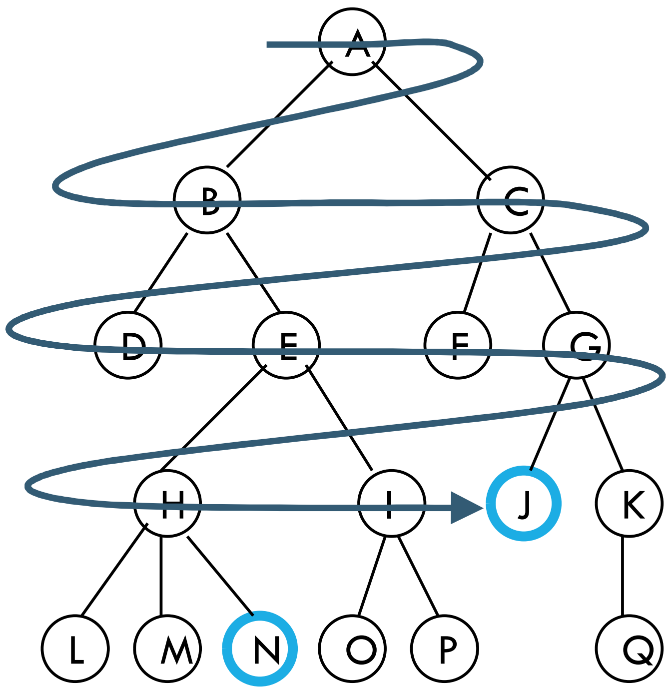
ARTIFICIAL INTELLIGENCE

Lecture 05

BFS: SUMMARY



BFS: COMPLETENESS

Completeness: Yes (guaranteed to find a solution if there exists one)

if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes (provided the branching factor b is finite) .

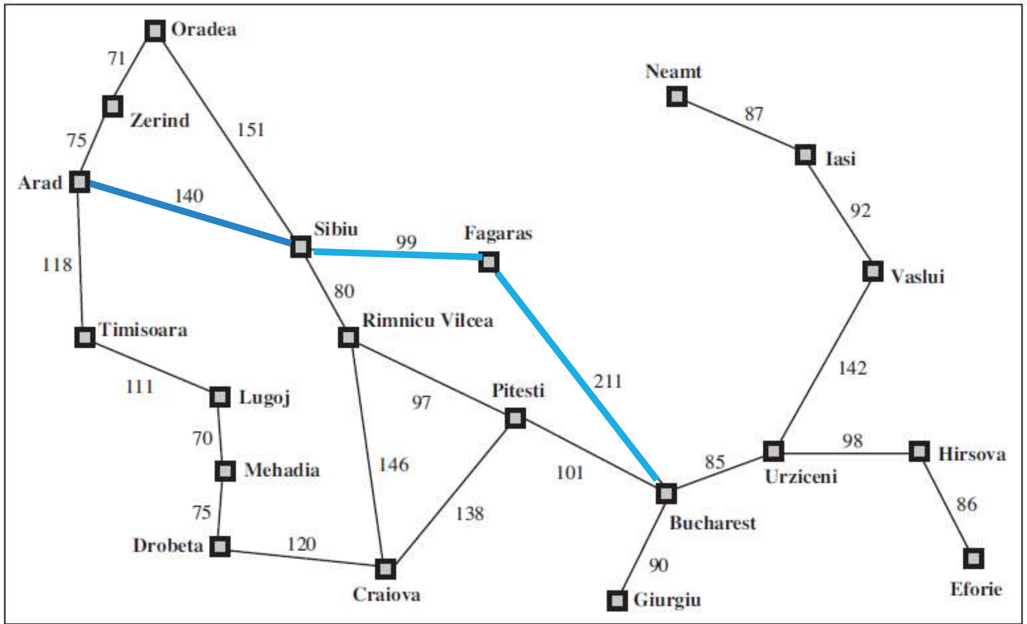


Figure 3.2 A simplified road map of part of Romania.

BFS goal path

BFS: OPTIMAL?

Not necessarily optimal

Only optimal if every action has same cost.

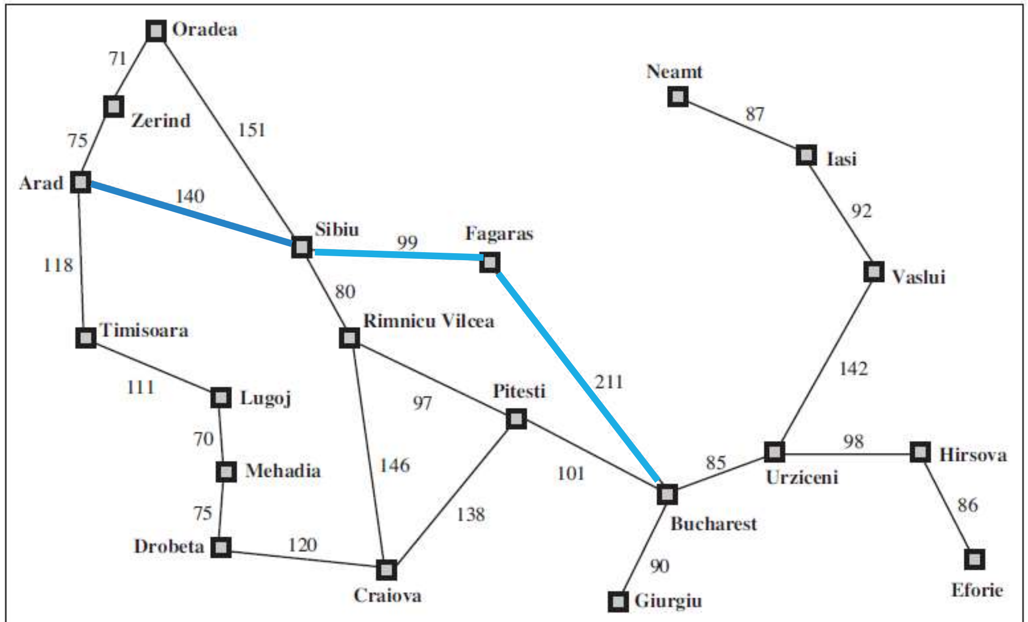


Figure 3.2 A simplified road map of part of Romania.

BFS goal path

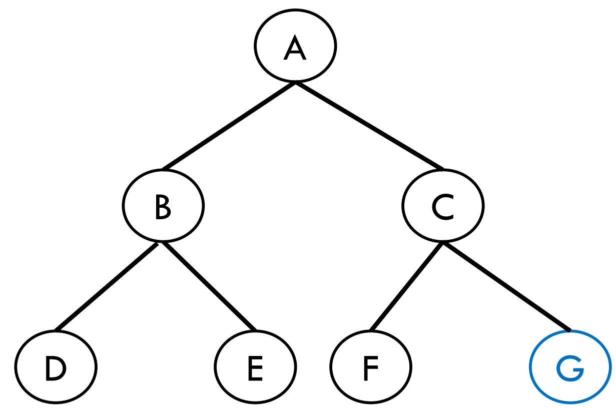
BFS: TIME COMPLEXITY?

Worst complexity is when G is a goal state.

In this case, total number of nodes generated is

$b+b^2+b^3 + \ldots + b^d = O(b^d)$ (The d^{th} layer contains nodes much larger than all the nodes in previous layers combined!)

So the time complexity is $O(b^d)$



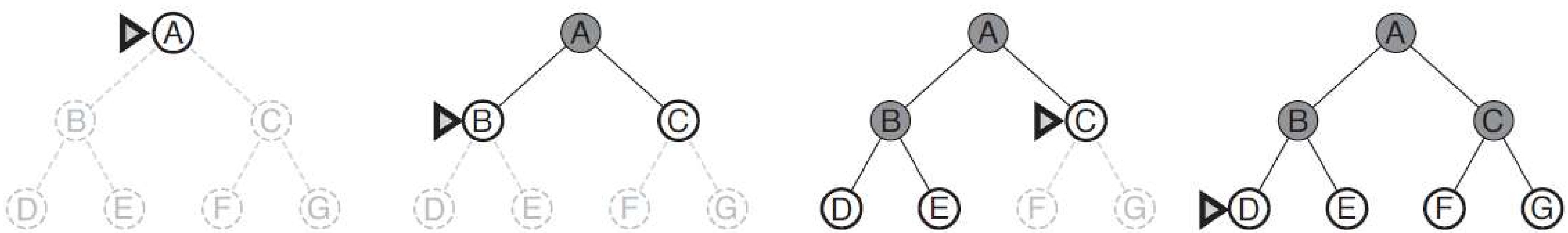
A binary tree, b=2, d=2

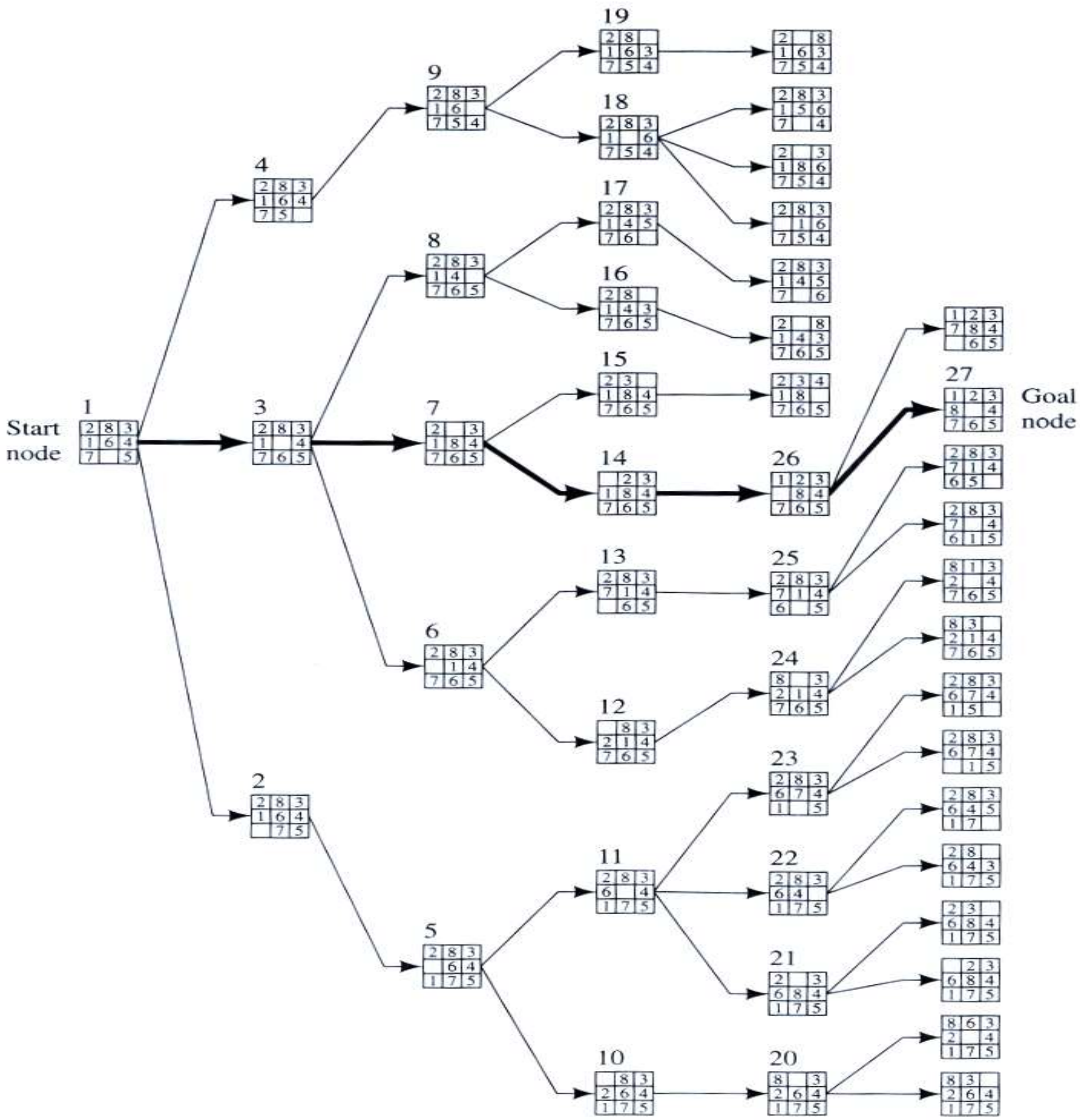
BFS: SPACE COMPLEXITY?

There will be $O(b^{d-1})$ nodes in the explored set and $O(b^{d})$ nodes in the frontier.

So the space complexity is $O(b^{d})$, i.e., it is dominated by the size of the frontier.

Exponential time complexity can be accepted but exponential space complexity is BAD !





Start node
Goal node
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

UNIFORM COST SEARCH

BFS is optimal only if each step has same cost.

Uniform cost search modifies BFS such that it works with any cost function.

Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the *lowest path cost* g(n).

This is done by storing the frontier as a priority queue ordered by g.

BFS – UCS DIFFERENCES

Goal test is applied to a node when it is *selected for expansion* rather than when it is first generated. (The reason is that the first goal node that is *generated* may be on a suboptimal path)

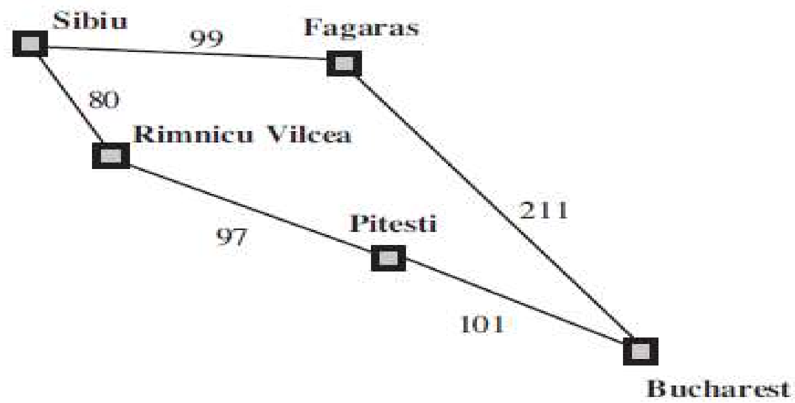
The second difference is that a test is added in case a better path is found to a node currently on the frontier

REMEMBER?

- n.STATE: the state in the state space to which the node corresponds;
- n.PARENT: the node in the search tree that generated this node;
- n.ACTION: the action that was applied to the parent to generate the node;
- n.PATH-COST: the cost, traditionally denoted by $g(n)$, of the path from the initial state to the node, as indicated by the parent pointers.

UCS- EXAMPLE

- Node
- Child




```
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
```

Frontier = [S]
Explored = [ ]

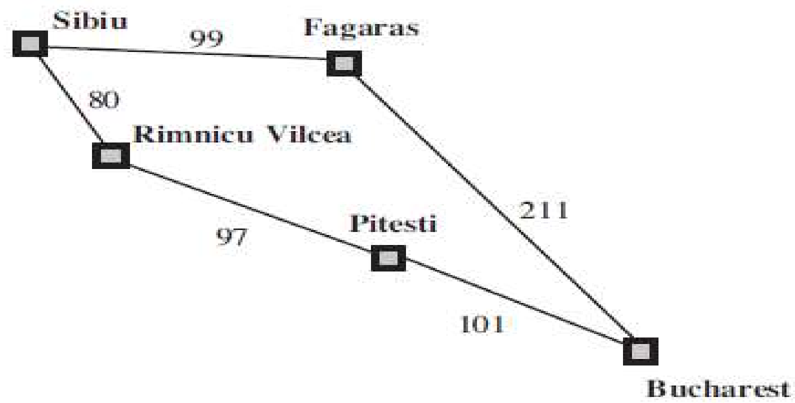
node.state="Sibiu"
node.parent=[]
node.action=[]
node.path_cost=0

S

UCS- EXAMPLE

- Node
- Child

S.state="Sibiu"
S.parent=[]
S.action=[]
S.path_cost=0



Frontier = [S]
Explored = [ ]

S.state=="Bucharest"?

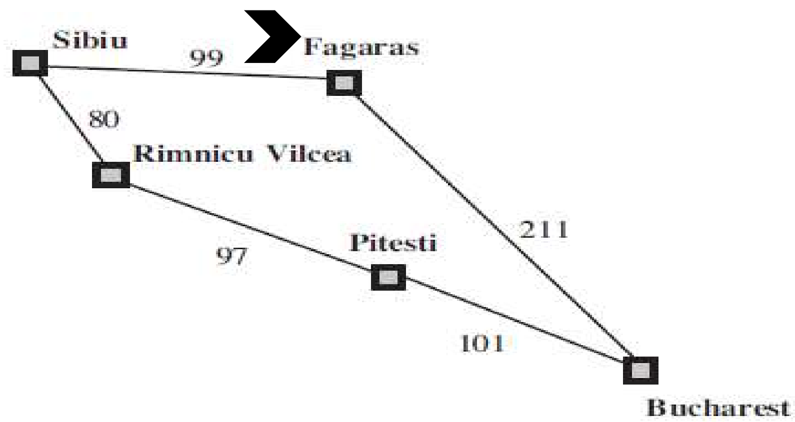
Frontier = [ ]
Explored = [S]

```
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- EXAMPLE

- Node
- Child

F.state="Fagaras"
F.parent=[S]
F.action=[Go(F)]
F.path_cost= 99




```
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
```

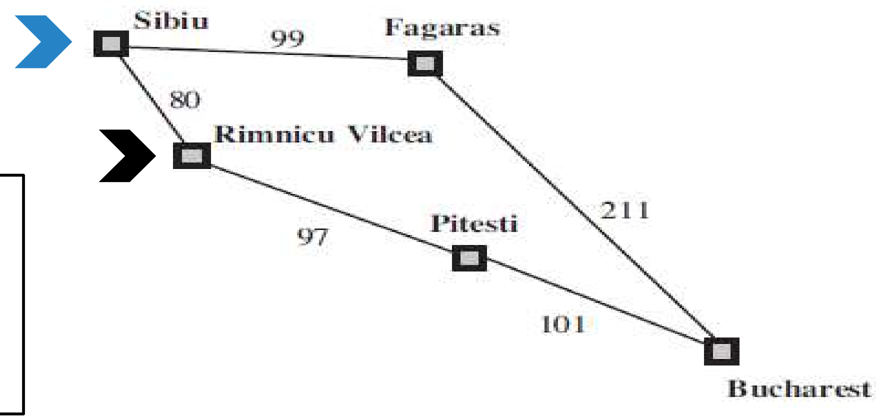
Frontier = [ ]
Explored = [S]

Frontier = [F]

UCS- EXAMPLE

- Node
- Child

R.state="Rimnicu"
R.parent=[S]
R.action=[Go(R)]
R.path_cost= 80




```
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
```

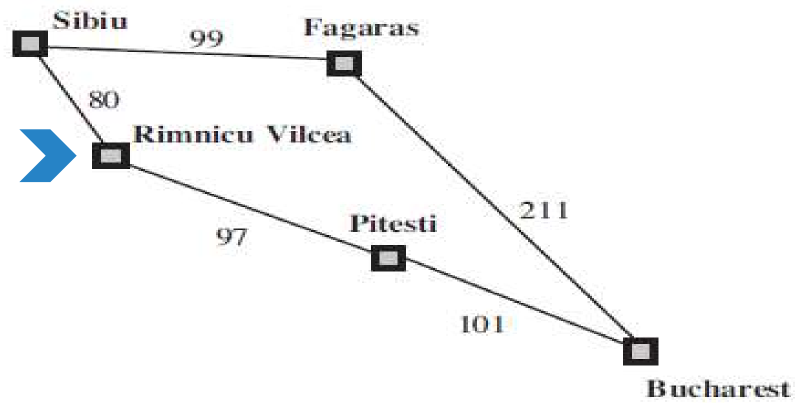
Frontier = [F]
Explored = [S]

Frontier = [F, R]

UCS- EXAMPLE

- Node
- Child

R.state="Rimnicu"
R.parent=[S]
R.action=[Go(R)]
R.path_cost= 80



Frontier = [F, R]
Explored = [S]

R.state=="Bucharest"?

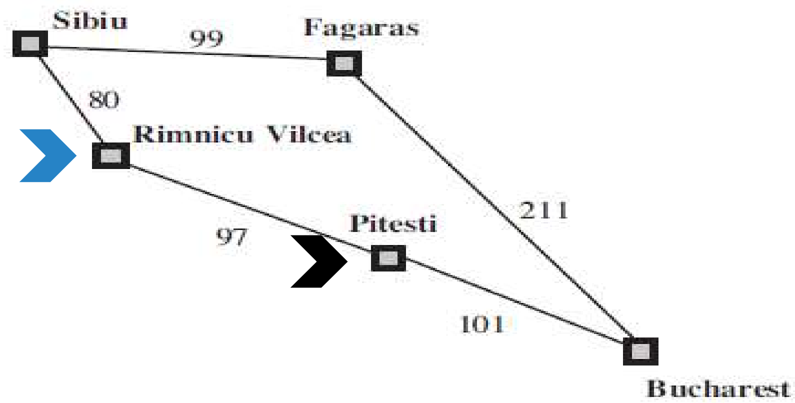
Frontier = [F]
Explored = [S, R]

```
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- EXAMPLE

- Node
- Child

P.state="Pitesti"
P.parent=[R]
P.action=[Go(P)]
P.path_cost= 80+97



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*

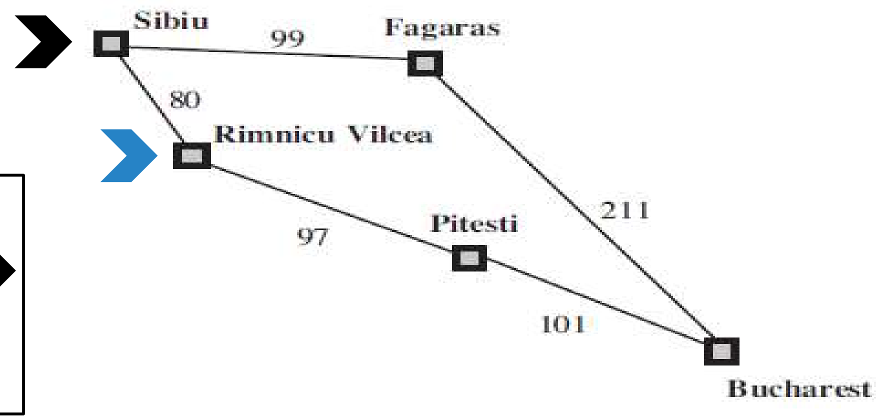
Frontier = [F]
Explored = [S, R]

Frontier = [F, P]

UCS- EXAMPLE

- Node
- Child

S.state="Sibiu"
S.parent=[R]
S.action=[Go(S)]
S.path_cost=80+80



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*

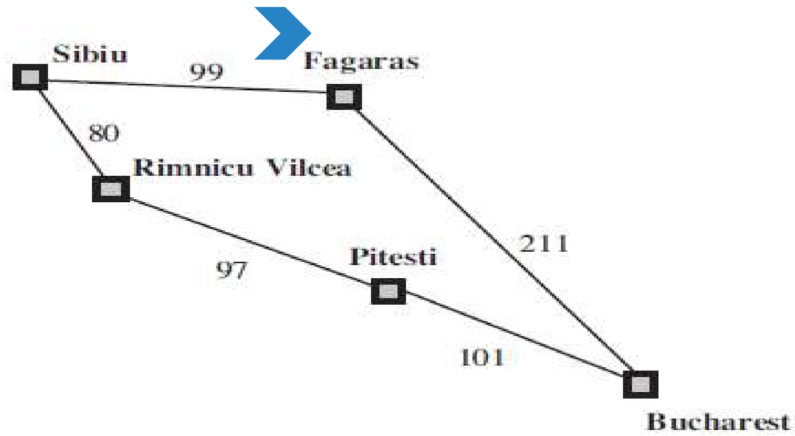
Frontier = [F, P]
Explored = [S, R]

Both are false, no
Change in frontier
Or Explored

UCS- EXAMPLE

- Node
- Child

F.state="Fagaras"
F.parent=[S]
F.action=[Go(F)]
F.path_cost= 99




```
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
```

Frontier = [F, P]
Explored = [S, R]

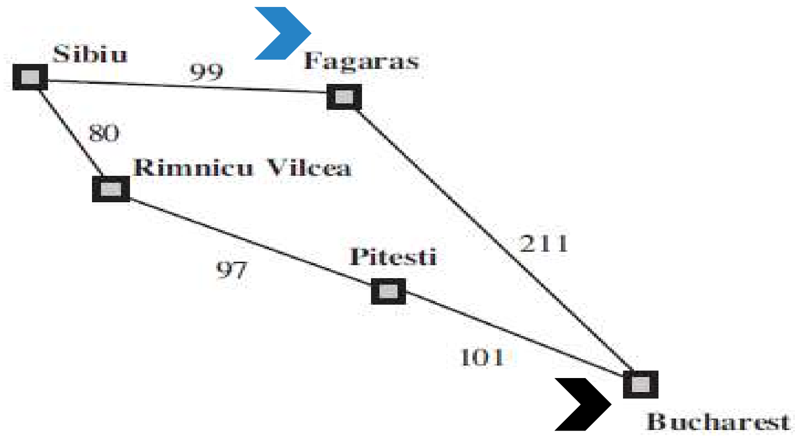
F.state=="Bucharest"?

Frontier = [P]
Explored = [S, R, F]

UCS- EXAMPLE

- Node
- Child

B.state="Bucharest"
B.parent=[F]
B.action=[Go(B)]
B.path_cost= 99 + 211




```
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
```

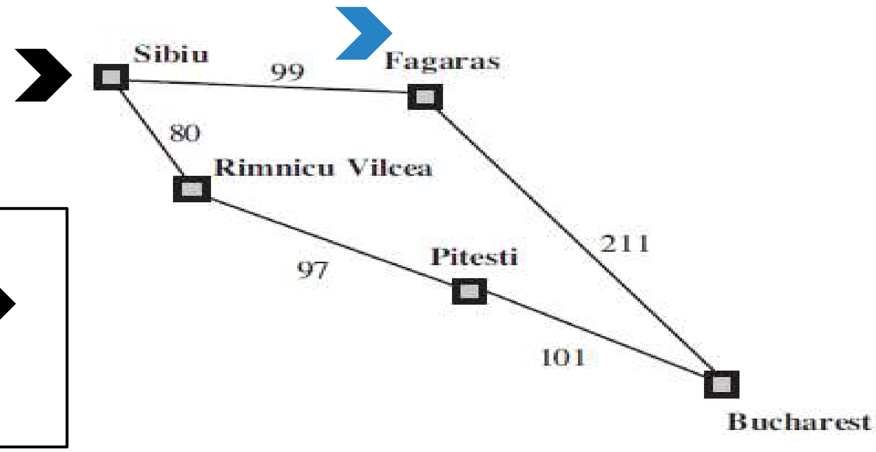
Frontier = [P]
Explored = [S, R, F]

Frontier = [P, B]

