Busca Cega (Exaustiva) e Heurística

Busca – Aula 2

Ao final desta aula a gente deve saber:

- Conhecer as várias estratégias de realizar Busca não-informada (Busca Cega)
- Determinar que estratégia se aplica melhor ao problema que queremos solucionar
- Evitar a geração de estados repetidos.
- Entender o que é Busca Heurística
- Saber escolher heurísticas apropriadas para o problema.

Busca Cega (Exaustiva)

- Estratégias para determinar a ordem de expansão dos nós

1. Busca em largura
2. Busca de custo uniforme
3. Busca em profundidade
4. Busca com aprofundamento iterativo

Critérios de Avaliação das Estratégias de Busca

- Completude (completeza):
  - a estratégia **sempre** encontra uma solução quando existe alguma?
- Custo do tempo:
  - quanto **tempo** gasta para encontrar uma solução?
- Custo de memória:
  - quanta **memória** é necessária para realizar a busca?
- Qualidade/otimalidade (*optimality*):
  - a estratégia encontra **a melhor solução** quando existem soluções diferentes?
    - menor custo de caminho

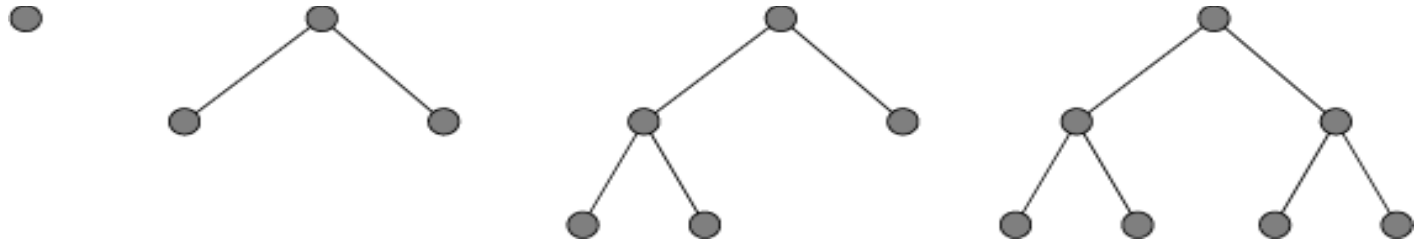
Busca em Largura

- Ordem de expansão dos nós:
  1. Nó raiz
  2. Todos os nós de profundidade 1
  3. Todos os nós de profundidade 2, etc…

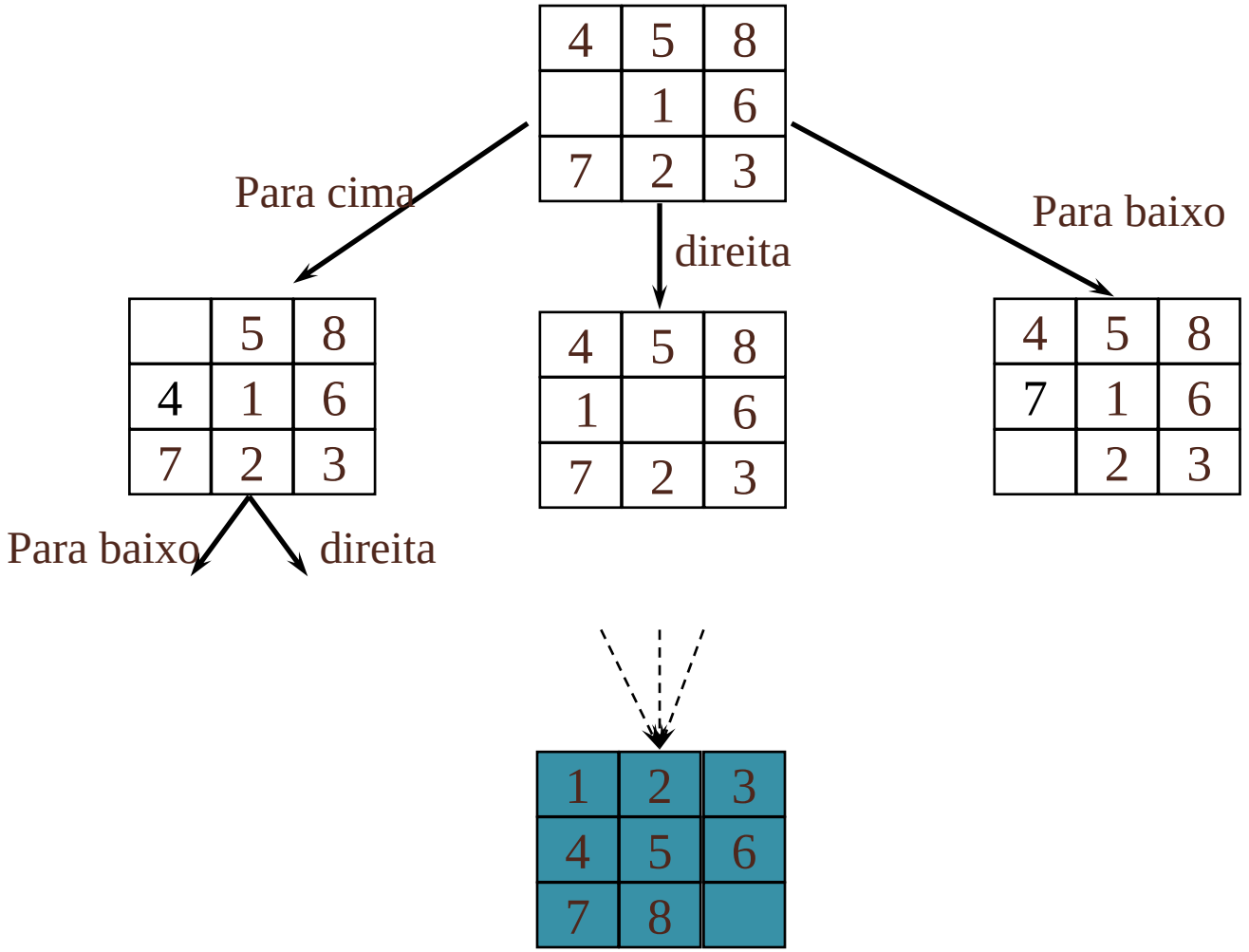
- Algoritmo:

função **Busca-em-Largura** (***problema***)
retorna **uma solução ou falha**

Busca-Genérica (*problema*, Insere-no-Fim)



Exemplo: Jogo dos 8 números



Busca em Largura
Qualidade

- Esta estratégia é *completa*
- É *ótima ?*
  - Sempre encontra a solução mais “rasa”

    → que nem sempre é a solução de menor **custo de caminho,** caso os operadores tenham valores diferentes.
- É *ótima* se
  - $\forall n,n'$ profundidade(n’) $\geq$ profundidade(n) $\Rightarrow$

    custo de caminho(n’) $\geq$ custo de caminho (n).
    - A função **custo de caminho** é não-decrescente com a profundidade do nó.
    - Essa função acumula o custo do caminho da origem ao nó atual.
  - Geralmente, isto só ocorre quando todos os operadores têm o mesmo custo (=1)

Busca em Largura
Custo

- Fator de expansão da árvore de busca:
  - número de nós gerados a partir de cada nó (b)
- Custo de tempo:
  - se o fator de expansão do problema = b, e a primeira solução para o problema está no nível d,
  - então o número máximo de nós gerados até se encontrar a solução = $1 + b + b^2 + b^3 + \ldots + b^d$
  - **custo exponencial** = $O(b^d)$.
- Custo de memória:
  - a *fronteira* do espaço de estados deve permanecer na memória
  - é um problema mais crucial do que o tempo de execução da busca

Busca em Largura

- Esta estratégia só dá bons resultados quando a *profundidade* da árvore de busca é *pequena*.
- Exemplo:
  - fator de expansão b = 10
  - 1.000 nós gerados por segundo
  - cada nó ocupa 100 bytes

Profundidade	Nós	Tempo	Memória
0	1	1 milissegundo	100 bytes
2	111	0.1 segundo	11 quilobytes
4	11111	11 segundos	1 megabytes
6	10^6	18 minutos	111 megabytes
8	10^8	31 horas	11 gigabytes
10	10^{10}	128 dias	1 terabyte
12	10^{12}	35 anos	111 terabytes
14	10^{14}	3500 anos	11111 terabytes

Busca de Custo Uniforme

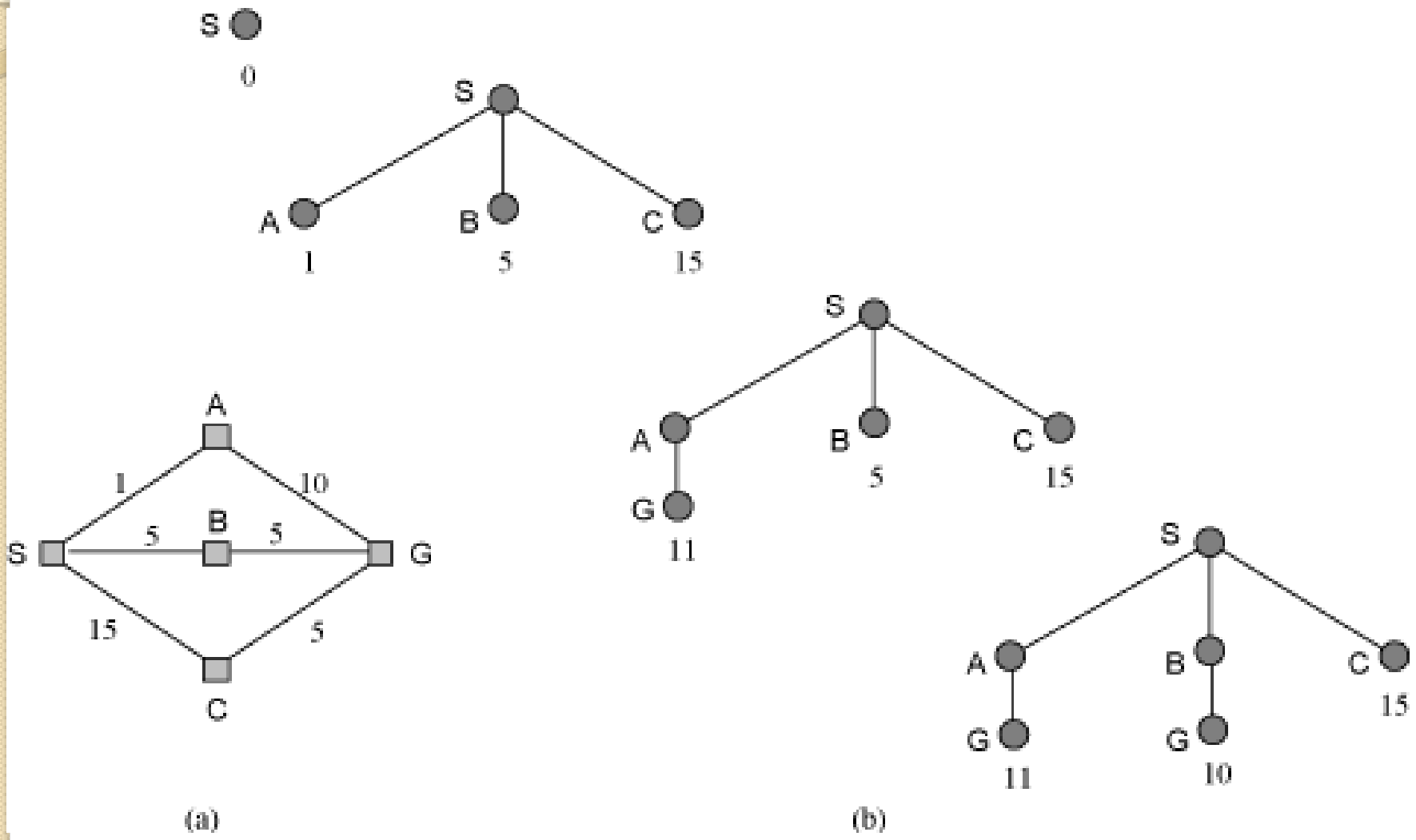
- Modifica a busca em largura:
  - expande o nó da fronteira com menor custo de caminho na fronteira do espaço de estados
  - cada operador pode ter um custo associado diferente, medido pela função *g(n)*, para o nó *n*.
    - onde g(n) dá o custo do caminho da origem ao nó n
- Na busca em largura: ***g(n) = profundidade (n)***
- Algoritmo:

função **Busca-de-Custo-Uniforme (*problema*)**

retorna **uma solução ou falha**

Busca-Genérica (*problema*, Insere-Ordem-Crescente)

Busca de Custo Uniforme



Busca de Custo Uniforme
Fronteira do exemplo anterior

- F = {S}
  - testa se S é o estado objetivo, expande-o e guarda seus filhos A, B e C ordenadamente na fronteira
- F = {A, B, C}
  - testa A, expande-o e guarda seu filho G_A ordenadamente
  - **obs.:** o algoritmo de geração e teste guarda na fronteira todos os nós gerados, testando se um nó é o objetivo apenas quando ele é retirado da lista!
- F= {B, G_A, C}
  - testa B, expande-o e guarda seu filho G_B ordenadamente
- F= {G_B, G_A, C}
  - testa G_B e para!

Busca de Custo Uniforme

- Esta estratégia é *completa*
- É *ótima* se
  - g (sucessor(n)) $\geq$ g (n)
    - custo de caminho **no mesmo caminho** não decresce
    - i.e., não tem operadores com **custo negativo**
  - caso contrário, teríamos que expandir todo o espaço de estados em busca da melhor solução.
    - Ex. Seria necessário expandir também o nó C do exemplo, pois o próximo operador poderia ter custo associado = -13, por exemplo, gerando um caminho mais barato do que através de B
- Custo de tempo e de memória
  - teoricamente, igual ao da Busca em Largura

Busca em Profundidade

- Ordem de expansão dos nós:
  - sempre expande o nó no *nível mais profundo* da árvore:

    1. nó raiz
    2. primeiro nó de profundidade 1
    3. primeiro nó de profundidade 2, etc.…

