Introdução à Inteligência Artificial

Busca em grafos

Prof. Dr. Alneu de Andrade Lopes

LABIC - ICMC - USP São Carlos

Conteúdo

- Parte I
  - Busca exaustiva
- Parte II
  - Busca informada

Problema de busca

- Definido por meio de
  - Um espaço de busca, o qual é um grafo com um ou mais vértices iniciais e um ou mais vértices metas.
  - A solução é um caminho, começando em um vértice inicial e terminando em um vértice meta.
  - Função de custo (um número p/ cada aresta)

Técnicas exaustivas de busca

- Completude: a solução sempre será encontrada?
- “Optimalidade”: o caminho mais curto será encontrado antes dos caminhos mais longos?
- Eficiência: quais são os requisitos de memória e tempo de execução?

Busca Baseada em Agenda

1. pegue o próximo nó na agenda;
2. Se é um nó meta, pare;
3. Senão
   1. Obtenha seus filhos;
   2. Coloque-os na agenda;
   3. Vá para o passo 1.

Agenda: lista de nós a serem avaliados (backtracking)

Agenda-based search

```
% search(Agenda,Goal) <- Goal é um nó meta, e um
%                        nó filho de um dos nós em
%                        Agenda
search(Agenda,Goal):-
   next(Agenda,Goal,Rest),
   goal(Goal).
search(Agenda,Goal):-
   next(Agenda,Current,Rest),
   children(Current,Children),
   add(Children,Rest,NewAgenda),
   search(NewAgenda,Goal).
```

Depth-first vs. breadth-first search

```
search_df([Goal|Rest],Goal):-
   goal(Goal).
search_df([Current|Rest],Goal):-
   children(Current,Children),
   append(Children,Rest,NewAgenda),
   search_df(NewAgenda,Goal).

search_bf([Goal|Rest],Goal):-
   goal(Goal).
search_bf([Current|Rest],Goal):-
   children(Current,Children),
   append(Rest,Children,NewAgenda),
   search_bf(NewAgenda,Goal).

children(Node,Children):-
   findall(C,arc(Node,C),Children).
```

Depth-first vs. breadth-first search

- Busca *Breadth-first*
  - agenda = fila (first-in first-out)
  - completa: guarante encontrar todas as soluções
  - Primeira solução encontrada no menor caminho
  - Requisito de memória: $O(B^n)$
- Busca *Depth-first*
  - agenda = pilha (last-in first-out)
  - incompleta: pode ficar preso em um ramo infinito
  - Nenhuma propriedade de menor caminho
  - Requisito de memória: $O(B \times n)$
    n= n.médio de filhos
    B=profundidade

Detecção de loop

```
% depth-first search with loop detection
search_df_loop([Goal|Rest],Visited,Goal):-
  goal(Goal).
search_df_loop([Current|Rest],Visited,Goal):-
  children(Current,Children),
  add_df(Children,Rest,Visited,NewAgenda),
  search_df_loop(NewAgenda,[Current|Visited],Goal).

add_df([],Agenda,Visited,Agenda).
add_df([Child|Rest],OldAgenda,Visited,[Child|NewAgenda]):-
  not element(Child,OldAgenda),
  not element(Child,Visited),
  add_df(Rest,OldAgenda,Visited,NewAgenda).
add_df([Child|Rest],OldAgenda,Visited,NewAgenda):-
  element(Child,OldAgenda),
  add_df(Rest,OldAgenda,Visited,NewAgenda).
add_df([Child|Rest],OldAgenda,Visited,NewAgenda):-
  element(Child,Visited),
  add_df(Rest,OldAgenda,Visited,NewAgenda).
```

Busca Best-first (informada)

```
search_bstf([Goal|Rest],Goal):-
  goal(Goal).
search_bstf([Current|Rest],Goal):-
  children(Current,Children),
  add_bstf(Children,Rest,NewAgenda),
  search_bstf(NewAgenda,Goal).

% add_bstf(A,B,C) <- C contains the elements of A and B
%                    (B and C sorted according to eval/2)
add_bstf([],Agenda,Agenda).
add_bstf([Child|Children],OldAgenda,NewAgenda):-
  add_one(Child,OldAgenda,TmpAgenda),
  add_bstf(Children,TmpAgenda,NewAgenda).

% add_one(S,A,B) <- B is A with S inserted acc. to eval/2
add_one(Child,OldAgenda,NewAgenda):-
  eval(Child,Value),
  add_one(Value,Child,OldAgenda,NewAgenda). implementar
```

