivamente nas coordenadas (X_1, Y_1) e (X_2, Y_2) é a distância Euclidiana:

$$\boxed{\sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)^2}}$$

Suponha que a investigação da máquina entre b e s atinja a cidade d. Pode-se dizer então que:

- o custo $g(d)$ para se chegar a ela é a soma dos custos para viajar pelas cidade que ligam b a d, ou seja, a distância Euclidiana de b até c somada com a de c até d;
- a estimativa de custo $h(d)$ para viajar da cidade d à cidade s também é asumida como a distância Euclidiana (em linha reta) entre elas (mesmo que não haja ligação direta).

O programa em Prolog abaixo realiza a busca A^* pela trajetória mínima entre duas cidades. Algumas de suas características são:

- o primeiro argumento da estrutura de dado $t(_,_,_)$ fornece $f(E_a)$;

- o segundo argumento da mesma estrutura fornece $g(E_a)$;
- o terceiro argumento de t fornece os ramos da trajetória, tal qual tínhamos adotado na busca cega;
- a lista de prioridades é sempre mantida em ordem crescente das trajetórias com base na chave $f(E_a)$ e, por isso, serão investigadas antes.

```
% --------------------------------------------------------------
%  PREDICADOS UTILITARIOS (PROBLEMA DO CAMINHO ENTRE 2 CIDADES)
% --------------------------------------------------------------

pertence_a(X, [X|_]) :-
  !.
pertence_a(X, [_|C]) :-
  pertence_a(X, C).

trajetoria_impressa(t(F,G,T)) :-
  trajetoria_impressa_1(T).

trajetoria_impressa_1([ r(raiz,Raiz) ]) :-
  write('Estado Inicial: '),
  write(Raiz),
  write('. \n'),
  !.
trajetoria_impressa_1([ r(Operacao,Nodo) | Resto ]) :-
  trajetoria_impressa_1(Resto),
  write(Operacao),
  write(' a cidade: '),
  write(Nodo),
  write('. \n').

ligada_a(a,b). ligada_a(b,m).
ligada_a(m,f). ligada_a(f,q).
ligada_a(q,p). ligada_a(p,n).
ligada_a(p,s). ligada_a(b,c).
ligada_a(c,d). ligada_a(d,q).
ligada_a(d,n). ligada_a(d,g).
ligada_a(n,h). ligada_a(n,s).
ligada_a(h,g).

coord(a,  2, 4) :- !.  coord(b,  5, 6) :- !.
coord(c,  4, 2) :- !.  coord(d,  7, 4) :- !.
coord(f,  7, 8) :- !.  coord(g,  8, 2) :- !.
coord(h, 10, 1) :- !.  coord(m,  9, 6) :- !.
coord(n, 11, 3) :- !.  coord(p, 12, 6) :- !.
coord(q, 11, 7) :- !.  coord(s, 13, 2).


inserida(T, [], [T]).
inserida(t(F1,G1,Ramos1), [ t(F2,G2,Ramos2) | Resto ],
                          [ t(F1,G1,Ramos1), t(F2,G2,Ramos2) | Resto ]) :-
```

```
  F1 < F2,
  !.
inserida(t(F1,G1,Ramos1), [ t(F2,G2,Ramos2) | Resto ],
                          [ t(F2,G2,Ramos2) | Resto2 ]) :-
  inserida(t(F1,G1,Ramos1), Resto, Resto2).


inseridas_as_trajetorias([], Fila, Fila).
inseridas_as_trajetorias([T|C], Fila, Fila3) :-
  inserida(T,Fila, Fila2),
  inseridas_as_trajetorias(C, Fila2, Fila3).

trajetoria_expandida(t(F,  G,  [r(Ramo,     Nodo)              | Resto]),
                     t(F1, G1, [r(vai_para, Filho), r(Ramo,Nodo) | Resto])) :-
  ligada_a(Nodo, Filho),
  not(produz_ciclo(Filho, [r(Ramo,Nodo) | Resto])),
  funcao_h(Filho, H1),
  custo(Nodo, Filho, Custo),
  G1 is G  + Custo,
  F1 is G1 + H1.

produz_ciclo(Estado, Trajetoria) :-
  pertence_a(r(_,Estado), Trajetoria).

custo(Nodo, Filho, Custo) :-
  coord(Nodo,  XN, YN),
  coord(Filho, XF, YF),
  X is (XN-XF)*(XN-XF) + (YN-YF)*(YN-YF),
  prolog_eval(sqrt(X),Custo),
  !.

funcao_h(Ea, H) :-
  e_estado_final(Ef),
  coord(Ef, XEf, YEf),
  coord(Ea, XEa, YEa),
  X is (XEa-XEf)*(XEa-XEf) + (YEa-YEf)*(YEa-YEf),
  prolog_eval(sqrt(X),H),
  !.

adjacentes(X, Y, Z) :-
  findall(X, Y, Z),
  !.
adjacentes(_ , _ , [] ).

e_estado_final(s).

possui_estado_final(t(_,_,[r(_,Nodo)|_])) :-
  e_estado_final(Nodo).

% ---------------------------------------------------------------
%              BUSCA HEURISTICA PELO  A*
```

```
% ------------------------------------------------------------

solucao_computada_por_A_estrela([ Solucao | _ ], Solucao) :-
  possui_estado_final(Solucao),
  !.
solucao_computada_por_A_estrela([ Trajetoria | Fila ], Solucao) :-
  adjacentes(T2, trajetoria_expandida(Trajetoria, T2), Lista_de_Trajetorias_Expandidas),
  inseridas_as_trajetorias(Lista_de_Trajetorias_Expandidas, Fila, Fila_Expandida),
  solucao_computada_por_A_estrela(Fila_Expandida, Solucao).

% *** Exemplo de Estado Inicial: b

solucao_da_busca_por_A_estrela_a_partir_do(Estado_Inicial) :-
  funcao_h(Estado_Inicial, H),
  solucao_computada_por_A_estrela([ t(H,0,[r(raiz,Estado_Inicial)]) ], Trajetoria),
  trajetoria_impressa(Trajetoria).
```

