Busca cega ou não informada

Universidade Federal de Pelotas
Centro de Desenvolvimento Tecnológico
Fundamentos de Inteligência Artificial
Prof.: Anderson Priebe Ferrugem
E-mail: ferrugem@inf.ufpel.edu.br

Busca cega

- Busca em largura (Breadth-first search)
- Busca de custo uniforme (Uniform cost search)
- Busca em profundidade(Depth-first search)
- Busca em profundidade limitada(Depth-limited search)
- Busca de aprofundamento iterativo (Interative deepening search)

Busca em largura (Breadth-first search)

Busca em largura

1. Crie uma variável chamada Lista-de-Nós e insira o estado inicial.
2. Até ser encontrado um estado-meta ou Lista –de- Nós ficar vazia, faça:

   (a)Remova o primeiro elemento de Lista –de- Nós e chame-o de E. Se Lista –de- Nós estiver vazia, saia.

   (b)Para cada maneira como cada regra pode ser casada com o estado descrito por E, faça:

   Aplique a regra para gerar um novo estado.

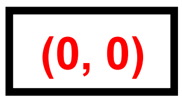
   Se o novo estado for um estado-meta, saia e retorne este estado.

   Caso contrario, acrescente o novo estado ao final de Lista -de- Nós.

Busca em largura

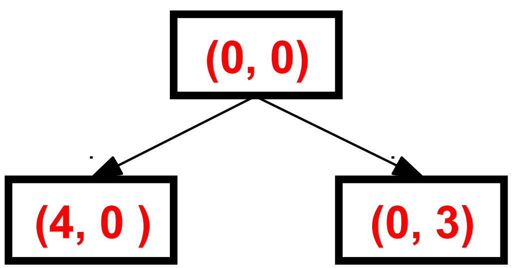
- Lista de nós

▪(0,0)		



- Lista de nós

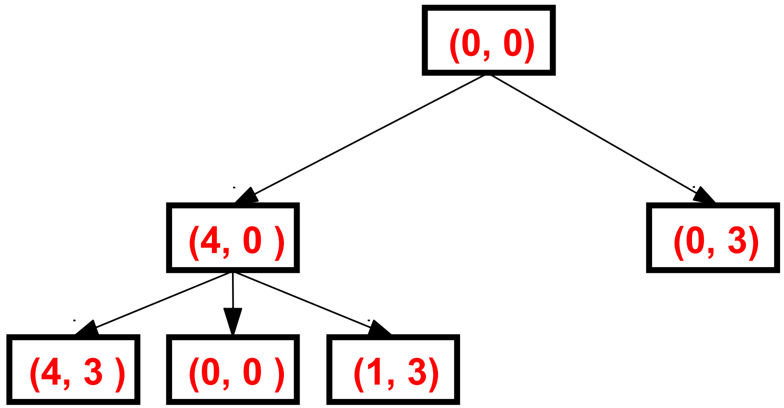
E = (0, 0)



- Lista de nós

(4, 0)	(0, 3)	

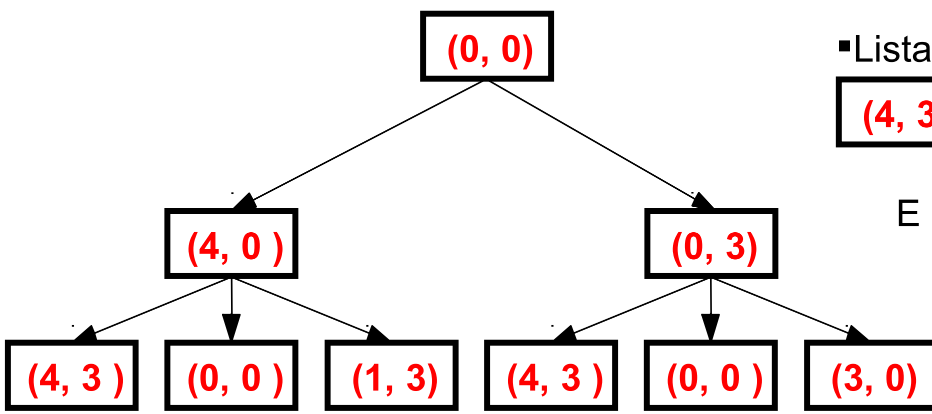
Busca em largura



- Lista de nós

(0, 3)	(4, 3)	(0, 0)	(1, 3)

E = (4, 0)

