CEFET/RJ

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

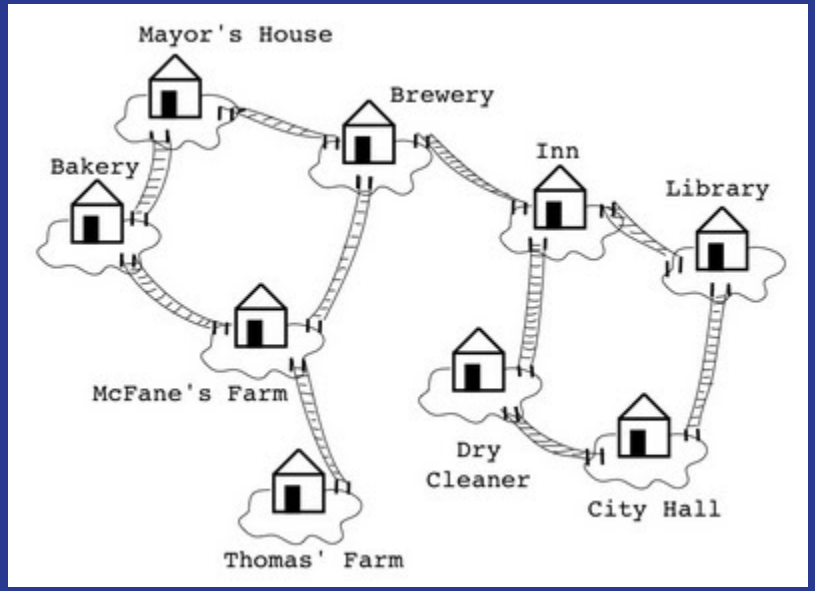
GCC1734 - INTELIGÊNCIA ARTIFICIAL

Prof. Eduardo Bezerra
ebezerra@cefet-rj.br

Créditos

- Essa apresentação é uma tradução e/ou adaptação feita pelo prof. Eduardo Bezerra (ebezerra@cefet-rj.br) do material cuja autoria é dos professores Dan Klein e Pieter Abbeel (UC Berkeley).
- O material original é usado no curso CS188 (Introduction to Artificial Intelligence).
  - https://inst.eecs.berkeley.edu/~cs188

BUSCA SEM INFORMAÇÃO



O que estudamos até aqui

- Agentes
  - Tipos de agentes
  - Agentes planejadores
- Problemas de busca
- Conceito de estratégia de busca

Implementação: SearchProblem

- Um problema de busca é formulado pela definição de cinco componentes:
  1. Conjunto S (espaço de estados), com um estado inicial,
  2. Função ACTIONS(*s*): produz as ações possíveis em cada estado,
  3. Função RESULT(*s, a*): modelo de transição ou função sucessora, produz o estado resultante de selecionar a ação *a* no estado *s*.
  4. Função teste de objetivo, que permite ao agente determinar se seu objetivo foi alcançado.
  5. Função custo de caminho (função aditiva e cumulativa), que permite ao agente comparar planos alternativos.

Implementação: SearchProblem

- Um problema de busca é formulado pela definição de cinco componentes:
  1. Conjunto S (espaço de estados), com um estado inicial,
  2. Função ACTIONS(*s*): produz as ações possíveis em cada estado,
  3. Função RESULT(*s, a*): modelo de transição ou função sucessora, produz o estado resultante de selecionar a ação *a* no estado *s*.
  4. Função teste de objetivo, que permite ao agente determinar se seu objetivo foi alcançado.
  5. Função custo de caminho (função aditiva e cumulativa), que permite ao agente comparar planos alternativos.

```
class SearchProblem:
    """

    def getStartState(self):

    def isGoalState(self, state):

    def expand(self, state):

    def getActions(self, state):

    def getActionCost(self, state, action, next_state):

    def getNextState(self, state, action):

    def getCostOfActionSequence(self, actions):
```

search.py

Implementação: SearchProblem

- Um problema de busca é formulado pela definição de cinco componentes:
  1. Conjunto S (espaço de estados), com um estado inicial,
  2. Função ACTIONS(*s*): produz as ações possíveis em cada estado,
  3. Função RESULT(*s, a*): modelo de transição ou função sucessora, produz o estado resultante de selecionar a ação *a* no estado *s*.
  4. Função teste de objetivo, que permite ao agente determinar se seu objetivo foi alcançado.
  5. Função custo de caminho (função aditiva e cumulativa), que permite ao agente comparar planos alternativos.

```
class SearchProblem:
    """
    def getStartState(self):                                    1
    def isGoalState(self, state):                               4
    def expand(self, state):
    def getActions(self, state):                                2
    def getActionCost(self, state, action, next_state):
    def getNextState(self, state, action):                      3
    def getCostOfActionSequence(self, actions):                 5
```

Implementação: getChildNode

AIMA, pp 79

function CHILD-NODE(*problem*, *parent*, *action*) **returns** a node
  return a node with
    STATE = *problem*.RESULT(*parent*.STATE, *action*),
    PARENT = *parent*, ACTION = *action*,
    PATH-COST = *parent*.PATH-COST + *problem*.STEP-COST(*parent*.STATE, *action*)

```
def getChildNode(sucessor, parent_node):
    child_node = {'STATE': sucessor[0],
                  'PARENT': parent_node,
                  'ACTION': sucessor[1],
                  'PATH-COST': parent_node['PATH-COST'] + sucessor[2]}
    return child_node
```

Implementação: getChildNode

AIMA, pp 79

function CHILD-NODE(*problem*, *parent*, *action*) **returns** a node
  return a node with
    STATE = *problem*.RESULT(*parent*.STATE, *action*),
    PARENT = *parent*, ACTION = *action*,
    PATH-COST = *parent*.PATH-COST + *problem*.STEP-COST(*parent*.STATE, *action*)

```
def getChildNode(sucessor, parent_node):
    child_node = {'STATE': sucessor[0],
                  'PARENT': parent_node,
                  'ACTION': sucessor[1],
                  'PATH-COST': parent_node['PATH-COST'] + sucessor[2]}
    return child_node
```

Próximo estado

Ação selecionada

Custo da ação

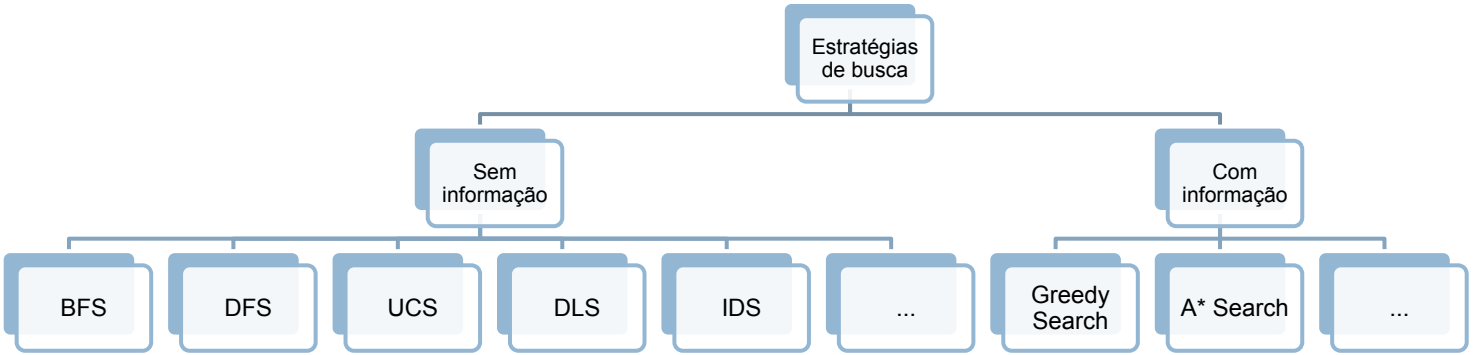
Implementação: getStartNode

```
def getStartNode(problem):
    node = {'STATE': problem.getStartState(), 'PATH-COST': 0}
    return node
```

Implementação: getActionSequence

```
def getActionSequence(node):
    actions = []
    while node['PATH-COST'] > 0:
        actions.insert(0,node['ACTION'])
        node = node['PARENT']
    return actions
```

Estratégias de busca



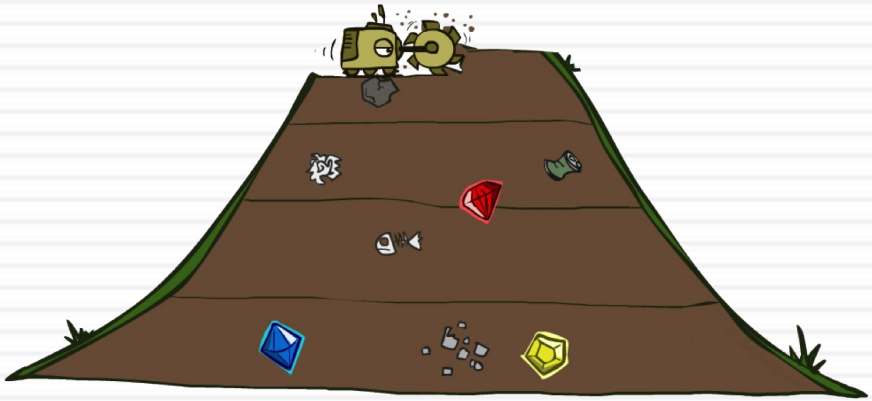
Estratégias de Busca sem Informação

- Estratégias de busca sem informação (ou busca cega - *blind search*) usam apenas a informação disponível na formulação do problema de busca para escolher o plano.
  - Estado: em que situação o agente encontra
  - Função sucessora: o método para ir para o próximo estado
  - Teste de objetivo: um teste para verificar se o objetivo foi atingido
  - Custo do caminho: a penalidade do caminho
- Geram estados sucessores e verificam se o estado objetivo foi atingido.

Estratégias de Busca sem Informação

- Estratégias de busca sem informação se distinguem pela ordem em que os nós são selecionados na borda para ser expandidos.
  1. Busca em Largura (*breadth-first search, BFS*)
  2. Busca com Custo Uniforme (*uniform-cost search, UCS*)
  3. Busca em Profundidade (*depth-first search, DFS*)
  4. Busca em Profundidade Limitada (*depth-limited search, DLS*)
  5. Busca em Profundidade Iterativa (*iterative deepening search, IDS*)

Busca em Largura (*Breadth-First Search* - BFS)



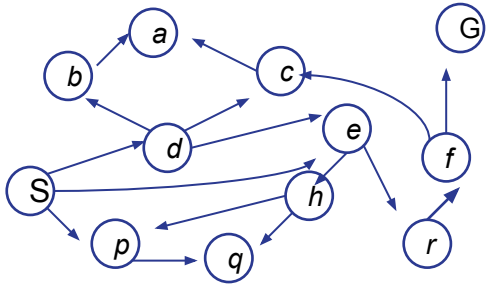
BFS

- Estratégia: expandir a árvore de busca usando o nó que esteja mais perto da raiz.
- “BFS: do all the shallow things before we go the the deep things.”
- Todos nós na profundidade **d** da árvore devem ser expandidos e visitados antes dos nós na profundidade **d+1**.
- Implementação:
  - a *borda* é uma fila FIFO (first-in, first-out), isto é, novos itens entram no final.

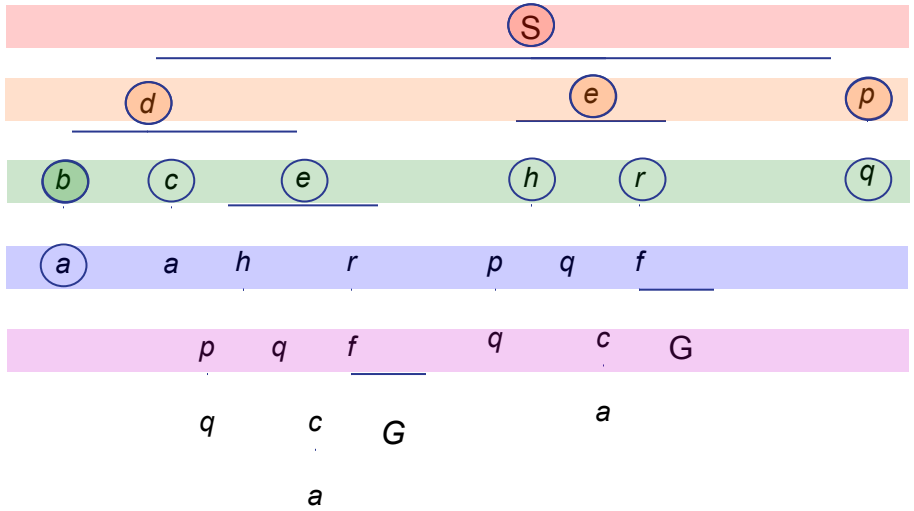
BFS: exemplo

Estratégia: expandir primeiro o nó mais próximo da raiz

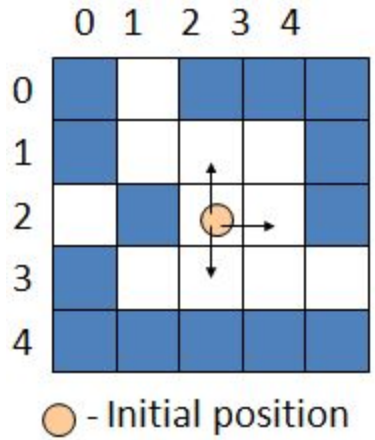
Implementação: borda é uma fila FIFO

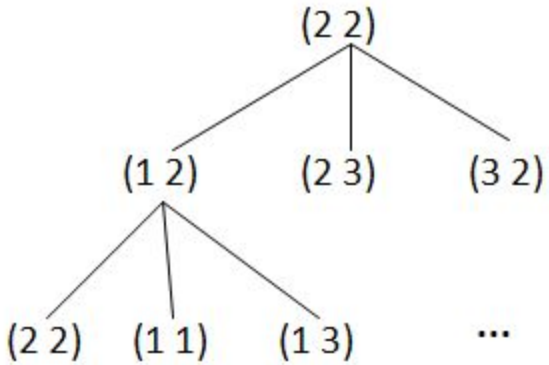


Camadas da busca



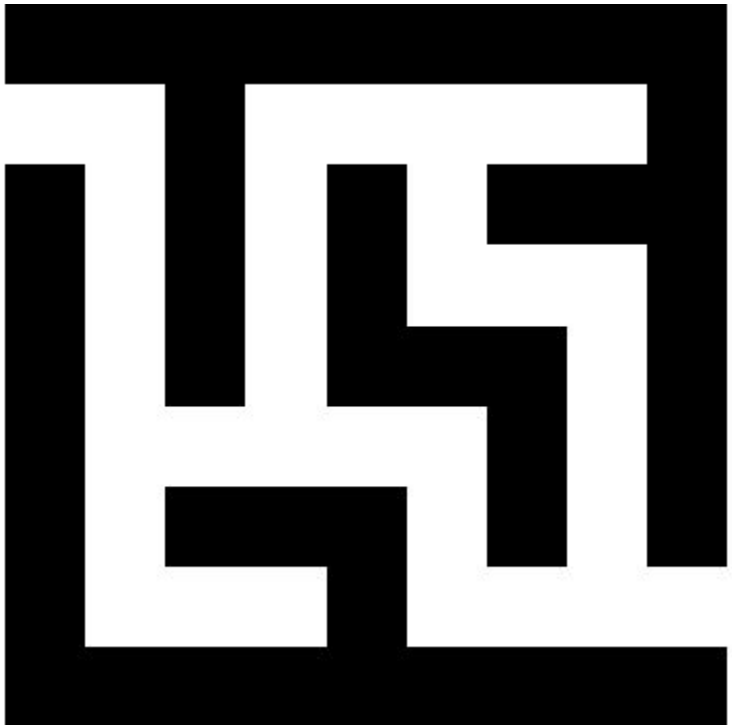
Exemplo: BFS versus *maze solving*





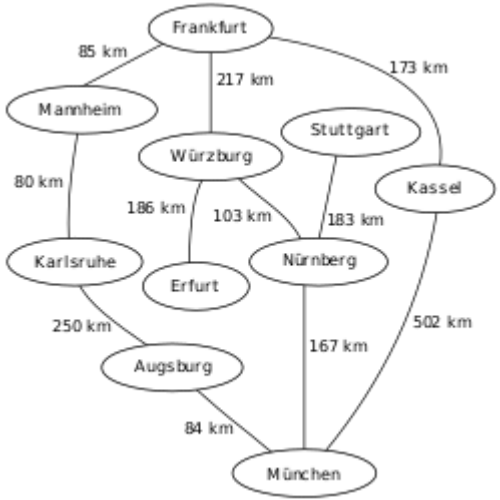
Fonte: https://www.cs.iusb.edu/~danav/teach/c463/4_blind_search.html

Exemplo: BFS versus *maze solving*

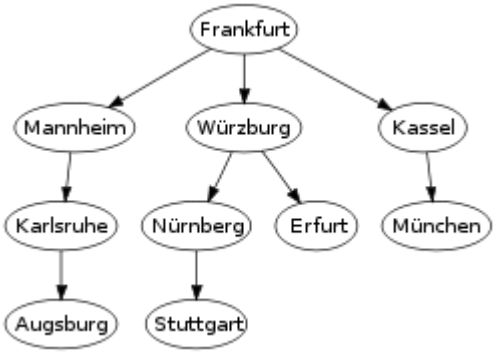


Fonte: https://en.wikipedia.org/wiki/Breadth-first_search#/media/File:BFS-Algorithm_Search_Way.gif

Exemplo: cidades alemãs



Grafo de espaço de estados



Árvore de busca

Fonte: https://en.wikipedia.org/wiki/Breadth-first_search

BFS: propriedades

- Completa?
  - Sim (se *b* é finito)
- Complexidade de tempo?
  - $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$
- Complexidade de espaço?
  - $O(b^d)$ (mantém todos os nós na memória até encontrar o objetivo; complexidade é dominada pelo tamanho da borda.)
- Ótima?
  - Sim (se todas as ações tiverem o mesmo custo, i.e., se o custo do caminho for uma função **não-decrescente** da profundidade do nó.)

BFS: requisitos de tempo e memória

- Suposições
  - Problema de busca com fator de ramificação b=10;
  - 1.000.000 nós podem ser gerados por segundo;
  - Um nó exige 1KB de espaço.

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

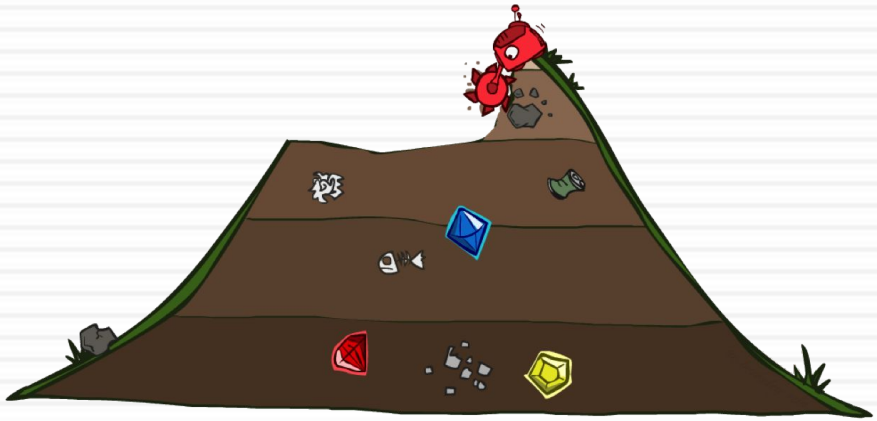
AIMA, Figure 3.13

BFS: implementação

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    node = getStartNode(problem)

    frontier = util.Queue()
    frontier.push(node)
    explored = set()

    while not frontier.isEmpty():
        node = frontier.pop()

        if node['STATE'] in explored:
            continue

        explored.add(node['STATE'])

        if problem.isGoalState(node['STATE']):
            return getActionSequence(node)

        for sucessor in problem.expand(node['STATE']):
            child_node = getChildNode(sucessor,node)
            frontier.push(child_node)

    return []
```

search.py

Busca com Custo Uniforme (*Uniform Cost Search, UCS*)



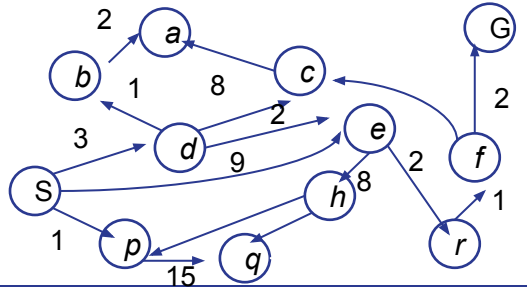
UCS

- Similar à BFS, só que expande o nó ainda não expandido *n* que tenha o custo de caminho *g(n)* mais baixo.
- Implementação:
  - *fronteira* = fila ordenada por g(n) (i.e., uma **fila de prioridades**)
- Equivalente à busca em largura se os custos são todos iguais.
- Particularidades da UCS:
  - O teste de objetivo é aplicado a um nó quando ele é selecionado para expansão, e não quando ele é gerado.
  - Há um teste para verificar se um caminho para um nó na fronteira é mais curto do que algum previamente encontrado.

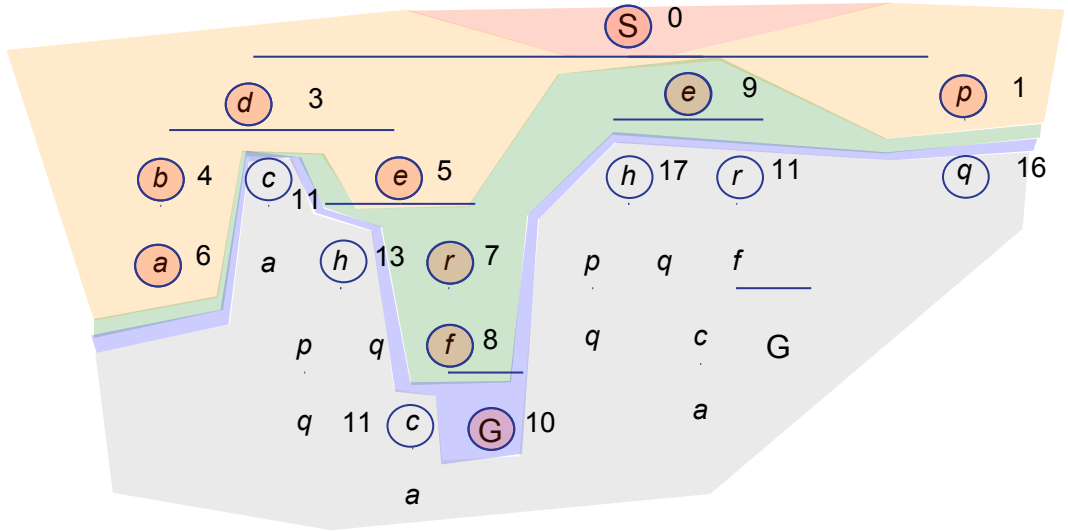
UCS

Estratégia: expandir o nó mais barato primeiro:

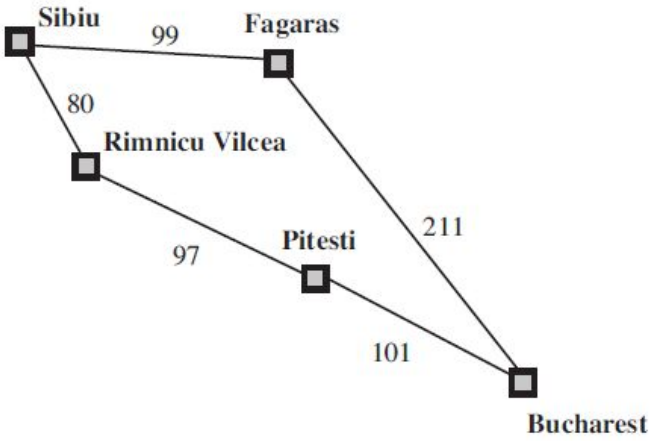
Borda é uma fila de prioridade com chave igual ao custo cumulativo, g(n).



Curvas de custo



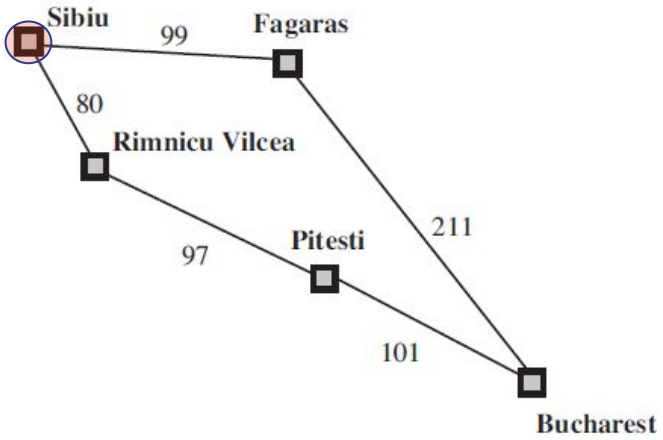
UCS: algoritmo e exemplo




```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

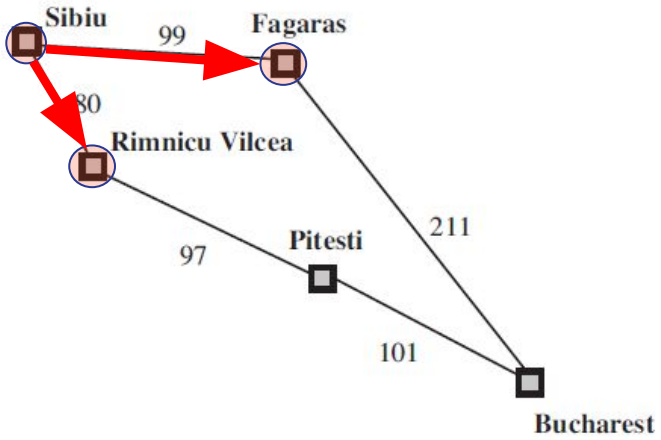
UCS: algoritmo e exemplo

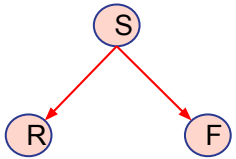
Fronteira: {(S,0)}



UCS: algoritmo e exemplo

Fronteira: {(SR,80); (SF,99)}

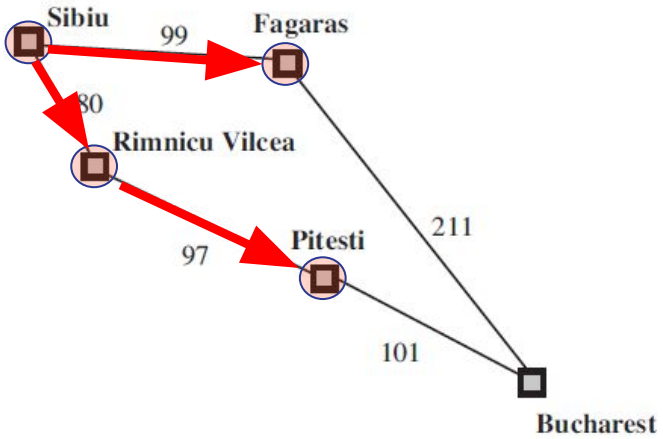


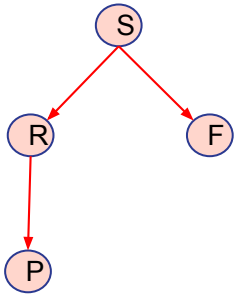


Os sucessores de **S**ibiu são **R**imnicu Vilcea e **F**agaras, com custos de 80 e 99, respectivamente.

UCS: algoritmo e exemplo

Fronteira: {(SRP,177); (SF,99)}

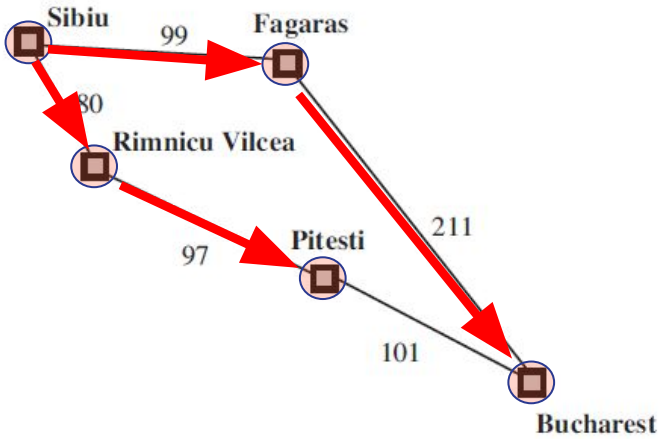


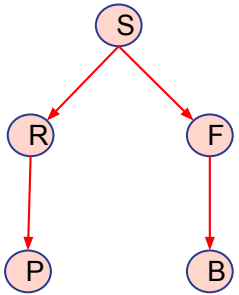


O nó de menor custo, Rimnicu Vilcea, é expandido em seguida, acrescentando **P**itesti com custo de 80 + 97 = 177.

UCS: algoritmo e exemplo

Fronteira: {(SRP,177); (SFB,310)}

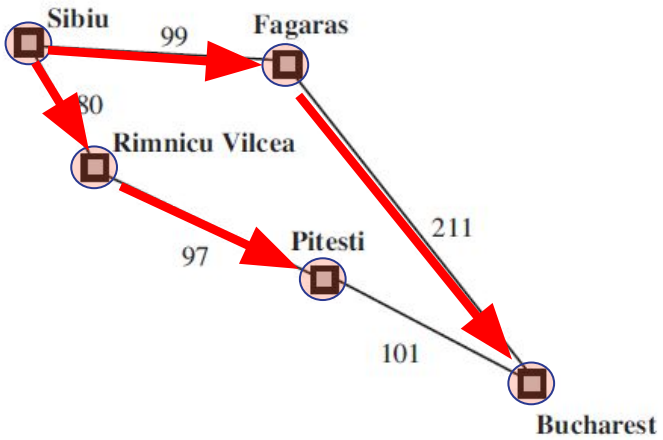


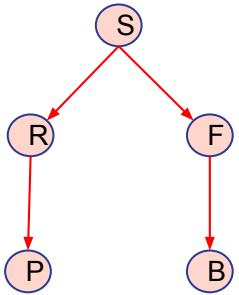


O nó de menor custo é agora **F**agaras; sendo assim, ele é expandido, acrescentando **B**ucareste com custo 99 + 211 = 310.

UCS: algoritmo e exemplo

Fronteira: {(SRP,177); (SFB,310)}

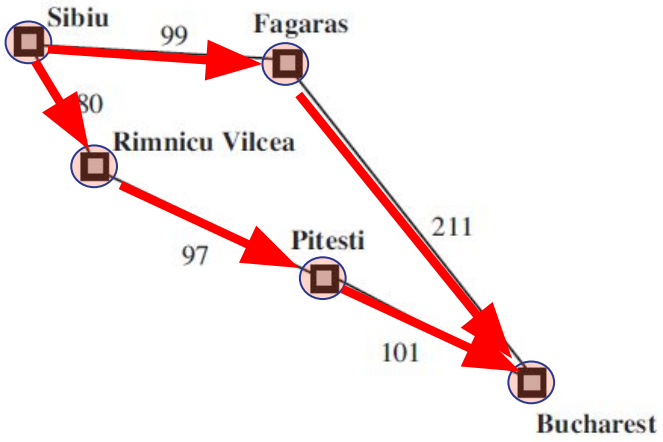


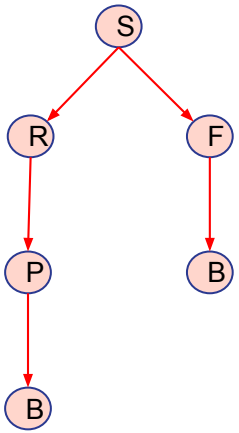


Agora um nó objetivo foi gerado, mas o algoritmo continua escolhendo SRP para expansão.

UCS: algoritmo e exemplo

Fronteira: {(SRPB,278); (SFB,310)}

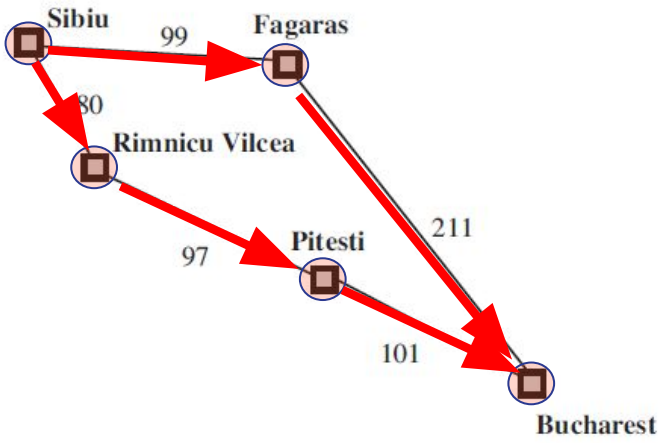


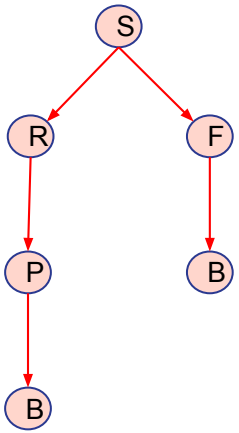


Como resultado, um segundo caminho para Bucareste (com custo 80 + 97 + 101 = 278) é adicionado à fronteira.

UCS: algoritmo e exemplo

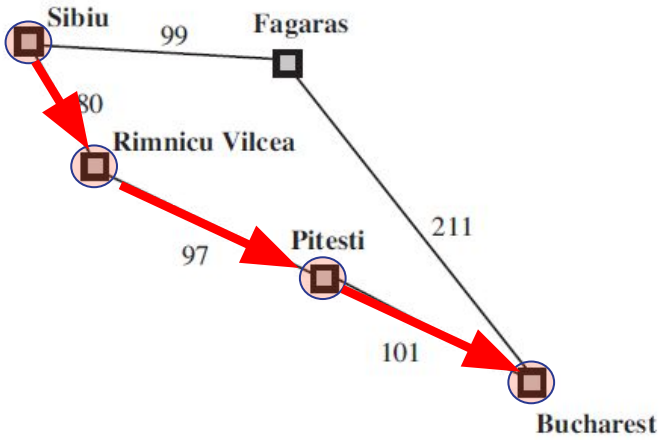
Fronteira: {(SFB,310)}

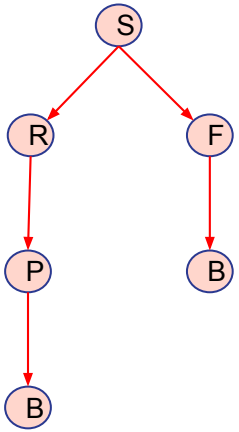




O algoritmo agora seleciona para expansão o plano de menor custo. Repare que o plano anterior é descartado (i.e., permanece da fronteira).

UCS: algoritmo e exemplo

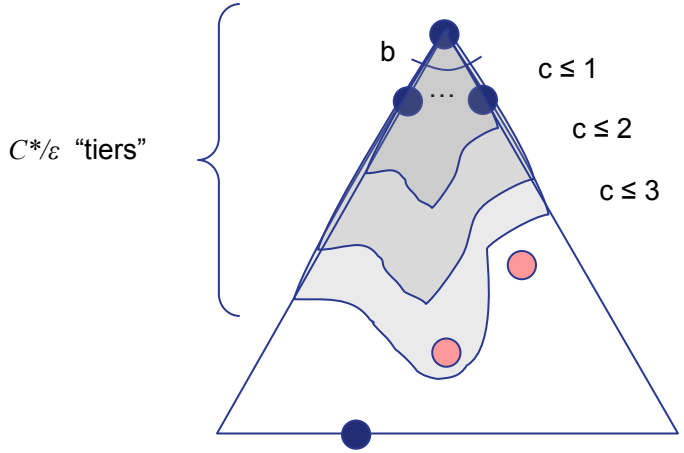




O novo plano, com o valor g(SRPB) = 278, é a solução encontrada.

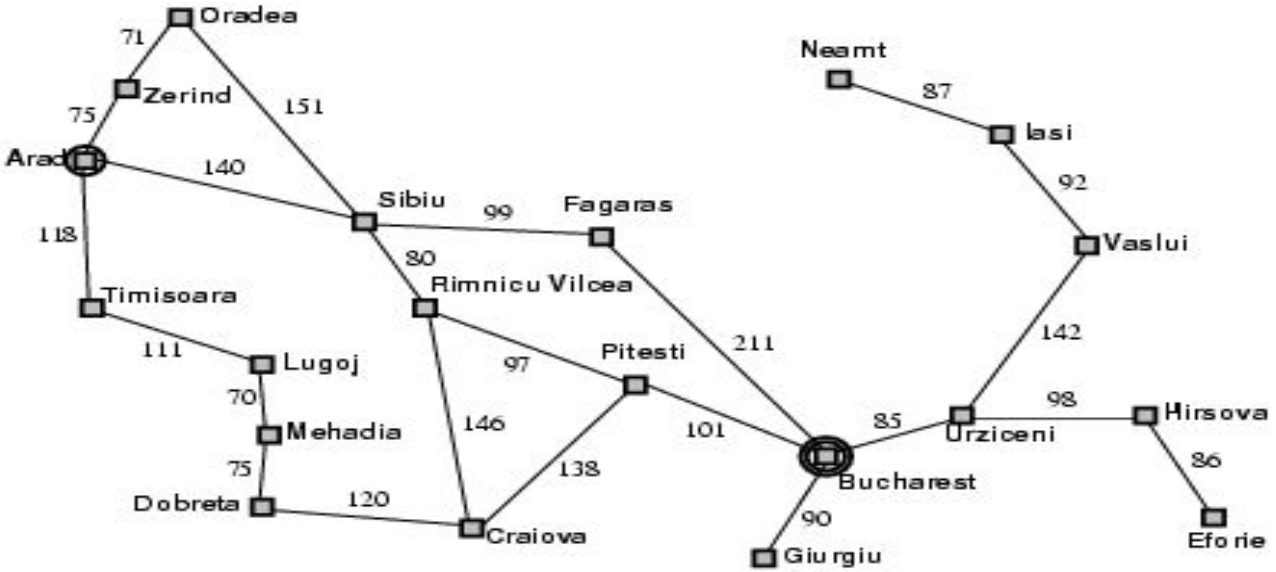
UCS: propriedades

- Completa? Sim, se o custo de cada passo ≥ ε
- Complexidade de Tempo? Quantidade de nós com $g \leq$ custo da solução ótima, $O(b^{\lceil C^*/\varepsilon \rceil})$ onde C^* é o custo da solução ótima
- Complexidade de Espaço? Quantidade de nós com $g \leq$ custo da solução ótima, $O(b^{\lceil C^*/\varepsilon \rceil})$
- Ótima? Sim, pois os nós são expandidos em ordem crescente de custo total.

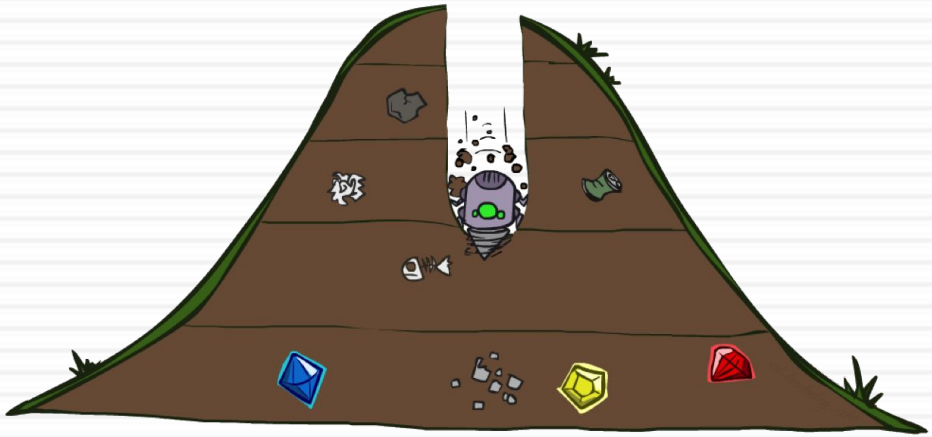


UCS: exercício

- Aplicar UCS para achar o caminho mais curto entre Arad e Bucareste.



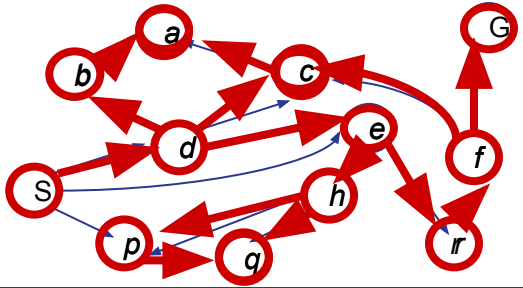
Busca em Profundidade (*Depth First Search - DFS*)

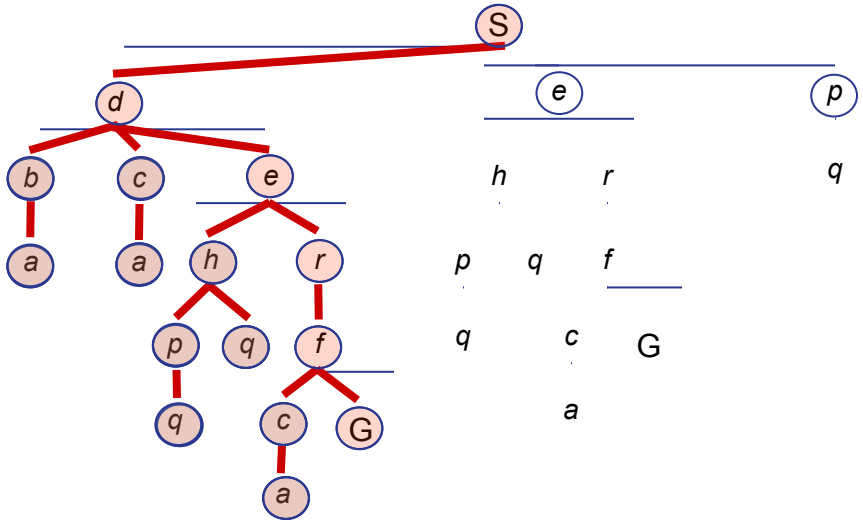


DFS: exemplo

Estratégia: expandir primeiro o nó mais distante da raiz

Implementação: borda é uma pilha LIFO stack





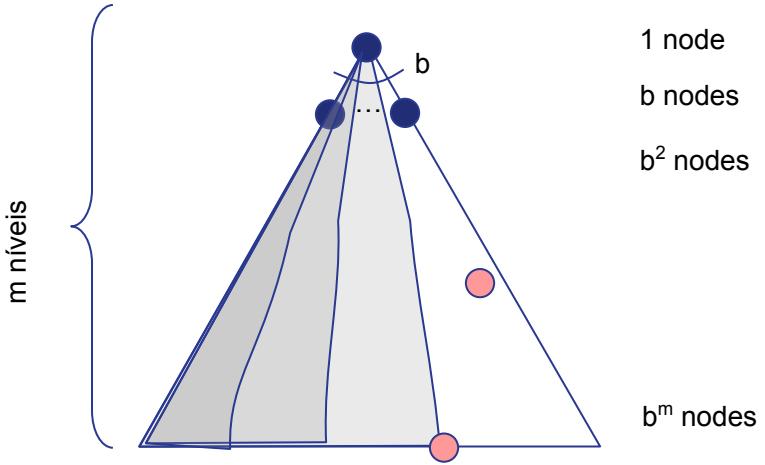
DFS: implementação

```
130  def depthFirstSearch(problem):
131      node = getStartNode(problem)
132
133      frontier = util.Stack()
134      frontier.push(node)
135      explored = set()
136
137      while not frontier.isEmpty():
138          node = frontier.pop()
139
140          if node['STATE'] in explored:
141              continue
142
143          explored.add(node['STATE'])
144
145          if problem.isGoalState(node['STATE']):
146              return getActionSequence(node)
147
148          for sucessor in problem.expand(node['STATE']):
149              child_node = getChildNode(sucessor,node)
150              frontier.push(child_node)
151
152      return []
```

search.py

DFS: propriedades

- Complexidade de tempo?
  - A DFS expande algum prefixo à esquerda da árvore de busca.
  - Potencial de processar a árvore inteira!
  - Se m é finito, complexidade de espaço é $O(b^d)$
- Complexidade de espaço?
  - Tem apenas “irmãos” no caminho para a raiz, então O(bd)
- Completa?
  - d pode ser infinito, então completa apenas se evitar ciclos (ver DLS)
- Ótima?
  - Não. DFS encontra a solução “mais à esquerda” na árvore de busca, independente da profundidade ou custo correspondente.



Busca em Profundidade Limitada (*depth-limited search, DLS*)

Busca em profundidade limitada (*depth limited search*, DLS)

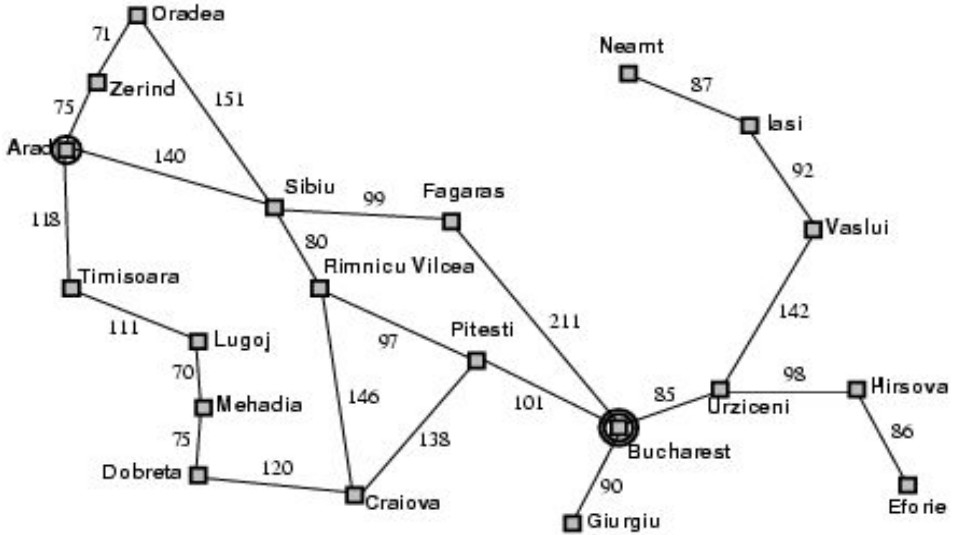
- DFS não vai encontrar um objetivo se a busca entrar em um caminho de comprimento infinito.
  - A menos que evite ciclos.
- **Solução (**DLS**)**: definir um limite de profundidade (*L) para a árvore a ser expandida*.
  - Planos com profundidade maior do que *L* na árvore de busca não são expandidos.
- A DFS é um caso particular da DLS, com $L = \infty$

Busca em profundidade limitada (DLS) – valor de L

- Repare que, se L for...
  - ...muito pequeno (i.e., o objetivo se encontra em uma profundidade maior do que L), uma solução não será encontrada.
  - ...muito grande, soluções subótimas podem ser encontradas.
- Estratégia útil quando a profundidade máxima da solução para o problema de busca é conhecida.

Busca em profundidade limitada (DLS) – valor de L

- Qual seria um valor adequado para *L* no caso do problema da Romênia?
  - Há 20 cidades no mapa; então não existe plano com mais de 19 ações.
  - Na verdade, cada cidade pode ser alcançada a partir de outra com até 9 ações.