A algorithm

- Um ***algoritmo A*** é um algoritmo de busca best-first que objetiva minimizar o custo total do caminho do no início ao no meta.
  - f(n) = g(n) + h(n)

Estimativa do custo total ao longo do caminho através de n

Custo real para alcançar n

Estimativa para alcançar a meta a partir de n

- O algoritmo A é completo: desde que g(n) previne que a busca fique presa em um caminho infinito (comporta-se como na busca breadth-first);
- Aliás, bf search é um caso especial do algoritmo A, com h(n)=0 para todos os nós.

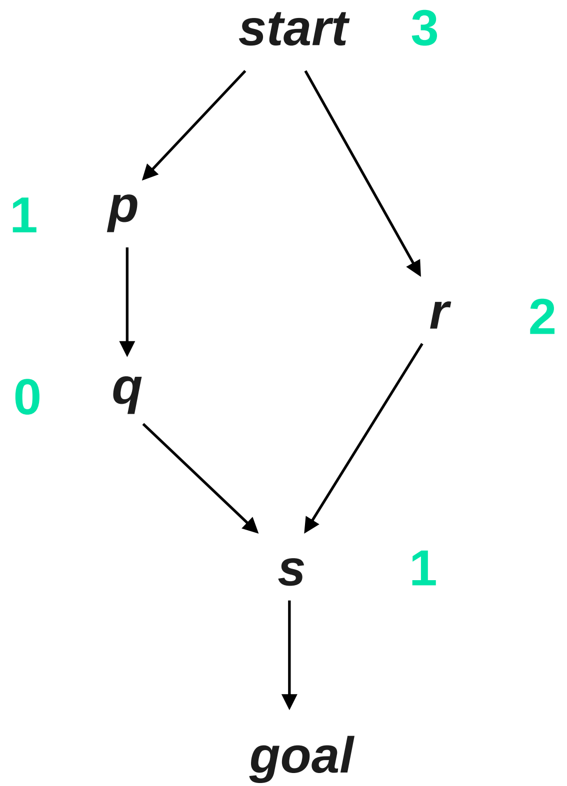
- É ótimo: sempre leva a solução de menor caminho (se arcos têm os mesmos custos);
- Um algoritmo A com um heurística otimista é chamado A*.
  - Otimista: imagina que o custo de solução é menor do que ele é
  - Obs. Melhor heurística não quer dizer mais otimista. Ao contrário uma boa heurística é tão pessimista quanto possível sem se tornar não admissível.
  - Heurística admissível h(n) nunca superestime o custo para alcançar o objetivo. Ex: distância em linha reta

- Em geral,
  - Se h1(n)>=h2(n) para qualquer nó n, então a heurística h1 é pelo menos tão *informada* quanto h2. (h1 é mais restritiva: espaço de busca menor)

exemplo

- Menor caminho:
  start-r-s-goal
- Não é encontrado imediatamente.
  - Custo aresta = 1
  - h(p) < h(r)
  - Depois de q ser investigado r é colocado na agenda, com f(r)=1+2=3

    f(p)=1+1
    f(s)=3+1=4



Busca informada não exaustiva

- A busca exaustiva (anteriores) no pior caso examinam todos os nós;
- Isto por porque todos os filhos de um nó são adicionados a agenda.
- Adicionando só os nós "mais promissores" tem-se um busca não exaustiva (não necessariamente completa).

```
search_beam(Agenda,Goal):-
  search_beam(1,Agenda,[],Goal).

search_beam(D,[],NextLayer,Goal):-
  D1 is D+1,
  search_beam(D1,NextLayer,[],Goal).
search_beam(D,[Goal|Rest],NextLayer,Goal):-
  goal(Goal).
search_beam(D,[Current|Rest],NextLayer,Goal):-
  children(Current,Children),
  add_beam(D,Children,NextLayer,NewNextLayer),
  search_beam(D,Rest,NewNextLayer,Goal).
```

