Estratégias de Busca em Espaços de Estados

- Busca não informada
  - Em profundidade e variações
  - Em largura
- Busca informada
  - Gulosa
  - A*
  - Hill-climbing

Inteligência Artificial

Estratégias de Busca em Espaços de Estados

- Busca não informada
  - Em profundidade e variações
  - Em largura
- Busca informada
  - Gulosa
  - A*
  - Hill-climbing

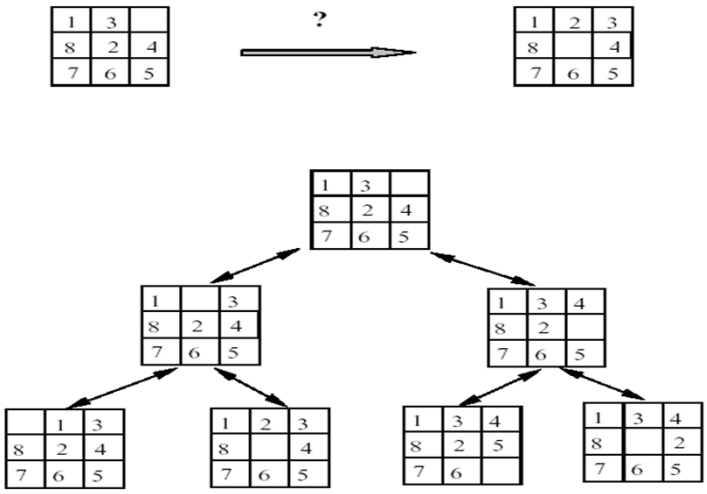
Inteligência Artificial

Busca em Espaço de Estados

- Um grafo pode ser usado para representar um **espaço de estados** onde:
  - Os nós correspondem a situações de um problema
  - As arestas correspondem a movimentos permitidos ou ações ou passos da solução
  - Um dado problema é solucionado encontrando-se um caminho no grafo
- Um **problema** é definido por
  - Um espaço de estados (um grafo)
  - Um estado (nó) inicial
  - Uma condição de término ou critério de parada; estados (nós) terminais são aqueles que satisfazem a condição de término
- Se não houver custos, há interesse em soluções de caminho mínimo
- No caso em que custos são adicionados aos movimentos, normalmente há interesse em soluções de custo mínimo
  - O custo de uma solução é o custo das arestas ao longo do caminho da solução

3

Exemplo: Quebra-Cabeça-8



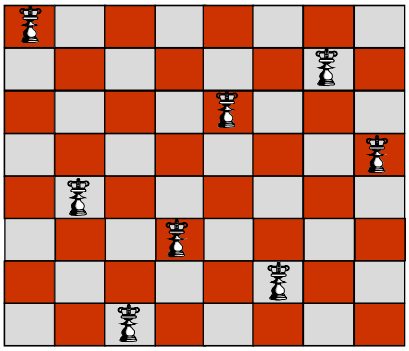
4

Exemplo: Quebra-Cabeça-8

- Estados: posições inteiras dos quadrados (ignorar posições intermediárias)
- Operadores: mover branco para esquerda, direita, acima, embaixo
- Estado Final: = estado fornecido (único)
- Custo do caminho: 1 por movimento

5

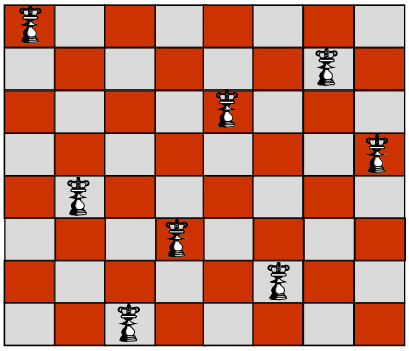
Exemplo: 8 Rainhas



- Estados?
- Operadores?
- Estado Final?
- Custo do caminho?

6

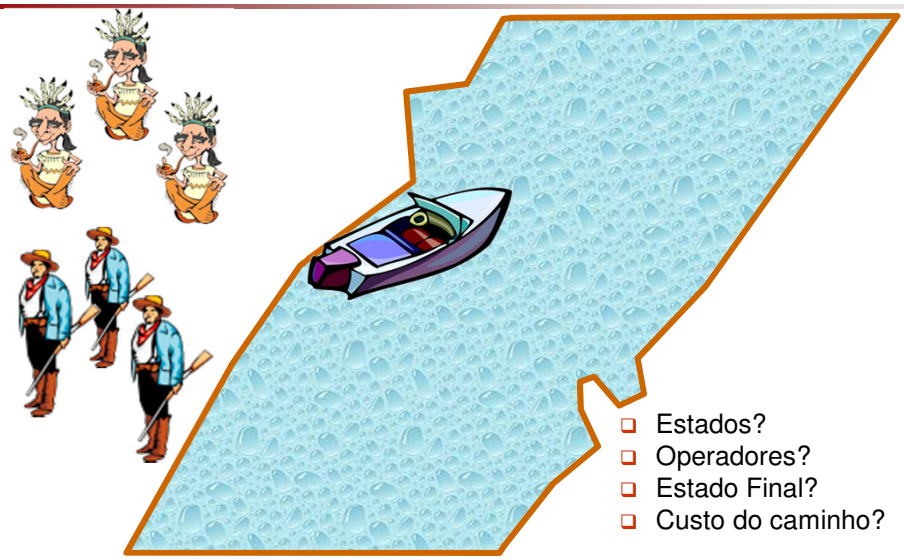
Exemplo: 8 Rainhas



- Estado: qualquer arranjo de 0 a 8 rainhas no tabuleiro
- Operador: adicionar uma rainha a qualquer quadrado
- Estado Final: 8 rainhas no tabuleiro, sem ataque (múltiplos estados)
- Custo do caminho: zero (apenas o estado final é interessante)

7

Exemplo: Missionários e Canibais



- Estados?
- Operadores?
- Estado Final?
- Custo do caminho?

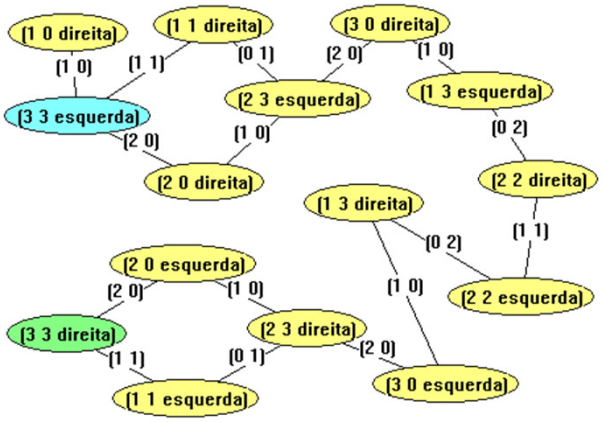
8

Exemplo: Missionários e Canibais

- Estados
  - um estado é uma sequência ordenada de três números representando o número de canibais, missionários e bote na margem do rio na qual eles iniciam
  - Assim, o estado inicial é [3,3,esquerda]
- Operadores
  - a partir de cada estado os operadores possíveis são: tomar um canibal, um missionário, um missionário e um canibal, dois missionários ou dois canibais
  - Há no máximo 5 operadores, embora alguns estados tenham menos operadores uma vez que se deve evitar estados inválidos
- Estado Final: [3,3,direita] (único)
- Custo do caminho: número de travessias

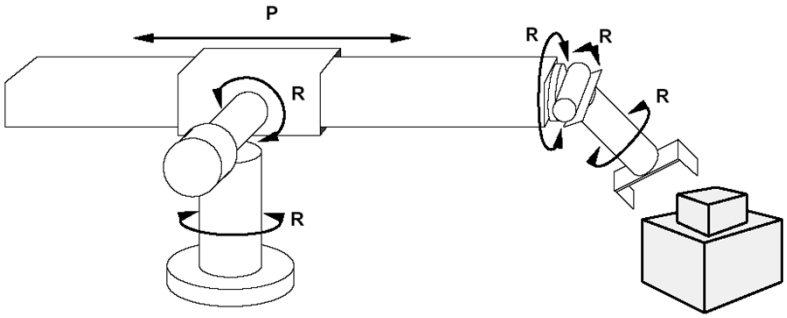
9

Exemplo: Missionários e Canibais



10

Exemplo: Montagem com Robô

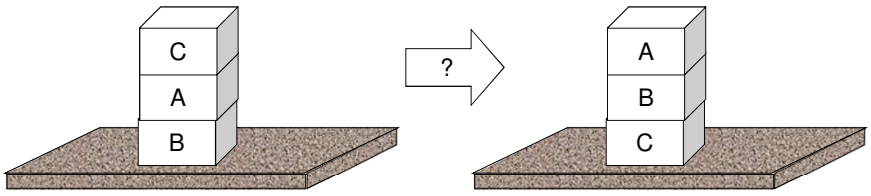


- Estados:
  - coordenadas de valor real de ângulos das junções do robô
  - partes do objeto a ser montado
- Operadores: movimentos contínuos das junções do robô
- Estado Final: montagem completa (único)
- Custo do caminho: tempo para montagem

11

Exemplo: Pilha de Blocos

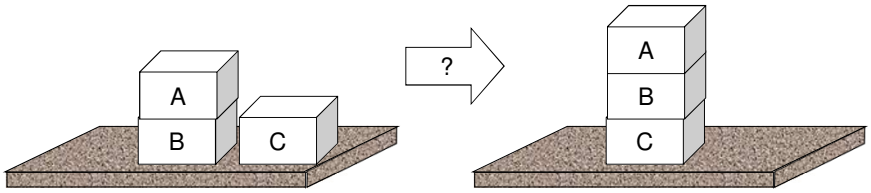
- Considere o problema de encontrar um plano (estratégia) para rearranjar uma pilha de blocos como na figura
  - Somente é permitido um movimento por vez
  - Um bloco somente pode ser movido se não há nada em seu topo
  - Um bloco pode ser colocado na mesa ou acima de outro bloco



12

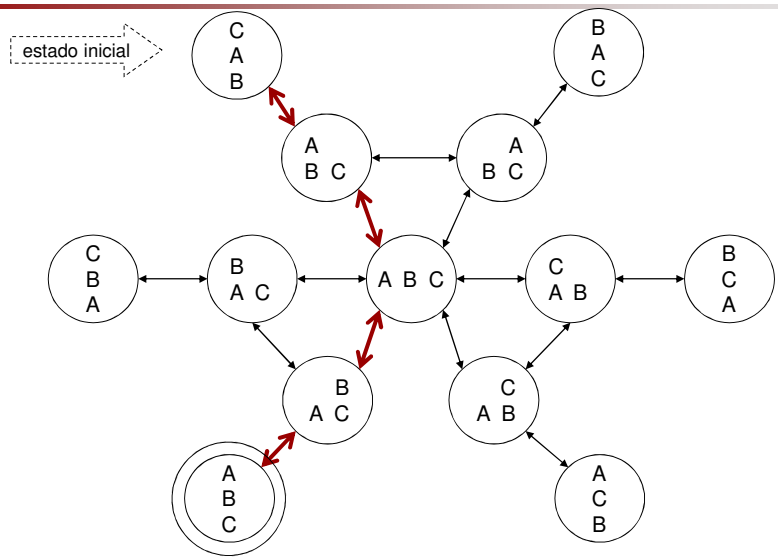
Exemplo: Pilha de Blocos

- Na situação inicial do problema, há apenas um movimento possível: colocar bloco C na mesa
- Depois que C foi colocado na mesa, há três alternativas
  - Colocar A na mesa ou
  - Colocar A acima de C ou
  - Colocar C acima de A (movimento que não deve ser considerado pois retorna a uma situação anterior do problema)



13

Exemplo: Pilha de Blocos



14

Espaço de Estados

- Árvore ou Grafo?

Neste caso, um **grafo pode ser transformado em uma árvore**, com um mesmo nó ocorrendo em diferentes subárvores, indicando que se pode passar por um estado por diferentes caminhos a partir do nó inicial (raiz).