Para ilustrar a aplicação do predicado `solucao_da_busca_por_A_estrela_a_partir_do` basta compilar seu código e executar a seguinte consulta:

```
?- solucao_da_busca_por_A_estrela_a_partir_do(b).
Estado Inicial: b.
vai_para a cidade: c.
vai_para a cidade: d.
vai_para a cidade: n.
vai_para a cidade: s.
yes
```

Note os seguintes pontos:

- o algoritmo A^* é um método de busca admissível quando a sua função heurística for admissível;
- o algoritmo de busca em $Profundidade$ não é um método admissível;
- é importante não confundir "encontrar a trajetória mais curta" com "encontrar mais rapidamente a trajetória" (a busca em *Amplitude* encontra a trajetória mais curta, dadas as condições acima, mas isso não quer dizer que ela a encontre mais rapidamente);
- a busca de *Custo Uniforme* pode ser vista como uma variação do método A^* quando $h(E_a) = 0$, ou seja, se $f(E_a) = g(E_a)$, então o método investigará primeiro os nodos mais próximos do E_i (aproximadamente como a busca em *Amplitude* se comporta para os casos em que a função de custo é monotônica e ainda obedece à desigualdade triangular);
- o algoritmo de busca de *Custo Uniforme* é um método admissível apenas quando a função de custo real é monotonicamente crescente;
- nos casos onde a estimativa $h(E_a)$ é bem menor que o custo real, o A^* tenderá a gastar bastante memória com a manutenção de sua lista de prioridades.

2.2 Algoritmo IDA^*

O nome IDA^* significa *Iterative Deepening* A^*. Para descrever melhor os seus conceitos básicos, além de conhecer o A^*, é preciso abordar os seguintes elementos:

- busca de altura fixa;
- aumento iterativo da altura de busca;
- contornos de custo do espaço de busca.