- Here, the number of children to be added to the beam is made dependent on the depth **D** of the node
  - in order to keep depth as a 'global' variable, search is layer-by-layer

Beam search

Hill-climbing

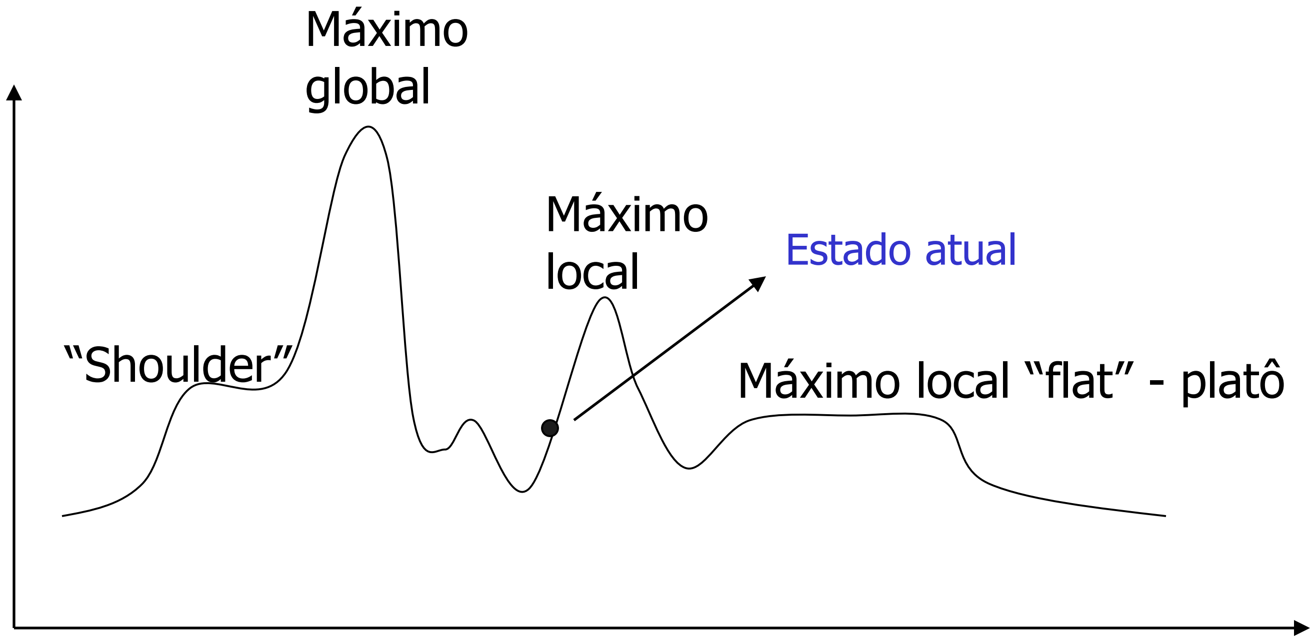
```
search_hc(Goal,Goal):-
 goal(Goal).
search_hc(Current,Goal):-
 children(Current,Children),
 select_best(Children,Best),
 search_hc(Best,Goal).
```

```
% hill_climbing as a variant of best-first search
search_hc([Goal|_],Goal):-
   goal(Goal).
search_hc([Current|_],Goal):-
   children(Current,Children),
   add_bstf(Children,[],NewAgenda),
   search_hc(NewAgenda,Goal).
```

- Limita o tamanho da Agenda em 1.
- Chamada greedy search ou busca gulosa, uma vez que não usa backtracking, $(f(n)=h(n))$.
- Pode atingir um máximo local, e não alcançar o máximo global.