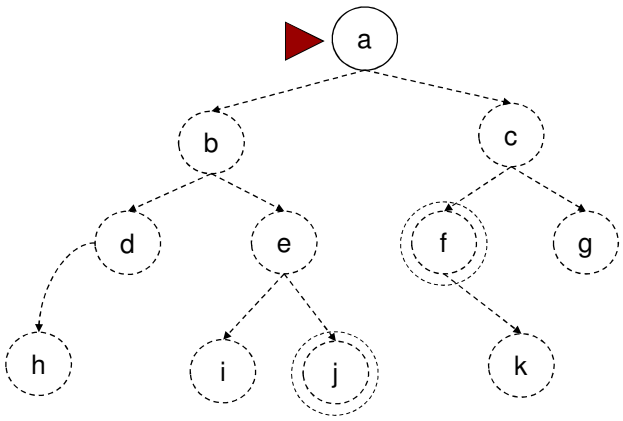
15

Busca em Espaço de Estados

- Estratégias Básicas de Busca (Uniforme, Exaustiva ou Cega)
  - não utiliza informações sobre o problema para guiar a busca
  - estratégia de busca exaustiva aplicada até uma solução ser encontrada (ou falhar)
    - Profundidade (Depth-first)
    - Profundidade limitada (Depth-first Limited)
    - Largura (Breadth-first)
    - Bidirecional
- Estratégias Heurísticas de Busca (Busca Informada)
  - utiliza informações específicas do domínio para ajudar na decisão
    - Gulosa
    - A*

16

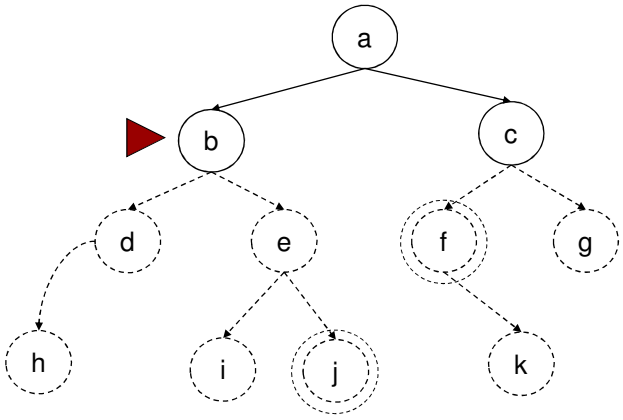
Percurso em Profundidade



Inserir na frente, remover da frente: a

17

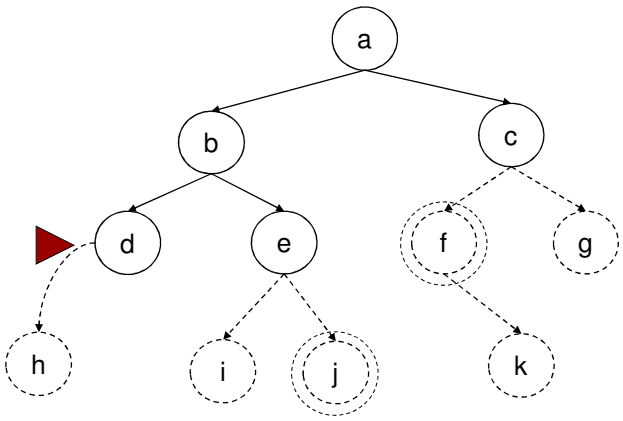
Percurso em Profundidade



Inserir na frente, remover da frente: b, c

18

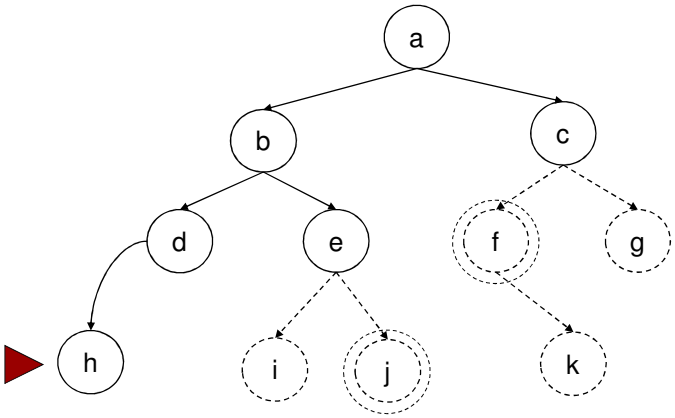
Percurso em Profundidade



Inserir na frente, remover da frente: d, e, c

19

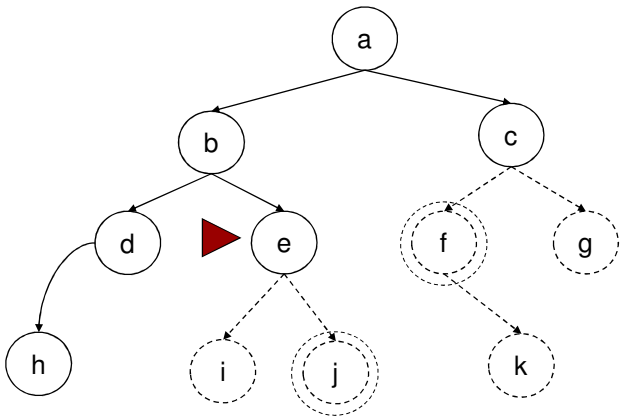
Percurso em Profundidade



Inserir na frente, remover da frente: h, e, c

20

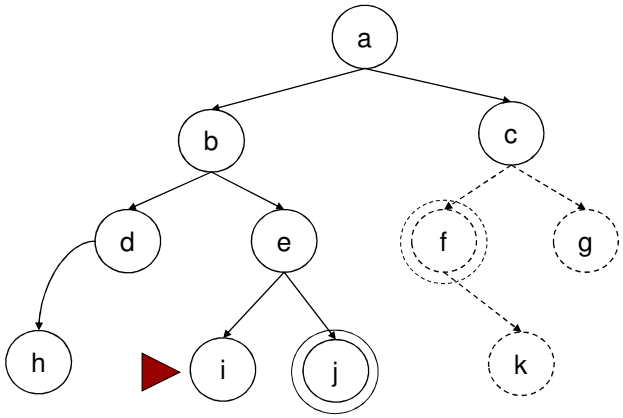
Percurso em Profundidade



Inserir na frente, remover da frente: e, c

21

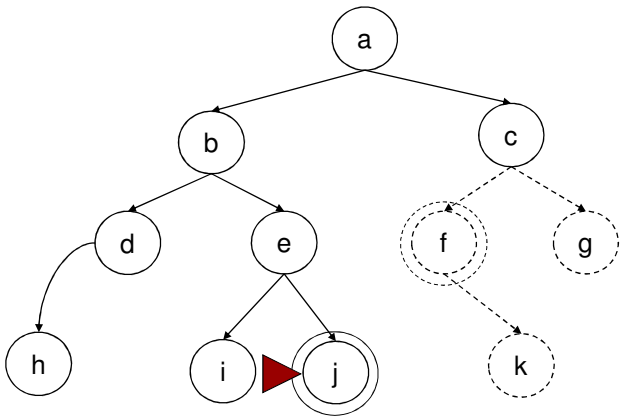
Percurso em Profundidade



Inserir na frente, remover da frente: i, j, c

22

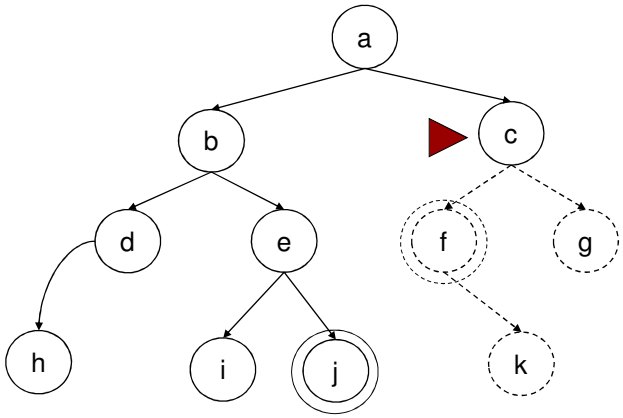
Percurso em Profundidade



Inserir na frente, remover da frente: j, c

23

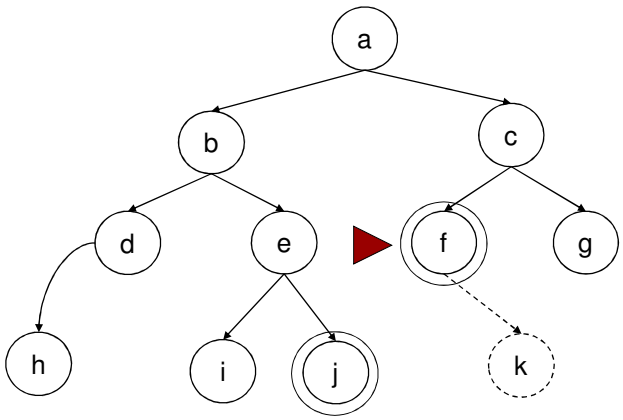
Percurso em Profundidade



Inserir na frente, remover da frente: c

24

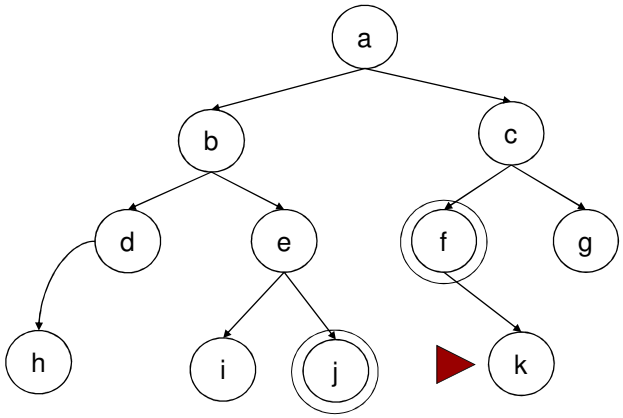
Percurso em Profundidade



Inserir na frente, remover da frente: f, g

25

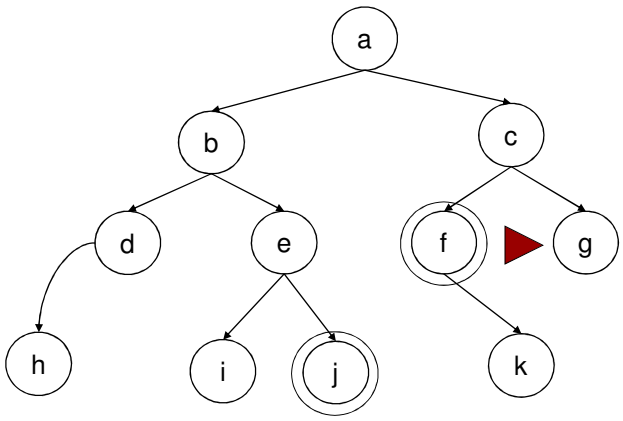
Percurso em Profundidade



Inserir na frente, remover da frente: k, g

26

Percurso em Profundidade



Inserir na frente, remover da frente: g

27

Busca em Espaço de Estados

- O programa de busca será implementado como a relação:

resolva(Início,Solucao)

onde **Início** é o nó inicial no espaço de estados e
Solucao é um caminho entre Início e qualquer nó final.

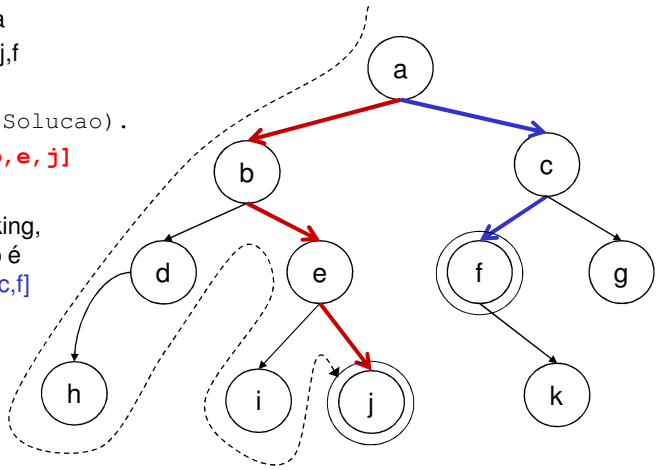
28

Busca em Profundidade

- Estado inicial: a
- Estados finais: j,f

```
?- resolva(a, Solucao).
Solucao = [a,b,e,j]
```

- Após backtracking, a outra solução é encontrada: [a,c,f]



29

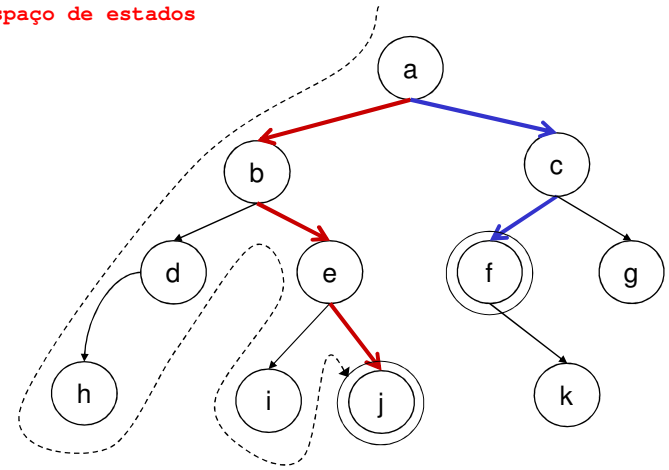
Busca em Espaço de Estados

- Vamos representar um espaço de estados pelas relações
  - **s(X,Y)** que é verdadeira se há um movimento permitido no espaço de estados do nó X para o nó Y; neste caso, Y é um **sucessor** de X
  - **final(X)** que é verdadeira se X é um estado final
- Se houver custos envolvidos, um terceiro argumento será adicionado: o custo do movimento
  - **s(X,Y,Custo)**
- A relação **s** pode ser representada explicitamente no programa por um conjunto de fatos
- Entretanto, para espaços de estado complexos, a relação **s** é usualmente definida implicitamente através de regras que permitam calcular o sucessor de um dado nó

30

Exercício: parte 1

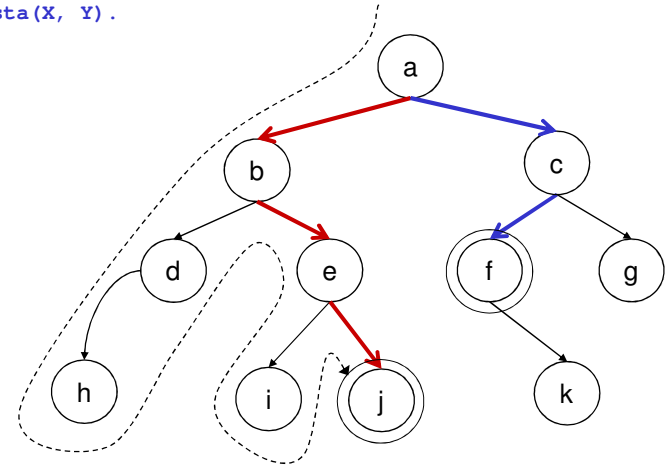
Represente o espaço de estados
ao lado!