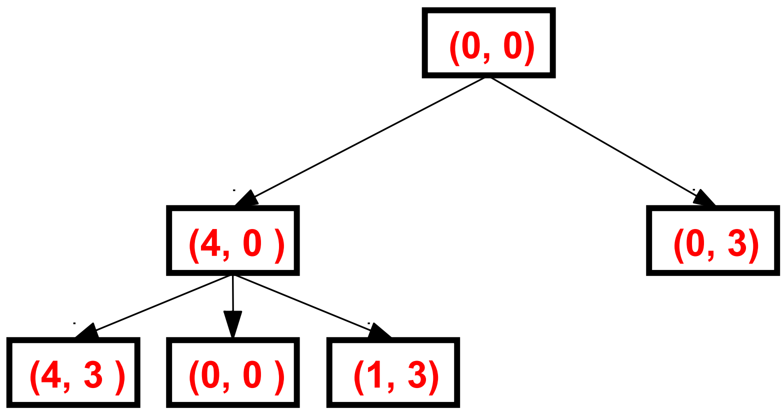


- Lista de nós

(4, 3)	(0, 0)	(1, 3)

E = (3, 0)

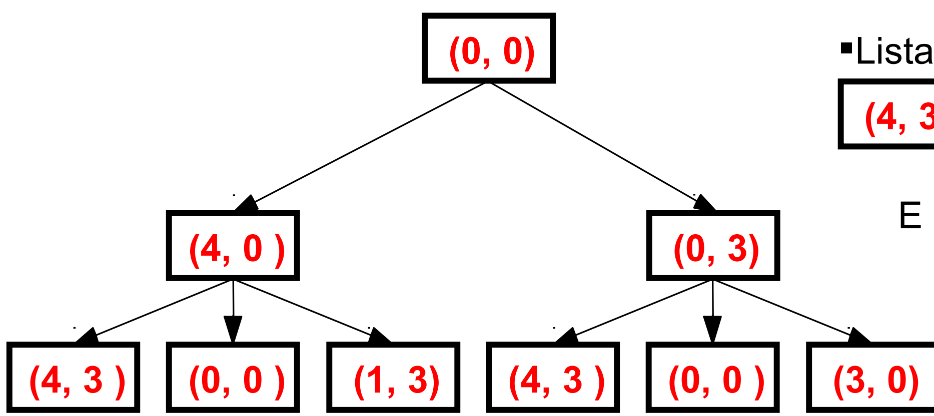
Busca em largura



- Lista de nós

(0, 3)	(4, 3)	(0, 0)	(1, 3)

E = (4, 0)



- Lista de nós

(4, 3)	(0, 0)	(1, 3)

E = (3, 0)

Busca em largura

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

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 = $O(b^d)$.

Busca em largura

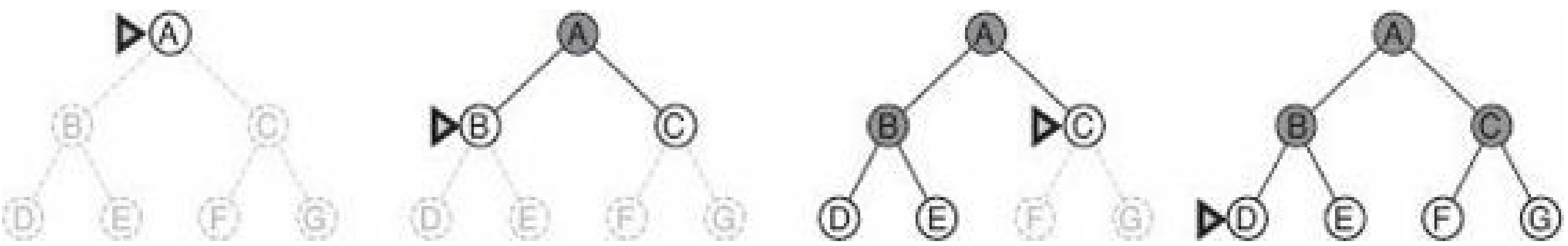
Custo de memória:

- a *fronteira* do espaço de estados deve permanecer na memória é o problema mais crucial do que o tempo de execução da busca

Esta estratégia só dá bons resultados quando a *profundidade* da árvore de busca é *pequena*.

Busca em largura

função BUSCA-EM-LARGURA(*problema*) **retorna** uma solução ou falha
    nó ← um nó com ESTADO = *problema*.ESTADO-INICIAL, CUSTO-DE-CAMINHO = 0
    se *problema*.TESTE-DE-OBJETIVO(nó.ESTADO) **senão retorne** SOLUÇÃO(*nó*),
    borda ← uma fila FIFO com *nó* como elemento único
    explorado ← conjunto vazio
    repita
        se VAZIO?(*borda*), **então retorne** falha
        nó ← POP(*borda*) / * escolhe o nó mais raso na *borda* */
        adicione *nó*.ESTADO para *explorado*
        para cada *ação* **em** *problema*.AÇÕES(nó.ESTADO) **faça**
            filho ← NÓ-FILHO(*problema, nó, ação*),
            se (filho.ESTADO)não está em *explorado* ou *borda* **então**
                se *problema*.TESTE-DE-OBJETIVO(filho.ESTADO) **então retorne** SOLUÇÃO(*filho*)
                borda ← INSIRA(*filho, borda*)



Busca de custo uniforme (Uniform cost search)

Busca de custo uniforme

Se todos os custos de passos forem iguais, a busca em largura será ótima:

- Sempre expande o nó mais raso não expandido.

Através de uma simples extensão, podemos encontrar um algoritmo que é ótimo para qualquer função de custo do passo.

Busca de custo uniforme

Em vez de expandir o nó mais raso, a busca de custo uniforme expande o nó n com o custo de caminho g(n) mais baixo. Isso é feito através do armazenamento da borda como uma fila de prioridade ordenada por g.

Busca de custo uniforme

```
função BUSCA-DE-CUSTO-UNIFORME(problema) retorna uma solução ou falha
  nó ← um nó com ESTADO = problema.ESTADO-INICIAL, CUSTO-DE-CAMINHO = 0
  borda ← fila de prioridade ordenada pelo CUSTO-DE-CAMINHO, com nó como elemento único
  explorado ← um conjunto vazio
  repita
    se VAZIO?(borda), então retornar falha
    nó ← POP(borda) / * escolhe o nó de menor custo na borda */
    se problema.TESTE-OBJETIVO(nó.ESTADO) então retornar SOLUÇÃO(nó)
    adicionar (nó.ESTADO) para explorado
    para cada ação em problema. AÇÕES(nó.ESTADO) faça
      filho ← NÓ-FILHO (problema, nó, ação)
      se (filho.ESTADO) não está na borda ou explorado então
        borda ← INSIRA (filho, borda)
      senão se (filho.ESTADO) está na borda com o maior CUSTO-DE-CAMINHO então
        substituir aquele nó borda por filho
```

How to do Uniform Cost Search

Algorithm

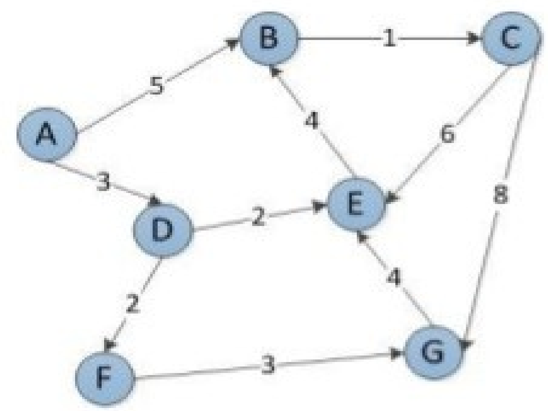
if

- Frontier : empty >Return fail

else

- Add node to frontier.
- Check: node (goal)>solution
- Add node to explored.
- Neighbor s: if not explored >add to frontier
- Else :if was with higher cost replace it .

Example



Solution
Explored : A D B E F C
path: A to D to F to G
Cost = 8

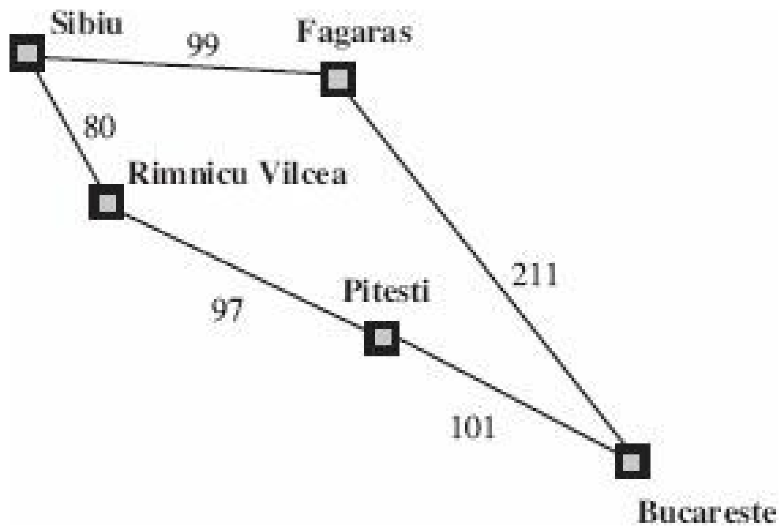
Busca de custo uniforme

Usa uma fila de prioridade

Possui uma verificação extra, caso um caminho mais curto para um estado de borda seja descoberto.

A estrutura de dados para a borda deve permitir os testes eficientes de pertinência em conjunto, por isso deve combinar os recursos de uma fila de prioridade e de uma tabela hash.

Busca de custo uniforme



Sibiu
–- Rimnicu Vilcea (80)
–- Fagaras (99)
Rimnicu Vilcea (Menor custo)
–- Pitesti (80 + 97 = 177).

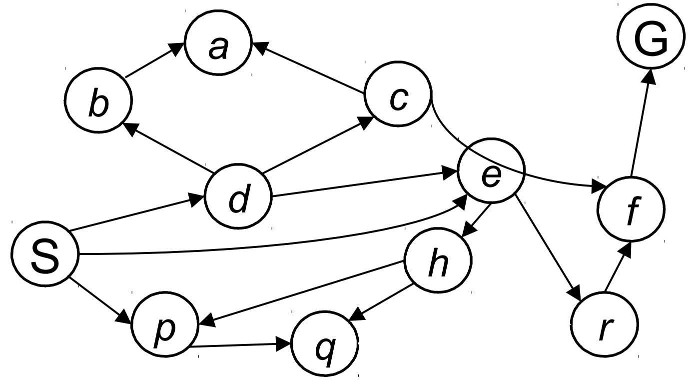
Fagaras (Menor custo)
–- Bucareste (99 + 211 = 310). META !!

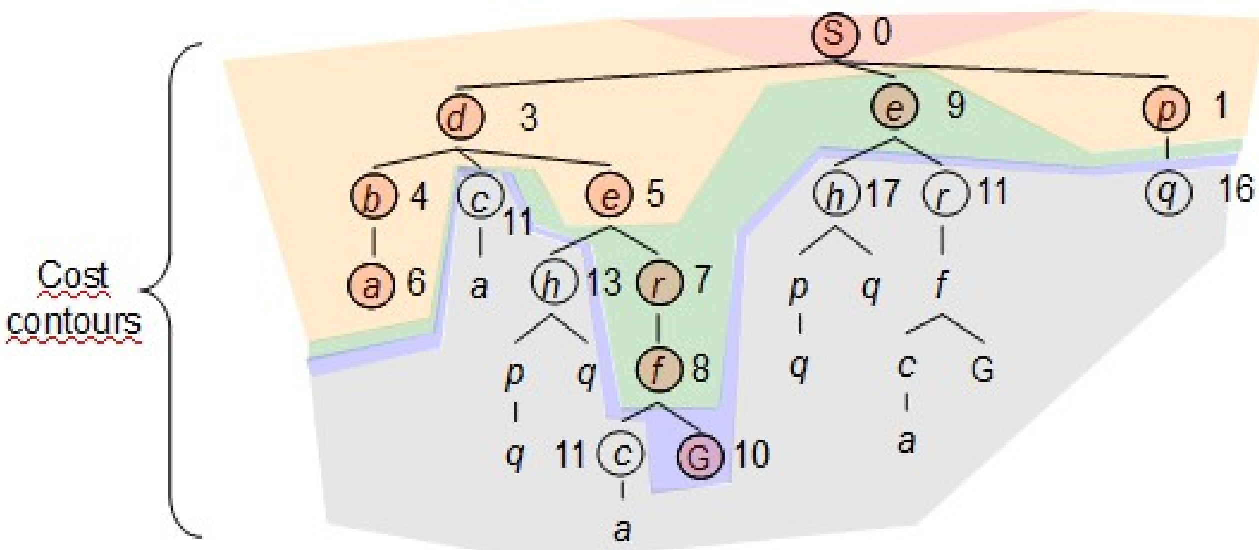
Mas a busca de custo uniforme continua
Pitesti
– Bucareste (80 + 97 + 101 = 278).

O algoritmo verifica se esse novo caminho é melhor do que o antigo, isto é, de modo que o antigo seja descartado.

Bucareste, agora com o custo g = 278, será selecionada para a expansão e a solução será devolvida.

Busca de custo uniforme





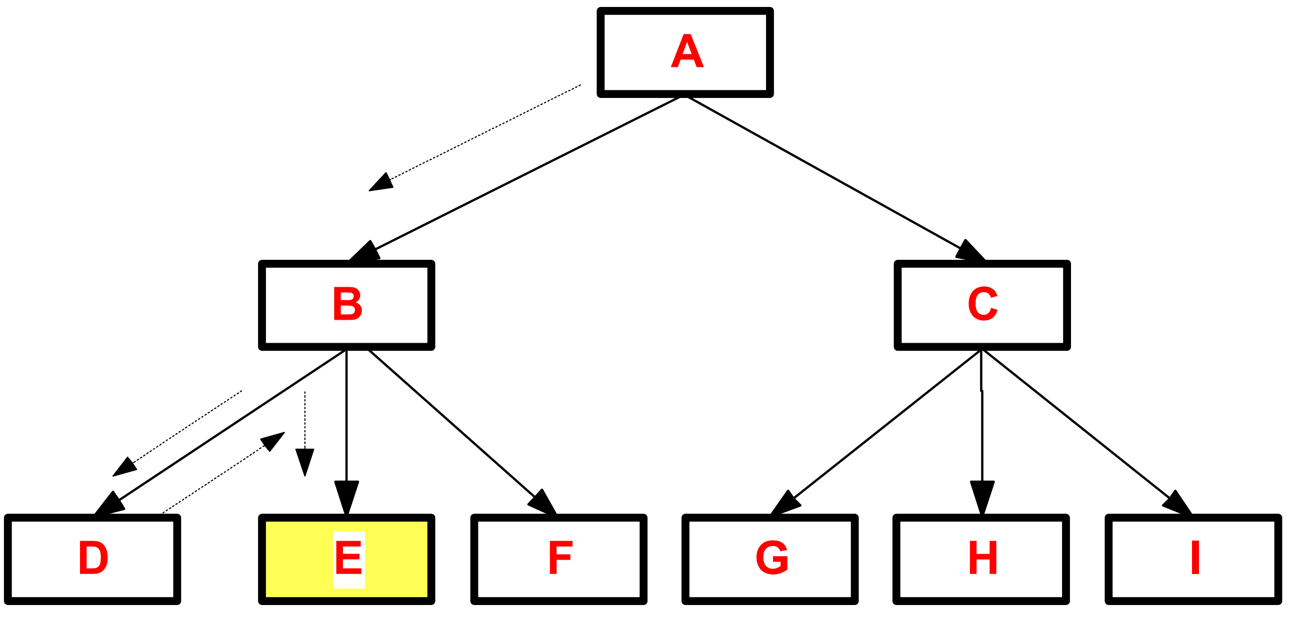
Busca em profundidade

Crie uma variável chamada Pilha-de-Nós e ajuste-a para o estado inicial.

Até ser encontrado um estado-meta (sucesso) ou Pilha – de-Nós ficar vazia, faça:

- Faça Estado atual receber o primeiro elemento de Pilha–de-Nós.
- Se Pilha–de- Nós estiver vazia, saia e sinalize fracasso.
- Gere um sucessor, E, do estado atual. Se não houver mais sucessores, sinalize fracasso e remova estado atual da pilha –de – nós, se E for um estado meta retorne sucesso.
- Se existir um sucessor E do estado – atual , insira E na pilha.

Busca em profundidade



Estado Meta : E

Busca em profundidade

Esta estratégia não é *completa* nem é *ótima*.

Custo de memória:

mantém na memória o caminho que está 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.

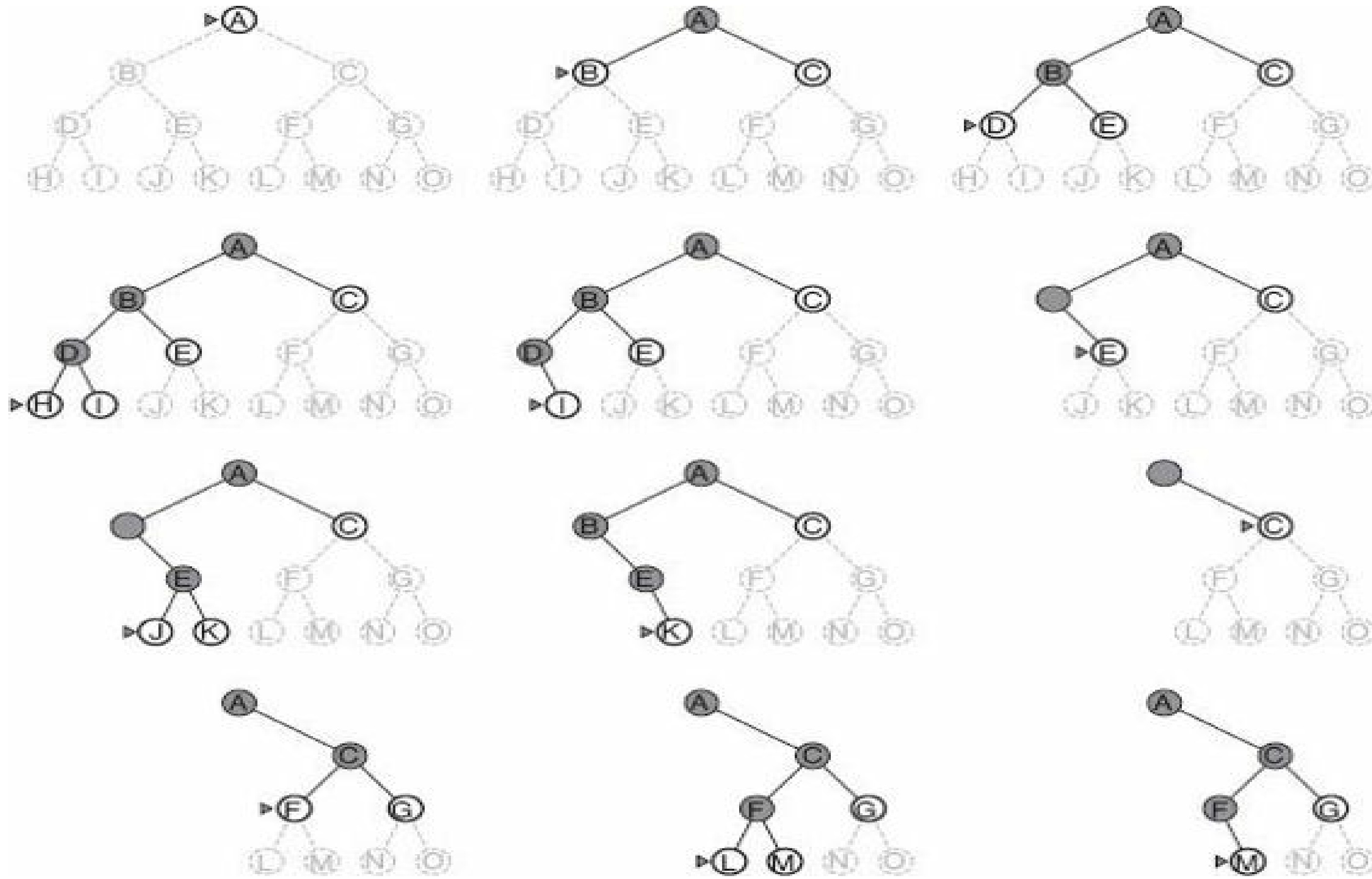
Para problemas com várias soluções, esta estratégia pode ser bem mais rápida do que busca em largura.

Esta estratégia deve ser evitada quando as árvores geradas são muito *profundas* ou geram *caminhos infinitos*.

Busca em profundidade

```
função BUSCA-EM-PROFUNDIDADE-LIMITADA(problema, limite) retorna uma solução ou falha/corte
retornar BPL-RECURSIVA (CRIAR-NÓ(problema, ESTADO-INICIAL), problema, limite)
função BPL-RECURSIVA(nó, problema, limite) retorna uma solução ou falha/corte
  se problema. TESTAR-OBJETIVO (nó.ESTADO) então, retorna SOLUÇÃO (nó)
  se não se limite = 0 então retorna corte
  senão
    corte_ocorreu? ← falso para cada ação no problema.AÇÕES(nó.ESTADO) faça
      filho ← NÓ-FILHO (problema, nó, ação)
      resultado ← BPL-RECURSIVA (criança, problema limite − 1)
      se resultado = corte  então corte_ocorreu? ← verdadeiro
      senão se resultado ≠ falha então retorna resultado
    se corte_ocorreu? então retorna corte senão retorna falha
```

Busca em profundidade



A região inexplorada é mostrada em cinza-claro. Os nós explorados sem descendentes na borda são removidos da memória. Os nós na profundidade 3 não têm sucessores e M é o único nó objetivo

Busca em profundidade limitada

Limita o nível de profundidade

função BUSCA-EM-PROFUNDIDADE-LIMITADA(problema, limite) **retorna** uma solução ou falha/corte
retornar BPL-RECURSIVA (CRIAR-NÓ(*problema,* ESTADO-INICIAL), *problema, limite*)
função BPL-RECURSIVA(*nó, problema, limite*) **retorna** uma solução ou falha/corte
  se *problema*. TESTAR-OBJETIVO (nó.ESTADO) então, **retorna** SOLUÇÃO (nó)
  se não se *limite* = 0 **então retorna** *corte*
  senão
    corte_ocorreu? ← falso **para cada** ação **no** problema.AÇÕES(nó.ESTADO) **faça**
      filho ← NÓ-FILHO (*problema, nó, ação*)
      resultado ← BPL-RECURSIVA (*criança, problema limite − 1*)
      se *resultado* = *corte* **então** *corte_ocorreu*? ← verdadeiro
      senão se *resultado* ≠ *falha* **então retorna** *resultado*
    se *corte_ocorreu*? **então retorna** *corte* **senão retorna** *falha*

Busca de aprofundamento iterativo (IDS – Iterative Deepening Search)

Usada com frequência em combinação com a busca em profundidade em árvore, que encontra o melhor limite de profundidade.

Aumenta gradualmente o limite — primeiro 0, depois 1, depois 2, e assim por diante até encontrar um objetivo.

Isso ocorrerá quando o limite de profundidade alcançar d, a profundidade do nó objetivo mais raso.

O aprofundamento iterativo combina os benefícios da busca em profundidade e da busca em largura. Como na busca em profundidade, seus requisitos de memória são muito modestos: O(bd), para sermos precisos. Como na busca em largura, ele é completo quando o fator de ramificação é finito, e ótimo quando o custo de caminho é uma função não decrescente da profundidade do nó.