Simulated annealing

- tenta evitar ficar preso em um máximo local



Referências

- Capítulo 4 (Russel & Norvig)
- Capítulo 5 e 6 (Peter Flach)

Complemento

Busca com Backtracking

```
% depth-first search by means of backtracking
search_bt(Goal,Goal):-
 goal(Goal).
search_bt(Current,Goal):-
 arc(Current,Child),
 search_bt(Child,Goal).

% backtracking depth-first search with depth
 bound
search_d(D,Goal,Goal):-
 goal(Goal).
search_d(D,Current,Goal):-
 D>0, D1 is D-1,
 arc(Current,Child),
 search_d(D1,Child,Goal).
```

Iterative deepening

```
search_id(First,Goal):-
   search_id(1,First,Goal).      % start with depth
   1

search_id(D,Current,Goal):-
   search_d(D,Current,Goal).
search_id(D,Current,Goal):-
   D1 is D+1,                    % increase depth
   search_id(D1,Current,Goal).
```

- Combina as vantagens da busca em largura (completa, menor caminho) com as da busca em profundidade (eficiência de memória)

Agenda-based SLD-prover

```
prove(true):-!.
prove((A,B)):-!,
   clause(A,C),
   conj_append(C,B,D),
   prove(D).
prove(A):-
   clause(A,B),
   prove(B).
```

```
prove_df_a(Goal):-
  prove_df_a([Goal]).

prove_df_a([true|Agenda]).
prove_df_a([(A,B)|Agenda]):-!,
  findall(D,(clause(A,C),conj_append(C,B,D)),Childr
  en),
  append(Children,Agenda,NewAgenda),
  prove_df_a(NewAgenda).
prove_df_a([A|Agenda]):-
  findall(B,clause(A,B),Children),
  append(Children,Agenda,NewAgenda),
  prove_df_a(NewAgenda).
```

Refutation prover for clausal logic

```
refute((false:-true)).
refute((A,C)):-
   cl(Cl),
   resolve(A,Cl,R),
   refute(R).
```

```
% refute_bf(Clause) <- Clause is
                       refuted by clauses
%                      defined by cl/1
%                      (breadth-first search strategy)
refute_bf_a(Clause):-
   refute_bf_a([a(Clause,Clause)],Clause).

refute_bf_a([a((false:-true),Clause)|Rest],Clause).
refute_bf_a([a(A,C)|Rest],Clause):-
   findall(a(R,C),(cl(Cl),resolve(A,Cl,R)),Children),
   append(Rest,Children,NewAgenda),     % breadth-first
   refute_bf_a(NewAgenda,Clause).
```