31

Exercício: parte 1

```
s(X, Y) :- aresta(X, Y).
aresta(a, b).
aresta(a, c).
aresta(b, d).
aresta(b, e).
aresta(c, f).
aresta(c, g).
aresta(d, h).
aresta(e, i).
aresta(e, j).
aresta(f, k).
final(j).
final(f).
```



32

Exercício: parte 2

- Implemente em Prolog a busca em profundidade, de forma que, a partir de um nó inicial, retorne o caminho da solução.

33

Busca em Profundidade

- Para encontrar Solucao a partir de um nó N (até um nó final):
  - Se N é um nó final, então Solucao = [N]
  - Se N tem um sucessor N1, tal que há um caminho N1 a um nó final, então Solucao = [N | Caminho_de_N1_a_no_final] (ordem inversa)
- Em Prolog:

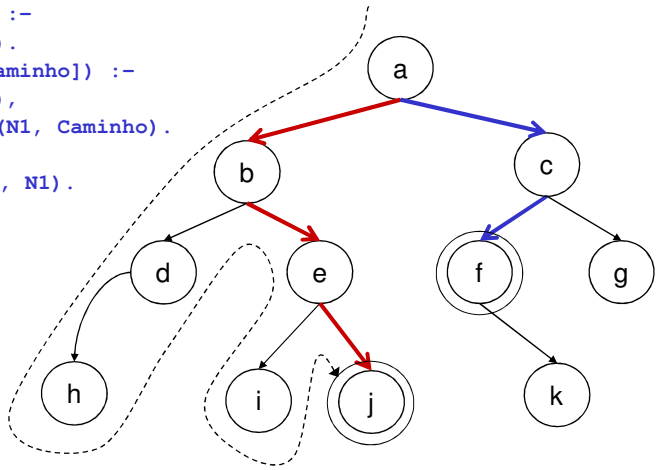
```
resolva(N,[N]) :- final(N).     % alcançou a meta
resolva(N,[N|Caminho]) :-
    s(N,N1),                    % faça um movimento válido
    resolva(N1,Caminho).        % recursividade
```

- A busca em profundidade é a mais natural quando se usa a recursão

34

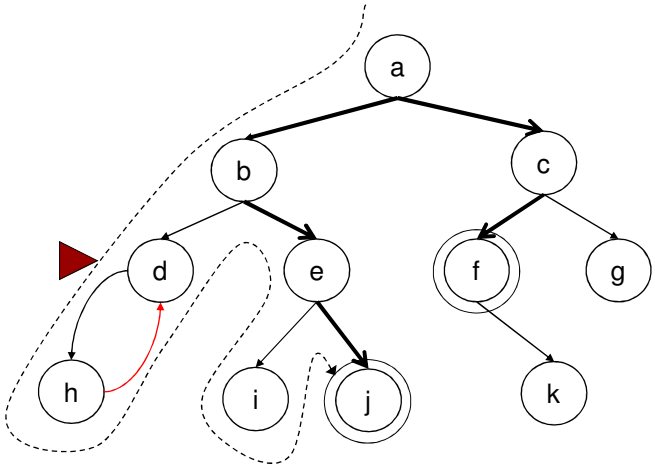
Busca em Profundidade

```
resolva(N, [N]) :-
        final(N).
resolva(N, [N|Caminho]) :-
        s(N, N1),
        resolva(N1, Caminho).
s(N, N1) :-
       aresta(N, N1).
aresta(a, b).
aresta(a, c).
aresta(b, d).
aresta(b, e).
aresta(c, f).
aresta(c, g).
aresta(d, h).
aresta(e, i).
aresta(e, j).
aresta(f, k).
final(j).
final(f).
```



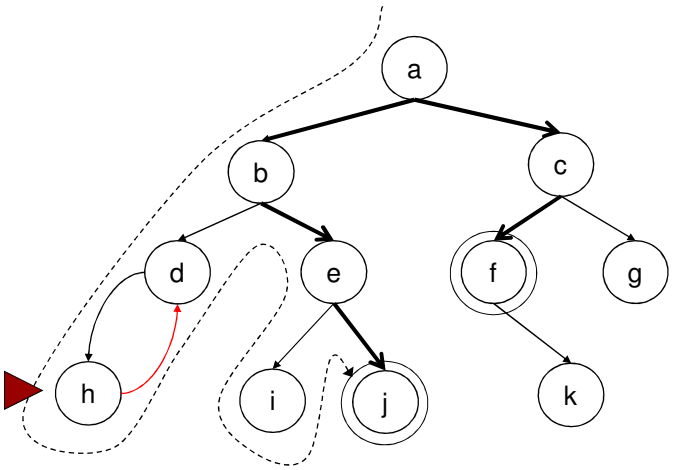
35

Busca em Profundidade: Ciclos



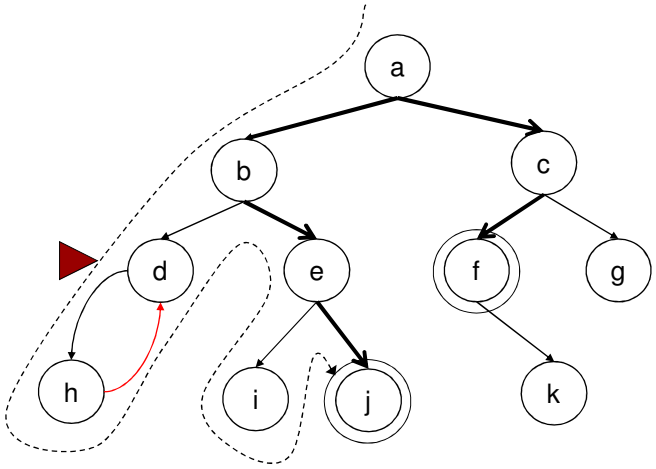
36

Busca em Profundidade: Ciclos



37

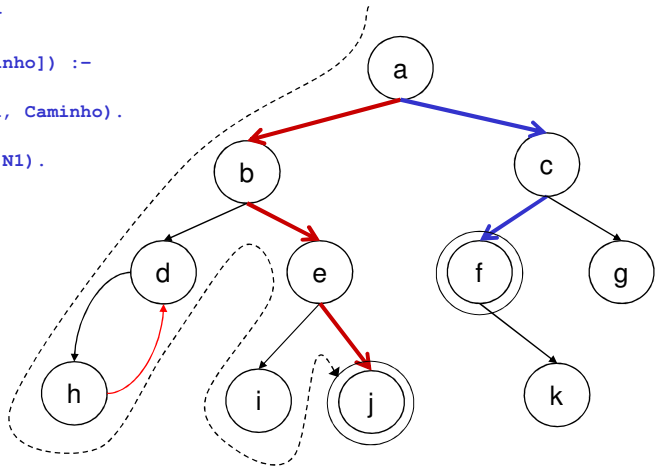
Busca em Profundidade: Ciclos



38

Busca em Profundidade

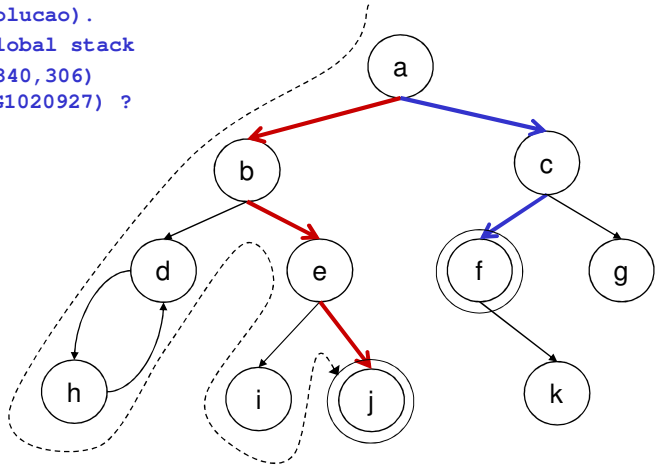
```
resolva(N, [N]) :-
        final(N).
resolva(N, [N|Caminho]) :-
        s(N, N1),
        resolva(N1, Caminho).
s(N, N1) :-
        aresta(N, N1).
aresta(a, b).
aresta(a, c).
aresta(b, d).
aresta(b, e).
aresta(c, f).
aresta(c, g).
aresta(d, h).
aresta(h, d).
aresta(e, i).
aresta(e, j).
aresta(f, k).
final(j).
final(f).
```



39

Busca em Profundidade

```
?- resolva(a, Solucao).
ERROR: Out of global stack
   Exception: (340,306)
   resolva(h, _G1020927) ?
```



Como resolver?

40

Busca em Profundidade: sem ciclos

```
% resolva(No,Solucao) Solucao é um caminho acíclico (na ordem
% reversa) entre nó inicial No e uma solução

resolva(No,Solucao) :-
  depthFirst([],No,Solucao).

% depthFirst(Caminho,No,Solucao): No é o estado a partir do qual se
   quer alcançar o estado final; Caminho é o caminho desde o estado
   inicial até No; Solucao é o Caminho até um nó final, estendido
   com No.

depthFirst(Caminho,No,[No|Caminho]) :-
  final(No).
depthFirst(Caminho,No,Solucao) :-
  s(No,No1),
  \+ pertence(No1,Caminho), % evita ciclo
  depthFirst([No|Caminho],No1,Solucao).

pertence(E,[E|_]).
pertence(E,[_|T]) :-
  pertence(E,T).
```

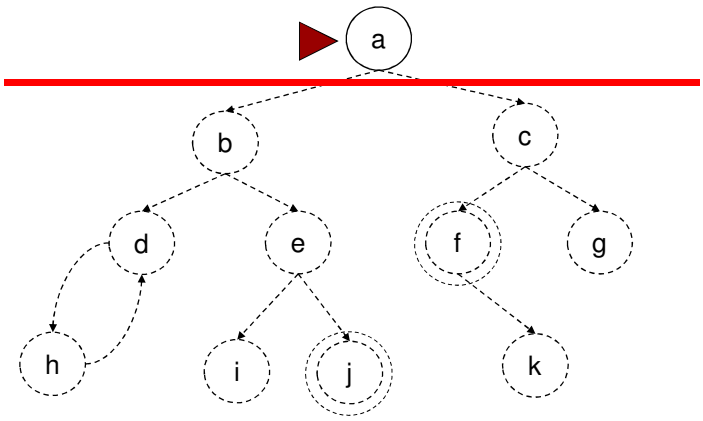
41

Busca em Profundidade

- Repare que o backtracking do Prolog faz com que não seja necessário o uso de uma pilha para guardar os estados anteriores
- Um problema com a busca em profundidade é que existem espaços de estado nos quais o algoritmo se perde: quando há ramos infinitos no espaço de estados
  - O algoritmo então explora esta parte infinita do espaço, nunca chegando perto de um nó final
  - Por exemplo, o problema das 8 Rainhas é suscetível a este tipo de armadilha, mas como o espaço é finito, as rainhas podem ser colocadas em segurança no tabuleiro, ou seja, uma solução é encontrada
  - Para evitar caminhos infinitos, um refinamento pode ser adicionado à busca em profundidade: limitar a profundidade de busca

42

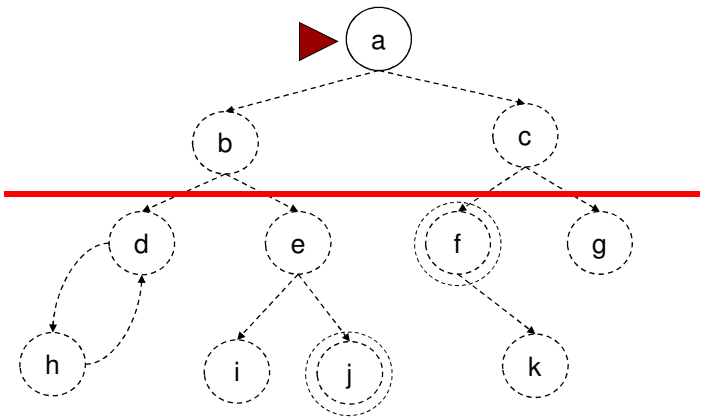
Busca em Profundidade Limitada (L=0)



Inserir na frente, remover da frente: a

43

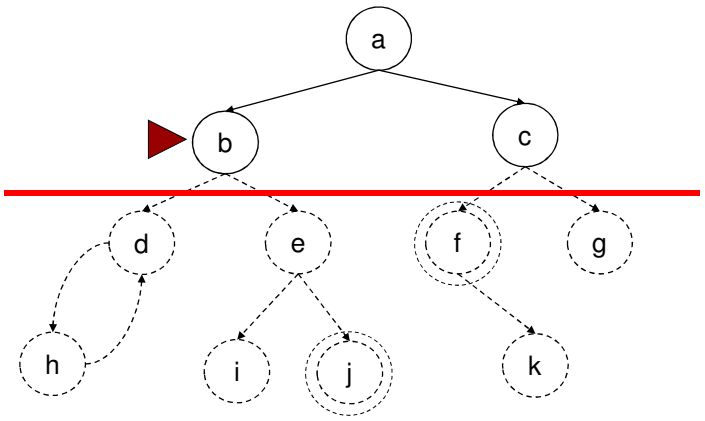
Busca em Profundidade Limitada (L=1)