UCS- EXAMPLE

- Node
- Child

S.state="Sibiu"
S.parent=[F]
S.action=[Go(S)]
S.path_cost=99+80



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
```

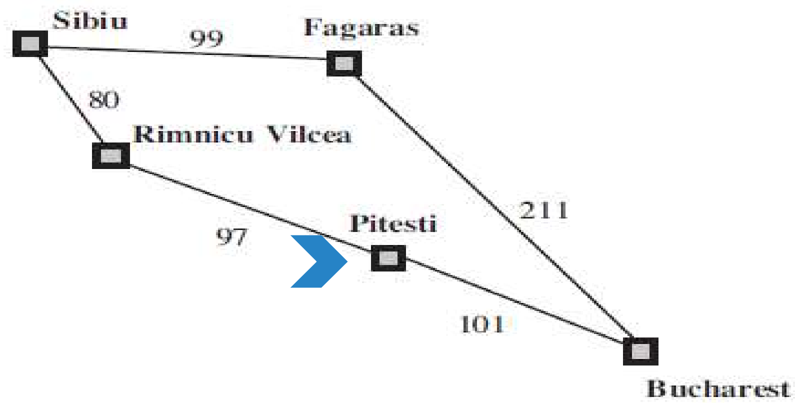
Frontier = [P, B]
Explored = [S, R, F]

Both are false, no
Change in frontier
Or Explored

UCS- EXAMPLE

- Node
- Child

P.state="Pitesti"
P.parent=[R]
P.action=[Go(P)]
P.path_cost= 80+97



Frontier = [P, B]
Explored = [S, R, F]

P.state=="Bucharest"?

Frontier = [B]
Explored = [S, R, F, P]

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

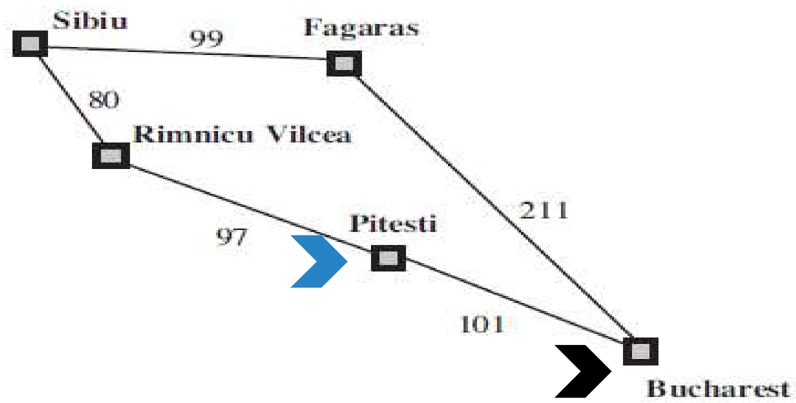
```
node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
frontier ← a priority queue ordered by PATH-COST, with node as the only element
explored ← an empty set
loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
    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- EXAMPLE

Node

Child

B.state="Bucharest"
B.parent=[P]
B.action=[Go(B)]
B.path_cost= 80+97+101




```
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
```

Frontier = [B]

Explored = [S, R, F, P]

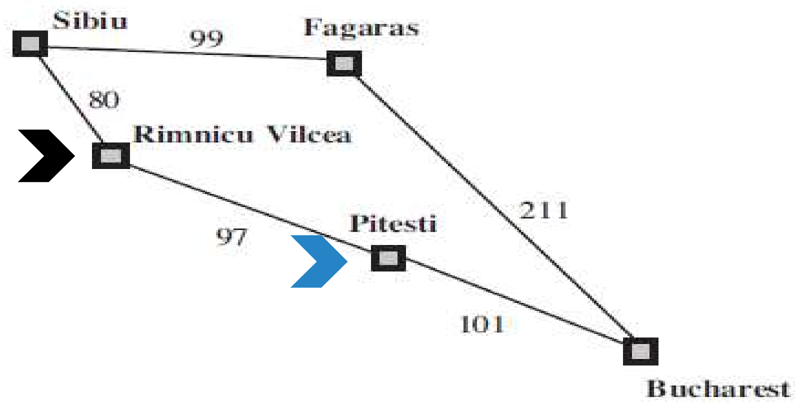
Replace old B with new B, both have same states but different parents, consequently different costs

Frontier = [B]

UCS- EXAMPLE

- Node
- Child

R.state="Rimnicu"
R.parent=[P]
R.action=[Go(R)]
R.path_cost= 80+97+97



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*

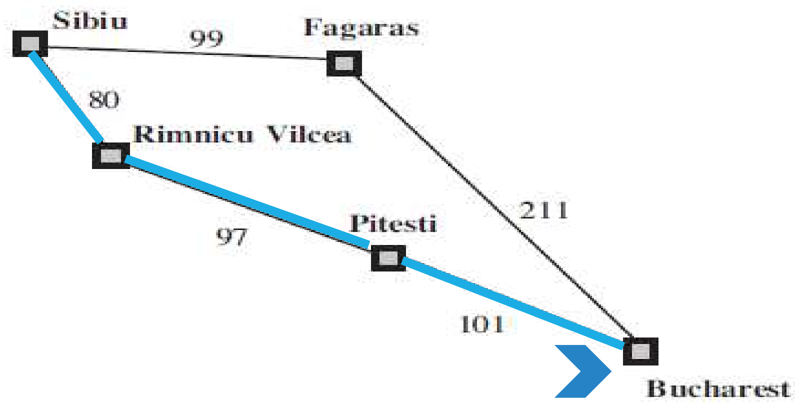
Frontier = [B]
Explored = [S, R, F, P]

Both are false, no
Change in frontier
Or Explored

UCS- EXAMPLE

- Node
- Child

B.state="Bucharest"
B.parent=[P]
B.action=[Go(B)]
B.path_cost= 80+97+101



Frontier = [B]
Explored = [S, R, F, P]

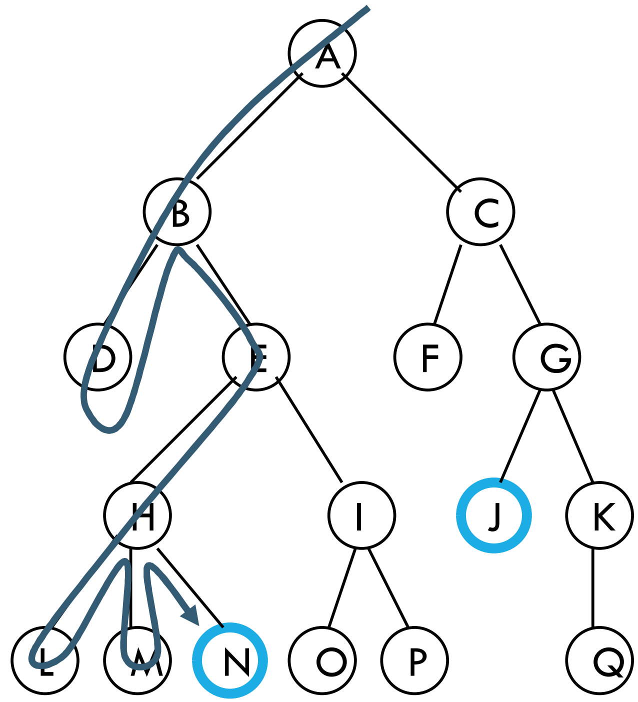
B.state=="Bucharest"?

