Estratégias de Busca Informada

Inteligência Artificial

- Nesta aula são descritas algumas estratégias de busca informada em espaços de estados que são usadas quando há informações específicas sobre o problema
- Em geral, as informações específicas tendem a melhorar o processo de busca

José Augusto Baranauskas
Departamento de Computação e Matemática – FFCLRP-USP

augusto@usp.br
http://dcm.fmrp.usp.br/~augusto

Busca Informada

- A busca em grafos pode atingir uma complexidade elevada devido ao número de alternativas
- Estratégias de busca informada utilizam conhecimento específico do problema (além da definição do próprio problema) para encontrar soluções de forma mais eficiente do que a busca sem informações
- A abordagem geral que utilizaremos é denominada **busca pela melhor escolha** (**best-first** search)
- O termo "busca pela melhor escolha" tem seu uso consagrado mas ele é inexato
  - Se fosse realmente possível expandir o melhor nó primeiro, não haveria a necessidade de se realizar uma busca; seria um caminho direto ao objetivo
  - O que podemos fazer é escolher o nó que *parece* ser o melhor de acordo com a função de avaliação
  - Se a função de avaliação for precisa, esse será de fato o melhor nó; caso contrário, a busca pode se perder

Busca Informada

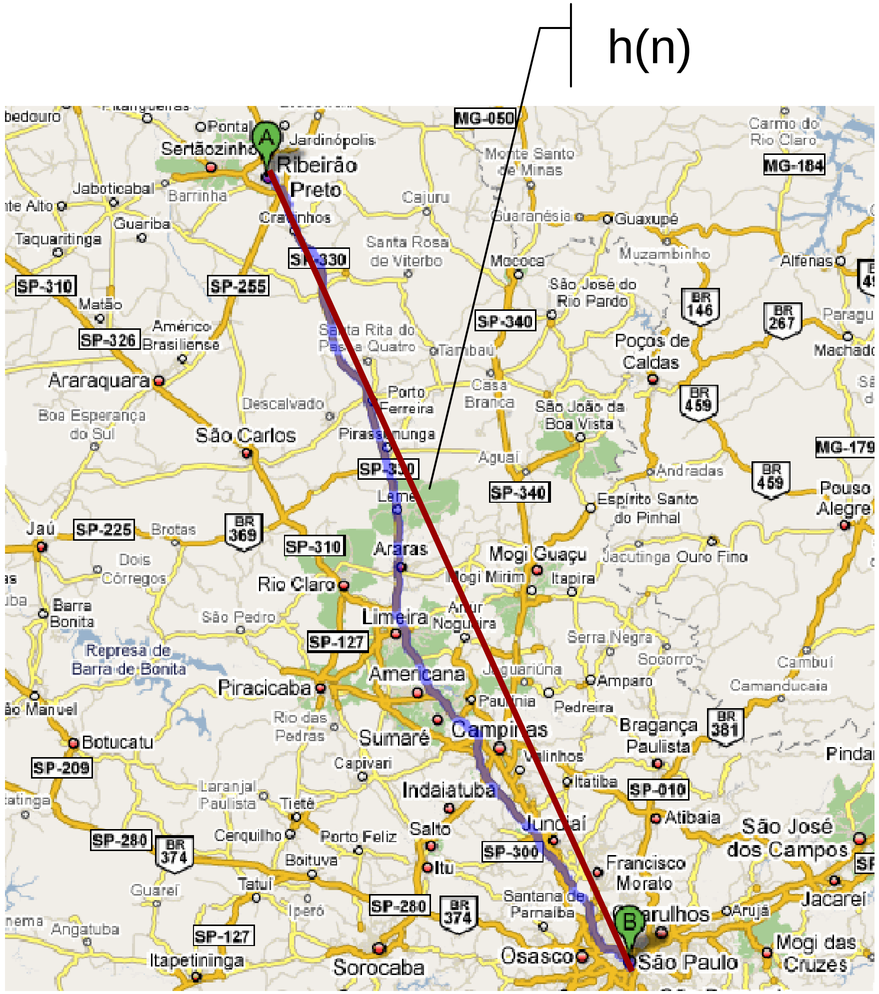
- Na busca best-first, a escolha do nó **n** a ser expandido é efetuada com base em uma função de avaliação **f(n)**
  - Em geral, o nó **n** com o menor valor **f(n)** é selecionado para a expansão, uma vez que a função de avaliação mede a “distância” de **n** até o objetivo, ou seja, até um nó final
- A forma mais comum de adicionar **conhecimento** do problema ao algoritmo de busca (daí o nome **busca informada**) e, portanto, um componente fundamental dos algoritmos de busca best-first é uma **função heurística**, denotada por **h(n)**
  - **h(n)** = custo **estimado** do caminho mais econômico partindo do nó **n** e chegando a um nó final
- As funções heurísticas são específicas do problema, exceto pela seguinte restrição que se aplica a todos
  - **h(n)**=0 se **n** é um nó final

Função Heurística

- **Heurística** (arte de descobrir): conhecimentos que permitem uma solução rápida para algum problema
  - Heureca (“Eu encontrei”, Arquimedes)
- Uma **heurística** é uma função que quando aplicada a um estado retorna um número que corresponde a uma **estimativa** da qualidade (mérito) do estado atual em relação a um estado final
- Em outras palavras, a heurística nos informa **aproximadamente** quão longe o estado atual está de um estado final
  - Heurísticas podem subestimar ou superestimar a qualidade de um estado
  - Heurísticas que apenas subestimam são altamente desejáveis e são chamadas **admissíveis**
    - i.e. número menores são melhores
  - Heurísticas admissíveis são **otimistas**, pois estimam que o custo da solução de um problema seja menor do que é na realidade

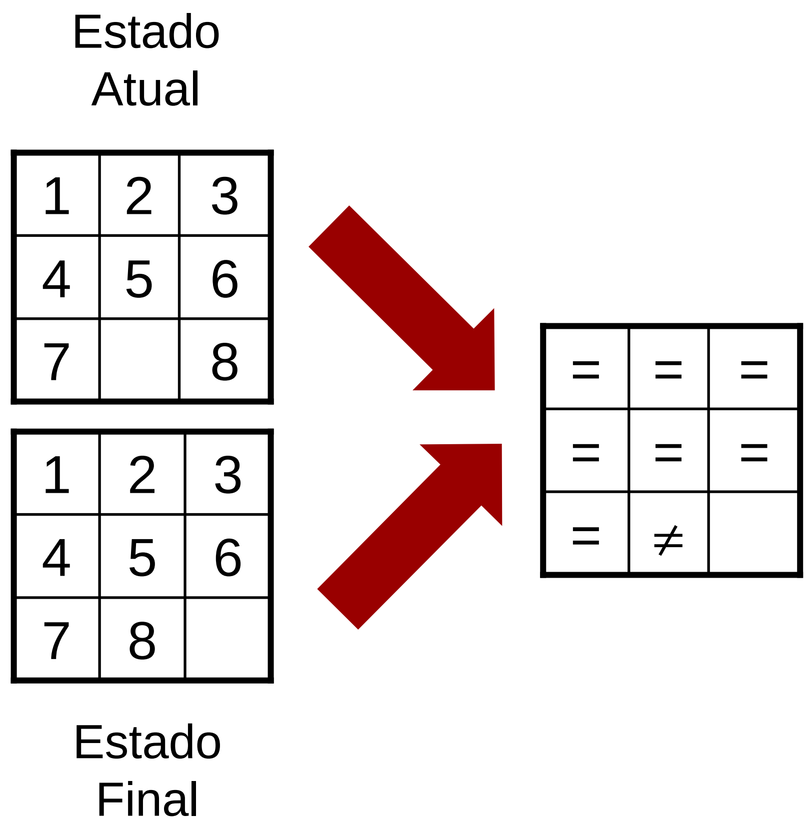
Função Heurística: Distância em Linha Reta

- Por exemplo, em um mapa contendo cidades e estradas que as interconectam, poderíamos estimar o custo do caminho mais curto (econômico) entre duas cidades como sendo a **distância em linha reta**
- Esta heurística é admissível pois o caminho mais curto entre dois pontos quaisquer no plano é uma linha reta e, portanto, não pode ser uma superestimativa



Função Heurística I para o Quebra-Cabeça de 8 Peças

- Número de peças fora do lugar (sem incluir o espaço vazio)
- Nesta situação, apenas a peça “8” está fora do lugar; portanto a função heurística é igual a 1
- Em outras palavras, a heurística nos informa que ela **estima** que uma solução pode estar disponível em apenas mais um movimento
  - h(estado atual) = 1



Função Heurística II para o Quebra-Cabeça de 8 Peças

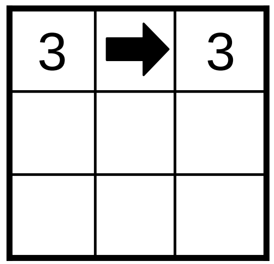
- Distância Manhattan (sem incluir o espaço vazio)
- Nesta situação, apenas as peças “1”, “3” e “8” estão fora do lugar por 2, 3 e 3 posições, respectivamente; portanto a função heurística é igual a 8
- Em outras palavras, a heurística nos informa que ela **estima** que uma solução pode estar disponível em apenas mais 8 movimentos
  - h(estado atual) = 8

Estado Atual

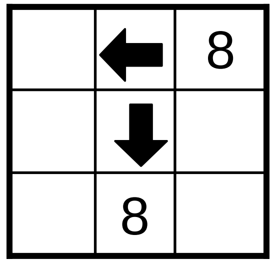
3	2	8
4	5	6
7	1	

1	2	3
4	5	6
7	8	

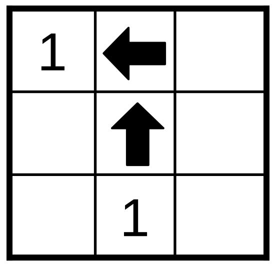
Estado Final



2 posições



3 posições



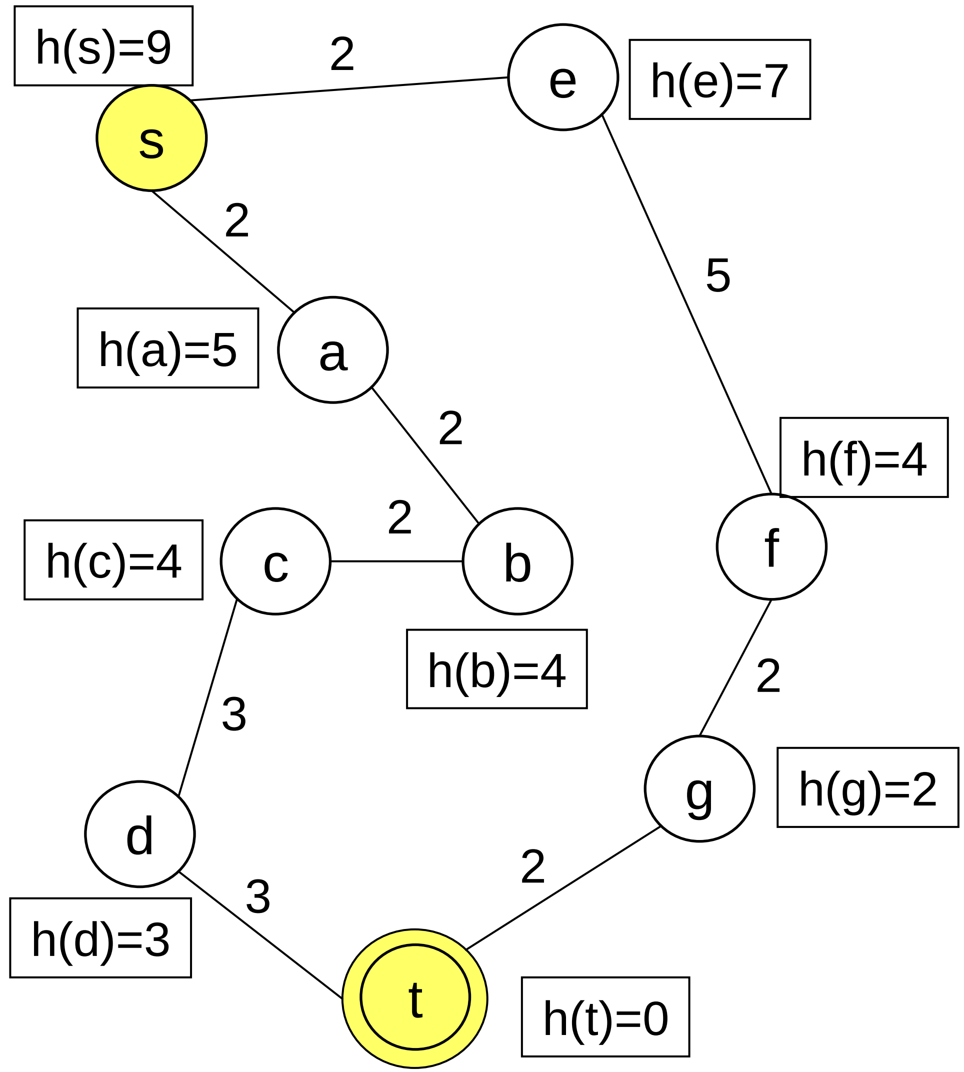
3 posições

Busca Gulosa pela Melhor Escolha

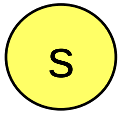
- A **busca gulosa pela melhor escolha** (greedy best-first search) tenta expandir o nó mais próximo ao nó final, assumindo que isso provavelmente levará a uma solução rápida
- Dessa forma, a busca avalia os nós utilizando apenas a função heurística
  - f(n) = h(n)
- Greedy best-first é semelhante à busca em profundidade
  - Prefere seguir um único caminho até o objetivo, retrocedendo somente se encontrar um beco sem saída
  - Mesmo defeitos da busca em profundidade: não é ótima, nem completa (pode entrar em um caminho infinito e nunca retornar para testar outras possibilidades)
  - Complexidade de tempo e espaço no pior caso são iguais a $O(b^m)$, onde **m** é a profundidade máxima do espaço de busca

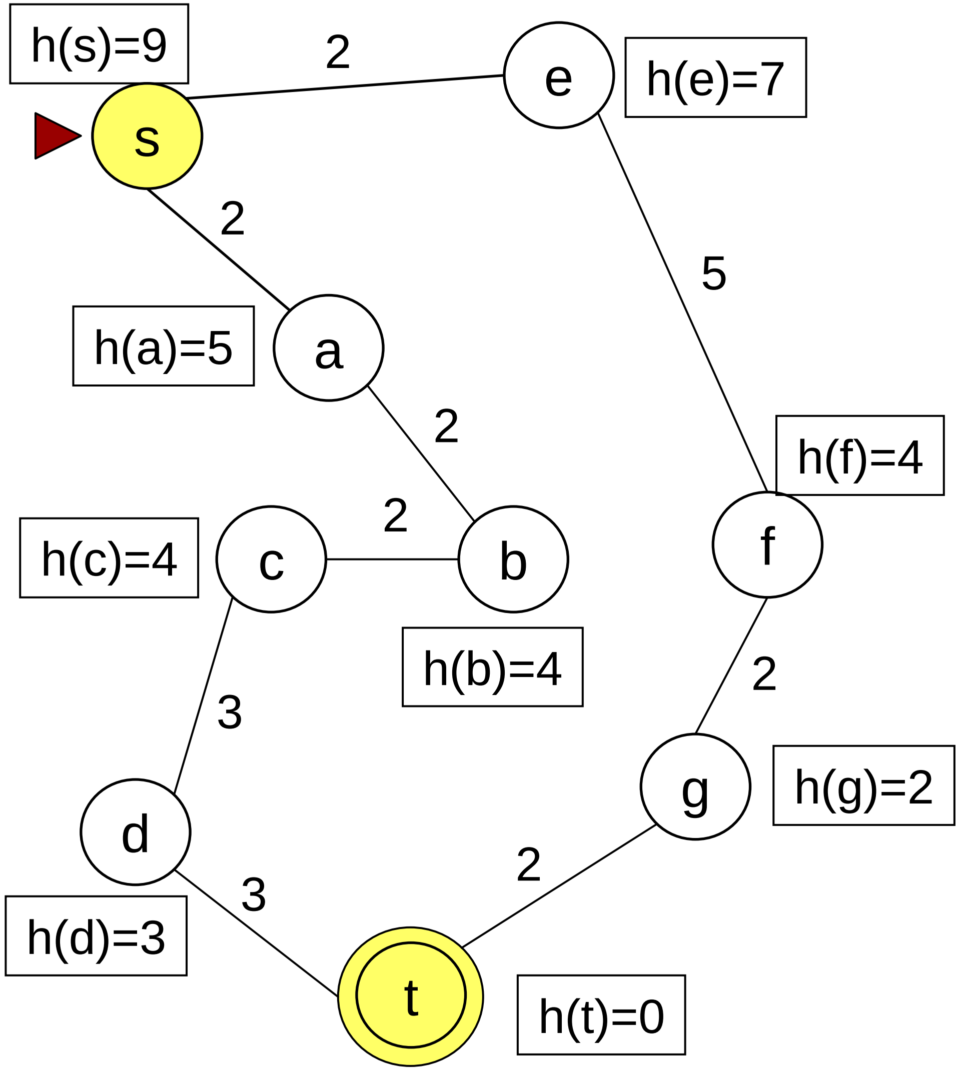
Exercício

- Encontre o caminho de **s** até **t** usando busca gulosa pela melhor escolha

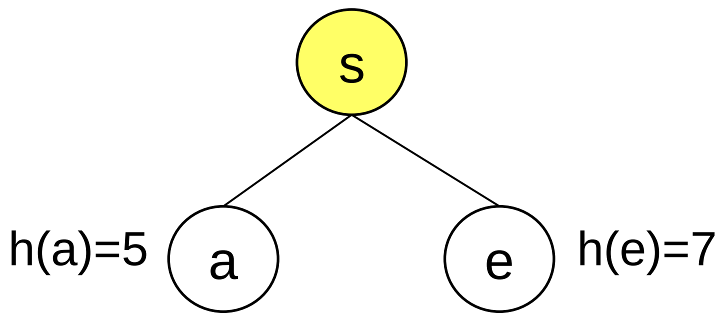


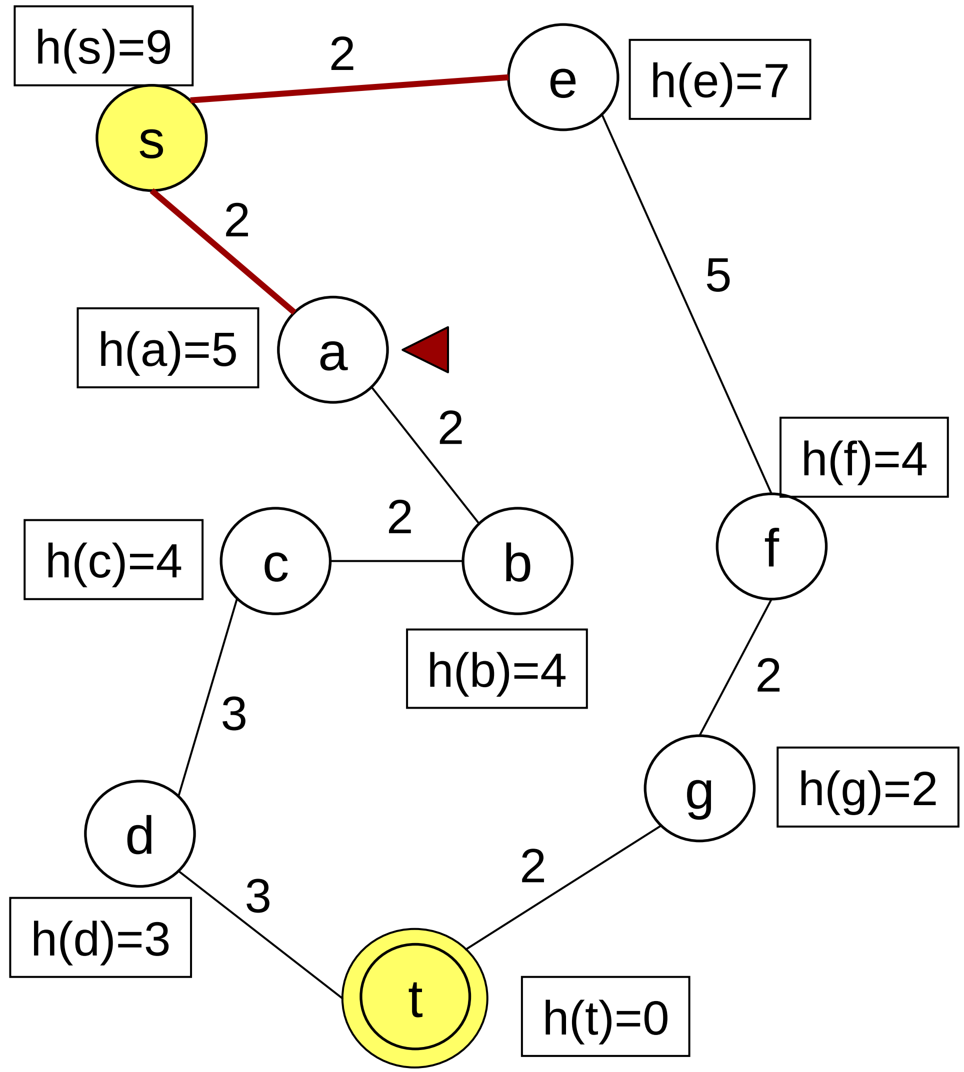
Busca Gulosa pela Melhor Escolha



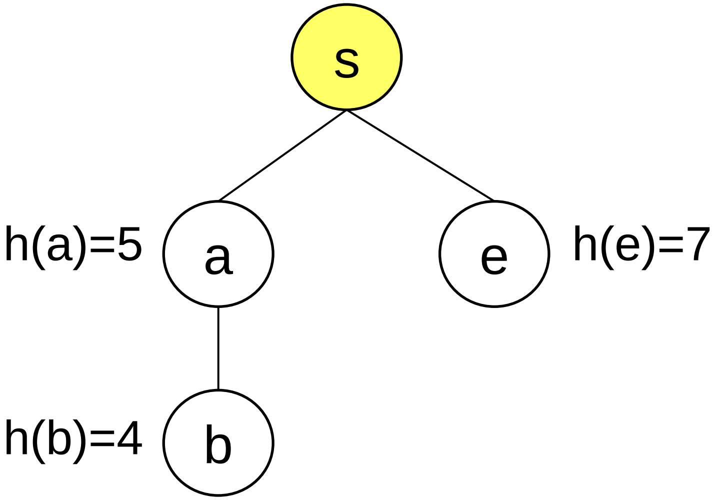


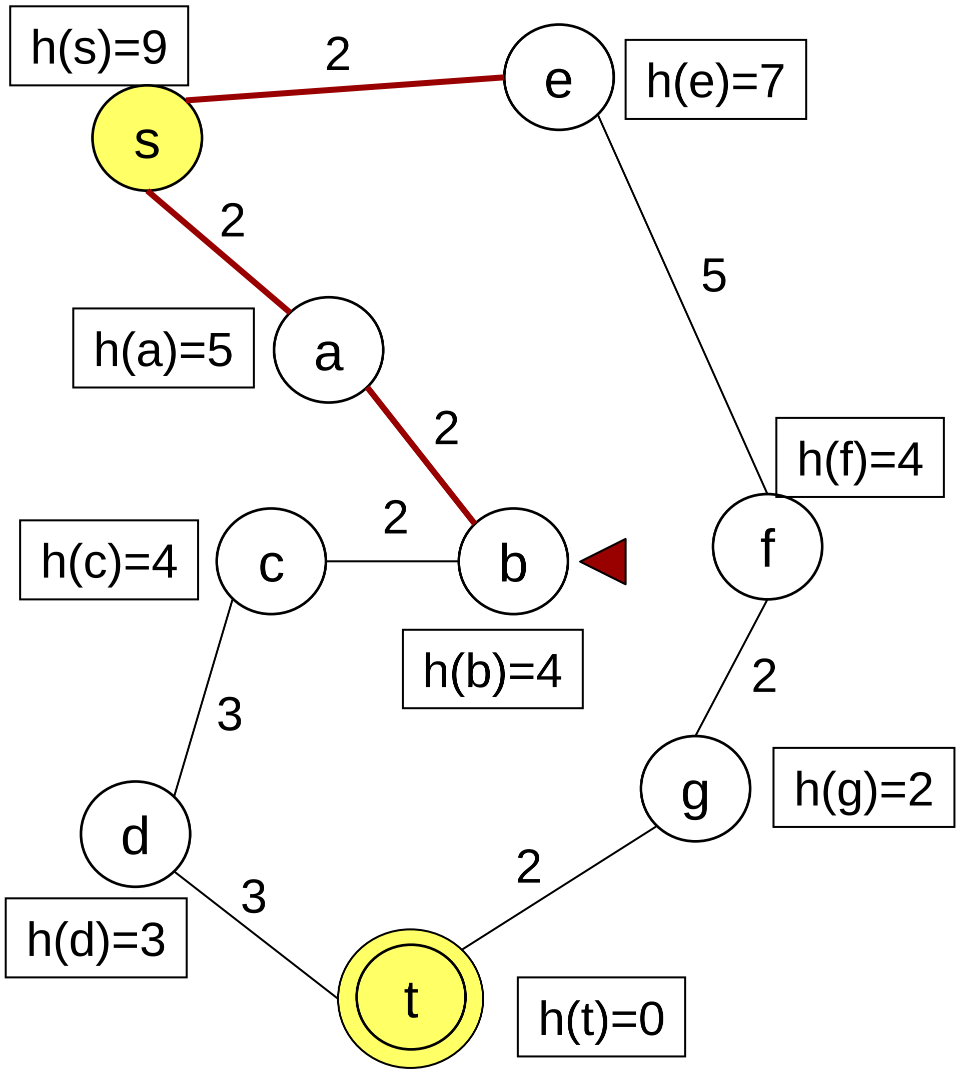
Busca Gulosa pela Melhor Escolha



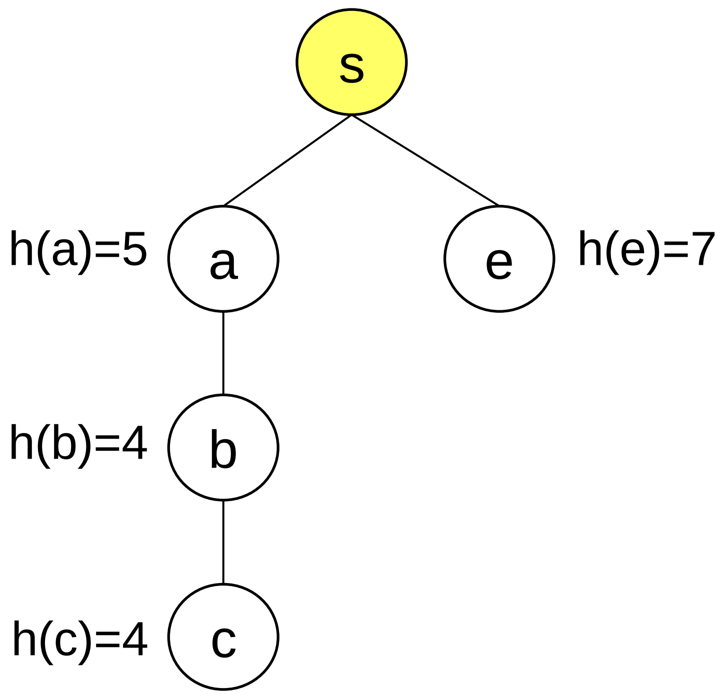


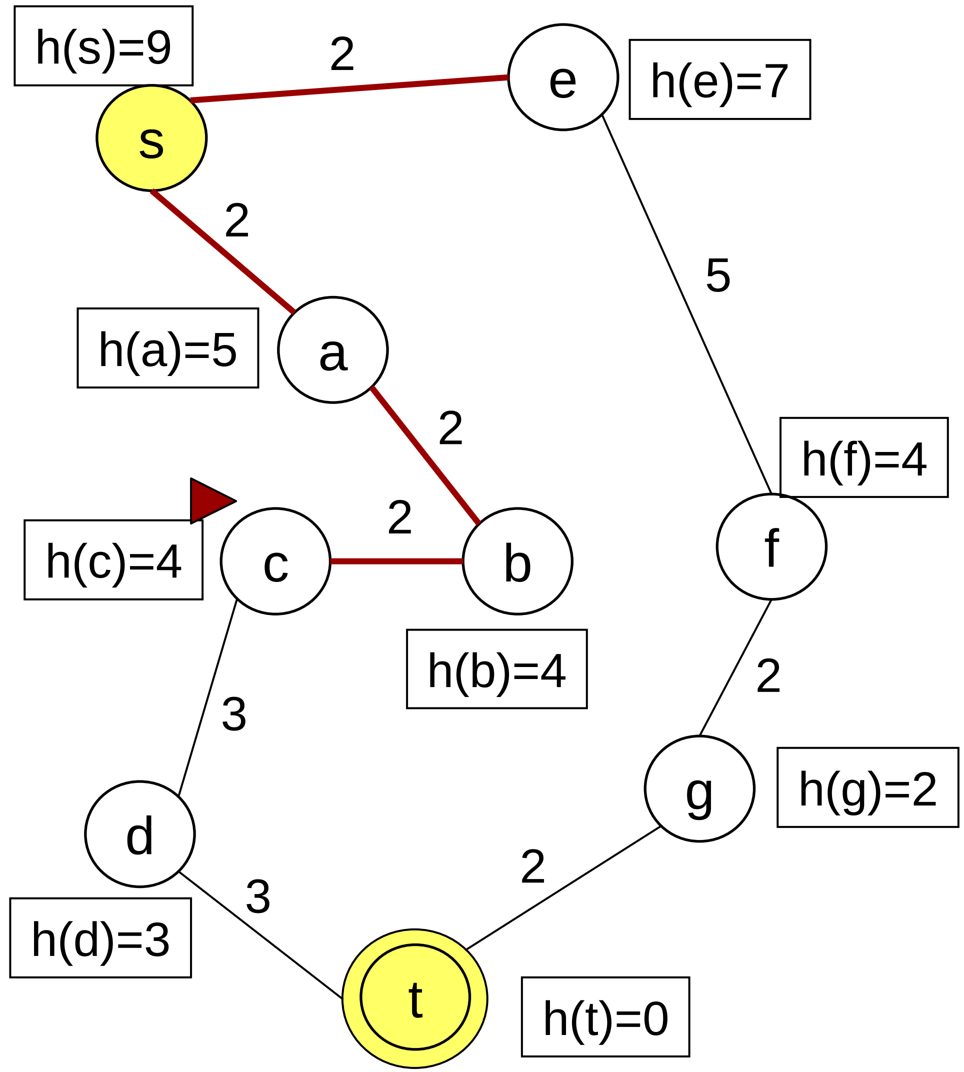
Busca Gulosa pela Melhor Escolha



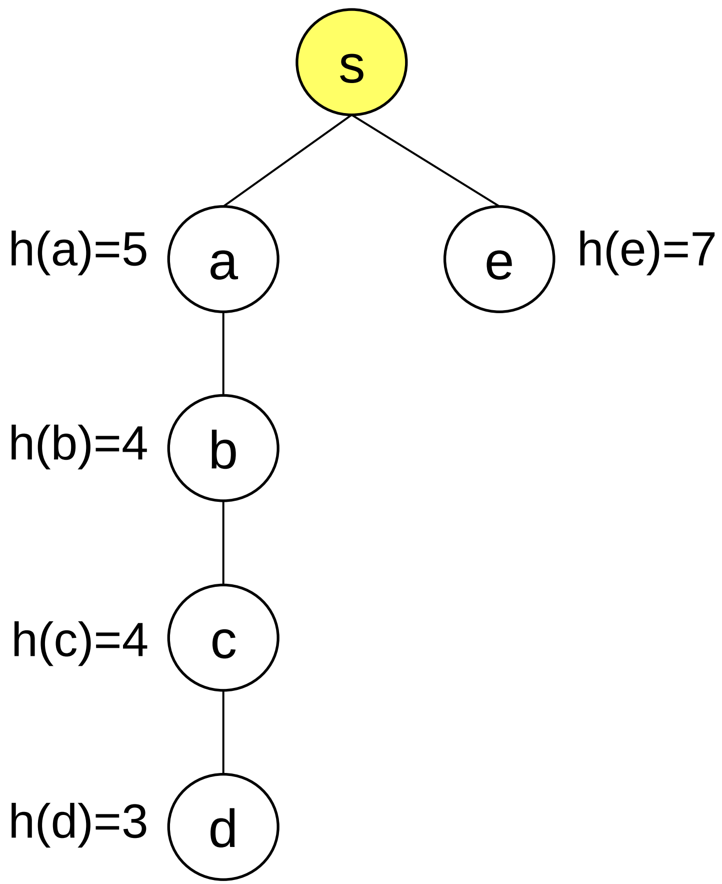


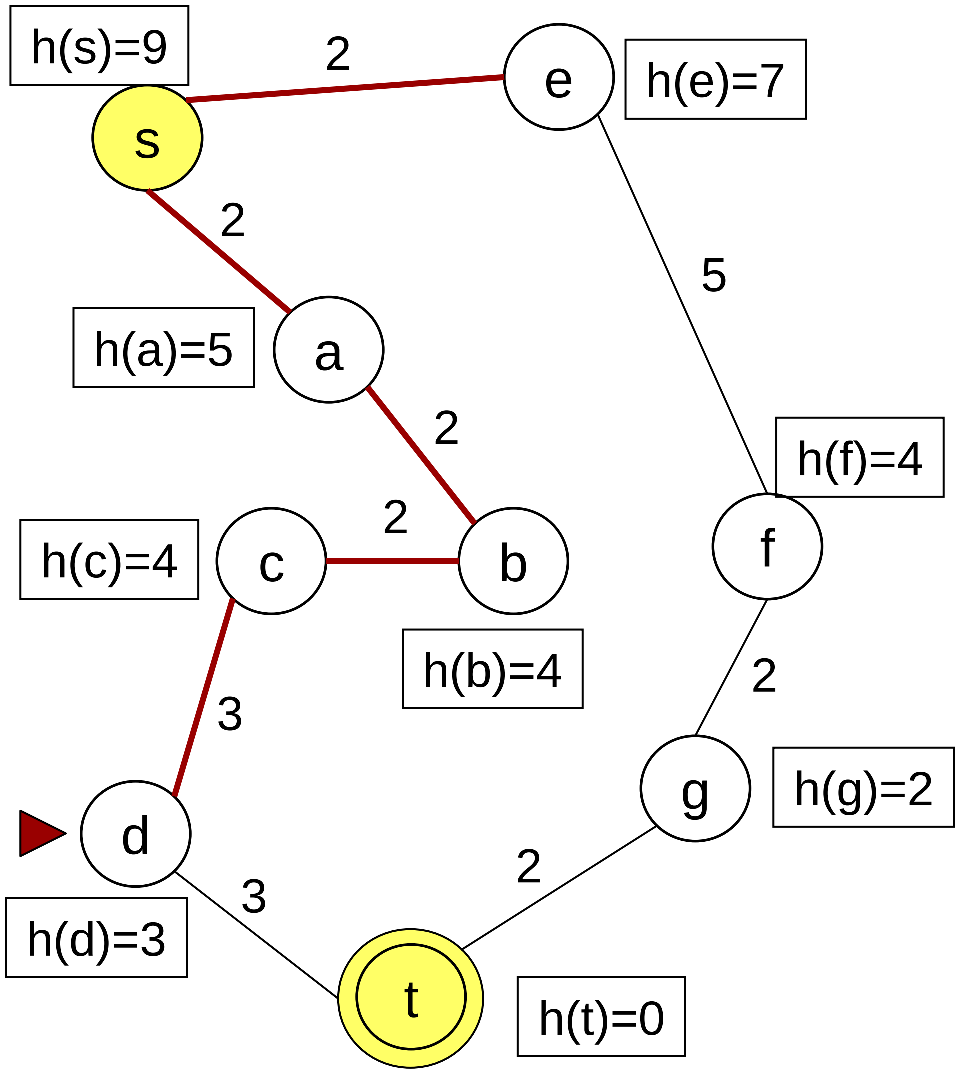
Busca Gulosa pela Melhor Escolha



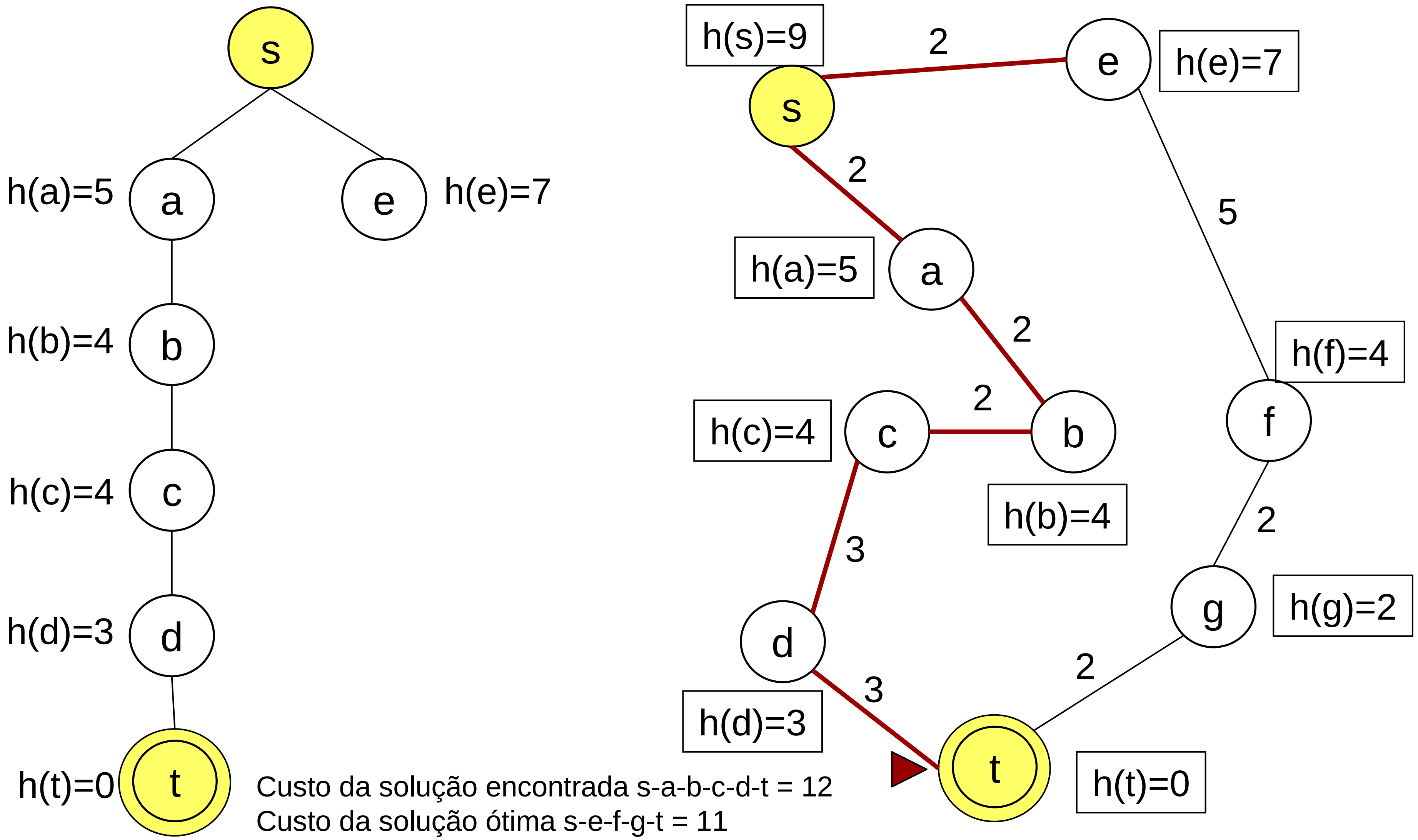


Busca Gulosa pela Melhor Escolha





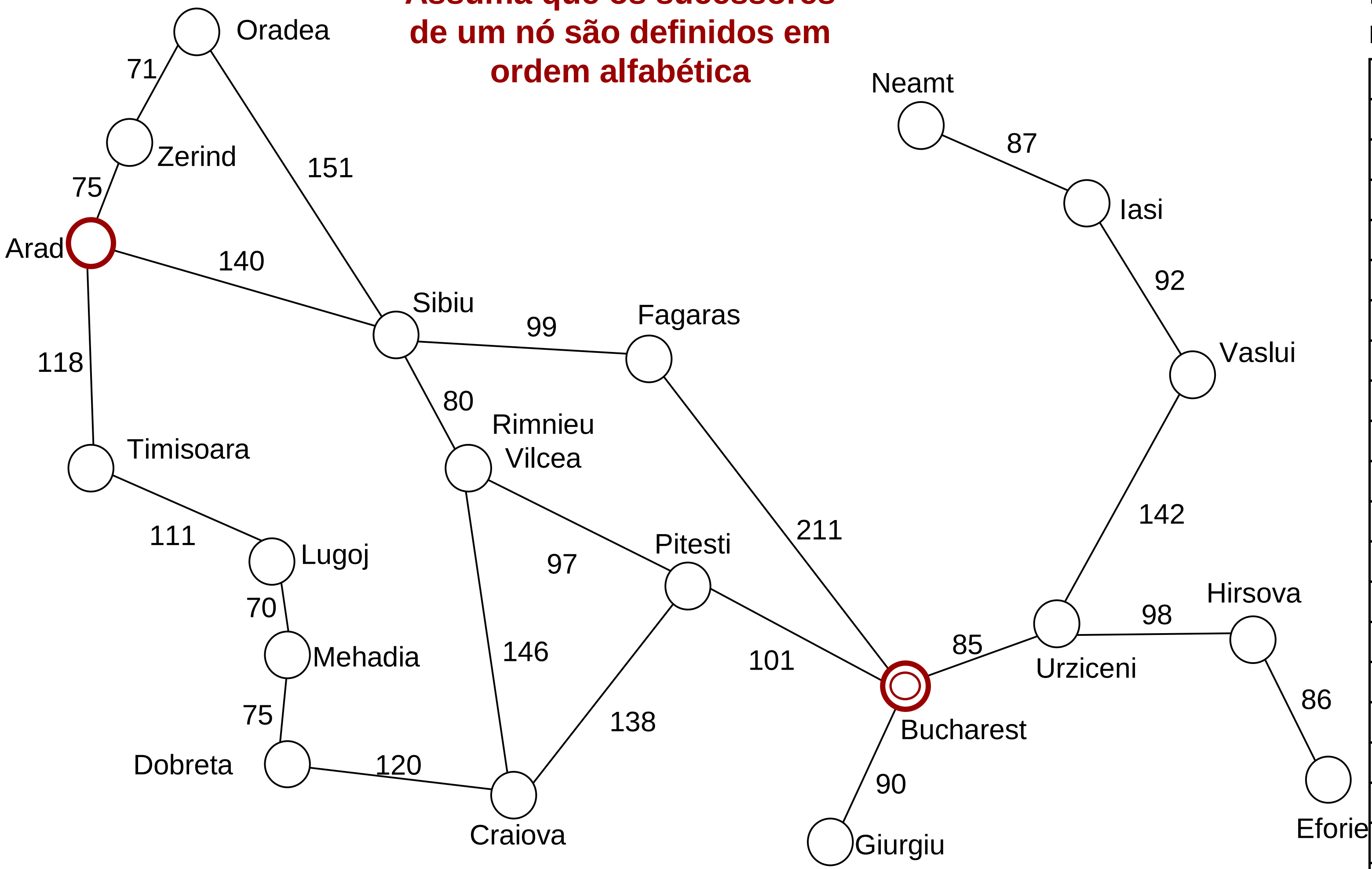
Busca Gulosa pela Melhor Escolha



Custo da solução encontrada s-a-b-c-d-t = 12
Custo da solução ótima s-e-f-g-t = 11