Inserir na frente, remover da frente: a

44

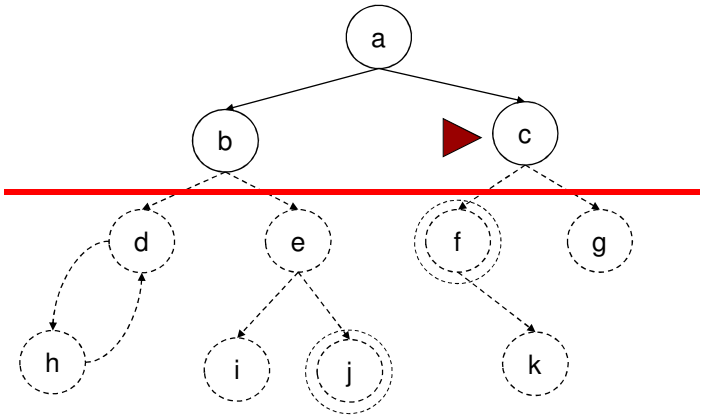
Busca em Profundidade Limitada (L=1)



Inserir na frente, remover da frente: b, c

45

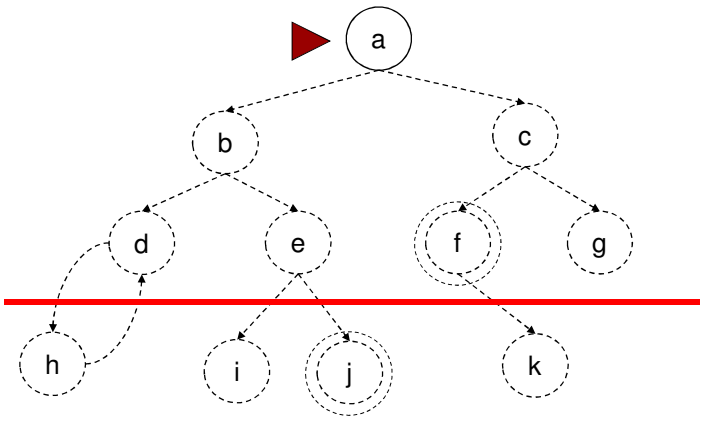
Busca em Profundidade Limitada (L=1)



Inserir na frente, remover da frente: c

46

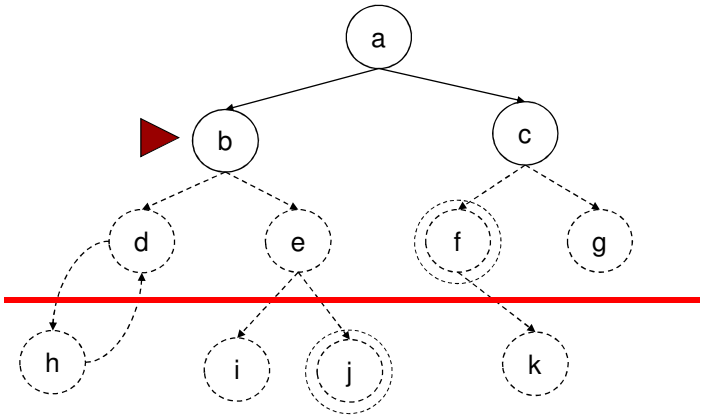
Busca em Profundidade Limitada (L=2)



Inserir na frente, remover da frente: a

47

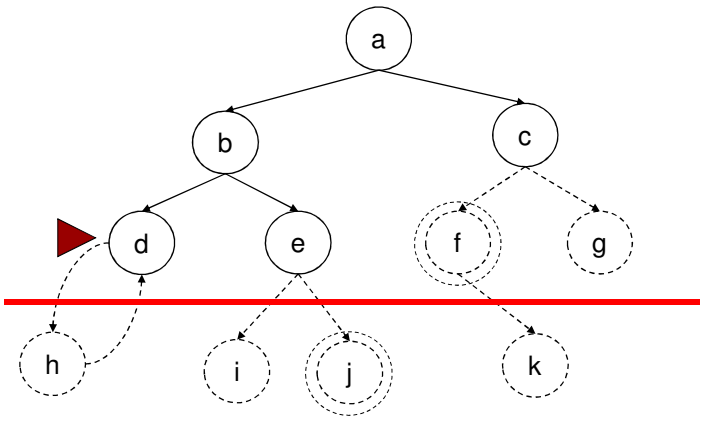
Busca em Profundidade Limitada (L=2)



Inserir na frente, remover da frente: b, c

48

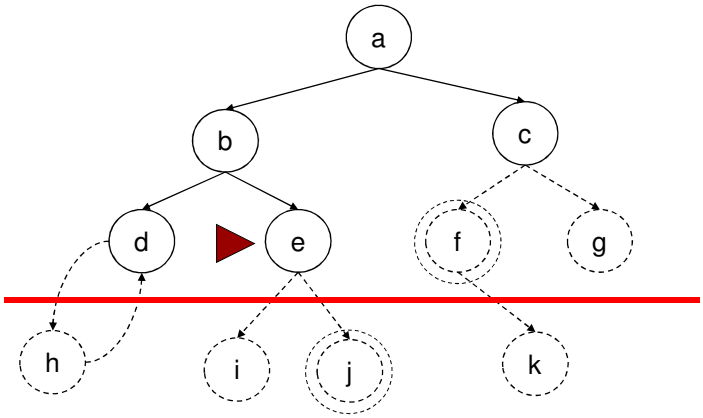
Busca em Profundidade Limitada (L=2)



Inserir na frente, remover da frente: d, e, c

49

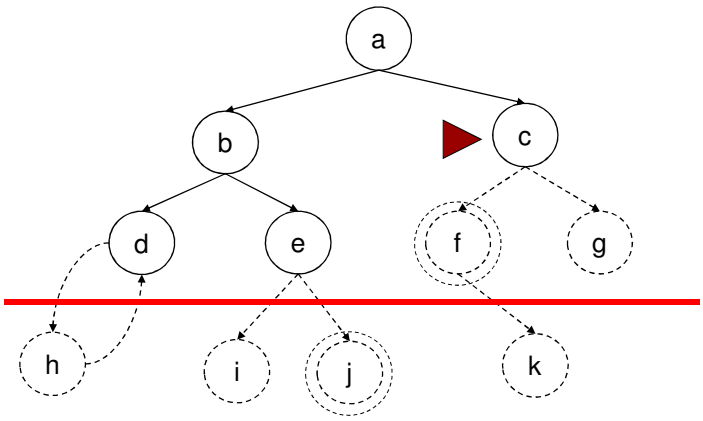
Busca em Profundidade Limitada (L=2)



Inserir na frente, remover da frente: e, c

50

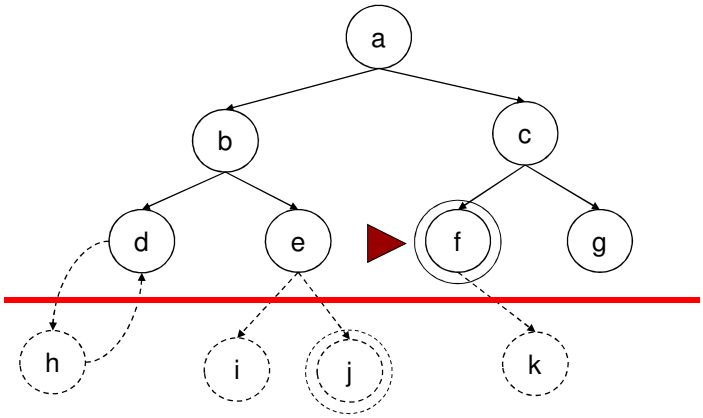
Busca em Profundidade Limitada (L=2)



Inserir na frente, remover da frente: c

51

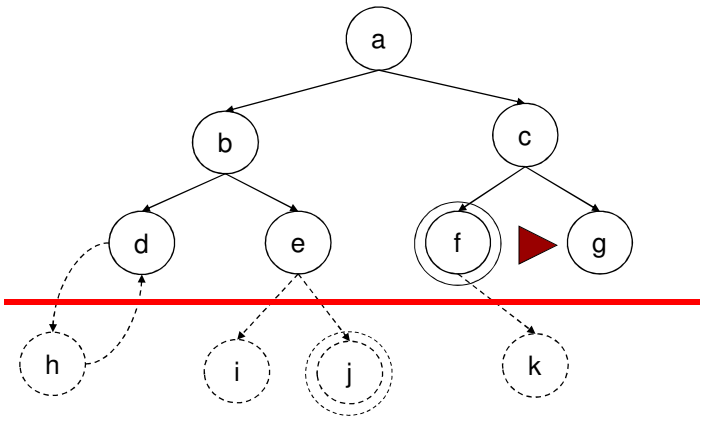
Busca em Profundidade Limitada (L=2)



Inserir na frente, remover da frente: f, g

52

Busca em Profundidade Limitada (L=2)



Inserir na frente, remover da frente: g

Busca em Profundidade Limitada

```
% resolva(No,Solucao,L) Solucao é um caminho acíclico de
   comprimento ≤ L
% (na ordem reversa) entre nó inicial No uma solução

resolva(No,Solucao,L) :-
  depthFirstLimited([],No,Solucao,L).

% depthFirstLimited(Caminho,No,Solucao,L) estende o caminho
% [No|Caminho] até um nó final obtendo Solucao com
% profundidade não maior que L

depthFirstLimited(Caminho,No,[No|Caminho],_) :-
  final(No).
depthFirstLimited(Caminho,No,Solucao,L) :-
  L > 0,                                % limita busca
  s(No,No1),
  /+ pertence(No1,Caminho),             % evita um ciclo
  L1 is L - 1,
  depthFirstLimited([No|Caminho],No1,Solucao,L1).
```

Busca em Profundidade Limitada

- Um problema com a busca em profundidade limitada é que não se tem previamente um limite razoável
  - Se o limite for muito pequeno (menor que qualquer caminho até uma solução) então a busca falha
  - Se o limite for muito grande, a busca se torna muito complexa
- Para resolver este problema, a busca em profundidade limitada pode ser executada de forma iterativa, variando o limite: comece com um limite de profundidade pequeno e aumente gradualmente o limite até que uma solução seja encontrada
- Esta técnica é denominada busca em profundidade iterativa

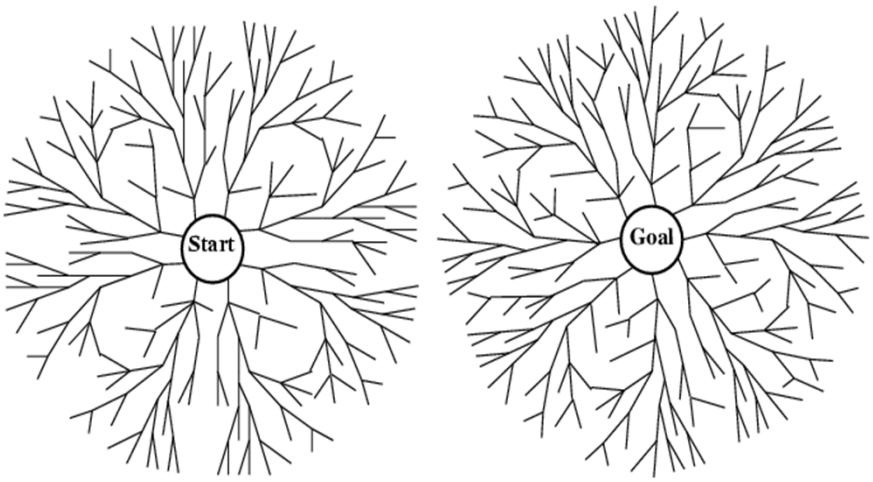
55

Sentido da Busca

- Busca *forward* e busca *backward*
  - Em muitos problemas o objetivo está apenas implícito (ex. palavras cruzadas)
  - Existem casos em que ambos os nós inicial e objetivo são conhecidos
  - Mover do estado inicial para o estado final ou vice-versa
  - Busca backward é mais eficiente quando o número de possíveis caminhos do nó objetivo para o nó inicial é menor que o número de possíveis caminhos no sentido contrário
- Mover em ambas as direções (busca bidirecional)

56

Busca Bidirecional



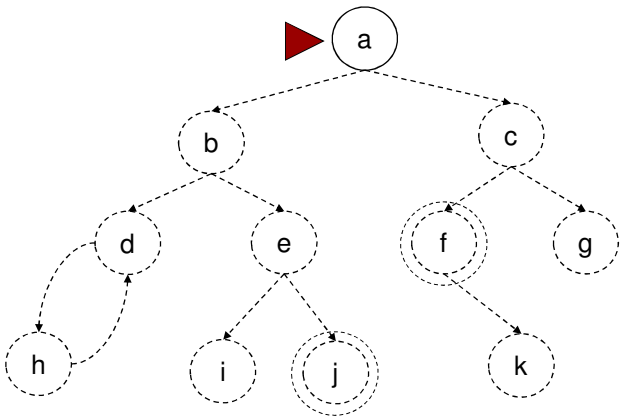
57

Busca em Largura

- Em contraste com a busca em profundidade, a busca em largura escolhe primeiro visitar aqueles nós mais próximos do nó inicial
- O algoritmo não é tão simples, pois é necessário manter um **conjunto** de nós candidatos alternativos e não apenas um único, como na busca em profundidade
- Além disso, só o conjunto não é suficiente se o caminho da solução também for necessário
- Assim, ao invés de manter um conjunto de nós candidatos, é necessário manter um conjunto de caminhos candidatos