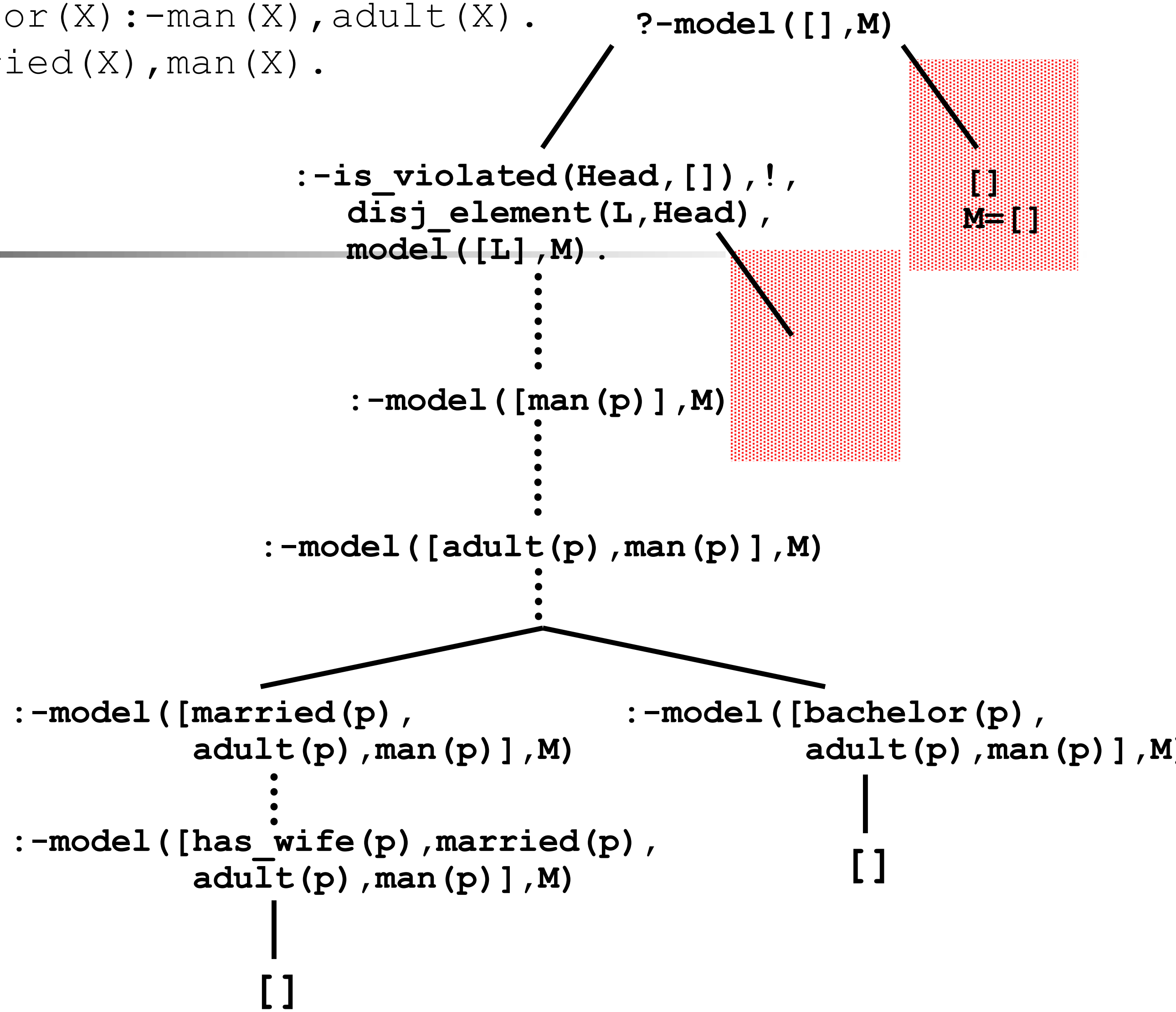
Forward chaining

```
% model(M) <- M is a model of the clauses defined by
  cl/1
model(M):-
  model([],M).

model(M0,M):-
  is_violated(Head,M0),!,     % instance of violated
  clause
  disj_element(L,Head),       % L: ground literal from head
  model([L|M0],M).            % add L to the model
model(M,M).                   % no more violated clauses

is_violated(H,M):-
  cl((H:-B)),
  satisfied_body(B,M),        % grounds the variables
  not satisfied_head(H,M).
```

```
married(X);bachelor(X):-man(X),adult(X).
has_wife(X):-married(X),man(X).
man(paul).
adult(paul).
```