Busca Gulosa pela Melhor Escolha: Encontre o caminho de Arad até Bucharest

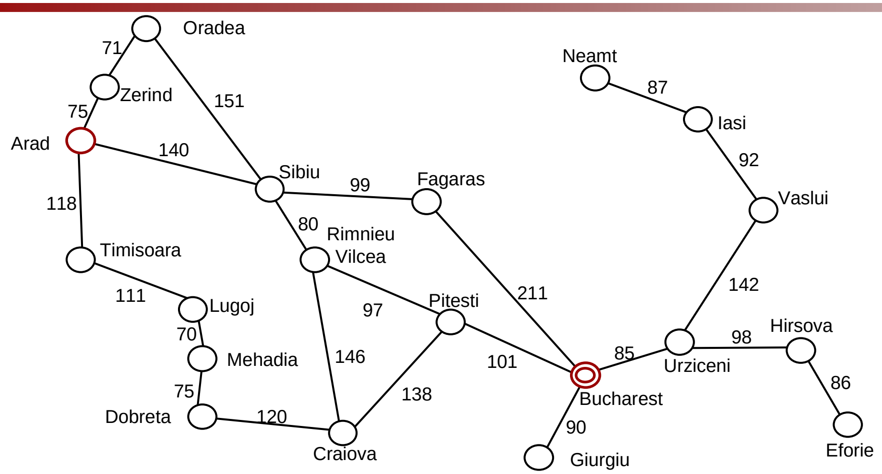
Assuma que os sucessores de um nó são definidos em ordem alfabética

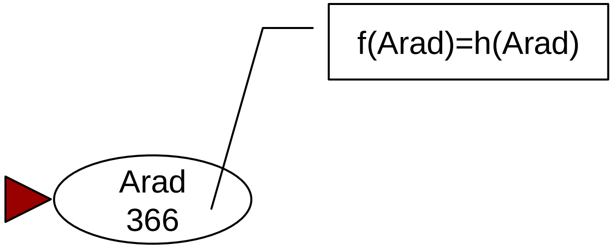


Distância em linha reta até Bucharest

h(n)	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnieu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

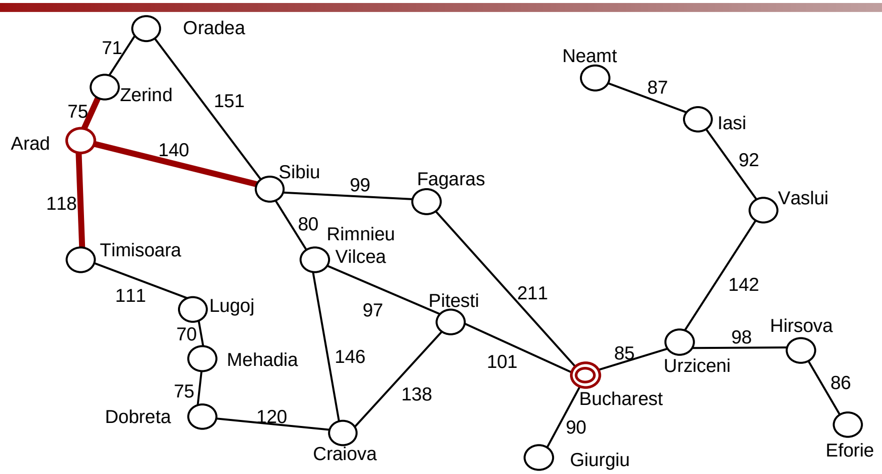
Encontre o caminho de Arad até Bucharest

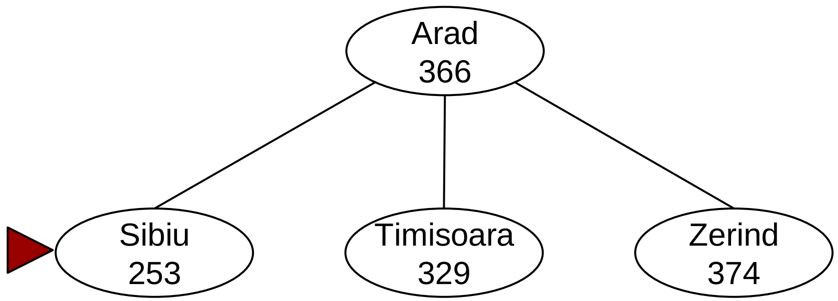




h(n)	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnieu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

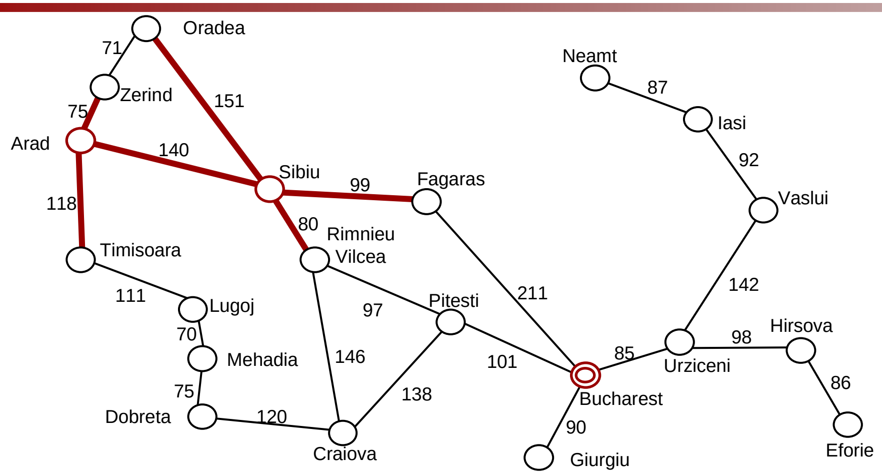
Encontre o caminho de Arad até Bucharest

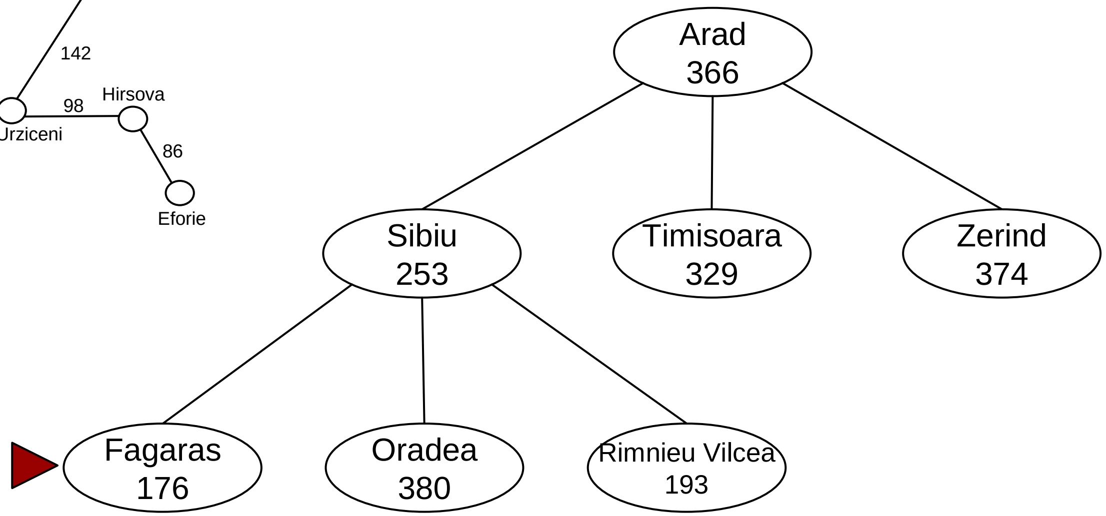




h(n)	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnieu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

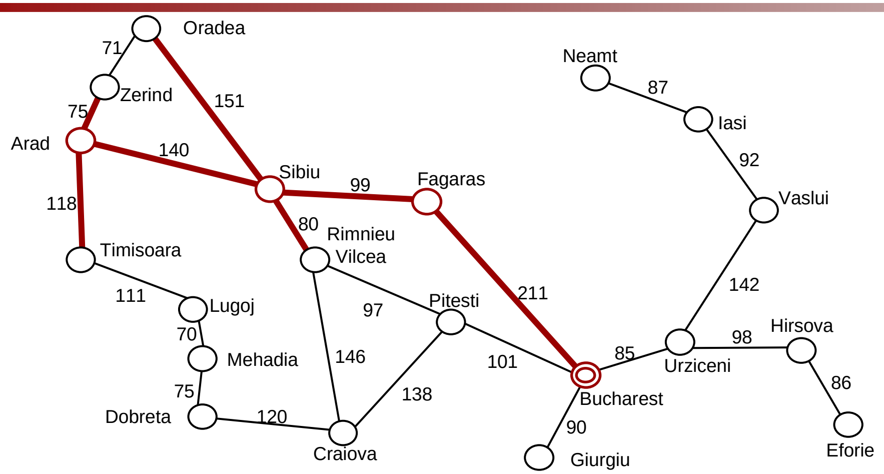
Encontre o caminho de Arad até Bucharest



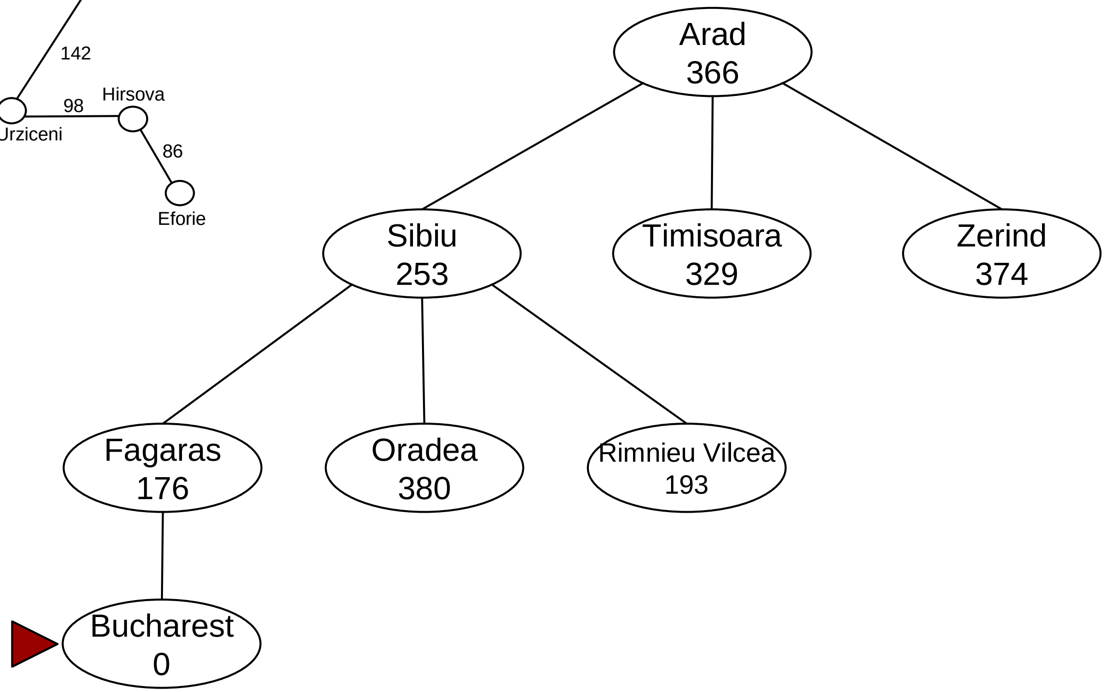


h(n)	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnieu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Encontre o caminho de Arad até Bucharest



h(n)	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnieu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



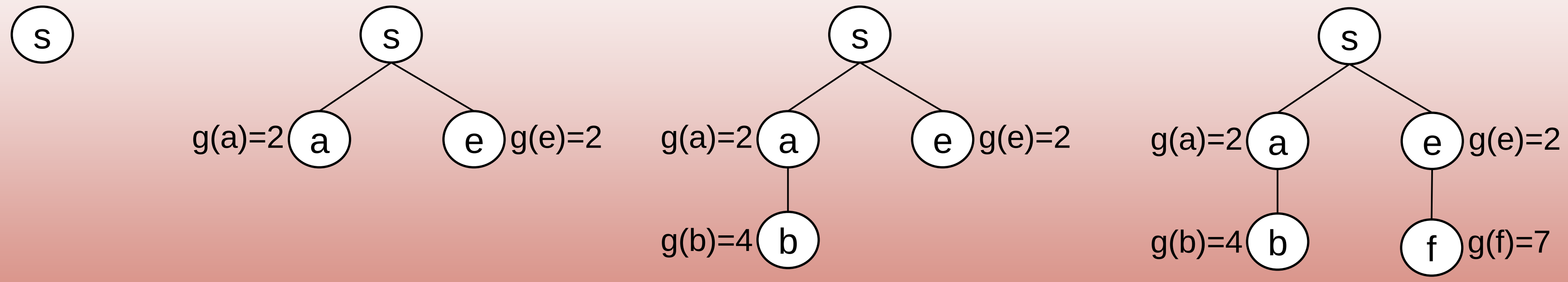
Custo da solução encontrada A-S-F-B = 450
Custo da solução ótima A-S-RV-P-B = 418

O que sabemos?

- Busca de custo uniforme (uniform cost)
  - Similar à busca em largura
  - Mede o custo de cada nó, desde a raiz da busca
  - É ótima e completa
  - Pode ser muito lenta
- Busca Gulosa pela Melhor Escolha (greedy best-first)
  - Similar à busca em profundidade
  - Mede o custo estimado de cada nó até um nó final
  - Não é ótima nem completa
  - Pode ser muito rápida, dependendo da qualidade da heurística e do problema específico
- Seria possível combinar estas estratégias para criar um algoritmo ótimo e completo que seja também muito rápido?

Busca de Custo Uniforme

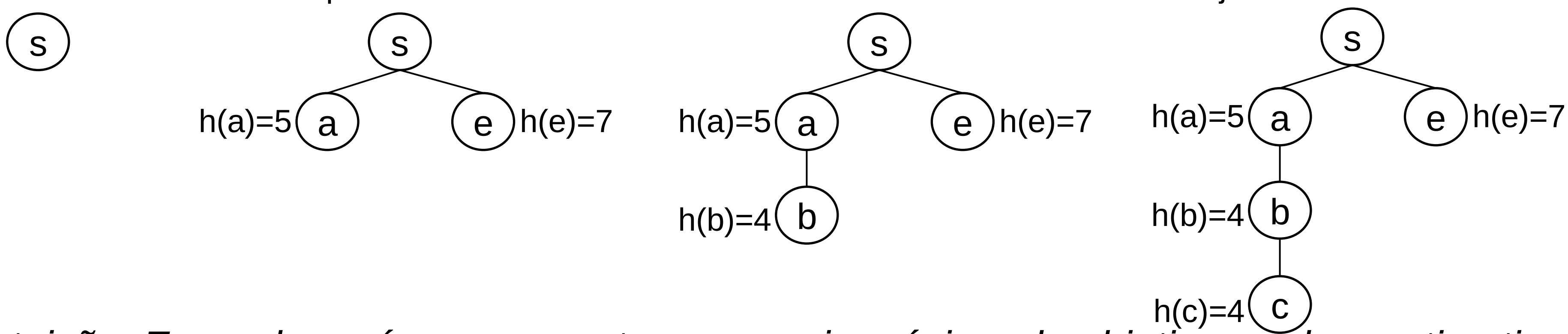
Coloque os estados na fila em ordem de custo



Intuição: Expanda o nó mais barato, onde o custo é o custo do caminho g(n)

Busca Gulosa pela Melhor Escolha

Coloque os estados na fila em ordem de distância estimada até o objetivo



Intuição: Expanda o nó que aparenta ser o mais próximo do objetivo, onde a estimativa da distância até o objetivo é h(n)

Busca A*

Coloque os estados na fila em ordem de custo total até o objetivo, f(n)

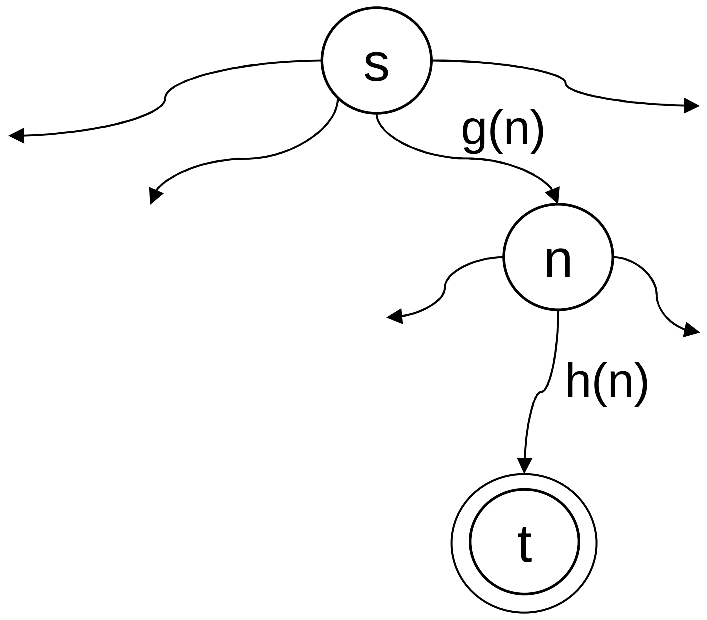
- Resumidamente:
  - **g(n)** é o custo até chegar ao nó **n**
  - **h(n)** é a distância estimada do nó **n** até um nó final
  - **f(n)** = **g(n)** + **h(n)**
- Podemos imaginar **f(n)** como uma estimativa de custo da solução mais barata que passa pelo nó **n**
- Note que é possível usar os algoritmos de busca em largura ou de custo uniforme alterando apenas a estratégia na qual os nós são colocados na fila
- Se a heurística utilizada nunca superestima a distância até o nó final (heurística otimista), então o algoritmo A* é ótimo e completo

Busca A*

- É conveniente relembrar que uma estratégia de busca é definida por meio da ordem de expansão dos nós
- Na estratégia de busca *best-first* a idéia básica é prosseguir com a busca sempre a partir do nó mais promissor
- A **Busca pela Melhor Escolha** (best-first) é um refinamento da busca em largura, que em sua forma completa também é conhecido como **busca A***
  - 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

A*

- 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**
- 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**



A*

- 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**

A*

- Vamos estudar o algoritmo best-first em sua forma completa A* que minimiza o custo total estimado da solução
- 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

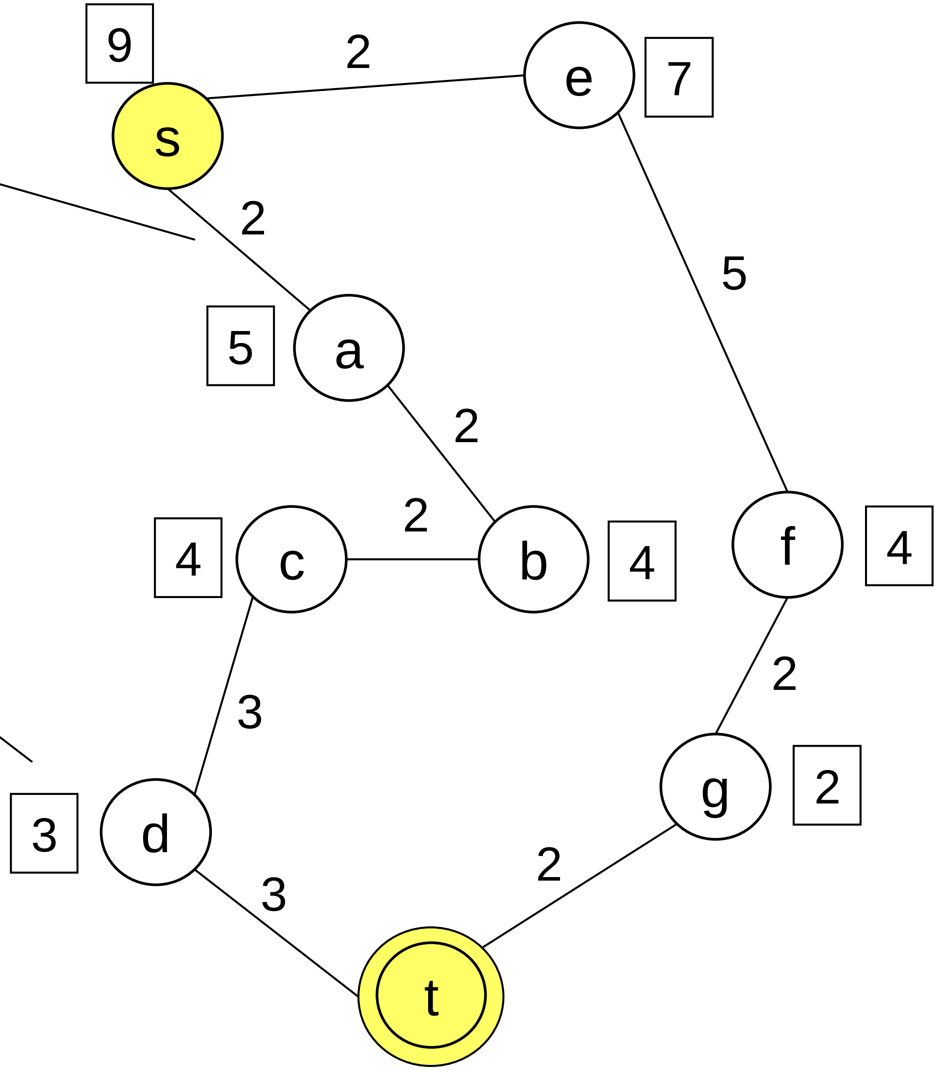
A*

- 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

A*

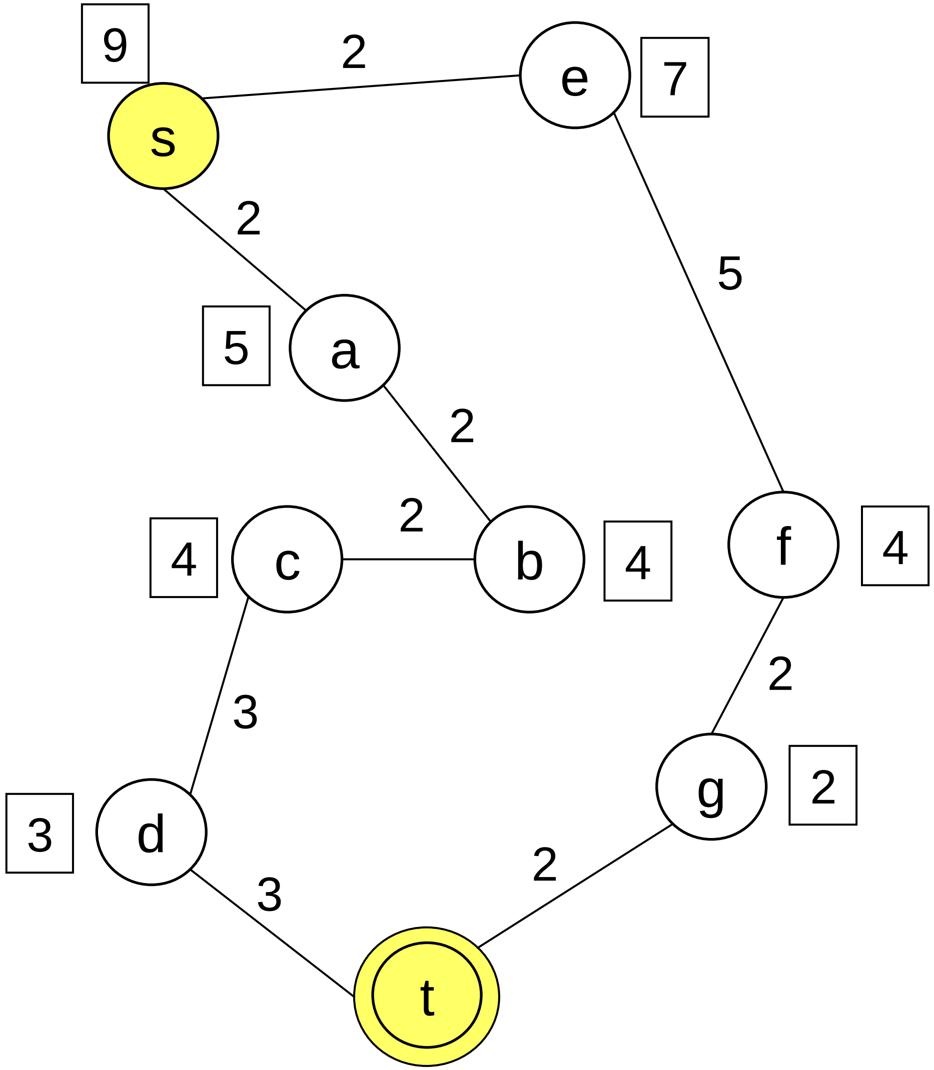
Distância entre duas cidades através de um caminho (rodovia)

Distância entre a cidade em questão e a cidade destino (t) em linha reta



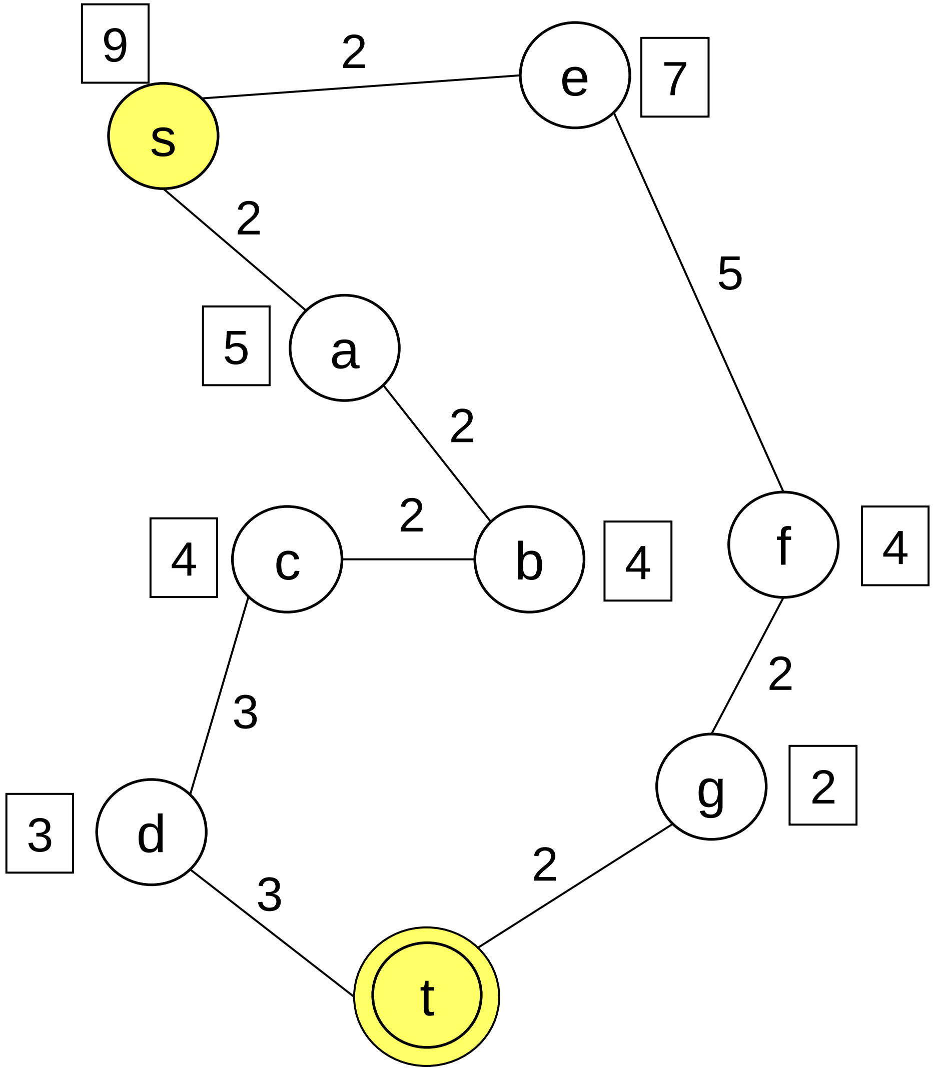
A*

- 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)
- Note que quando X é á cidade inicial (X=**s**) então g(**s**)=0
  - f(**s**)=0+9=9
- O primeiro nó a ser expandido é o nó inicial (raiz da busca)



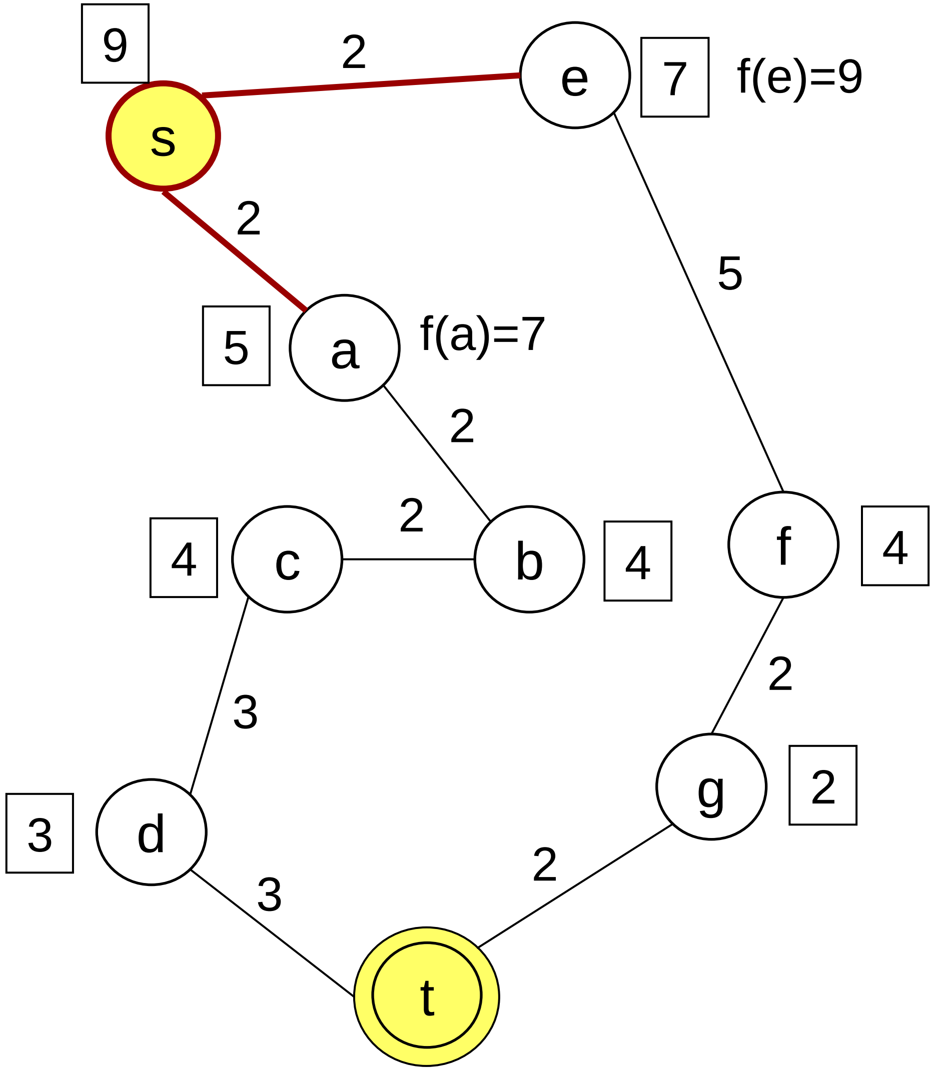
A*

- Neste exemplo, podemos imaginar a busca A* 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**



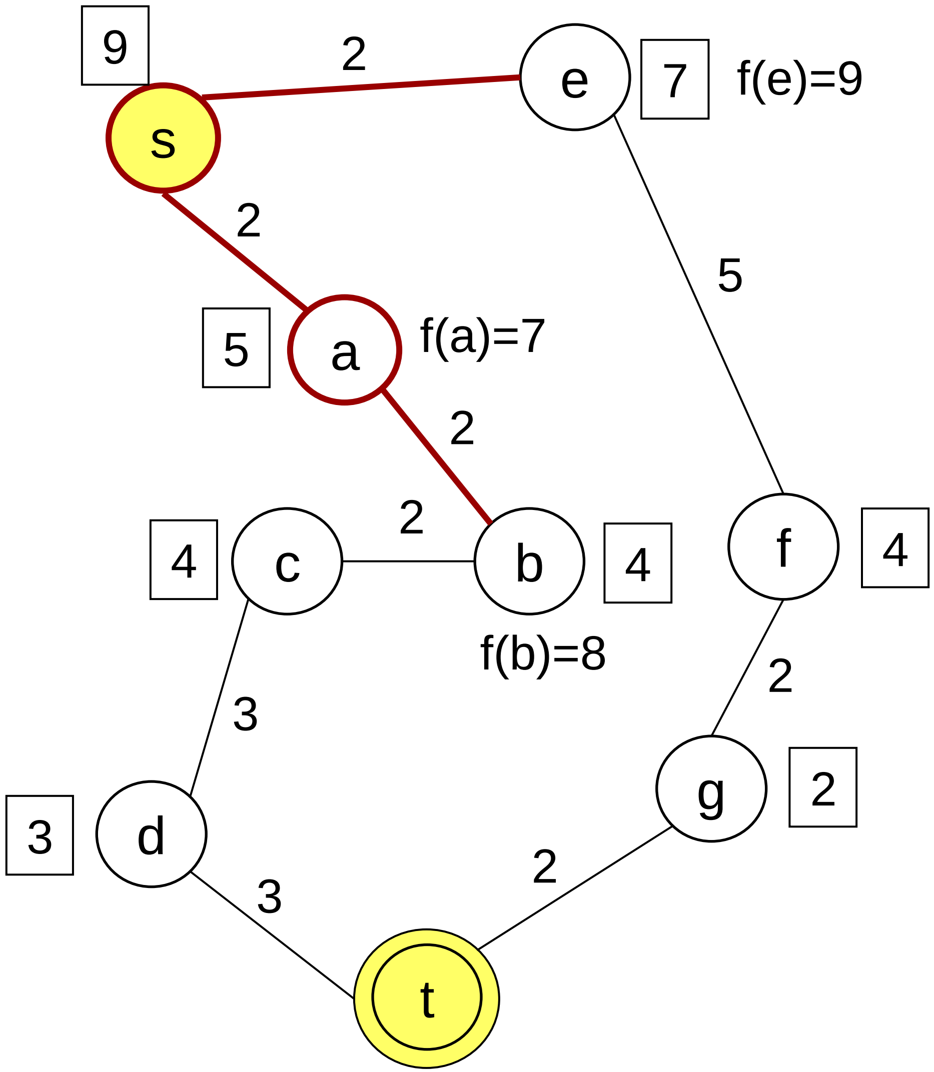
A*

- 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



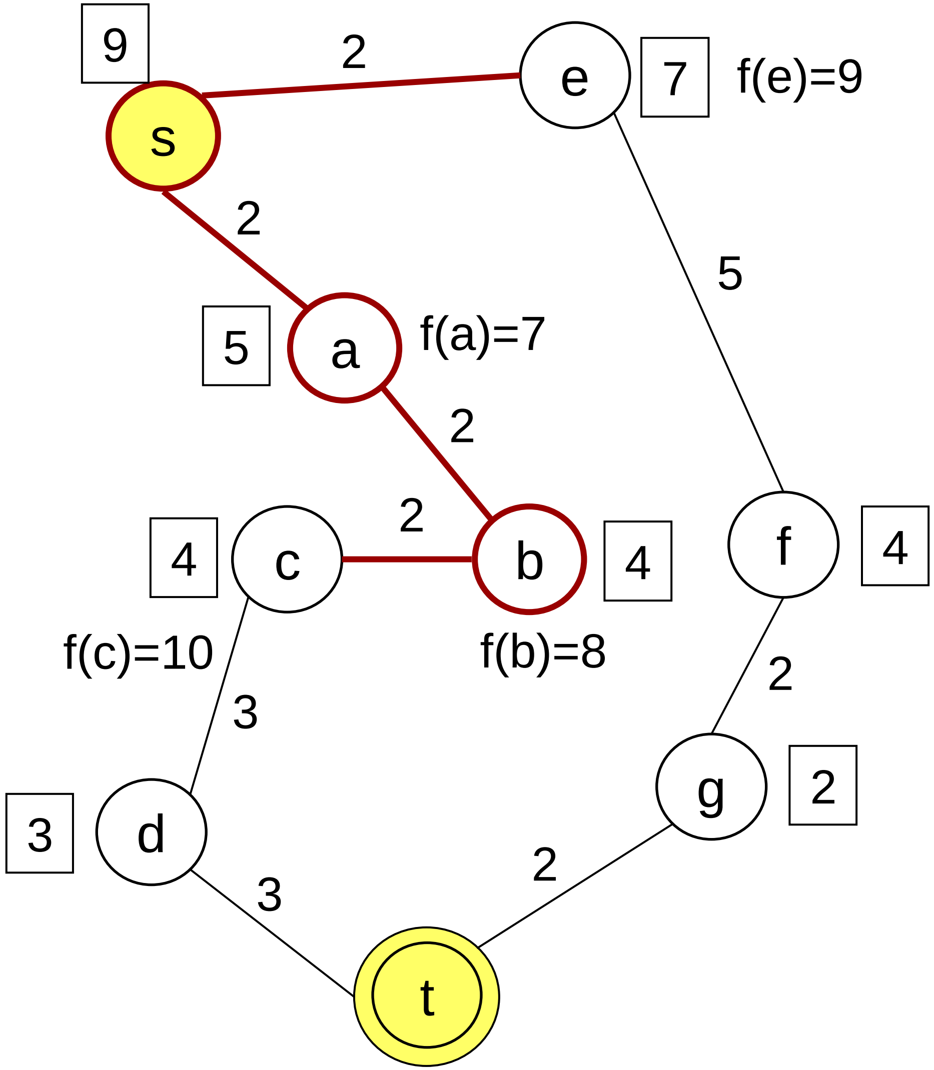
A*

- 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



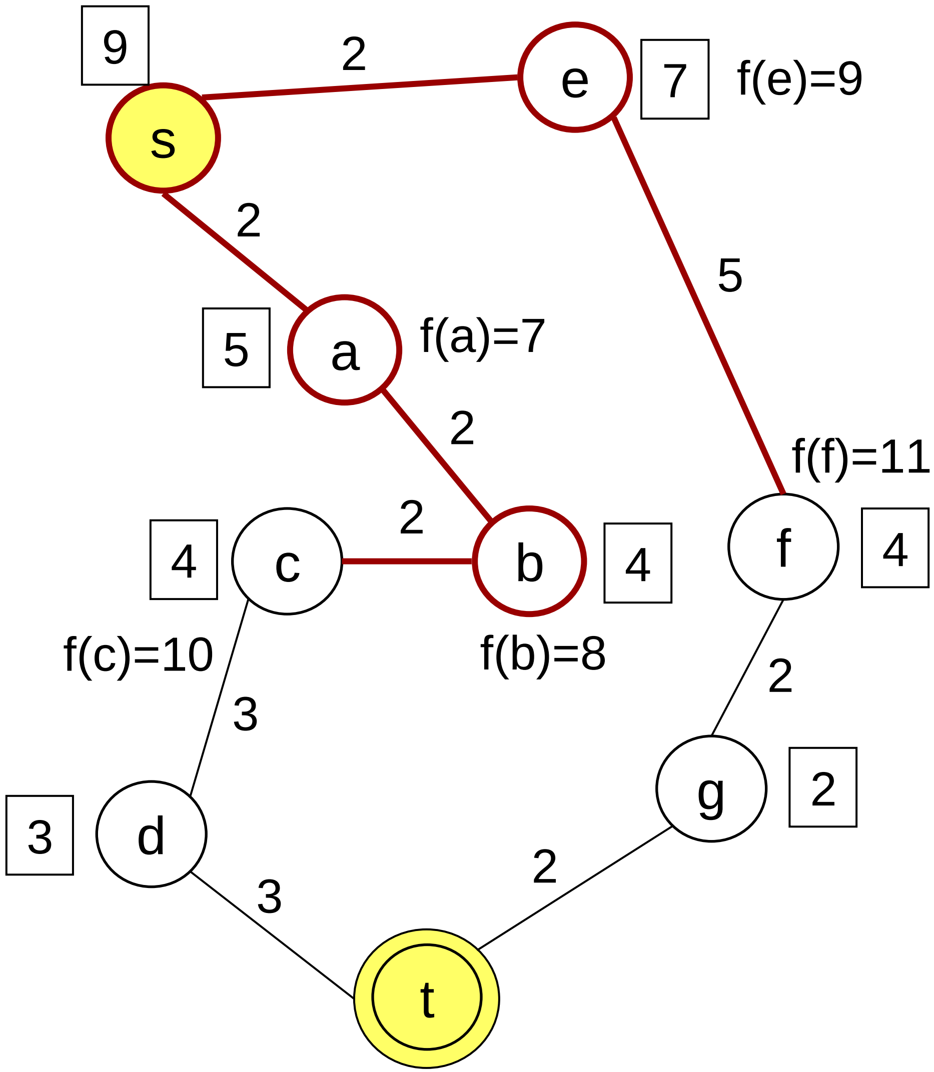
A*

- 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**



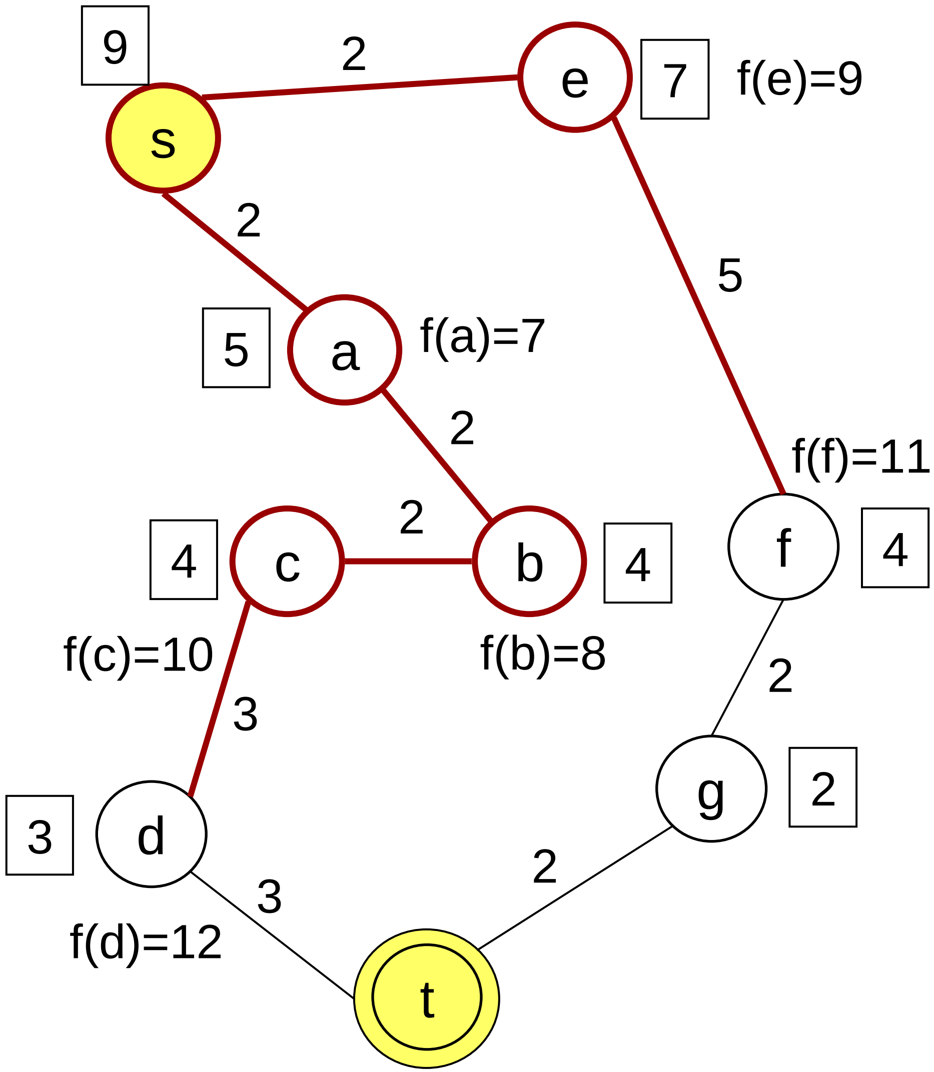
A*

- 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



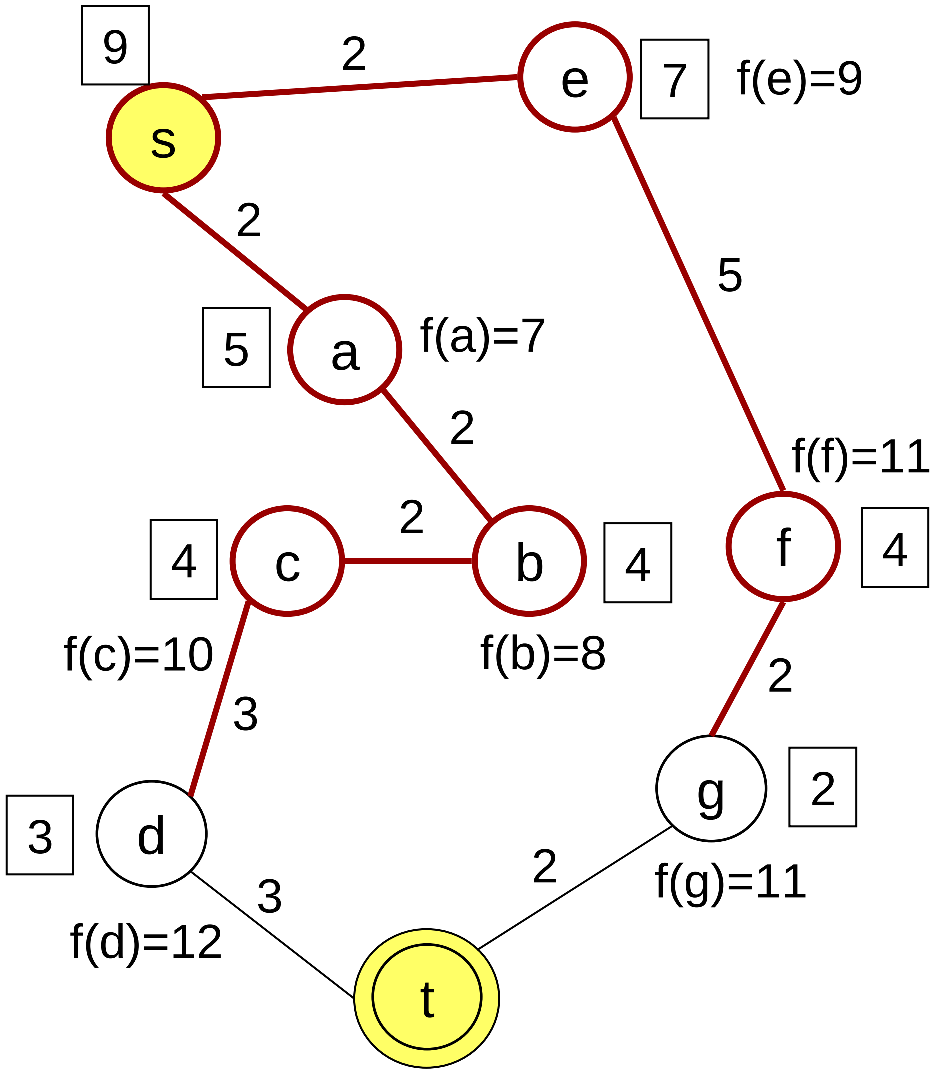
A*

- 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



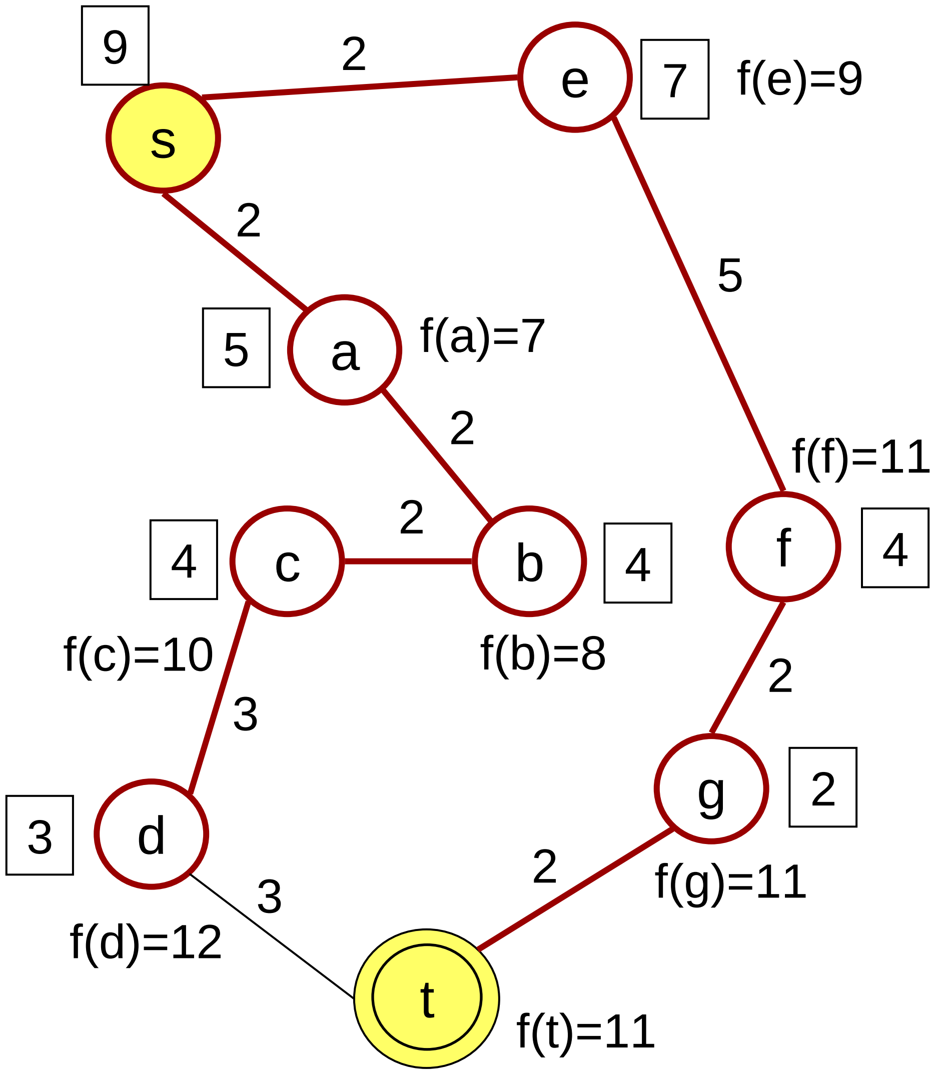
A*

- 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*

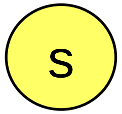
- f(f)=g(f)+dist(f,t)=7+4=11
- Como f(f)>f(c) agora o processo 2 espera e o processo 1 prossegue
- f(d)=g(d)+dist(d,t)=9+3=12
- Como f(d)>f(f) o processo 2 reinicia chegando até o destino **t**
- f(g)=g(g)+dist(g,t)=9+2=11
- f(t)=g(t)+dist(t,t)=11+0=11

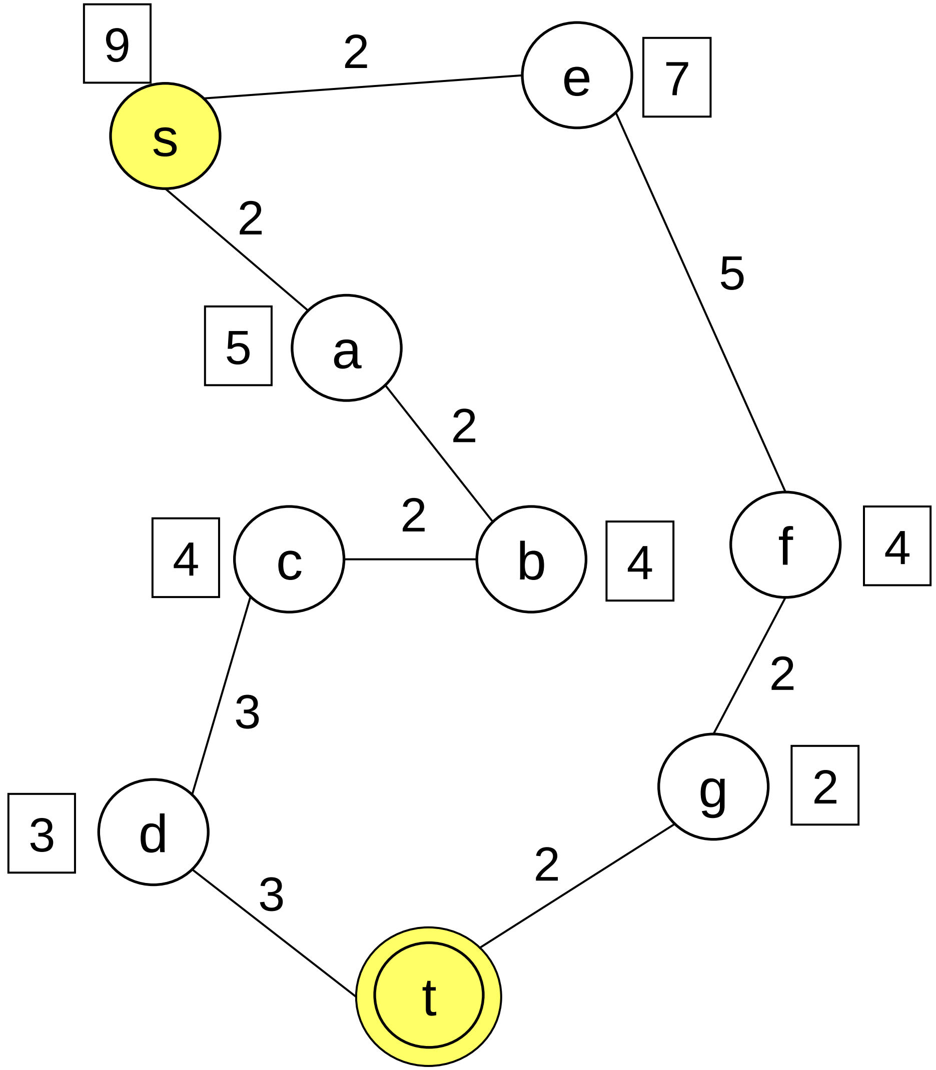


A*

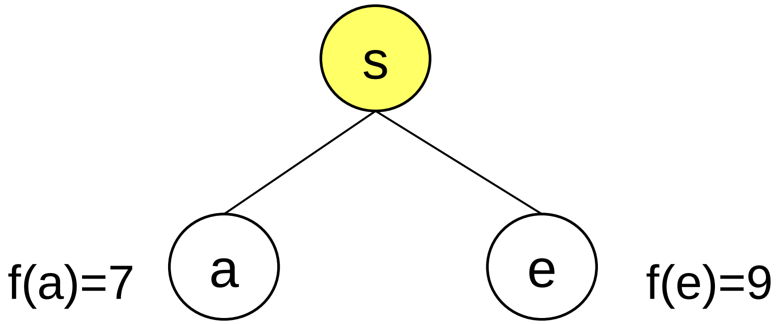
- A busca, começando pelo nó inicial continua gerando novos nós sucessores, sempre expandindo na direção mais promissora de acordo com os valores-f
- Durante este processo, uma árvore de busca é gerada tendo como raiz o nó inicial e o algoritmo A* continua expandindo a árvore de busca até que uma solução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

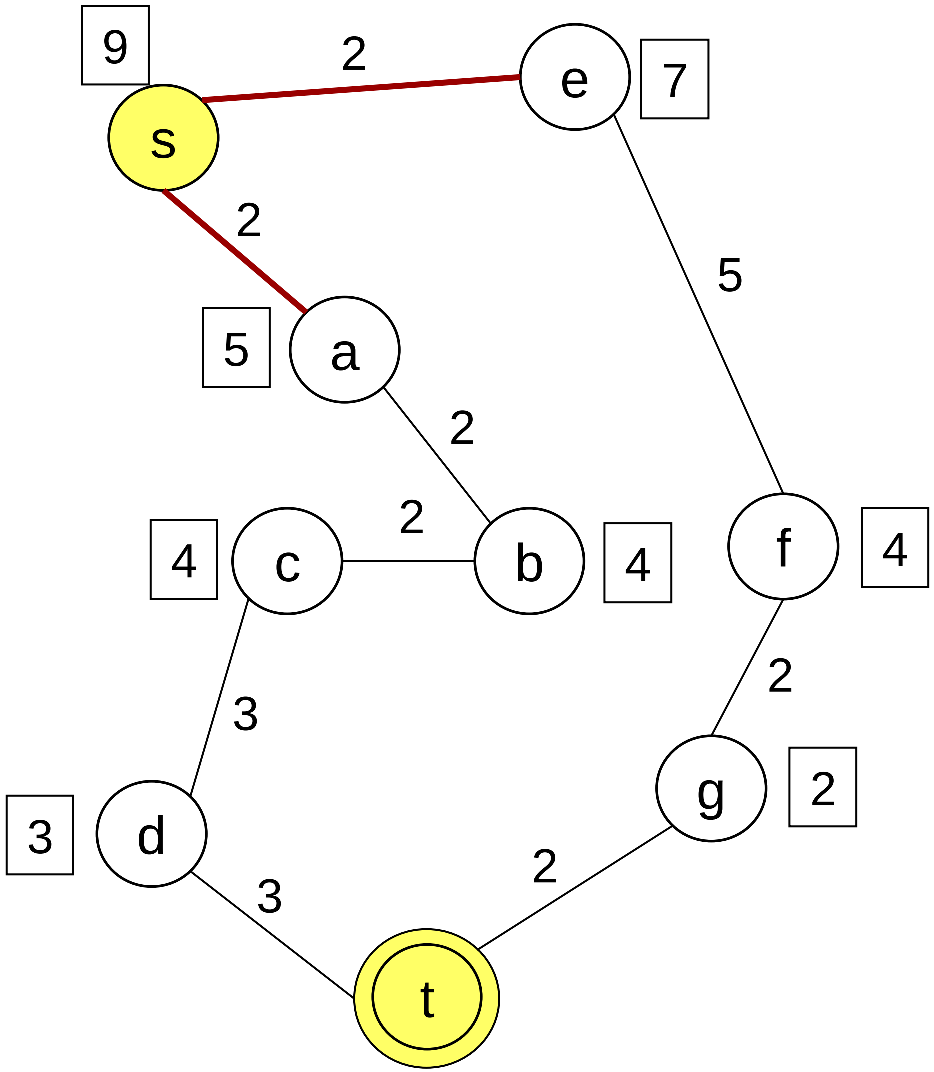
A*



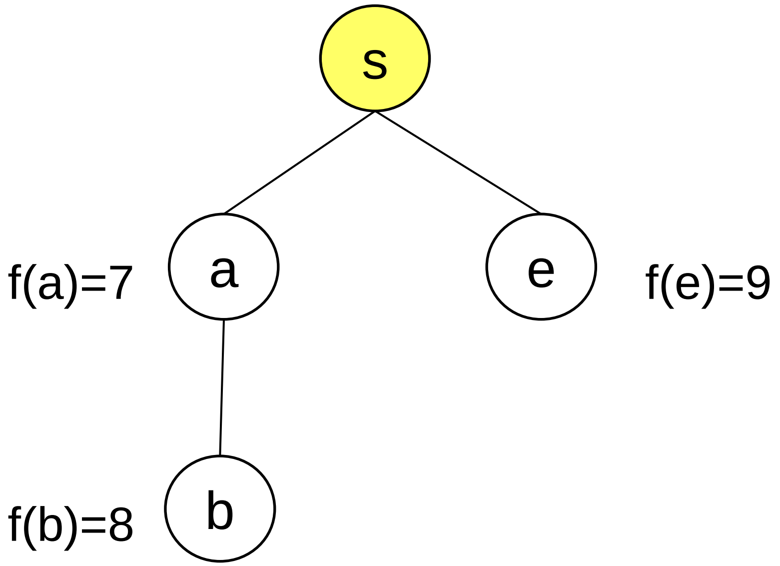


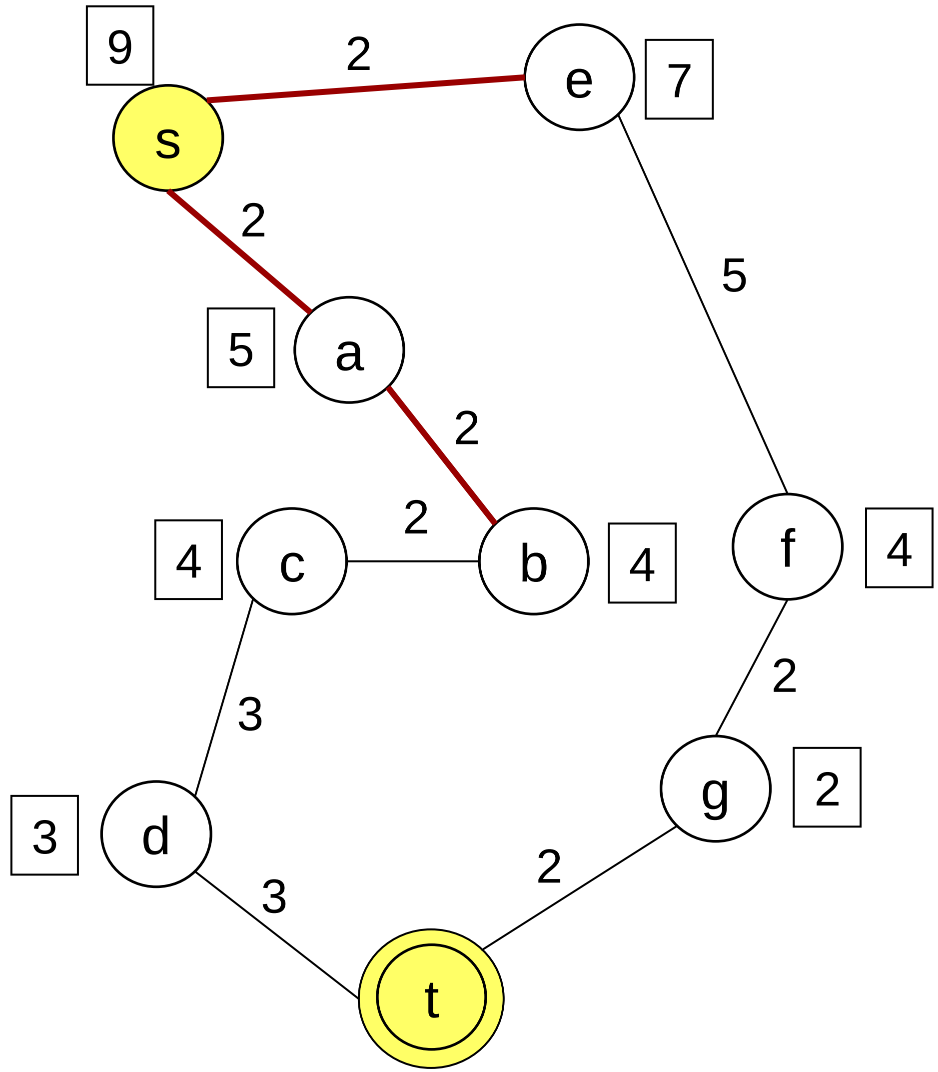
A*



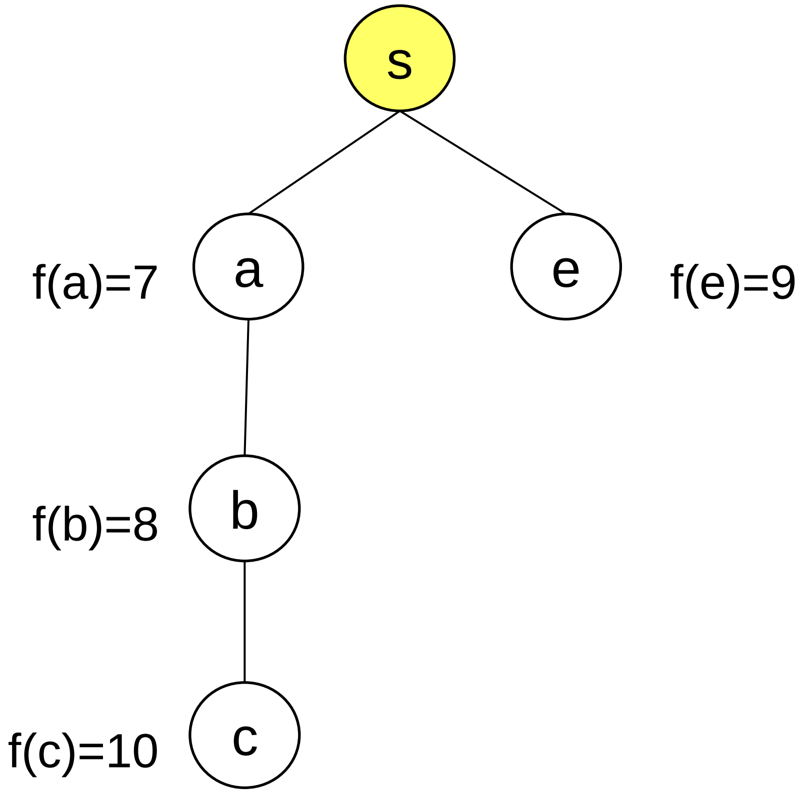


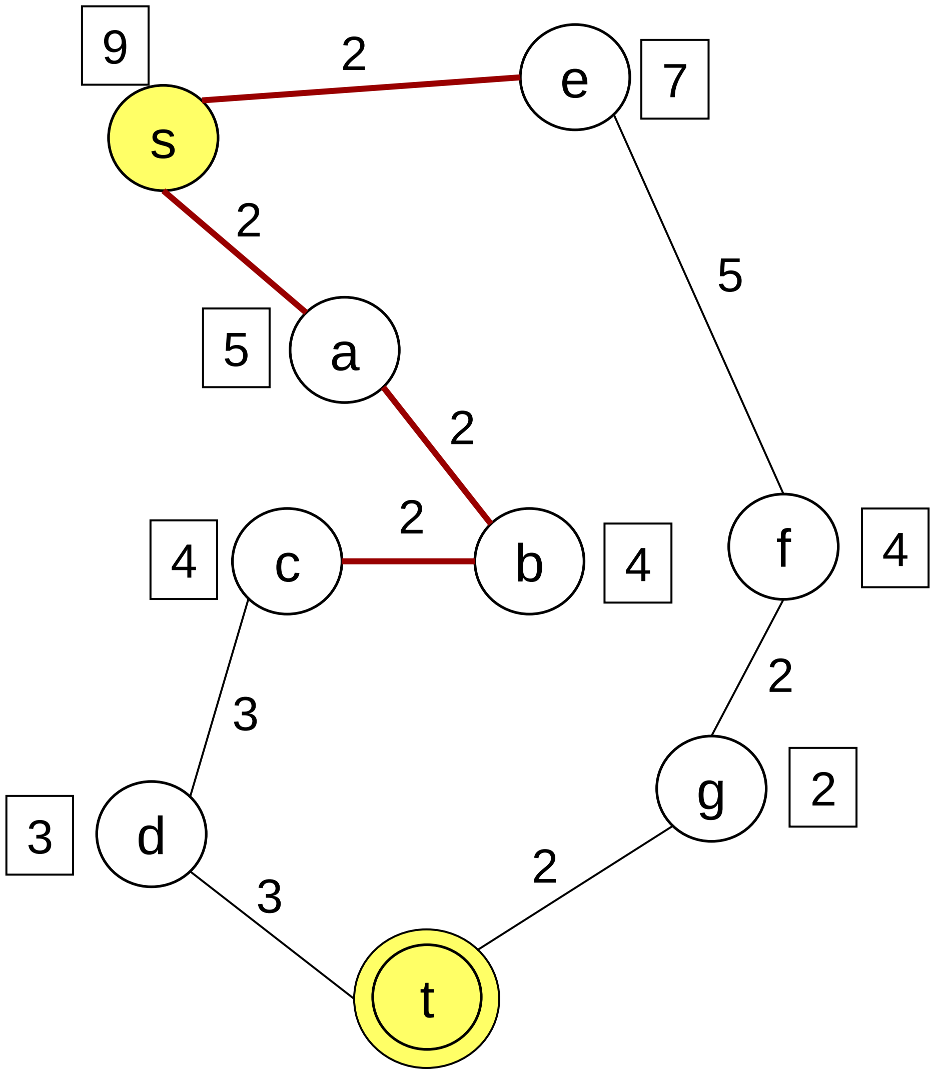
A*



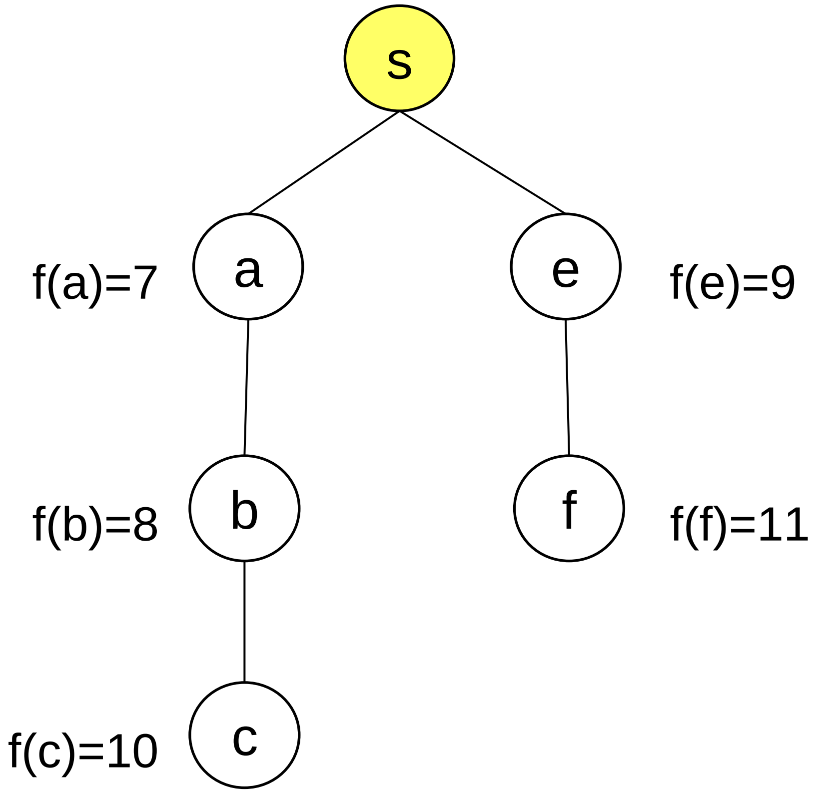


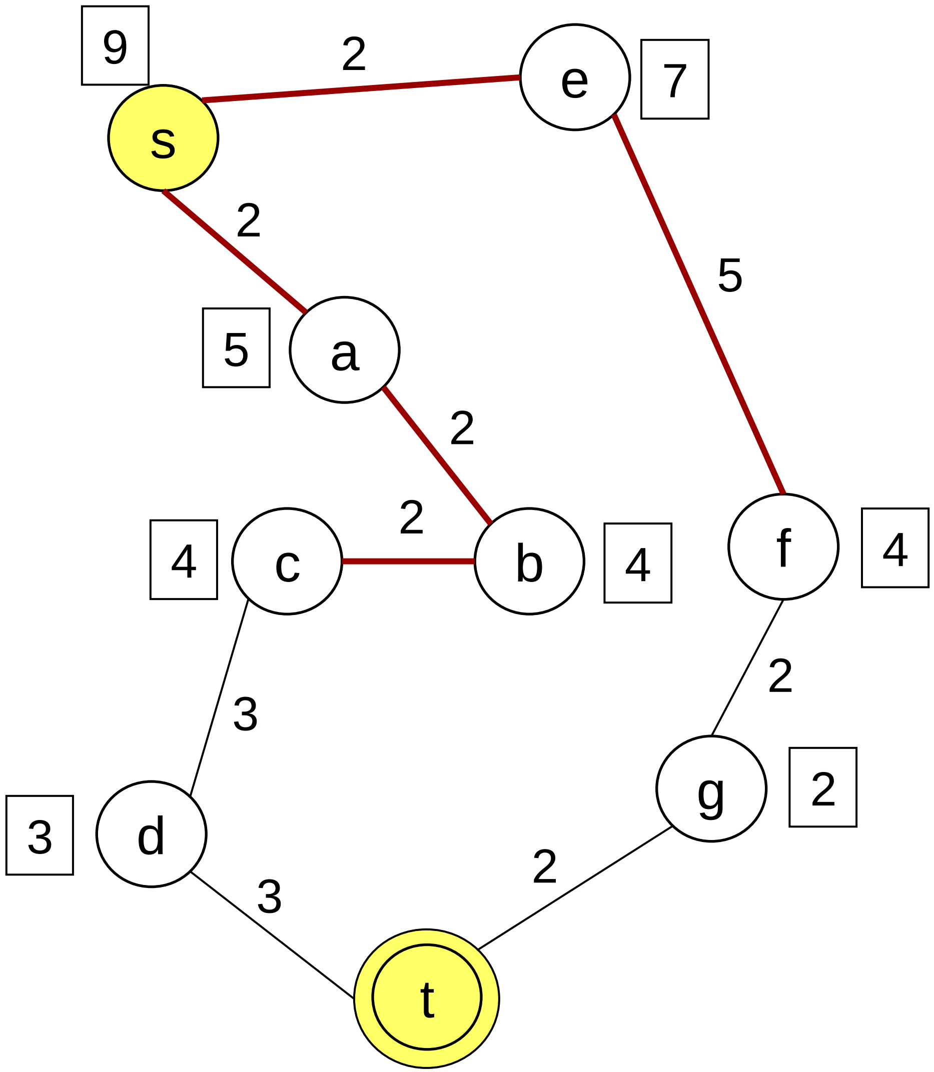
A*



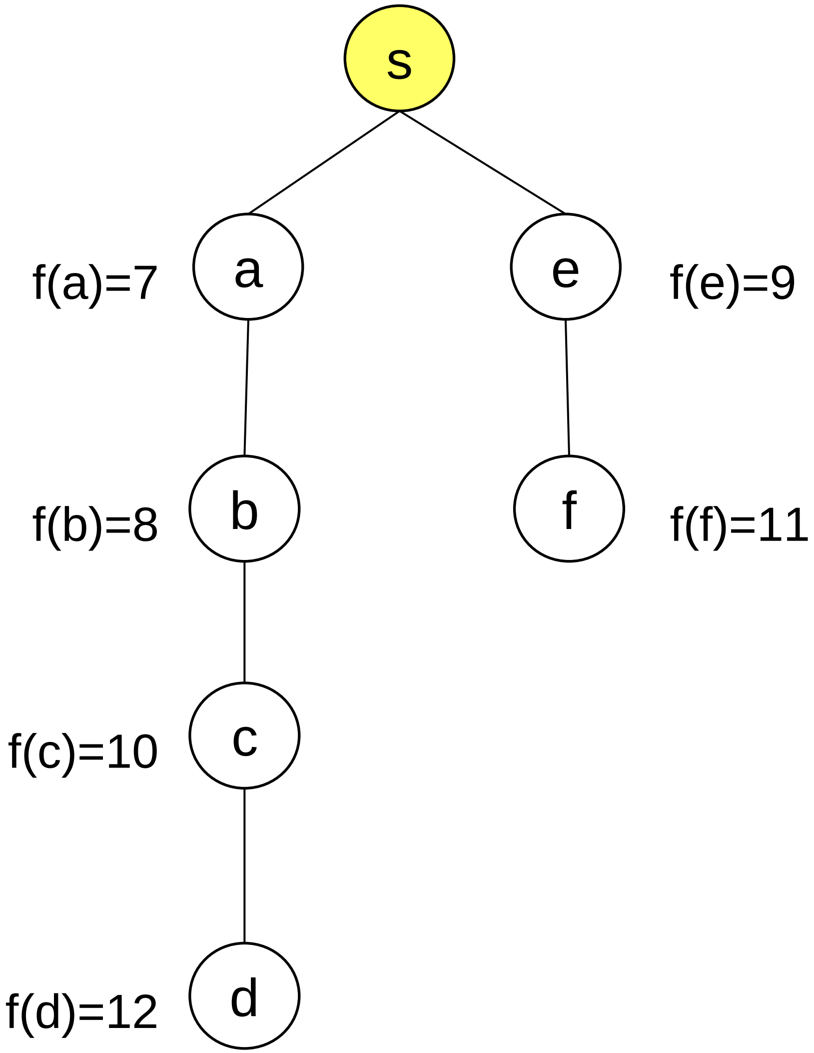


A*

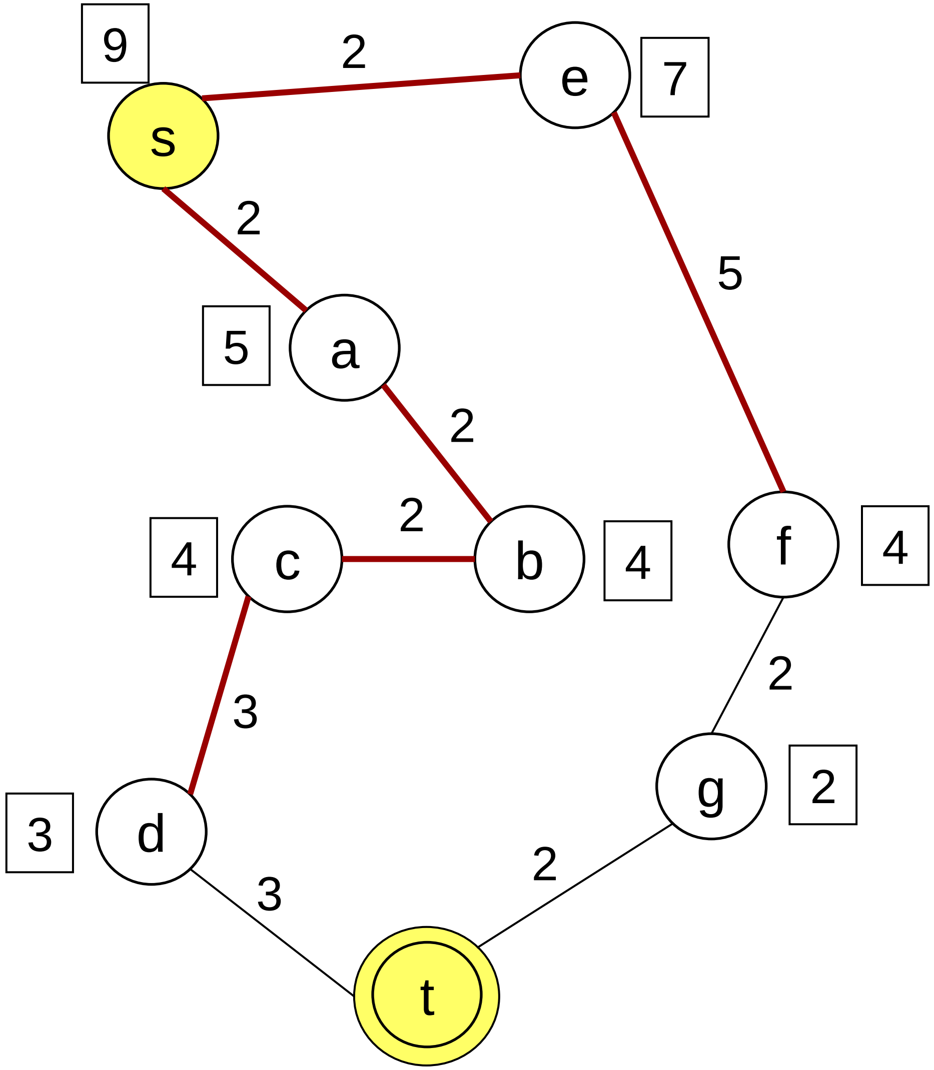




A*

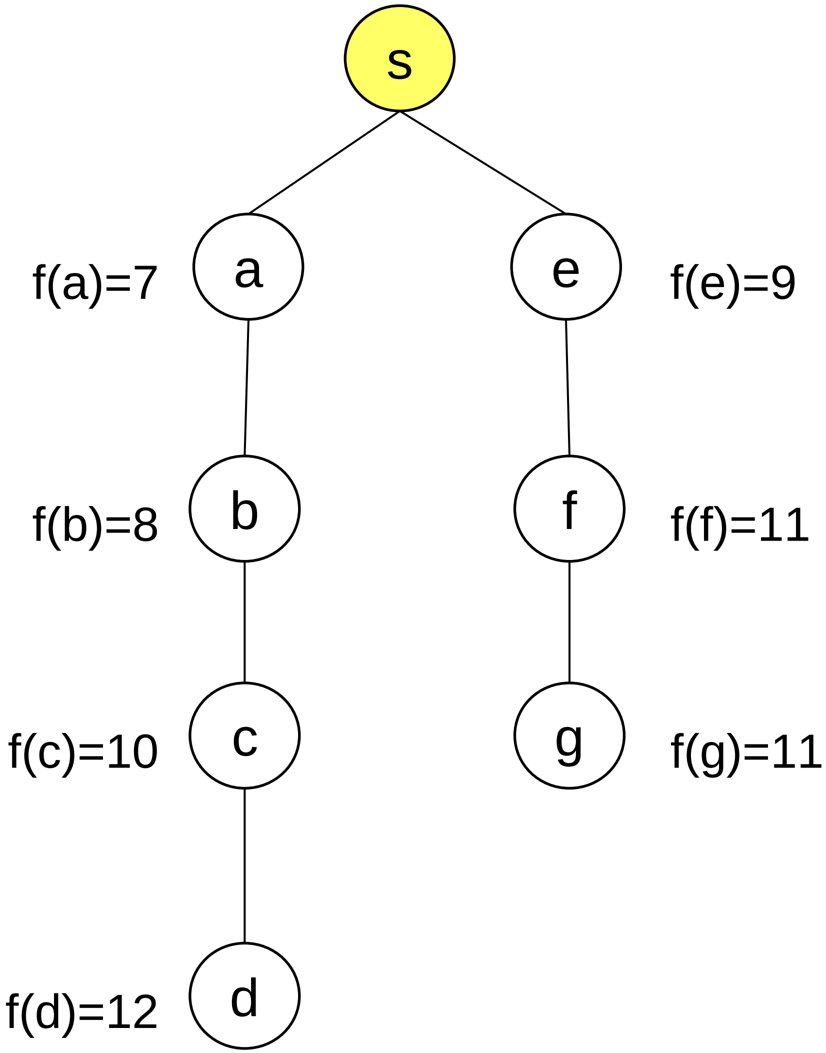


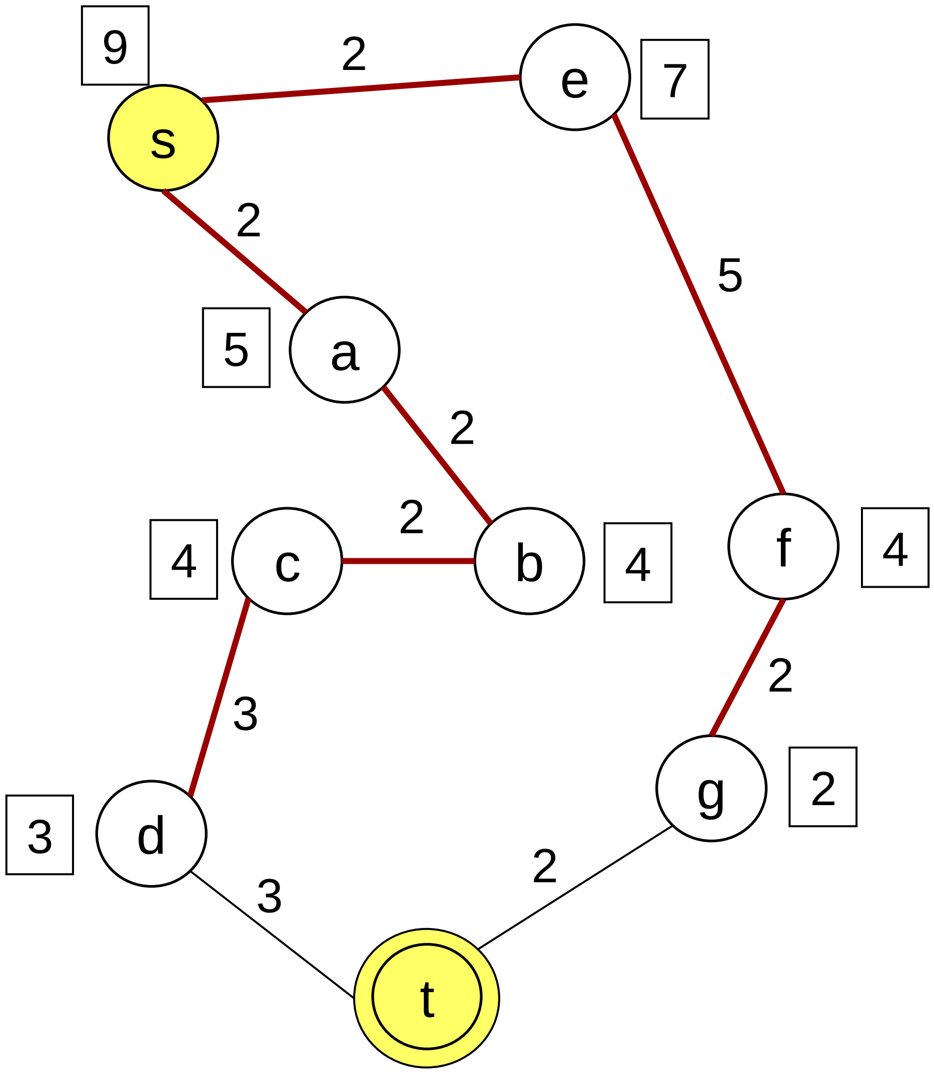
s
a
e
f(a)=7
f(e)=9
b
f
f(b)=8
f(f)=11
c
f(c)=10
d
f(d)=12



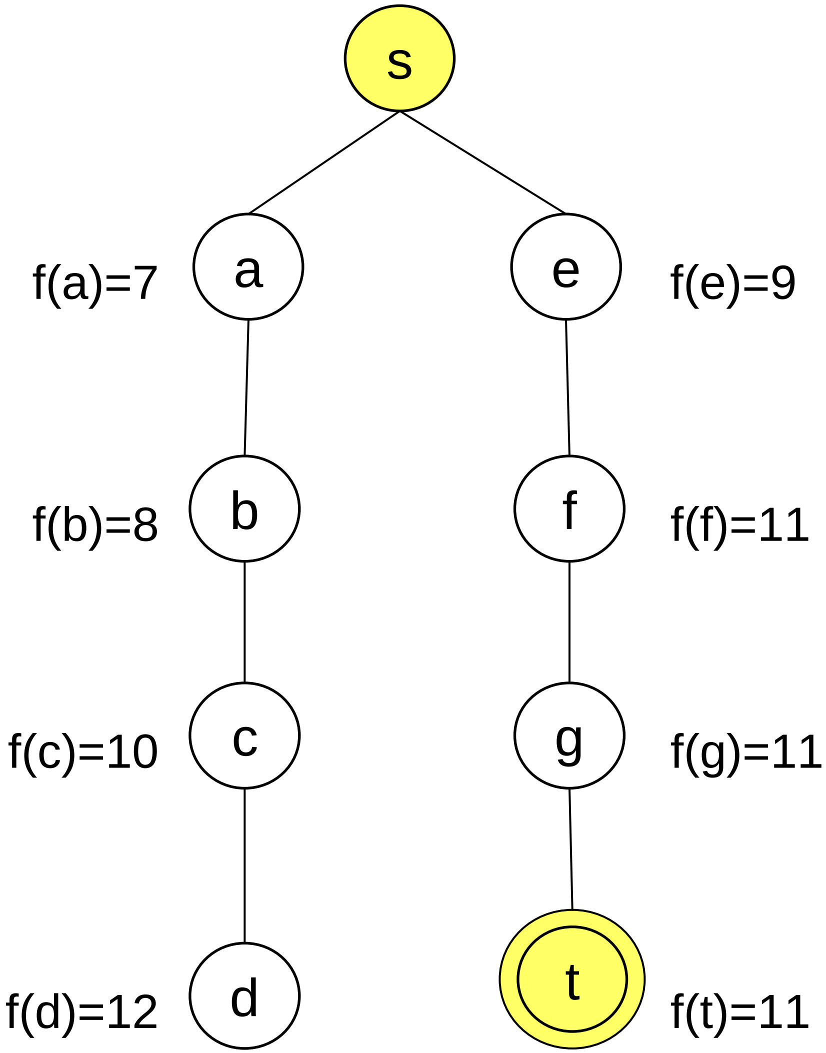
9
s
2
e
7
2
5
5
a
2
2
4
c
b
4
f
4
3
2
g
2
3
d
3
2
t

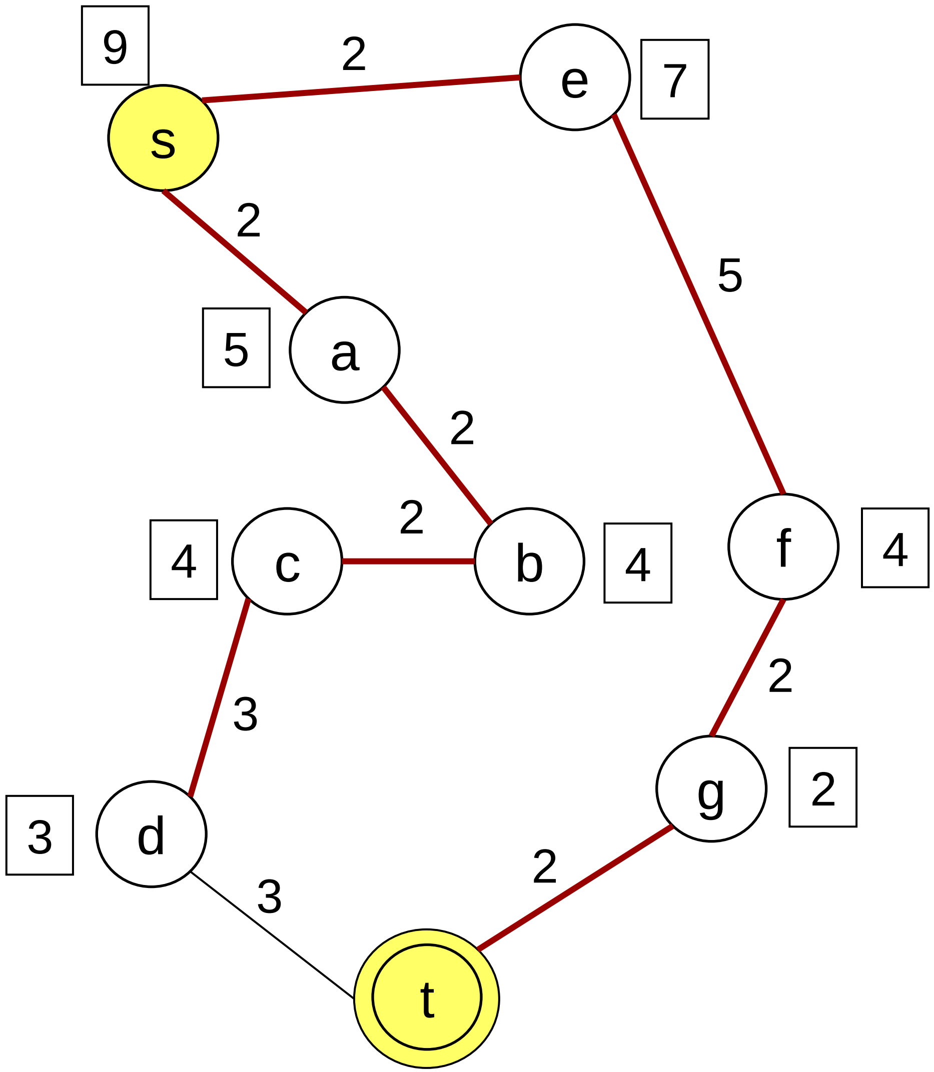
A*


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


9
s
2
e
7
2
5
5
a
2
2
4
c
b
4
f
4
3
2
g
2
3
d
3
2
t

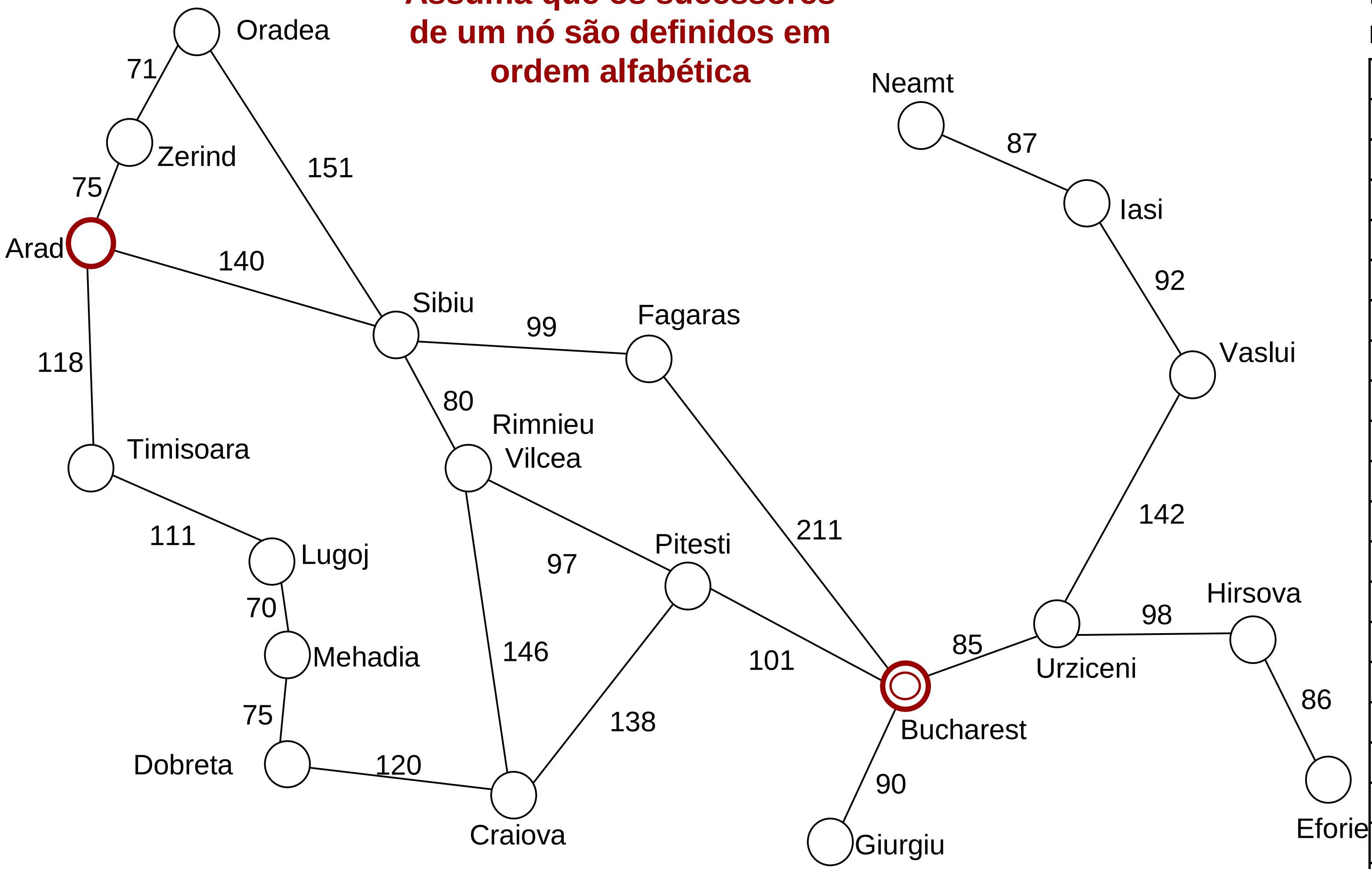
A*


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


9
s
2
e
7
2
5
5
a
2
2
4
c
b
4
f
4
3
2
3
d
g
2
3
2
t

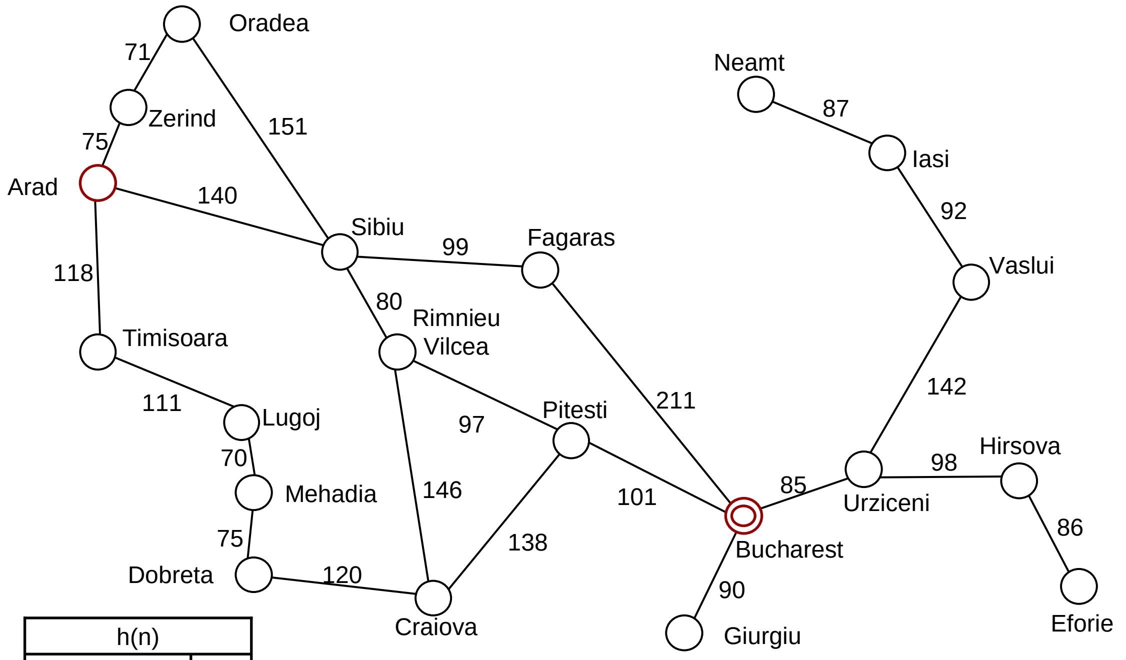
A*: Encontre o caminho de Arad até Bucharest

Assuma que os sucessores de um nó são definidos em ordem alfabética

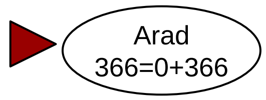


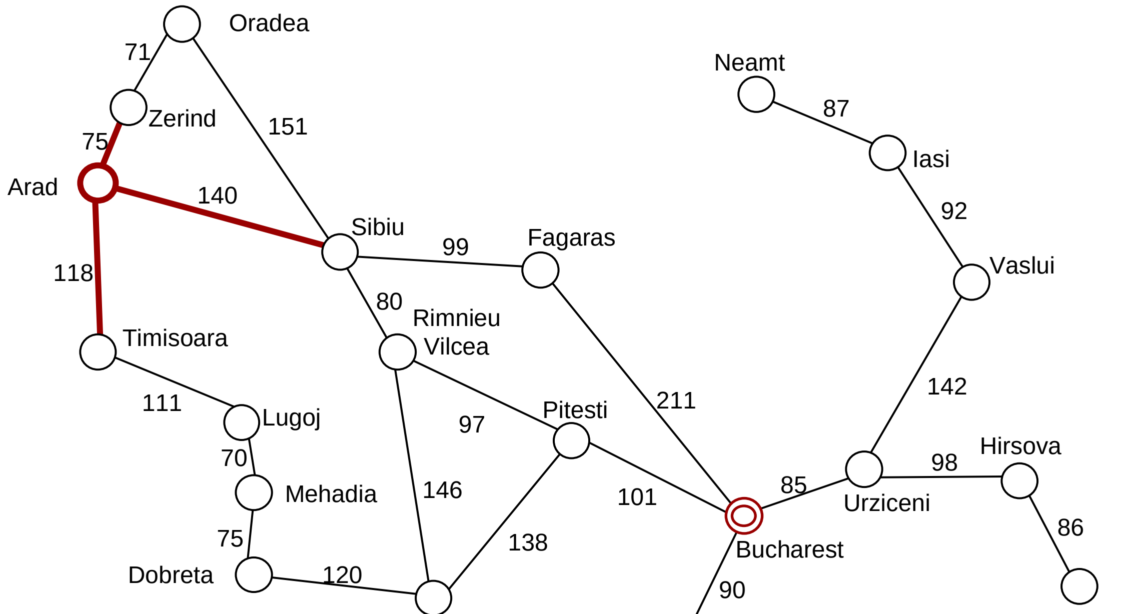
Distância em linha reta até Bucharest

h(n)	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnieu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

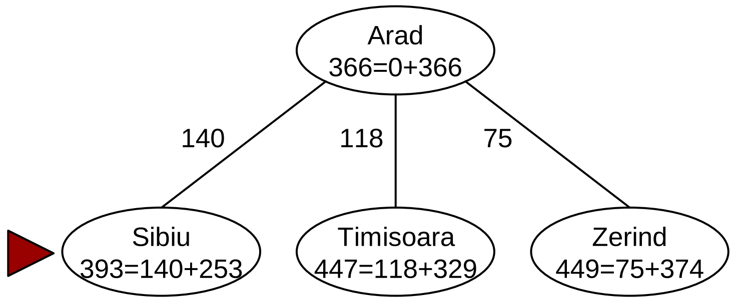


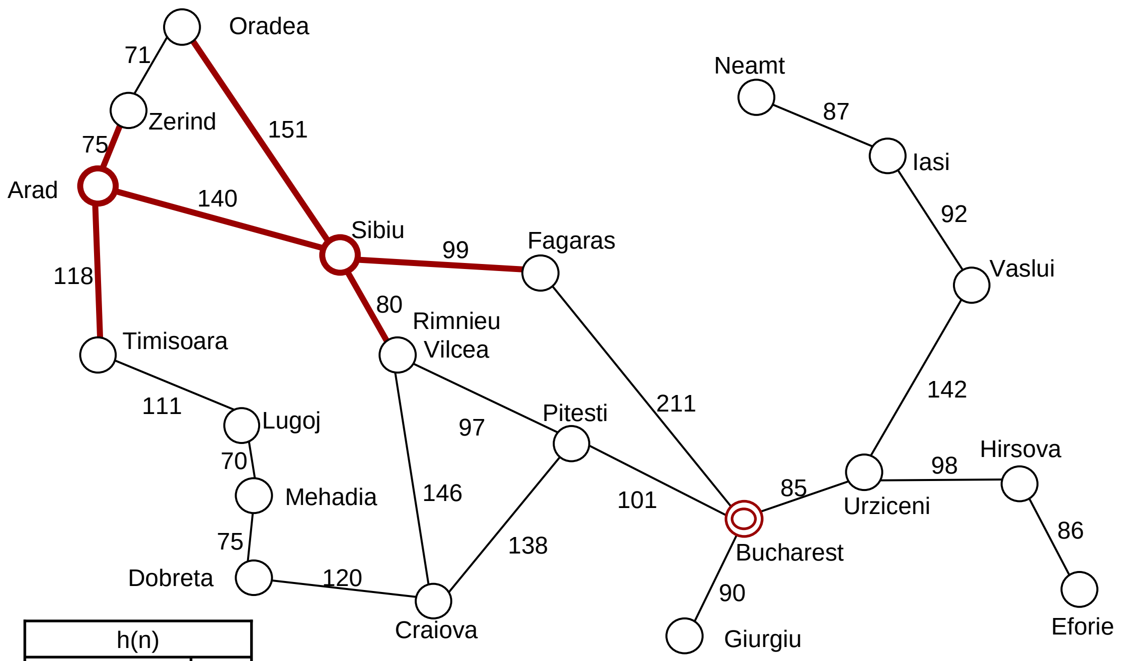
h(n)	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnieu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



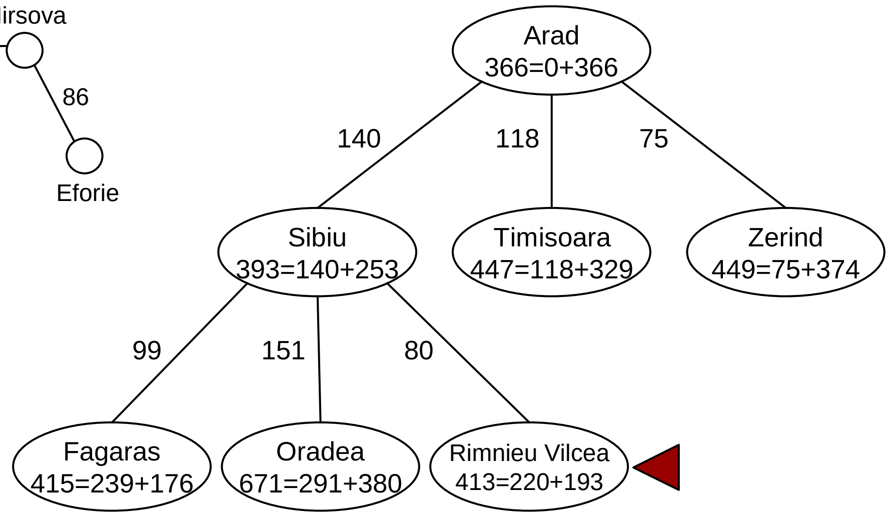


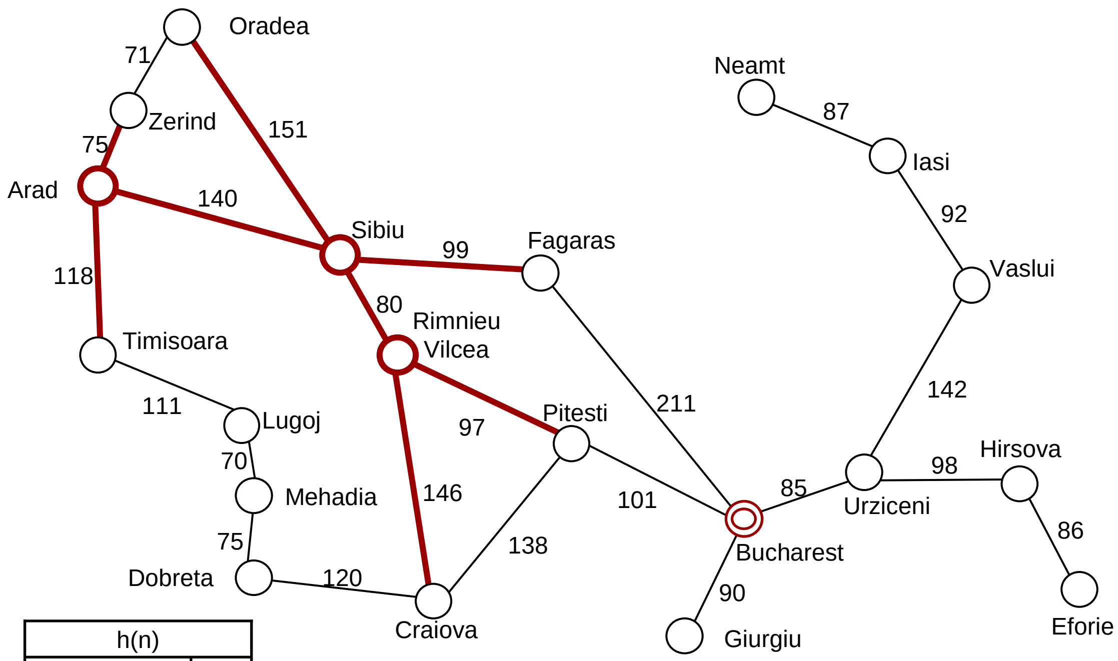
h(n)	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnieu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



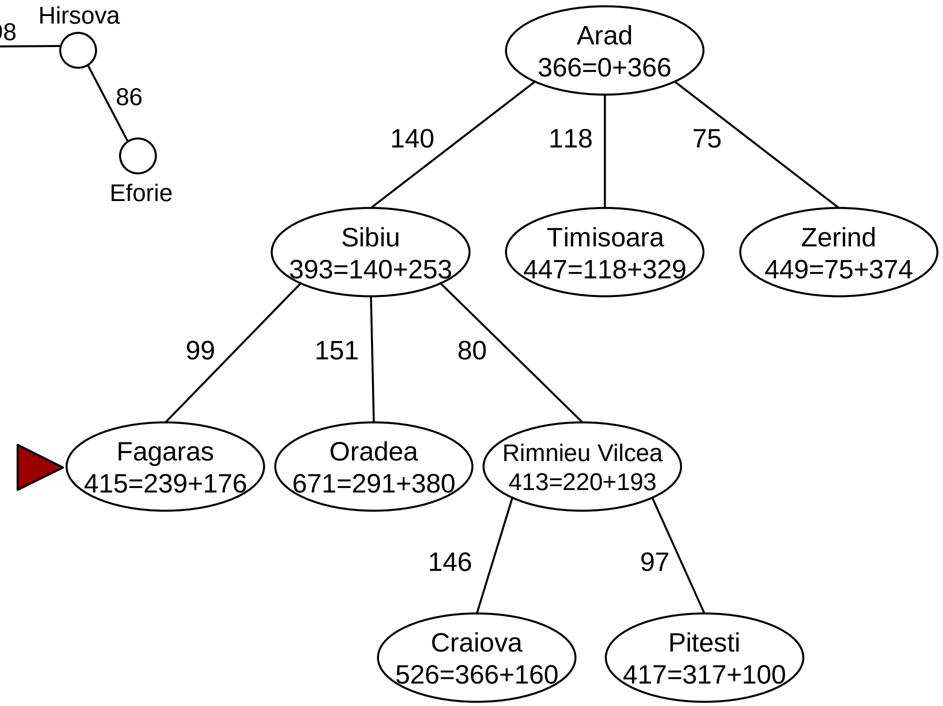


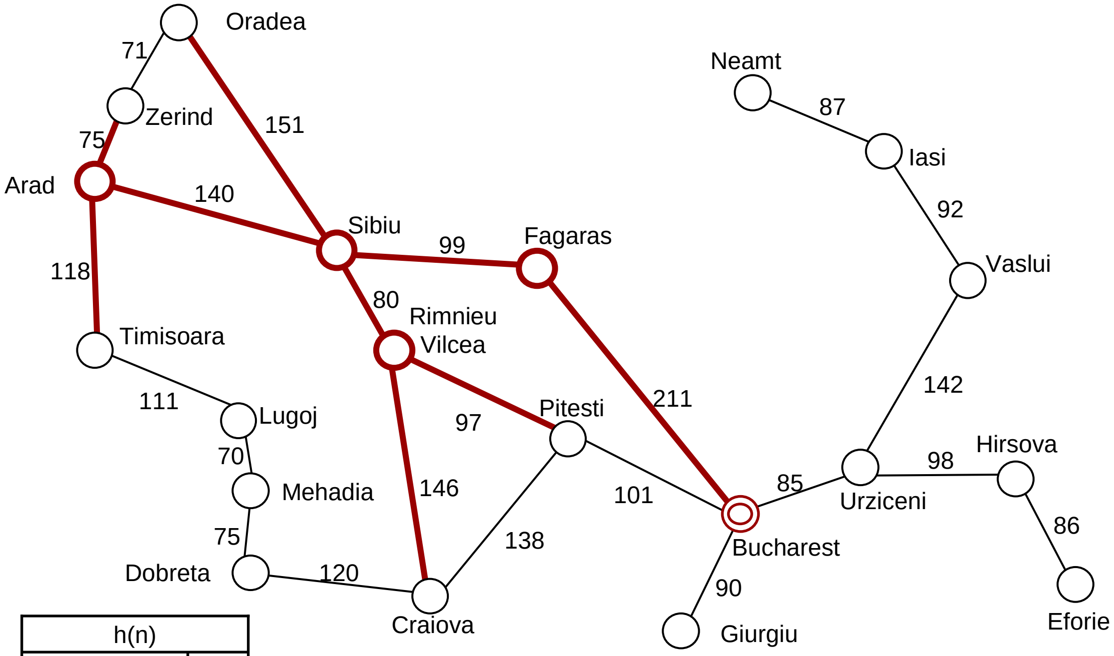
h(n)	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnieu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



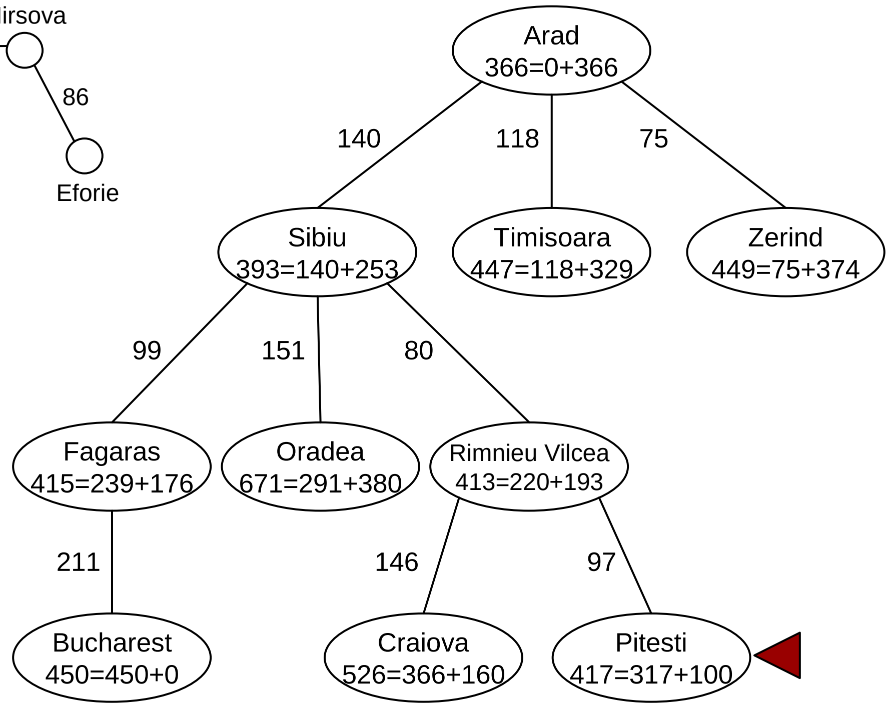


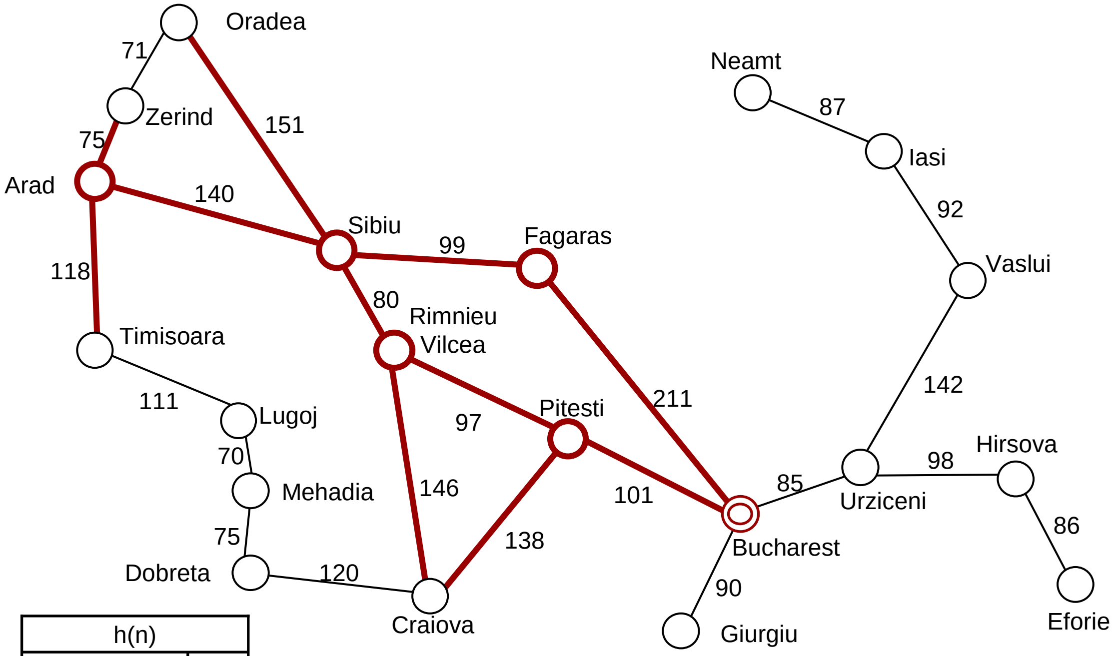
h(n)	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnieu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



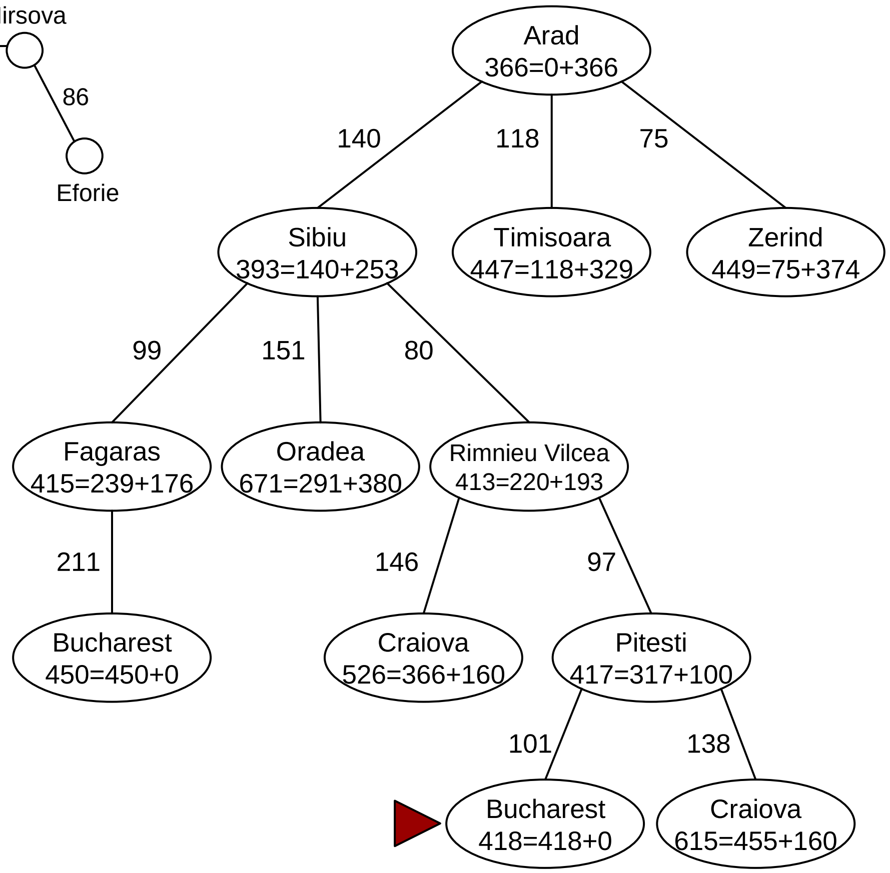


h(n)	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnieu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

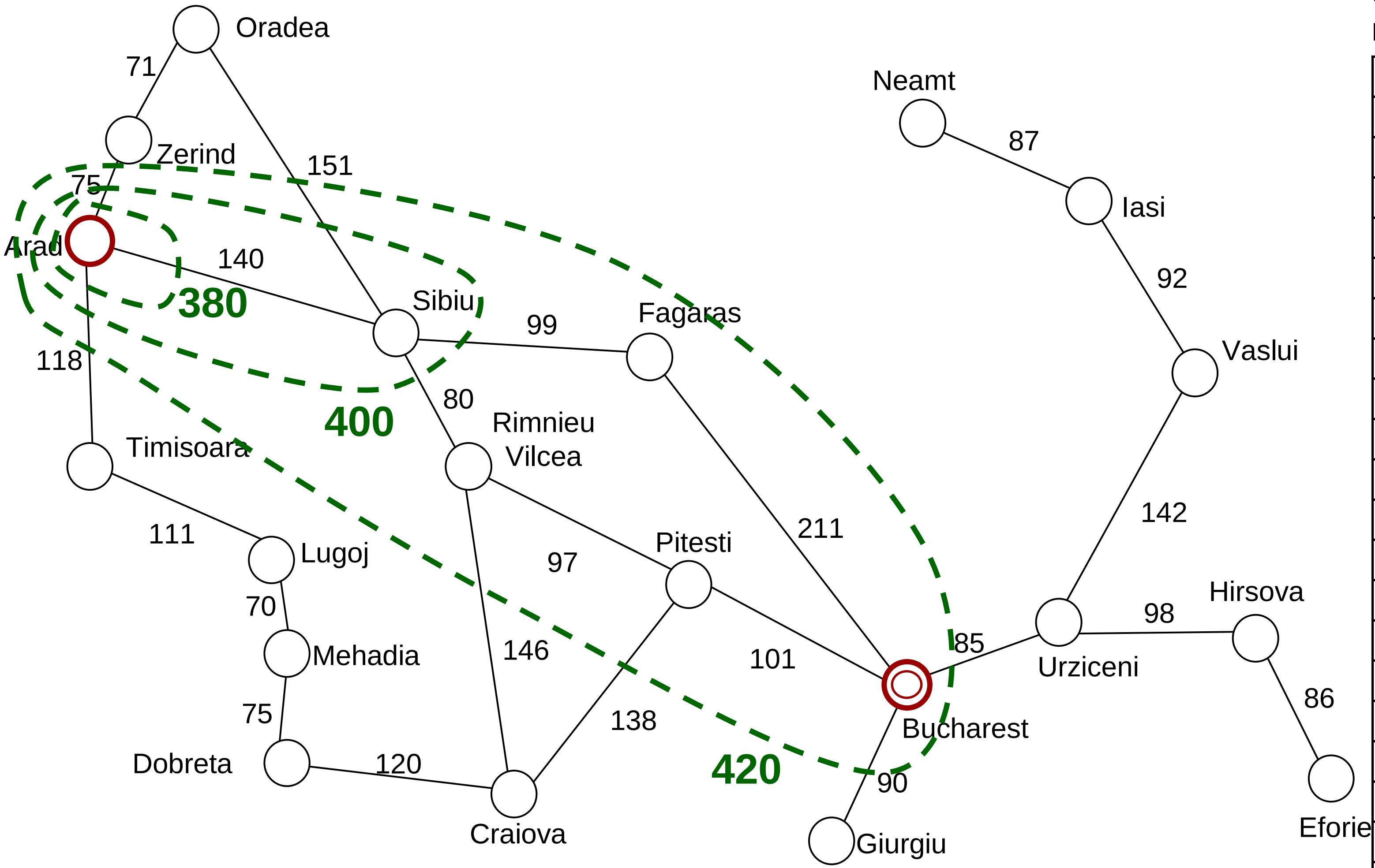




h(n)	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnieu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



A*: Contornos em f=380, f=400 e f=420 (nós no interior do contorno têm valores-f menores ou iguais aos do contorno)



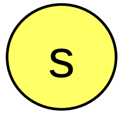
Distância em linha reta até Bucharest

h(n)	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnieu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A*

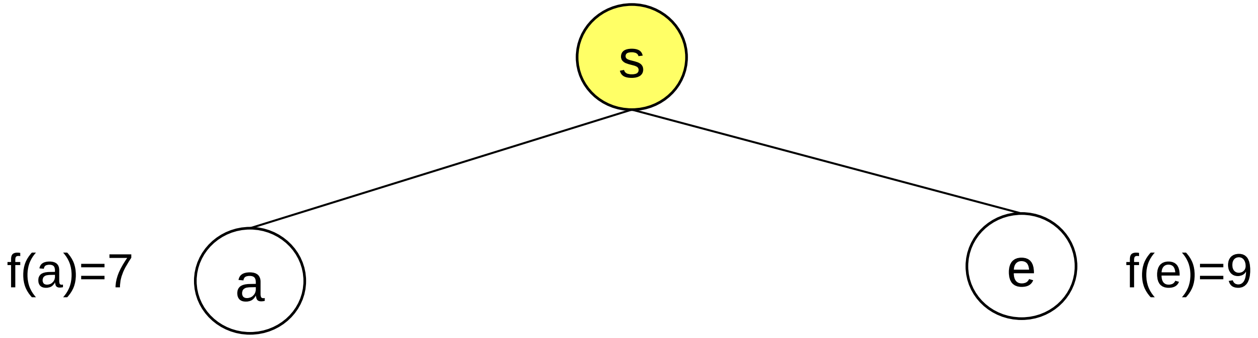
- A árvore de busca será representada de duas formas:
  - l(N,F/G) representa um único nó folha (*leaf*)
    - N é um nó do espaço de estados
    - G é g(N), custo do caminho encontrado desde o nó inicial até N
    - F é f(N) = G + h(N)
  - t(N,F/G,Subs) representa uma árvore com sub-árvores não vazias
    - 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

A*



l(s,0/0)

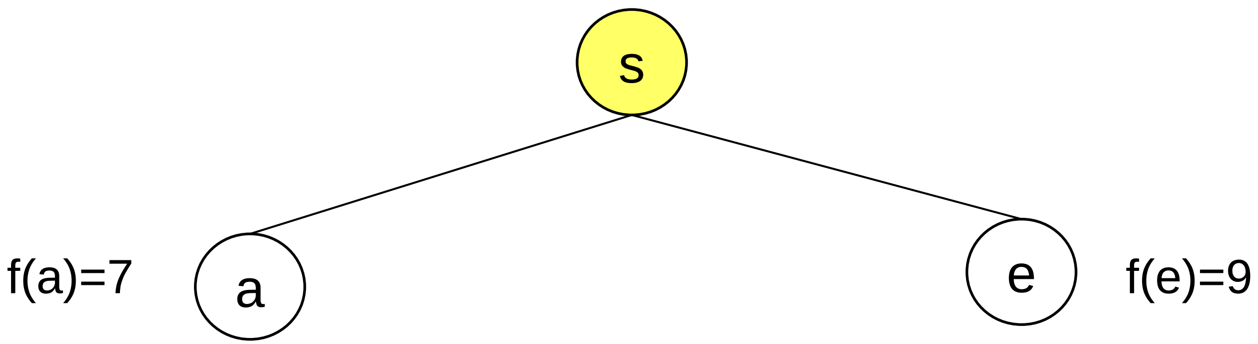
A*



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**

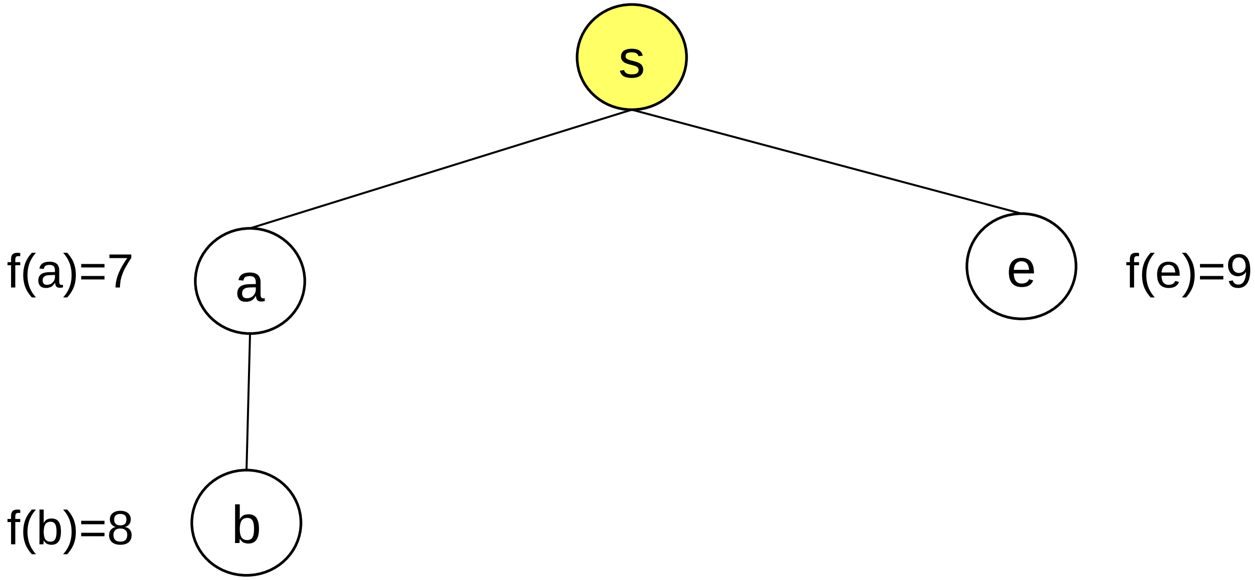
A*



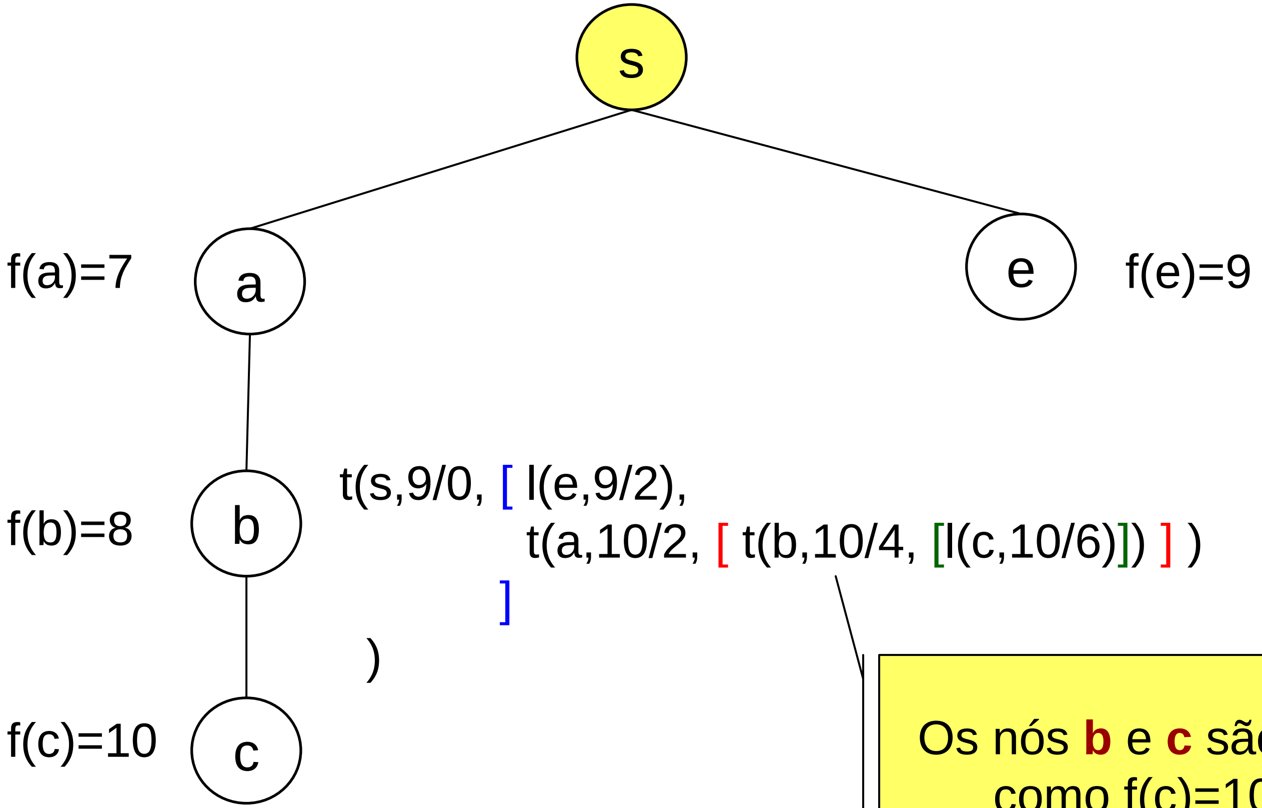
t(s,7/0, [l(a,7/2), l(e,9/2)])

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

A*



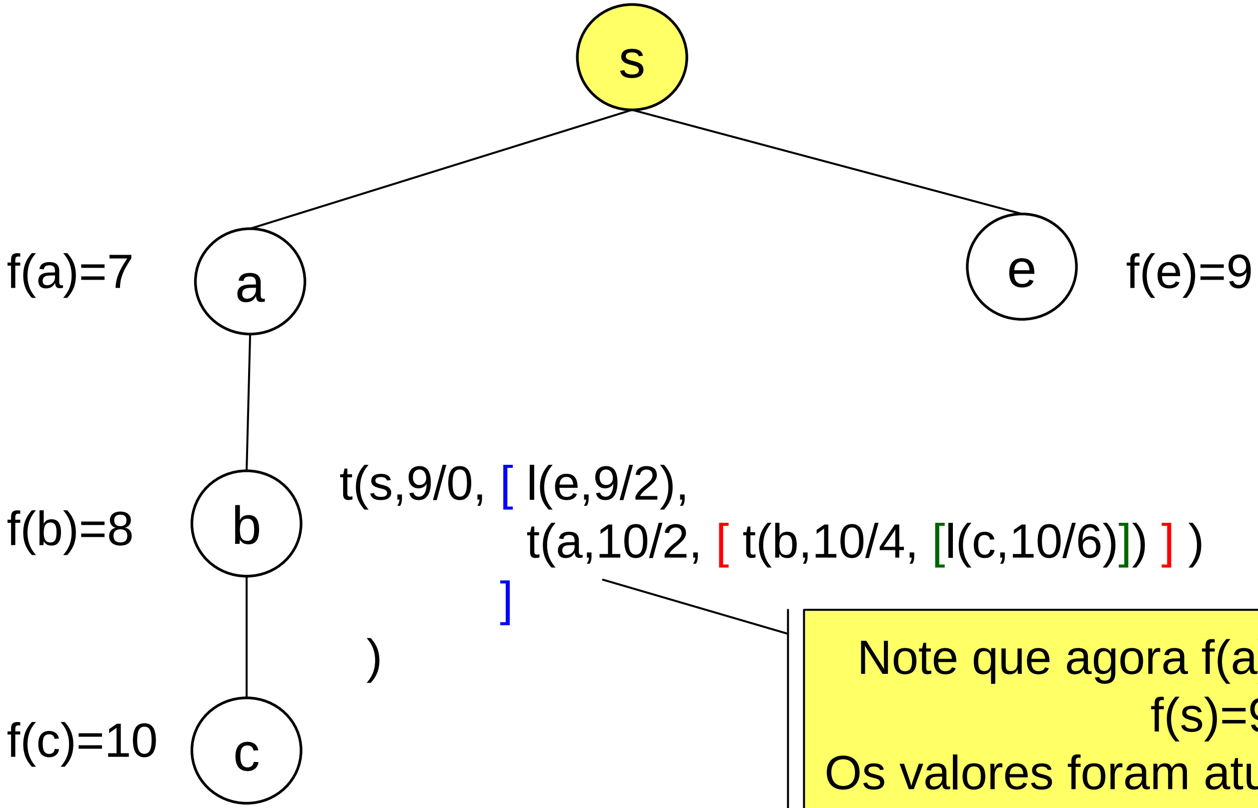
A*




```
t(s,9/0, [ l(e,9/2),
           t(a,10/2, [ t(b,10/4, [l(c,10/6)]) ] )
         ]
  )
```

Os nós **b** e **c** são expandidos e como f(c)=10 o limite de expansão é atingido e então a sub-árvore via **a** não tem mais permissão para expandir

A*



t(s,9/0, [l(e,9/2),
t(a,10/2, [t(b,10/4, [l(c,10/6)])])
]
)

Note que agora f(a)=10 enquanto f(s)=9
Os valores foram atualizados devido ao fato que novos nós, **b** e **c**, foram gerados
Agora o sucessor mais promissor de **s** é **e** com f(e)=9

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

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
  - $f(T) = \min f(S_i) \qquad 1 <= i <= k$

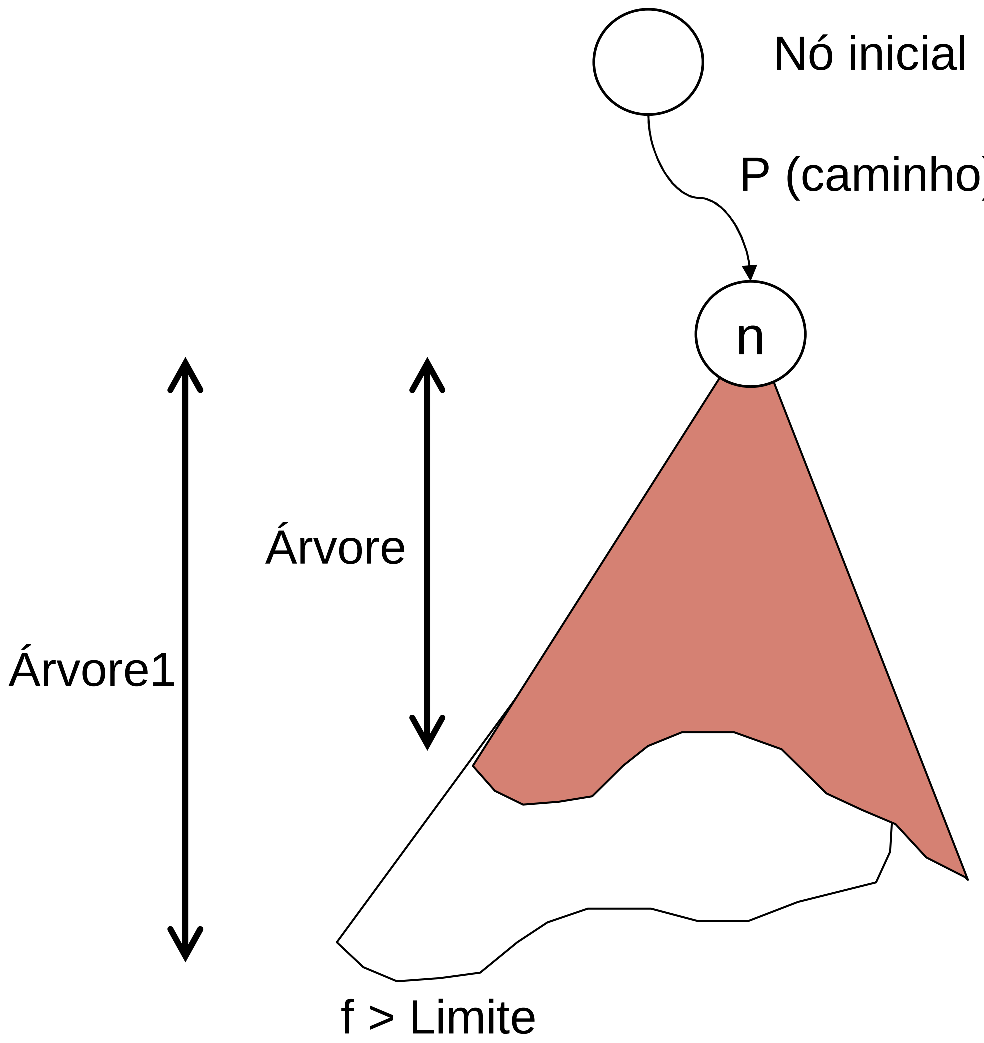
A*

- 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)

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



Expandindo Árvore até que seu valor-f exceda Limite resulta em Árvore1

Algoritmo A*

```
% Assuma que 9999 é maior que qualquer valor-f
resolvai1(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 A* (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 A* (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 A* (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).
```

Admissibilidade de 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 por meio 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
- 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

Admissibilidade de 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(**n**)=0 comporta-se de forma similar à busca em largura e igual à busca de custo uniforme
  - 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)

Admissibilidade de 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:
  - $h(n) = \text{máximo } \{h_1(n), h_2(n), \ldots, h_m(n)\}$
- De maneira ideal, se h* é conhecida, podemos utilizar h* diretamente
  - A* utilizando h* encontra uma solução ótima diretamente, sem nunca precisar realizar *backtracking*

A*: Função de Avaliação

- A função heurística h é monotônica (consistente) se
  - h(n) >= h(sucessor(n))
    - isso se aplica à maioria das funções heurísticas
    - Toda heurística consistente é admissível
- Transferindo esse resultado para a função de avaliação *f=g+h*:
  - f(sucessor(n)) >= f(n)
    - uma vez que g é não decrescente
- Em outras palavras, o custo de cada nó gerado no mesmo caminho nunca diminui
- Se *h* é não monotônica, para se garantir a monotonicidade de *f*:
  - quando *f(suc(n)) < f(n)*
  - usa-se *f(suc(n)) = max(f(n), g(suc(n))+h(suc(n)))*

Complexidade de A*

- A utilização de heurística para guiar o algoritmo A* 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 $O(b^d)$
  - 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*

- **A* é o algoritmo de busca mais rápido**, ou seja, para uma dada heurística, nenhum outro algoritmo pode expandir menos nós que A*
- A velocidade de A* depende da qualidade da heurística
  - Se a heurística é desprezível (por exemplo, h(n)=0 para todo n) o algoritmo degenera para a busca de custo uniforme
  - Se a heurística é perfeita (h=h*) não há busca de fato; o algoritmo marcha diretamente até o objetivo
- Geralmente, os problemas reais se encontram entre as duas situações acima e, portanto, o tempo de execução de A* vai depender da qualidade da heurística

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)

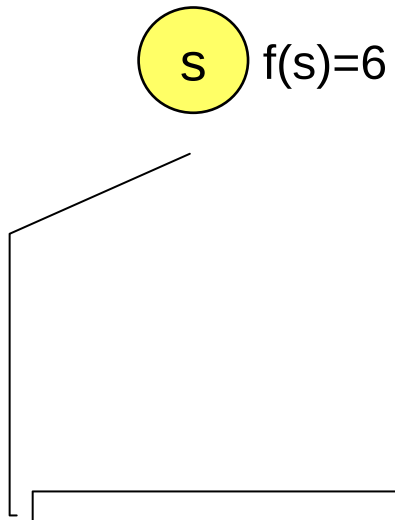
IDA*

- 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
```

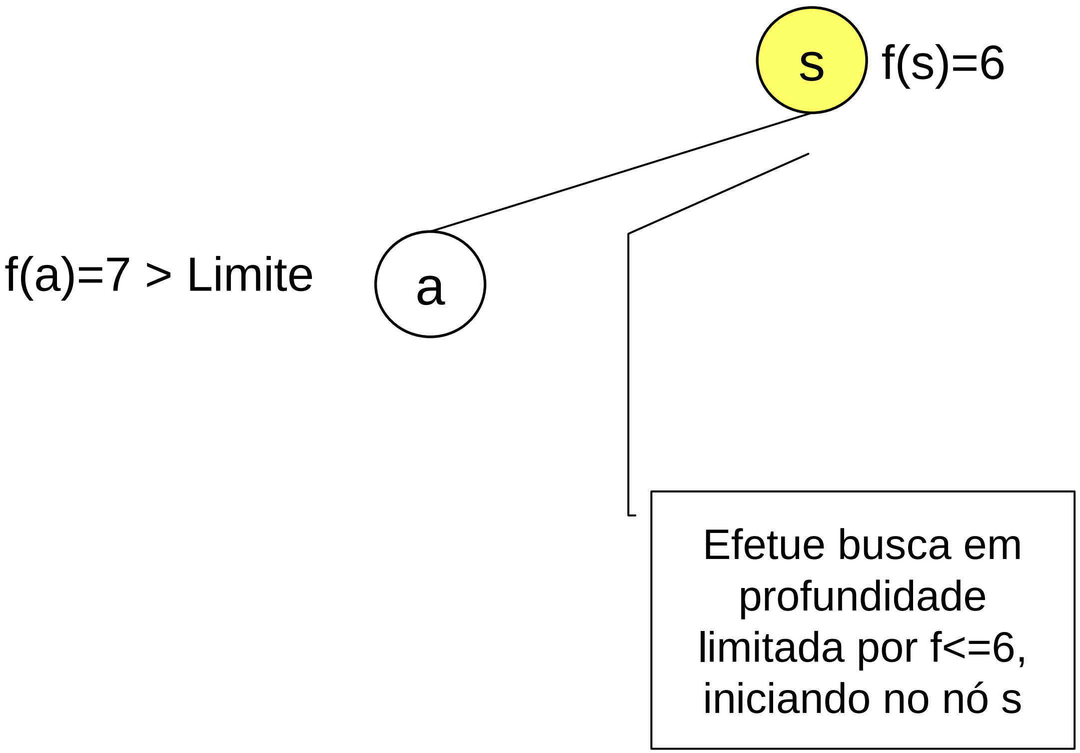
IDA*



Efetue busca em profundidade limitada por f<=6, iniciando no nó s

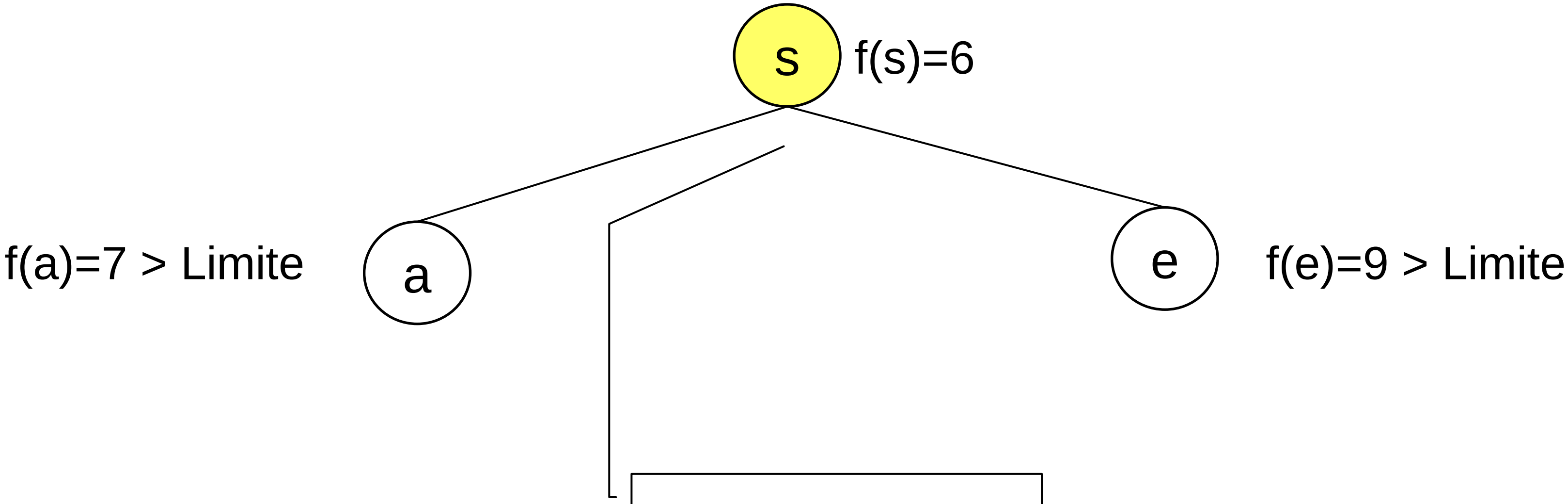
Limite = 6

IDA*


s
f(s)=6
f(a)=7 > Limite
a
Efetue busca em profundidade limitada por f<=6, iniciando no nó s

Limite = 6

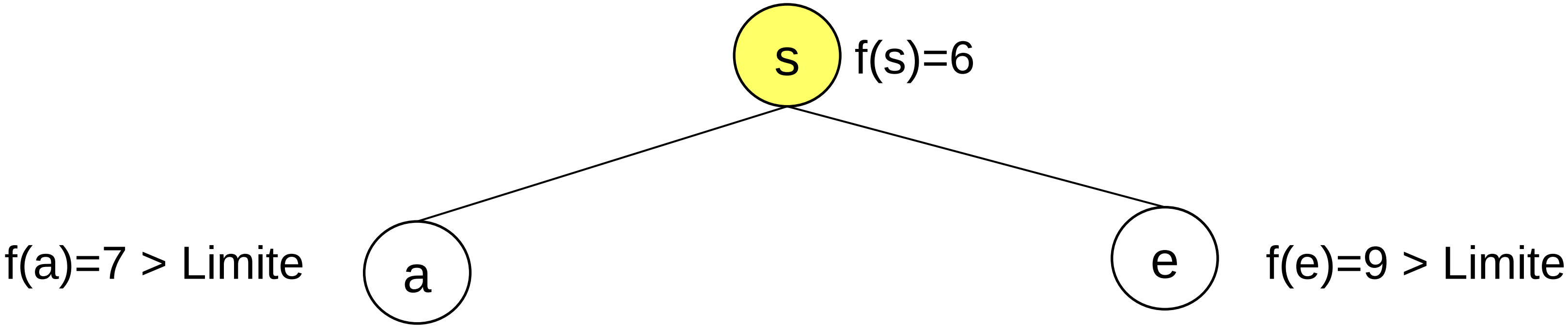
IDA*



Efetue busca em profundidade limitada por f<=6, iniciando no nó s

Limite = 6

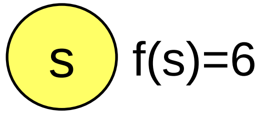
IDA*



NovoLimite = mín{7,9} = 7
Efetue busca em profundidade limitada por f<=7, iniciando pelo nó s

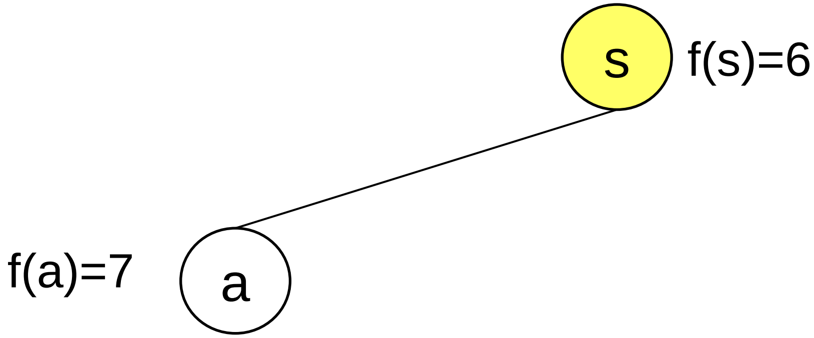
Limite = 7

IDA*



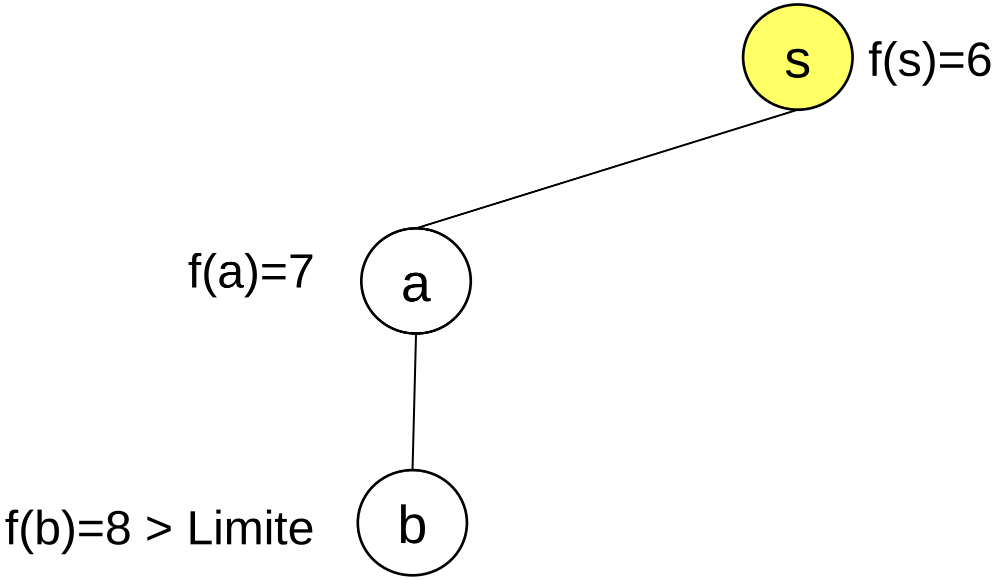
Limite = 7

IDA*



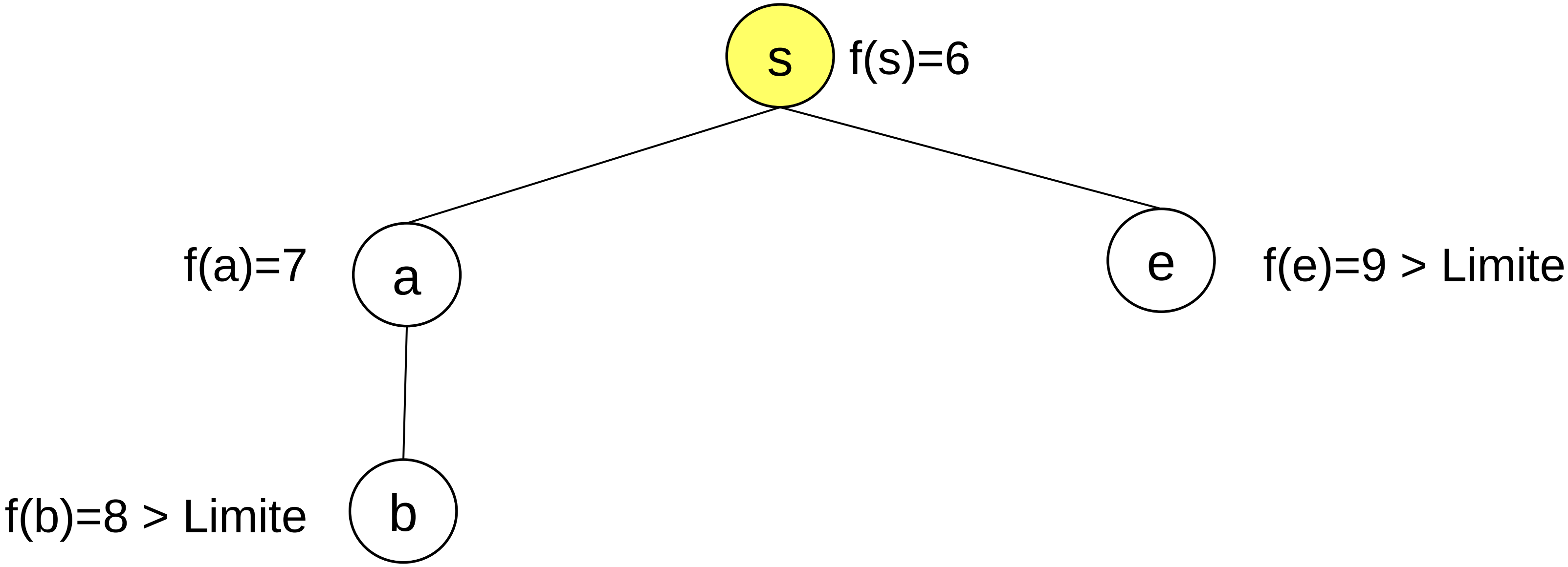
Limite = 7

IDA*



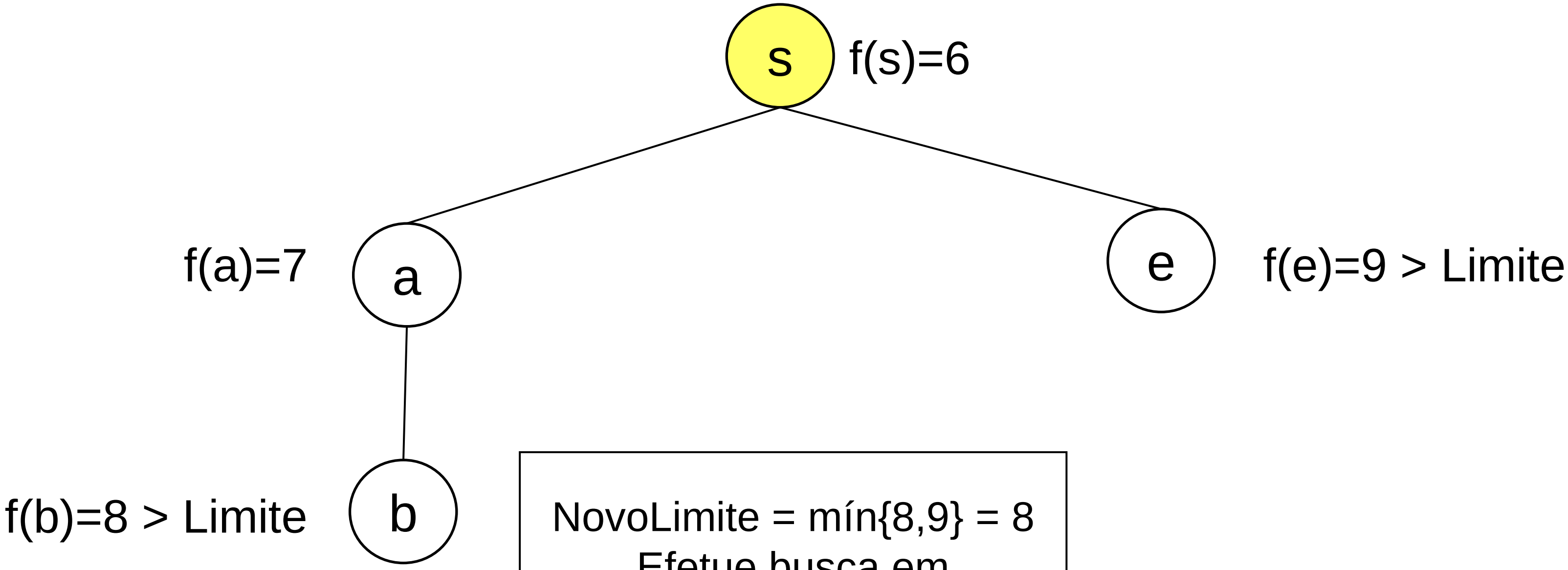
Limite = 7

IDA*



Limite = 7

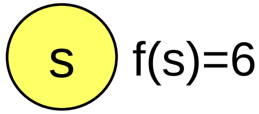
IDA*



NovoLimite = mín{8,9} = 8
Efetue busca em profundidade limitada por f<=8, iniciando pelo nó s

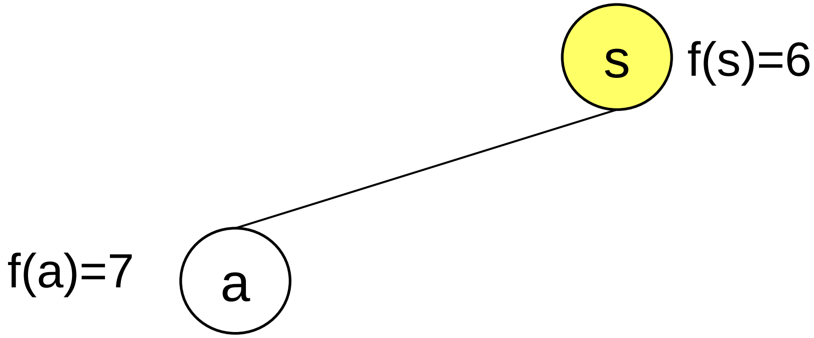
Limite = 8

IDA*



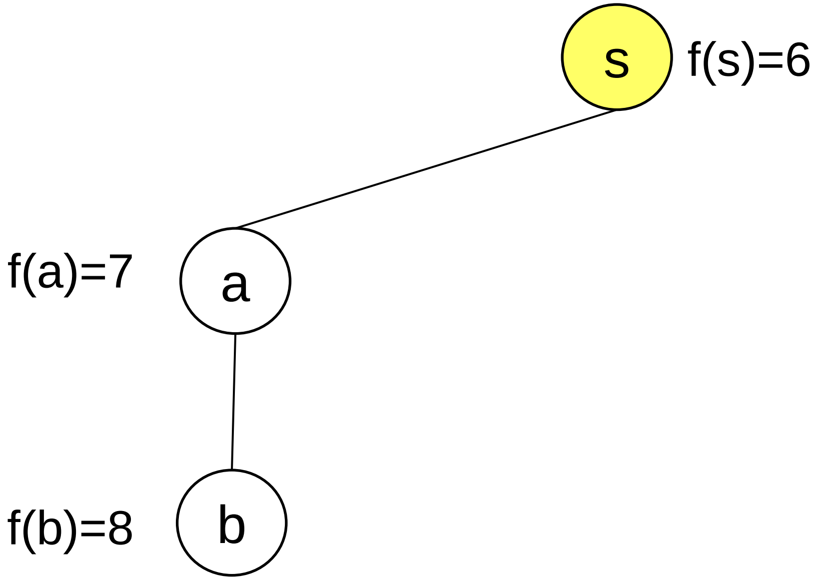
Limite = 8

IDA*



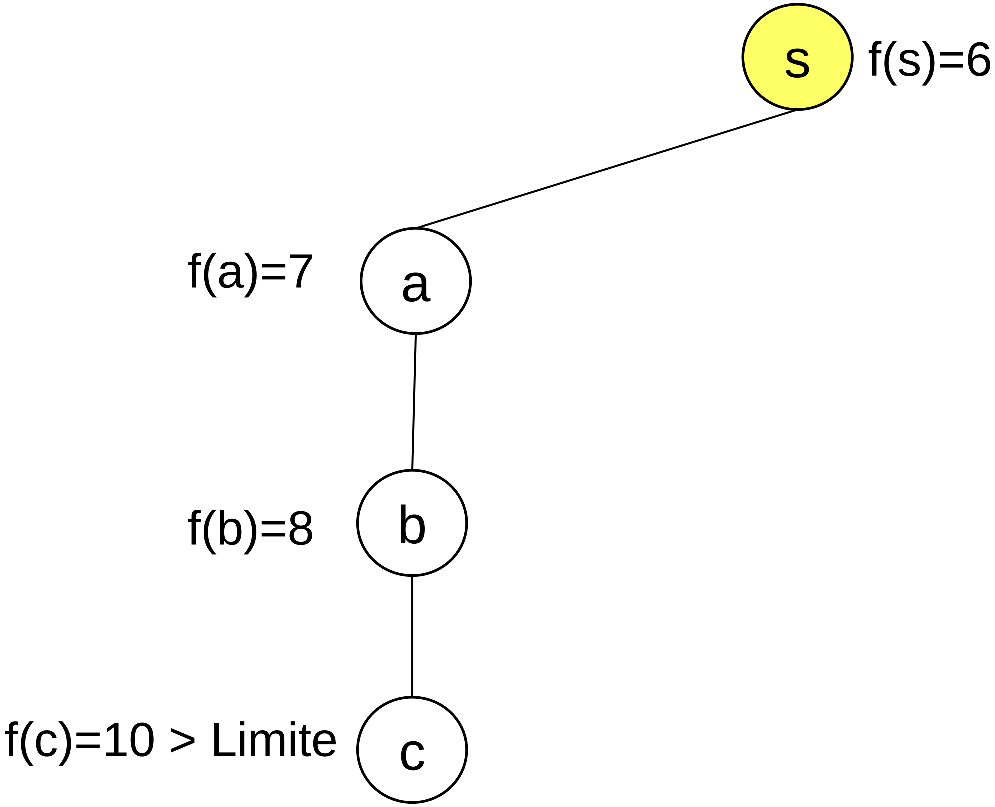
Limite = 8

IDA*



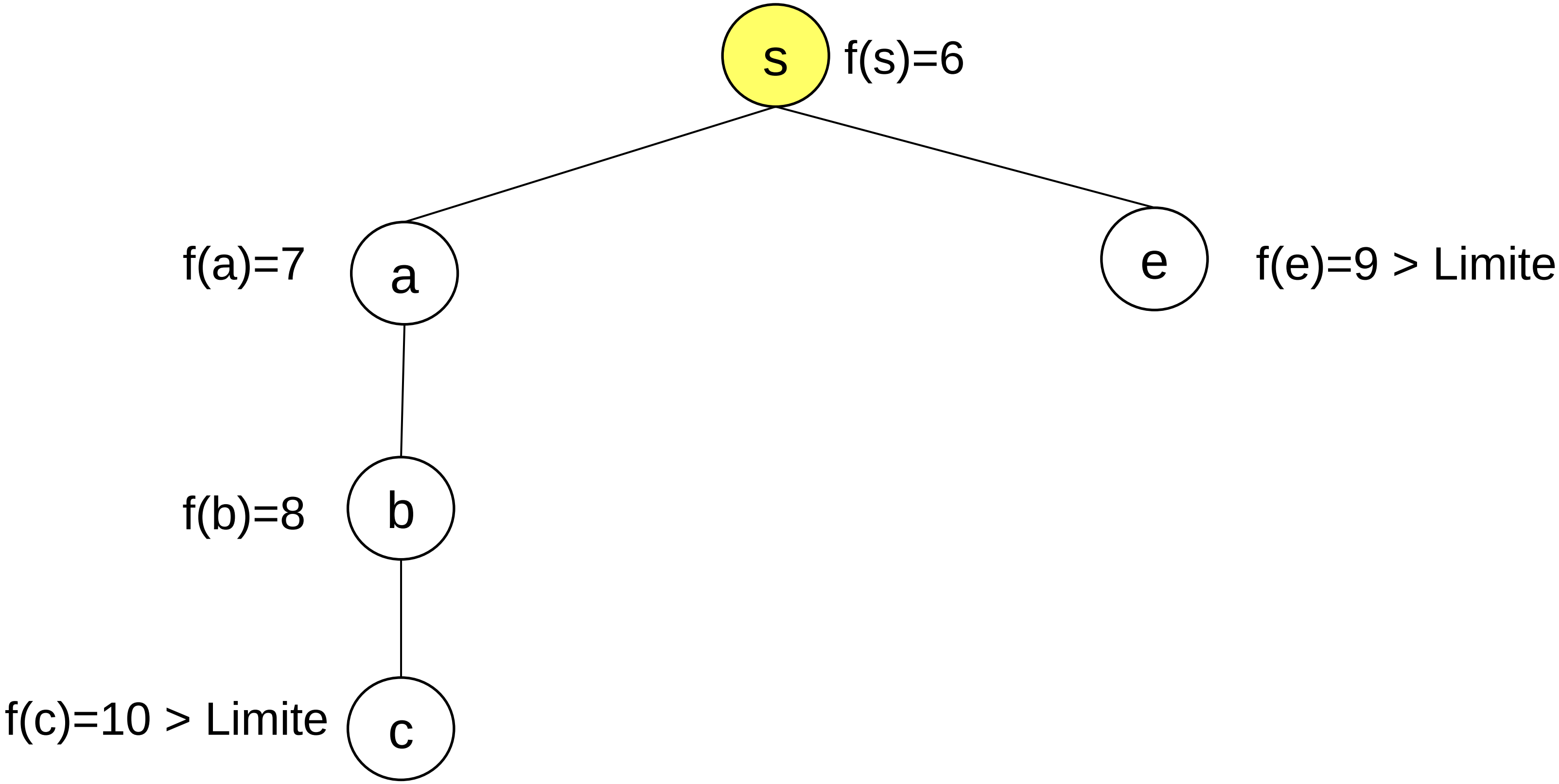
Limite = 8

IDA*



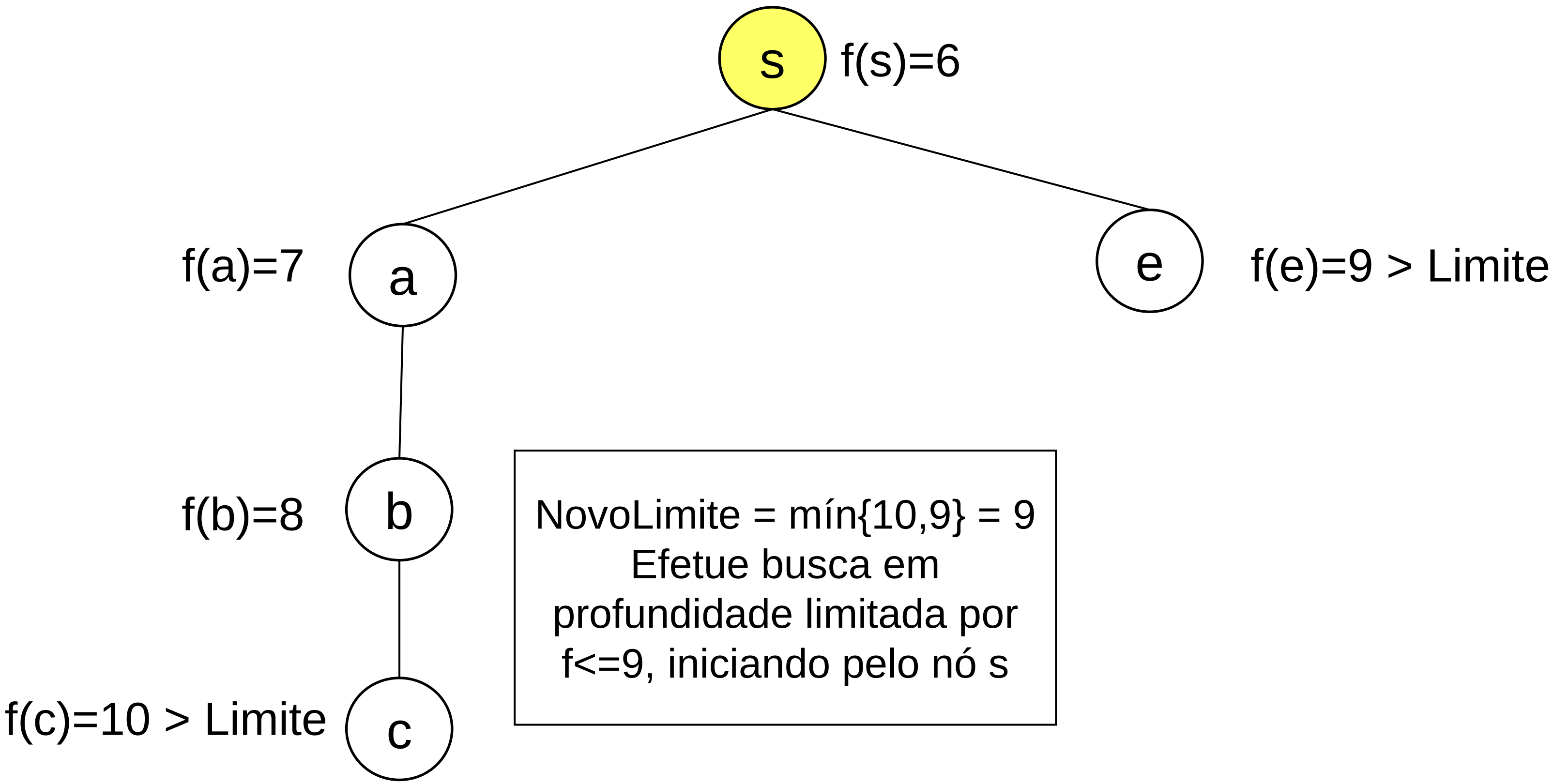
Limite = 8

IDA*



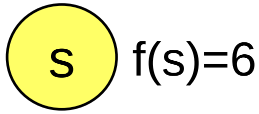
Limite = 8

IDA*


s
f(s)=6
f(a)=7
a
e
f(e)=9 > Limite
f(b)=8
b
NovoLimite = mín{10,9} = 9
Efetue busca em
profundidade limitada por
f<=9, iniciando pelo nó s
f(c)=10 > Limite
c

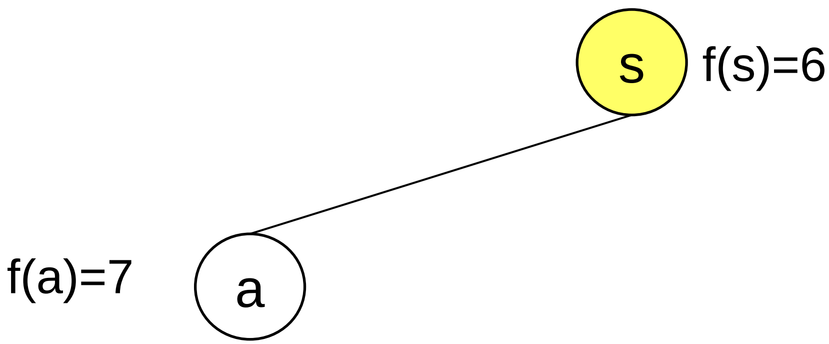
Limite = 9

IDA*



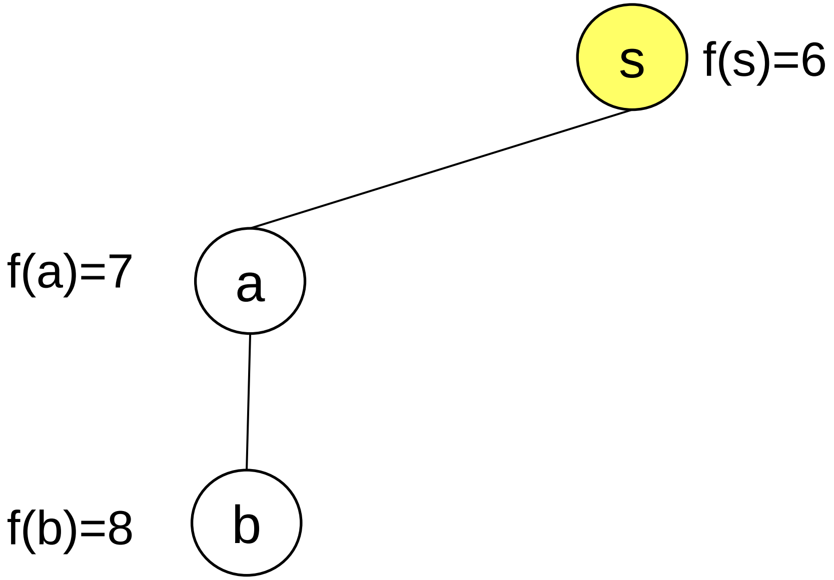
Limite = 9

IDA*



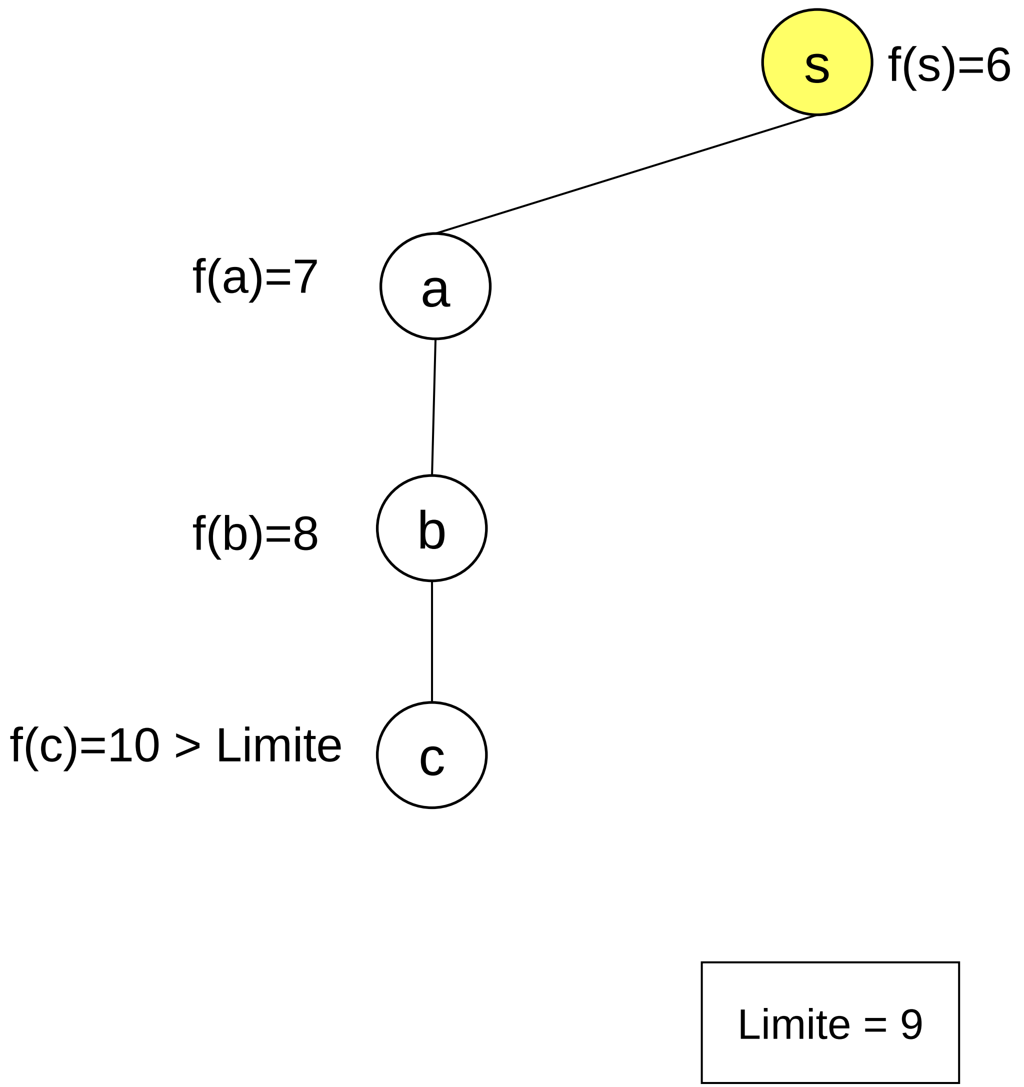
Limite = 9

IDA*

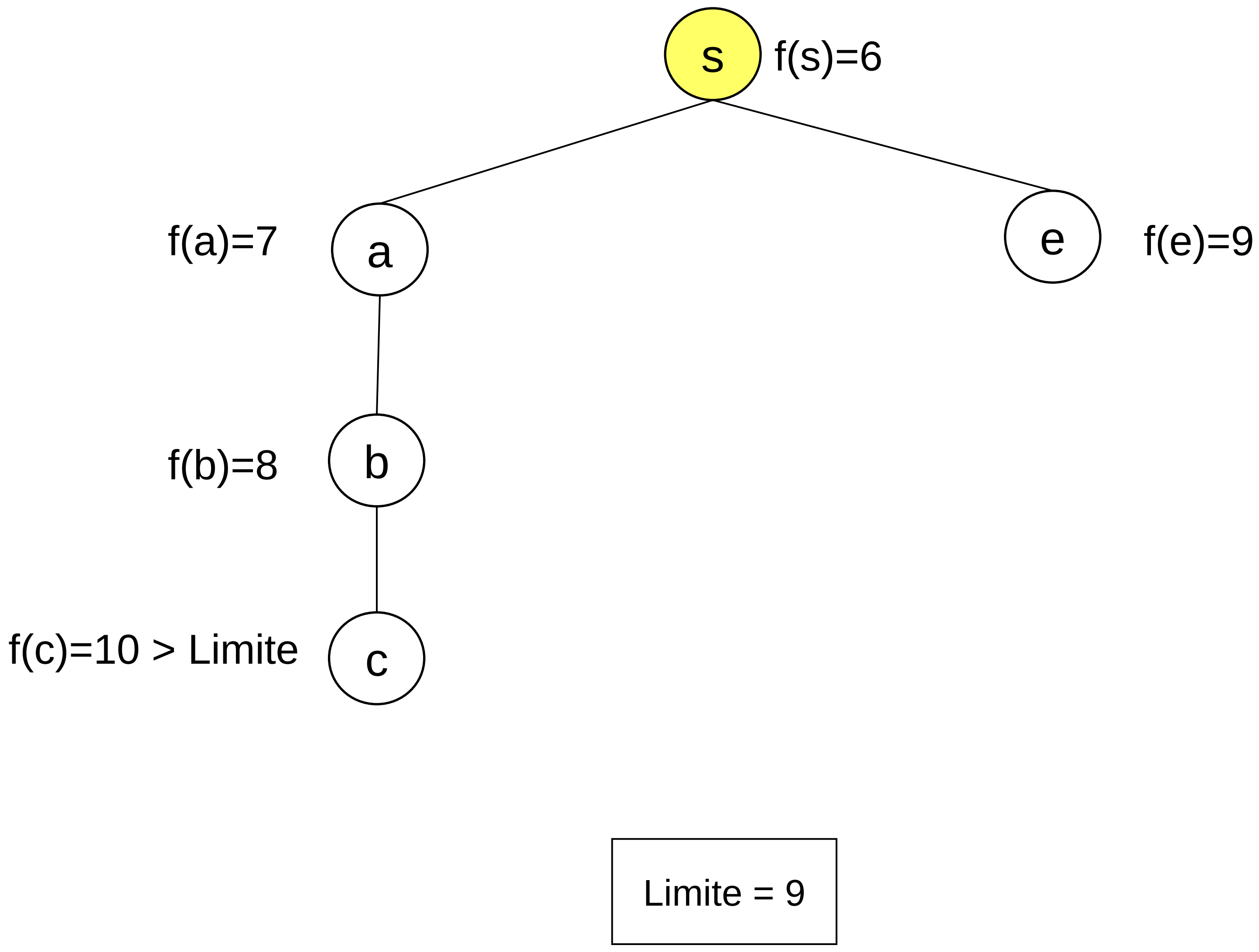


Limite = 9

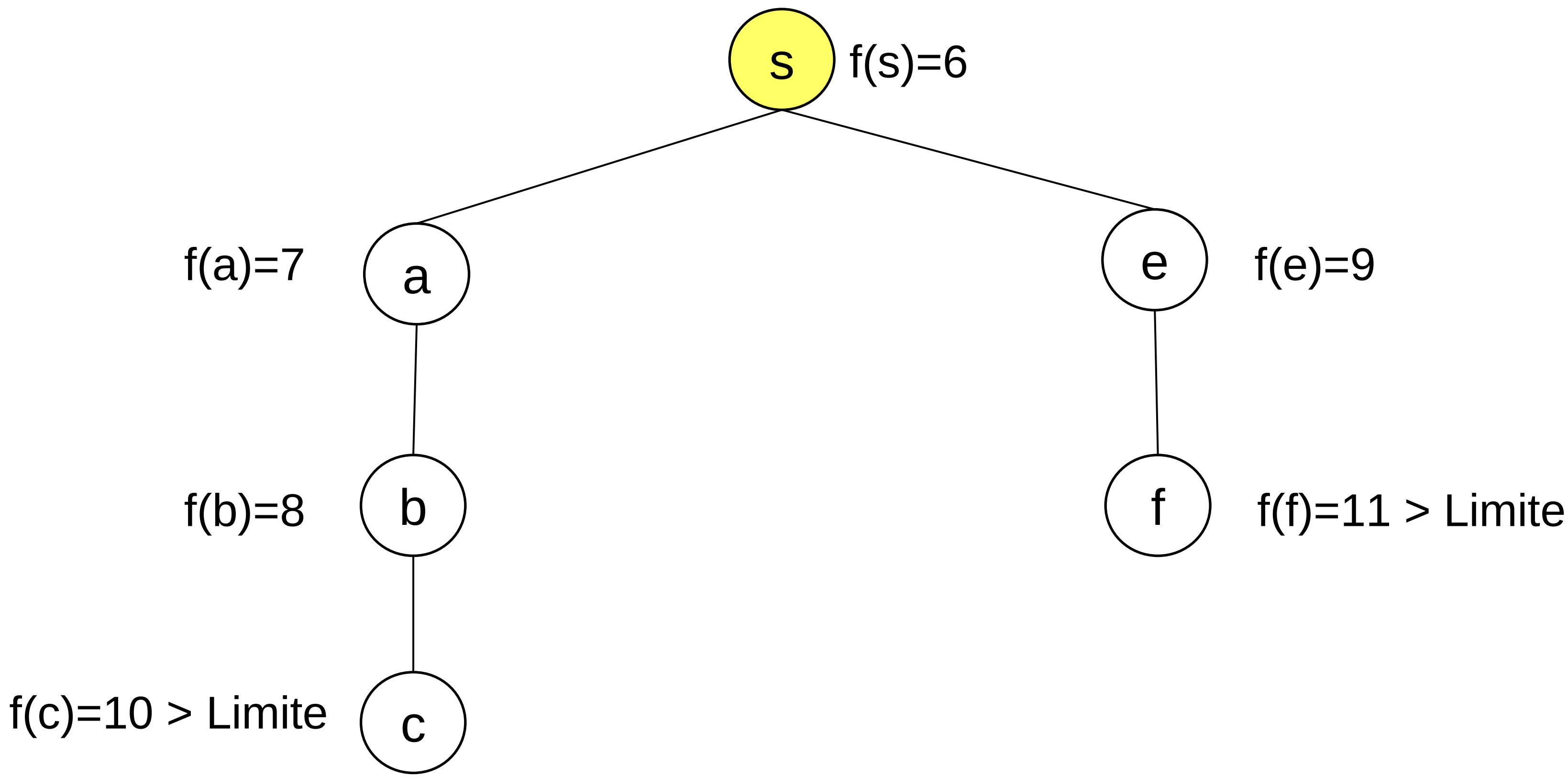
IDA*



IDA*

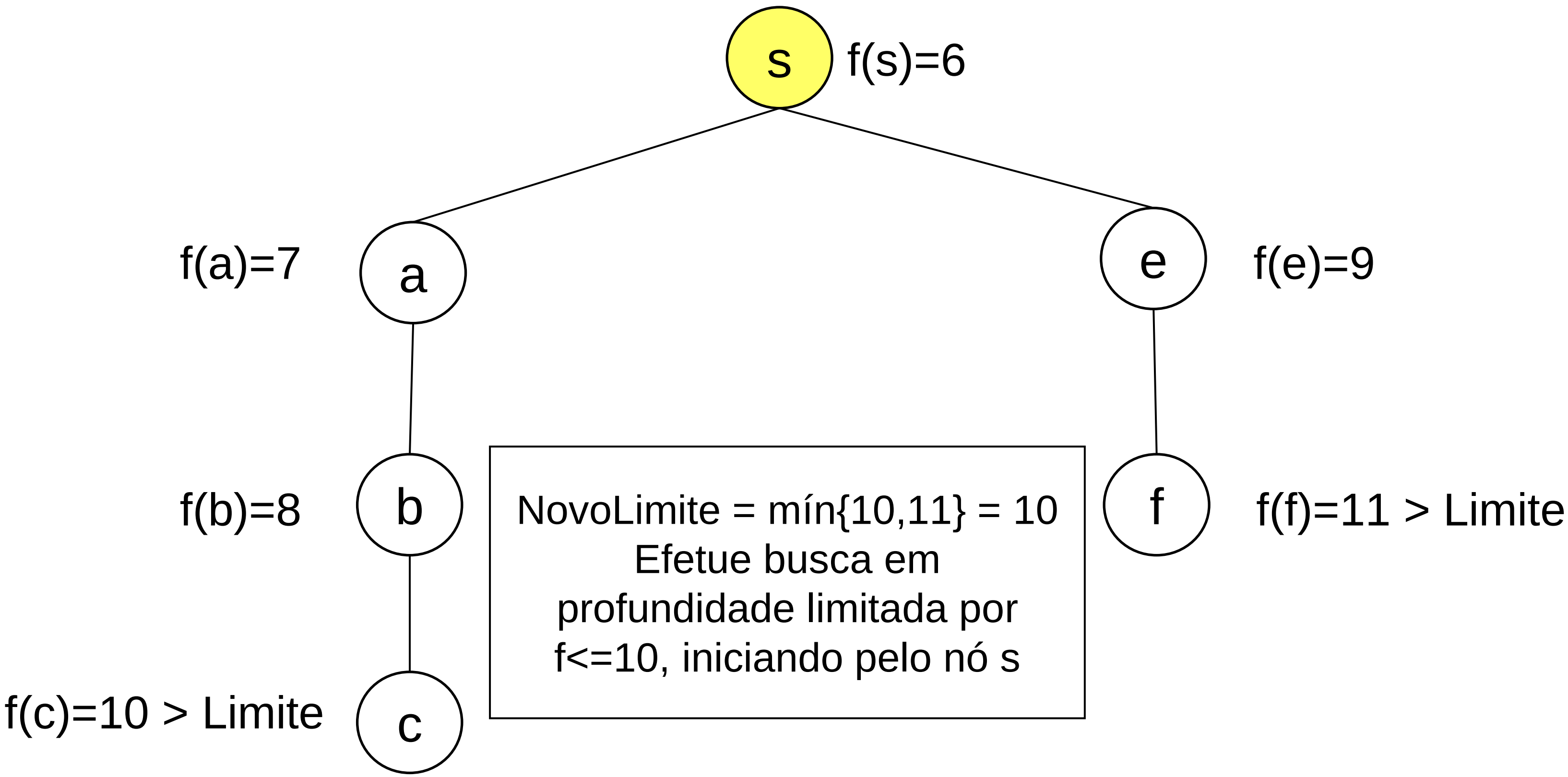


IDA*



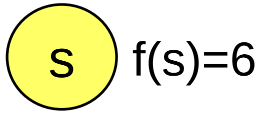
Limite = 9

IDA*



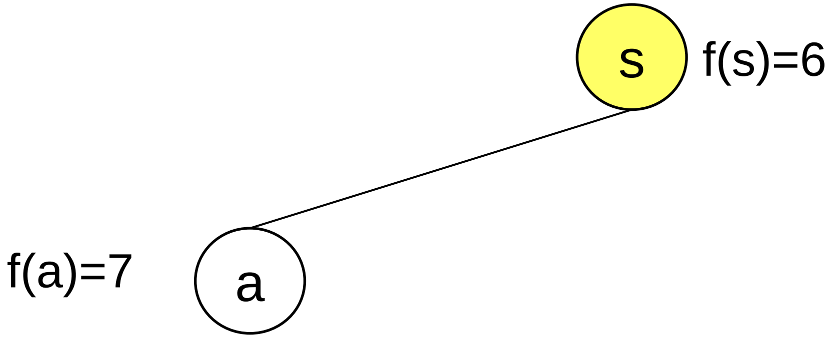
Limite = 10

IDA*



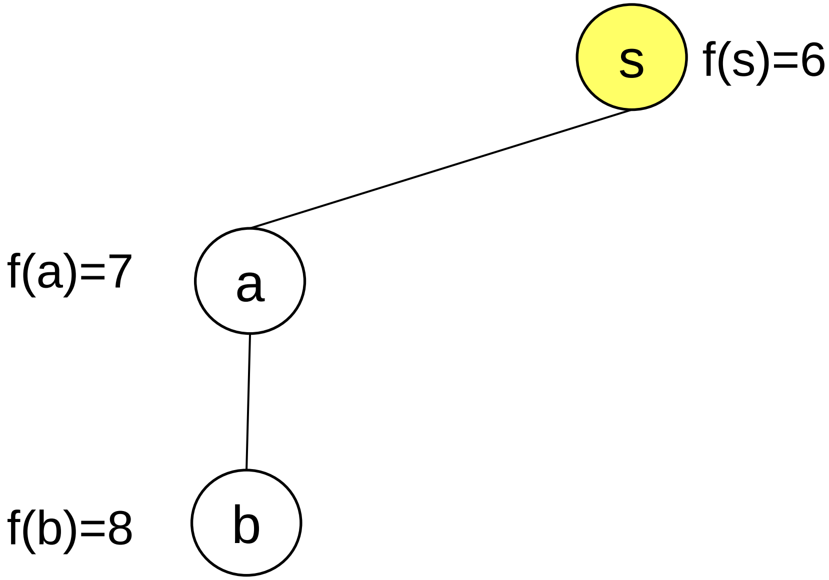
Limite = 10

IDA*



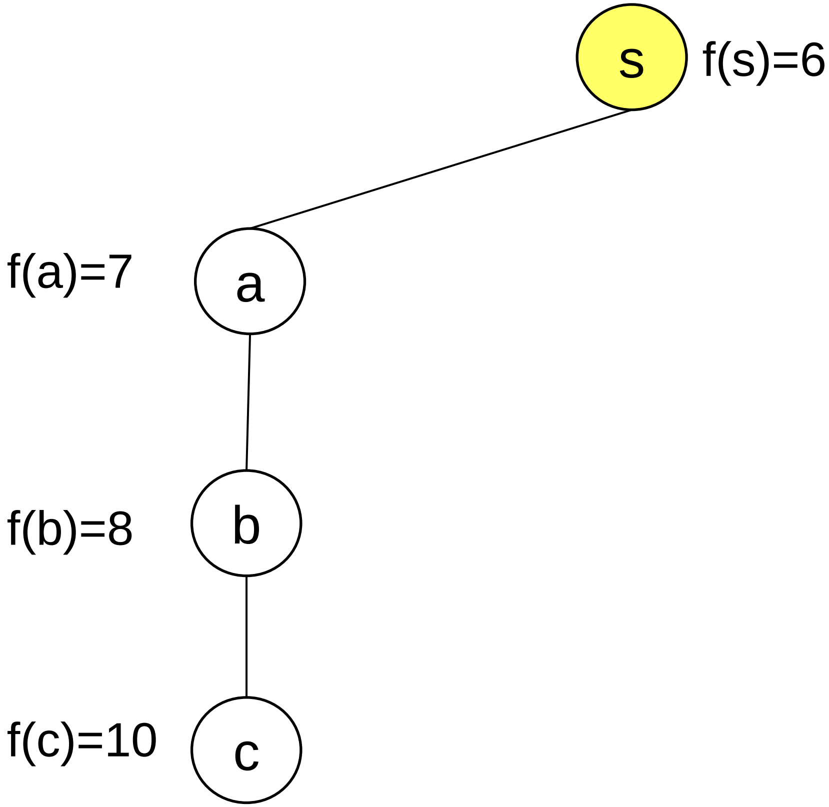
Limite = 10

IDA*



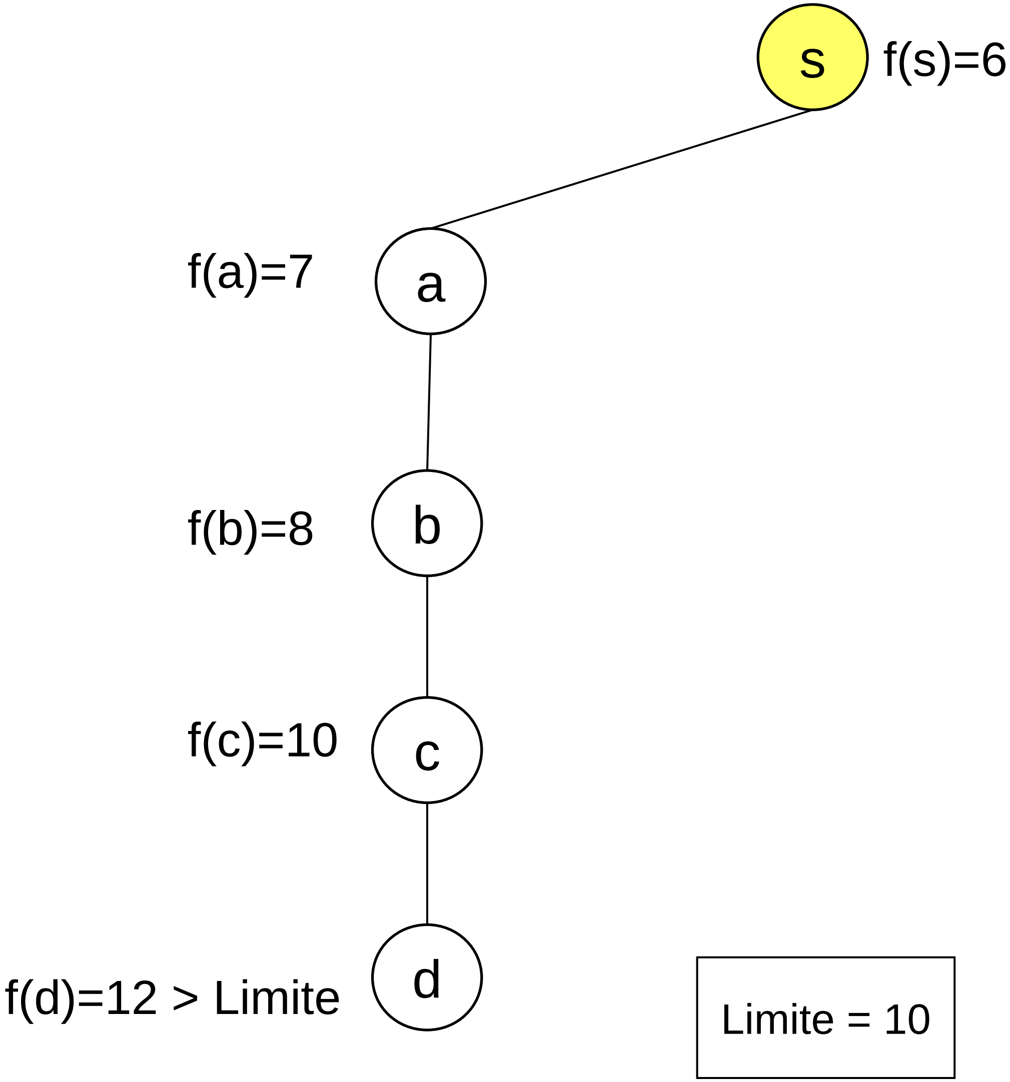
Limite = 10

IDA*

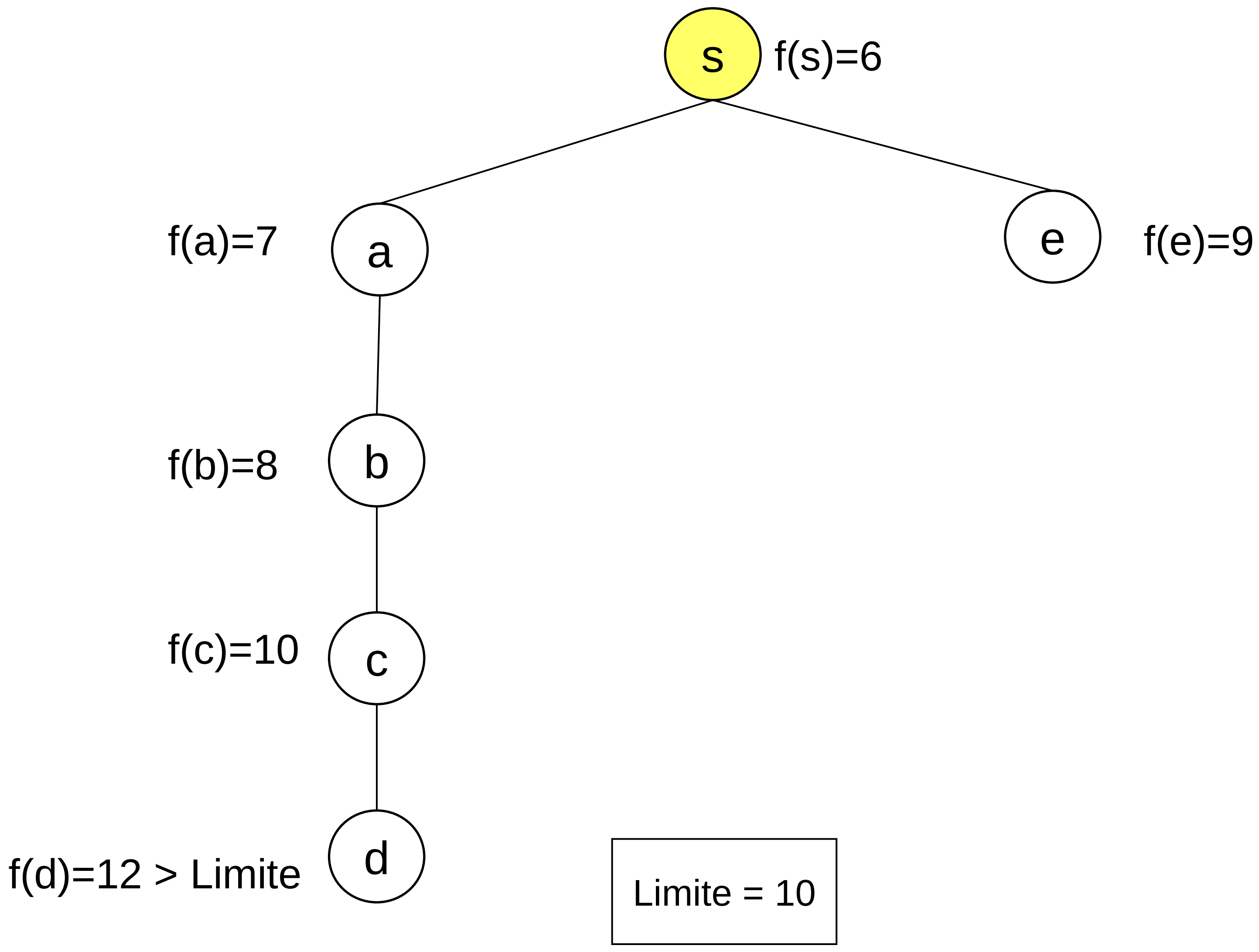


Limite = 10

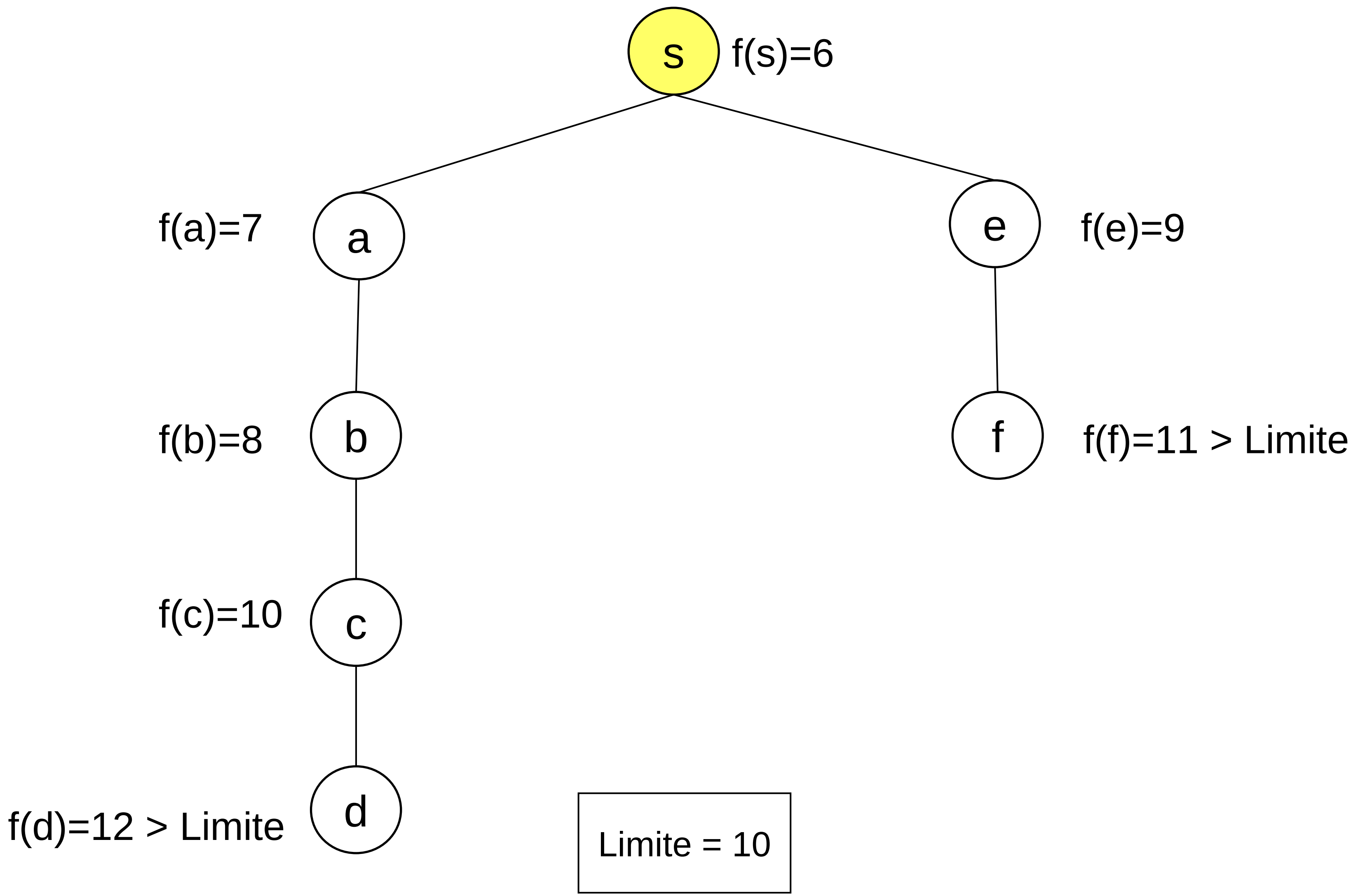
IDA*



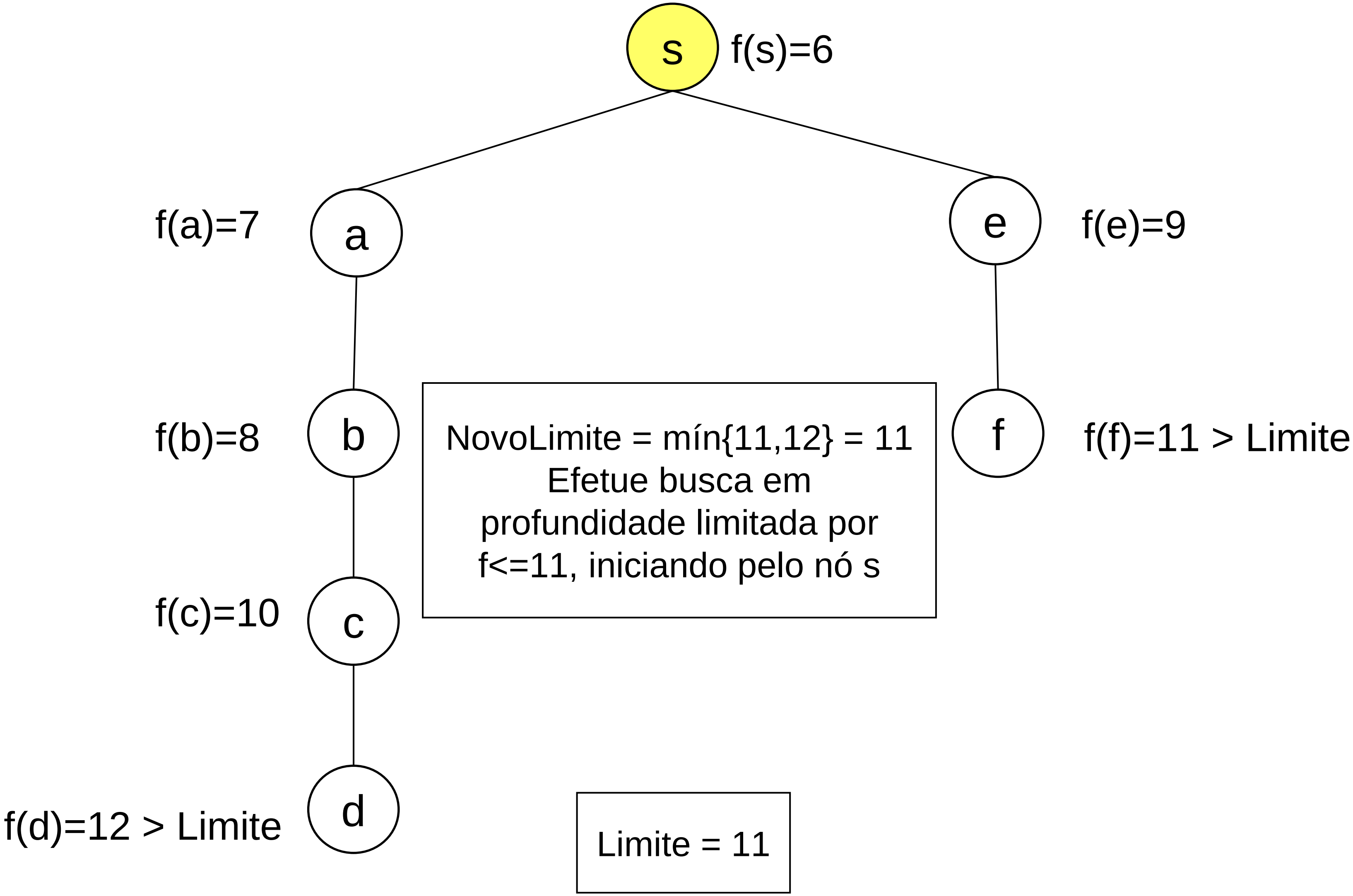
IDA*



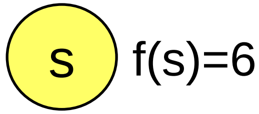
IDA*


s
f(s)=6
f(a)=7
a
e
f(e)=9
f(b)=8
b
f
f(f)=11 > Limite
f(c)=10
c
f(d)=12 > Limite
d
Limite = 10

IDA*

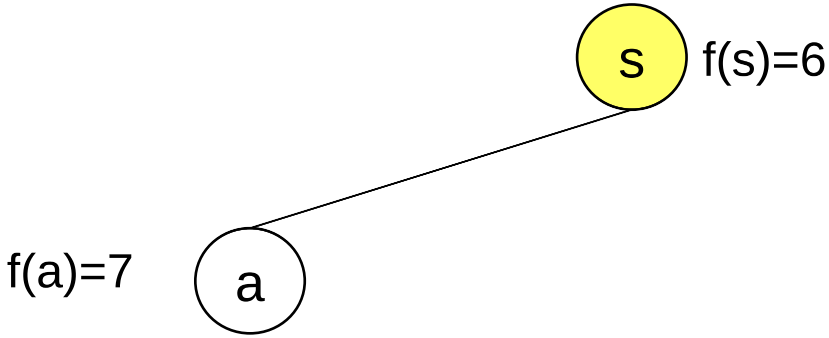

s
f(s)=6
f(a)=7
a
e
f(e)=9
f(b)=8
b
NovoLimite = mín{11,12} = 11
Efetue busca em
profundidade limitada por
f<=11, iniciando pelo nó s
f
f(f)=11 > Limite
f(c)=10
c
f(d)=12 > Limite
d
Limite = 11

IDA*



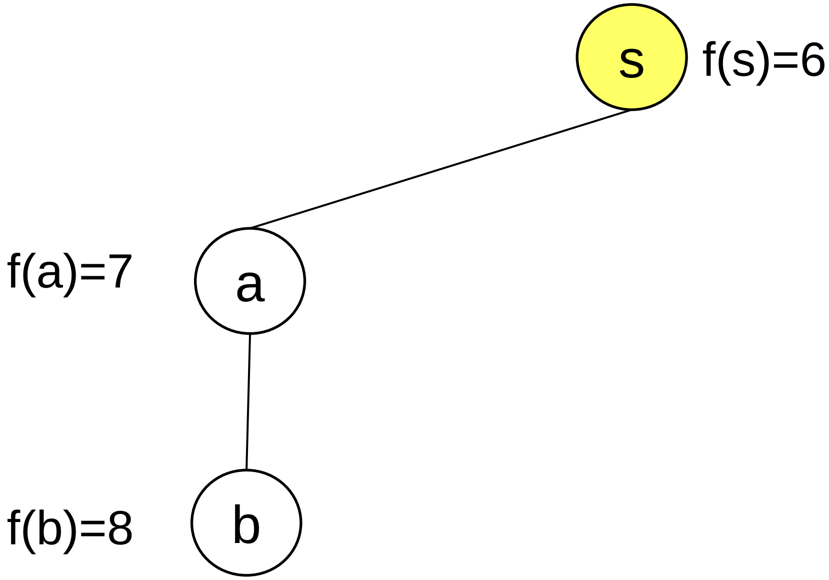
Limite = 11

IDA*



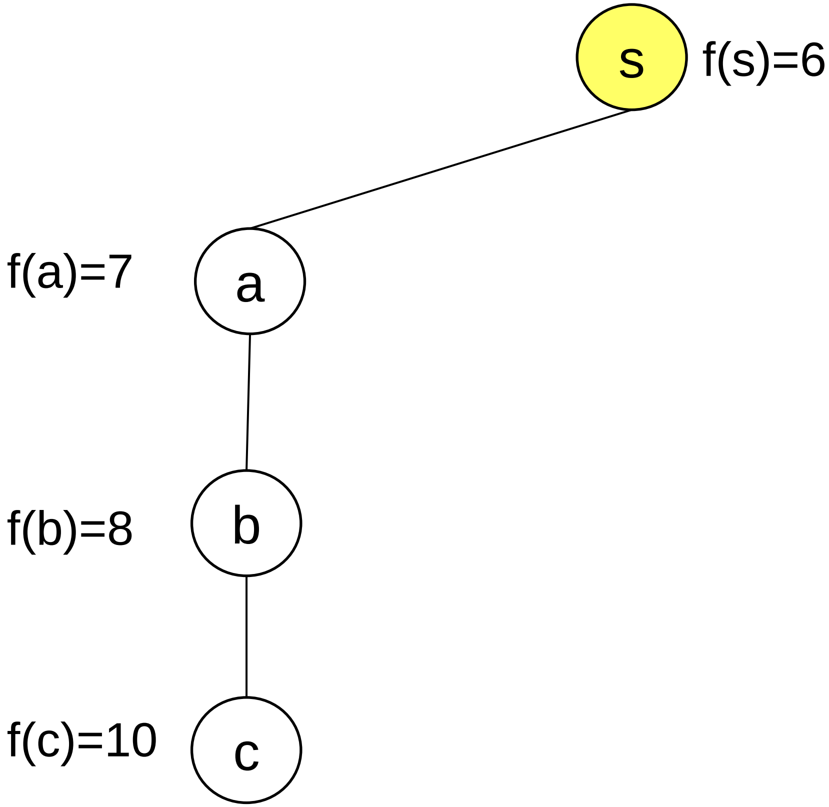
Limite = 11