58

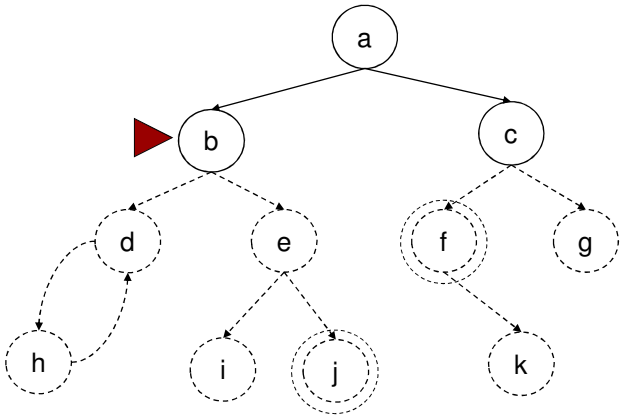
Busca em Largura



Inserir no final, remover da frente: a

59

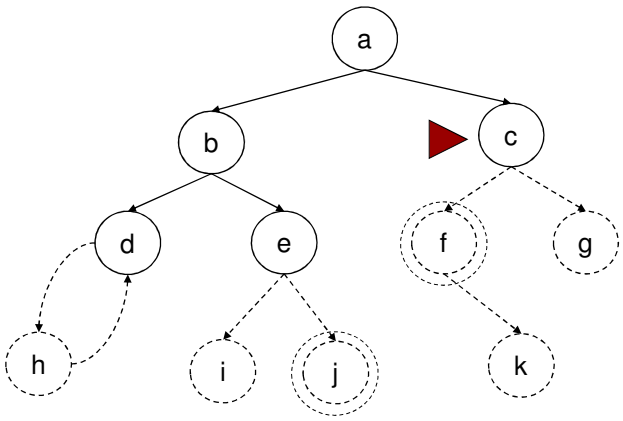
Busca em Largura



Inserir no final, remover da frente: b, c

60

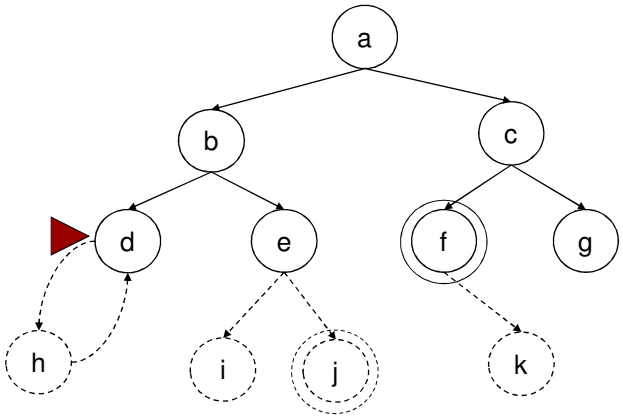
Busca em Largura



Inserir no final, remover da frente: c, d, e

61

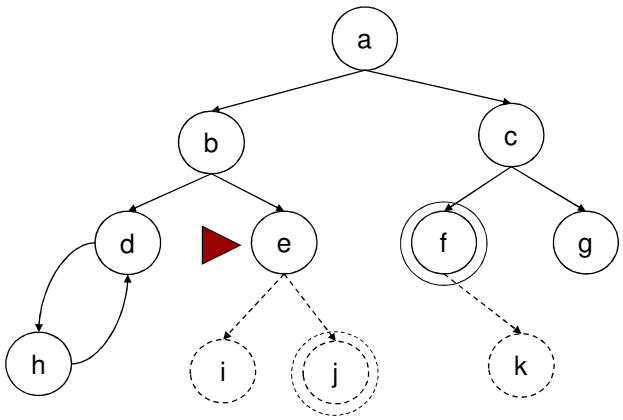
Busca em Largura



Inserir no final, remover da frente: d, e, f, g

62

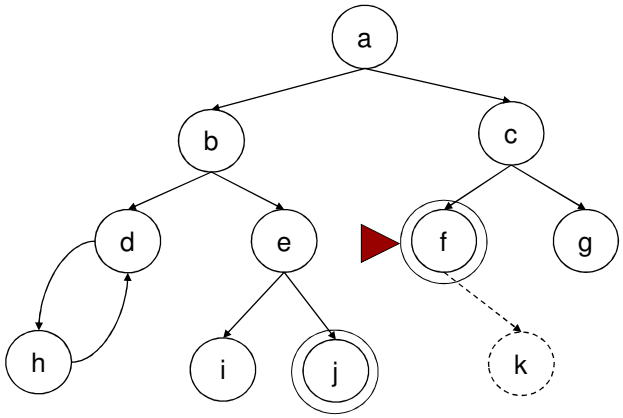
Busca em Largura



Inserir no final, remover da frente: e, f, g, h

63

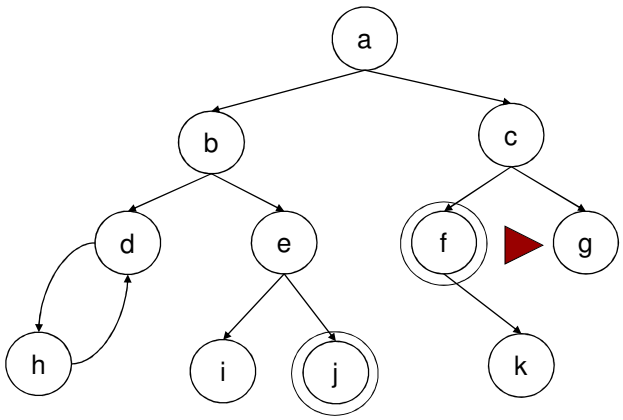
Busca em Largura



Inserir no final, remover da frente: f, g, h, i, j

64

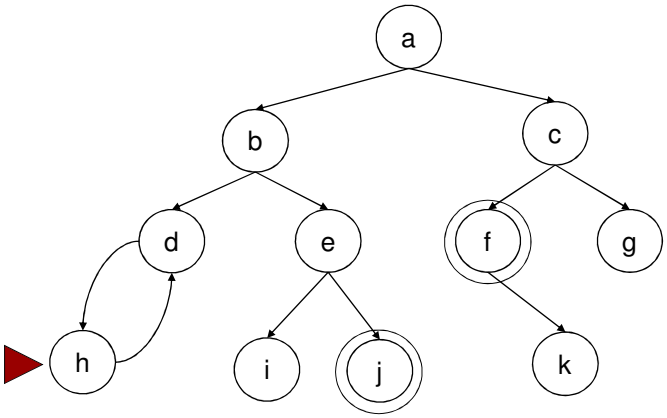
Busca em Largura



Inserir no final, remover da frente: g, h, i, j, k

65

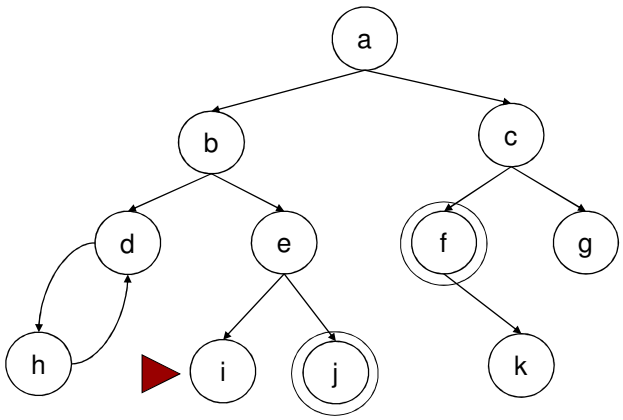
Busca em Largura



Inserir no final, remover da frente: h, i, j, k

66

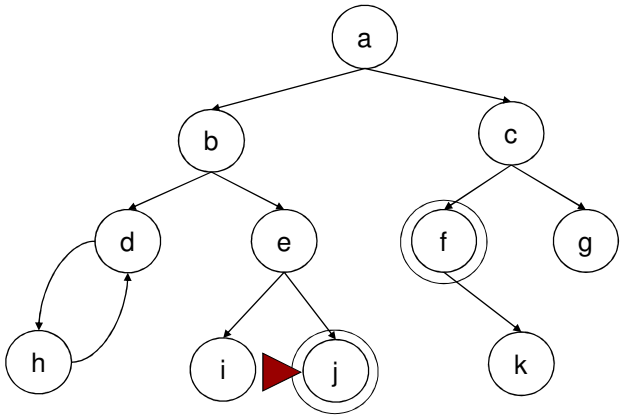
Busca em Largura



Inserir no final, remover da frente: i, j, k

67

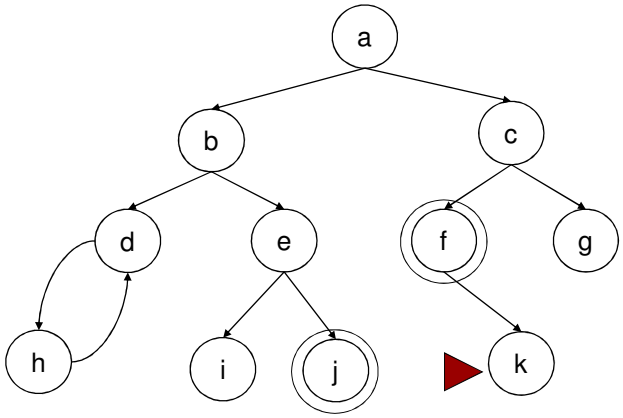
Busca em Largura



Inserir no final, remover da frente: j, k

68

Busca em Largura

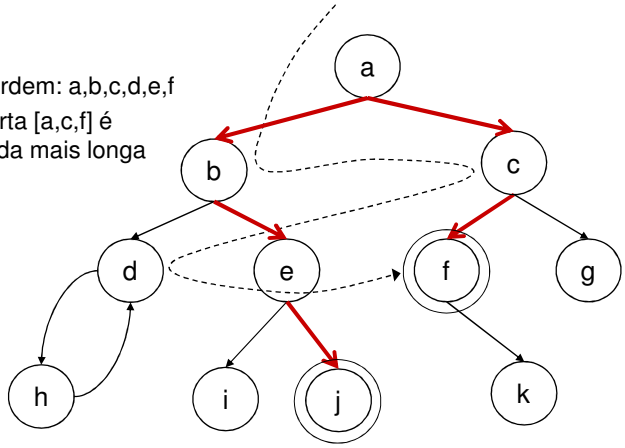


Inserir no final, remover da frente: k

69

Busca em Largura

- Estado inicial: a
- Estados finais: j,f
- Nós visitados na ordem: a,b,c,d,e,f
- A solução mais curta [a,c,f] é encontrada antes da mais longa [a,b,e,j]



70

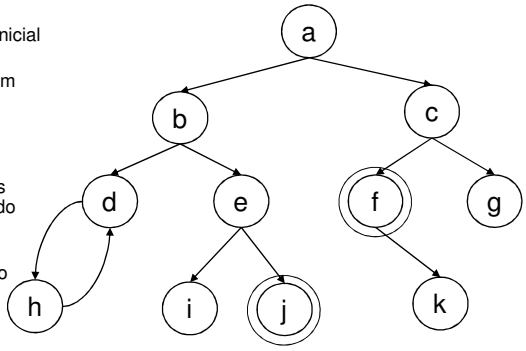
Busca em Largura em Prolog

- **breadthFirst(Caminhos,Solução)** é verdadeiro se algum caminho a partir do conjunto de candidatos **Caminhos** pode ser estendido para um nó final; **Solução** é o caminho estendido
- O conjunto de caminhos candidatos será representado como uma lista de caminhos e cada caminho será uma lista de nós na ordem reversa
- A busca inicia com um conjunto de um único candidato:
  - [[início]]
- O algoritmo é o seguinte:
  - Se a cabeça do primeiro caminho é um nó final então este caminho é uma solução; caso contrário
  - Remova o primeiro caminho do conjunto de candidatos e gere o conjunto de todas as extensões em um passo a partir deste caminho; adicione este conjunto de extensões ao final do conjunto de candidatos e execute busca em largura para atualizar este conjunto

71

Busca em Largura

- Comece com o conjunto de candidatos inicial
  - [[a]]
- Gere extensões de [a] (note que estão em ordem reversa) e teste 1o. nó do 1o. caminho:
  - [[b,a], [c,a]]
- Remova o primeiro candidato [b,a] do conjunto de candidatos e gere extensões deste caminho, adicionando-as ao final do conjunto de candidatos
  - [[c,a], [d,b,a], [e,b,a]]
- Remova [c,a] e adicione as extensões ao final
  - [[d,b,a], [e,b,a], [f,c,a], [g,c,a]]
- Estendendo [d,b,a]
  - [[e,b,a], [f,c,a], [g,c,a], [h,d,b,a]]
- Estendendo [e,b,a]
  - [[f,c,a], [g,c,a], [h,d,b,a], [i,e,b,a], [j,e,b,a]]
- A busca encontra [f,c,a] que contém um nó final, portanto o caminho é retornado como uma solução




```
s(N, N1) :-
      aresta(N, N1).
aresta(a, b).
aresta(a, c).
aresta(b, d).
aresta(b, e).
aresta(c, f).
aresta(c, g).
aresta(d, h).
aresta(h, d).
aresta(e, i).
aresta(e, j).
aresta(f, k).
final(j).
final(f).
```

Busca em Largura

- Implementar busca em largura em Prolog

73

Busca em Largura

```
% resolva(No,Solucao) Solucao é um caminho acíclico
% (na ordem reversa) entre nó inicial No e uma solução
resolva(No,Solucao) :-
  breadthFirst([[No]],Solucao).

% breadthFirst([Caminho1,Caminho2,...],Solucao), Solucao é uma
% extensão para um nó final de um dos caminhos
breadthFirst([[No|Caminho]|_],[No|Caminho]) :-
  final(No).
breadthFirst([Caminho|OutrosCaminhos],Solucao) :-
  estender(Caminho,NovosCaminhos),
  concatenar(OutrosCaminhos,NovosCaminhos,Caminhos1),
  breadthFirst(Caminhos1,Solucao).

estender([No|Caminho],NovosCaminhos) :-
  findall([NovoNo,No|Caminho],
          (s(No,NovoNo), /+ pertence(NovoNo,[No|Caminho])),
          NovosCaminhos).
```

74

Busca em Largura

findall(+Termo, +Meta,-Lista)

Findall/3 busca na base por todas as ocorrências do Termo que satisfazem à Meta, e retorna as instâncias em uma lista não ordenada contendo também as repetições.