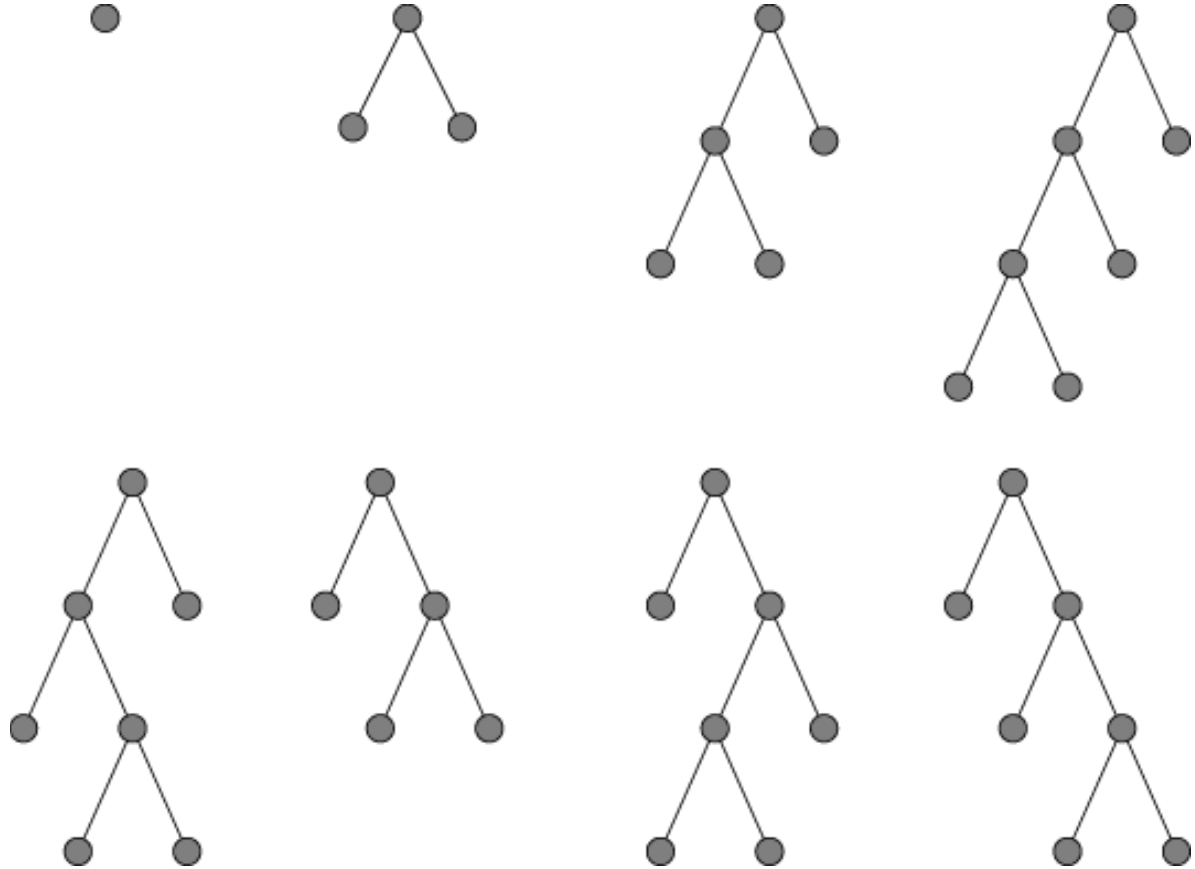
  - Quando um nó final não é solução, o algoritmo volta para expandir os nós que ainda estão na fronteira do espaço de estados
- Algoritmo:

função Busca-em-Profundidade (*problema*)

retorna **uma solução ou falha**

Busca-Genérica (*problema*, Insere-no-Começo)

Busca em Profundidade



Busca em Profundidade

- Esta estratégia não é *completa* nem é *ótima*.
- Custo de memória:
  - mantém na memória o caminho sendo expandido no momento, e os nós irmãos dos nós no caminho (para possibilitar o *backtracking*)
    - necessita armazenar apenas $b.m$ nós para um espaço de estados com fator de expansão b e profundidade m, onde m pode ser maior que d (profundidade da 1a. solução)
- Custo de tempo: $O(b^m)$, no pior caso.
- Observações:
  - Para problemas com várias soluções, esta estratégia pode ser bem mais rápida do que busca em largura.
  - Esta estratégia deve ser evitada quando as árvores geradas são muito *profundas* ou geram *caminhos infinitos*.

Busca com Aprofundamento Iterativo

- Evita o problema de caminhos muito longos ou infinitos impondo um limite máximo *(l)* de profundidade para os caminhos gerados.
  - $l \geq d$, onde *l* é o limite de profundidade e *d* é a profundidade da primeira solução do problema

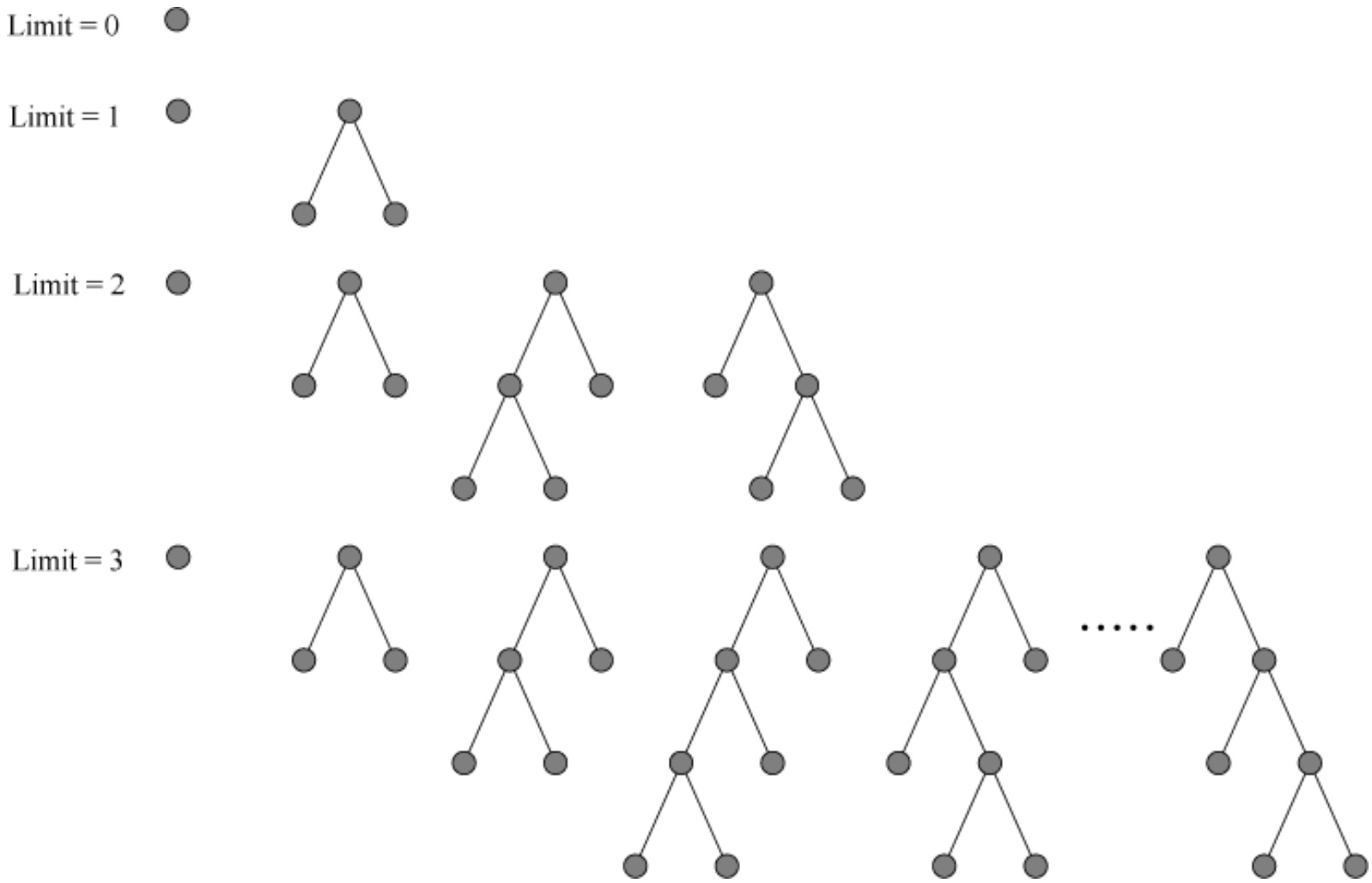
Busca com Aprofundamento Iterativo

- Esta estratégia tenta limites com valores crescentes, partindo de zero, até encontrar a primeira solução
  - fixa profundidade = *i*, executa busca
  - se não chegou a um objetivo, recomeça busca com profundidade = *i* + *n* (*n* qualquer)
  - piora o tempo de busca, porém melhora o custo de memória!
- Igual à Busca em Largura para *i=1* e *n=1*

Busca com Aprofundamento Iterativo

- Combina as vantagens de *busca em largura* com *busca em profundidade.*
- É *ótima* e *completa*
  - com n = 1 e operadores com custos iguais
- Custo de memória:
  - necessita armazenar apenas *b.d* nós para um espaço de estados com fator de expansão *b* e limite de profundidade *d*
- Custo de tempo:
  - $O(b^d)$
- Bons resultados quando o espaço de estados é *grande* e de *profundidade desconhecida*.

