祝恩

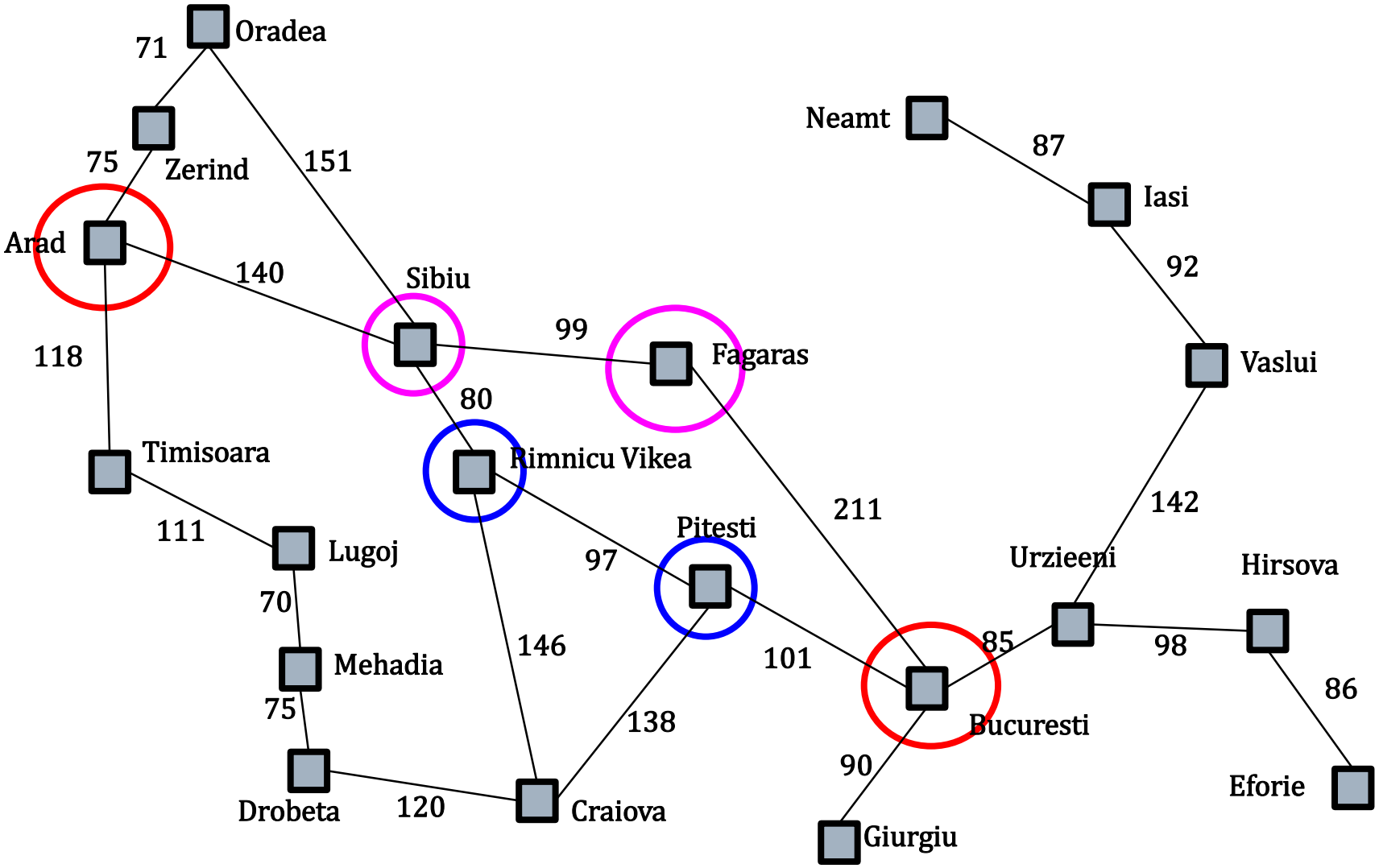
搜索

(搜索Agent,一种基于目标的Agent)

- 搜索问题的形式化
- 树搜索，图搜索，及算法评估
- 策略：宽度优先，一致代价，深度优先，深度受限，迭代加深，双向搜索

搜索问题的形式化

Romania


Oradea
71
Neamt
87
Zerind
75
151
Iasi
Arad
140
92
Sibiu
99
Fagaras
Vaslui
118
80
Timisoara
Rimnicu Vikea
211
142
111
Pitesti
Lugoj
97
Urzieeni
Hirsova
70
146
101
85
98
Mehadia
86
75
138
Bucuresti
90
Eforie
Drobeta
120
Craiova
Giurgiu

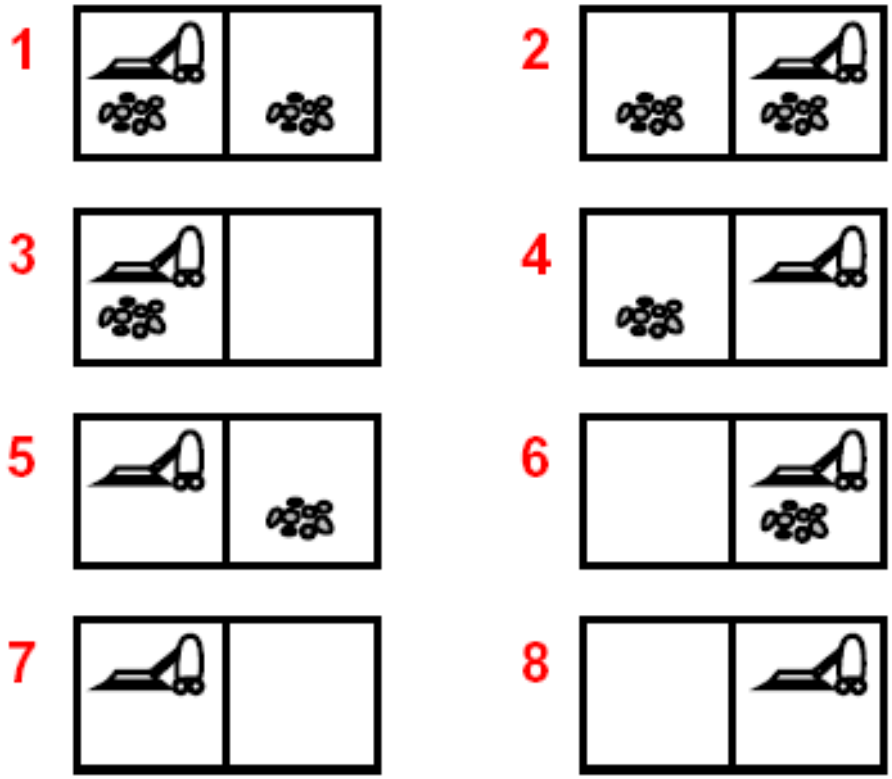
形式化一个搜索问题

well-defined problems and solutions

- a problem can be defined formally by five components:
  - initial state — In(Arad)
  - actions — ACTIONS(s)
  - transition model — RESULT(s,a)
  - goal test — {In(Bucuresti)}
  - path cost — length in kilometers
- a solution to a problem is an action sequence that leads from the initial state to a goal state.

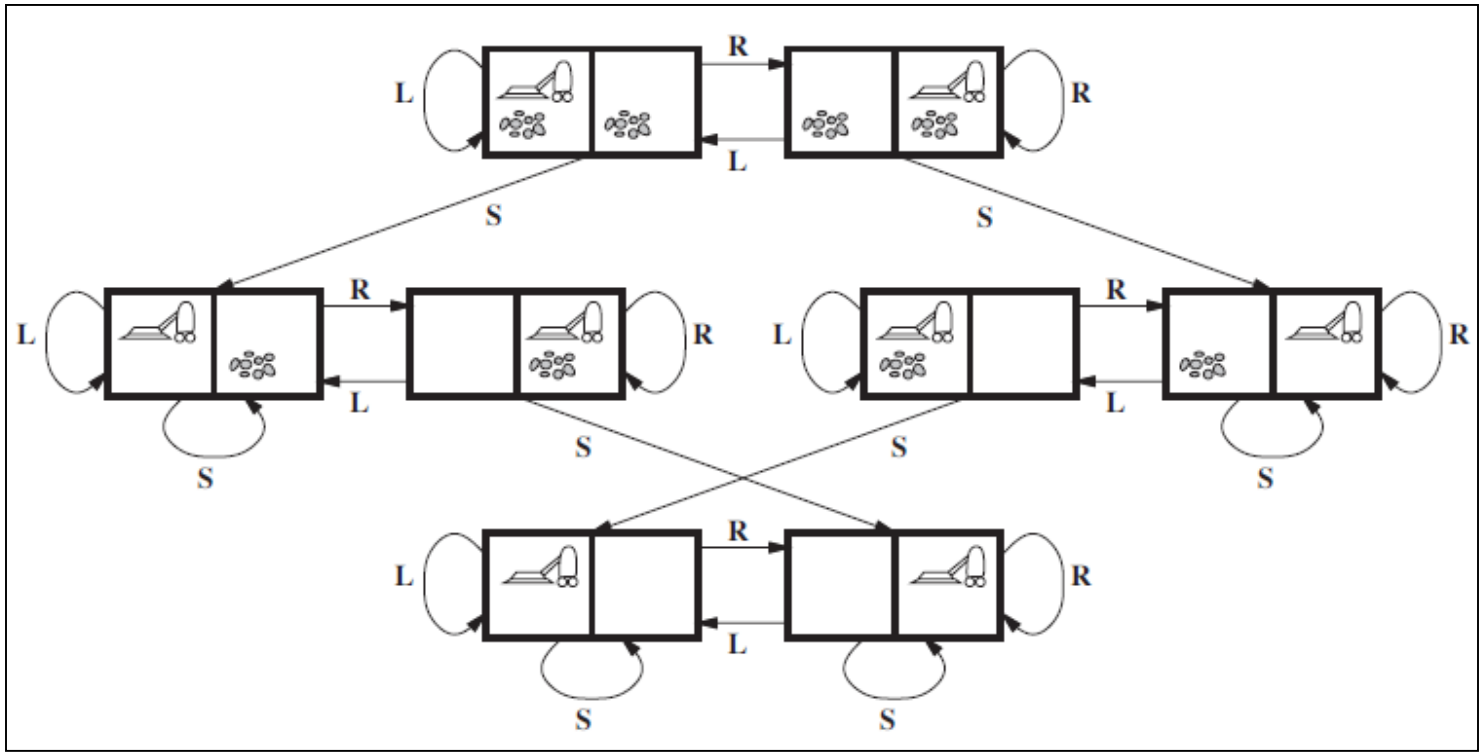
Example: Vacuum World {3.2.1}

- States
- Initial State
  - 5
- Actions
  - Right, Left, Suck
- transition model
- Goal
  - {7,8}
  - Goal Test
- Path Cost
- Solution
  - [Right, Suck]



Represent problem as a graph. {3.2.1}

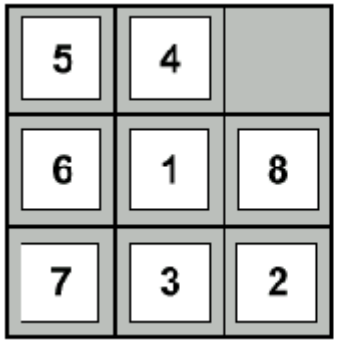
- Represent problem as a graph
  - Nodes are states
  - Arcs are actions



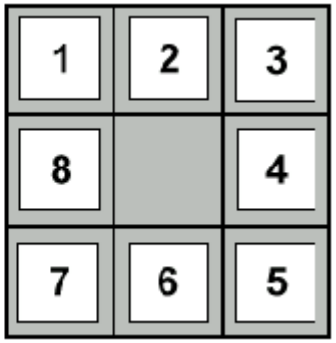
树搜索、图搜索、及算法评估

Why searching?

- Why search algorithms?
  - 8-puzzle has 362,880 states
  - 15-puzzle has 20 922 789 888 000 states
  - 24-puzzle has 620 448 401 733 239 439 360 000 states
- So, we need a principled way to look for a solution in these huge search spaces…



start state



goal state

15-Puzzle {3.3+}*

- Sam Loyd (1878) offered $1,000 of his own money to the first person who would solve the following problem:

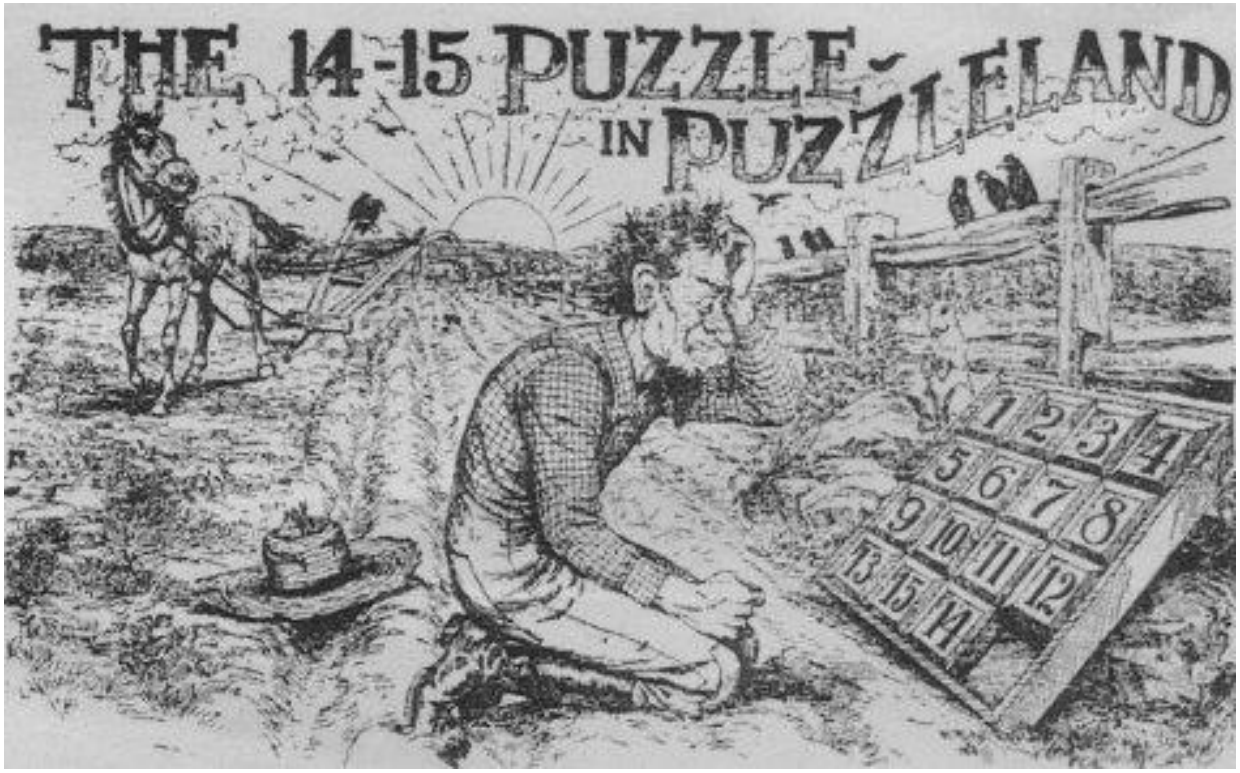
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

?

1	2	3	4
5	6	7	8
9	10	11	12
13	15	14	

15-Puzzle {3.3+}*

- But no one ever won the prize !!

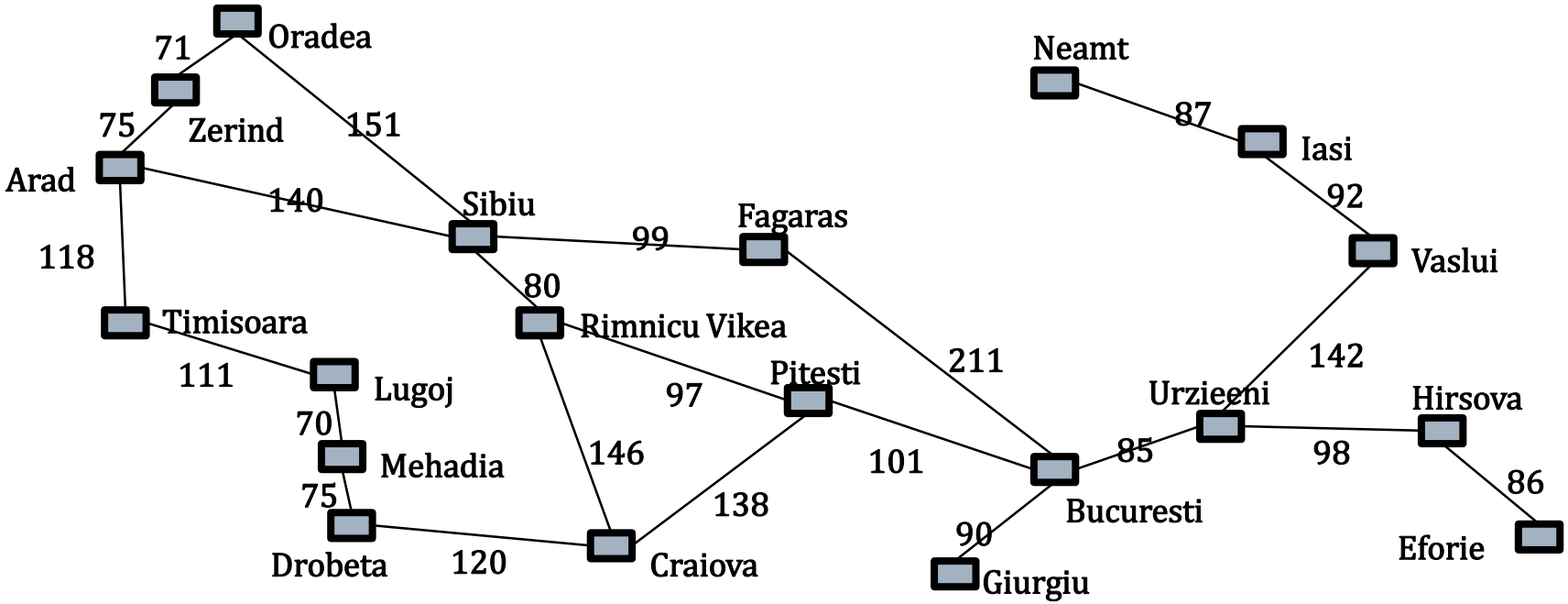


☐ How to search?

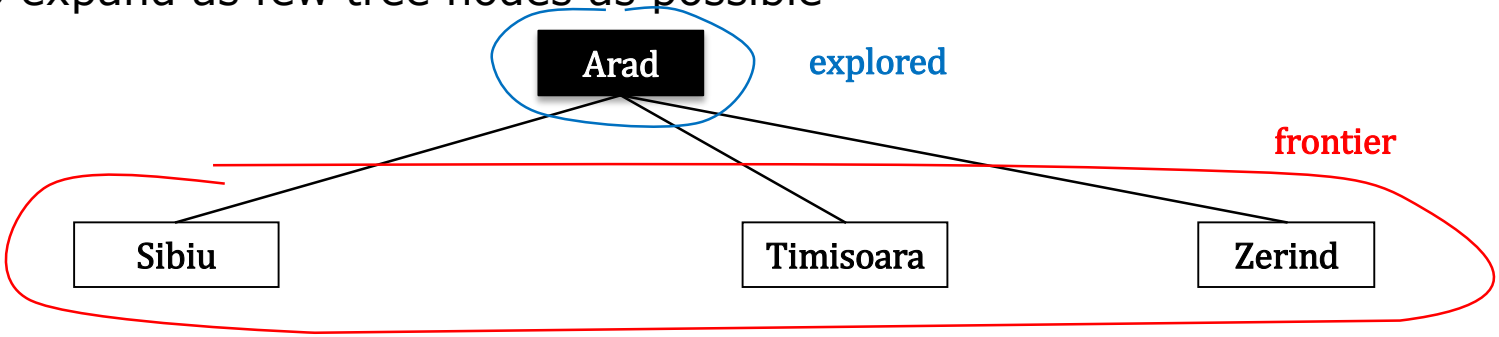
starting from the initial node which represents the initial state, repeatedly generate nodes in terms of available actions of a state, until a node representing the goal state is generated. The nodes are connected by arcs which represent actions.

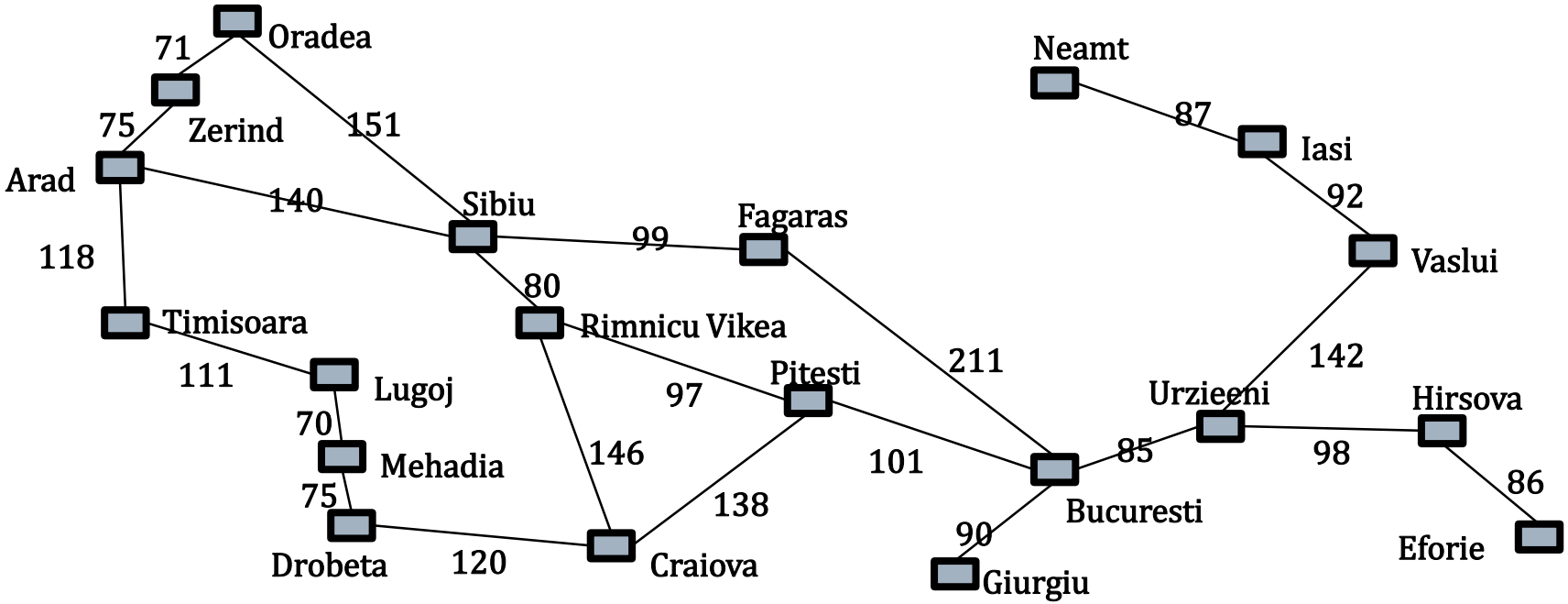
- Search
  - Expand out possible plans
  - Try to expand as few tree nodes as possible

Arad frontier

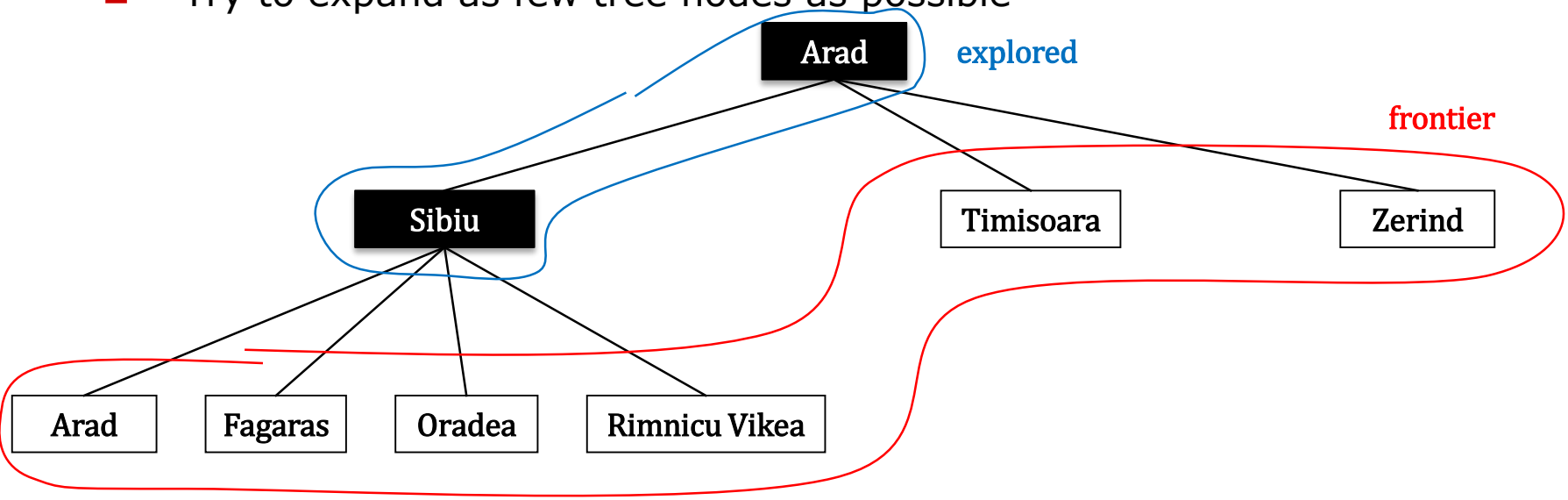


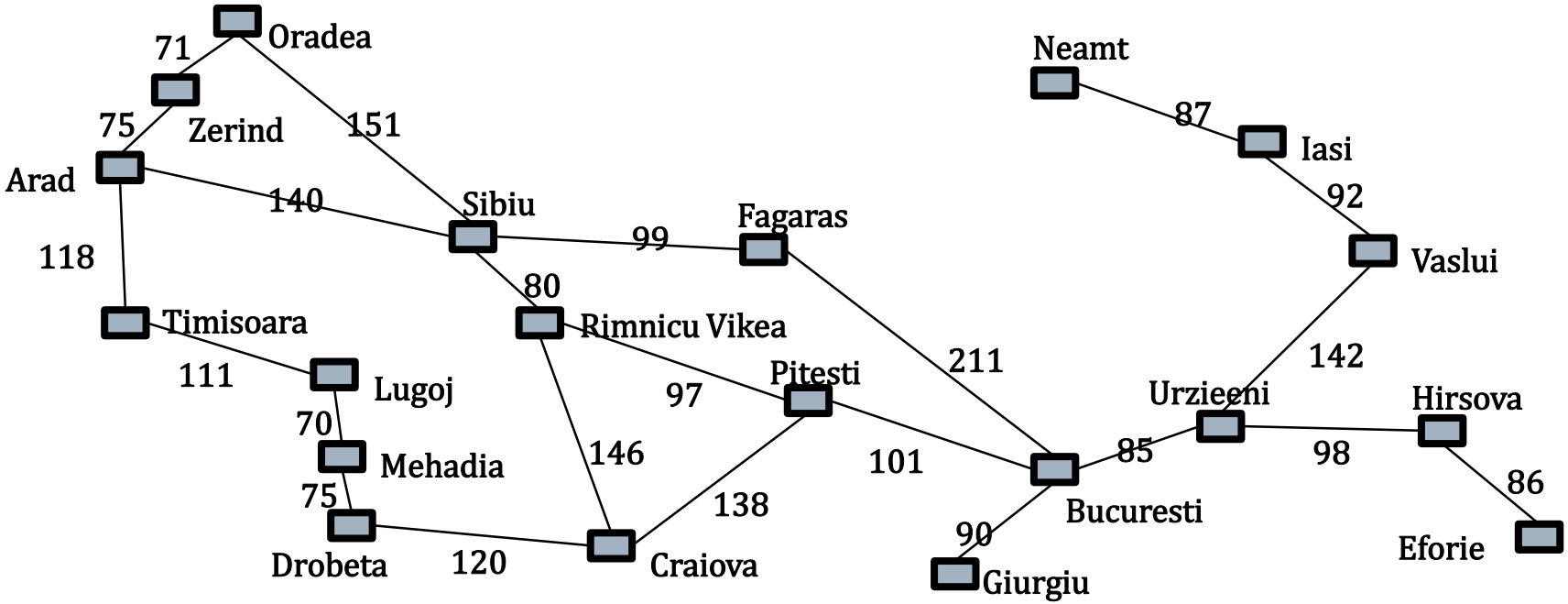
- Search
  - Expand out possible plans
  - Try to expand as few tree nodes as possible



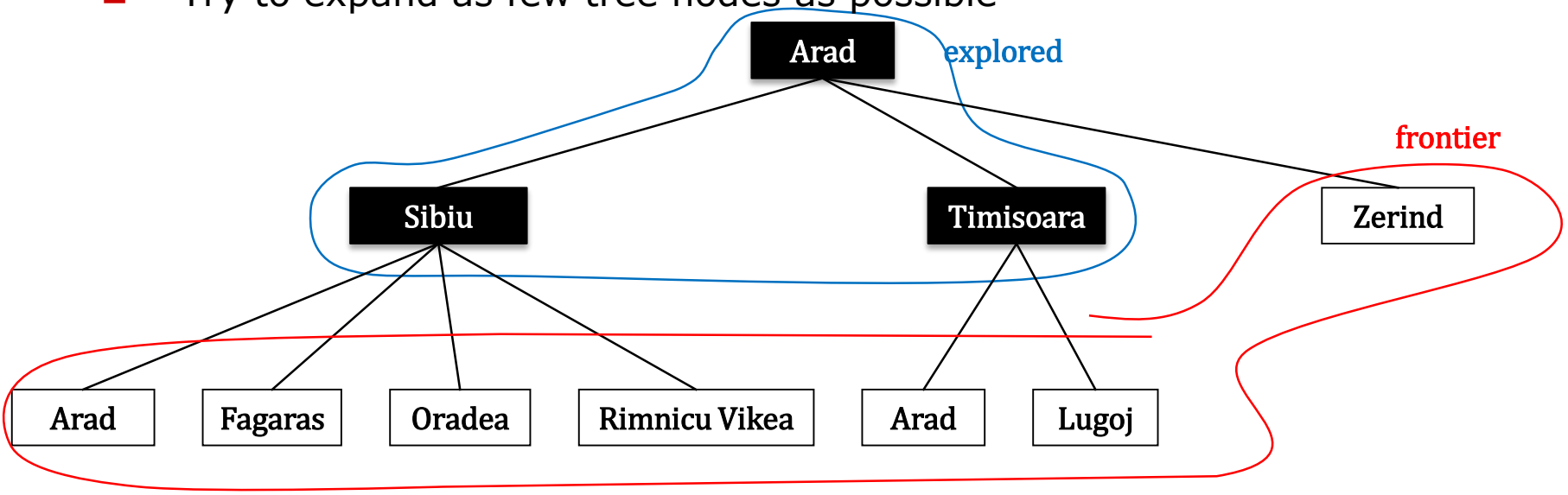


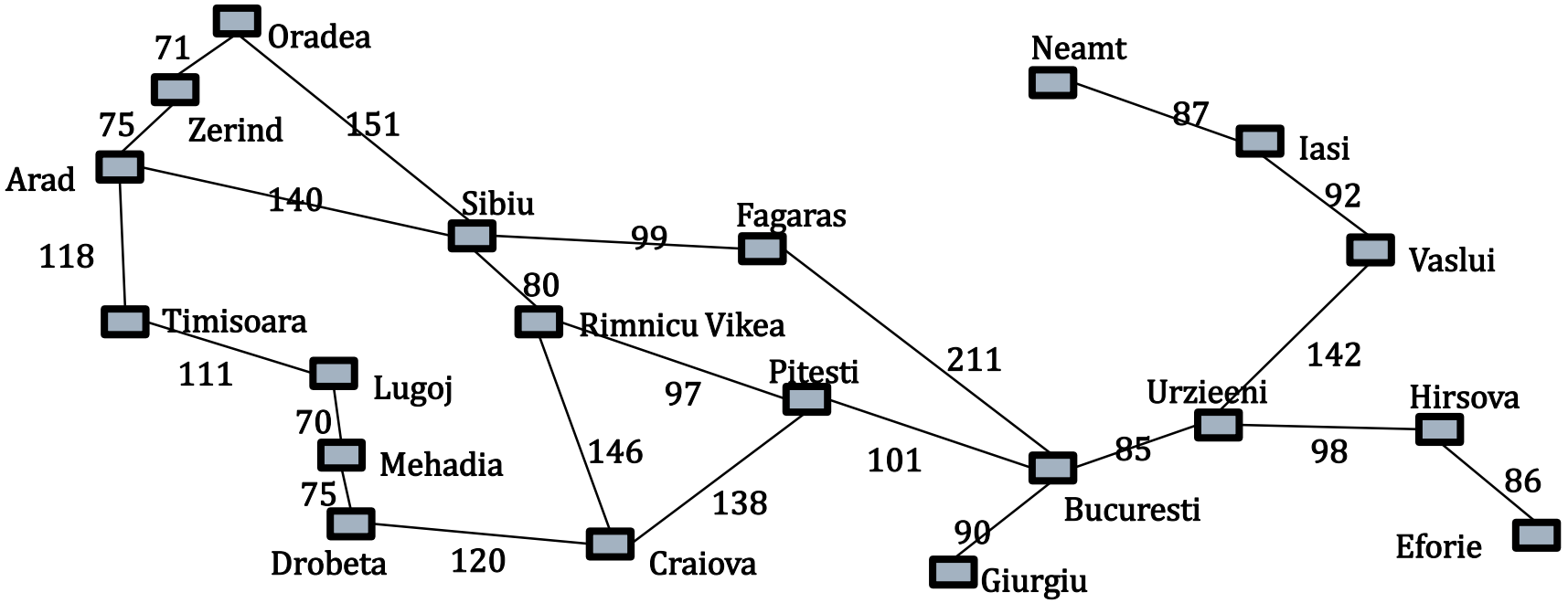
- Search
  - Expand out possible plans
  - Try to expand as few tree nodes as possible



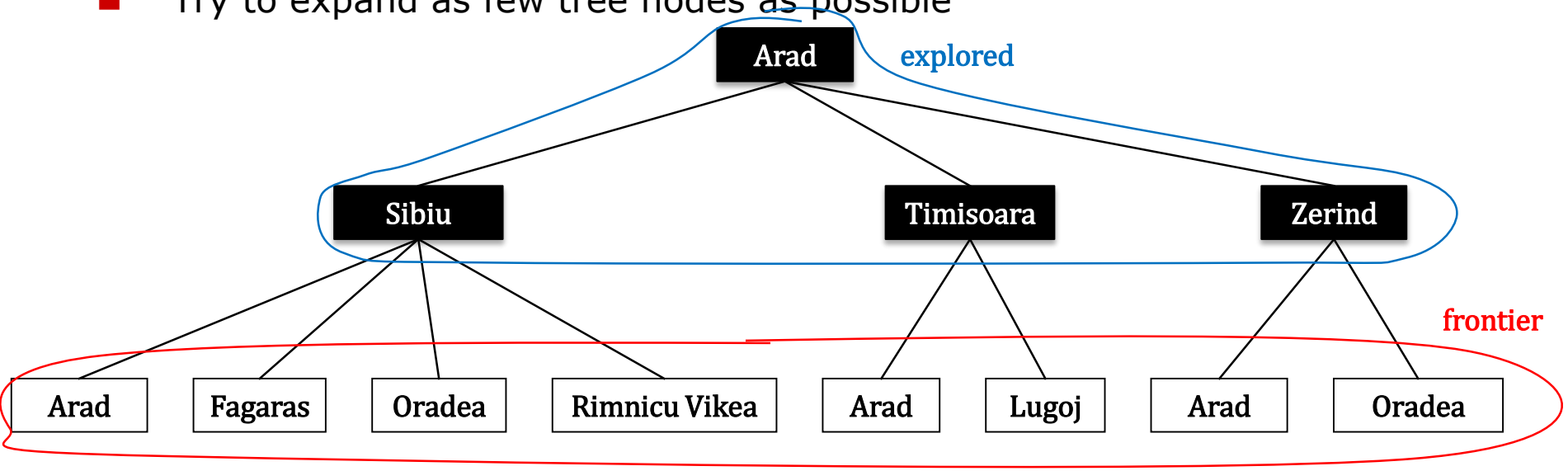


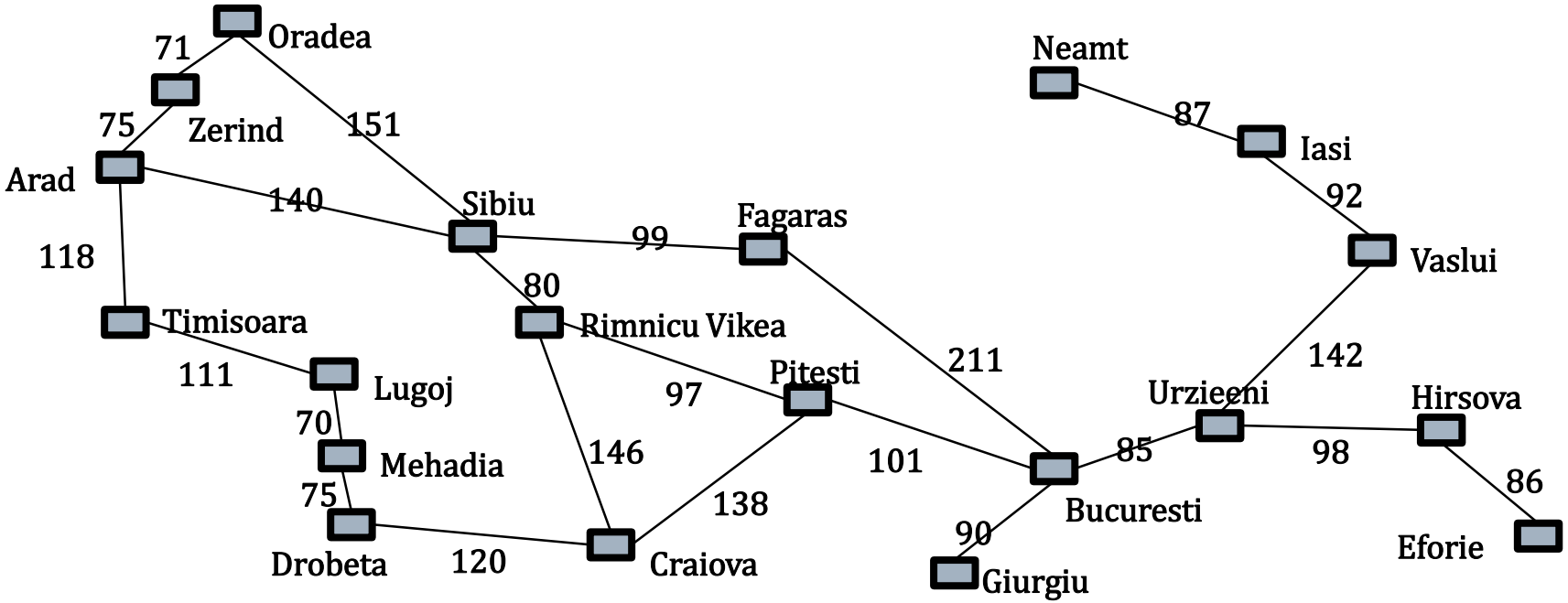
- Search
  - Expand out possible plans
  - Try to expand as few tree nodes as possible





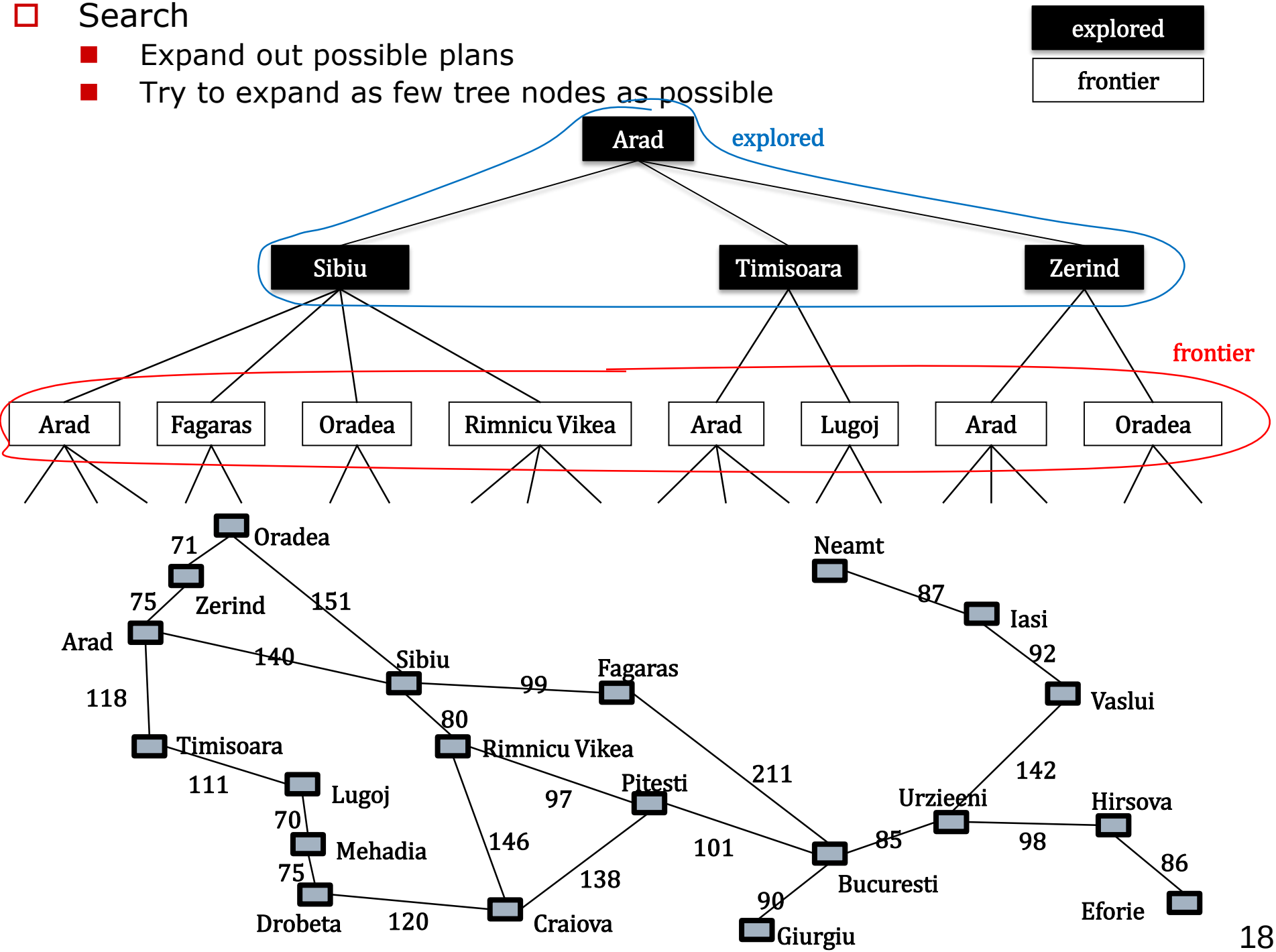
- Search
  - Expand out possible plans
  - Try to expand as few tree nodes as possible





Search

- Expand out possible plans
- Try to expand as few tree nodes as possible

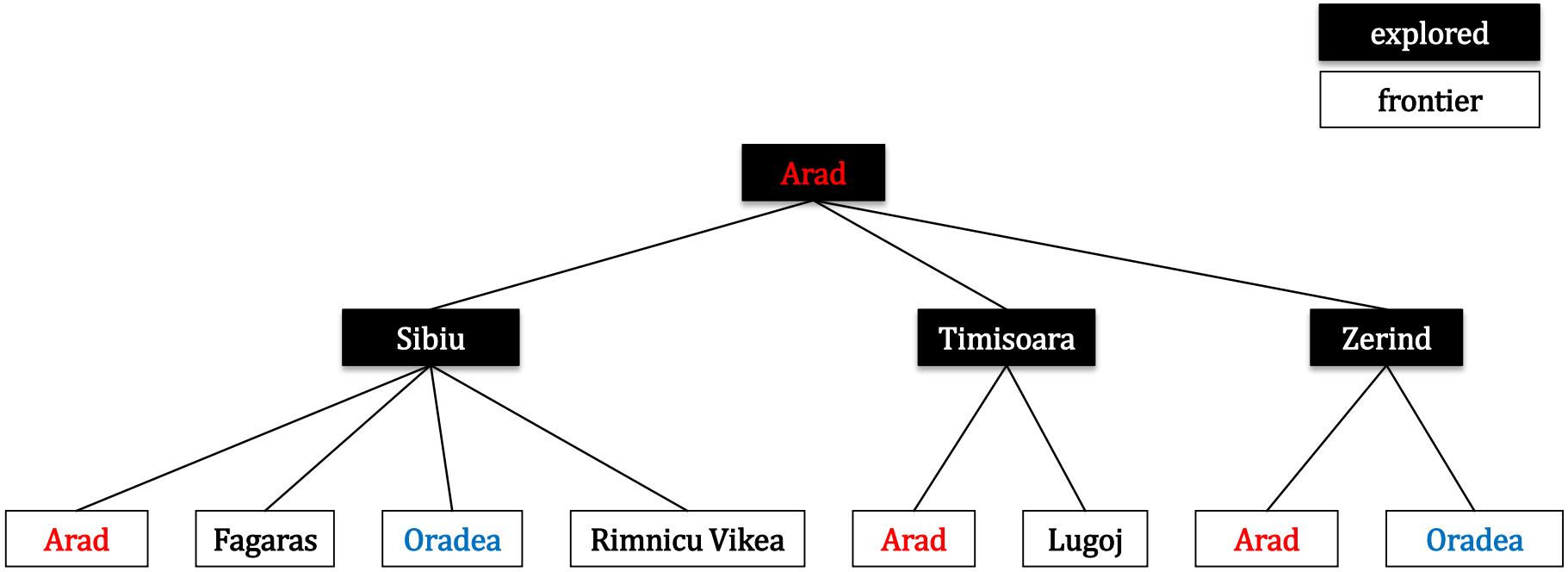


Tree Search {3.3}

- Basic solution method for graph problems
  - Offline simulated exploration of state space
  - Searching a model of the space, not the real world

function Tree-Search(problem) **returns** a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

after it is removed from the frontier



- How to avoid revisiting a state?

Graph search {3.3}

function Graph-Search(problem) **returns** a solution, or failure

initialize the frontier using the initial state of problem

initialize the explored set to be empty

loop do

if the frontier is empty **then return** failure

choose a leaf node and remove it from the frontier

if the node contains a goal state **then return** the corresponding solution

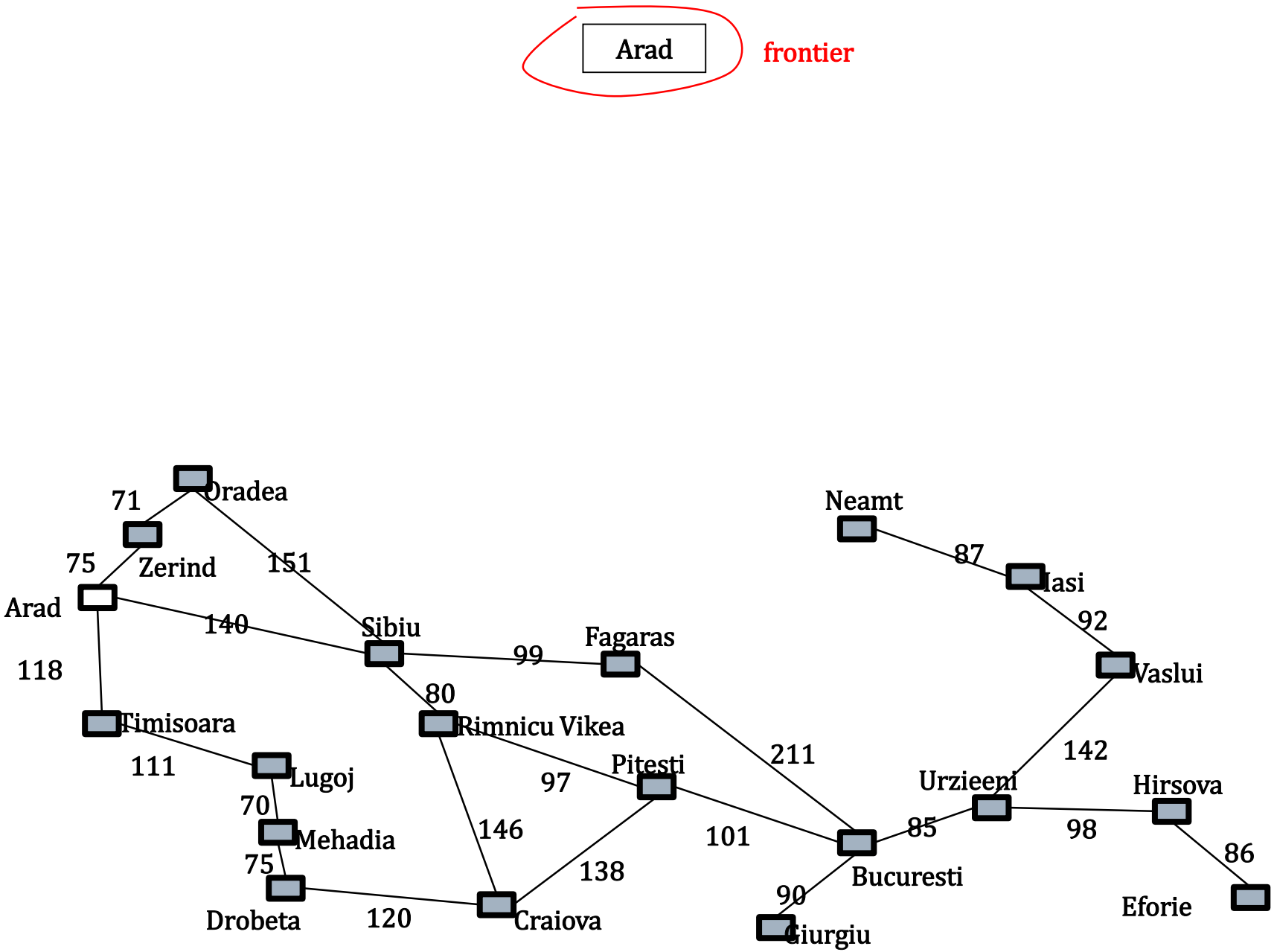
add the node to the explored set

expand the chosen node, adding the resulting nodes to the frontier

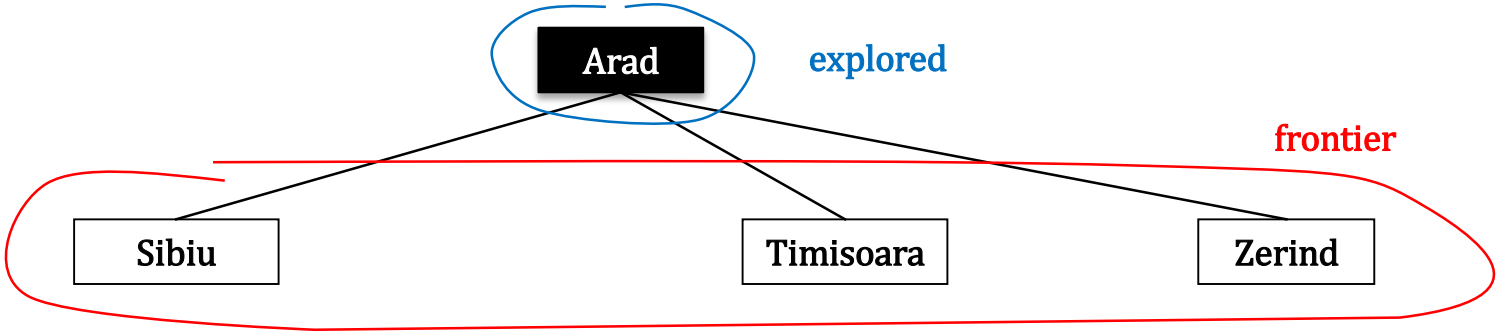
only if not in the frontier or explored set

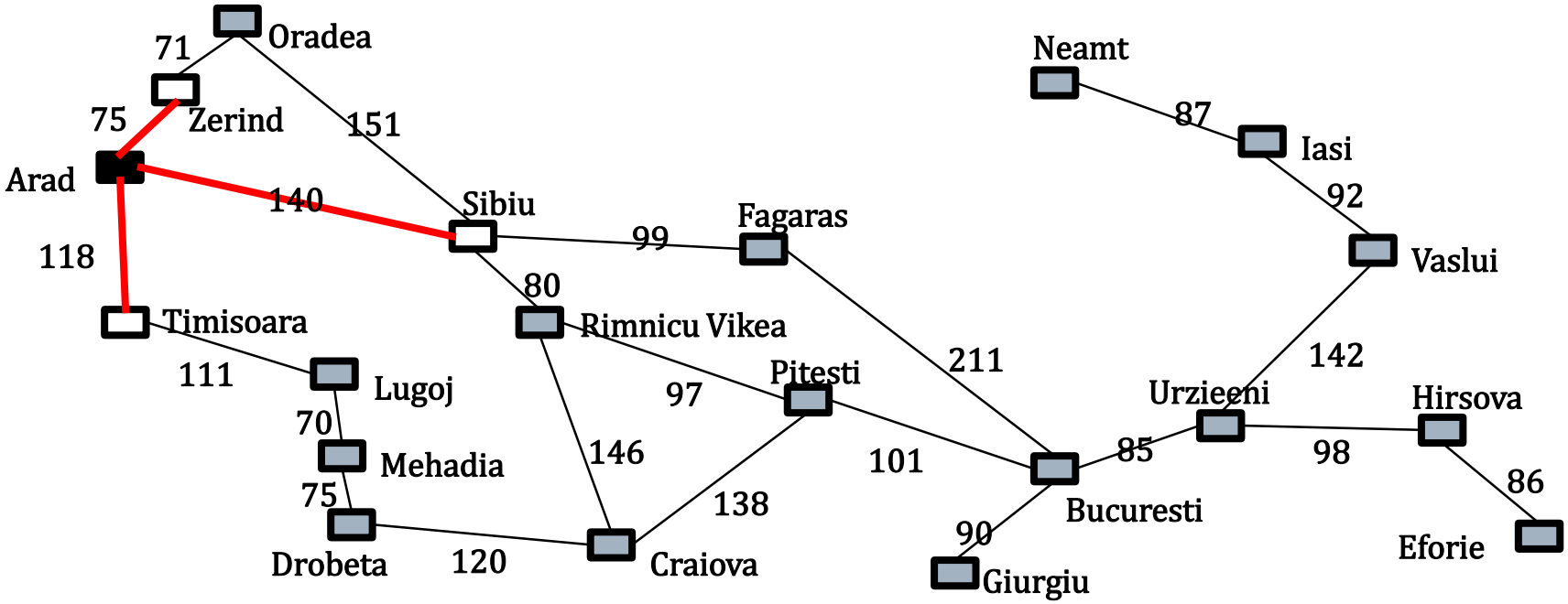
after it is removed from the frontier

Graph Search

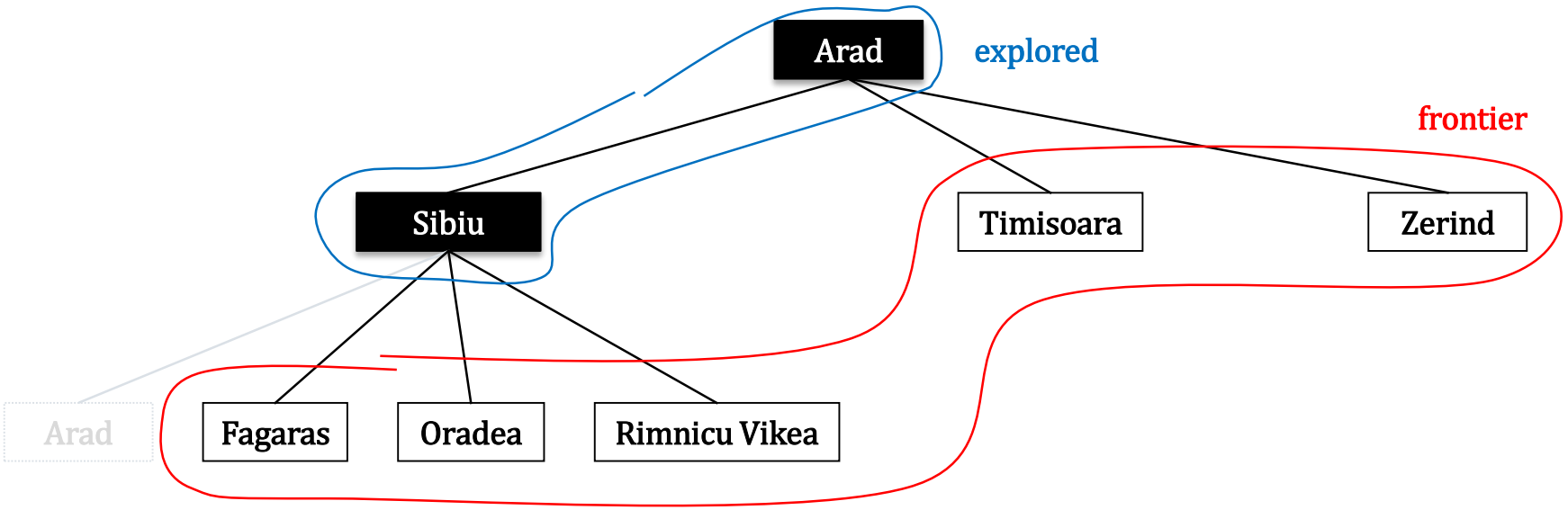


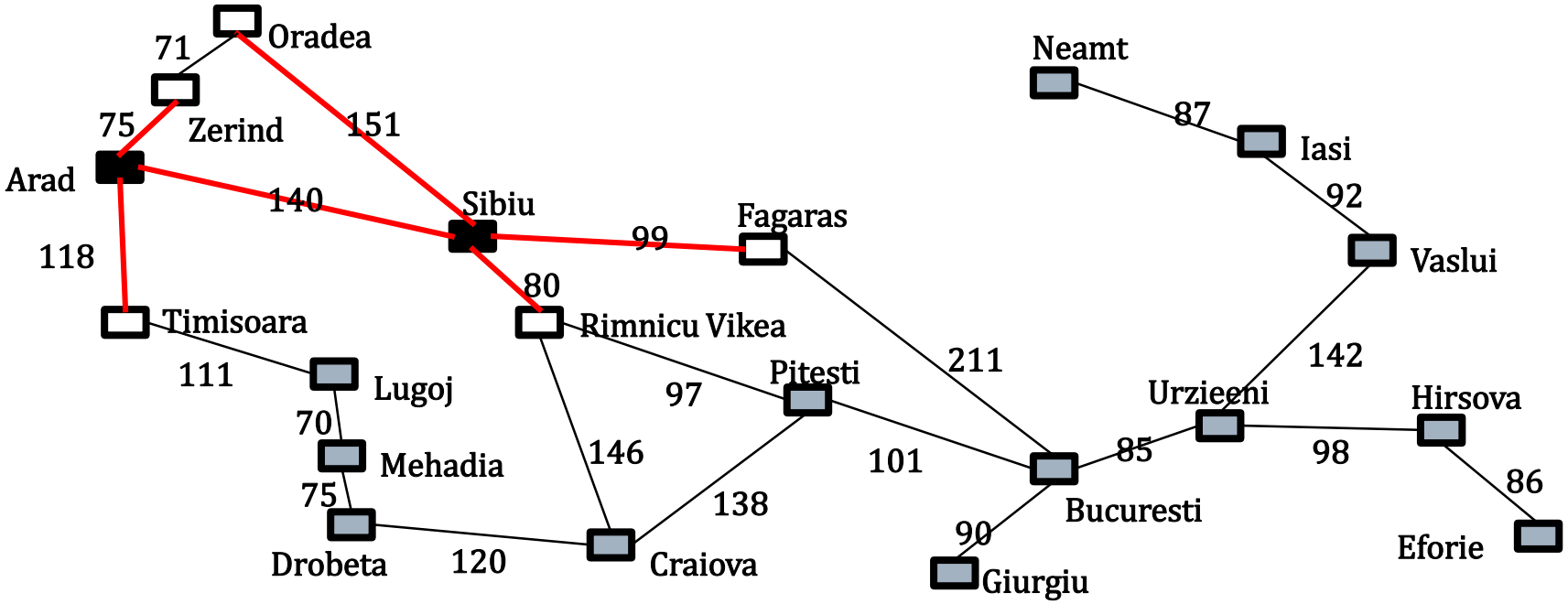
Graph Search



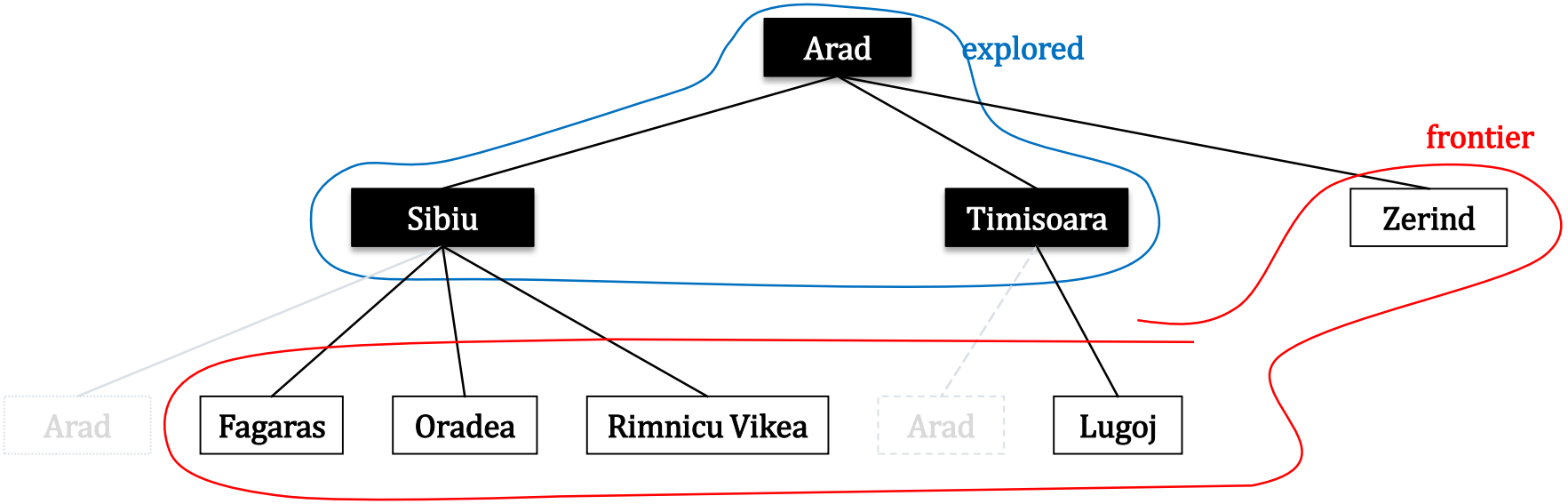


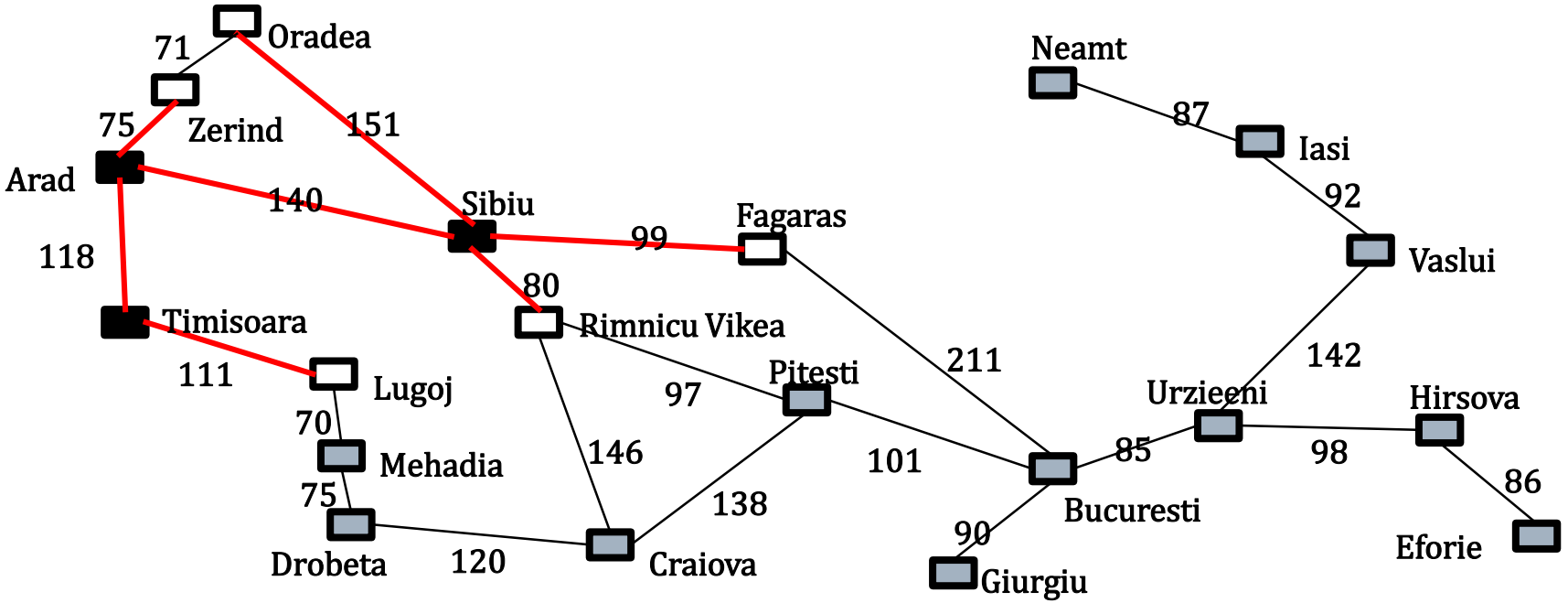
Graph Search



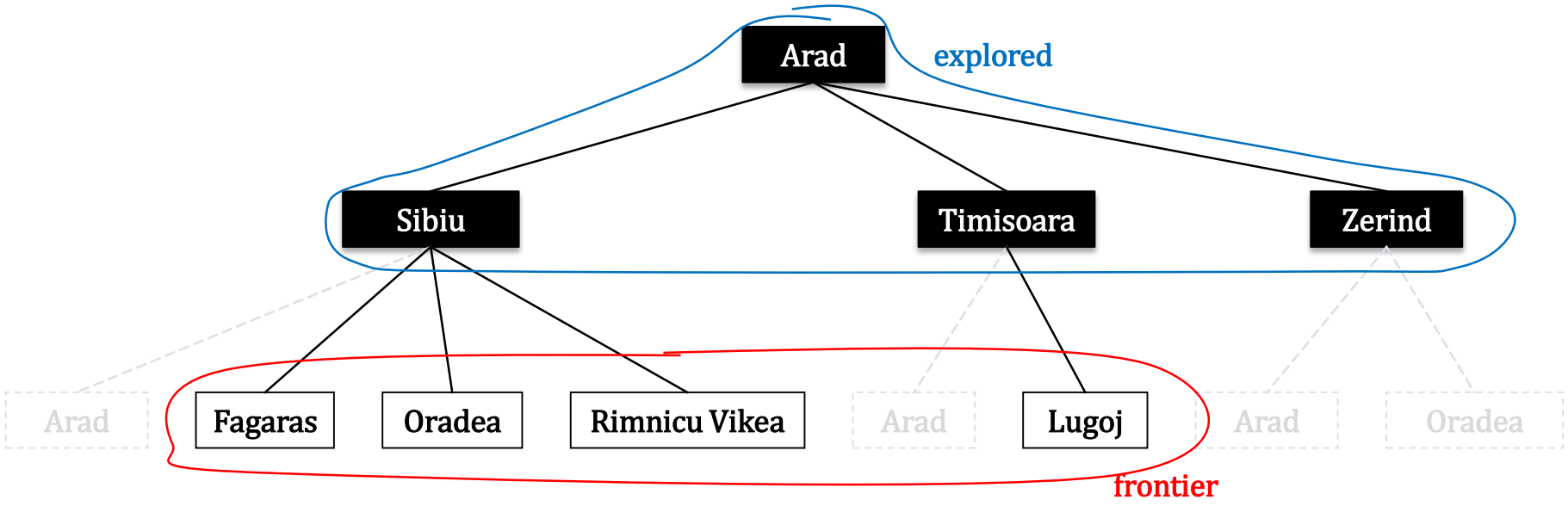


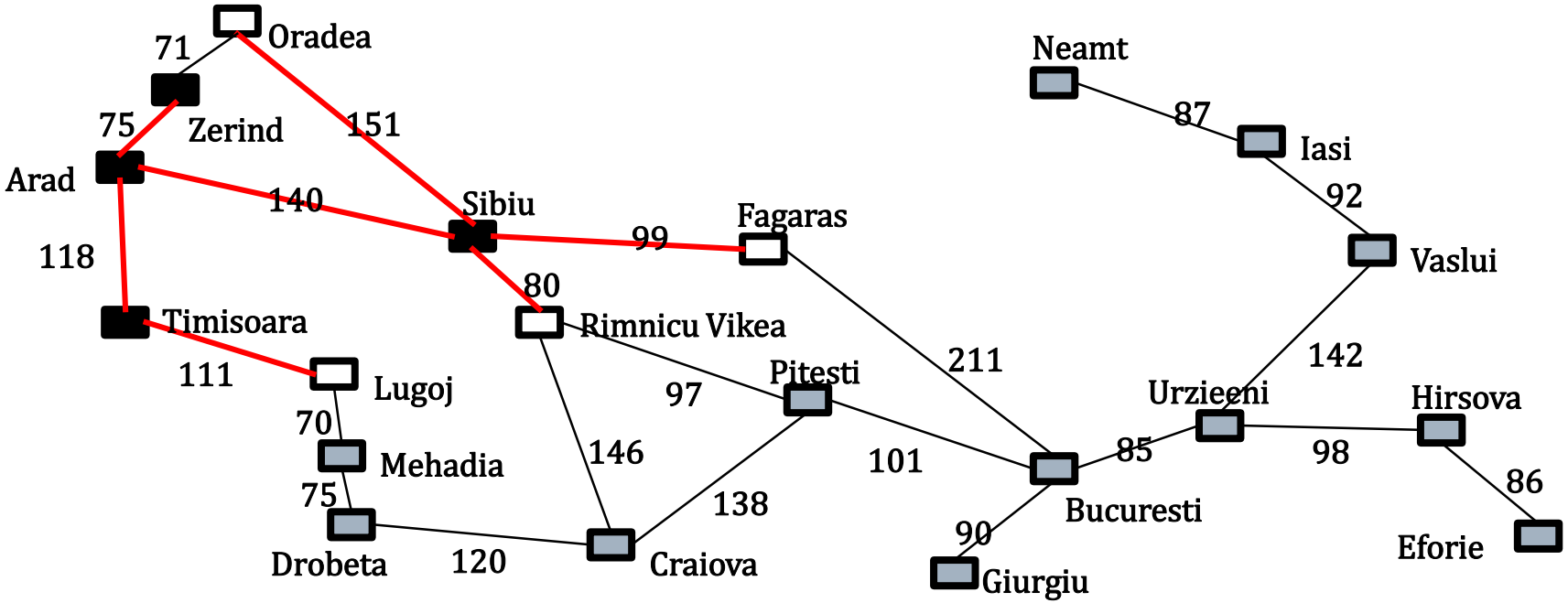
Graph Search

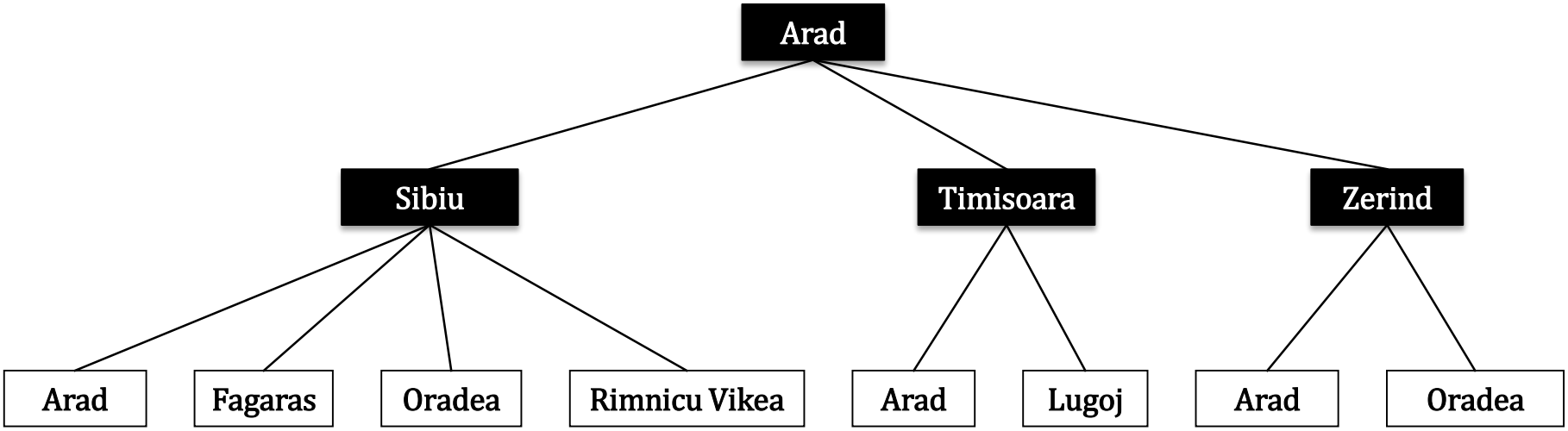




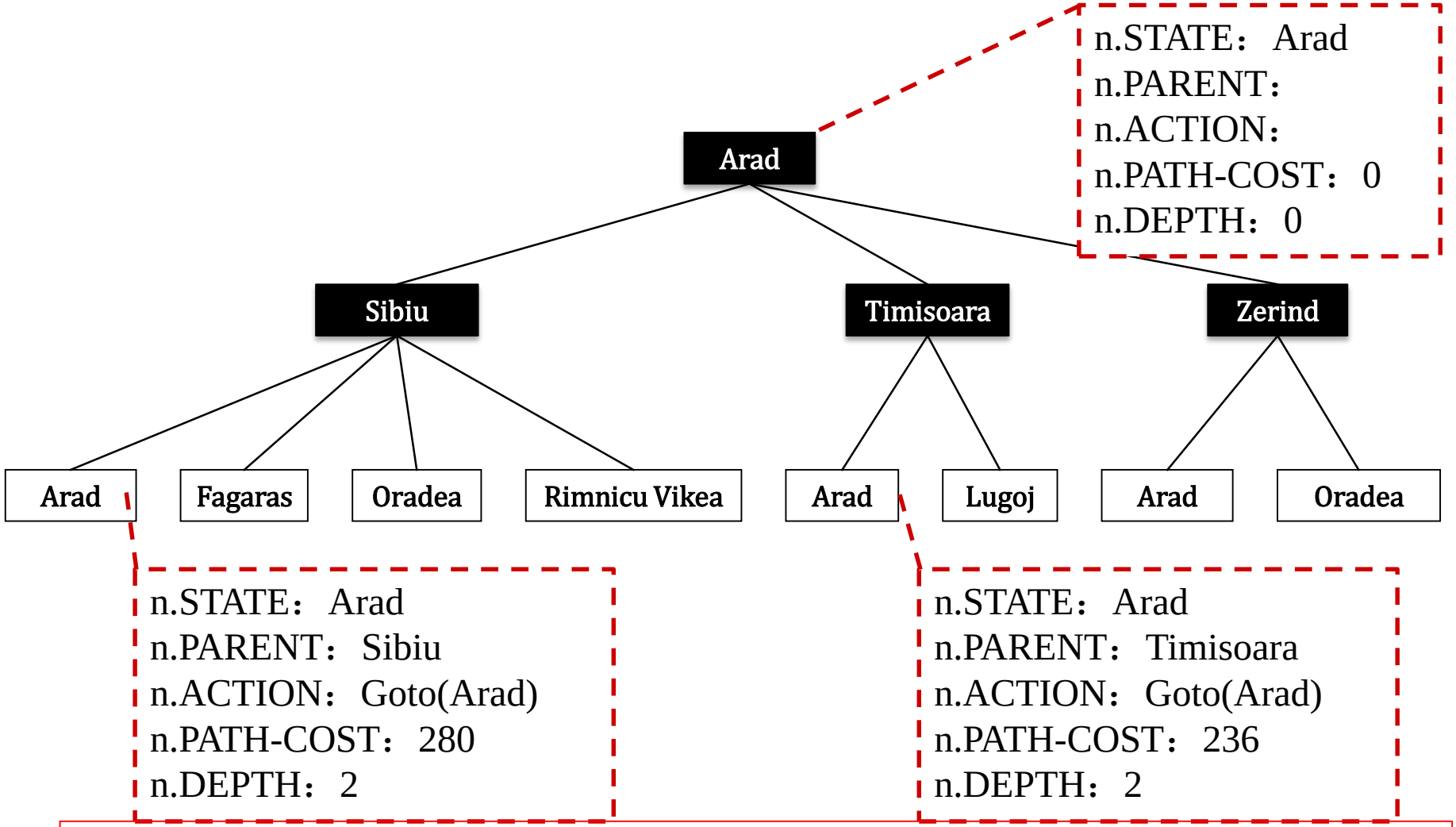
Graph Search







- What's the difference between nodes and states?



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*)

completeness, optimality, complexity{3.3.2}

- Completeness(完备性)
  - A search algorithm is complete if it finds a solution whenever one exists
- Optimality(最优性)
  - A search algorithm is optimal if it returns a minimum-cost path whenever a solution exists
- Complexity(复杂性)
  - It measures the time and amount of memory required by the algorithm

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

Iterative deepening search

Bidirectional search

（盲目搜索）策略

What determines the search strategy? {3.4}

- FRONTIER
  - is the set of all search nodes that haven't been expanded yet. The search algorithm repeatedly chooses and removes a node from the frontier, expand it and insert the resulting nodes to the frontier
  - REMOVE(FRONTIER)
  - INSERT(node, FRONTIER)
- The ordering of the nodes in FRONTIER defines the search strategy

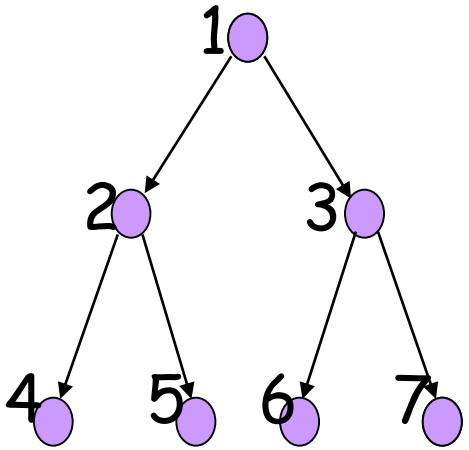
Breadth-First Strategy {3.4.1}

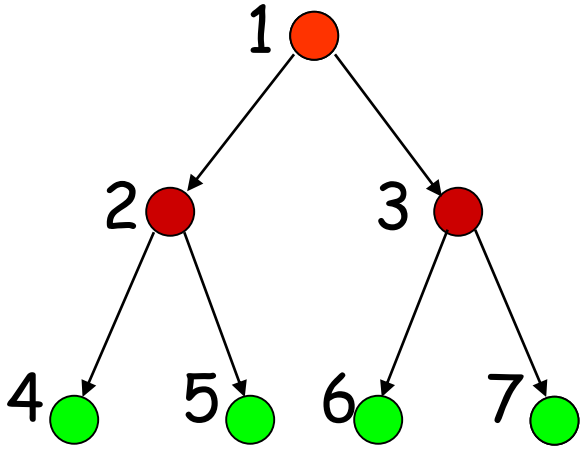
```
function Breadth-First-Search(problem) returns a solution, or failure
    node ← a node with State=problem.Initial-State, Path-Cost=0
    if problem.Goal-Test(node.State) then return Solution(node)
    frontier←{node}
    explored←an empty set
    loop do
        if Empty?(frontier) then return failure
        node ← POP the shallowest node in frontier
        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
                if problem.Goal-Test(child.State) then return Solution(child)
                frontier ← Insert(child, frontier)
```

加入**frontier**之前，为什么？

Breadth-First Strategy {3.4.1}

- New nodes are inserted at the end of FRONTIER
- and hence the first node is the shallowest node.





FRONTIER = (1)

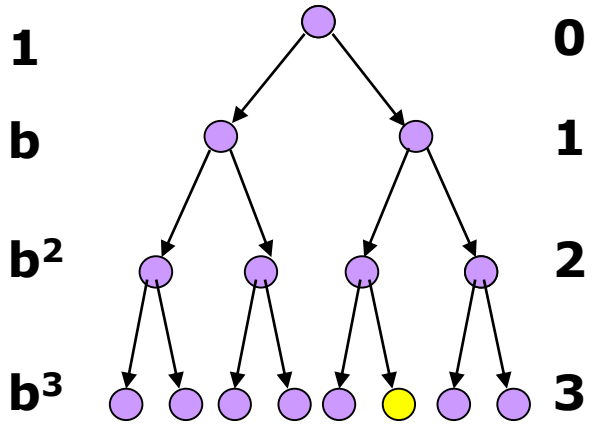
FRONTIER= (2, 3)

FRONTIER = (3, 4, 5)

FRONTIER = (4, 5, 6, 7)

complete? optimal? complexity {3.4.1}

- Branching factor ***b*** of the search tree
  - Maximum number of successors of any state
  - 假设分支因子b是有限的
- Depth ***d*** of the shallowest goal node in the search tree
  - Minimal length (≠ cost) of a path between the initial and a goal state



- Breadth-first search is
  - Complete if b is finite
  - Optimal if step cost is 1
- Time and space complexity is $1+b+\ldots+b^d=O(b^d)$

Breadth-First Strategy{3.4.1}

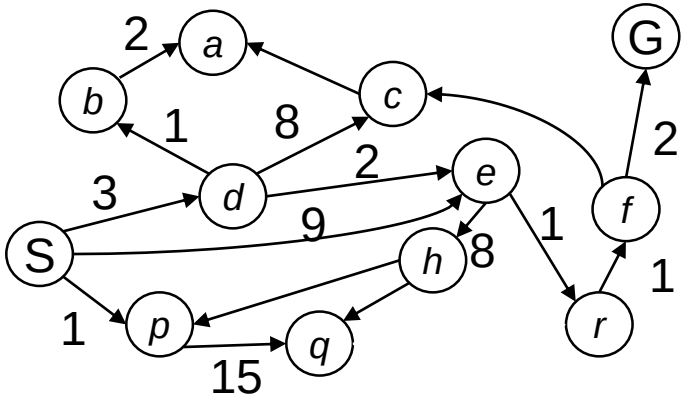
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