Exs:

```
gosta(maria, joao).
gosta(ana, pedro).
gosta(clara, joao).
?-findall(X, gosta(X,joao), Lista).
Lista = [maria, clara]
```

75

Busca em Largura

```
resolva(No,Solucao) :-
  breadthFirst([[No]],Solucao).
breadthFirst([Caminho|OutrosCaminhos],Solucao) :-
  estender(Caminho,NovosCaminhos),
  concatenar(OutrosCaminhos,NovosCaminhos,Caminhos1),
  breadthFirst(Caminhos1,Solucao).
estender([No|Caminho],NovosCaminhos) :-
  findall([NovoNo,No|Caminho],
          (s(No,NovoNo), not pertence(NovoNo,[No|Caminho])),
          NovosCaminhos).
s(a, X):- aresta(a,X).
aresta(a,b).
aresta(a,c).
================================================================================

?-resolva([[a]], Solucao).
   => breadthFirst([[a]|[]], Solucao) %Caminho = [a]; Caminhos=[]
     => estender([a], NovoCaminhos)
       => findall([NovoNo,a|[]], s(a, NovoNo), /+ pertence(NovoNo, [a|[]],
                  NovosCaminhos) => NovosCaminhos= [[b,a],[c,a]]
     => concatenar ([],[[b,a],[c,a]], Caminhos1) => Caminhos1=[[b,a], [c,a]]
     => breadthFirst([[b,a],[c,a]], Solucao) ....
```

76

Algoritmos de Busca

São avaliados pelas seguintes dimensões:

- Complexidade de tempo – número de expansões de nós
- Complexidade de espaço - número máximo de nós na memória
- Completeza - o algoritmo sempre encontra uma solução se ela existe?
- Admissibilidade - um algoritmo é admissível se ele garante encontrar uma solução ótima, quando ela existe

77

Complexidade dos Algoritmos de Busca

- b = número de caminhos alternativos/fator de bifurcação/ramificação (*branching factor*)
- d = profundidade da solução
- h = profundidade máxima da árvore de busca
- l = limite de profundidade

	Tempo	Espaço	Admissível? (solução mais curta)	Completa? (encontra uma solução quando ela existe)
Profundidade	$O(b^h)$	$O(bh)$	Não	Sim (espaços finitos) Não (espaços infinitos)
Profundidade limitada	$O(b^l)$	$O(bl)$	Não	Sim se $l \geq d$
Largura	$O(b^d)$	$O(b^d)$	Sim	Sim
Bidirecional	$O(b^{d/2})$	$O(b^{d/2})$	Sim	Sim

78

Busca Informada

- A busca em grafos pode atingir uma complexidade elevada devido ao número de alternativas
- Estratégias de busca informada utilizam informação heurística sobre o problema para encurtar a busca no espaço de estados
- Essa estimativa indica o quanto o nó é promissor com relação a atingir a meta estabelecida

79

Busca Informada

Estratégias de **busca heurística** utilizam *conhecimento específico* do problema na escolha do próximo nó a ser expandido e aplicam uma *função de avaliação* a cada nó na fronteira do espaço de estados.

- Essa função de avaliação estima o custo de caminho do nó atual ao objetivo mais próximo utilizando uma função heurística
- Qual dos nós supostamente é o mais próximo do objetivo

80

Busca Informada

Função heurística *h(n)*

- **estima** o custo do caminho entre o nó n e o objetivo
- depende do problema

Exemplo:

- encontrar a rota mais curta entre São Carlos e Porto Alegre
- $h_{dd}(n)$ = distância direta entre o nó *n* e o nó final.

Como escolher uma boa função heurística?

- ela deve ser admissível i.e., nunca *superestimar* o custo real da solução

  h(n) ≤ h* (h* é o custo real da solução)
- Distância direta (h_{dd}) é *admissível* porque o caminho mais curto entre dois pontos é sempre uma linha reta

81

Busca Informada

O nó de menor custo na fronteira do espaço de estados é expandido primeiro

Duas abordagens básicas de ***best-first search***:

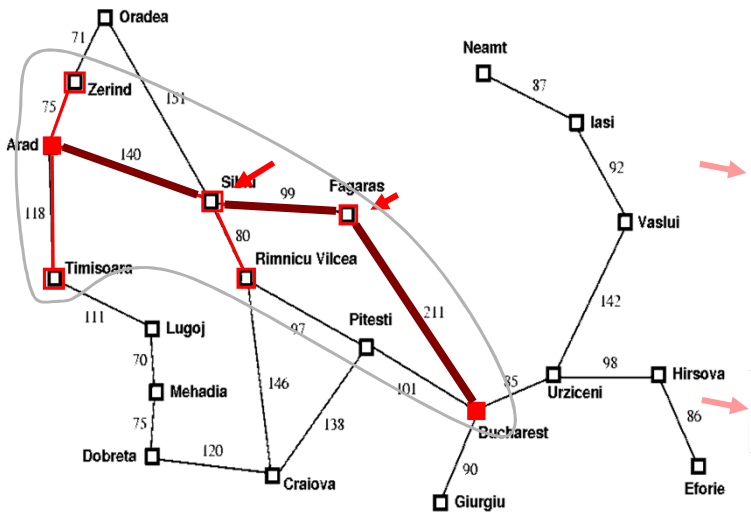
1. Busca Gulosa (*Greedy search*)
2. Algoritmo A*

82

Busca Gulosa

- Semelhante à busca em profundidade com backtracking
- Tenta expandir o nó mais próximo ao nó final com base na estimativa feita pela função heurística h
- Custo de busca é minimizado
  - não expande nós fora do caminho
- Escolhe o caminho que é mais econômico à primeira vista
- Não é ótima... (semelhante à busca em profundidade)
  - porque só olha para o futuro!
- ... nem é completa:
  - pode entrar em "loop" se não detectar a expansão de estados repetidos
  - pode tentar desenvolver um caminho infinito
- Custo de tempo e memória: O(bd)
  - guarda todos os nós expandidos na memória

Busca Gulosa



Distância em linha reta para Bucharest:

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Busca gulosa

- O que aconteceu? Por que não escolheu o melhor caminho?

85

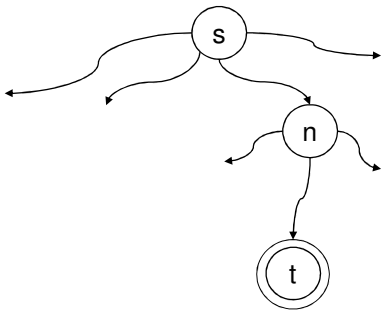
Exercício em duplas

- Implementar busca gulosa

86

Considerando também o caminho passado....

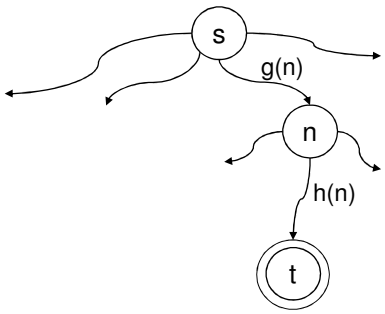
- Vamos assumir que há um custo envolvido entre cada arco:
  - **s(X,Y,C)** é verdadeira se há um movimento permitido no espaço de estados do nó X para o nó Y, de custo C; neste caso, Y é um **sucessor** de X
- Sejam dados um nó inicial **s** e um nó final **t**
- Seja o estimador heurístico a função **f** tal que, para cada nó **n** no espaço, **f(n)** é a estimativa do custo do caminho mais barato de **s** até **t,** via **n.**



87

Considerando também o caminho passado....

- A função f(n) será construída como: **f(n) = g(n) + h(n)**
  - **g(n)** é uma estimativa do custo do caminho ótimo de **s** até **n**
  - **h(n)** é uma estimativa do custo do caminho ótimo de **n** até **t**



88

Considerando também o caminho passado....

- Quando um nó **n** é encontrado pelo processo de busca temos a seguinte situação
  - Um caminho de **s** até **n** já foi encontrado e seu custo pode ser calculado como a soma dos custos dos arcos no caminho
    - Este caminho não é necessariamente um caminho ótimo de **s** até **n** (pode existir um caminho melhor de **s** até **n** ainda não encontrado pela busca) mas seu custo serve como uma estimativa **g(n)** do custo mínimo de **s** até **n**
  - O outro termo, **h(n)** é mais problemático pois o "mundo" entre **n** e **t** não foi ainda explorado
    - Portanto, **h(n)** é tipicamente uma heurística, baseada no conhecimento geral do algoritmo sobre o problema em questão
    - Como **h** depende do domínio do problema, não há um método universal para construir **h**

89

Algoritmo A*

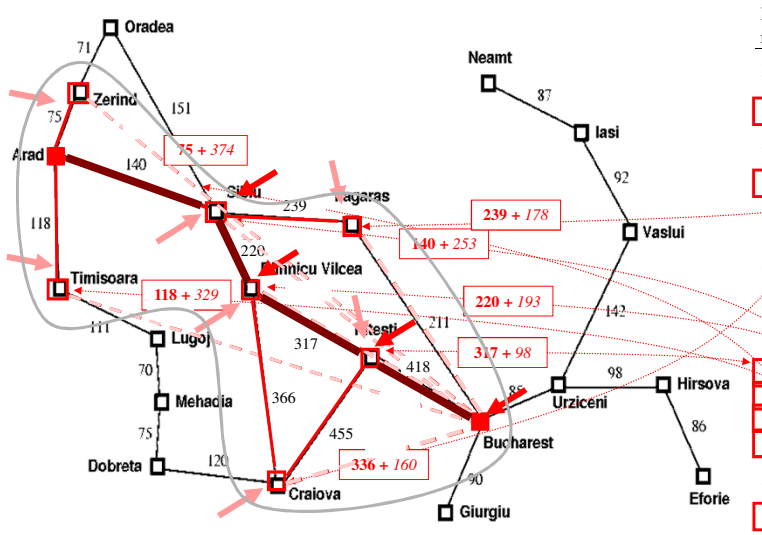
- A* expande o nó de menor valor de f na fronteira do espaço de estados.
  - Olha o futuro sem esquecer do passado!
- Se h é admissível, f(n) nunca irá superestimar o custo real da melhor solução através de n.

$f(n) \leq f^*$ (f^* é o custo ótimo)

- Neste caso, pode-se encontrar a rota de fato mais curta entre Arad e Bucarest.