Busca com Aprofundamento Iterativo



Comparando Estratégias de Busca Exaustiva

Critério	Largura	Custo Uniforme	Profun-didade	Aprofun-damento Iterativo
Tempo	b^d	b^d	b^m	b^d
Espaço	b^d	b^d	bm	bd
Otima?	Sim	Sim*	Não	Sim
Completa?	Sim	Sim	Não	Sim

Como Evitar Geração de Estados Repetidos?

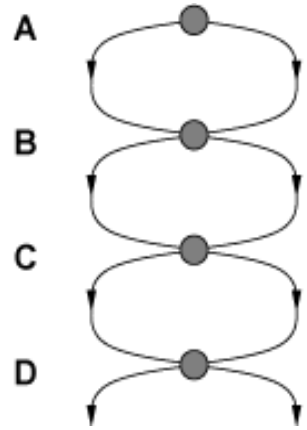
Evitar Geração de Estados Repetidos

- Problema geral em Busca
  - expandir estados presentes em caminhos já explorados
- É inevitável quando existe operadores reversíveis
  - ex. encontrar rotas, canibais e missionários, 8-números, etc.
  - a árvore de busca é potencialmente infinita
- Idéia
  - **podar** (*prune*) estados repetidos, para gerar apenas a parte da árvore que corresponde ao grafo do espaço de estados (que é finito!)
  - mesmo quando esta árvore é finita, evitar estados repetidos pode reduzir exponencialmente o custo da busca

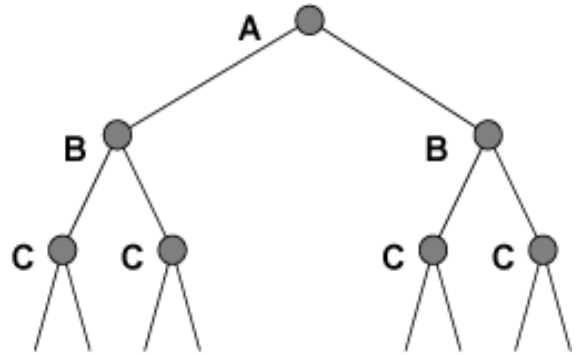
Evitar Geração de Estados Repetidos

- Exemplo:
  - (m + 1) estados no espaço => 2^m caminhos na árvore

Espaço de estados



Árvore de busca



- Questões
  - Como evitar expandir estados presentes em caminhos já explorados?
  - Em ordem crescente de eficácia e custo computacional?

Evitando operadores reversíveis

- se os operadores são reversíveis:
  - conjunto de predecessores do nó = conjunto de sucessores do nó
  - porém, esses operadores podem gerar árvores *infinitas*!

Como Evitar Estados Repetidos ? Algumas Dicas

1. Não retornar ao estado “pai”
   - função que rejeita geração de sucessor igual ao pai
2. Não criar caminhos com ciclos
   - não gerar sucessores para qualquer estado que já apareceu no caminho sendo expandido
3. Não gerar qualquer estado que já tenha sido criado antes (em qualquer ramo)
   - requer que todos os estados gerados permaneçam na memória
   - custo de memória: $O(b^d)$
   - pode ser implementado mais eficientemente com *hash tables*

Conflito (*trade-off*)

- Problema:
  - Custo de armazenamento e verificação ✗ Custo extra de busca
- Solução
  - depende do problema
  - quanto mais “loops”, mais vantagem em evitá-los!

A seguir...

- Busca heurística

Estratégias de Busca Exaustiva (Cega)

- Encontram soluções para problemas pela geração *sistemática* de novos estados, que são comparados ao objetivo;
- São *ineficientes* na maioria dos casos:
  - utilizam apenas o *custo de caminho* do nó atual ao nó inicial (função *g*) para decidir qual o próximo nó da fronteira a ser expandido.
  - essa medida nem sempre conduz a busca na direção do objetivo.
- Como encontrar um barco perdido?
  - não podemos procurar no oceano inteiro...
  - observamos as correntes marítimas, o vento, etc...

Estratégias Busca Heurística (Informada)

- Utilizam conhecimento específico do problema na escolha do próximo nó a ser expandido
  - barco perdido
    - correntes marítimas, vento, etc...
- Aplicam de uma função de avaliação a cada nó na fronteira do espaço de estados
  - essa função estima o custo de caminho do nó atual até o objetivo mais próximo utilizando uma função heurística.
  - Função heurística
    - estima o custo do caminho mais barato do estado atual até o estado final mais próximo.

Busca Heurística

- Classes de algoritmos para busca heurística:

1. Busca pela melhor escolha (*Best-First search*)
2. Busca com limite de memória
3. Busca com melhora iterativa

Busca pela Melhor Escolha
Best-First Search

- Busca pela Melhor Escolha - BME
  - Busca genérica onde o nó de menor custo "aparente" na fronteira do espaço de estados é expandido primeiro
- Duas abordagens básicas:
  - 1. Busca Gulosa (Greedy search)
  - 2. Algoritmo A*

Busca pela Melhor Escolha

Algoritmo geral

- Função-Insere
  - insere novos nós na fronteira ordenados com base na Função-Avaliação
    - Que está baseada na função heurística

função Busca-Melhor-Escolha (*problema*, *Função-Avaliação*)

retorna **uma solução**

Busca-Genérica (*problema, Função-Insere*)

BME: Busca Gulosa

- Semelhante à busca em profundidade com *backtracking*
- Tenta expandir o nó mais próximo do nó final com base na estimativa feita pela função heurística *h*
- *Função-Avaliação*
  - função heurística *h*

Funções Heurísticas

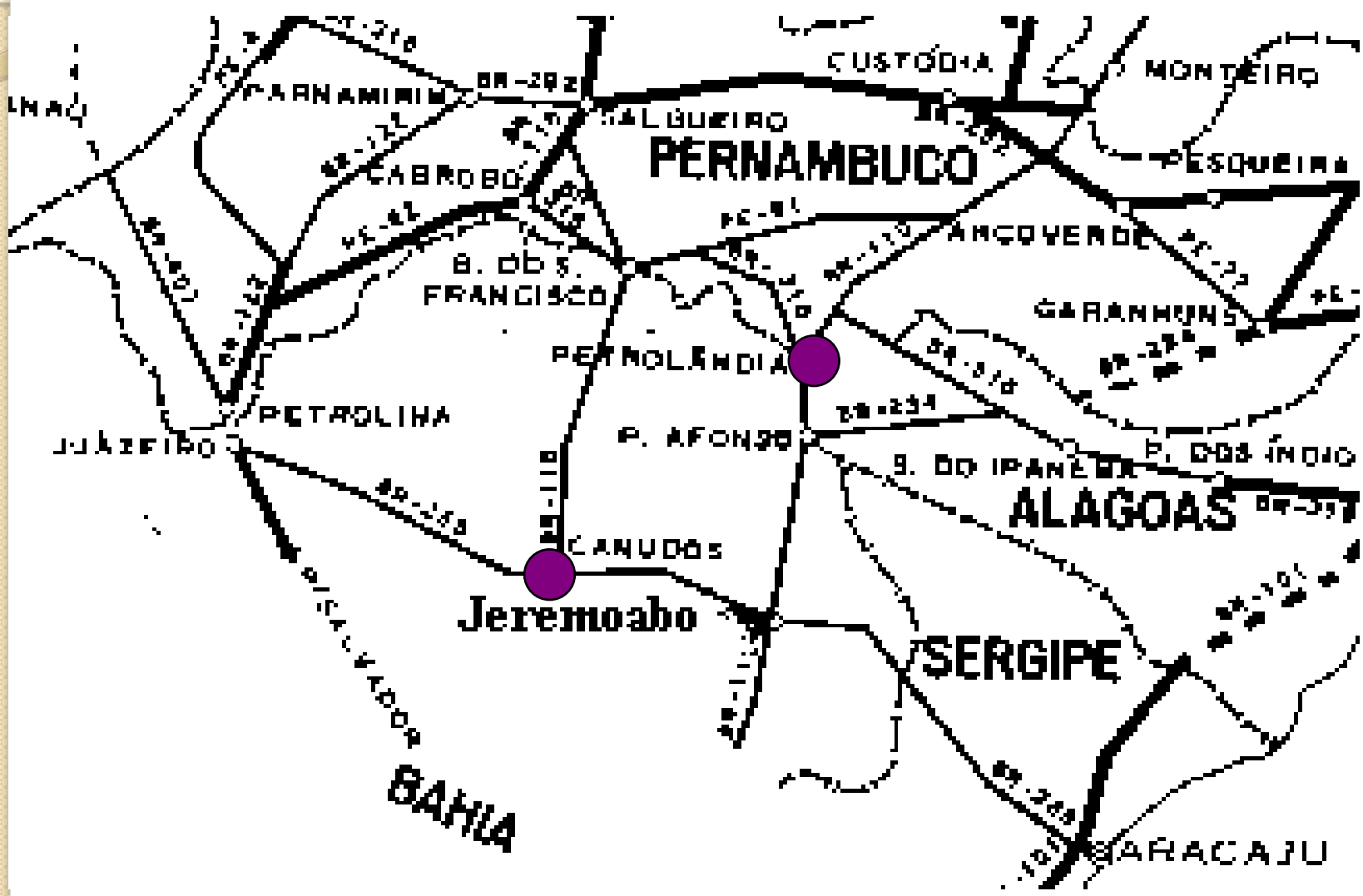
- Função heurística - *h*
  - estima o custo do caminho mais barato do estado atual até o estado final mais próximo.
- Funções heurísticas são específicas para cada problema
- Exemplo:
  - encontrar a rota mais barata de Canudos a Petrolândia
  - $h_{dd}(n)$ = distância direta entre o nó *n* e o nó final.

Funções Heurísticas

- Como escolher uma boa função heurística?
  - ela deve ser admissível
  - i.e., nunca *superestimar* o custo real da solução
- Distância direta (h_{dd}) é *admissível* porque o caminho mais curto entre dois pontos é sempre uma linha reta
- Veremos mais sobre isso na próxima aula

Exemplo: encontrar a rota mais barata de Canudos a Petrolândia

hdd(n) = distância direta entre o nó *n* e o nó final



Busca Gulosa

- Custo de busca mínimo!
  - não expande nós fora do caminho
- Porém *não* é ótima:
  - escolhe o caminho que é mais econômico à primeira vista
    - Belém do S. Francisco, Petrolândia = 4,4 unidades
  - porém, existe um caminho mais curto de Canudos a Petrolândia
    - Jeremoabo, P. Afonso, Petrolândia = 4 unidades
- A solução via Belém do S. Francisco foi escolhida por este algoritmo porque
  - $h_{dd}(BSF) = 1,5$ *u.,* enquanto $h_{dd}(Jer) = 2,1$ u.

Busca Gulosa

- Não é completa:
  - pode entrar em *looping* se não detectar a expansão de estados repetidos
  - pode tentar desenvolver um caminho infinito

- Custo de tempo e memória: $O(b^d)$
  - guarda todos os nós expandidos na memória

BME: Algoritmo A*

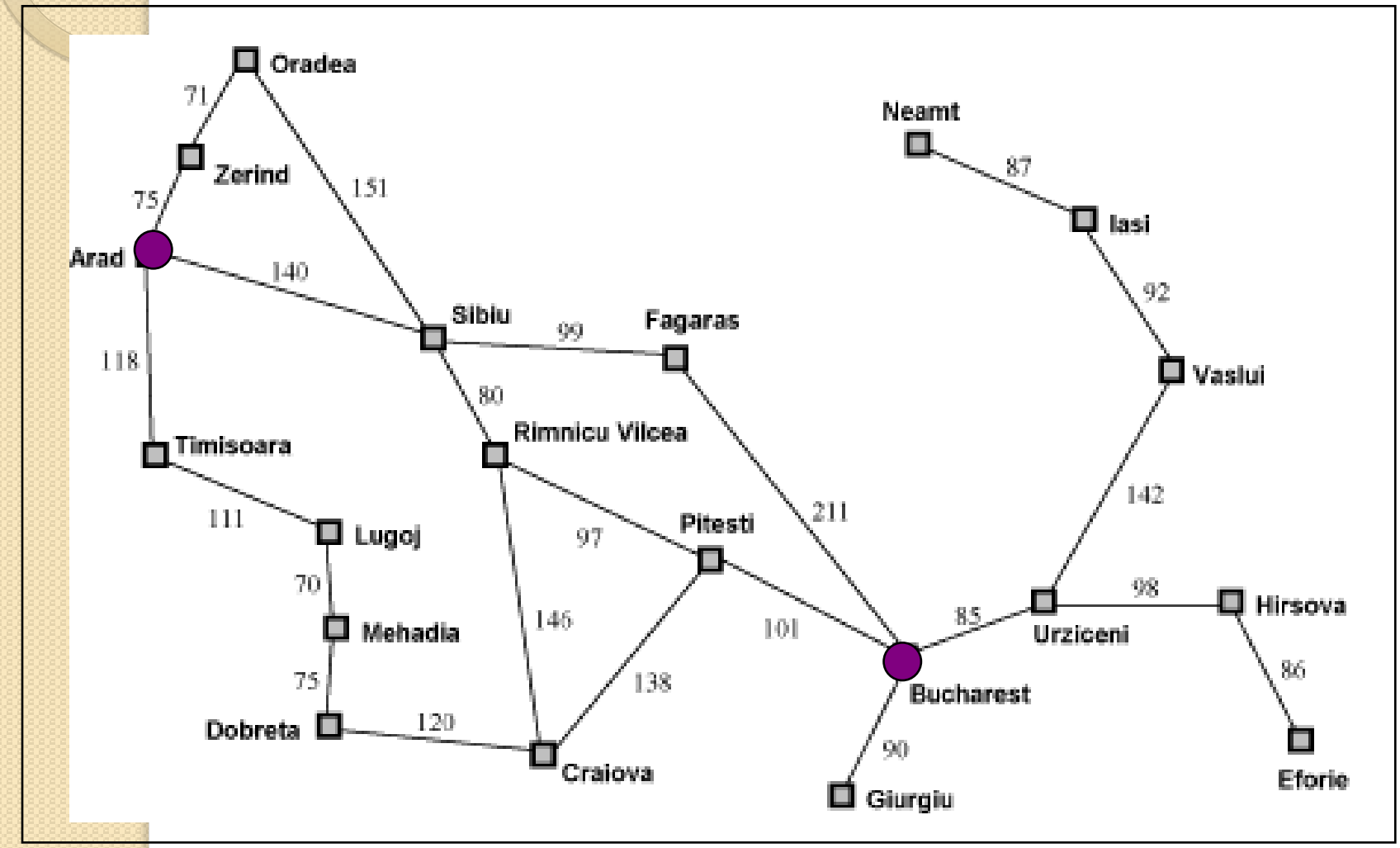
- A* expande o nó de menor valor de *f* na fronteira do espaço de estados
- Tenta minimizar o custo total da solução combinando:
  - Busca Gulosa (h)
    - econômica, porém não é completa nem ótima
  - Busca de Custo Uniforme (g)
    - ineficiente, porém completa e ótima
- *f* - Função de avaliação do A*:
  - *f (n) = g (n) + h (n)*
  - *g (n)* = distância de *n* ao nó inicial
  - *h (n)* = distância estimada de *n* ao nó final

Algoritmo A*

- Se *h* é *admissível*, então *f (n)* é admissível também
  - i.e., *f* nunca irá superestimar o custo real da melhor solução através de *n*
  - pois *g* guarda o valor exato do caminho já percorrido.
- Com A*, a rota escolhida entre *Canudos* e *Petrolândia* é de fato a mais curta, uma vez que:
  - *f (BSF) = 2,5 u + 1,5 u = 4 u*
  - *f (Jeremoabo) = 1,5 u + 2,1 u = 3,6 u*

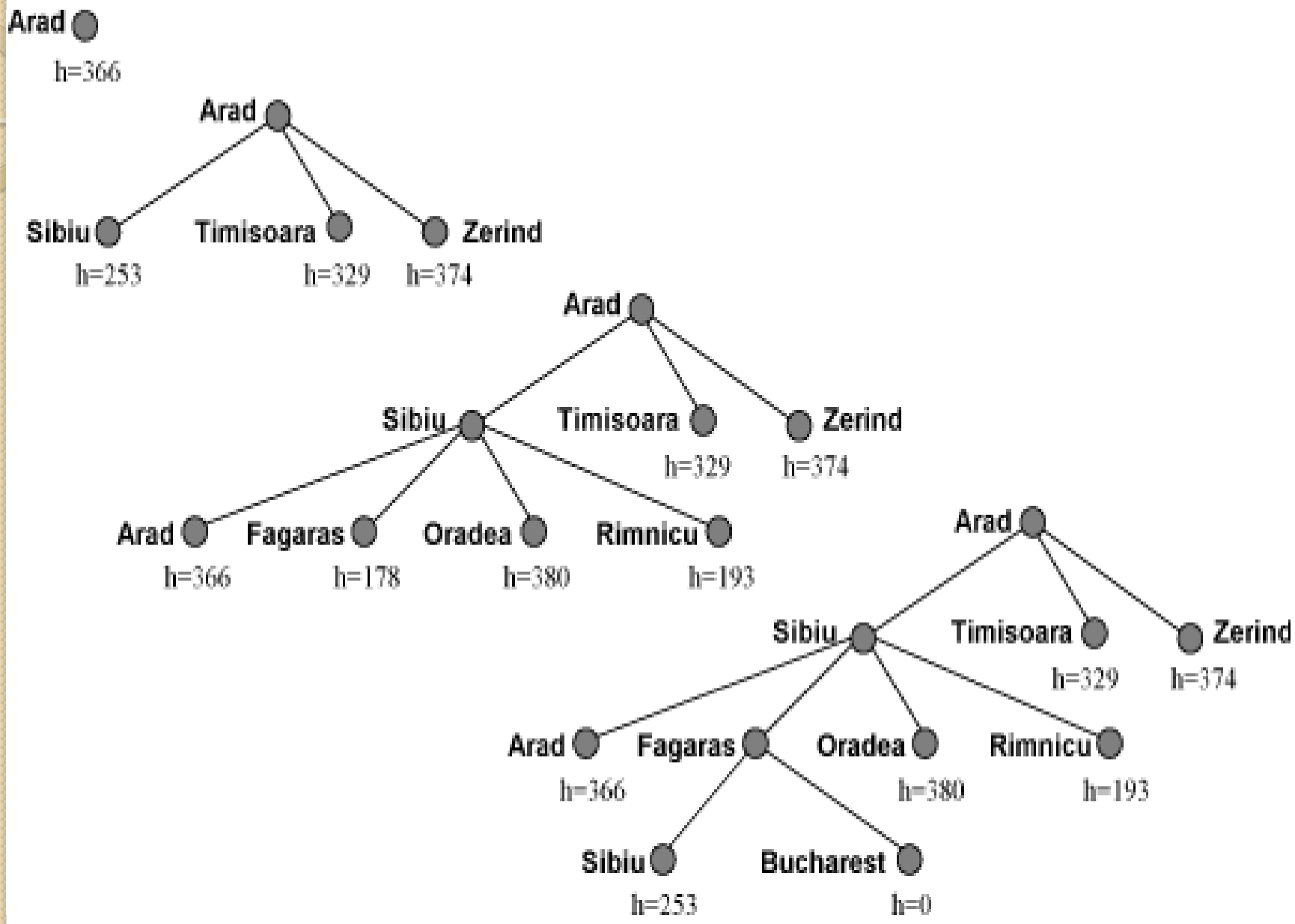
Algoritmo A*: outro exemplo

Viajar de Arad a Bucharest

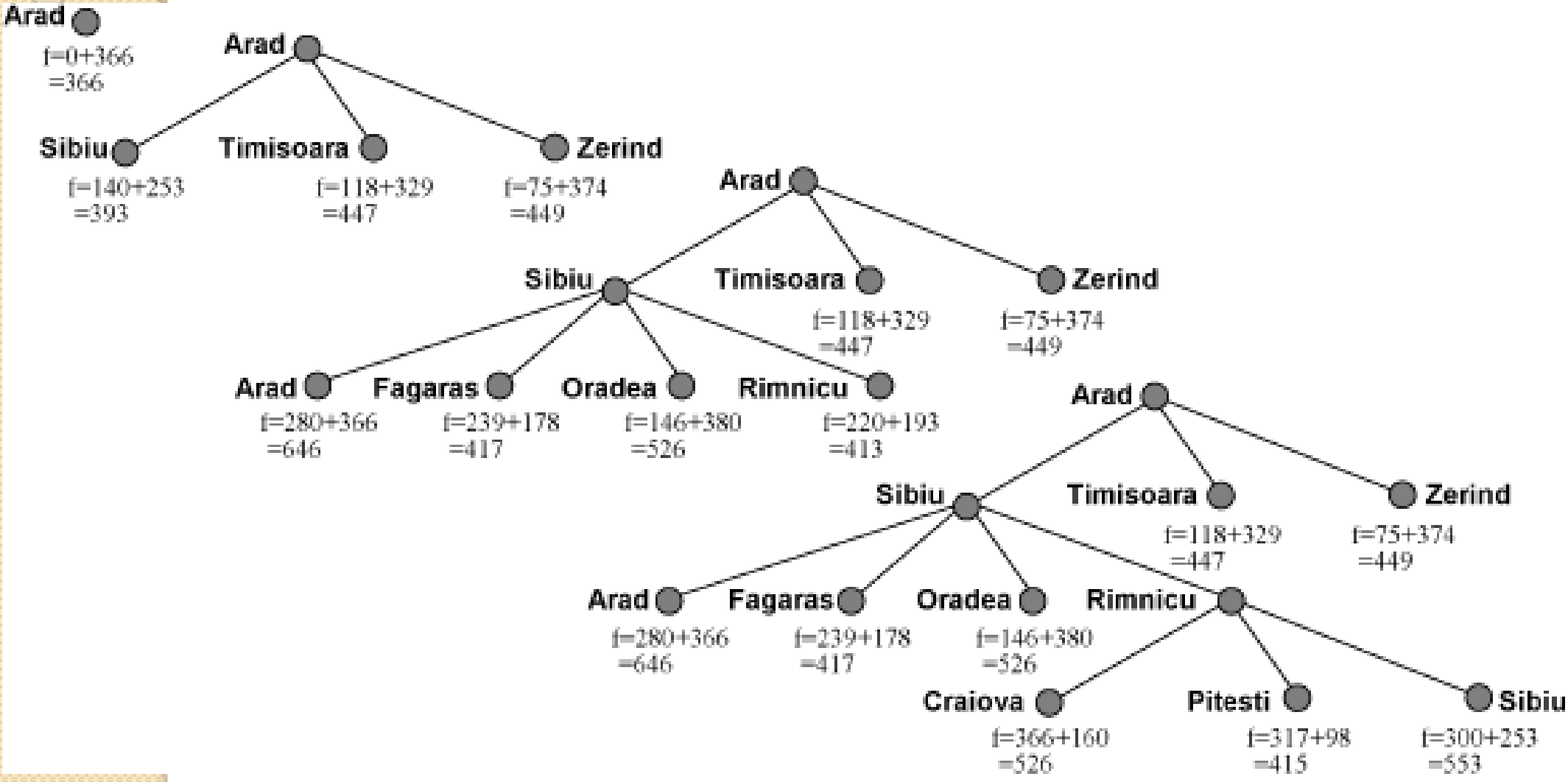


Straight line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Se fosse pela Busca Gulosa...



Usando A*


Arad
f=0+366
=366
Arad
Sibiu
Timisoara
Zerind
f=140+253
=393
f=118+329
=447
f=75+374
=449
Arad
Sibiu
Timisoara
Zerind
f=118+329
=447
f=75+374
=449
Arad
Fagaras
Oradea
Rimnicu
f=280+366
=646
f=239+178
=417
f=146+380
=526
f=220+193
=413
Arad
Sibiu
Timisoara
Zerind
f=118+329
=447
f=75+374
=449
Arad
Fagaras
Oradea
Rimnicu
f=280+366
=646
f=239+178
=417
f=146+380
=526
Craiova
Pitesti
Sibiu
f=366+160
=526
f=317+98
=415
f=300+253
=553

Algoritmo A*:
Análise do comportamento

- A estratégia é completa e ótima
- Custo de tempo:
  - exponencial com o comprimento da solução, porém boas funções heurísticas diminuem significativamente esse custo
    - o fator de expansão fica próximo de 1
- Custo memória: $O(b^d)$
  - guarda todos os nós expandidos na memória, para possibilitar o *backtracking*

Algoritmo A*

Análise do comportamento

- A estratégia apresenta eficiência ótima
  - nenhum outro algoritmo ótimo garante expandir menos nós
- A* só expande nós com $f(n) \leq C^*$, onde C^* é o custo do caminho ótimo
- Para se garantir otimalidade do A*, o valor de f em um caminho particular deve ser não decrescente!!!
  - $f(\text{sucessor}(n)) \geq f(n)$
  - i.e., o custo de cada nó gerado no mesmo caminho nunca é menor do que o custo de seus antecessores

Algoritmo A*

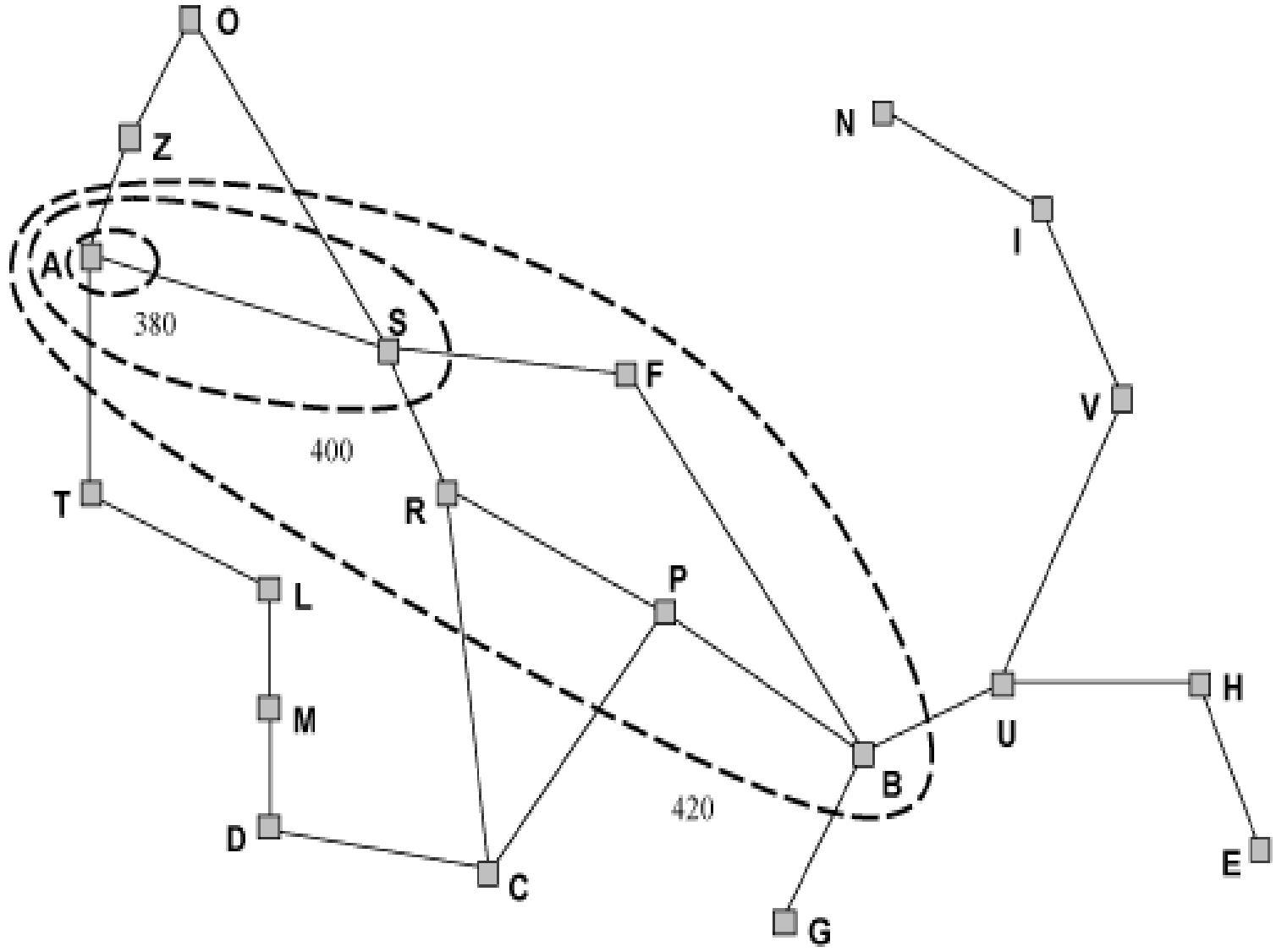
Análise do comportamento

- $f = g + h$ deve ser não decrescente
  - g é não decrescente (para operadores não negativos)
    - custo real do caminho já percorrido
  - h deve ser não-crescente (consistente, monotônica)
    - $h(n) \geq h(\text{sucessor}(n))$
    - i.e., quanto mais próximo do nó final, menor o valor de *h*
    - isso vale para a maioria das funções heurísticas
- Quando *h* não é consistente, para se garantir otimalidade do A*, temos:
  - quando $f(\text{suc}(n)) < f(n)$
  - usa-se $f(\text{suc}(n)) = \max(f(n), g(\text{suc}(n)) + h(\text{suc}(n)))$