Busca de aprofundamento iterativo (IDS – Iterative Deepening Search)

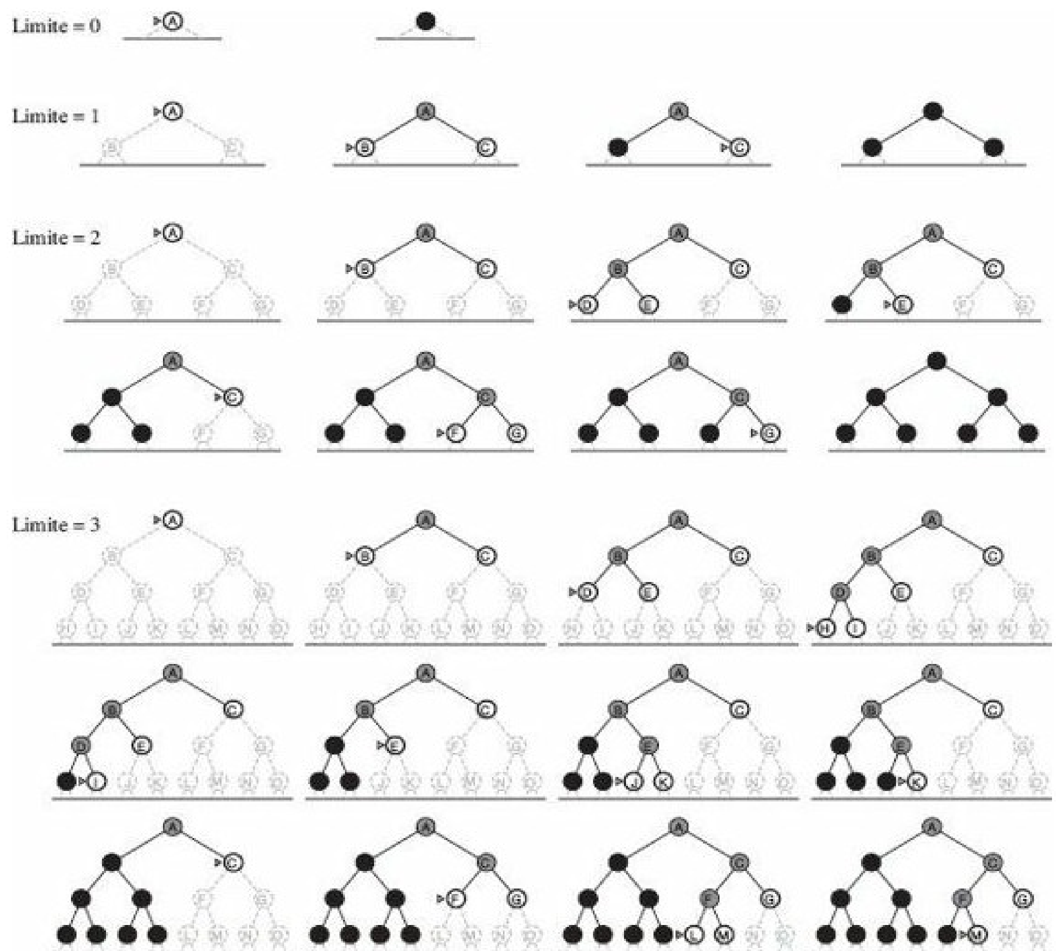
A ideia é recalcular os elementos da fronteira em vez de armazená-los

Cada recálculo pode ser uma busca em profundidade, que utiliza, portanto, menos espaço.

Busca de aprofundamento iterativo (IDS – Iterative Deepening Search)

função BUSCA-DE-APROFUNDAMENTO-ITERATIVO(*problema*) **retorna** uma solução ou falha
para profundidade = 0 **até** ∞ **faça**
resultado ← BUSCA-EM-PROFUNDIDADE-LIMITADA(*problema*, *profundidade*)
se *resultado* ≠ *corte* **então retornar** *resultado*

Busca de aprofundamento iterativo (IDS – Iterative Deepening Search)



9 Quatro iterações de busca de aprofundamento iterativo em uma árvore binária.

Exercícios

Considere um algoritmo para construir roteiro de vôos entre cidades com ou sem conexão direta, alimentado com os seguintes dados:

```
New York -> Chicago          = 1000 milhas
Chicago  -> Denver           = 1000 milhas
New York -> Toronto          = 800  milhas
New York -> Denver           = 1900 milhas
Toronto  -> Calgary          = 1500 milhas
Toronto  -> Los Angeles      = 1800 milhas
Toronto  -> Chicago          = 500  milhas
Denver   -> Urbana           = 1000 milhas
Denver   -> Houston          = 1500 milhas
Houston  -> Los Angeles      = 1500 milhas
Denver   -> Los Angeles      = 1000 milhas
```