A busca de altura fixa é capaz de evitar até mesmo as deficiências da busca em profundidade (que em geral acha soluções muito longas). Note os aspectos abaixo.

- O algoritmo precisa impor um corte na trajetória que atingir a altura máxima sem incluir o E_f para dar continuidade à busca por meio de outra trajetória que ainda possa incluí-lo (E_f) dentro da mesma altura máxima.
- Por exemplo, em um mapa com 20 cidades, se houver solução para um dado problema, a trajetória terá um tamanho máximo de 19 ramos.
- A busca de altura fixa é completa (dependendo de sua altura máxima) mas não é minimalista.

O aumento iterativo da busca, combinado com a busca de altura fixa inclui as seguintes características:

- O algoritmo inicia com altura zero (uma questão formal que traz resultados interessantes quando o limite passar a ser o custo da trajetória e não o seu tamanho).
- Cada ciclo que termina sem incluir o E_f é seguido de outro com exatamente 1 (um) ramo a mais no limite de tamanho da trajetória.
- A busca em profundidade de altura fixa com aumento iterativo da altura é minimalista e completa.

O conceito de contorno de custo do espaço de busca é dado por um valor-limite para custo. Isso resulta na possibilidade de se manter a expansão de trajetórias cujos valores de f estão comprendidos dentro do valor-limite. Note agora os seguintes itens.

- Cada ciclo do IDA* é realizado por meio de uma Busca em Profundidade dentro do contorno de custo, com aumento iterativo do valor-limite de custo (e não da altura máxima).
- O valor-limite do primeiro contorno de custo é $f(E_i)$.
- Não havendo solução dentro do contorno de custo atual, um novo valor-limite é ajustado pelo menor valor de f do ciclo anterior.
- Vários contornos de custo podem ser ajustados durante o processo para que um novo ciclo de busca possa se re-iniciar, sempre a partir do E_i, e tente incluir o E_f.

Exercício: Implemente a solução para o problema dos Ladrilhos Deslizantes ($8 - Puzzle$) por meio do algoritmo IDA^*.

2.3 Algoritmo SMA^*

O nome SMA^* significa *Simplified Memory-Bouded* A^*. Para descrever melhor os seus conceitos básicos, além de conhecer o A^*, é preciso abordar os seguintes elementos:

- busca de tamanho fixo de memória (a fila de trajetórias sofre controle por meio de um valor de tamanho máximo que representa a soma dos tamanhos das trajetórias registradas no momento);
- eliminação da pior trajetória para que uma nova trajetória seja inserida;
- dominuição da repetição de expansão de estados por meio do registro do custo de uma trajetória eliminada.

2.4 Algoritmo de Busca Gulosa (Greedy - Best-first)

O de busca pela Melhor Escolha é assim conhecido por ser uma variação da busca em profundidade re-orientada apenas pelo menor valor da função heurística h aplicada aos descendentes do E_a (estado atual). Em outras palavras, o algoritmo é guiado apenas pela estimativa de menor distância do E_a ao E_f. Suas características são:

- pode ter bom desempenho se a função heurística for boa;
- mas sofre dos mesmos problemas da busca em profundidade (pode seguir caminhos infrutíferos, inclusive infinitos);
- e não oferece nem completude nem minimização da solução.

2.5 Busca por Satisfação de Restrições

O de busca por Satisfação de Restrições é assim conhecido por ser mais uma variação da busca em profundidade, porém, re-orientada tipicamente por parâmetros de natureza meta-heuríistica. Suas características são:

- meta-heuríistica é gerada com base na quantidade de restrições e de valores representados nos estados utilizados para representar o domínio do problema;
- o E_f (estado final) geralmente não é conhecido em um enunciado típico da classe dos domínios nos quais a solução de problemas pode ser encontrada por este algoritmo;
- a solução de problema pode ser absoluta (o teste de E_f é definido precisamente) ou relativa (o teste de E_f se resume à ultrapassagem de um limiar).

As principais meta-heuísticas utilizadas por este algoritmo são duas.

- Variável mais restrita:
- Variável mais restritiva:
- Valor menos restritivo: