Estratégias de Busca

- Nesta aula são descritas algumas estratégias de busca em espaço de estados
  - Busca não informada
  - Busca informada

Inteligência Artificial

José Augusto Baranauskas
Departamento de Física e Matemática – FFCLRP-USP
E-mail: augusto@usp.br
URL: http://dfm.fmrp.usp.br/~augusto

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

Exemplo: Quebra-Cabeça-8

5	4	
6	1	8
7	3	2

Estado Inicial

1	2	3
8		4
7	6	5

Estado Final

- Estados?
- Operadores?
- Estado Final: = estado fornecido
- Custo do caminho?

Exemplo: Quebra-Cabeça-8

5	4	
6	1	8
7	3	2

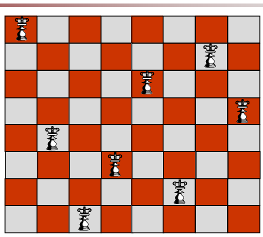
Estado Inicial

1	2	3
8		4
7	6	5

Estado Final

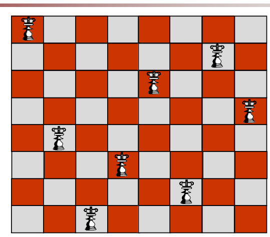
- Estados: posições inteiras dos quadrados (ignorar posições intermediárias)
- Operadores: mover se branco à esquerda, direita, em cima, em baixo (ignorar ação de desalojar, etc)
- Estado Final: = estado fornecido
- Custo do caminho: 1 por movimento

Exemplo: 8 Rainhas



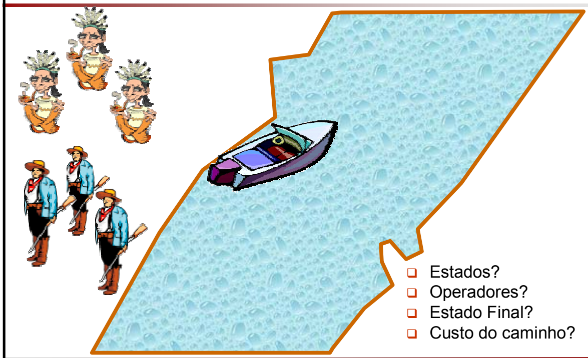
- Estados?
- Operadores?
- Estado Final?
- Custo do caminho?

Exemplo: 8 Rainhas



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

Exemplo: Missionários e Canibais

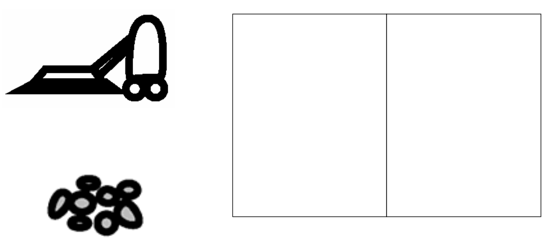


- Estados?
- Operadores?
- Estado Final?
- Custo do caminho?

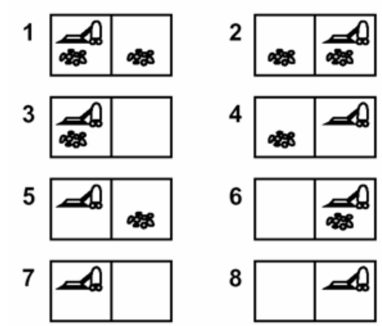
Exemplo: Missionários e Canibais

- Estados
  - um estado é uma seqüência ordenada de três números representando o número de missionários, canibais e botes na margem do rio na qual eles iniciam
  - Assim o estado inicial é [3,3,1]
- Operadores
  - a partir de cada estado os operadores possíveis são tomar um missionário e um canibal, dois missionários ou dois canibais
  - Há no máximo 5 operadores, embora alguns estados tenham menos operadores uma vez que deve-se evitar estados inválidos
  - Se for necessário distinguir os indivíduos, haveria 27 operadores ao invés de apenas 5
- Estado Final: [0,0,0]
- Custo do caminho: número de travessias

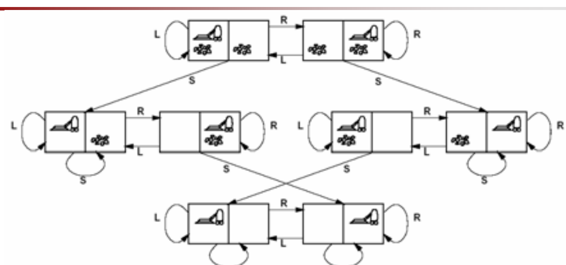
Exemplo: Mundo do Aspirador



Exemplo: Mundo do Aspirador

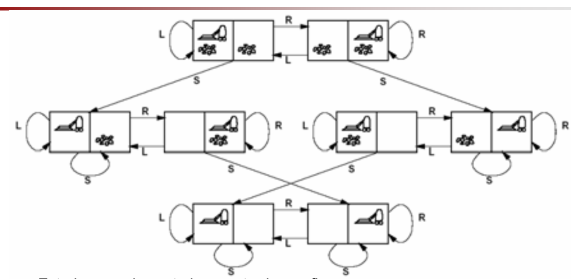


Exemplo: Mundo do Aspirador



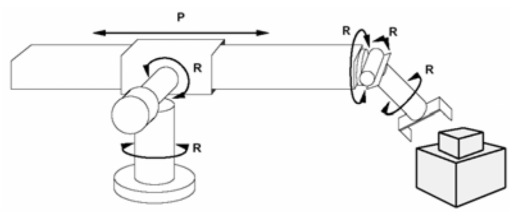
- Estados?
- Operadores?
- Estado Final?
- Custo do caminho?

Exemplo: Mundo do Aspirador



- Estados: um dos estados mostrados na figura
- Operadores: L (esquerda), R (direita), S (sucção)
- Estado Final: nenhuma sujeira em todos os ambientes
- Custo do caminho: cada ação tem custo unitário

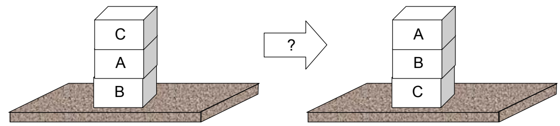
Exemplo: Montagem com Robô



- 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
- Custo do caminho: tempo para montagem

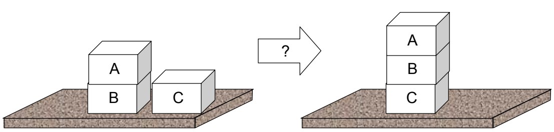
Exemplo: Pilha de Blocos

- 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

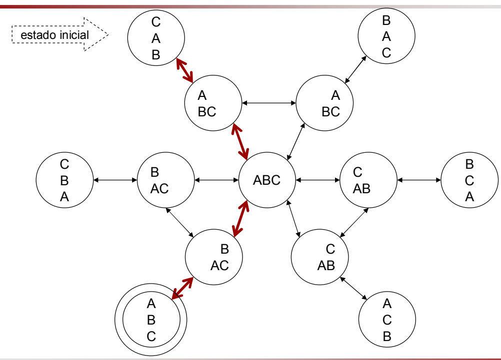


Exemplo: Pilha de Blocos

- 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)



Exemplo: Pilha de Blocos



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)
    - Profundidade iterativa (Iterative Deepening)
    - Largura (Breadth-first)
    - Bidirecional
- Estratégias Heurísticas de Busca (Busca Informada)
  - utiliza informações específicas do domínio para ajudar na decisão
    - Hill-Climbing
    - Best-First
    - A*
    - IDA* (Iterative Deepening A*)
    - RBFS (Recursive Best-First Search)

Busca em Espaço de Estados

- Vamos representar um espaço de estados pela 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

Busca em Espaço de Estados

- Outra questão importante é como representar situações do problema, que são por si só nós no espaço de estados
- A representação deve ser compacta e permitir execução eficiente das operações requeridas
- No exemplo da manipulação de blocos, vamos considerar o caso geral em que há um número qualquer de blocos que devem ser dispostos em uma ou mais pilhas:
  - O número de pilhas será limitado para tornar o problema mais interessante, além de ser realista uma vez que robôs que manipulam blocos possuem um espaço limitado na mesa
  - Uma situação do problema pode ser representada por uma lista de pilhas; cada pilha é representada por uma lista de blocos (de forma ordenada) de forma que o bloco no topo da pilha é a cabeça da lista
  - Pilhas vazias serão representadas por listas vazias
  - Situação inicial: [[c,a,b], [ ], [ ]]
  - Situação final:
    - [[a,b,c], [ ], [ ]]
    - [[ ], [a,b,c], [ ]]
    - [[ ], [ ], [a,b,c]]

Busca em Espaço de Estados

- A relação sucessor pode ser programada da seguinte forma: Situação2 é sucessora de Situação1 se há duas pilhas, Pilha1 e Pilha2 em Situação1 e o topo da Pilha1 pode ser movido para Pilha2
- Todas as situação podem ser representadas por listas de pilhas

```
s(Pilhas,[Pilha1, [Topo|Pilha2]|OutrasPilhas]) :- % Move Topo p/ Pilha2
  del([Topo|Pilha1],Pilhas,Resto),             % encontre 1a pilha
  del(Pilha2,Resto,OutrasPilhas).              % encontre 2a pilha

del(E,[E|L],L).
del(E,[Y|L],[Y|L1]) :-
  del(E,L,L1).
```

- A situação final do nosso exemplo é:

```
final(Situacao) :-
  pertence([a,b,c],Situacao).
```


Busca em Espaço de Estados

- Vamos programar algoritmos de busca 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
- No exemplo de manipulação de blocos, a chamada correspondente seria:

```
?- resolva([[c,a,b],[],[]],Solucao).
```

- Como resultado de uma busca bem sucedida, **Solucao** é instanciada com uma lista de arranjos de blocos que representa um plano para transformar o estado inicial em um estado em que todos os três blocos estão em uma única pilha arranjados como [a,b,c]

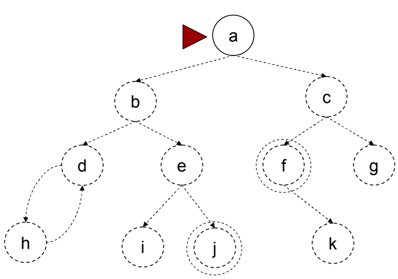
Busca em Profundidade

- O algoritmo para busca em profundidade é o seguinte: para encontrar uma solução S para um dado nó N (até um nó final):
  - Se N é um nó final então S = [N]
  - Se N tem um sucessor N1 tal que há um caminho S1 de N1 até um nó final, então S = [N | S1]
- Em Prolog:

```
resolva(N,[N]) :-
  final(N).
resolva(N,[N|S1]) :-
  s(N,N1),
  resolva(N1,S1).
```

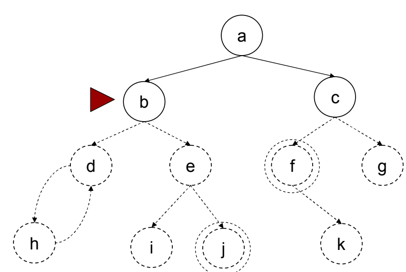
- A busca em profundidade é o recurso mais simples em programação recursiva e, por isso, Prolog quando executa metas explora alternativas utilizando-a
- Entretanto, não há detecção de ciclos

Busca em Profundidade



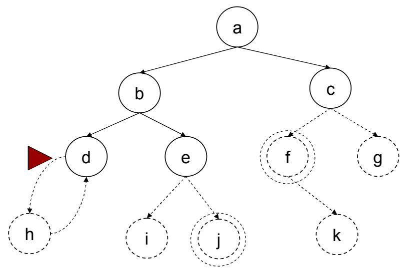
Inserir na frente, remover da frente: a

Busca em Profundidade



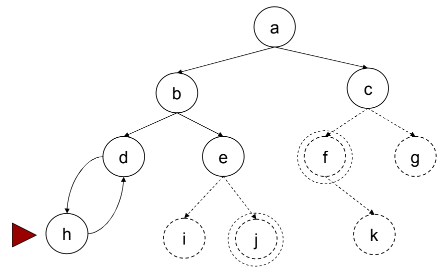
Inserir na frente, remover da frente: b, c

Busca em Profundidade



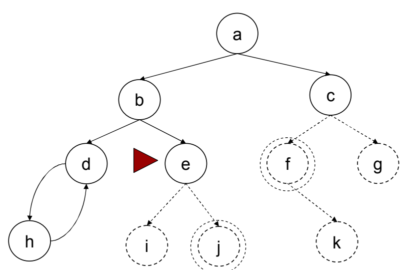
Inserir na frente, remover da frente: d, e, c

Busca em Profundidade



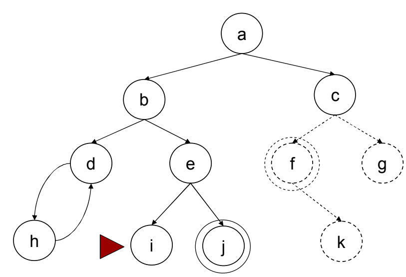
Inserir na frente, remover da frente: h, e, c

Busca em Profundidade



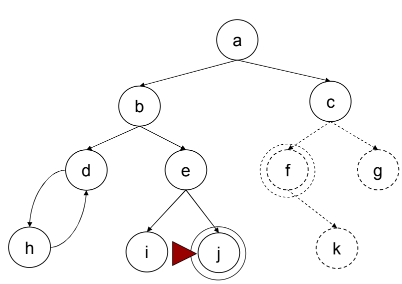
Inserir na frente, remover da frente: e, c

Busca em Profundidade



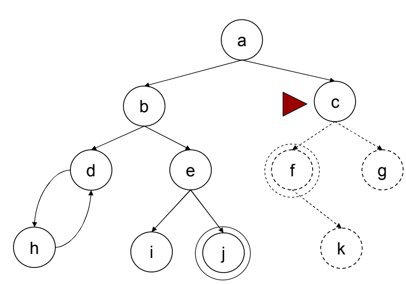
Inserir na frente, remover da frente: i, j, c

Busca em Profundidade



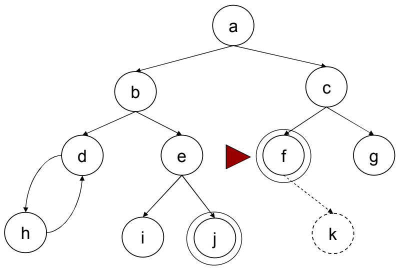
Inserir na frente, remover da frente: j, c

Busca em Profundidade



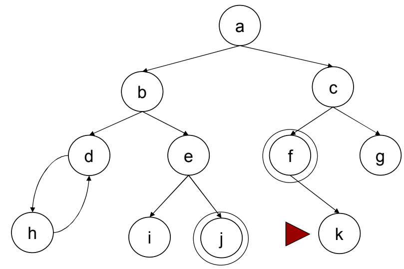
Inserir na frente, remover da frente: c

Busca em Profundidade



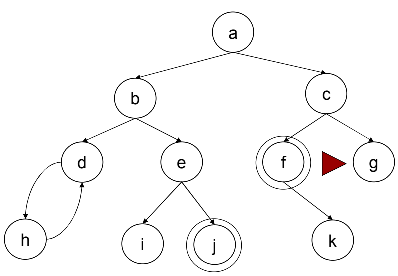
Inserir na frente, remover da frente: f, g

Busca em Profundidade



Inserir na frente, remover da frente: k, g

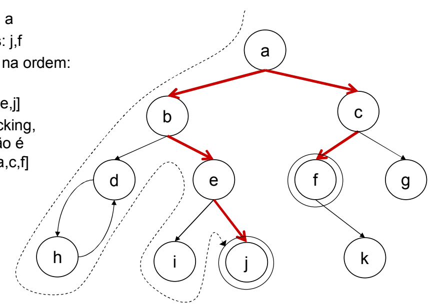
Busca em Profundidade



Inserir na frente, remover da frente: g

Busca em Profundidade

- Estado inicial: a
- Estados finais: j,f
- Nós visitados na ordem: a,b,d,h,e,i,j
- Solução: [a,b,e,j]
- Após backtracking, a outra solução é encontrada: [a,c,f]



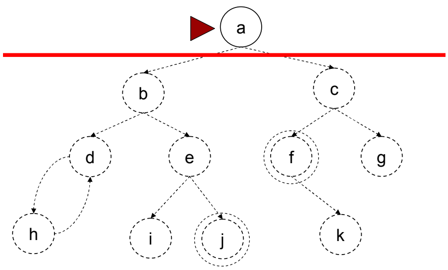
Busca em Profundidade

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

% depthFirst(Caminho,No,Solucao) estende o caminho [No|Caminho]
% até um nó final obtendo Solucao
depthFirst(Caminho,No,[No|Caminho]) :-
  final(No).
depthFirst(Caminho,No,S) :-
  s(No,No1),
  \+ pertence(No1,Caminho),            % evita um ciclo
  depthFirst([No|Caminho],No1,S).

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


Busca em Profundidade

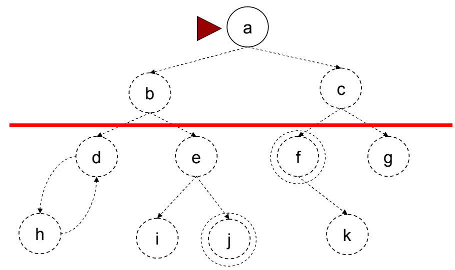
- Um problema com a busca em profundidade é que existem espaços de estado nos quais o algoritmo se perde
- Muitos espaços de estado são infinitos e, nesse caso, o algoritmo de busca em profundidade pode perder um nó final, prosseguindo por um caminho infinito no grafo
- O algoritmo então explora esta parte infinita do espaço, nunca chegando perto de um nó final
- Por exemplo, o problema das 8 Rainhas é susceptível 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 (sem ciclos), um refinamento pode ser adicionado à busca em profundidade: limitar a profundidade de busca

Busca em Profundidade Limitada (L=0)



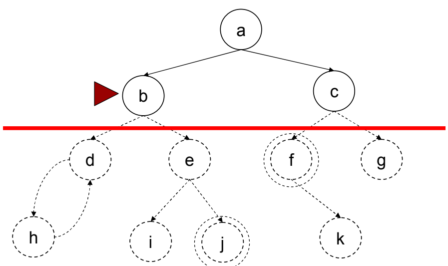
Inserir na frente, remover da frente: a

Busca em Profundidade Limitada (L=1)



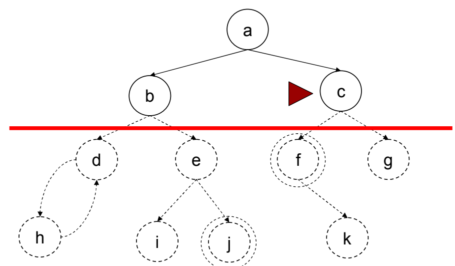
Inserir na frente, remover da frente: a

Busca em Profundidade Limitada (L=1)



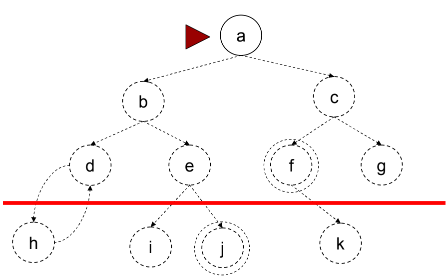
Inserir na frente, remover da frente: b, c

Busca em Profundidade Limitada (L=1)



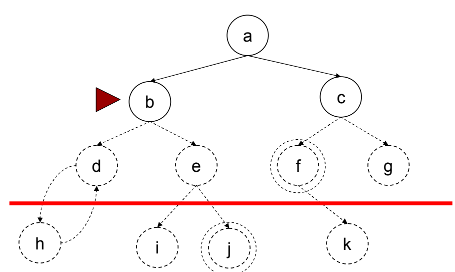
Inserir na frente, remover da frente: c

Busca em Profundidade Limitada (L=2)



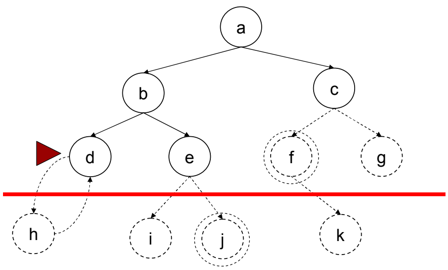
Inserir na frente, remover da frente: a

Busca em Profundidade Limitada (L=2)



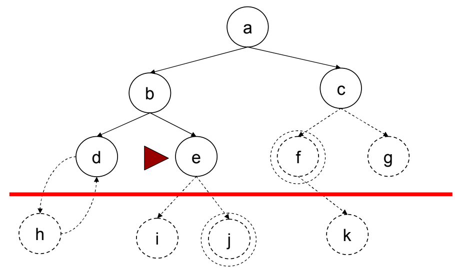
Inserir na frente, remover da frente: b, c

Busca em Profundidade Limitada (L=2)



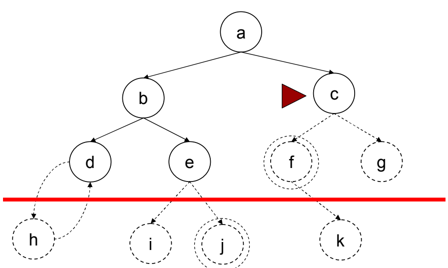
Inserir na frente, remover da frente: d, e, c

Busca em Profundidade Limitada (L=2)



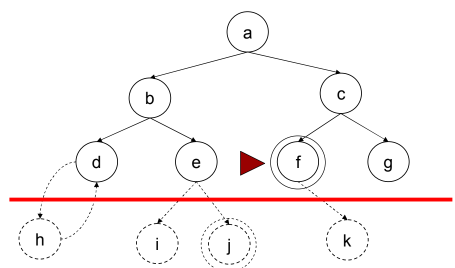
Inserir na frente, remover da frente: e, c

Busca em Profundidade Limitada (L=2)



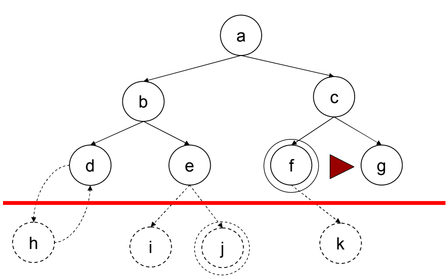
Inserir na frente, remover da frente: c

Busca em Profundidade Limitada (L=2)



Inserir na frente, remover da frente: f, g

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
% (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,S,L) :-
  L > 0,
  s(No,No1),
  \+ pertence(No1,Caminho),          % evita um ciclo
  L1 is L - 1,
  depthFirstLimited([No|Caminho],No1,S,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 e pode ser implementada chamando o procedimento depthFirstLimited/4 a partir de outro procedimento que, em cada chamada recursiva, incrementa o limite em uma unidade

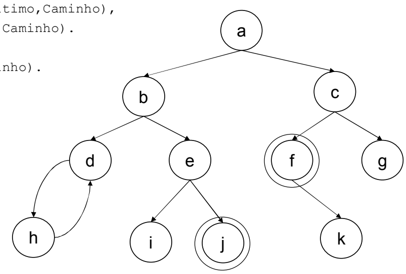
Busca em Profundidade Iterativa

- Entretanto, há uma implementação mais elegante baseado no procedimento path(No1,No2,Caminho) onde Caminho é um caminho acíclico (na ordem reversa) entre os nós No1 e No2 no espaço de estados

```
path(No,No,[No]).                        % caminho com um único nó
path(Primeiro,Ultimo,[Ultimo|Caminho]) :-
  s(Penultimo,Ultimo),                   % Há nó anterior ao último
  path(Primeiro,Penultimo,Caminho),      % Há caminho até penúltimo
  \+ pertence(Ultimo,Caminho).           % evita um ciclo
```


Busca em Profundidade Iterativa

```
path(No,No,[No]).
path(Primeiro,Ultimo,[Ultimo|Caminho]) :-
  s(Penultimo,Ultimo),
  path(Primeiro,Penultimo,Caminho),
  \+ pertence(Ultimo,Caminho).

?- path(a,Ultimo,Caminho).
Ultimo = a
Caminho = [a];
Ultimo = b
Caminho = [b,a];
Ultimo = c
Caminho = [c,a];
Ultimo = d
Caminho = [d,b,a];
...
```



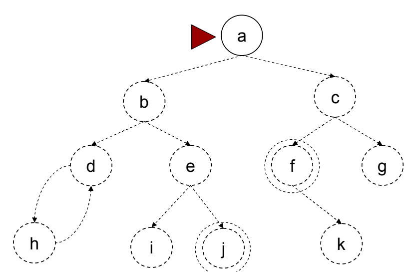
Busca em Profundidade Iterativa

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

% path(No1,No2,Caminho) encontra Caminho acíclico entre No1 e No2
path(No,No,[No]).                        % caminho com um único nó
path(Primeiro,Ultimo,[Ultimo|Caminho]) :-
  s(Penultimo,Ultimo),                   % Há nó anterior ao último
  path(Primeiro,Penultimo,Caminho),      % Há caminho até penúltimo
  \+ pertence(Ultimo,Caminho).           % evita um ciclo

% depthFirstIterativeDeepening(No,Solução) iterativamente
% aumente a profundidade do caminho
depthFirstIterativeDeepening(No,Solucao) :-
  path(No,Final,Solucao),
  final(Final).
```


Busca em Largura

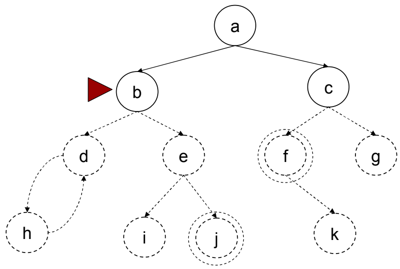
- Em contraste com a busca em profundidade, a busca em largura escolhe primeiro visitar aqueles nós mais próximos do nó inicial
- O algoritmo não é tão simples, pois é necessário manter um **conjunto** de nós candidatos alternativos e não apenas um único, como na busca em profundidade
- O conjunto é todo o nível inferior da árvore de busca
- Além disso, só o conjunto é insuficiente 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

Busca em Largura



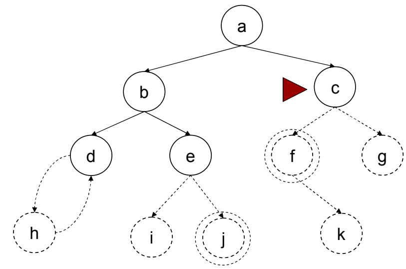
Inserir no final, remover da frente: a

Busca em Largura



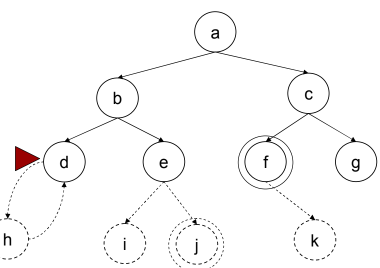
Inserir no final, remover da frente: b, c

Busca em Largura



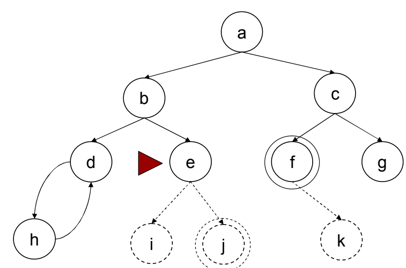
Inserir no final, remover da frente: c, d, e

Busca em Largura



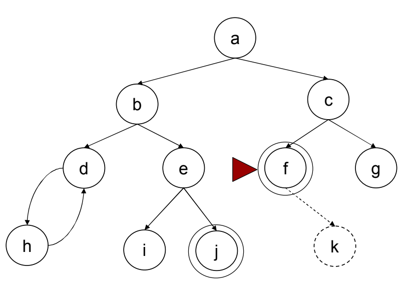
Inserir no final, remover da frente: d, e, f, g

Busca em Largura



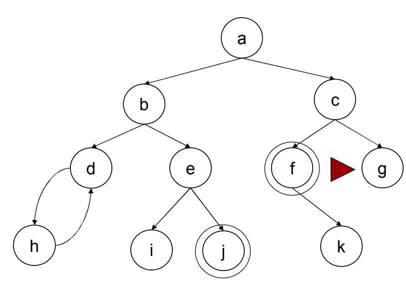
Inserir no final, remover da frente: e, f, g, h

Busca em Largura



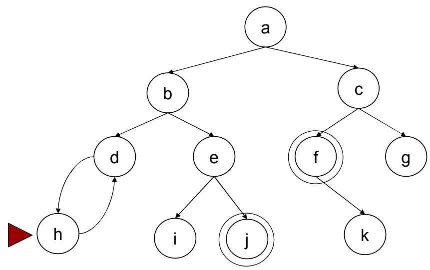
Inserir no final, remover da frente: f, g, h, i, j

Busca em Largura



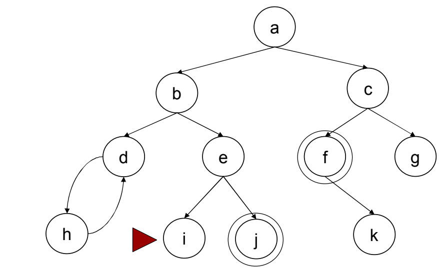
Inserir no final, remover da frente: g, h, i, j, k

Busca em Largura



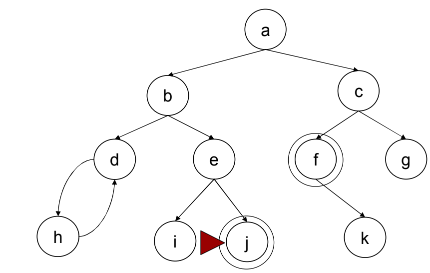
Inserir no final, remover da frente: h, i, j, k

Busca em Largura



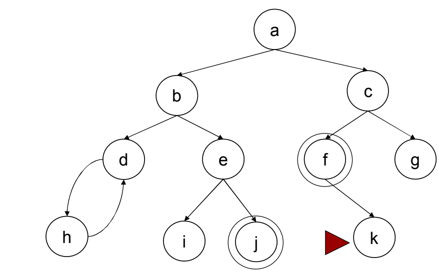
Inserir no final, remover da frente: i, j, k

Busca em Largura



Inserir no final, remover da frente: j, k

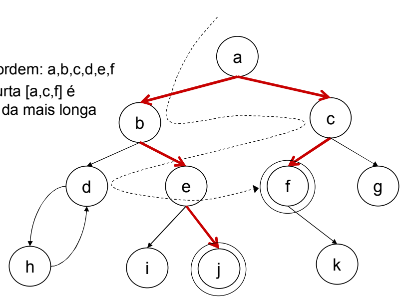
Busca em Largura



Inserir no final, remover da frente: k

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]

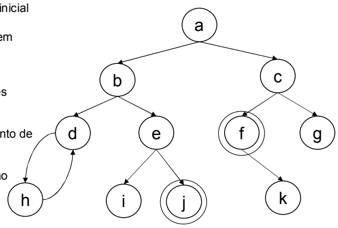


Busca em Largura

- **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

Busca em Largura

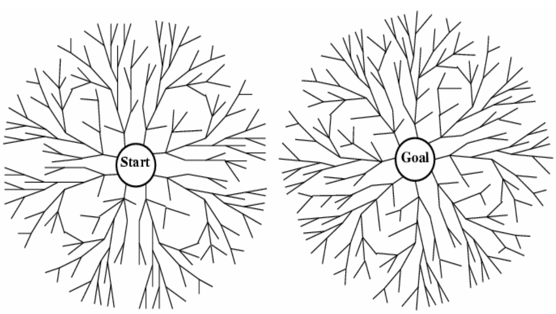
- Comece com o conjunto de candidatos inicial
  - [[a]]
- Gere extensões de [a] (note que estão em ordem reversa):
  - [[b,a], [c,a]]
- Remova o primeiro candidato [b,a] do conjunto de candidatos e gere extensões deste caminho
  - [[d,b,a], [e,b,a]]
- Adicione as extensões ao final do conjunto de candidatos
  - [[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



Busca em Largura

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

% breadthFirst([Caminho1,Caminhos2,...],Solucao) Solucao é uma
% extensão para um nó final de um dos caminhos
breadthFirst([[No|Caminho]|_],[No|Caminho]) :-
  final(No).
breadthFirst([Caminho|Caminhos],Solucao) :-
  estender(Caminho,NovosCaminhos),
  concatenar(Caminhos,NovosCaminhos,Caminhos1),
  breadthFirst(Caminhos1,Solucao).

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


Busca Bidirecional



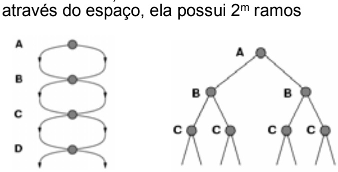
Complexidade dos Algoritmos de Busca

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

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

Evitando Estados Repetidos

- Em alguns problemas, existe a possibilidade de expandir estados que já foram expandidos antes em algum outro caminho
- As árvores de busca destes problemas são infinitas, mas se alguns dos estados repetidos são podados, a árvore se torna finita
- Mesmo quando a árvore é finita, evitar estados repetidos pode resultar numa redução exponencial do custo da busca
- No exemplo abaixo, o espaço contém apenas m+1 estados, onde m é a profundidade máxima; como a árvore de busca cada caminho possível através do espaço, ela possui 2^m ramos



Evitando Estados Repetidos

- Há três formas de tratar estados repetidos:
  - Não retornar ao estado do qual se acabou de sair; por exemplo, a função sucessor pode se recusar a gerar qualquer sucessor que é o mesmo estado que o nó pai (nó anterior)
  - Não criar caminhos com ciclos
  - Não gerar nenhum estado que foi gerado anteriormente; isto requer que cada estado gerado seja guardado em memória resultando potencialmente em complexidade de espaço de $O(b^d)$; utiliza-se normalmente uma tabela hash para armazenar todos os estados gerados

Busca Informada

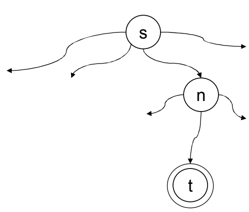
- A busca em grafos pode atingir uma complexidade elevada devido ao número de alternativas
- Estratégias de busca informada utilizam informação heurística sobre o problema para calcular estimativas heurísticas para os nós no espaço de estados
- Essa estimativa indica o quanto o nó é promissor com relação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

Estratégia Best-First

- É conveniente relembrar que uma estratégia de busca é definida por meio da ordem de expansão dos nós
- Na estratégia de busca *best-first* (começando pelo melhor), 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
- **Heurística** (arte de descobrir) consiste em conhecimentos que permitem uma solução rápida para algum problema ou dificuldade

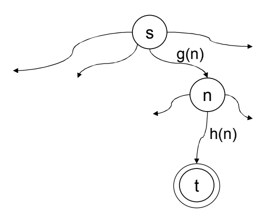
Estratégia Best-First

- Vamos assumir que há um custo envolvido entre cada arco:
  - **s(X,Y,C)** que é verdadeira se há um movimento permitido no espaço de estados do nó X para o nó Y ao 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)** estima a dificuldade de **n**, ou seja, **f(n)** é o custo do caminho mais barato de **s** até **t** via **n**



Estratégia Best-First

- 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**



Estratégia Best-First

- 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**

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: função de avaliação é vista como custo

Hill-Climbing

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

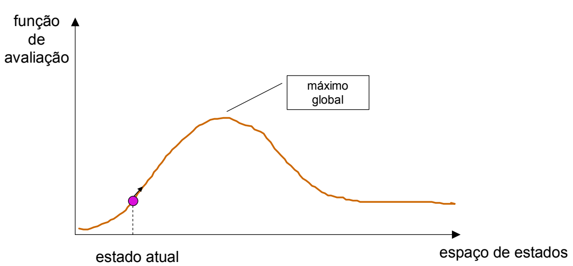
- Se há mais de um vizinho com a melhor qualidade:
  - Escolher o primeiro melhor
  - Escolher um entre todos de forma aleatória

Hill-Climbing

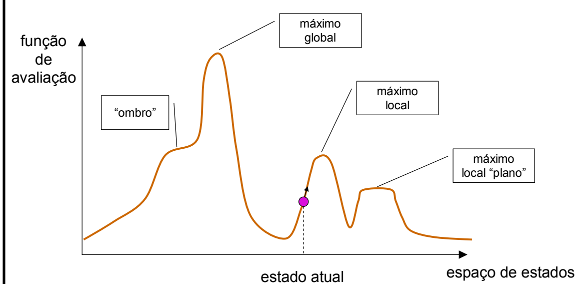
```
% resolva(No,Solucao) Solucao é um caminho
% acíclico (na ordem reversa) entre nó
% inicial No e uma solução
% l(N,F/G) denota o nó N com valores F=f(n) e
% G = g(N)

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.
```

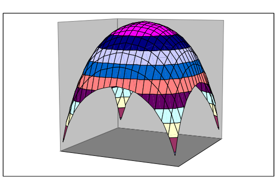

Hill-Climbing

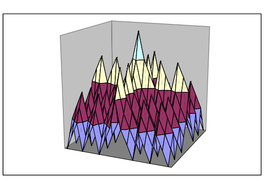


Hill-Climbing



Hill-Climbing





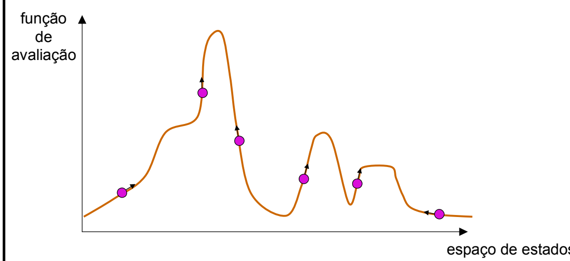
Hill-Climbing: Problemas

- 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

Hill-Climbing: Variações

- Hill-Climbing Estocástico
  - Nem sempre escolha o melhor vizinho
- Hill-Climbing Primeira Escolha
  - Escolha o primeiro bom vizinho que encontrar
    - Útil se é grande o número de sucessores de um nó
- Hill-Climbing Reinício Aleatório
  - Conduz uma série de buscas hill-climbing a partir de estados iniciais gerados aleatoriamente, executando cada busca até terminar ou até que não exista progresso significativo
  - O melhor resultado de todas as buscas é armazenado

Hill-Climbing Reinício Aleatório



Hill-Climbing: Variações

- Têmpera Simulada (Simulated Annealing)
  - Termo utilizado em metalurgia
  - Não é estratégia best-first mas é uma derivação
  - O objetivo é que as moléculas de metal encontrem uma localização estável em relação aos seus vizinhos
  - O aquecimento provoca movimento das moléculas de metal para localizações indesejáveis
  - Durante o resfriamento, as moléculas reduzem seus movimentos e situam-se em uma localização mais estável
  - Têmpera é o processo de aquecer um metal e deixá-lo esfriar lentamente de forma que as moléculas fiquem em localizações estáveis

Têmpera Simulada

1. Escolha um estado inicial do espaço de busca de forma aleatória
2. i ← 1
3. T ← Temperatura(i)
4. Enquanto (T > T_f) Faça
5.   Escolha um vizinho (sucessor) do estado atual de forma aleatória
6.   deltaE ← energia(vizinho) – energia(atual)
7.   Se (deltaE > 0) Então
        o movimento é aceito (mova para o vizinho de melhor qualidade)
    Senão
        o movimento é aceito com probabilidade exp(deltaE/T)
    Fim Se
8.   i ← i + 1
9.   T ← Temperatura(i)
10. Fim Enquanto
11. Retorne o estado atual como sendo a solução

- **energia(N)** é uma função que calcula a energia do estado N e pode ser vista como qualidade
- **Temperatura(i)** é uma função que calcula a temperatura na iteração i, assumindo sempre valores positivos
- T_f é a temperatura final (por exemplo, T_f = 0)

Têmpera Simulada

- No início qualquer movimento é aceito
- Quando a temperatura é reduzida, probabilidade de aceitar um movimento negativo é reduzida
- Movimentos negativos são as vezes essenciais para escapar de máximos locais
- Movimentos negativos em excesso afastam do máximo global

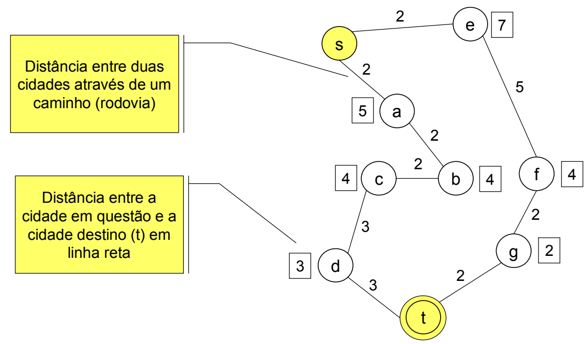
Best-First & A*

- Vamos estudar o algoritmo best-first em sua forma completa
- 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

Best-First & A*

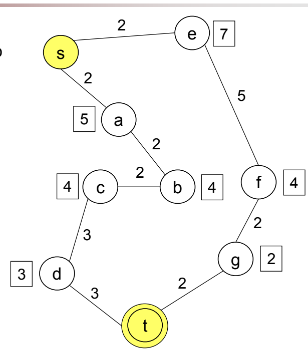
- Podemos imaginar o mecanismo de ativação-desativação da seguinte forma
  - O processo trabalhando na alternativa atual recebe um orçamento limite
  - Ele permanece ativo até que o orçamento seja exaurido
  - Durante o período em que está ativo, o processo continua expandindo sua sub-árvore e relata uma solução caso um nó final seja encontrado
  - O orçamento limite para essa execução é definido pela estimativa heurística da alternativa competidora mais próxima

Best-First & A*



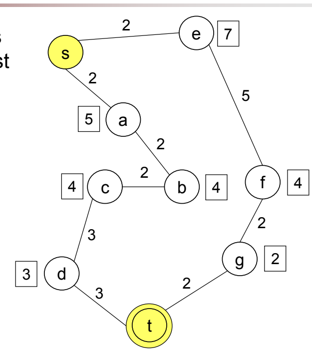
Best-First & A*

- 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)



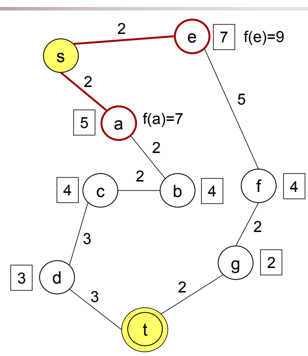
Best-First & A*

- 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**



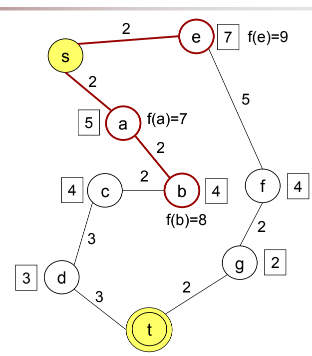
Best-First & A*

- 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



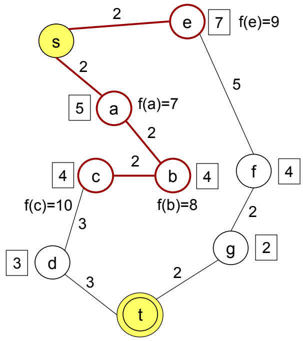
Best-First & A*

- 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



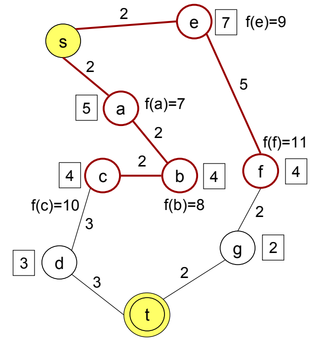
Best-First & A*

- 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**



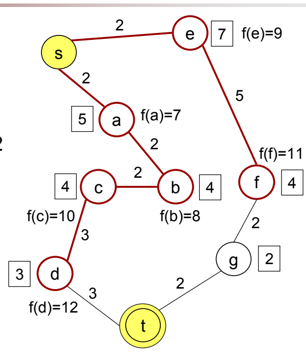
Best-First & A*

- 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



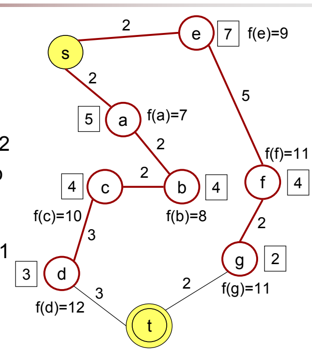
Best-First & A*

- 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



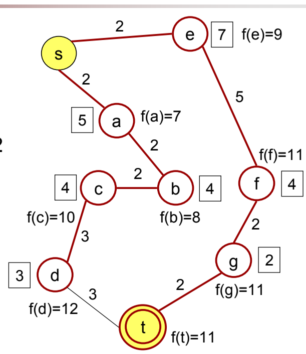
Best-First & A*

- 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



Best-First & A*

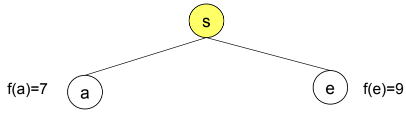
- f(f)=g(f)+dist(f,t)=7+4=11
- Como f(f)>f(c) agora o processo 2 espera e o processo 1 prossegue
- f(d)=g(d)+dist(d,t)=9+3=12
- Como f(d)>f(f) o processo 2 reinicia chegando até o destino **t**
- f(g)=g(g)+dist(g,t)=9+2=11
- f(t)=g(t)+dist(t,t)=11+0=11



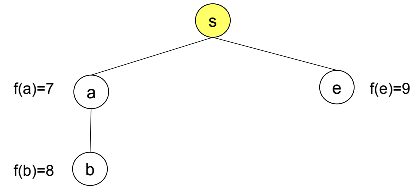
Best-First & A*

- 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

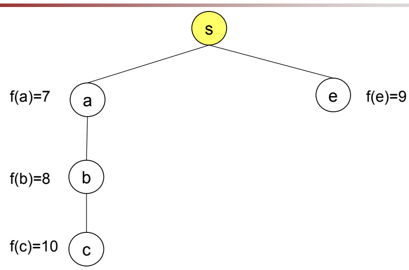
Best-First & A*



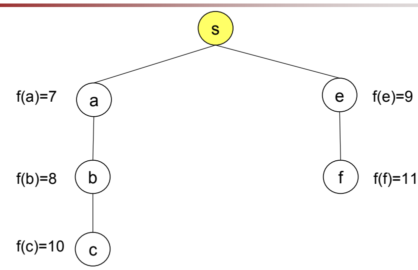
Best-First & A*



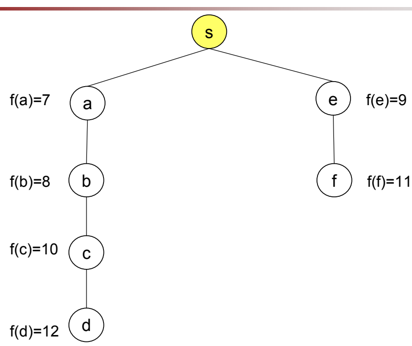
Best-First & A*



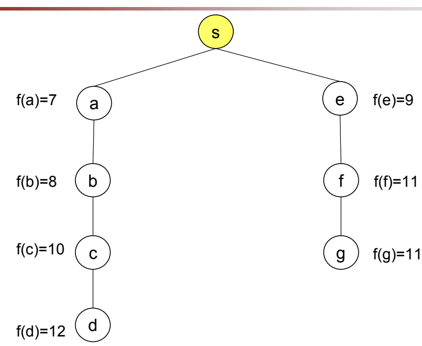
Best-First & A*



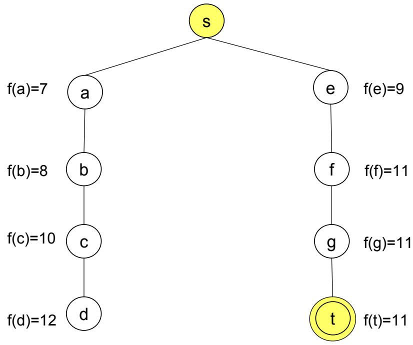
Best-First & A*



Best-First & A*



Best-First & A*

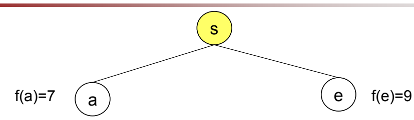


Best-First & A*

- A árvore de busca será representada de duas formas:
  - l(N,F/G) representa um único nó folha (*leaf*)
    - N é um nó do espaço de estados
    - G é g(N), custo do caminho encontrado desde o nó inicial até N
    - F é f(N) = G + h(N)
  - t(N,F/G,Subs) representa uma árvore com sub-árvores não vazias
    - N é a raiz da árvore
    - Subs é uma lista de suas sub-árvores (em ordem crescente de valores-f das sub-árvores)
    - G é g(N)
    - F é o valor-f atualizado de N, ou seja, o valor-f do sucessor mais promissor de N

Best-First & A*

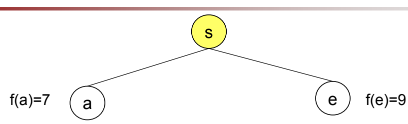
Best-First & A*



t(s,7/0, [l(a,7/2), l(e,9/2)])

O valor-f da raiz s é f(s)=7 pois é o valor-f do sucessor mais promissor de **s**

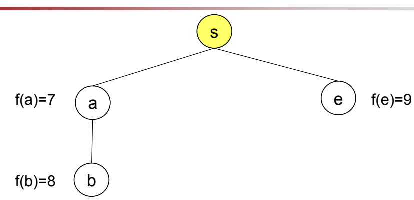
Best-First & A*



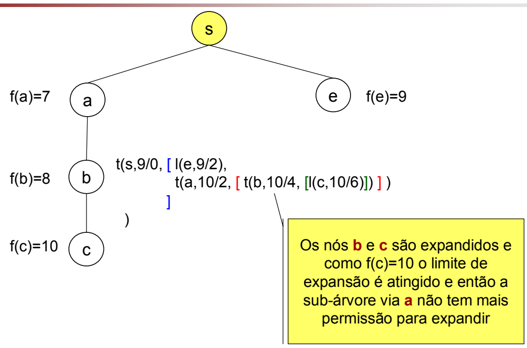
t(s,7/0, [l(a,7/2), l(e,9/2)])

O competidor mais próximo de **a** é **e**, com f(e)=9
Portanto, **a** é permitido expandir enquanto f(a) não exceder 9

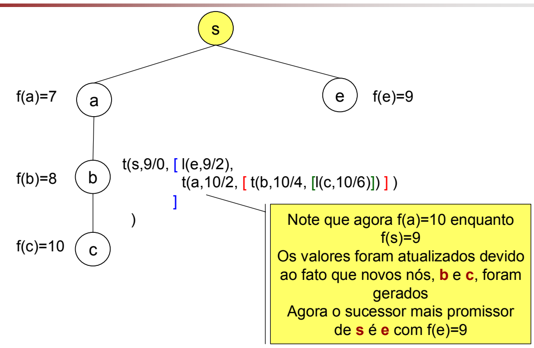
Best-First & A*



Best-First & A*



Best-First & A*



Best-First & A*

- A atualização dos valores-f é necessário para permitir o programa reconhecer a sub-árvore mais promissora em cada nível da árvore de busca (a árvore que contém o nó mais promissor)
- Este atualização leva a uma generalização da definição da função *f* de nós para árvores

Best-First & A*

- Para um único nó (folha) **n**, temos a definição original
  - $f(n) = g(n) + h(n)$
- Para uma árvore T, cuja raiz é **n** e as sub-árvores de **n** são S_1, S_2, ..., S_k
  - $f(T) = \min f(S_i) \qquad 1 <= i <= k$

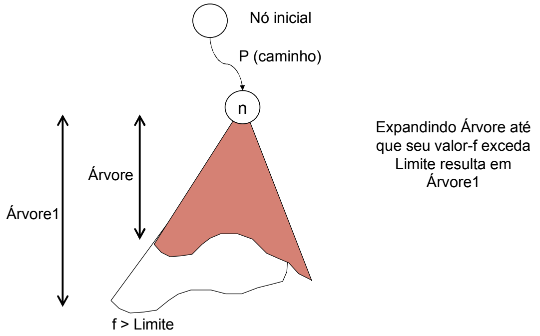
Best-First & A*

- O predicado principal é **expandir(P,Árvore,Limite,Árvore1,Resolvido,Solução)**
- Este predicado expande uma (sub)árvore atual enquanto o valor-f dela permaneça inferior ou igual à Limite
- Argumentos:
  - **P**: caminho entre o nó inicial e **Árvore**
  - **Árvore**: atual (sub)árvore
  - **Limite**: valor-f limite para expandir **Árvore**
  - **Árvore1**: **Árvore** expandida dentro de **Limite**; assim o valor-f de **Árvore1** é maior que **Limite** (a menos que um nó final tenha sido encontrado durante a expansão)
  - **Resolvido**: Indicador que assume 'sim', 'não' ou 'nunca'
  - **Solução**: Um caminho (solução) do nó inicial através de **Árvore1** até um nó final dentro de **Limite** (se existir tal nó)

Best-First & A*

- Os argumentos de entrada são **P**, **Árvore** e **Limite**
- **expandir/6** produz três tipos de resultados, indicados pelo valor do argumento **Resolvido**
  1. Resolvido = sim
     Solução = uma solução encontrada expandindo Árvore dentro de Limite
     Árvore1 = não instanciada
  2. Resolvido = não
     Árvore1 = Árvore expandida de forma que seu valor-f exceda Limite (vide slide seguinte)
     Solução = não instanciada
  3. Resolvido = nunca
     Árvore1 e Solução = não instanciadas
- O último caso indica que Árvore é uma alternativa inviável e nunca deve ter outra chance de crescer; isto ocorre quando o valor-f de **Árvore** <= **Limite** mas as árvore não pode crescer porque nenhuma folha dela possui sucessor ou o sucessor existente criaria um ciclo

A Relação expandir/6



Algoritmo Best-First