A* define Contornos

$f(n) \leq C^*$

fator de expansão próximo de 1

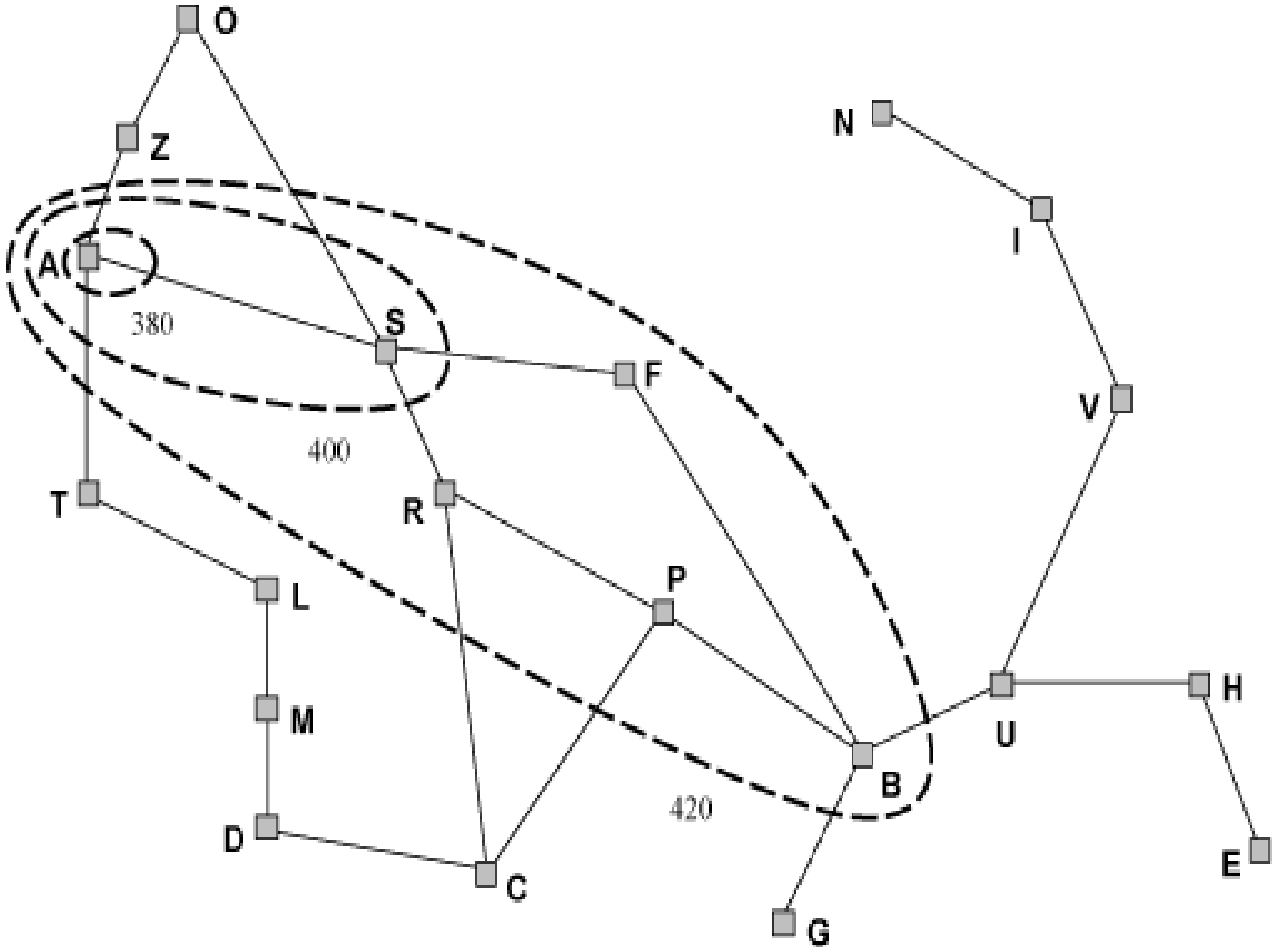


Busca com Limite de Memória
Memory Bounded Search

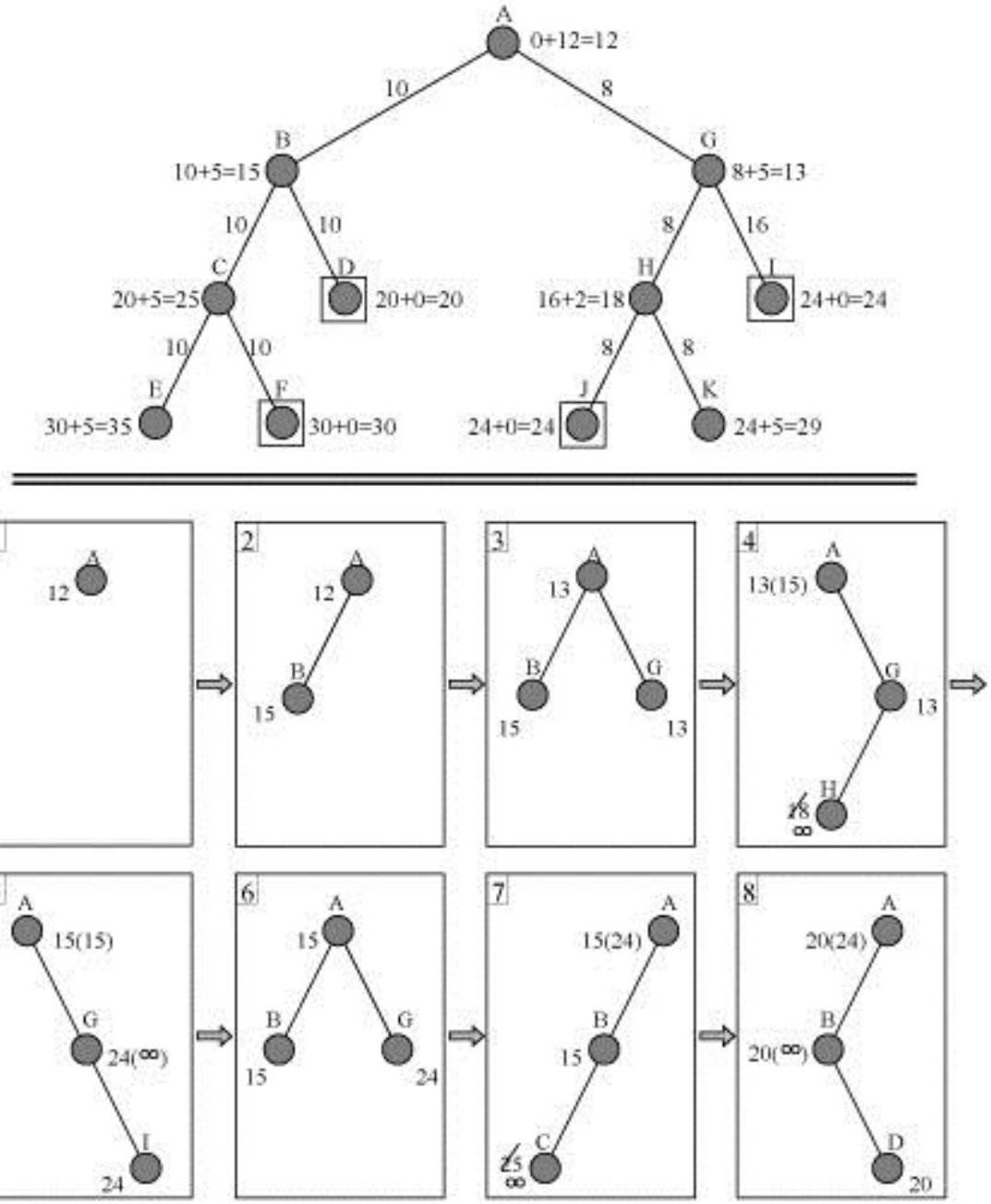
- IDA* (Iterative Deepening A*)
  - igual ao aprofundamento iterativo, porém seu limite é dado pela função de avaliação (f) , e não pela profundidade (d).
  - necessita de menos memória do que A*

- SMA* (Simplified Memory-Bounded A*)
  - O número de nós guardados em memória é fixado previamente

IDA* - Iterative Deepening A*



SMA* - Simplified Memory-Bounded A*



Próxima aula

- Vamos ver como criar nossas próprias funções heurísticas
- Algoritmos de Melhorias Iterativas
- Para não perder o costume... Lembrem de imprimir os slides, certo? ☺
  - www.cin.ufpe.br/~if684/EC/2011-1