90

Algoritmo A*



Distância em linha reta para Bucharest:

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

91

A*

- É conveniente relembrar que uma estratégia de busca é definida por meio da ordem de expansão dos nós
- Em estratégias de busca *best-first* (*o melhor primeiro*), a idéia básica é prosseguir com a busca sempre a partir do nó mais promissor
- Best-First é um refinamento da busca em largura
  - Ambas estratégias começam pelo nó inicial e mantêm um conjunto de caminhos candidatos
  - Busca em largura expande o caminho candidato mais curto
  - Best-First refina este princípio calculando uma estimativa heurística para cada candidato e escolhe expandir o melhor candidato segundo esta estimativa

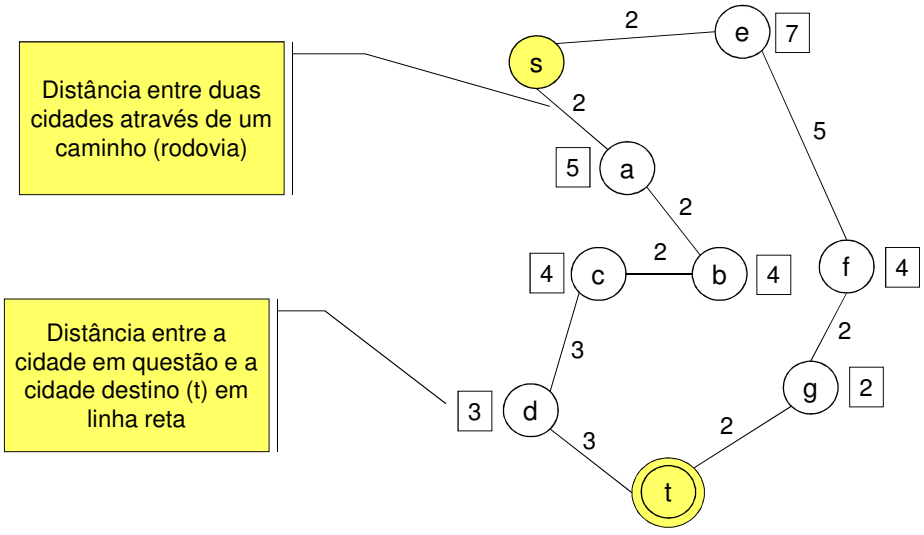
92

A*

- O processo de busca pode ser visto como um conjunto de sub-processos, cada um explorando sua própria alternativa, ou seja, sua própria sub-árvore
- Sub-árvores têm sub-árvores que são exploradas por sub-processos dos sub-processos, etc.
- Dentre todos os processos apenas um encontra-se ativo a cada momento: aquele que lida com a alternativa atual mais promissora (aquela com menor valor f)
- Os processos restantes aguardam silenciosamente até que a estimativa f atual se altere e alguma outra alternativa se torne mais promissora
- Então, a atividade é comutada para esta alternativa

93

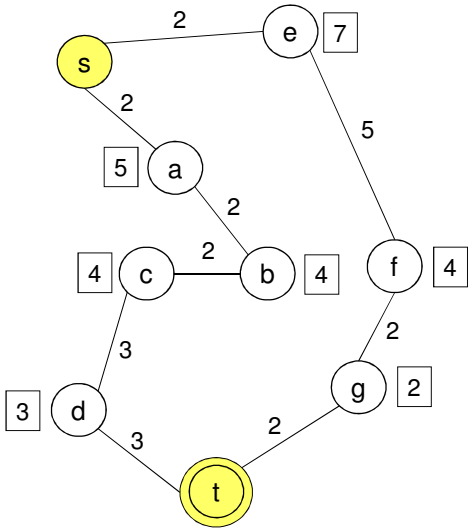
A*



94

A*

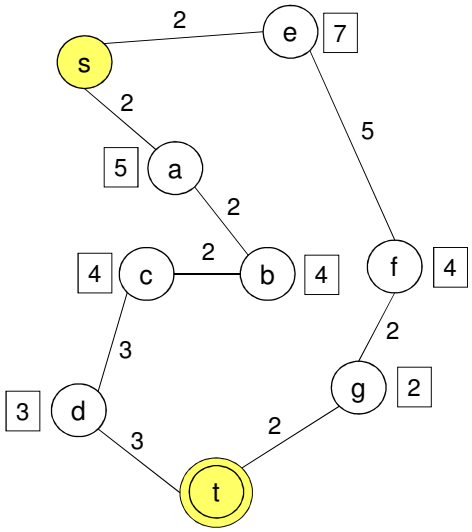
- Dado um mapa, o objetivo é encontrar o caminho mais curto entre a cidade inicial **s** e a cidade destino **t**
- Para estimar o custo do caminho restante da cidade X até a cidade **t** utilizaremos a distância em linha reta denotada por **dist(X,t)**
- f(X) = g(X) + h(X) =
  = g(X) + dist(X,t)



95

A*

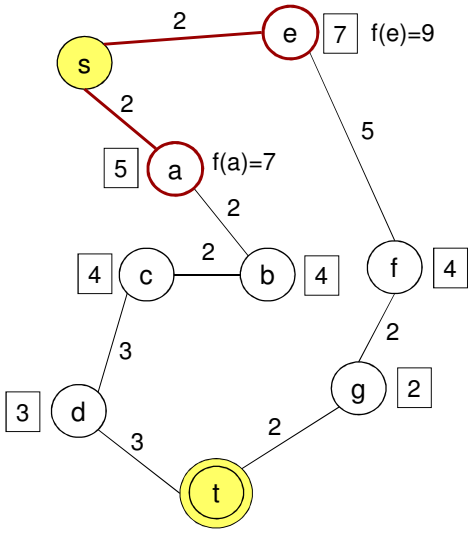
- Neste exemplo, podemos imaginar a busca best-first consistindo em dois processos, cada um explorando um dos caminhos alternativos
- Processo 1 explora o caminho via **a**
- Processo 2 explora o caminho via **e**



96

A*

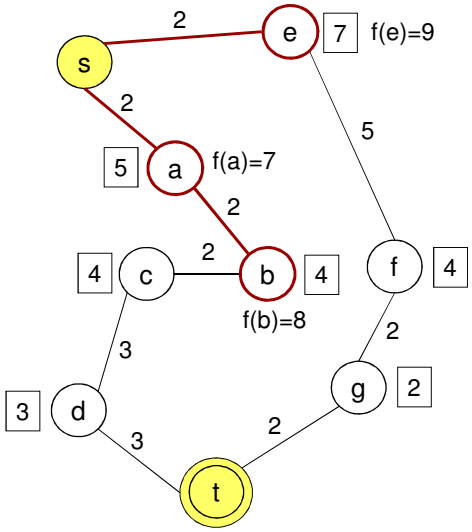
- f(a)=g(a)+dist(a,t)=2+5=7
- f(e)=g(e)+dist(e,t)=2+7=9
- Como o valor-f de **a** é menor do que de **e**, o processo 1 (busca via **a**) permanece ativo enquanto o processo 2 (busca via **e**) fica em estado de espera



97

A*

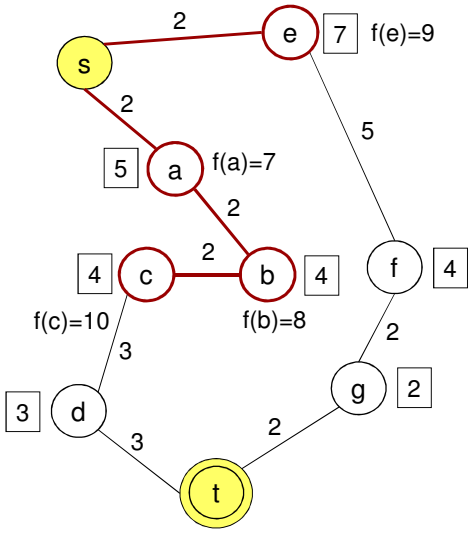
- f(a)=g(a)+dist(a,t)=2+5=7
- f(e)=g(e)+dist(e,t)=2+7=9
- Como o valor-f de **a** é menor do que de **e**, o processo 1 (busca via **a**) permanece ativo enquanto o processo 2 (busca via **e**) fica em estado de espera
- f(b)=g(b)+dist(b,t)=4+4=8



98

A*

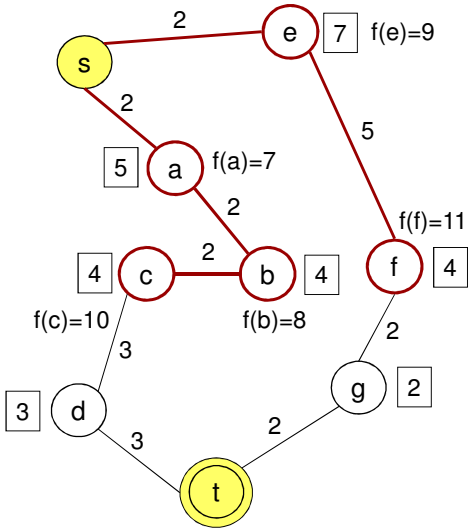
- f(a)=g(a)+dist(a,t)=2+5=7
- f(e)=g(e)+dist(e,t)=2+7=9
- Como o valor-f de **a** é menor do que de **e**, o processo 1 (busca via **a**) permanece ativo enquanto o processo 2 (busca via **e**) fica em estado de espera
- f(b)=g(b)+dist(b,t)=4+4=8
- f(c)=g(c)+dist(c,t)=6+4=10
- Como f(e)<f(c) agora o processo 2 prossegue para a cidade **f**



99

A*

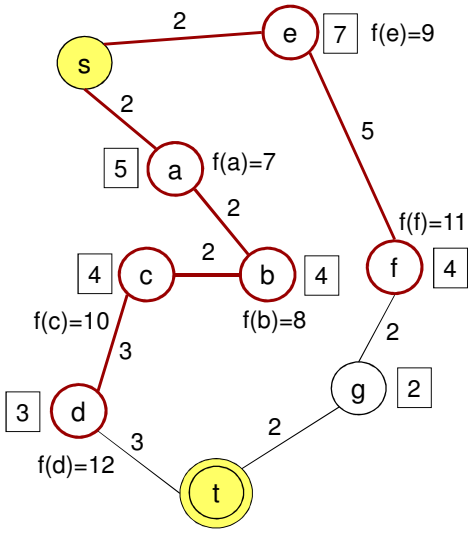
- f(f)=g(f)+dist(f,t)=7+4=11
- Como f(f)>f(c) agora o processo 2 espera e o processo 1 prossegue



100

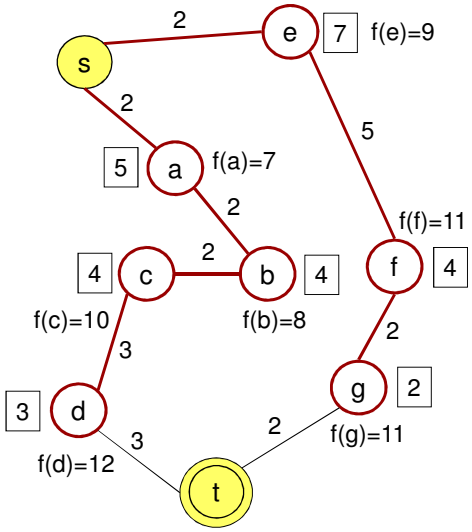
A*

- f(f)=g(f)+dist(f,t)=7+4=11
- Como f(f)>f(c) agora o processo 2 espera e o processo 1 prossegue
- f(d)=g(d)+dist(d,t)=9+3=12
- Como f(d)>f(f) o processo 2 reinicia



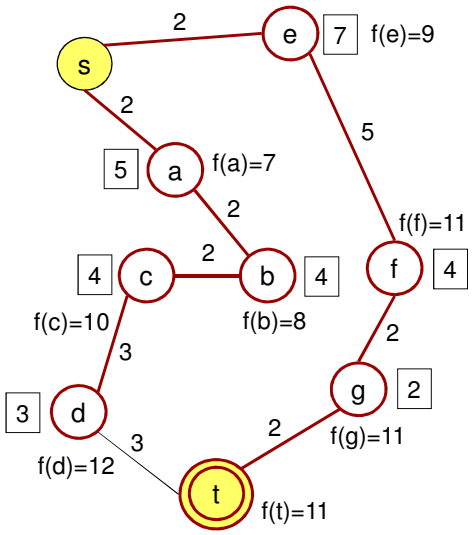
A*

- f(f)=g(f)+dist(f,t)=7+4=11
- Como f(f)>f(c) agora o processo 2 espera e o processo 1 prossegue
- f(d)=g(d)+dist(d,t)=9+3=12
- Como f(d)>f(f) o processo 2 reinicia chegando até o destino **t**
- f(g)=g(g)+dist(g,t)=9+2=11



A*

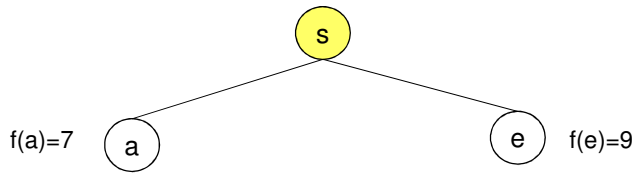
- f(f)=g(f)+dist(f,t)=7+4=11
- Como f(f)>f(c) agora o processo 2 espera e o processo 1 prossegue
- f(d)=g(d)+dist(d,t)=9+3=12
- Como f(d)>f(f) o processo 2 reinicia chegando até o destino **t**
- f(g)=g(g)+dist(g,t)=9+2=11
- f(t)=g(t)+dist(t,t)=11+0=11



A*

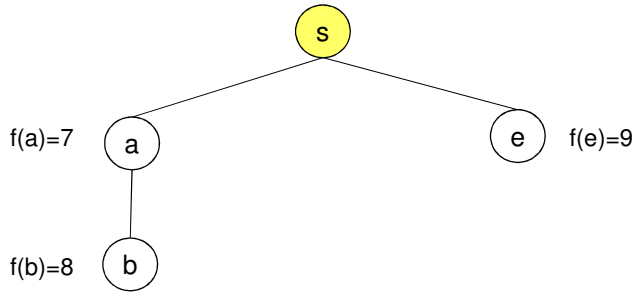
- A busca, começando pelo nó inicial continua gerando novos nós sucessores, sempre expandindo na direção mais promissora de acordo com os valores-f
- Durante este processo, uma árvore de busca é gerada tendo como raiz o nó inicial e o algoritmo best-first continua expandindo a árvore de busca até que uma solução seja encontrada