IDA*



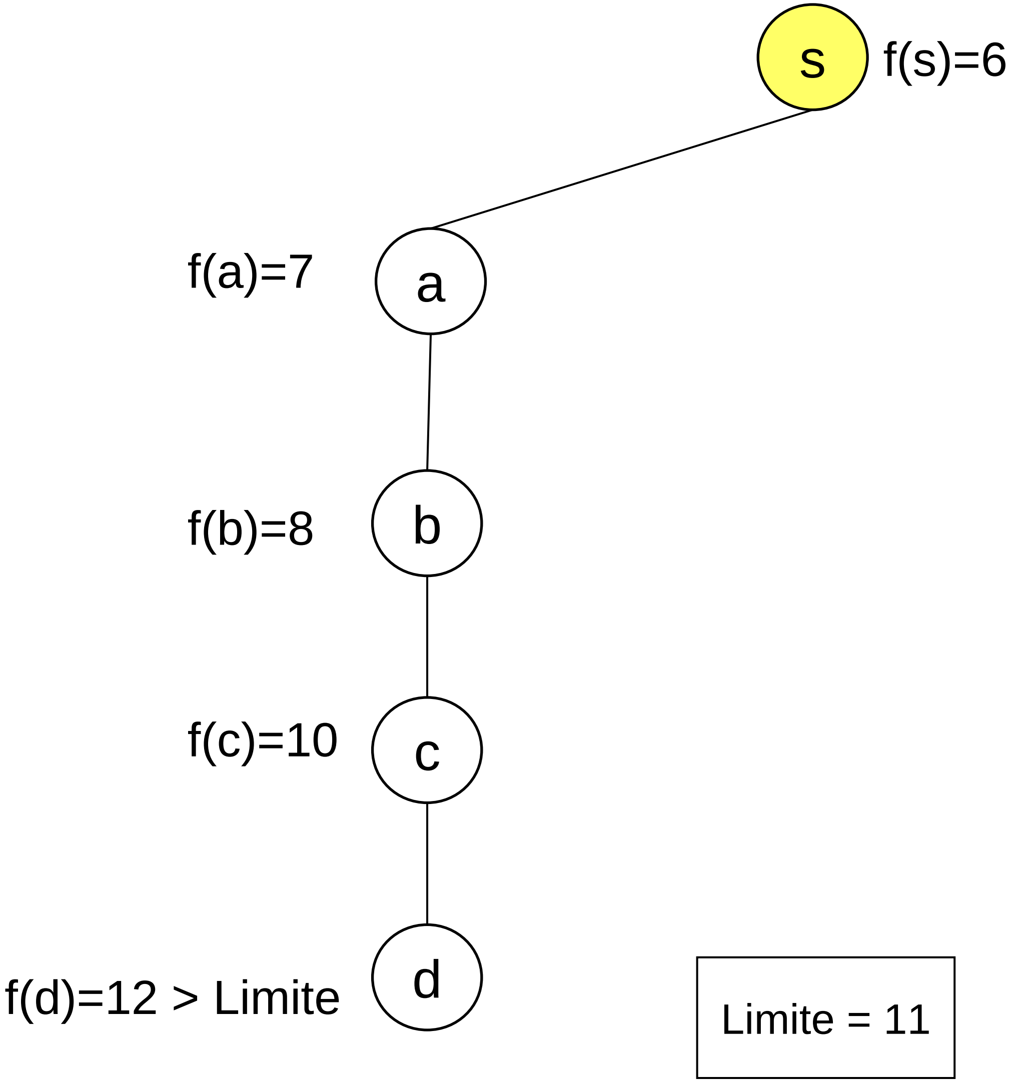
Limite = 11

IDA*

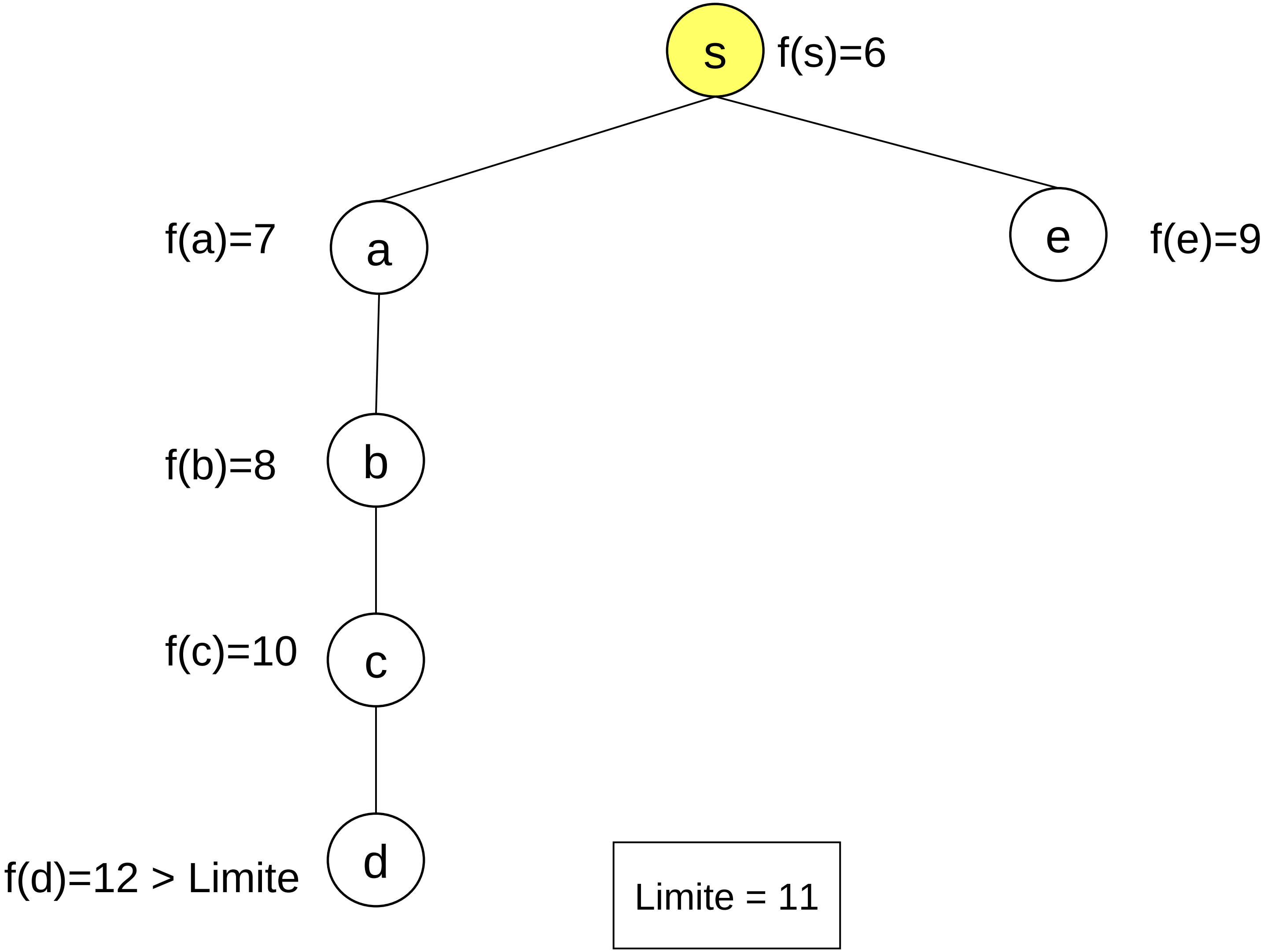


Limite = 11

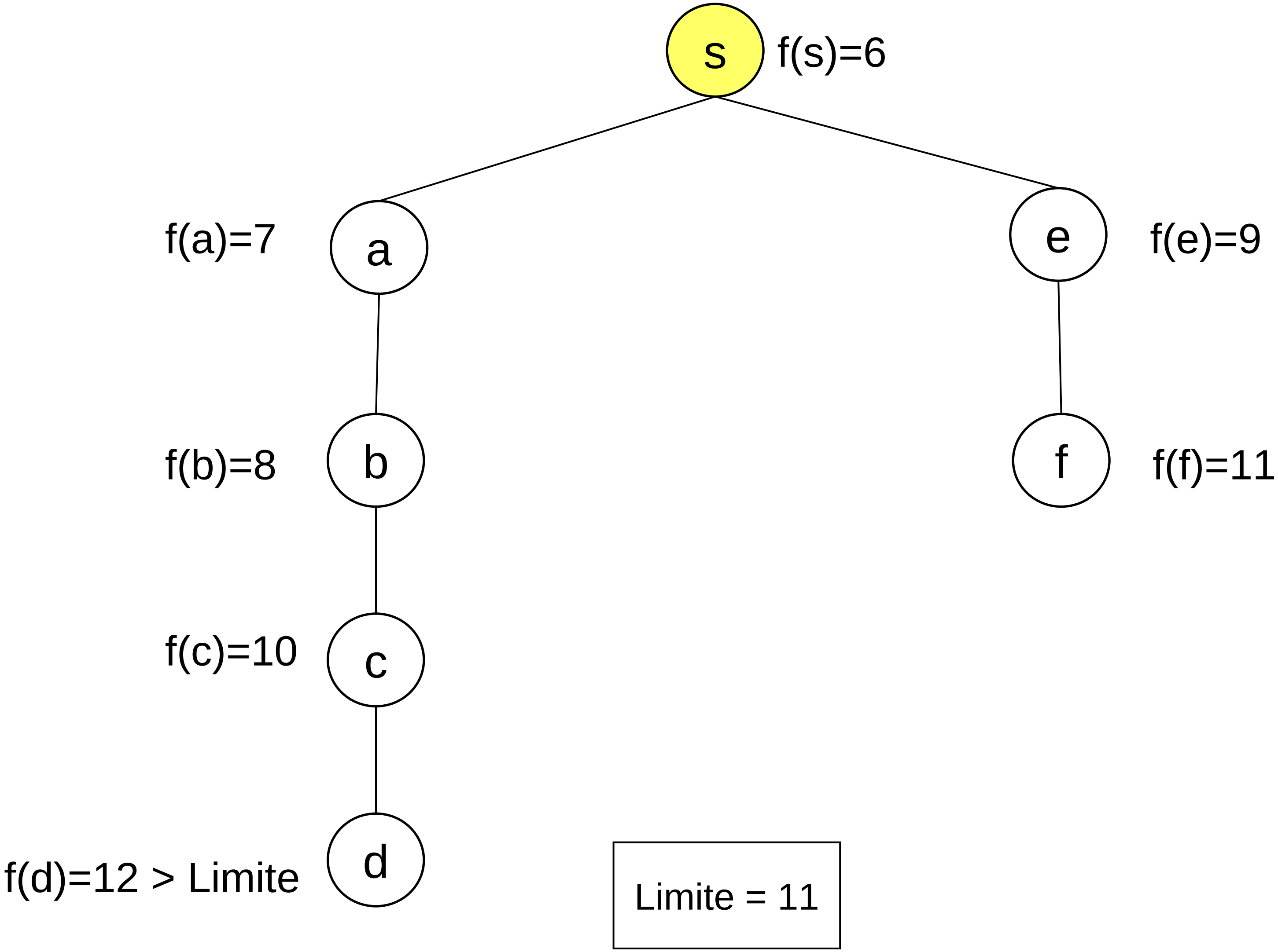
IDA*



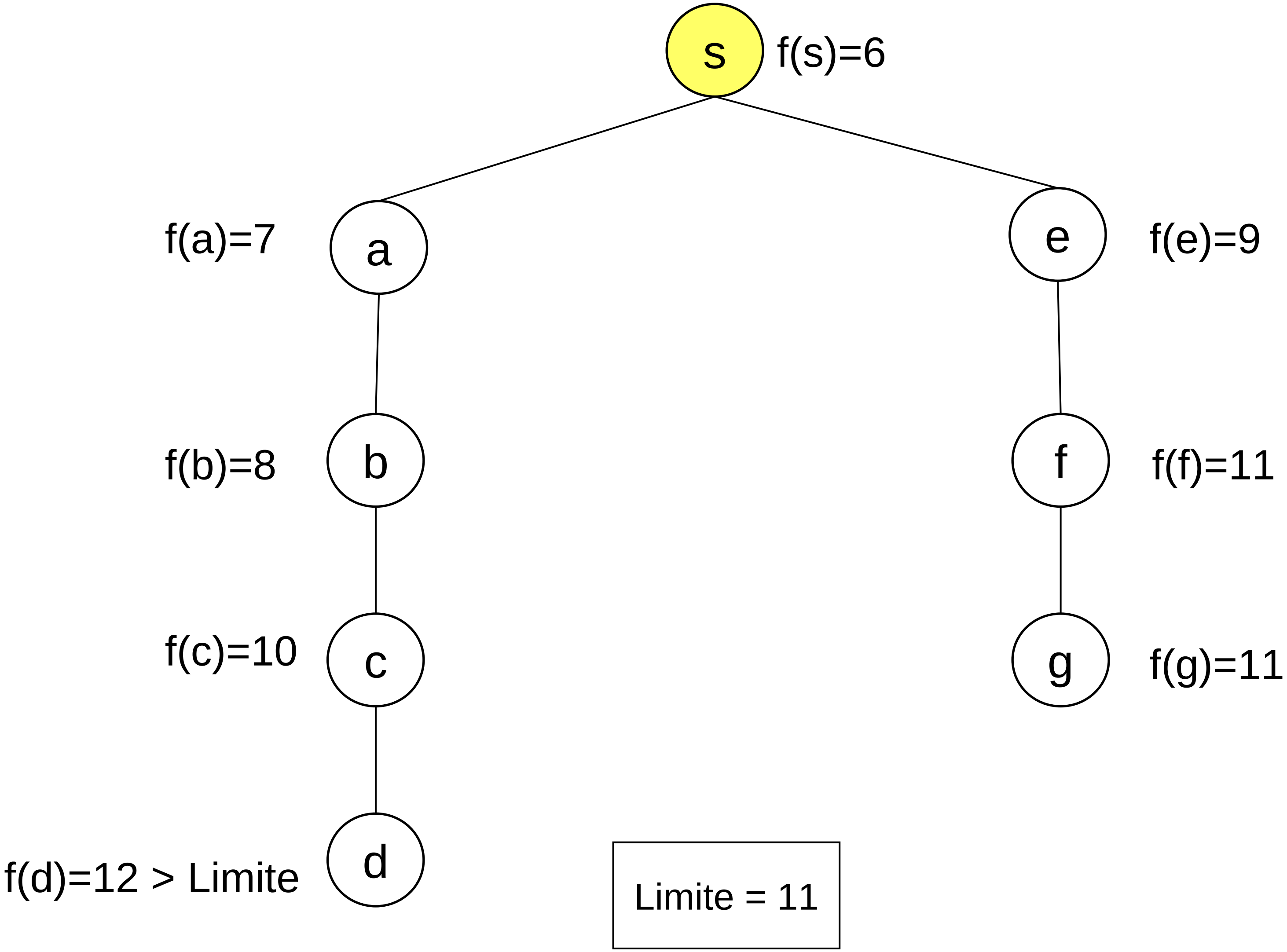
IDA*



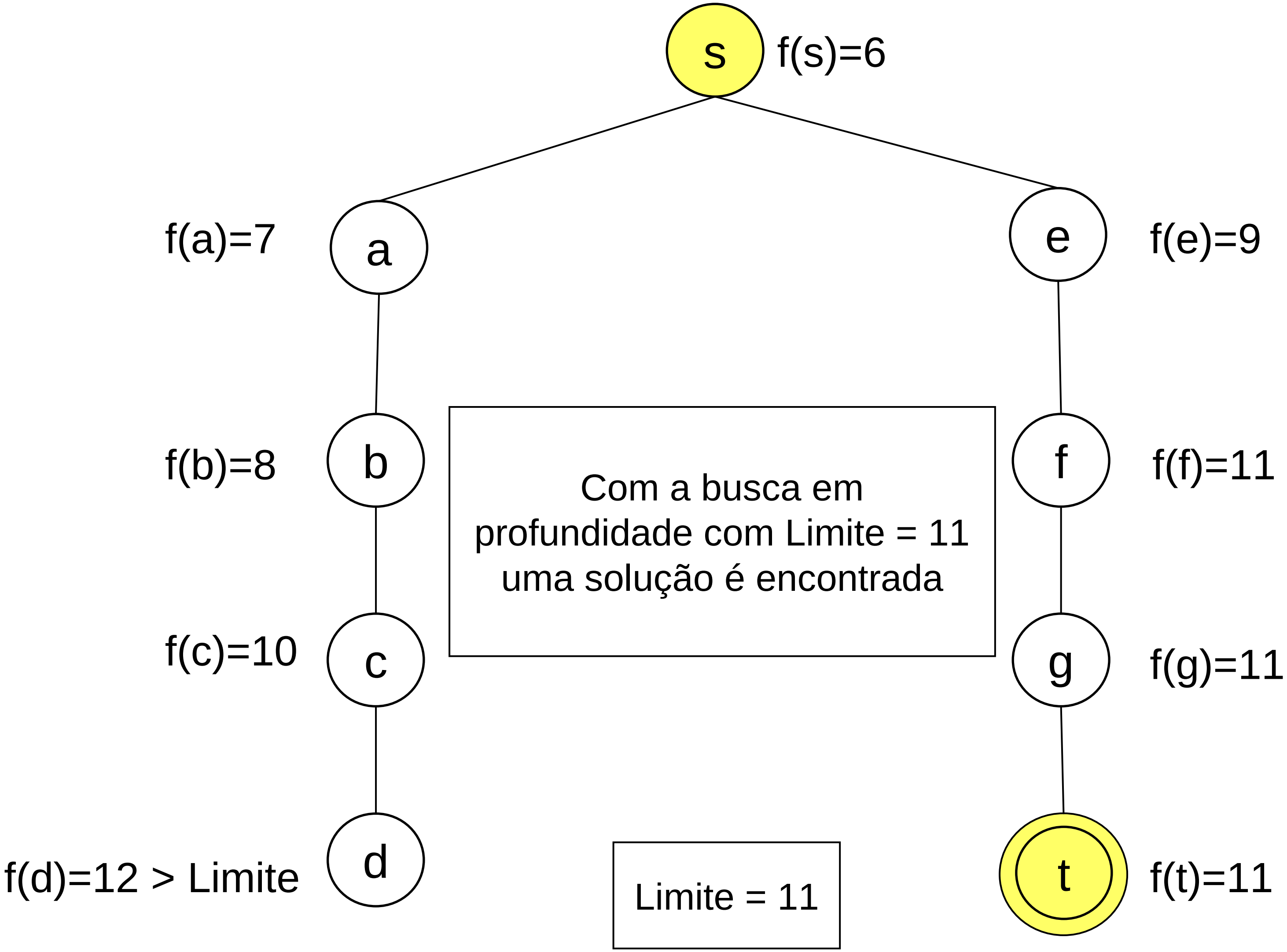
IDA*



IDA*


s
f(s)=6
a
f(a)=7
e
f(e)=9
b
f(b)=8
f
f(f)=11
c
f(c)=10
g
f(g)=11
d
f(d)=12 > Limite
Limite = 11

IDA*



s
f(s)=6
f(a)=7
a
e
f(e)=9
f(b)=8
b
Com a busca em
profundidade com Limite = 11
uma solução é encontrada
f
f(f)=11
f(c)=10
c
g
f(g)=11
f(d)=12 > Limite
d
Limite = 11
t
f(t)=11

IDA*

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

resolvai2(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 A* (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($\mathbf{n_i}$) : $\mathbf{n_i}$ é um sucessor de **n**}

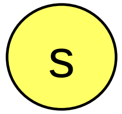
RBFS

- 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

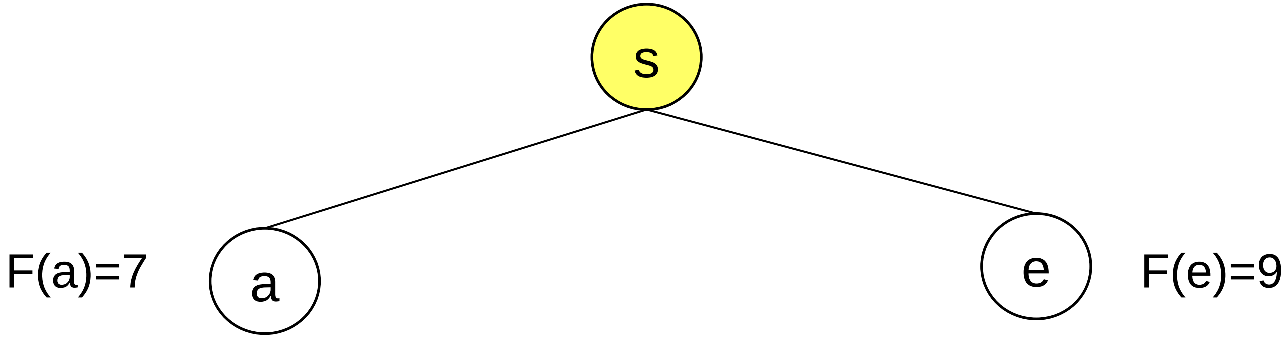
RBFS

- 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 $f(\mathbf{n_i})<F(\mathbf{n})$ then $F(\mathbf{n_i}) \leftarrow F(\mathbf{n})$ else $F(\mathbf{n_i}) \leftarrow f(\mathbf{n_i})$
- que pode ser escrito como:
  - $F(\mathbf{n_i}) \leftarrow \text{máx}\{F(\mathbf{n}), f(\mathbf{n_i})\}$

RBFS



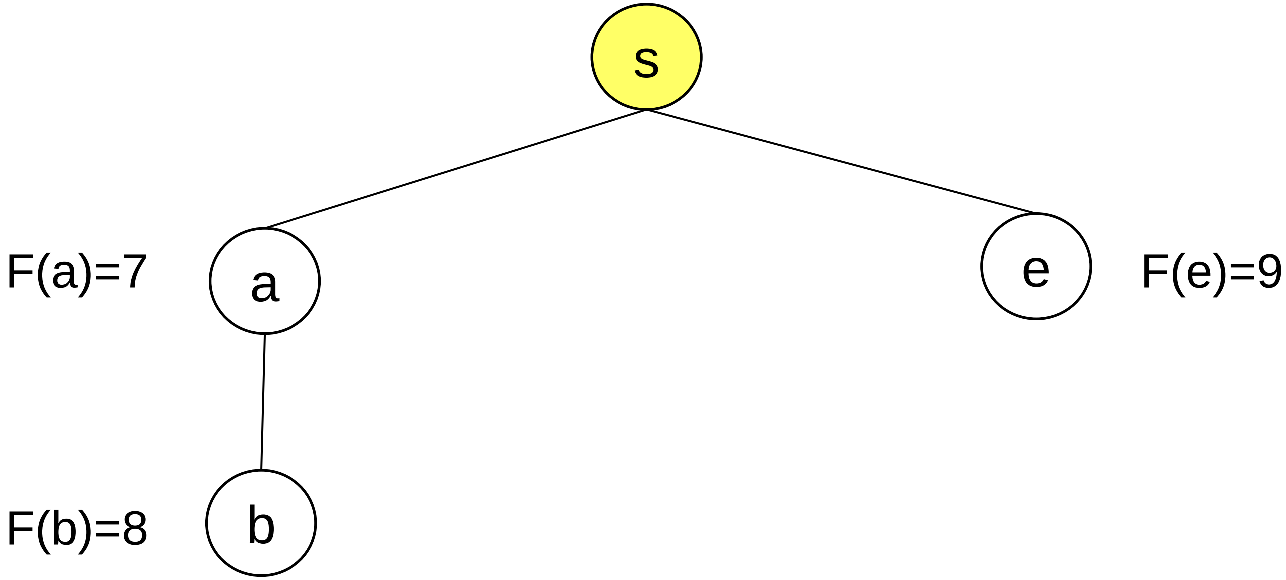
RBFS



O melhor candidato é o nó **a**, pois F(a)<F(e). A busca prossegue via **a**

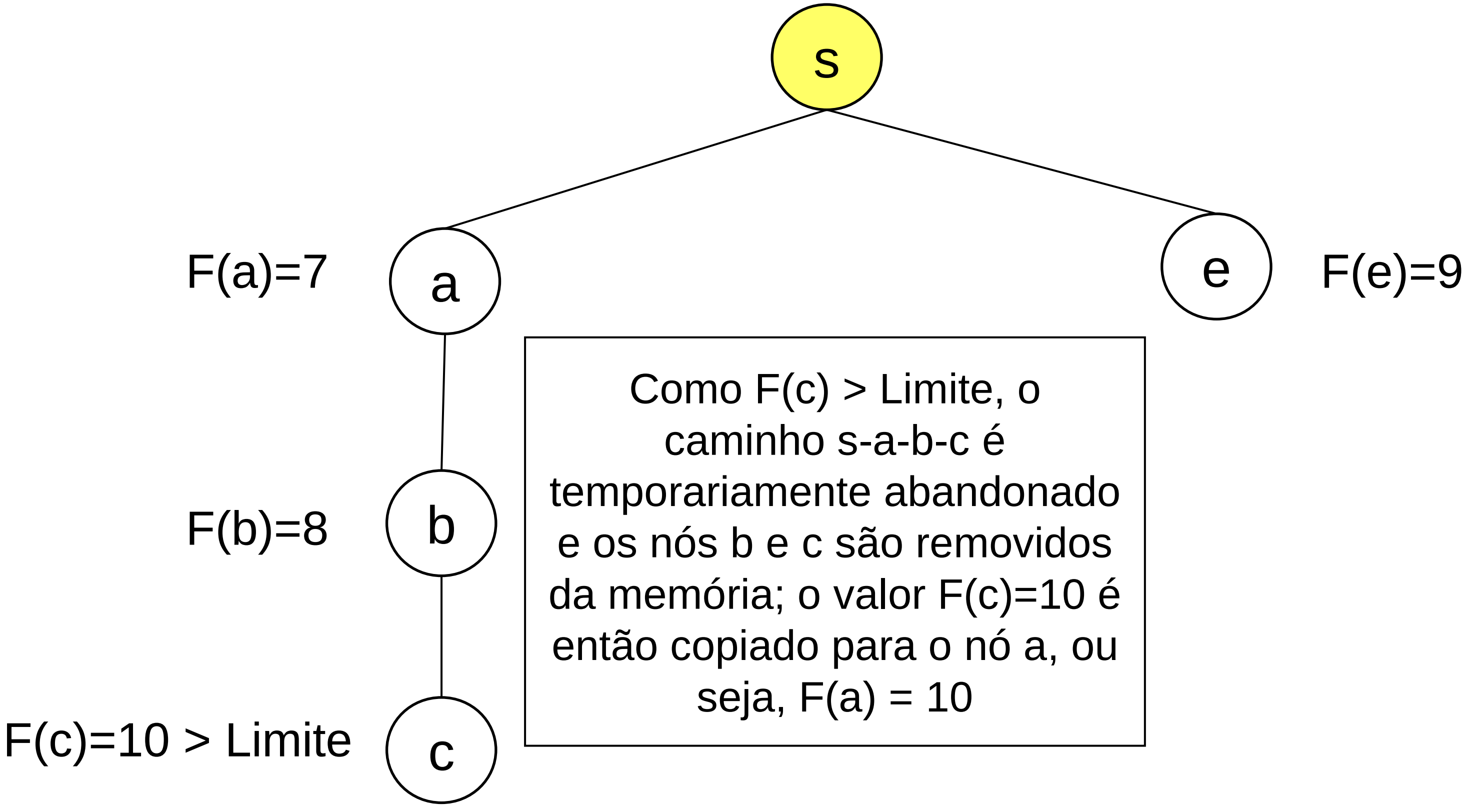
Limite = 9

RBFS



Limite = 9

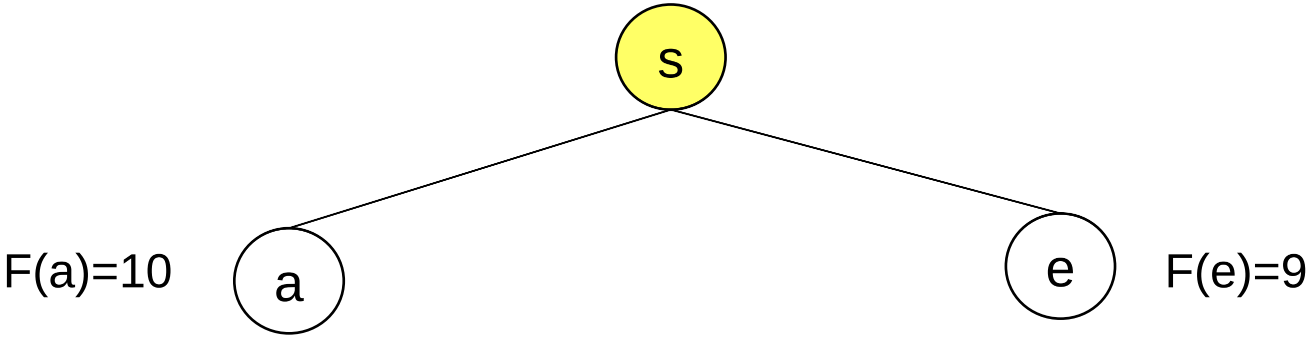
RBFS



Como F(c) > Limite, o caminho s-a-b-c é temporariamente abandonado e os nós b e c são removidos da memória; o valor F(c)=10 é então copiado para o nó a, ou seja, F(a) = 10

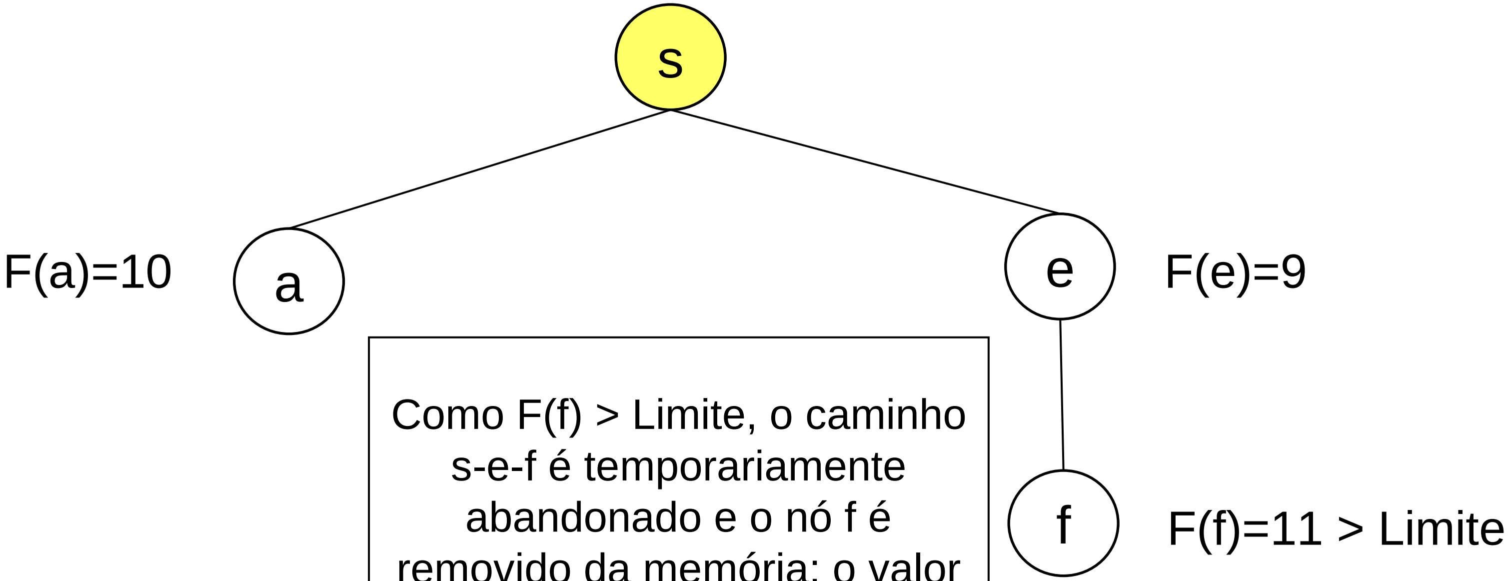
Limite = 9

RBFS



Limite = 10

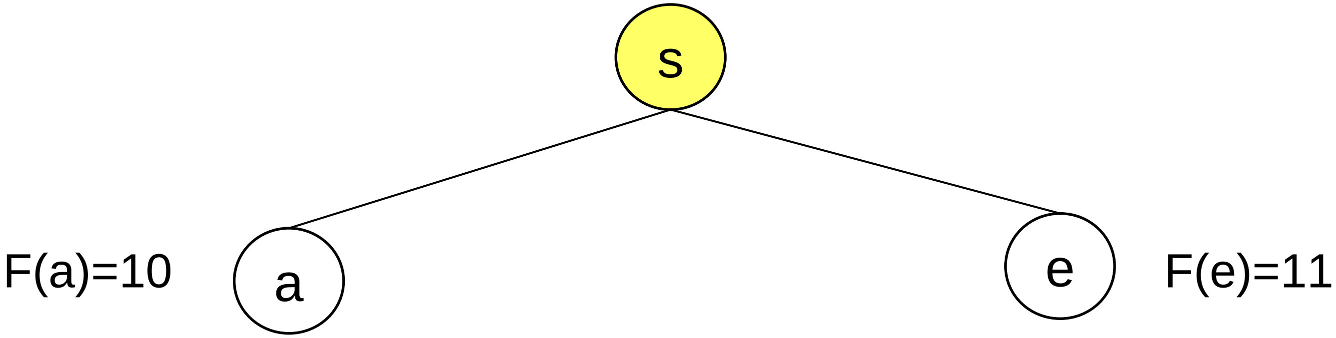
RBFS



Como F(f) > Limite, o caminho s-e-f é temporariamente abandonado e o nó f é removido da memória; o valor F(f)=11 é então copiado para o nó e, ou seja, F(e) = 11

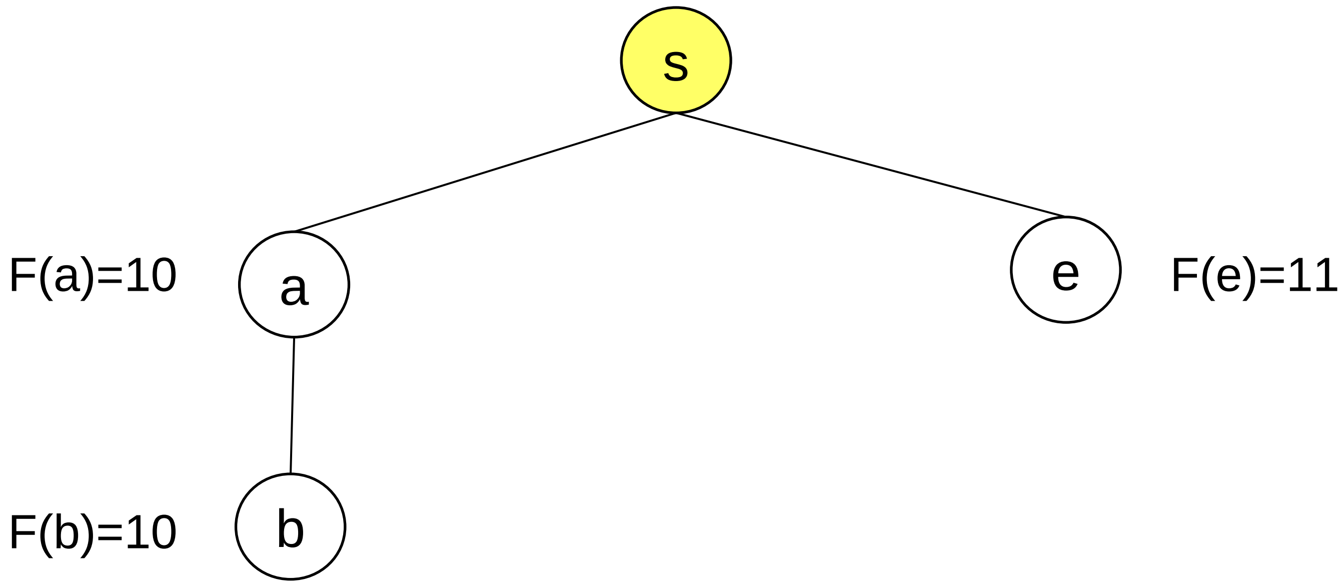
Limite = 10

RBFS



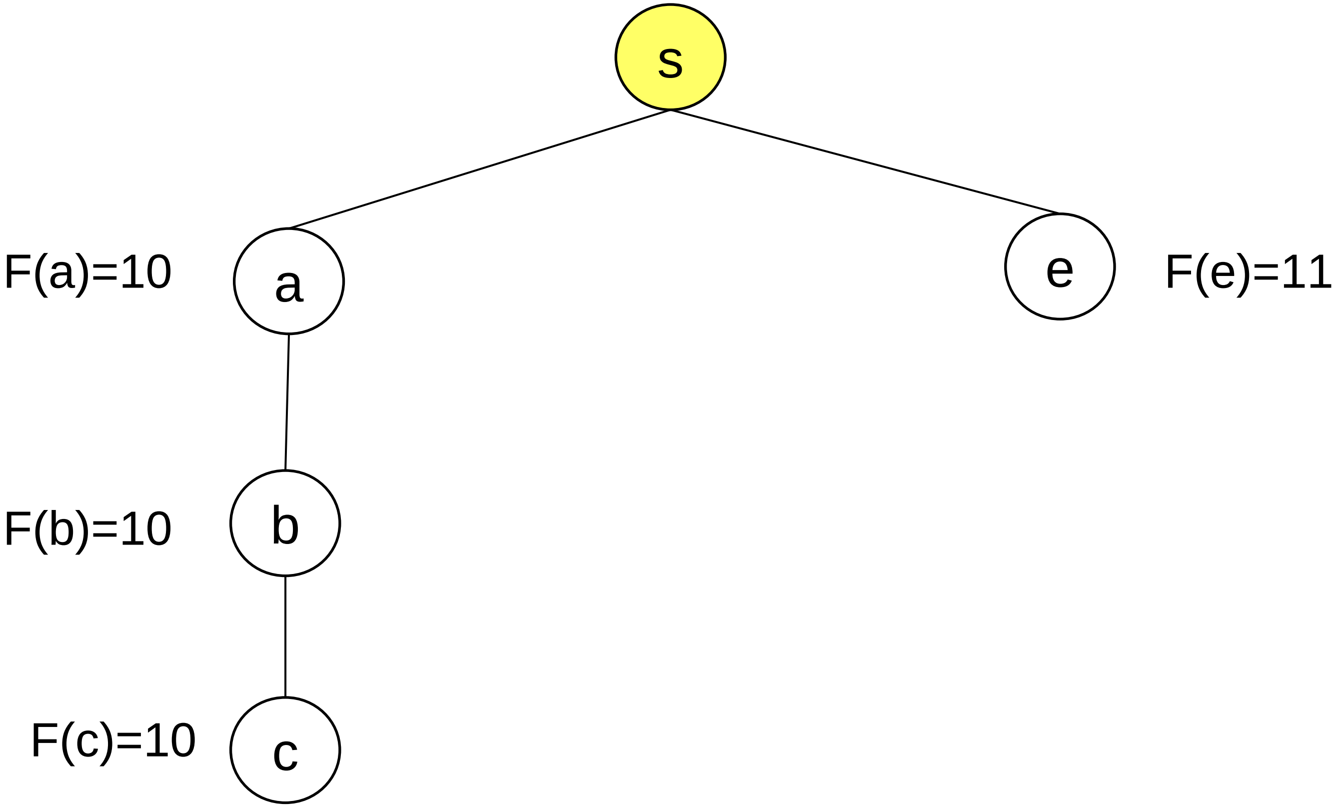
Limite = 11

RBFS



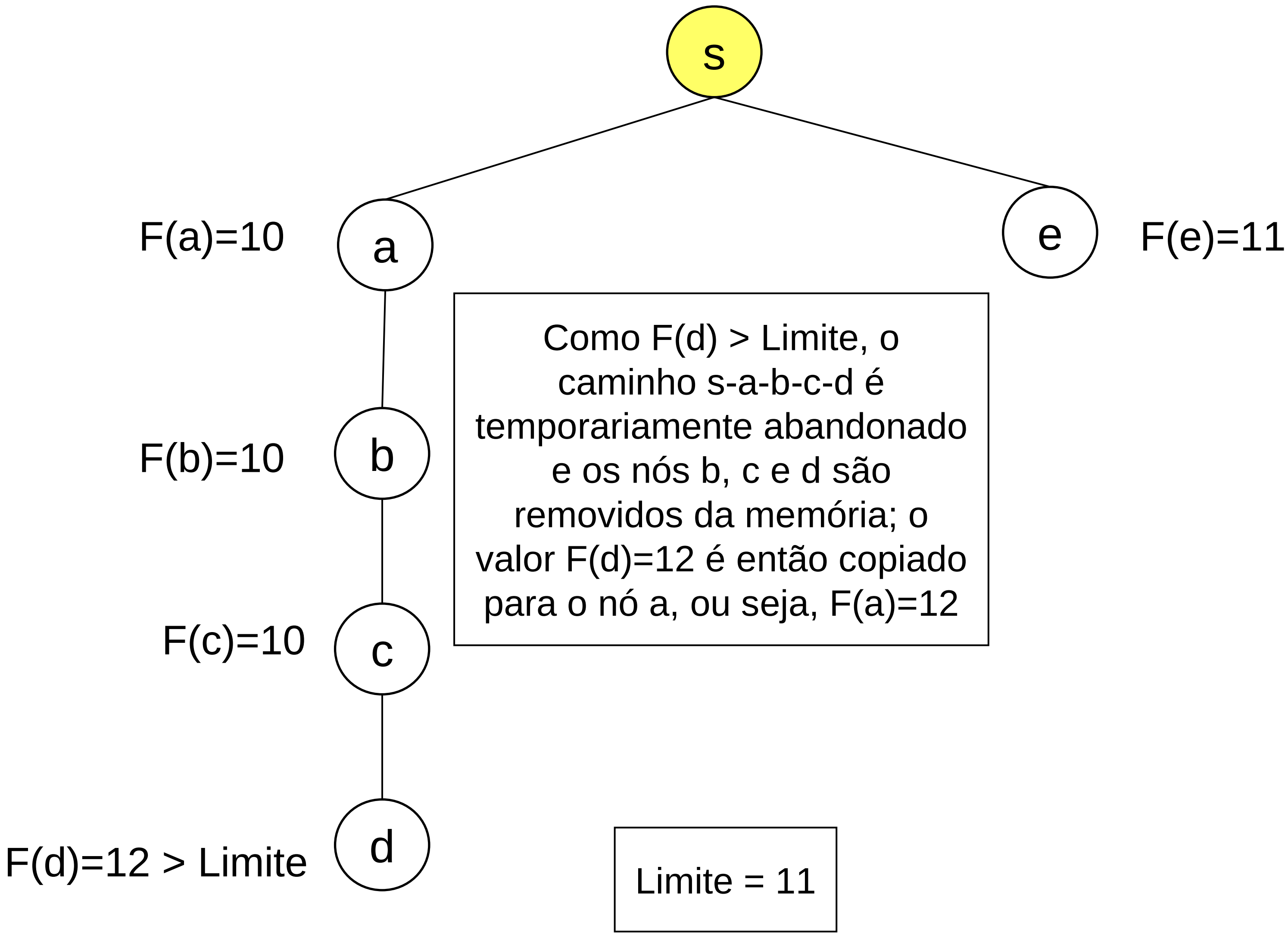
Limite = 11

RBFS

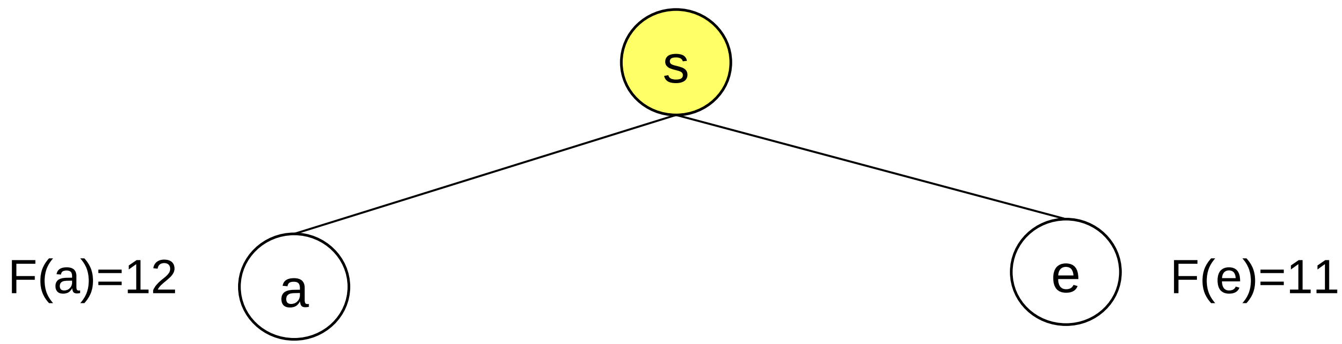


Limite = 11

RBFS

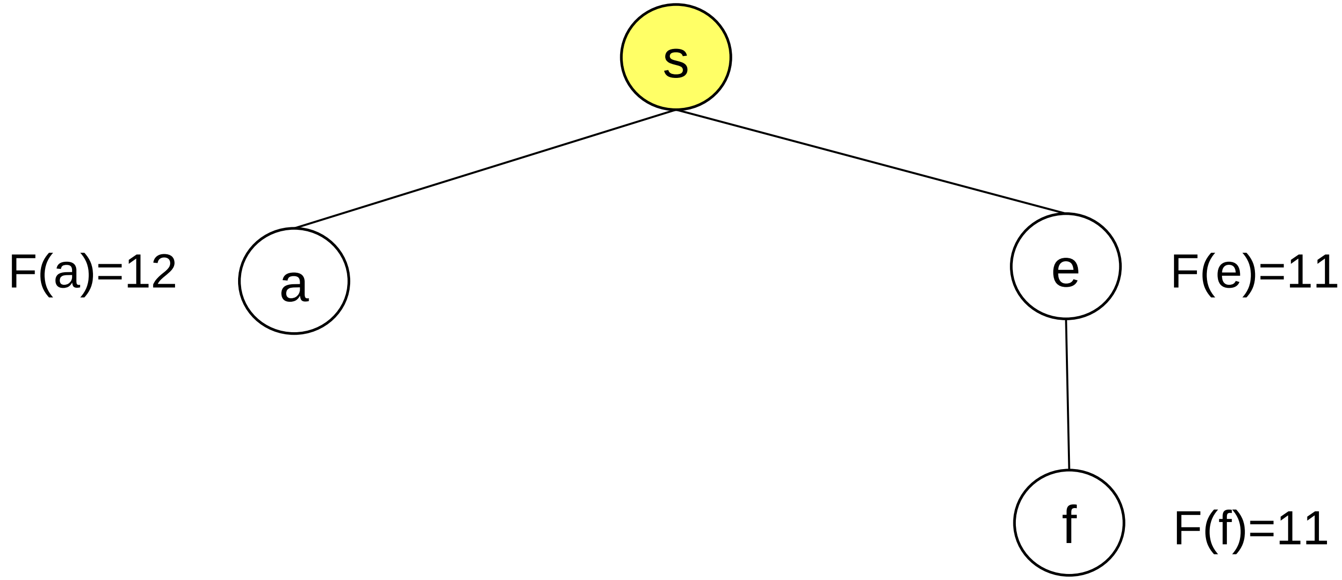

s
F(a)=10
a
e
F(e)=11
F(b)=10
b
F(c)=10
c
F(d)=12 > Limite
d
Como F(d) > Limite, o caminho s-a-b-c-d é temporariamente abandonado e os nós b, c e d são removidos da memória; o valor F(d)=12 é então copiado para o nó a, ou seja, F(a)=12
Limite = 11

RBFS



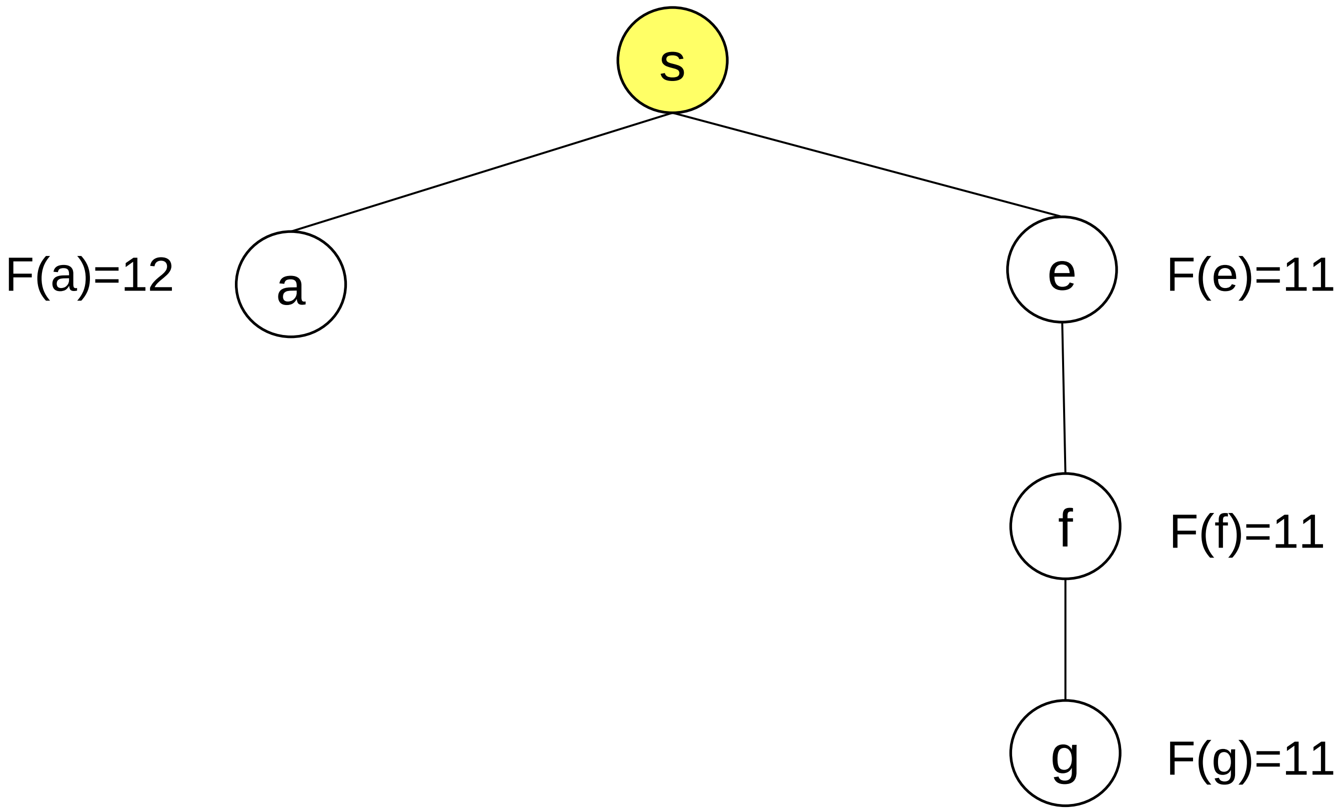
Limite = 12

RBFS



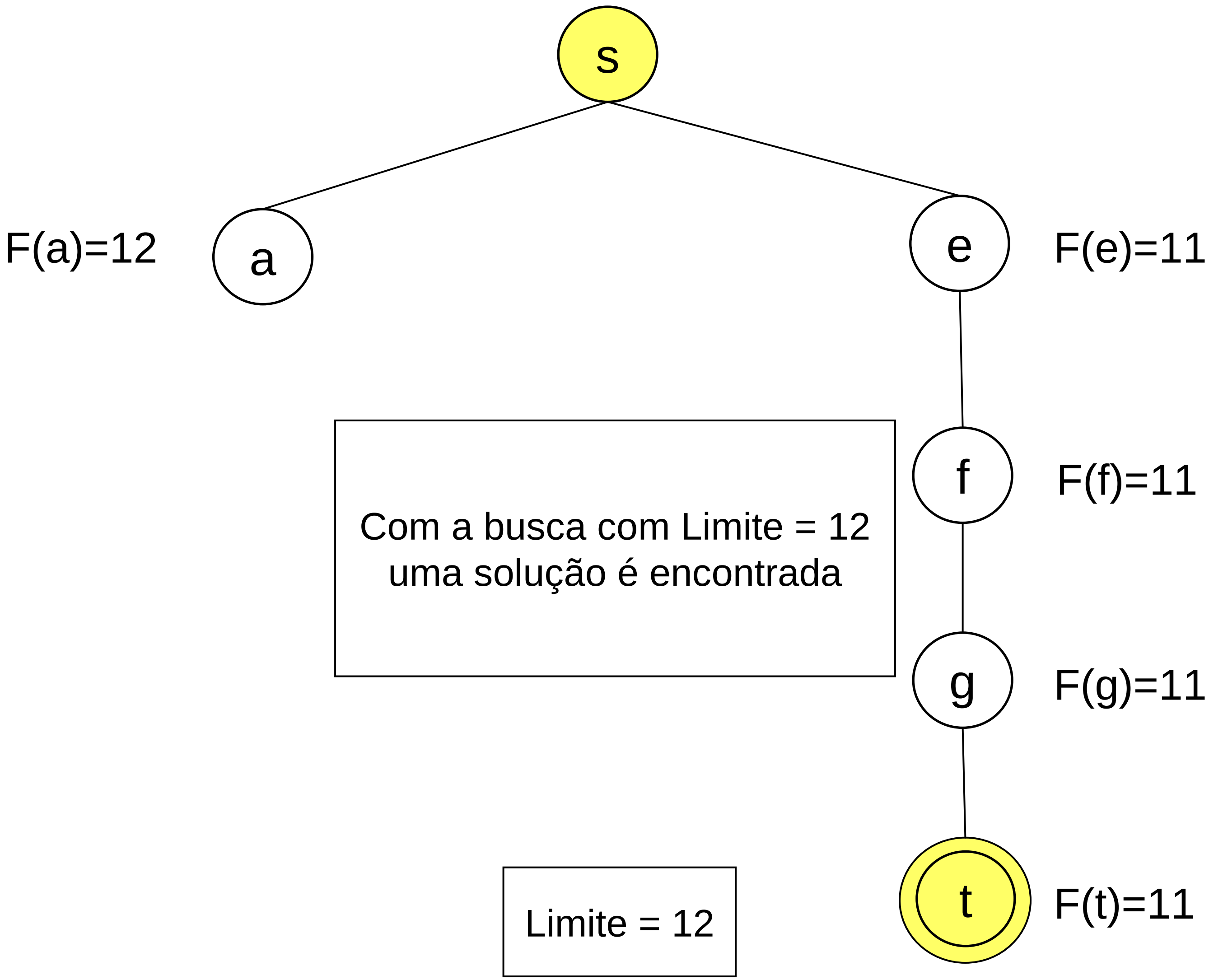
Limite = 12

RBFS



Limite = 12

RBFS


s
F(a)=12
a
e
F(e)=11
f
F(f)=11
Com a busca com Limite = 12
uma solução é encontrada
g
F(g)=11
Limite = 12
t
F(t)=11

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
resolvai3(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).
```