```
% Assuma que 9999 é maior que qualquer valor-f
resolva(No,Solucao) :-
  expandir([],l(No,0/0),9999,_,sim,Solucao).

% expandir(P,Arvore,Limite,Arvore1,Resolvido,Solucao)
% P é um caminho entre nó inicial da busca e subárvore Arvore, Arvore1 é Arvore
% expandida até Limite. Se um nó final é encontrado então Solucao é a solução e
% Resolvido = sim

% Caso 1: nó folha final, construir caminho da solução
expandir(P,l(N,_),_,_,sim,[N|P]) :-
  final(N).

% Caso 2: nó folha, valor-f <= Limite. Gerar sucessores e expandir dentro de Limite
expandir(P,l(N,F/G),Limite,Arvore1,Resolvido,Solucao) :-
  F =< Limite,
  (findall(M/Custo, (s(N,M,Custo), \+ pertence(M,P)),Vizinhos),
   Vizinhos \= [],
   !,                            % nó N tem sucessores
   avalie(G,Vizinhos,Ts),        % crie subárvores
   melhorf(Ts,F1),               % valor-f do melhor sucessor
   expandir(P,t(N,F1/G,Ts),Limite,Arvore1,Resolvido,Solucao)
   ;
   Resolvido = nunca             % N não tem sucessores - beco sem saída
  ).
```


Algoritmo Best-First (cont.)

```
% Caso 3: não-folha, valor-f <= Limite. Expanda a subárvore mais
% promissora; dependendo dos resultados, o predicado continue
% decide como proceder
expandir(P,t(N,F/G,[T|Ts]),Limite,Arvore1,Resolvido,Solucao) :-
  F =< Limite,
  melhorf(Ts,MF),
  min(Limite,MF,Limite1),   % Limite1 = min(Limite,MF)
  expandir([N|P],T,Limite1,T1,Resolvido1,Solucao),
  continue(P,t(N,F/G,[T1|Ts]),Limite,Arvore1,Resolvido1,Resolvido,Solucao).

% Caso 4: não-folha com subárvores vazias
% Beco sem saída que nunca será resolvido
expandir(_,t(_,_,[]),_,_,nunca,_) :- !.

% Caso 5: valor f > Limite, árvore não pode crescer
expandir(_,Arvore,Limite,Arvore,nao,_) :-
  f(Arvore,F),
  F > Limite.
```


Algoritmo Best-First (cont.)

```
% continue(Caminho,Arvore,Limite,NovaArvore,SubarvoreResolvida,ArvoreResolvida,Solucao)
continue(_,_,_,_,sim,sim,_).   % solução encontrada
% Limite ultrapassado, procurar outra subárvore para expandir
continue(P,t(N,F/G,[T1|Ts]),Limite,Arvore1,nao,Resolvido,Solucao) :-
  inserir(T1,Ts,NTs),
  melhorf(NTs,F1),
  expandir(P,t(N,F1/G,NTs),Limite,Arvore1,Resolvido,Solucao).
% abandonar T1 pois é beco sem saída
continue(P,t(N,F/G,[T1|Ts]),Limite,Arvore1,nunca,Resolvido,Solucao) :-
  melhorf(Ts,F1),
  expandir(P,t(N,F1/G,Ts),Limite,Arvore1,Resolvido,Solucao).

% avalie(G0,[No1/Custo1,...],[l(MelhorNo,MelhorF/G,...])
% ordena a lista de folhas pelos seus valores-f
avalie(_,[],[]).
avalie(G0,[N/C|NaoAvaliados],Ts) :-
  G is G0 + C,
  h(N,H),
  F is G + H,
  avalie(G0,NaoAvaliados,Avaliados),
  inserir(l(N,F/G),Avaliados,Ts).
```


Algoritmo Best-First (cont.)

```
% insere T na lista de árvore Ts mantendo a ordem dos valores-f
inserir(T,Ts,[T|Ts]) :-
  f(T,F),
  melhorf(Ts,F1),
  F =< F1, !.
inserir(T,[T1|Ts],[T1|Ts1]) :-
  inserir(T,Ts,Ts1).

% Obter o valor f
f(l(_,F/_),F).          % valor-f de uma folha
f(t(_,F/_,_),F).        % valor-f de uma árvore

melhorf([T|_],F) :-     % melhor valor-f de uma lista de árvores
  f(T,F).
melhorf([],9999).       % Nenhuma árvore: definir valor-f ruim

min(X,Y,X) :-
  X =< Y, !.
min(X,Y,Y).