Forward chaining: example

Forward chaining with depth-bound

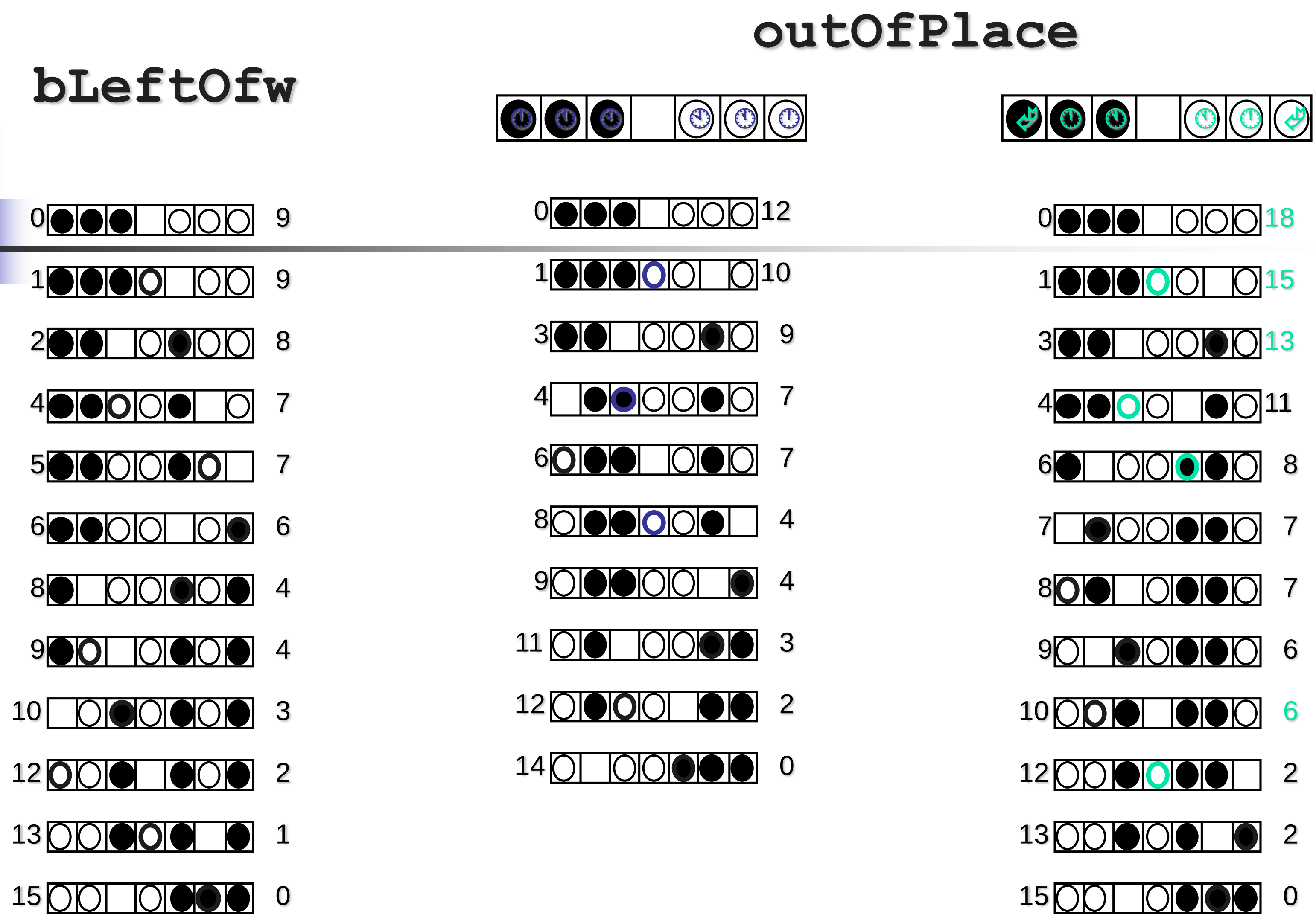
```
% model_d(D,M) <- M is a submodel of the clauses
%                 defined by cl/1
model_d(D,M):-
  model_d(D,[],M).

model_d(0,M,M).
model_d(D,M0,M):-
  D>0,D1 is D-1,
  findall(H,is_violated(H,M0),Heads),
  satisfy_clauses(Heads,M0,M1),
  model_d(D1,M1,M).

satisfy_clauses([],M,M).
satisfy_clauses([H|Hs],M0,M):-
  disj_element(L,H),
  satisfy_clauses(Hs,[L|M0],M).
```

Solving a puzzle

```
% tiles_a(A,M,V0,V) <- goal position can be reached from
%                       one of the positions on A with last
%                       move M (best-first strategy)
tiles_a([v(V,LastMove)|Rest],LastMove,Visited,Visited):-
 goal(LastMove).
tiles_a([v(V,LastMove)|Rest],Goal,Visited0,Visited):-
 show_move(LastMove,V),
 setof0(v(Value,NextMove),
        ( move(LastMove,NextMove),
          eval(NextMove,Value) ),
        Children),                   % Children sorted on Value
 merge(Children,Rest,NewAgenda),     % best-first
 tiles_a(NewAgenda,Goal,[LastMove|Visited0],Visited).
```



Comparing heuristics

Global and local optimism

- A heuristic is (globally) ***optimistic*** or ***admissible*** if the estimated cost of **reaching a goal** is always less than the actual cost.
  - $h(n) \leq h^*(n)$