A*



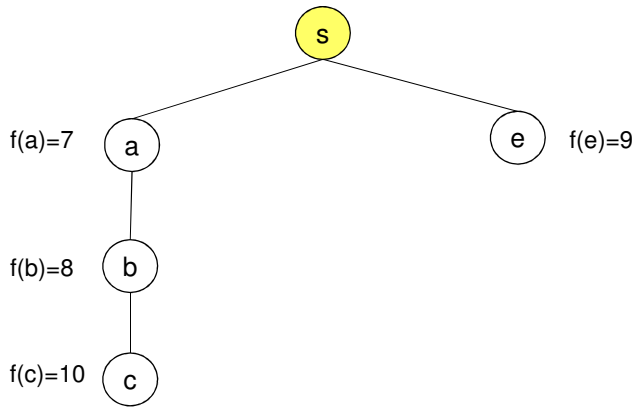
105

A*



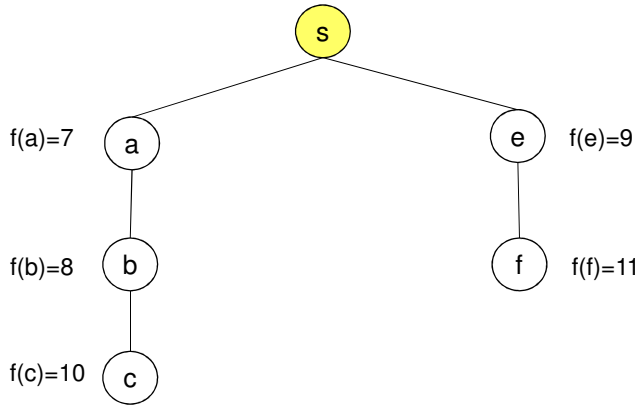
106

A*



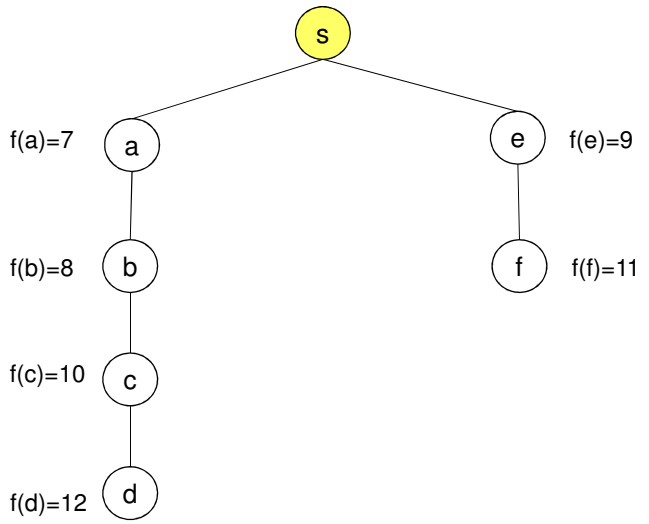
107

A*



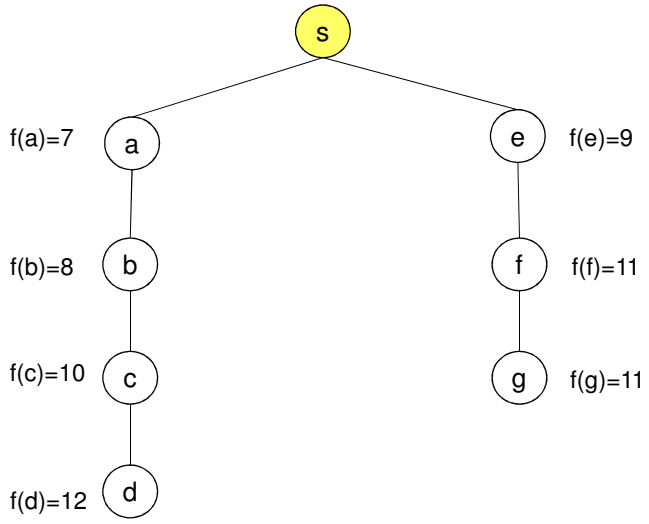
108

A*


s
f(a)=7
a
e
f(e)=9
f(b)=8
b
f
f(f)=11
f(c)=10
c
f(d)=12
d

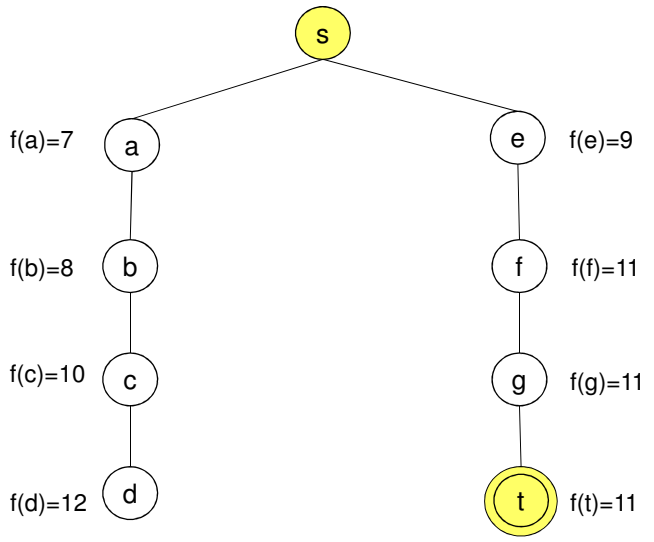
109

A*


s
f(a)=7
a
e
f(e)=9
f(b)=8
b
f
f(f)=11
f(c)=10
c
g
f(g)=11
f(d)=12
d

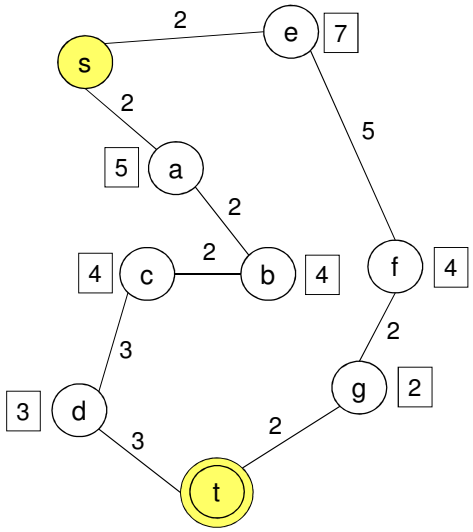
110

A*



111

O que a busca gulosa faria?



112

Algoritmo A* : análise do comportamento

- Custo de tempo:
  - exponencial com o comprimento da solução, porém boas funções heurísticas diminuem significativamente esse custo
- Custo memória: O (bd)
  - guarda todos os nós expandidos na memória
    - **para possibilitar o backtracking**
- Eficiência ótima
  - só expande nós com $f(n) \leq f^*$, onde f^* é o custo do caminho ótimo
    - **f é não decrescente**
  - nenhum outro algoritmo ótimo garante expandir menos nós.

Exercício em duplas

- Implementar busca A*
  - Tente alterar a implementação da busca gulosa

Busca local

- Até agora, vimos métodos de busca que exploram o espaço de busca sistematicamente
  - Muitas vezes, guardam o "caminho" para a solução
- Se o caminho não interessa, algoritmos de busca local são úteis
  - Consideram somente o estado atual
  - Movem-se para estados vizinhos do estado atual
  - Usam pouca memória
  - Podem encontrar uma boa solução em espaços de busca grandes ou infinitos, nos quais as buscas sistemáticas falhariam
    - Úteis em problemas de design de circuitos integrados, layout de chão de fábrica, otimização de redes de telecomunicações, problemas de otimização em geral, etc.

115

Hill-Climbing

- "É como escalar o monte Everest em um nevoeiro denso com amnésia"
- "É como usar óculos que limitam sua visão a 3 metros"
- Hill-Climbing: função de avaliação é vista como qualidade
- Também conhecido como gradiente descendente

116

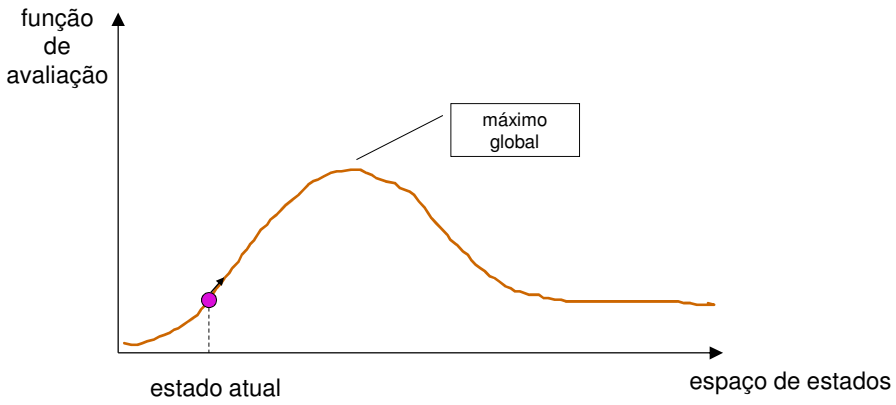
Hill-Climbing

Algoritmo

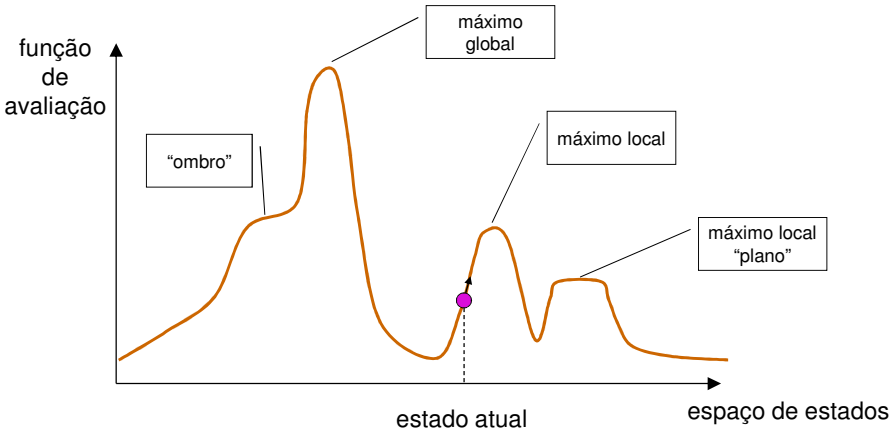
- Escolha um estado inicial do espaço de busca de forma aleatória
- Considere todos os vizinhos (sucessores) no espaço de busca
- Escolha o vizinho com a melhor qualidade e mova para aquele estado
- Repita os passos de 2 até 4 até que todos os estados vizinhos tenham menos qualidade que o estado atual
- Retorne o estado atual como sendo a solução

  - Se há mais de um vizinho com a melhor qualidade:
    - Escolher o primeiro melhor
    - Escolher um entre todos de forma aleatória

Hill-Climbing



Hill-Climbing



119

Hill-Climbing: Problemas

- Máximo local: uma vez atingido, o algoritmo termina mesmo que a solução esteja longe de ser satisfatória
- Platôs (regiões planas): regiões onde a função de avaliação é essencialmente plana; a busca torna-se como uma caminhada aleatória
- Cumes ou "ombros": regiões que são alcançadas facilmente mas até o topo a função de avaliação cresce de forma amena; a busca pode tornar-se demorada

120

Hill-Climbing: Análise

- O algoritmo é completo?
  - **SIM**, uma vez que cada nó tratado pelo algoritmo é sempre um estado completo (uma solução)
- O algoritmo é ótimo?
  - **TALVEZ**, quando iterações suficientes forem permitidas...
  - O sucesso deste método depende muito do formato da superfície do espaço de estados:
    - Se há poucos máximos locais, o reinício aleatório encontra uma boa solução rapidamente

Hill-Climbing

```
% resolva(No,Solucao) Solucao é um caminho
% acíclico (na ordem reversa) entre nó
% inicial No e uma solução
% l(N,F/G) denota o nó N com valores F=f(n) e
% G = g(N)

resolva(No,Solucao) :-
  hillclimbing([],l(No,0/0),Solucao).

hillclimbing(Caminho,l(No,F/G),[No|Caminho]) :-
  final(No).
hillclimbing(Caminho,l(No,F/G),S) :-
  findall(No1/Custo,
          (s(No,No1,Custo),\+ pertence(No1,Caminho)),
          Vizinhos
         ),
  avalie(G,Vizinhos,VizinhosAvaliados),
  melhor_qualidade(VizinhosAvaliados,MelhorNo),
  hillclimbing([No|Caminho],MelhorNo,S).

avalie(_,[],[]).
avalie(G0,[No/Custo|NaoAvaliados],[l(No,F/G)|Avaliados]) :-
  G is G0 + Custo,
  h(No,H),           % H = h(No) função heurística
  F is G + H,
  avalie(G0,NaoAvaliados,Avaliados).

melhor_qualidade([No|Nos],Melhor) :-
  minimo_f(Nos,No,Melhor).
minimo_f([],No,No).
minimo_f([No|Nos],MinAtual,Min) :-
  gt(No,MinAtual), !,
  minimo_f(Nos,MinAtual,Min).
minimo_f([No|Nos],MinAtual,Min) :-
  minimo_f(Nos,No,Min).

gt(l(_,F1/_),l(_,F2/_)) :-
  F1 > F2.
```

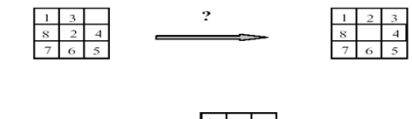
Hill-climbing

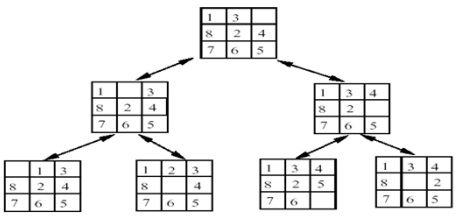
- Exemplo de problema
  - Tradução automática

123

Exercício

- Implementação em prolog do problema do jogo 8-puzzle
  - Representação do estado
  - Busca informada
    - Função de avaliação





124