Busca em profundidade limitada (DLS)

L=3

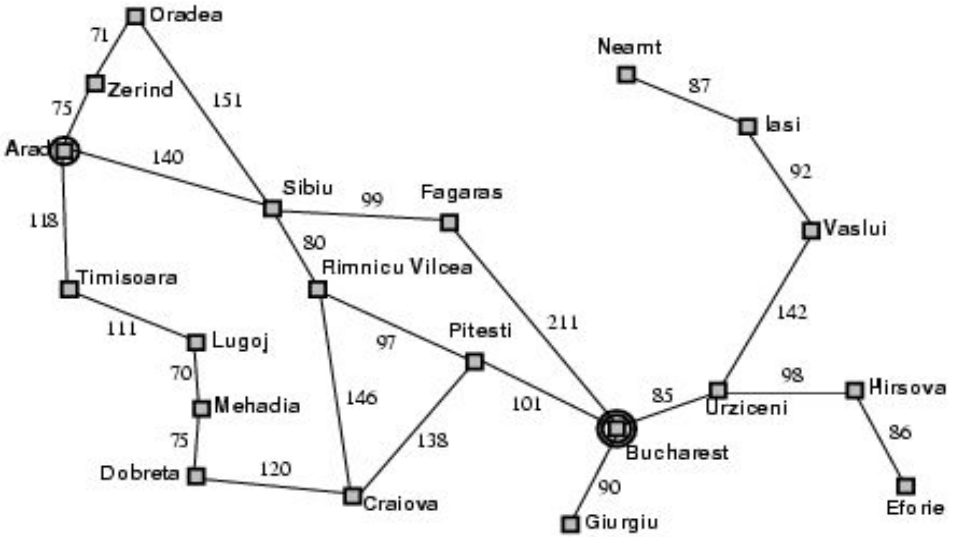
{A}
{AZ, AS, AT}
{AZ, ASF, ASR, ASO, AT}
{AZ, ASFB, ASR, ASO, AT}

Solução encontrada (ASFB)

L=2

{A}
{AZ, AS, AT}
{AZ, ASF, ASR, AT}
{AZ, ASR, AT}
{AZ, AT}
{AZ, ATL}
{AZ}
{AZO}
{}

Solução não encontrada



NB: ignorando nós mais recentemente expandido

DLS: pseudocódigo

- Implementação recursiva:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

cutoff indica que o limite de profundidade foi alcançado.

chamada recursiva

failure indica que nenhuma solução foi encontrada

DLS: propriedades

- Completa? Não, pois a solução (objetivo) pode estar além do limite estabelecido (i.e., pode ser que $L < m$).
- Tempo? $O(b^L)$
- Espaço? $O(bL)$
- Ótima? Não

Busca em Profundidade Iterativa (*Depth-First Iterative Deepening Search, IDS*)

IDS

- A IDS consiste em aplicar repetidamente a busca em profundidade limitada (DLS), com limites gradativamente crescentes.
- A IDS termina quando uma solução for encontrada, ou se a busca em profundidade limitada retorna falha (*failure*), o que significa que não existe uma solução.

IDS

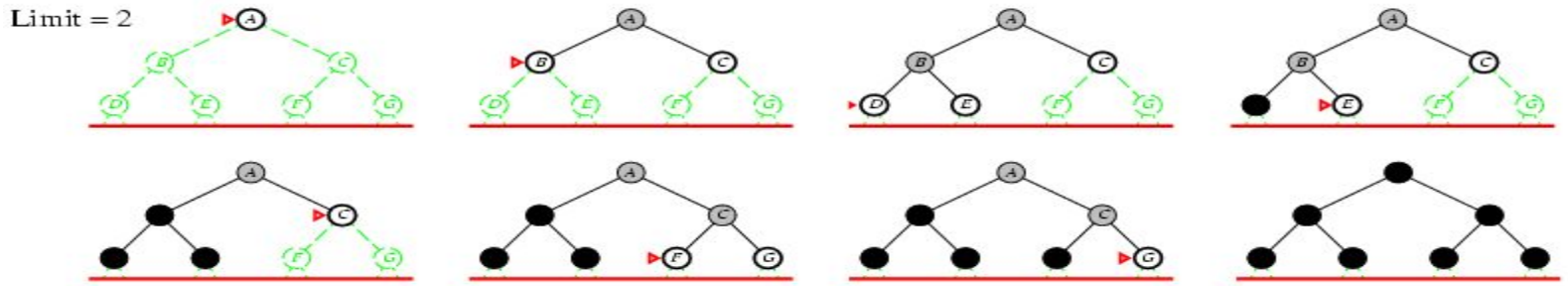
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

IDS – exemplo: / =0

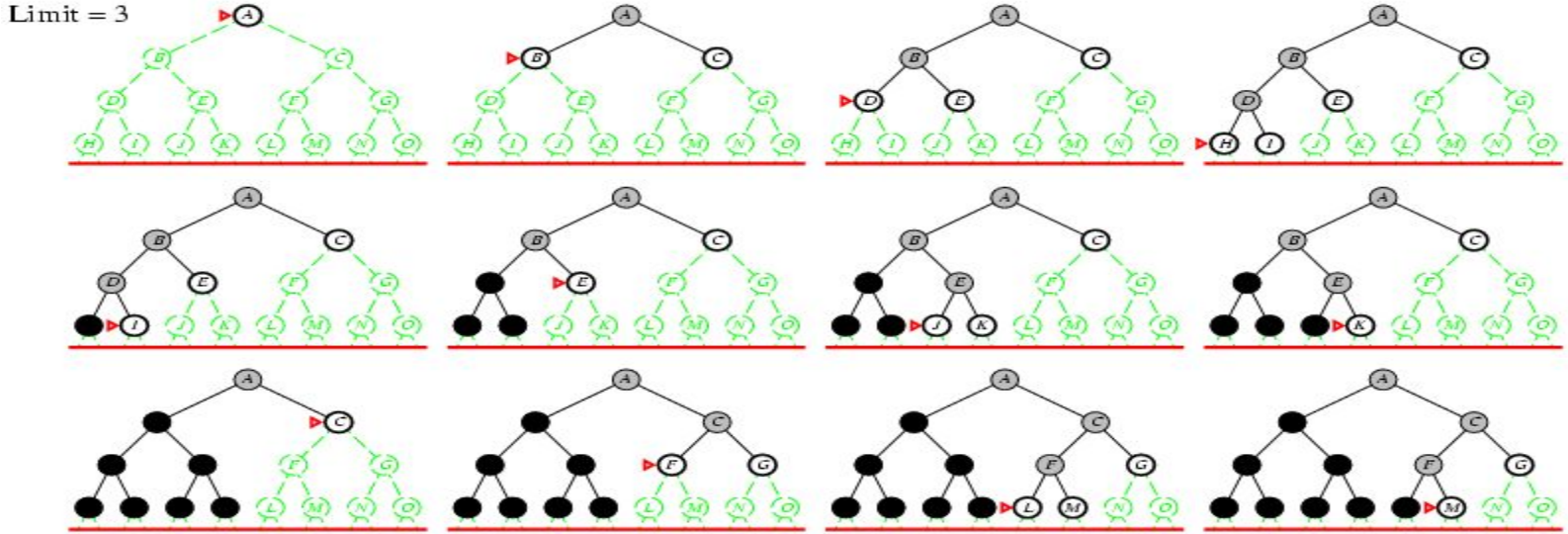
$\mathbf{Limit = 0}$

IDS – exemplo: *l* =1

IDS – exemplo: *l* =2

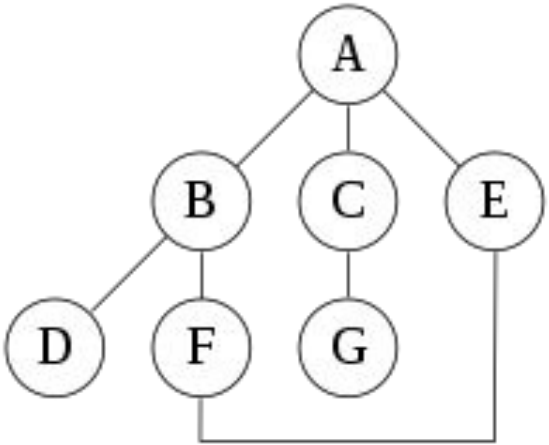


IDS – exemplo: *l* =3



IDS – exemplo

- Considere o grafo de espaço de estados a seguir, em que A é o estado inicial, e G é o único objetivo.