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


Best-First & A*

- O algoritmo apresentado é uma variação do algoritmo conhecido com A*
- Um algoritmo de busca é chamado de **admissível** se ele sempre produz uma solução ótima (caminho de custo mínimo), assumindo que uma solução exista
- A implementação apresentada, que produz todas as soluções através de *backtracking* e pode ser considerada admissível se a primeira solução encontrada é ótima
- Para cada nó **n** no espaço de estados vamos denotar **h*(n)** como sendo o custo de um caminho ótimo de **n** até um nó final
- Um teorema sobre a admissibilidade de A* diz que um algoritmo A* que utiliza uma função heurística **h** tal que para todos os nós no espaço de estados h(n) <= h*(n) é admissível

Best-First & A*

- Este resultado tem grande valor prático
- Mesmo que não conheçamos o exato valor de h*, nós só precisamos achar um limite inferior para h* e utilizá-la como h em A*
- Isto é suficiente para garantir que A* irá encontrar uma solução ótima

Best-First & A*

- Há um limite inferior trivial
  - h(n) = 0 para todo **n** no espaço de estados
- Embora este limite trivial garanta admissibilidade sua desvantagem é que não há nenhuma heurística e assim não há como fornecer nenhum auxílio para a busca, resultando em alta complexidade
- A* usando h=0 comporta-se de forma similar à busca em largura
- De fato, A* se comporta exatamente igual à busca em largura se todos os arcos entre nós têm custo unitário, ou seja, s(X,Y,1)

Best-First & A*

- Portanto é interessante utilizar h>0 para garantir admissibilidade e h o mais próximo possível de h* (h<=h*) para garantir eficiência
- Se múltiplas heurísticas estão disponíveis:
  - $h(n) = \max\{h_1(n), h_2(n), \ldots, h_m(n)\}$
- De maneira ideal, se h* é conhecida, podemos utilizar h* diretamente
  - A* utilizando h* encontra uma solução ótima diretamente, sem nunca precisar realizar *backtracking*

Complexidade de A*

- A utilização de heurística para guiar o algoritmo best-first reduz a busca a apenas uma região do espaço do problema
- Apesar da redução no esforço da busca, a ordem de complexidade é ainda exponencial na profundidade de busca
  - Isso é válido para tempo e memória uma vez que o algoritmo mantém todos os nós gerados
- Em situações práticas o espaço de memória é mais crítico e A* pode utilizar toda a memória disponível em questão de minutos

Complexidade de A*

- Algumas variações de A* foram desenvolvidas para utilizar menos memória, penalizando o tempo
  - A idéia básica é similar à busca em profundidade iterativa
  - O espaço necessário reduz de exponencial para linear na profundidade de busca
  - O preço é a re-expansão de nós já expandidos no espaço de busca
- Veremos duas dessas técnicas:
  - IDA* (Iterative Deepening A*)
  - RBFS (Recursive Best-First Search)

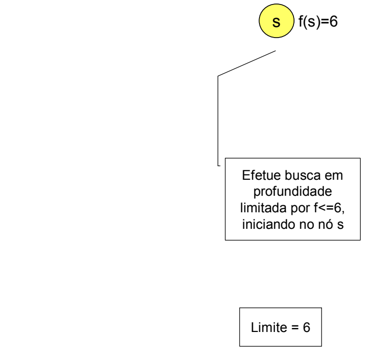
IDA*

- IDA* é similar à busca em profundidade iterativa
  - Na busca em profundidade iterativa as buscas em profundidade são realizadas em limites crescentes de profundidade; em cada iteração a busca em profundidade é limitada pelo limite de profundidade atual
  - Em IDA* as buscas em profundidade são limitadas pelo limite atual representando valores-f dos nós

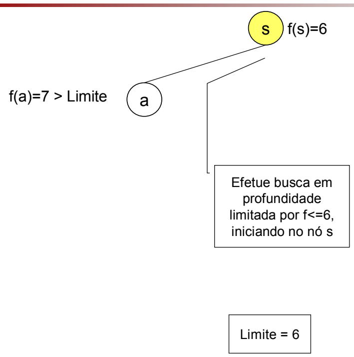
IDA*

```
procedure idastar(Inicio,Solucao)

Limite ← f(Inicio)
repeat
  Iniciando no nó Início, realize busca em
  profundidade sujeita à condição que um nó N é
  expandido apenas se f(N) <= Limite
  if busca em profundidade encontrou nó final then
    indique 'Solução encontrada'
  else
    Calcule NovoLimite como o mínimo valor-f dos nós
    alcançados ao ultrapassar Limite, ou seja,
    NovoLimite ← min{f(N): N gerado pela busca e f(N) > Limite}
  endif
  Limite ← NovoLimite
until Solução encontrada
```

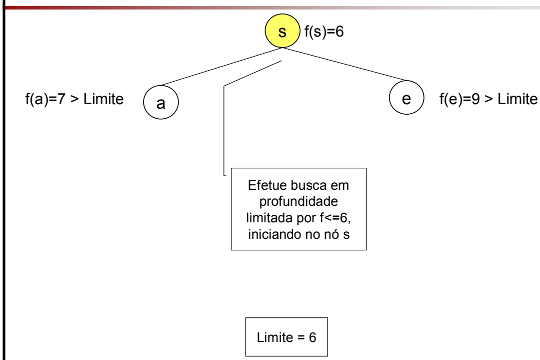

IDA*



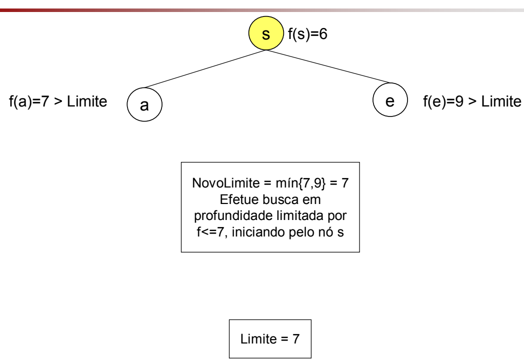
IDA*



IDA*

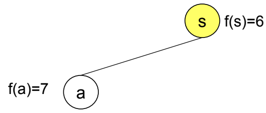


IDA*



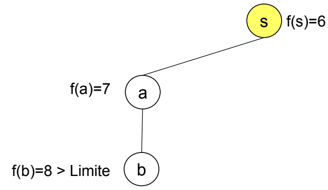
IDA*

IDA*



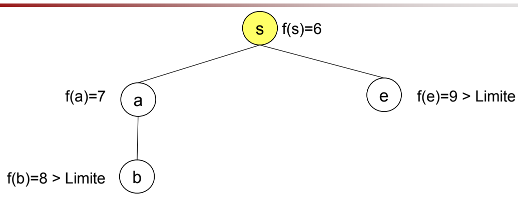
Limite = 7

IDA*



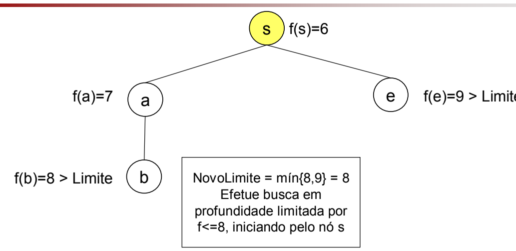
Limite = 7

IDA*



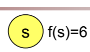
Limite = 7

IDA*



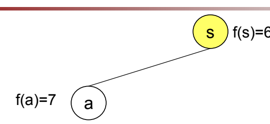
Limite = 8

IDA*



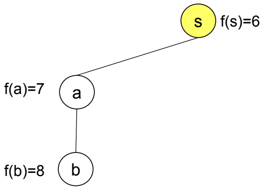
Limite = 8

IDA*



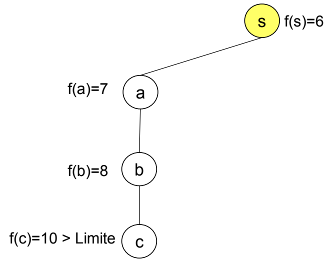
Limite = 8

IDA*



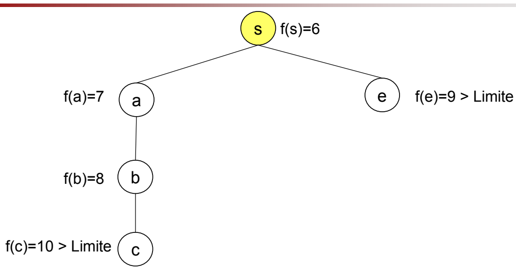
Limite = 8

IDA*



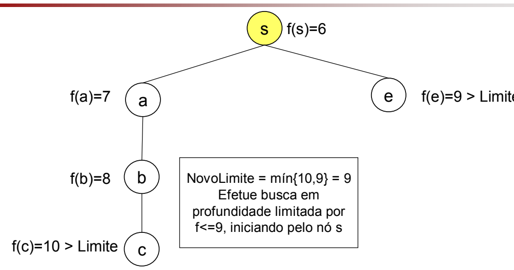
Limite = 8

IDA*



Limite = 8

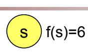
IDA*



NovoLimite = mín{10,9} = 9
Efetue busca em profundidade limitada por f<=9, iniciando pelo nó s