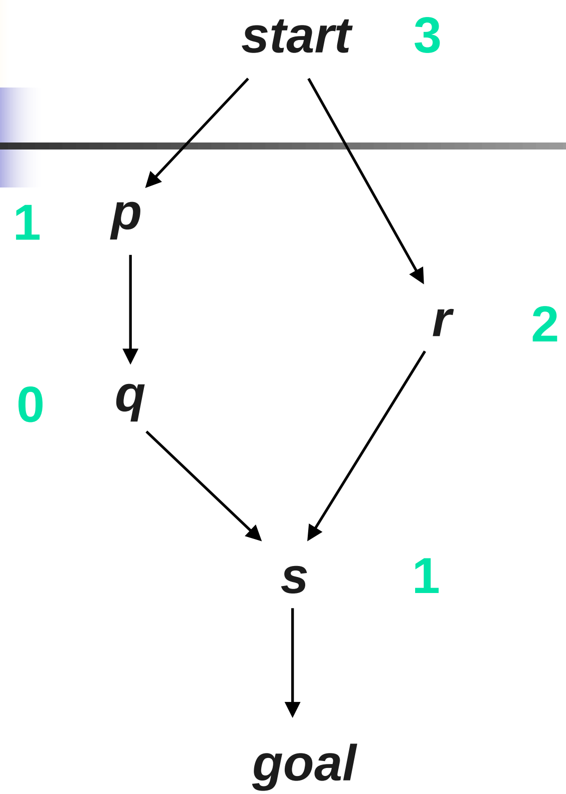
estimate of cost to reach goal from n

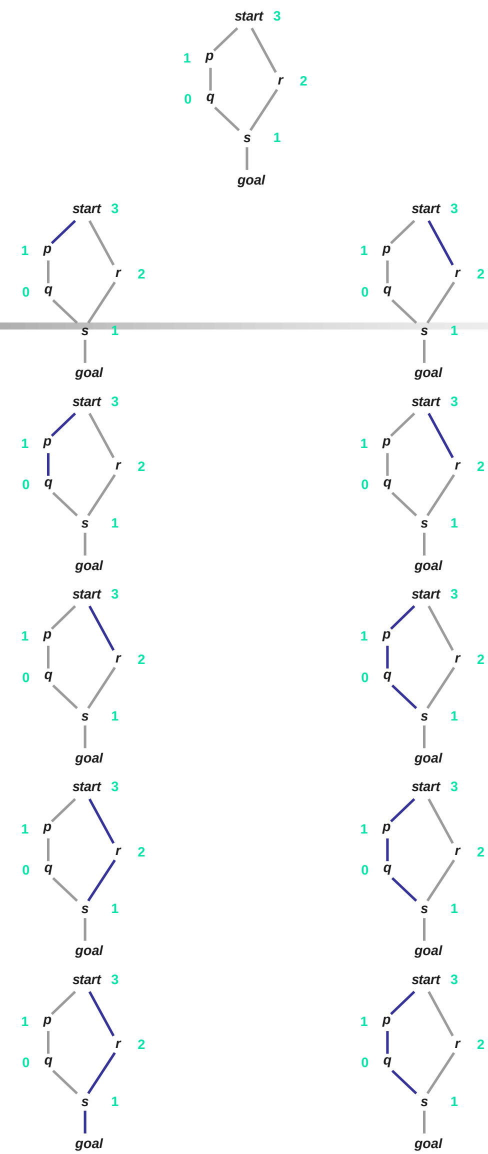
actual (unknown) cost to reach goal from n

- A heuristic is ***monotonic*** (locally optimistic) if the estimated cost of **reaching any node** is always less than the actual cost.
  - $h(n_1) - h(n_2) \leq h^*(n_1) - h^*(n_2)$

p.126

Non-monotonic heuristic





[start-3]

[p-2,r-3]

[q-2,r-3]

[r-3,s-4]

[s-3,s-4]

[goal-3,s-4]

- Embora uma busca admissível (globally optimistic) leve a uma solução ótima, não é necessariamente verdade que cada nó seja alcançado ao longo do caminho ótimo.