Fonte do exemplo: Wikipedia (https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search)

IDS – exemplo (cont.)

- 0: A
- 1: A, B, C, E
  - Observe que a IDS já viu C; uma busca usando DFS não o faria.
- 2: A, B, D, F, C, G, E, F
  - Note que IDS ainda vê C, mas um pouco depois. Note também que ele vê E por meio de um caminho diferente, e volta em F duas vezes.
- 3: A, B, D, F, E, C, G, E, F, B

Fonte do exemplo: Wikipedia (https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search)

IDS - propriedades

- Número de nós gerados em uma busca de extensão com fator de ramificação *b*:

$$N_{BE} = b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d + (b^{d+1} - b)$$

- Número de nós gerados em uma busca de aprofundamento iterativo até a profundidade *d* com fator de ramificação *b*:

$$N_{BAI} = (d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- Para *b = 10*, *d = 5*,
  - $N_{BE} = 10 + 100 + 1.000 + 10.000 + 100.000 + 999.990 = 1.111.100$
  - $N_{BAI} = 6 + 50 + 400 + 3.000 + 20.000 + 100.000 = 123.456$
- Overhead = (123.456 – 111.111)/111.111 = 11%

IDS – propriedades (cont.)

- Completa? Sim
- Tempo? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Espaço? $O(bd)$
- Ótima? Sim, se custo de caminho = 1

- Repare que a IDS usa somente espaço linear e não muito mais tempo que outros algoritmos de busca sem informação.

Comentários finais

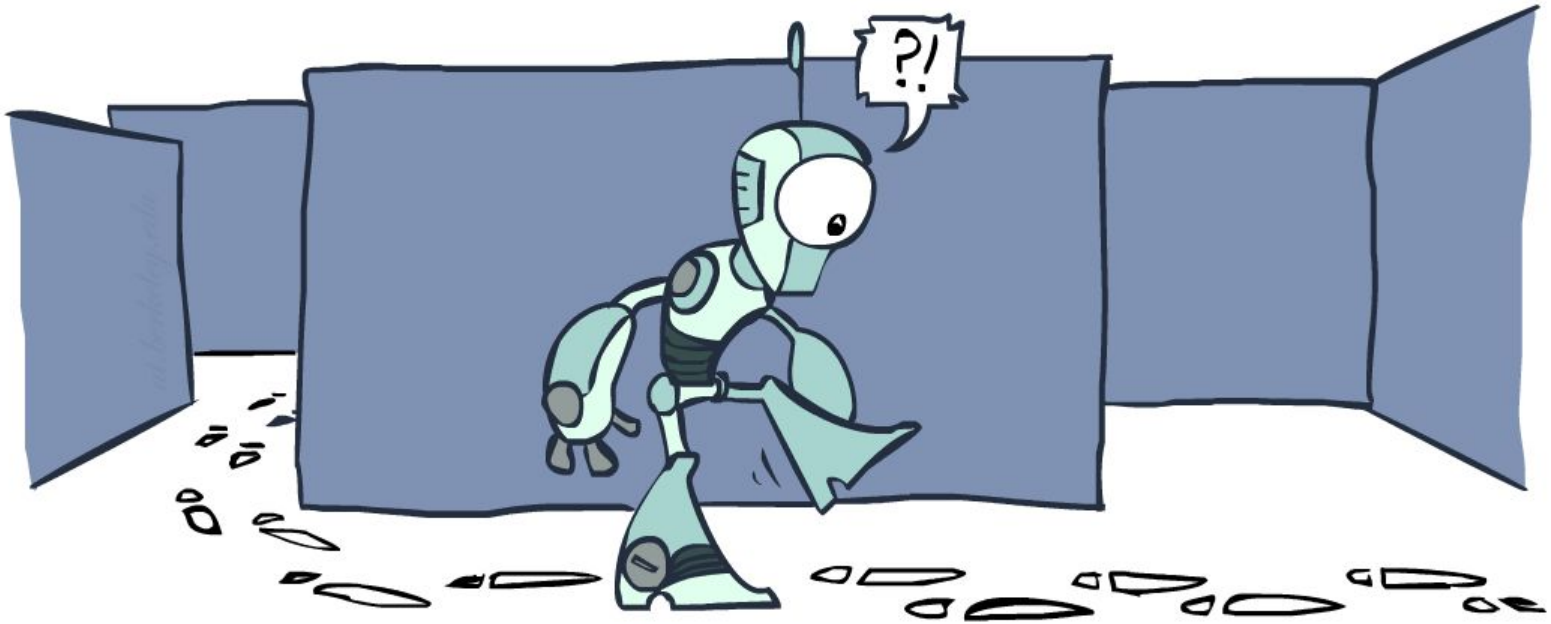
Comentários finais

- A formulação de problemas usualmente requer a abstração de detalhes do mundo real para que seja definido um espaço de estados que possa ser explorado por meio de algoritmos de busca.
- Há uma variedade de estratégias de busca sem informação (ou busca cega) além dos estudados aqui.

Comentários finais

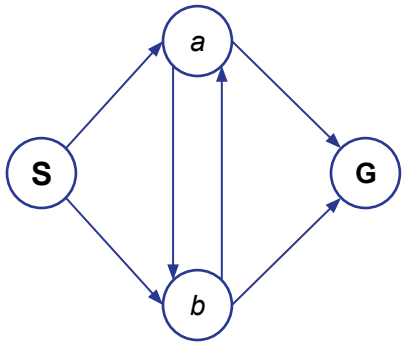
- Algoritmos de busca ainda ocupam uma posição fundamental no projeto de agentes inteligentes.
- Diversas abordagens em IA têm alguma relação com algum tipo de busca em estados.
- Ler o cap. 3 do livro texto (Russell & Norvig)
- Ver documentário “AlphaGo” (procurar no Youtube)

Busca em Grafo (*Graph Search*)



Grafos de Espaço de Estados vs. Árvores de Busca

Considere este grafo com 4 estados:

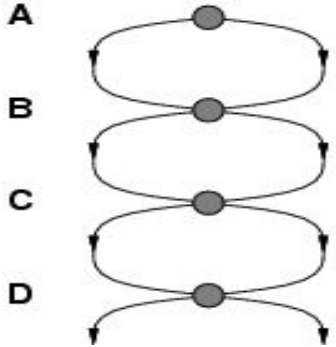


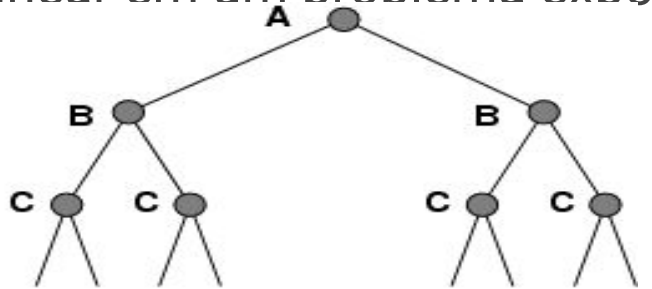
O quão grande é a árvore de busca correspondente (a partir de **S**)?

Em geral, há problemas que podem gerar estados repetidos em uma árvore de busca!

Estados repetidos

- A busca em árvore pode perder tempo expandindo nós já explorados antes.
- Estados repetidos podem
  - levar a loops infinitos;
  - transformar um problema linear em um problema exponencial.





Detecção de estados repetidos

- Comparar os nós prestes a serem expandidos com nós já visitados.
  - Se o nó já tiver sido visitado, será descartado; coleção “closed” armazena nós já visitados.

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

Tree Search vs Graph Search

- Duas formas possíveis de implementar algoritmos de busca.
  - Graph Search é apenas uma extensão da Tree Search.

Tree Search

```
open <- []
next <- start
while next isn't goal {
  open += successors of next
  next <- select from open
  remove next from open
}
return next
```

Graph Search

```
open <- []
closed <- []
next <- start
while next isn't goal {
  closed += next
  open += successors of next, which are not in closed
  next <- select from open
  remove next from open
}
return next
```