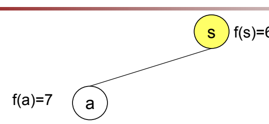
Limite = 9

IDA*



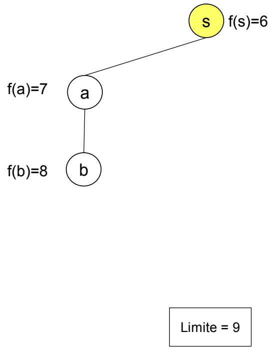
Limite = 9

IDA*

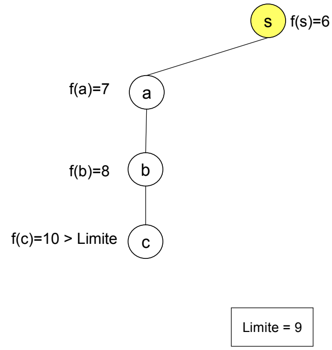


Limite = 9

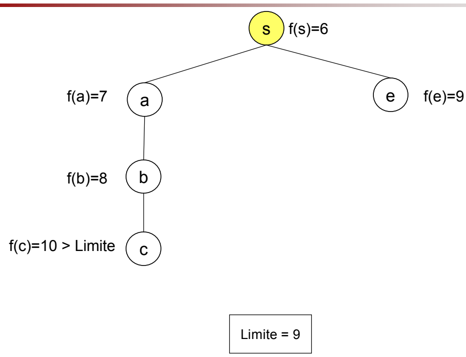
IDA*



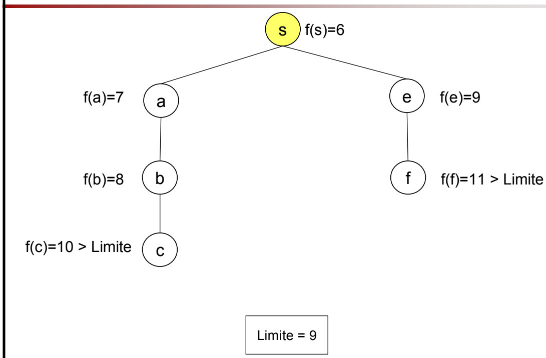
IDA*



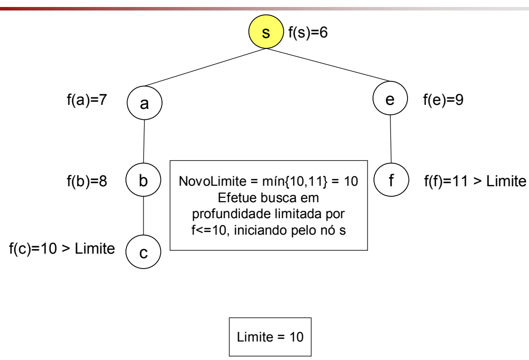
IDA*



IDA*

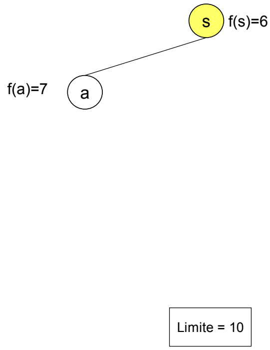


IDA*

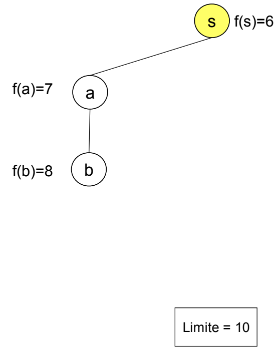


IDA*

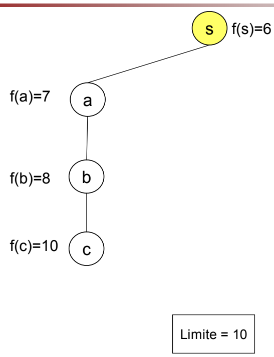
IDA*



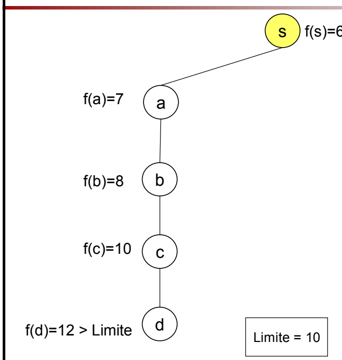
IDA*



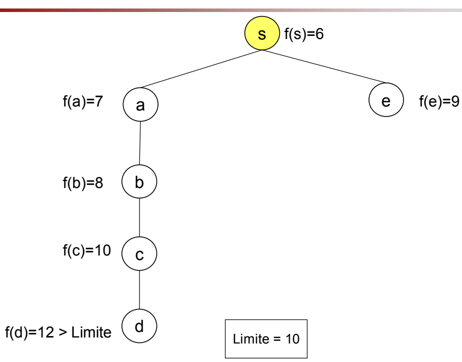
IDA*



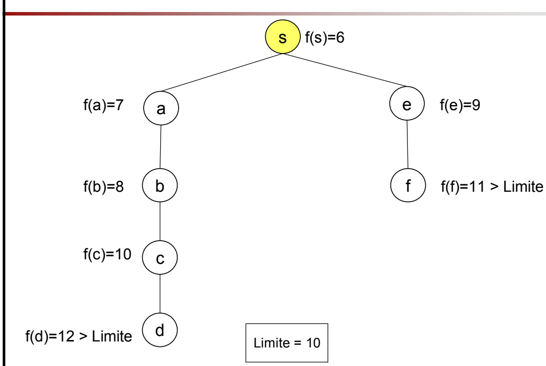
IDA*



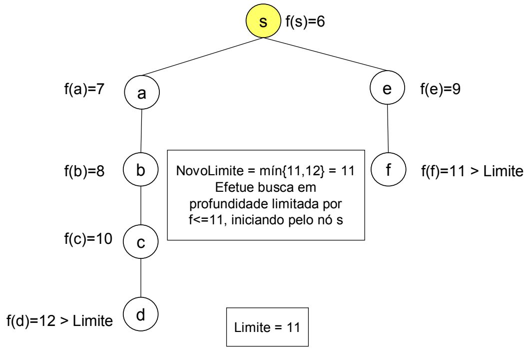
IDA*



IDA*

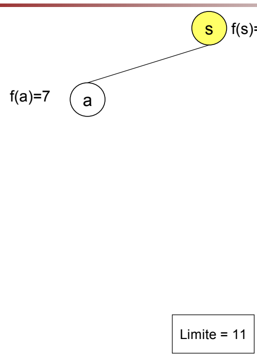


IDA*

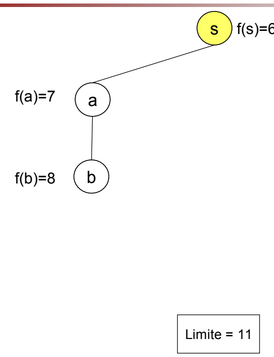


IDA*

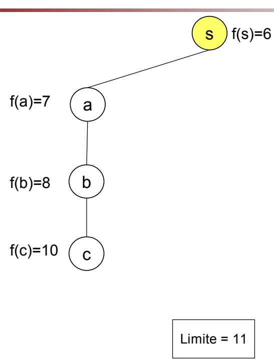
IDA*



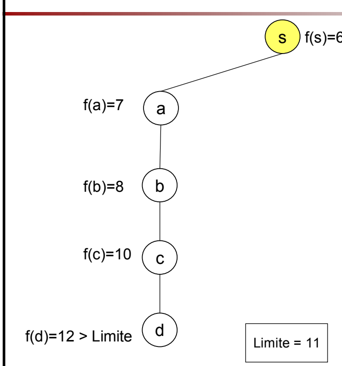
IDA*



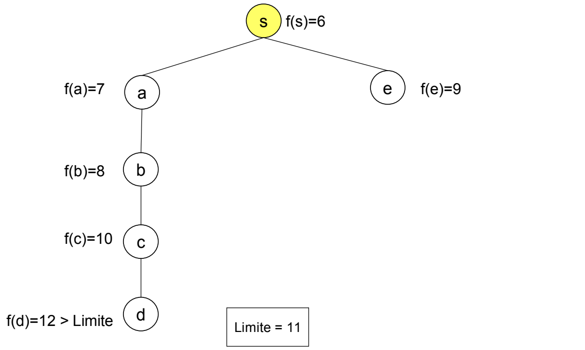
IDA*



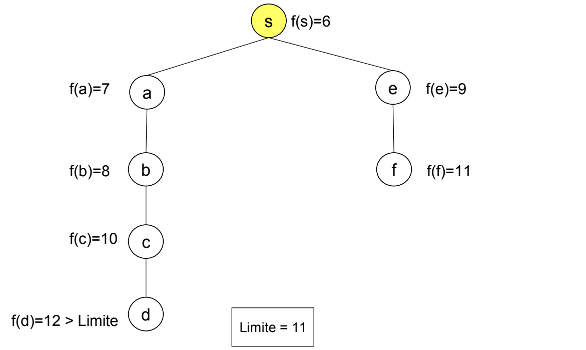
IDA*



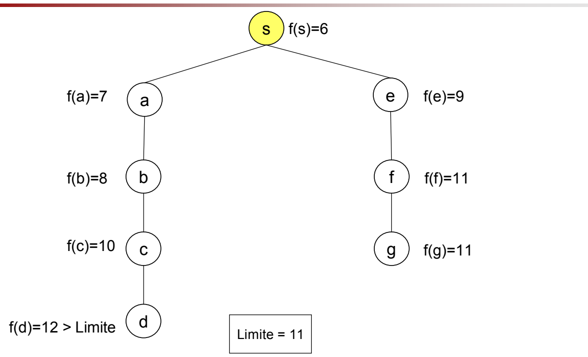
IDA*



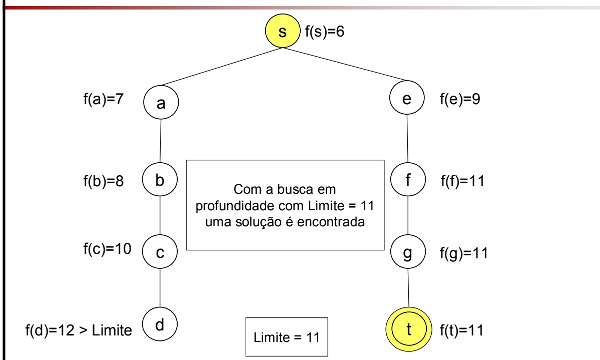
IDA*



IDA*



IDA*



IDA*

```
% Assuma que 9999 é maior que qualquer valor-f
:- dynamic proximo_limite/1,solucao/1.

resolva(No,Solucao) :-
  retract(proximo_limite(_)),  % limpa proximo limite
  fail
  ;
  assert(proximo_limite(0)),  % inicializa proximo limite
  idastar(l(No,0/0),Solucao).

idastar(l(N,F/G),Solucao) :-
  retract(proximo_limite(Limite)),
  assert(proximo_limite(9999)),
  df(l(N,F/G),[N],Limite,Solucao).
idastar(No,Solucao) :-
  proximo_limite(Limite),
  Limite < 9999,
  idastar(No,Solucao).
```


IDA*

```
% df(No,Caminho,Limite,Solucao)
% Realiza busca em profundidade dentro de Limite
% Caminho é um caminho entre nó inicial ate o No atual
% F e' o valor-f do no atual que se encontra no inicio do Caminho

% Caso 1: nó N final dentro de Limite, construir caminho da solucao
df(l(N,F/G),[N|P],Limite,[N|P]) :-
  F =< Limite,
  final(N).

% Caso 2: nó N com valor-f <= Limite
% Gerar sucessor de N e expandir dentro de Limite
df(l(N,F/G),[N|P],Limite,Solucao) :-
  F =< Limite,
  s(N,M,Custo),
  \+ pertence(M,P),
  Gm is G + Custo,    % avaliar no' M
  h(M,Hm),
  Fm is Gm + Hm,
  df(l(M,Fm/Gm),[M,N|P],Limite,Solucao).
```