Algoritmos de Busca Local

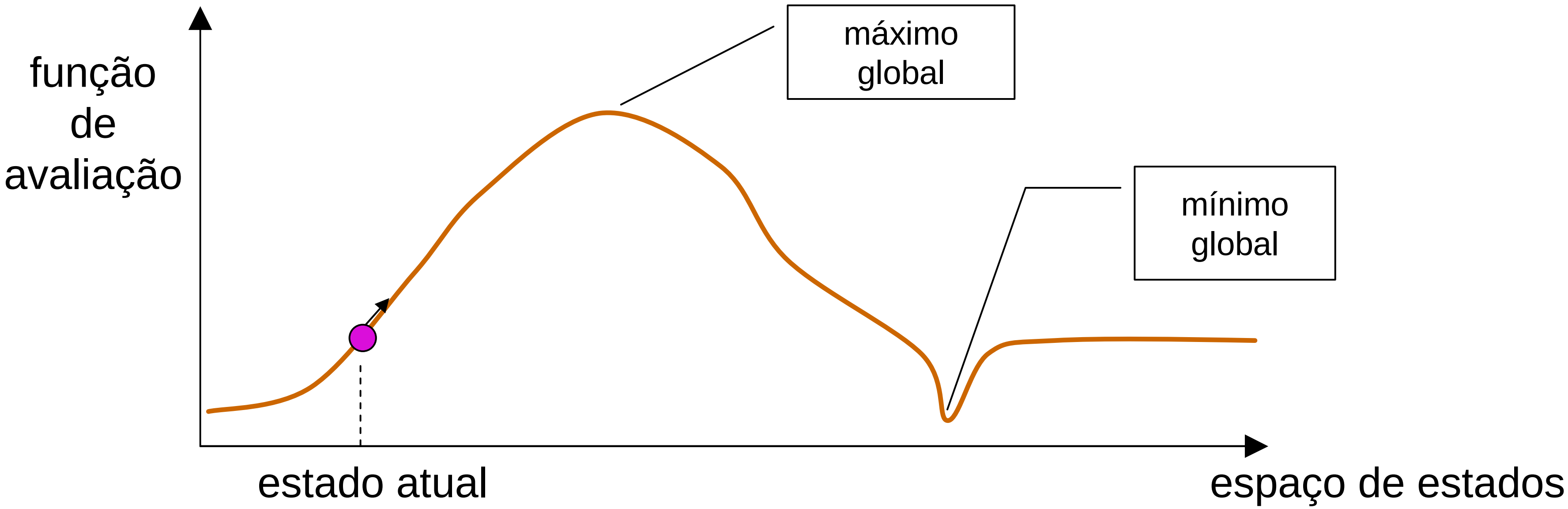
- Os algoritmos de busca (informada ou não) exploram sistematicamente o espaço de busca, mantendo um ou mais caminhos na memória
  - Quando um nó final é encontrado, o caminho até ele constitui uma solução para o problema
- Todavia, em algumas situações, o caminho até o nó final é irrelevante
  - No problema da 8-rainhas o que importa é a configuração final no tabuleiro e não a ordem em que as rainhas são acrescentadas
  - Outros exemplos incluem projetos de circuitos integrados, otimização de rede de comunicação, roteamento de veículos, etc

Algoritmos de Busca Local

- Dessa forma, se o caminho não é importante, podemos considerar uma classe diferente de algoritmos de busca
- Os algoritmos de **busca local** trabalham usando um **único estado corrente** (ao invés de vários caminhos) e, em geral, se movem apenas para os vizinhos desse estado
  - Normalmente, os caminhos não são armazenados
- Embora os algoritmos de busca local não sejam sistemáticos, eles apresentam vantagens:
  - Utilizam pouca memória (em geral, um valor constante)
  - Podem encontrar soluções razoáveis em espaços muito grande ou infinito, para os quais os algoritmos sistemáticos não são adequados
- Além de encontrar estados finais, os algoritmos de busca local são úteis para resolver **problemas de otimização**, nos quais o objetivo é encontrar o melhor estado de acordo com uma função objetivo

Algoritmos de Busca Local

- Uma forma de entender os algoritmos de busca local consiste em considerar a **topologia do espaço de estados** definida pela posição no espaço de estados versus a elevação, definida pelo valor da função de avaliação
  - Se a elevação corresponder ao custo, o objetivo será encontrar um **mínimo global** (vale)
  - Se a elevação corresponder a uma função objetivo que deve ser maximizada, então o objetivo será encontrar um **máximo global** (pico)



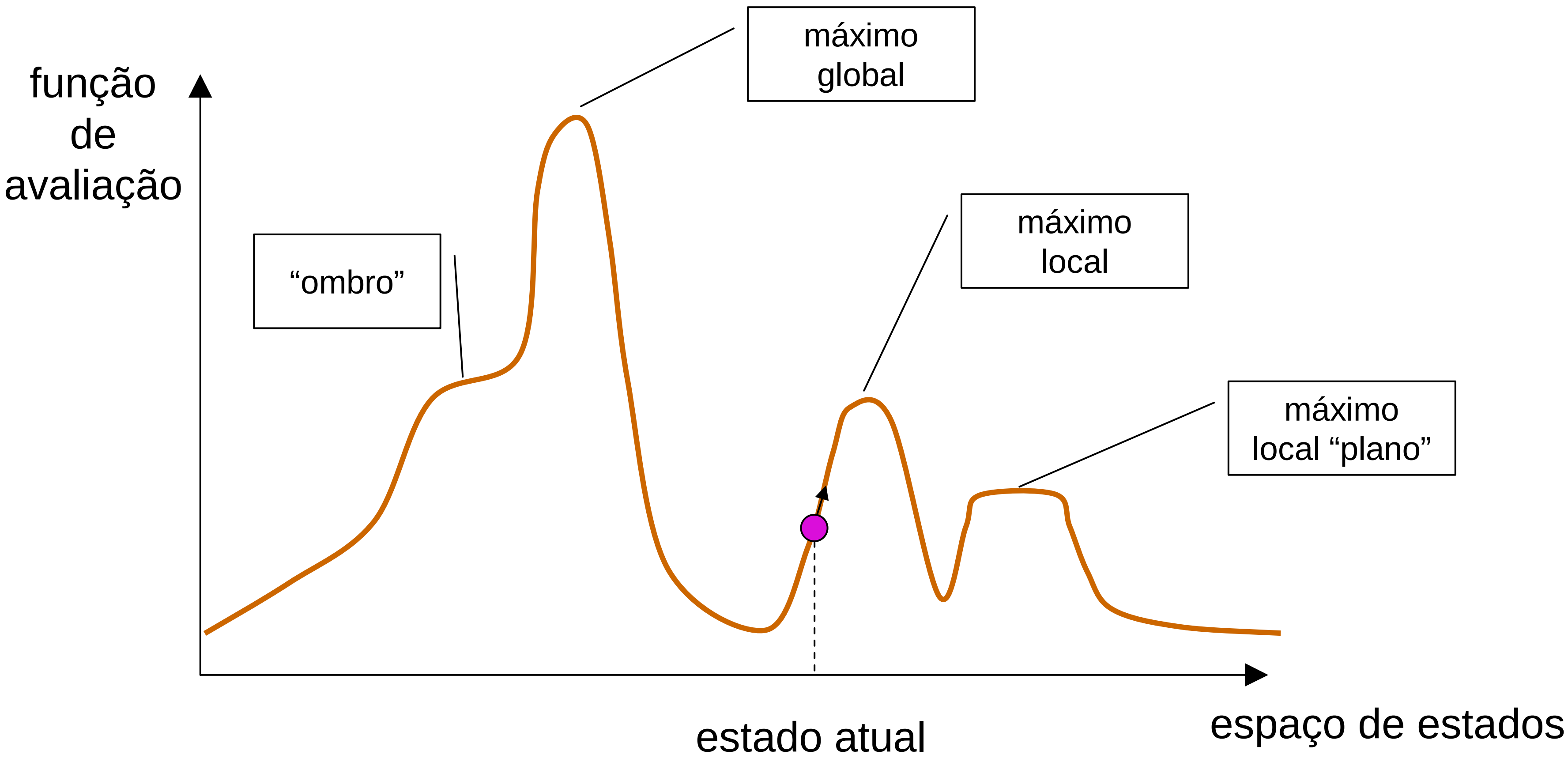
Algoritmo de Subida de Encosta

- O algoritmo de **subida de encosta** (***hill-climbing***) se move de forma contínua no sentido crescente da função de avaliação, ou seja, encosta acima; o algoritmo termina ao encontrar um pico (um máximo)
  - O algoritmo também é conhecido como **gradiente descendente**, se o objetivo é minimizar a função de avaliação
- O algoritmo não mantém uma árvore de busca mas somente o estado atual e o valor de sua função de avaliação
- O algoritmo não explora valores da função de avaliação além dos vizinhos imediatos ao estado atual
  - "É como escalar o monte Everest em um nevoeiro denso com amnésia"
  - "É como usar óculos que limitam sua visão a 3 metros"

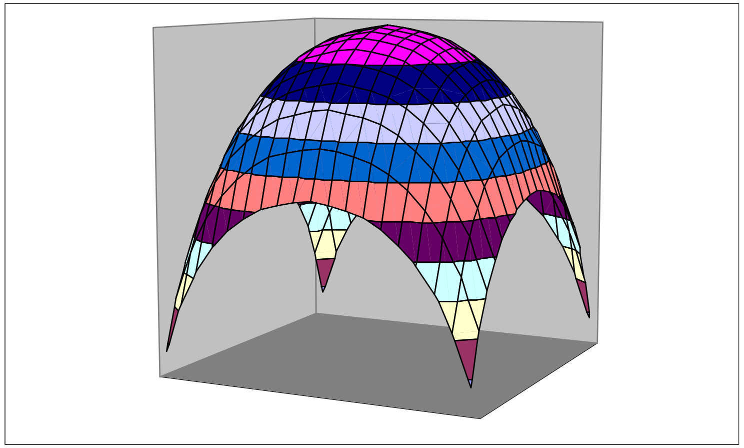
Algoritmo de Subida de Encosta

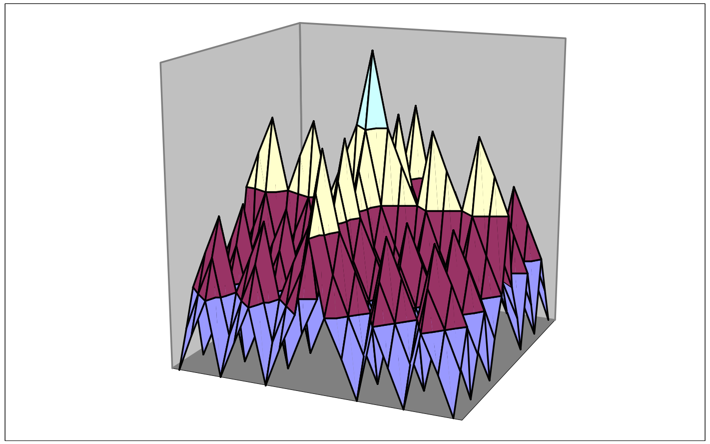
- O algoritmo também é chamado de busca gulosa local, pois captura um bom estado vizinho sem decidir com antecedência para onde irá em seguida
- O algoritmo tende a ser rápido ao encontrar uma solução pois é sempre fácil melhorar um estado ruim
- Problemas
  - Máximo local: uma vez atingido, o algoritmo termina mesmo que a solução esteja longe de ser satisfatória
  - Platôs (regiões planas): regiões onde a função de avaliação é essencialmente plana; isso pode impedir o algoritmo de encontrar uma saída do platô
  - Picos: vários máximos locais não interligados no espaço de estados, o que dificulta a navegação pelo algoritmo

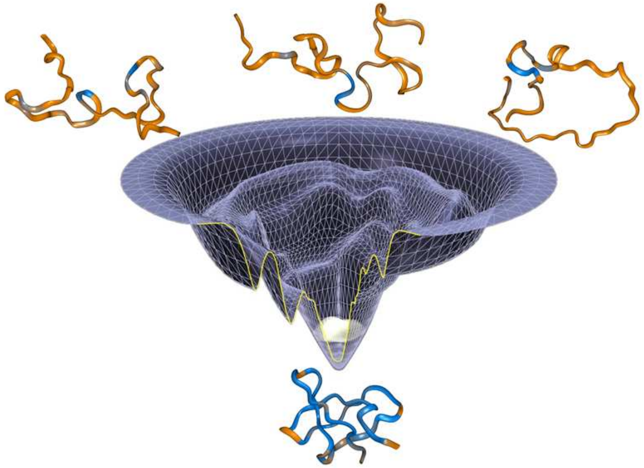
Algoritmo de Subida de Encosta

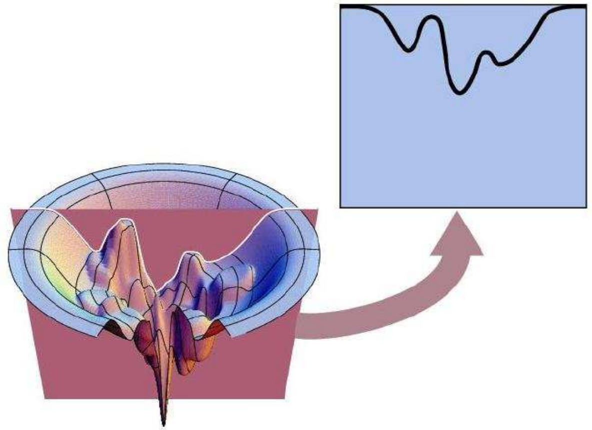


Algoritmo de Subida de Encosta









Algoritmo de Subida de Encosta

1. Escolha um estado inicial do espaço de busca (pode ser escolhido de forma aleatória)
2. Considere todos os vizinhos (sucessores) no espaço de busca
3. Escolha o vizinho com a melhor qualidade (função de avaliação) e mova para aquele estado
4. Repita os passos de 2 até 4 até que todos os estados vizinhos tenham qualidade menor 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

Algoritmo de Subida de Encosta

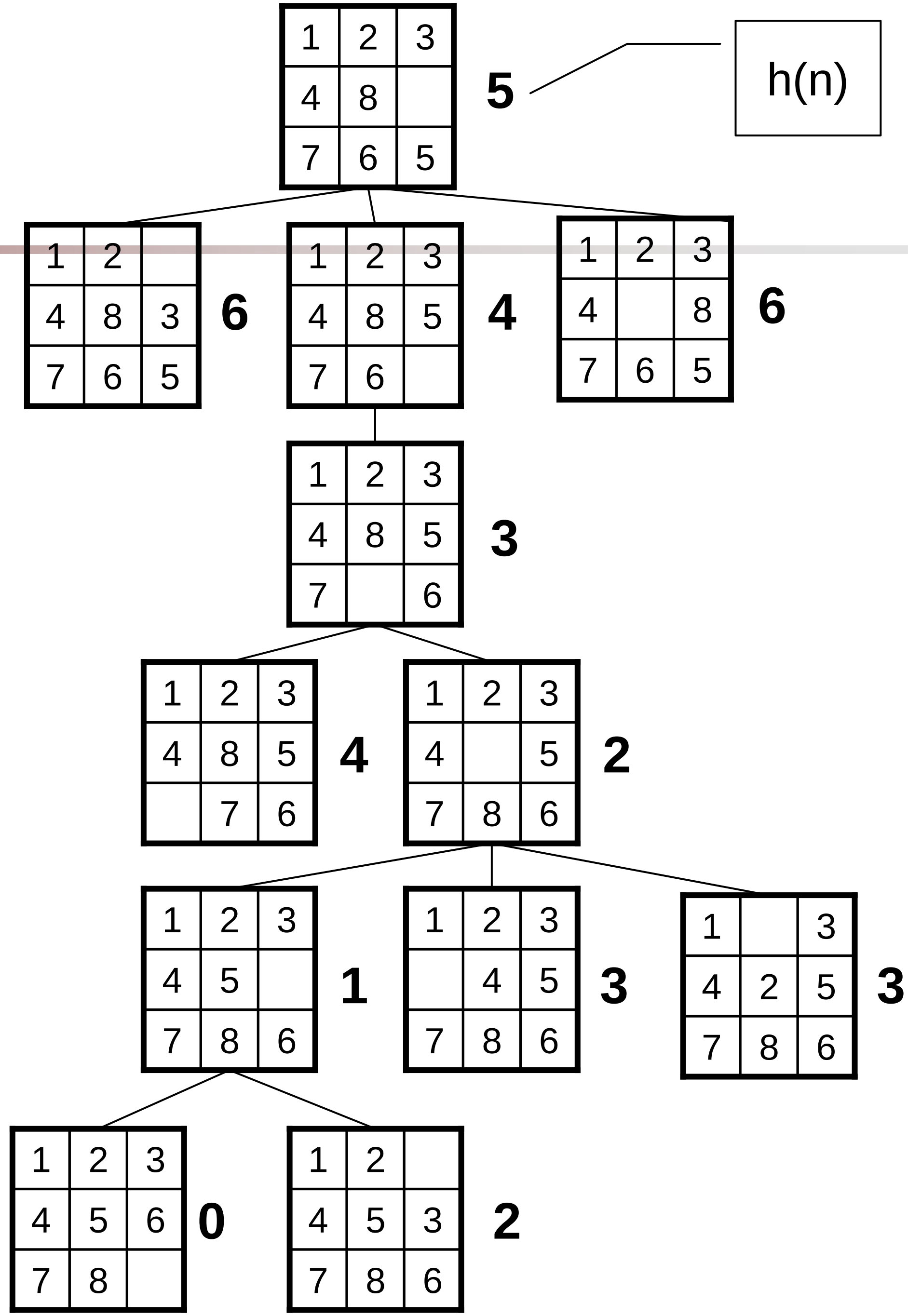
- O algoritmo dado a seguir consiste numa extensão do algoritmo básico de subida de encosta
  - Detectando ciclos
  - Retrocedendo se não encontrar uma solução
- O algoritmo foi deixado o mais próximo possível aos já vistos, mas pode ser otimizado de várias formas
  - O retrocesso pode ser eliminado (por exemplo, por um corte)
  - A fila de prioridade pode ser substituída por um algoritmo de ordenação (por exemplo, *quicksort*)
  - De fato, a fila de prioridade pode ser eliminada, pois o que importa é somente o sucessor de melhor qualidade (entretanto, se a fila for removida isso também elimina a possibilidade de retrocesso)
  - Tais otimizações são deixadas como exercícios

Hill-Climbing (com retrocesso e detecção de ciclos)

```
resolvai4(No,Solucao) :-
  hillclimbing([No]:0,Solucao).

hillclimbing([No|Caminho]:H,[No|Caminho]) :-
  final(No).
hillclimbing(Caminho:H,Solucao) :-
  estender(Caminho,NovosCaminhos),
  fila_prioridade(NovosCaminhos,[],Caminhos1),  % ordenacao por h
  pertence(Melhor,Caminhos1),                   % seleciona melhor vizinho
  hillclimbing(Melhor,Solucao).                 % e continua a partir dele

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

% fila_prioridade(NovosCaminhos,CaminhosExistentes,CaminhosOrdenados)
% concatena (com prioridade) os NovosCaminhos em CaminhosExistenes formando CaminhosOrdenados
% assumindo que CaminhosExistentes ja esta ordenado por prioridade (ou seja, por h)
fila_prioridade([],L,L).
fila_prioridade([Caminho:H|Caminhos],Existente,Final) :-
  insert(Caminho:H,Existente,ExistenteAumentado),
  fila_prioridade(Caminhos,ExistenteAumentado,Final).

insert(Caminho:H,[Path:Hs|Paths],[Path:Hs|NewPaths]) :-
  H > Hs, !,
  insert(Caminho:H,Paths,NewPaths).
insert(C,Paths,[C|Paths]).
```

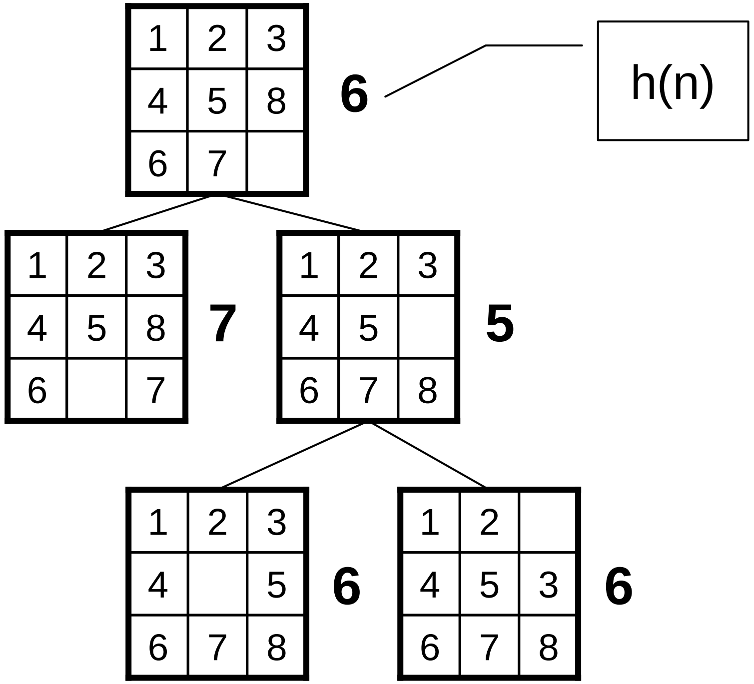
Exemplo 1

- Neste exemplo, a heurística distância Manhattan permite encontrar uma solução rapidamente por hill-climbing



Exemplo 2

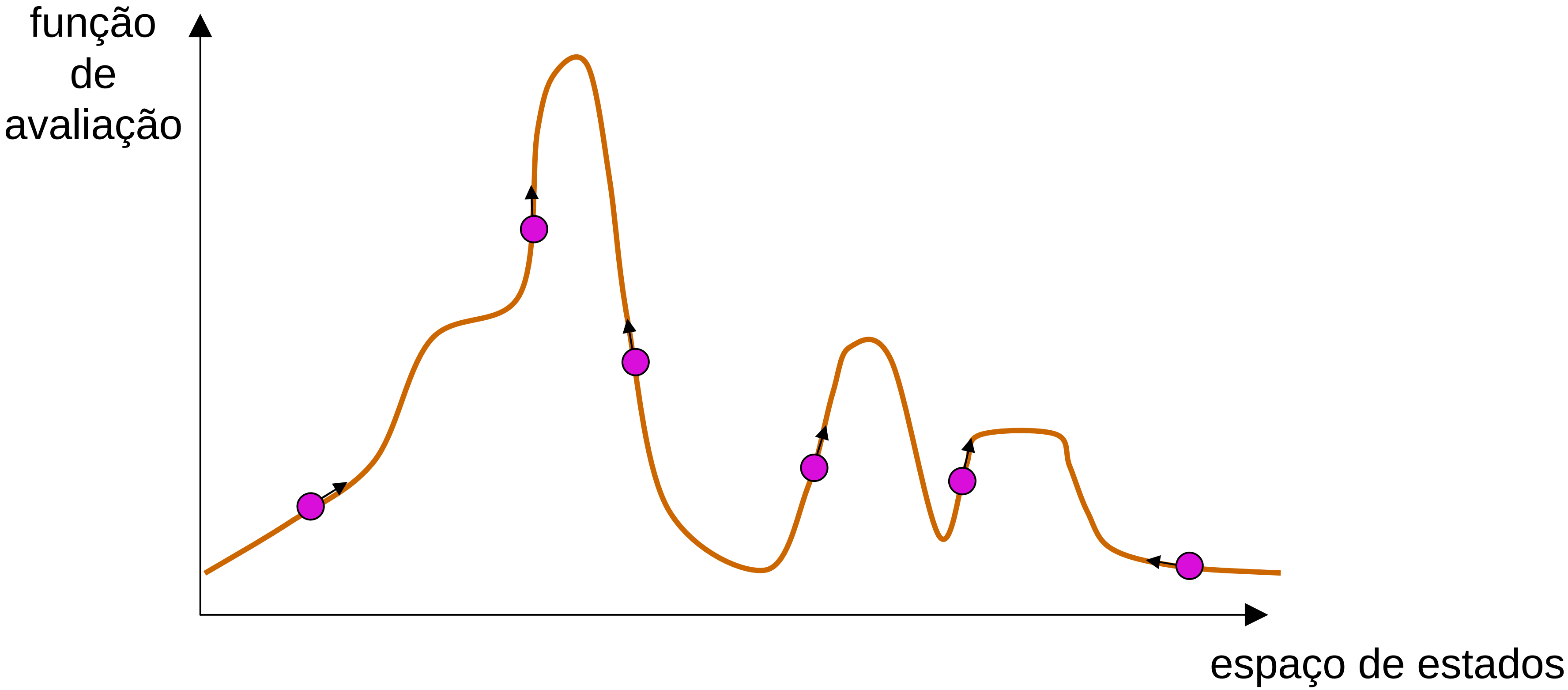
- Neste exemplo, com a mesma heurística, hill-climbing não consegue encontrar uma solução
- Todos os nós vizinhos (sucessores) têm valores maiores (mínimo local)
- Observe que este jogo tem solução em apenas mais 12 movimentos



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

Busca em Feixe Local

- O algoritmo de busca em feixe local (*beam search*) mantém k estados em memória
- Inicialmente, os k estados são escolhidos de forma aleatória
- Em cada passo, são gerados todos os sucessores de todos os k estados
  - Se algum deles for o estado final, então o algoritmo termina
  - Caso contrário, o algoritmo seleciona apenas os k melhores sucessores a partir da lista completa e repete o processo

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 2009