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*

DEPTH FIRST SEARCH

DFS always expand deepest node first.

DFS uses LIFO (whereas BFS used FIFO)



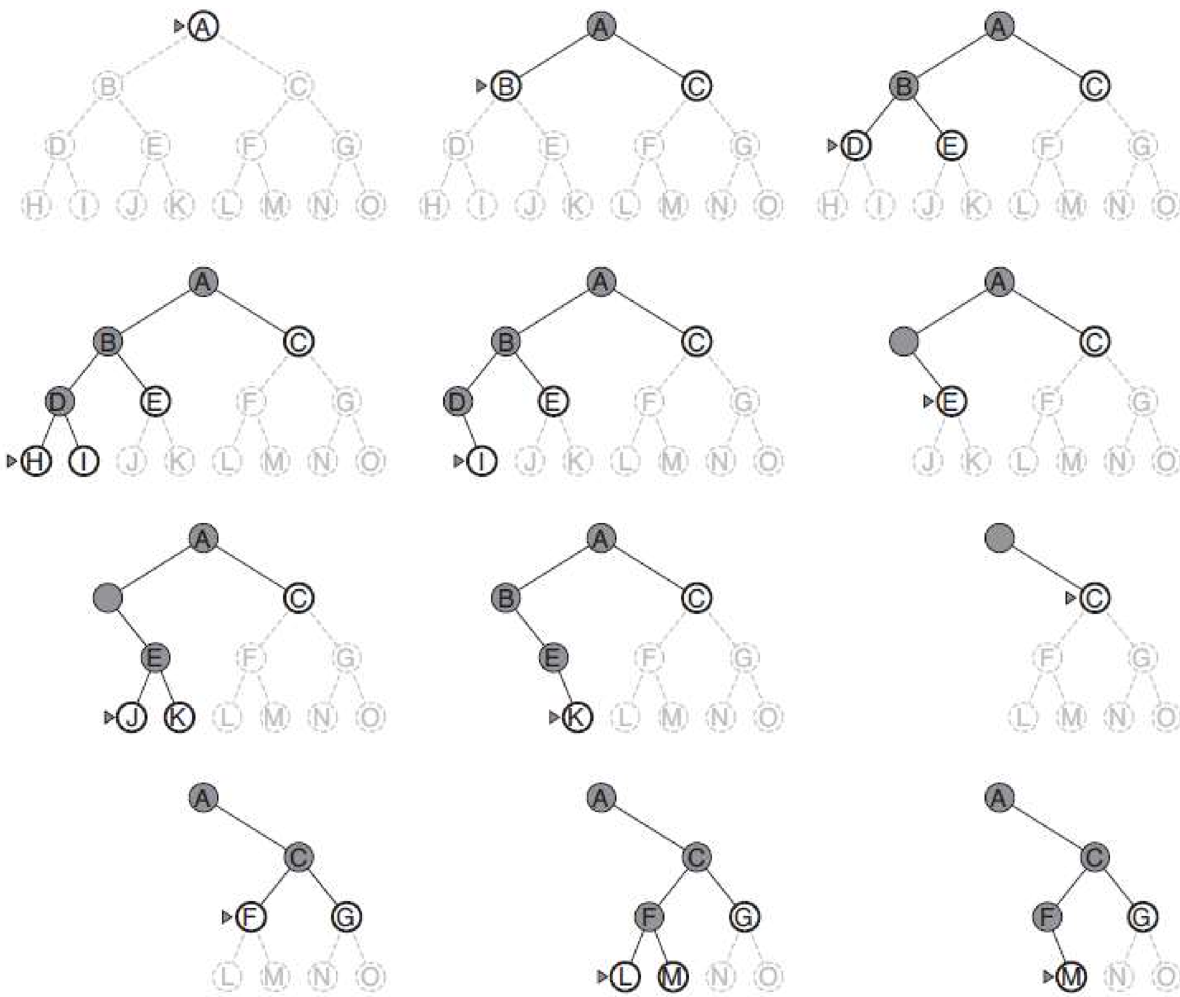


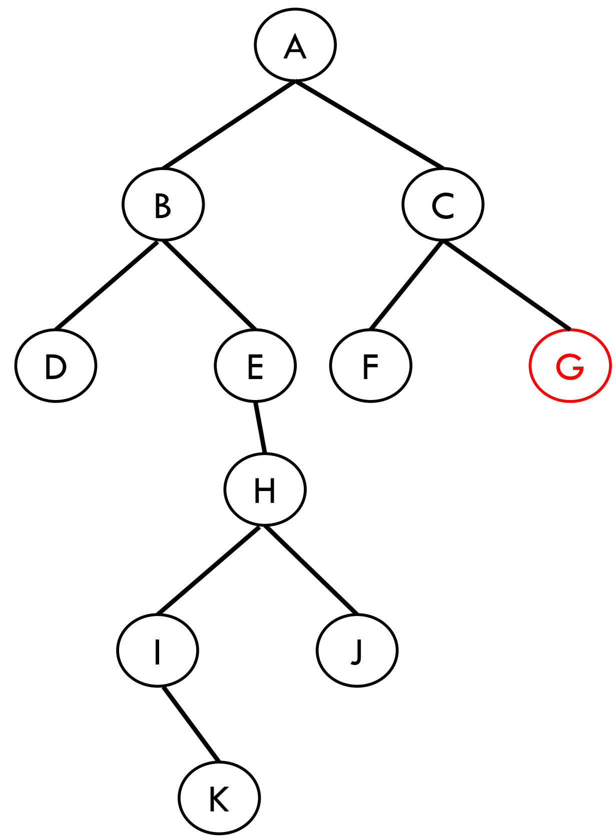
Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

COMPLETENESS?