IDA*

```
% Caso 3: valor f > Limite, atualizar proximo limite
% e falhar
df(l(N,F/G),_,Limite,_) :-
  F > Limite,
  atualize_proximo_limite(F),
  fail.

atualize_proximo_limite(F) :-
  proximo_limite(Limite),
  Limite =< F, !                          % nao altere proximo limite
  ;
  retract(proximo_limite(Limite)),!,  % diminua proximo limite
  assert(proximo_limite(F)).

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


IDA*

- Uma propriedade interessante de IDA* refere-se à sua admissibilidade
  - Assumindo f(n) = g(n)+h(n), se h é admissível (h(n) <= h*(n)) então é garantido que IDA* encontre uma solução ótima
- Entretanto, não é garantido que IDA* explore os nós mesma ordem que best-first (ou seja, na ordem de valores-f crescentes)
  - Quando f não é da forma f=g+h e f é não monotônica

RBFS

- Vimos que IDA* possui uma implementação simples
- Entretanto, no pior caso, quando os valores-f não são compartilhados entre vários nós então muitos limites sucessivos de valores-f são necessários e a nova busca em profundidade expandirá apenas um novo nó enquanto todos os demais são apenas re-expansões de nós expandidos e esquecidos
- Nessa situação existe uma técnica para economizar espaço denominada RBFS (recursive best-first search)

RBFS

- RBFS é similar a A*, mas enquanto A* mantém em memória todos os nós expandidos, RBFS apenas mantém o caminho atual assim como seus irmãos
- Quando RBFS suspende temporariamente um subprocesso de busca (porque ele deixou de ser o melhor), ela ‘esquece’ a subárvore de busca para economizar espaço
- Assim como IDA*, RBFS é apenas linear na profundidade do espaço de estados em quantidade de memória necessária

RBFS

- O único fato que RBFS armazena sobre a subárvore de busca abandonada é o valor-f atualizado da raiz da subárvore
- Os valores-f são atualizados copiando-se os valores-f de forma similar ao algoritmo A*
- Para distinguir entre os valores-f estáticos e aqueles copiados, usaremos:
  - f(**n**) = valor-f do nó **n** utilizando a função de avaliação (sempre o mesmo valor durante a busca)
  - F(**n**) = valor-f copiado (é alterado durante a busca uma vez que depende dos nós descendentes de **n**)
- F(**n**) é definida como:
  - F(**n**) = f(**n**) se **n** nunca foi expandido durante a busca
  - F(**n**) = mín{F(n_i) : n_i é um filho de **n**}

RBFS

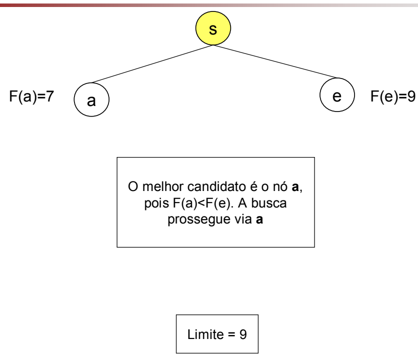
- Assim como A*, RBFS explora subárvores dentro de um limite de valor-f
- O limite é determinado pelos valores-F dos filhos ao longo do caminho atual (o melhor valor-F dos filhos, ou seja, o valor-F do competidor mais promissor do nó atual)
- Seja **n** o melhor nó (aquele com menor valor-F)
  - Então **n** é expandido e seus filhos são explorados até algum limite de valor-f
  - Quando o limite é excedido (F(**n**) > Limite) então todos os nós expandidos a partir de **n** são ‘esquecidos’
  - Entretanto, o valor F(**n**) atualizado é retido e utilizado na decisão em como a busca deve continuar

RBFS

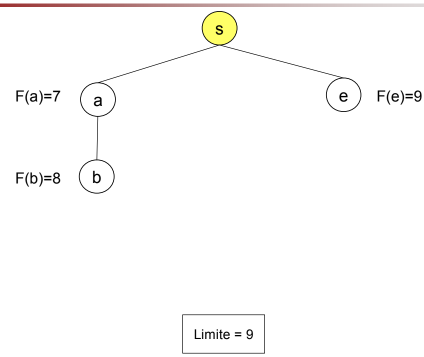
- Os valores-F são determinados não apenas copiando os valores obtidos a partir de um dos filhos mas também são herdados dos nós pais da seguinte forma
- Seja **n** um nó que deve ser expandido pela busca
- Se F(**n**)>f(**n**) então sabemos que **n** deve ter sido expandido anteriormente e que F(**n**) foi determinado a partir dos filhos de **n**, mas os filhos foram removidos da memória
- Suponha que um filho $\mathbf{n_i}$ de **n** seja novamente expandido e o valor estático f($\mathbf{n_i}$) seja calculado novamente
- Então F($\mathbf{n_i}$) é determinado como sendo
  - if f($\mathbf{n_i}$)<F(**n**) then F($\mathbf{n_i}$) $\leftarrow$ F(**n**) else F($\mathbf{n_i}$) $\leftarrow$ f($\mathbf{n_i}$)
- que pode ser escrito como:
  - F($\mathbf{n_i}$) $\leftarrow$ máx{F(**n**), f($\mathbf{n_i}$)}

RBFS

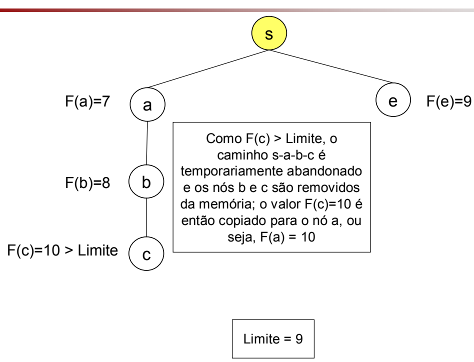
RBFS



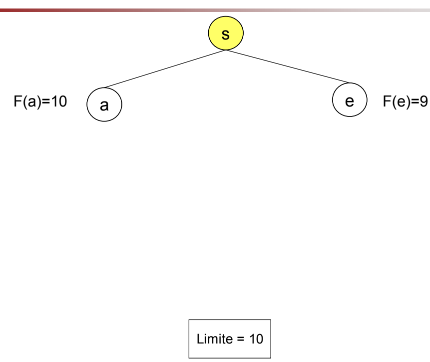
RBFS



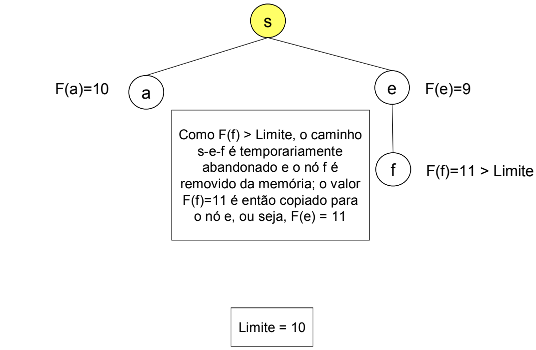
RBFS



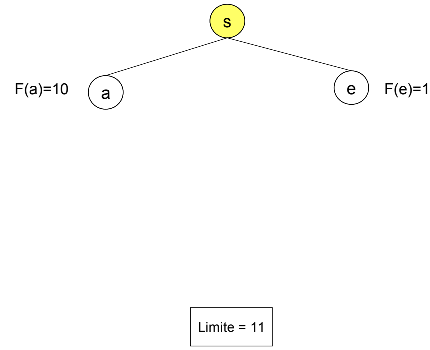
RBFS



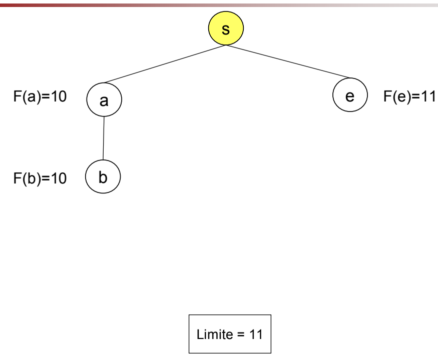
RBFS



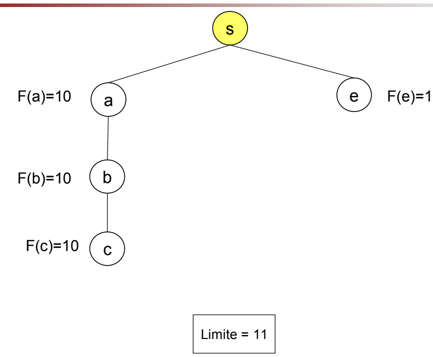
RBFS



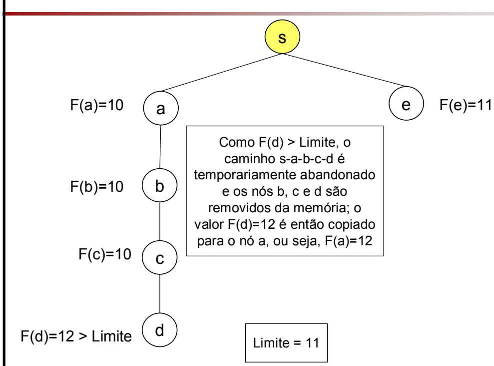
RBFS



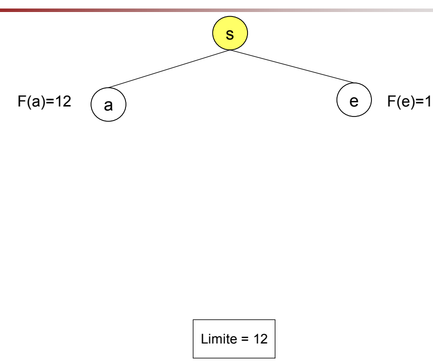
RBFS



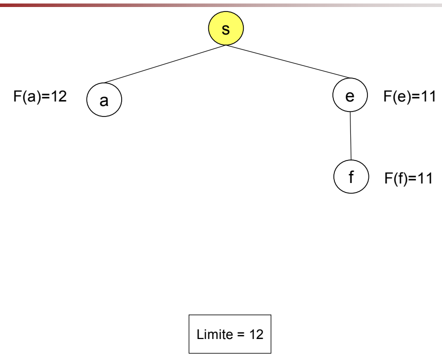
RBFS



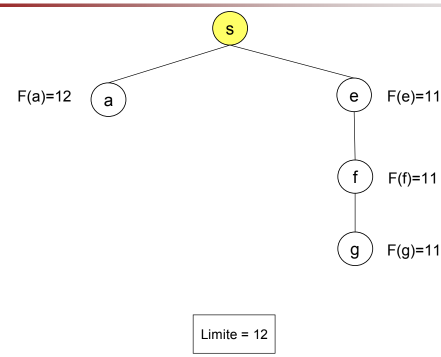
RBFS



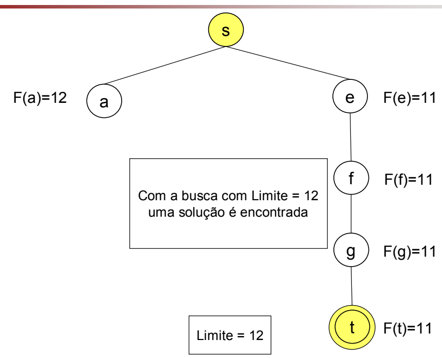
RBFS



RBFS



RBFS



RBFS

- rbfs(Caminho,Filhos,Limite,NovoMelhorFF,Resolvido, Solucao):
  - Caminho = caminho até então na ordem reversa
  - Filhos = filhos da cabeça do Caminho
  - Limite = limite superior no valor-F da busca para os Filhos
  - NovoMelhorFF = melhor valor-f justamente quando busca ultrapassa Limite
  - Resolvido = sim, não, nunca
  - Solucao = caminho da soluçao, se Resolvido = sim
- Representacao dos nos: No = l(Estado,G/F/FF)
  - G é o custo até Estado
  - F é o valor-f estático de Estado
  - FF é o valor-f de Estado copiado

RBFS

```
% Assuma que 9999 é maior que qualquer valor-f
resolva(No,Solucao) :-
  rbfs([],[l(No,0/0/0)],9999,_,sim,Solucao).

% rbfs(Caminho,Filhos,Limite,NovoMelhorFF,Resolvido,Solucao)
rbfs(Caminho,[l(No,G/F/FF)|Nos],Limite,FF,nao,_) :-
  FF > Limite, !.
rbfs(Caminho,[l(No,G/F/FF)|_],_,_,sim,[No|Caminho]) :-
  F = FF,     % Mostrar solucao apenas uma vez,quando F=FF
  final(No).
rbfs(_,[],_,_,nunca,_) :- !.   % Sem candidatos,beco sem saida
rbfs(Caminho,[l(No,G/F/FF)|Ns],Limite,NovoFF,Resolvido,Sol) :-
  FF =< Limite,                % Dentro de Limite: gerar filhos
  findall(Filho/Custo,
          (s(No,Filho,Custo),\+ pertence(Filho,Caminho)),
          Filhos),
  herdar(F,FF,FFherdado),    % Filhos podem herdar FF
  avalie(G,FFherdado,Filhos,Sucessores), % Ordenar filhos
  melhorff(Ns,ProximoMelhorFF),  % FF do competidor mais promissor dos filhos
  min(Limite,ProximoMelhorFF,Limite2), !,
  rbfs([No|Caminho],Sucessores,Limite2,NovoFF2,Resolvido2,Sol),
  continue(Caminho,[l(No,G/F/NovoFF2)|Ns],Limite,
           NovoFF,Resolvido2,Resolvido,Sol).
```


RBFS

```
% continue(Caminho,Nos,Limite,NovoFF,FilhoResolvido,Resolvido,Solucao)
continue(Caminho,[N|Ns],Limite,NovoFF,nunca,Resolvido,Sol) :-
  !,
  rbfs( Caminho,Ns,Limite,NovoFF,Resolvido,Sol). % N é um beco sem saida
continue(_,_,_,_,sim,sim,Sol).
continue(Caminho,[N|Ns],Limite,NovoFF,nao,Resolvido,Sol) :-
  inserir(N,Ns,NovoNs), !,% Assegurar que filhos sao ordenados pelos valores
  rbfs(Caminho,NovoNs,Limite,NovoFF,Resolvido,Sol).

avalie(_,_,[],[]).
avalie(G0,FFherdado,[No/C|NCs],Nos) :-
  G is G0 + C,
  h(No,H),
  F is G + H,
  max(F,FFherdado,FF),
  avalie(G0,FFherdado,NCs,Nos2),
  inserir(l(No,G/F/FF),Nos2,Nos).

herdar(F,FF,FF) :-  % Filho herda FF do pai se
  FF > F, !.        % FF do pai e' maior que F do pai
herdar(F,FF,0).
```

RBFS

```
inserir(l(N,G/F/FF),Nos,[l(N,G/F/FF)|Nos]) :-
  melhorff(Nos,FF2),
  FF =< FF2, !.
inserir(N,[N1|Ns],[N1|Ns1]) :-
  inserir(N,Ns,Ns1).

melhorff([l(N,F/G/FF)|Ns],FF).  % Primeiro no' = melhor FF
melhorff([],9999).              % Sem nos FF = "infinito"

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

min(X,Y,X) :-
  X =< Y, !.
min(X,Y,Y).

max(X,Y,X) :-
  X >= Y, !.
max(X,Y,Y).
```


Resumo

- Vimos que os algoritmos IDA* e RBFS necessitam de quantidade de espaço linear na profundidade da busca
- Diferentemente de IDA* e como A*, RBFS expande nós na ordem best-first mesmo no caso de uma função f não monotônica

Slides baseados nos livros:

Bratko, I.;
Prolog Programming for Artificial Intelligence,
3rd Edition, Pearson Education, 2001.

Clocksin, W.F.; Mellish, C.S.;
Programming in Prolog,
5th Edition, Springer-Verlag, 2003.

Material elaborado por
José Augusto Baranauskas
Revisão 2006