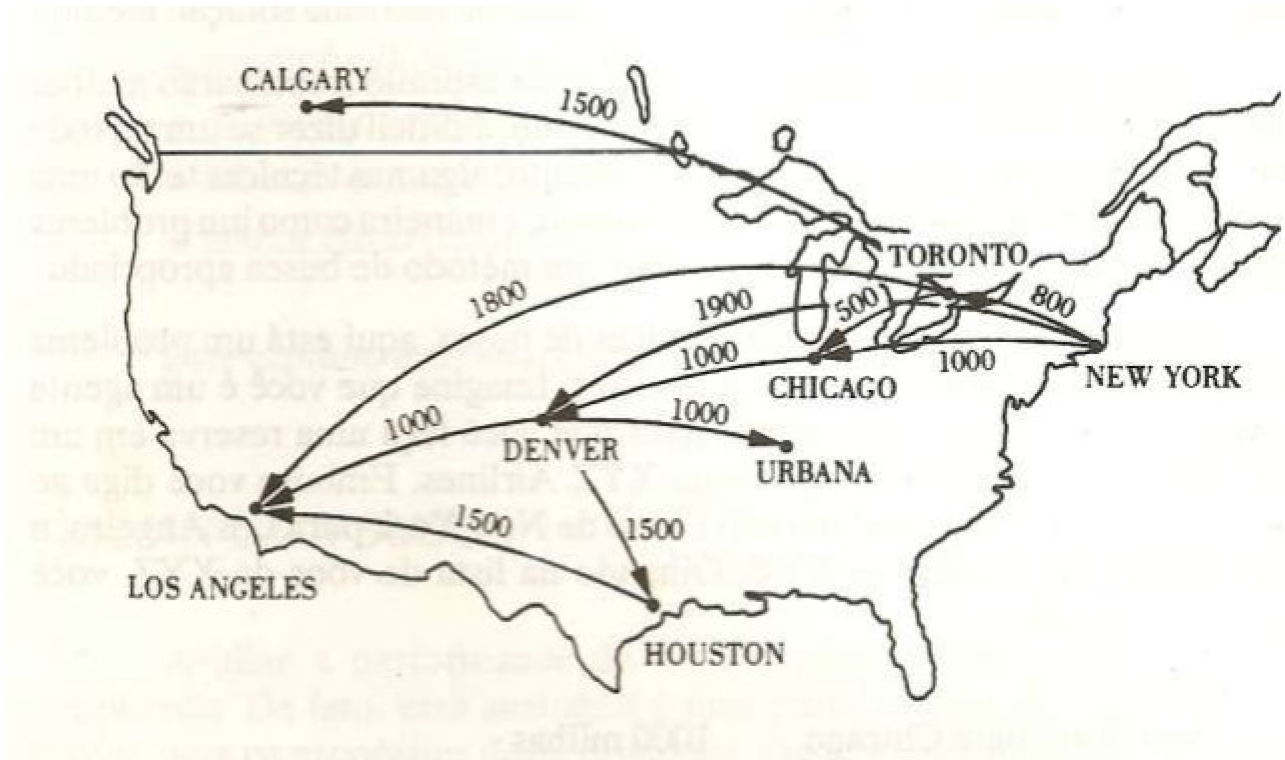
Exercícios

Construa o grafo e árvore de busca (partindo de New York)

```
New York -> Chicago          = 1000 milhas
Chicago  -> Denver           = 1000 milhas
New York -> Toronto          = 800  milhas
New York -> Denver           = 1900 milhas
Toronto  -> Calgary          = 1500 milhas
Toronto  -> Los Angeles      = 1800 milhas
Toronto  -> Chicago          = 500  milhas
Denver   -> Urbana           = 1000 milhas
Denver   -> Houston          = 1500 milhas
Houston  -> Los Angeles      = 1500 milhas
Denver   -> Los Angeles      = 1000 milhas
```

Exercícios

Construa o grafo e árvore de busca (partindo de New York)

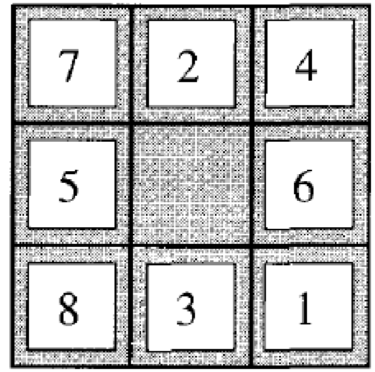


Exercícios

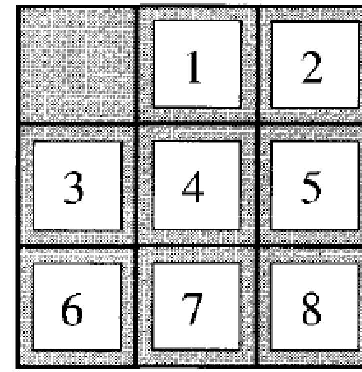
Implemente um algoritmo de busca em profundidade e largura para solucionar o quebra-cabeça oito.

Critérios

- Grupos de no máximo 3 pessoas;
- Implementação linguagem selecionada;
- Deve retornar
  - o número de estados (movimentos) testados
  - o caminho e número de estados para a solução;



Estado Inicial



Estado Meta

Exemplo de possível solução;

Característica interessante deste problema:

O espaço de estados é dividido em dois subespaços de maneira que, a partir de um estado em um subespaço é impossível alcançar qualquer estado do outro subespaço