Goal, G is at depth, d=2

Max depth of tree, m = 5

DFS is complete ONLY if m is finite



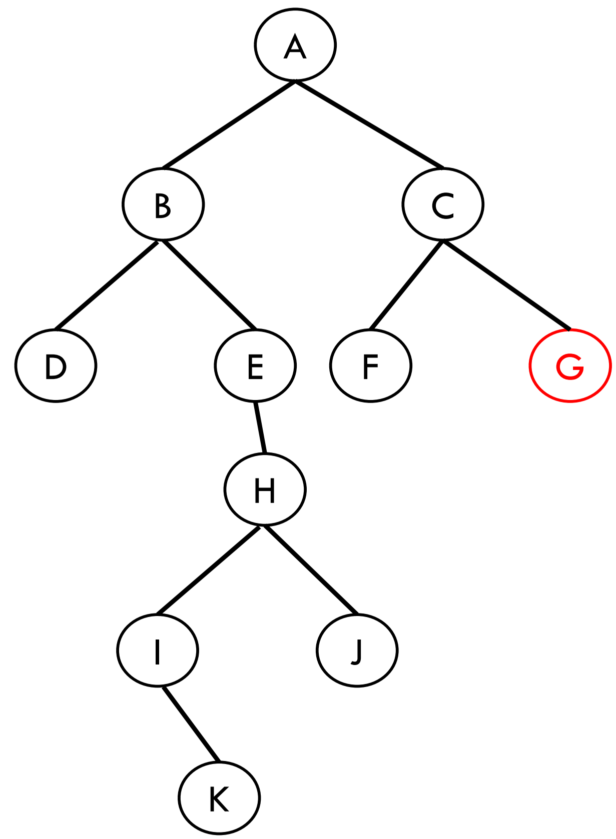
TIME COMPLEXITY

Number of nodes visited? $O(b^m)$

For BFS, remember it was $O(b^d)$

DFS is terrible if *m* is much larger than *d*

- but if solutions are dense (goal is at K instead of G), DFS is faster than BFS.



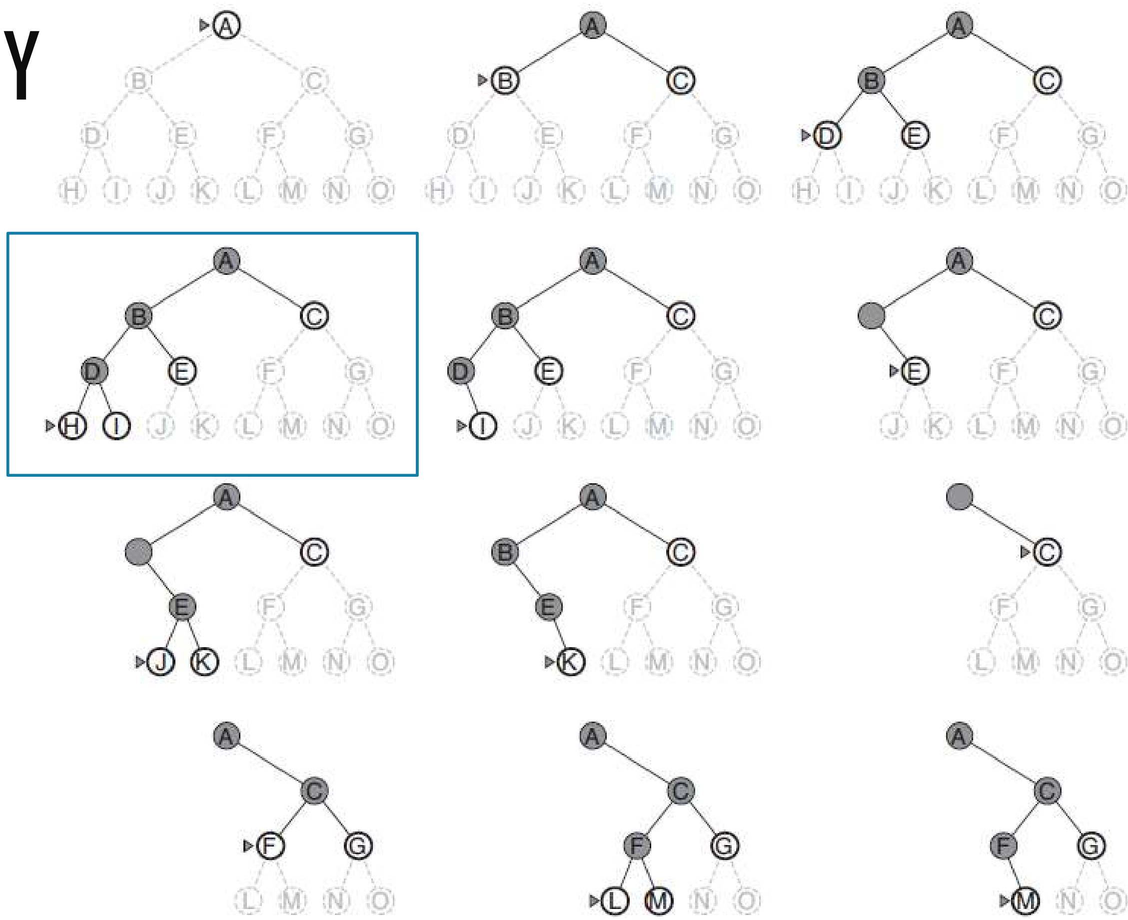
SPACE COMPLEXITY

O*(bm)*, i.e., linear space!

(we only need to remember a single path + expanded unexplored nodes)

BFS had O(b^d) i.e., exponential space requirement

O(bm)? E.g. m=3, b=2 (we need to save max of 3*2=6 nodes in memory)



OPTIMAL?

Remember BFS was optimal if each step has same cost.

Even if this is the case, DFS is not guaranteed to find the optimum path to the goal.

For the graph on the right, BFS will return optimal path but DFS won't (assuming each step has same cost)