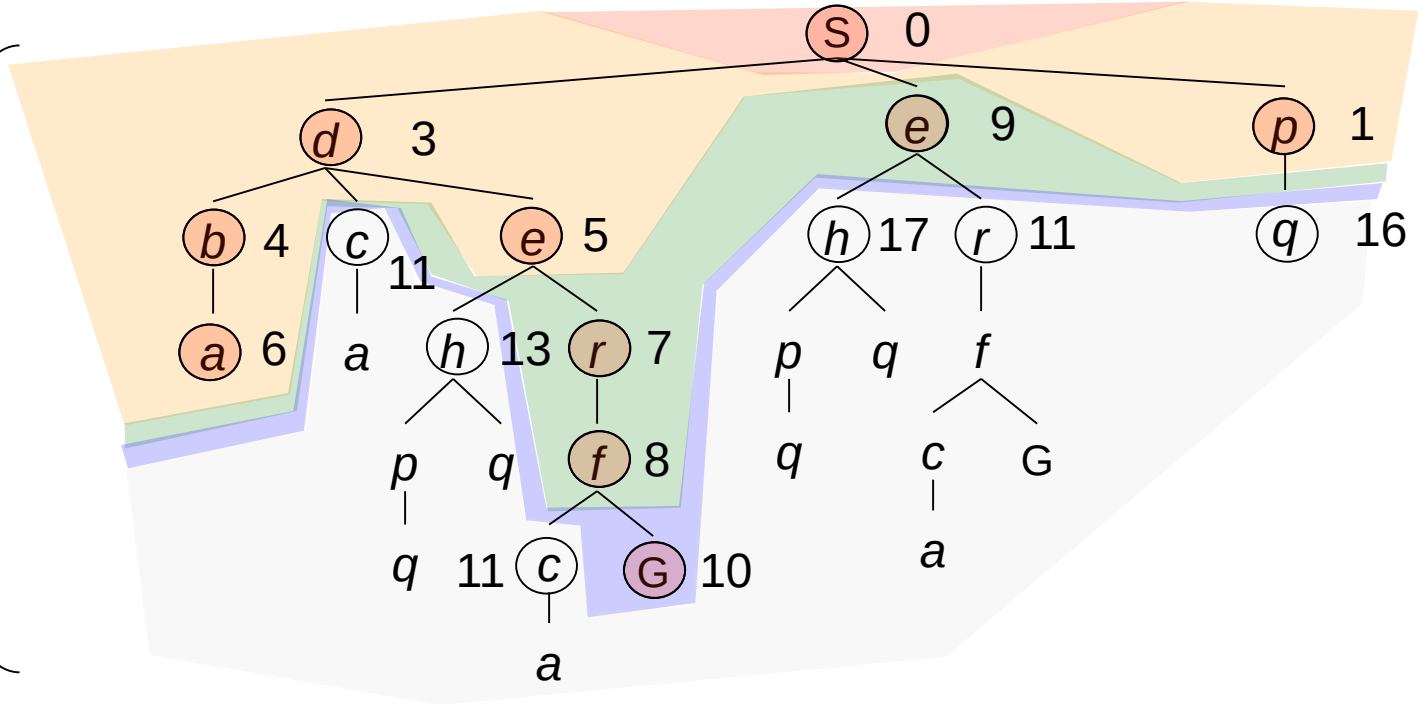
- Breadth-first search gets the shallowest solution!
- If we want to get the lowest-cost solution…

Uniform Cost Search {3.4.2}

- Expand the cheapest node
- Tree search version:

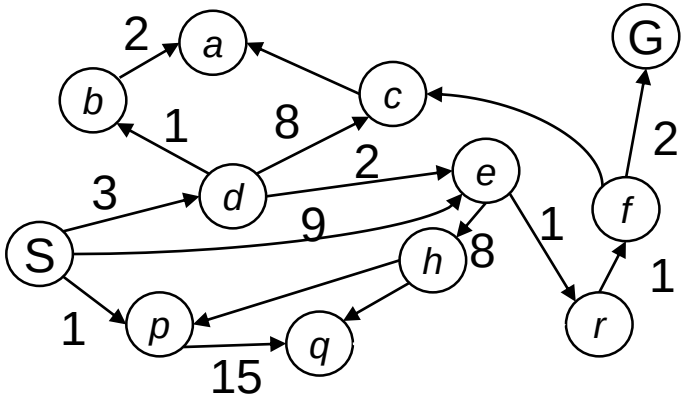


Cost contours

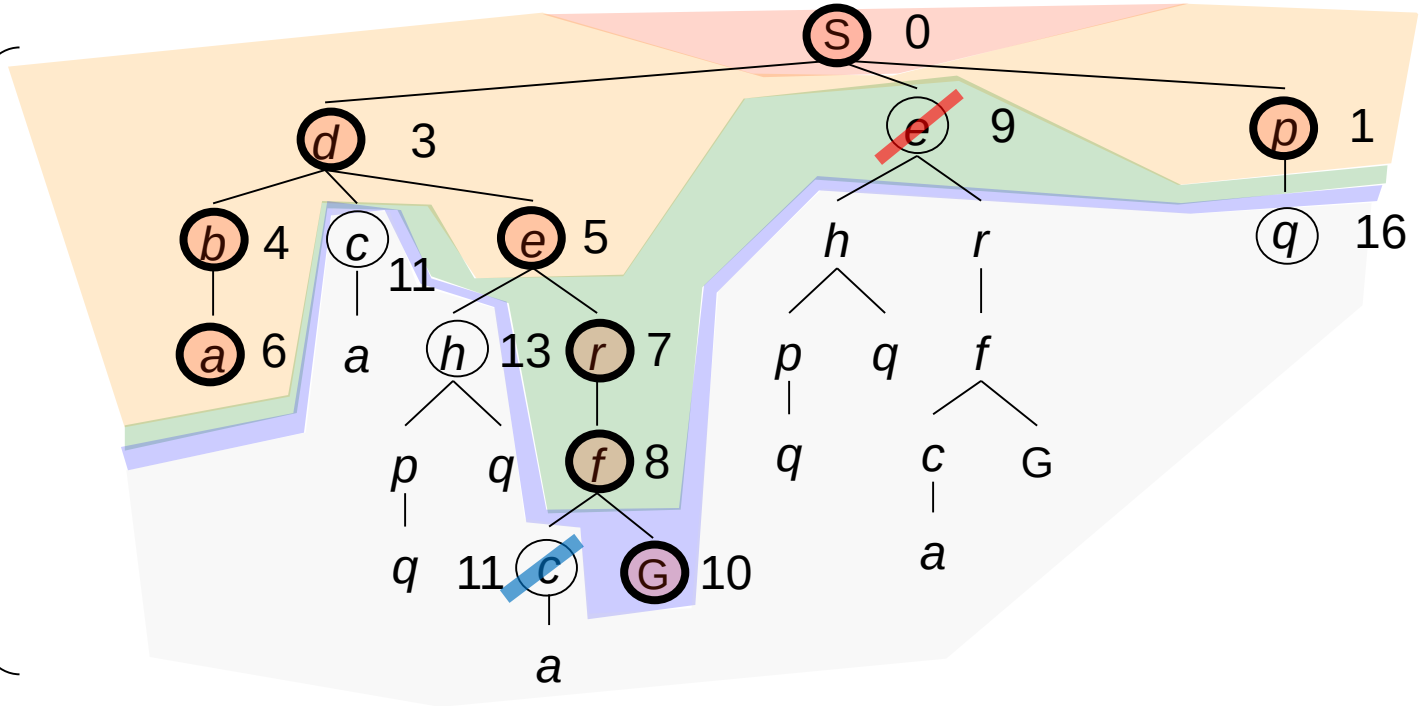


Uniform Cost Search {3.4.2}

- Expand the cheapest node
- Graph search version



Cost contours



Uniform Cost Search {3.4.2}

```
function Uniform-Cost-Search(problem) returns a solution, or failure
    node ← a node with State=problem.Initial-State, Path-Cost=0
    if problem.Goal-Test(node.State) then return Solution(node)
    frontier←{node}
    explored←an empty set
    loop do
        if Empty?(frontier) then return failure
        node ← POP the lowest-cost node in frontier
        if problem.Goal-Test(child.State) then return Solution(child)
        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
```

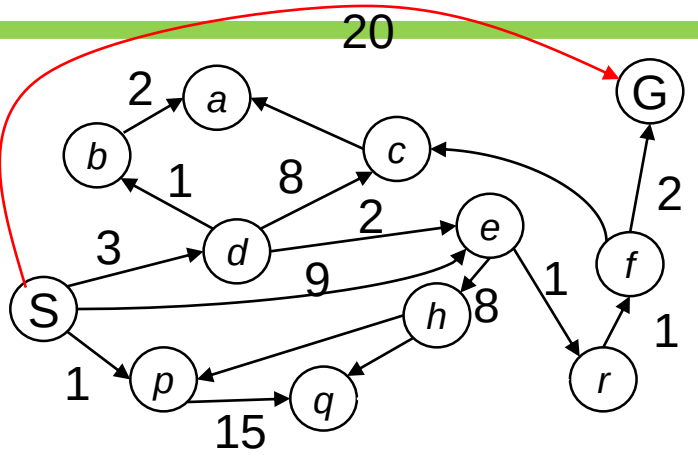
从**frontier**中移出后, **why?**

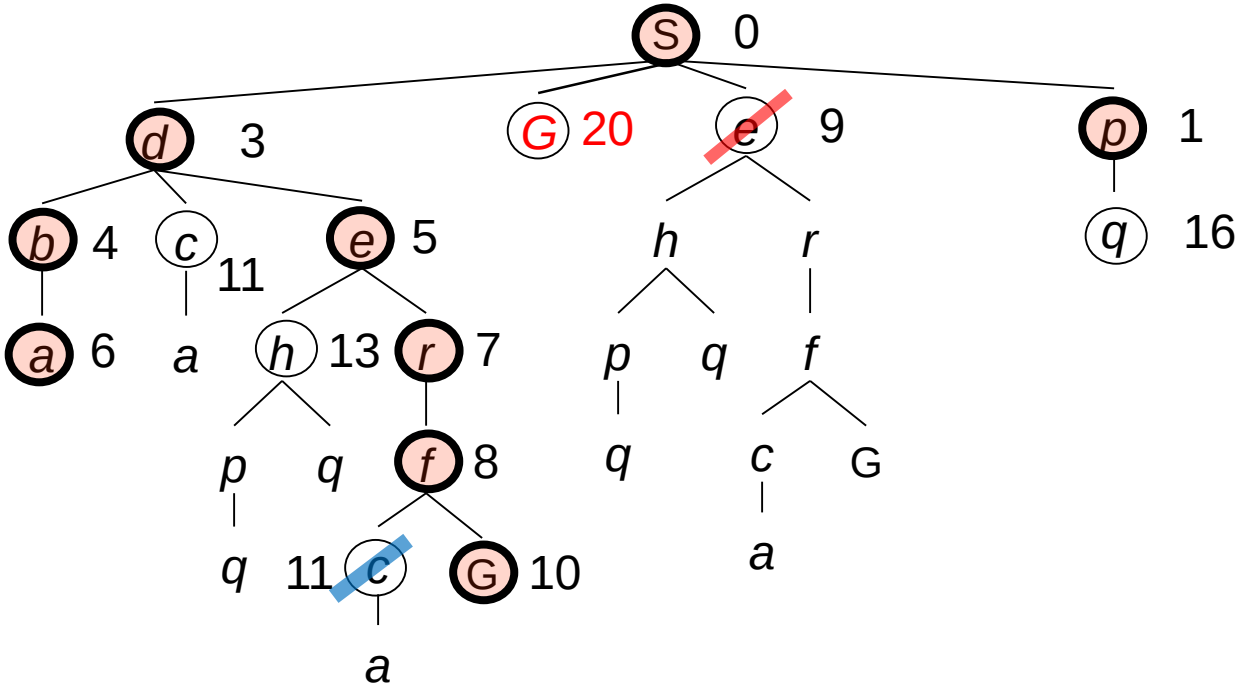
UCS Graph Search

新结点的路径代价一定高于任何**explored**结点

Uniform Cost Search {3.4.2}

- If goal-test before the node is added to the frontier, the lowest-cost solution may not be found. The solution will be S-G.



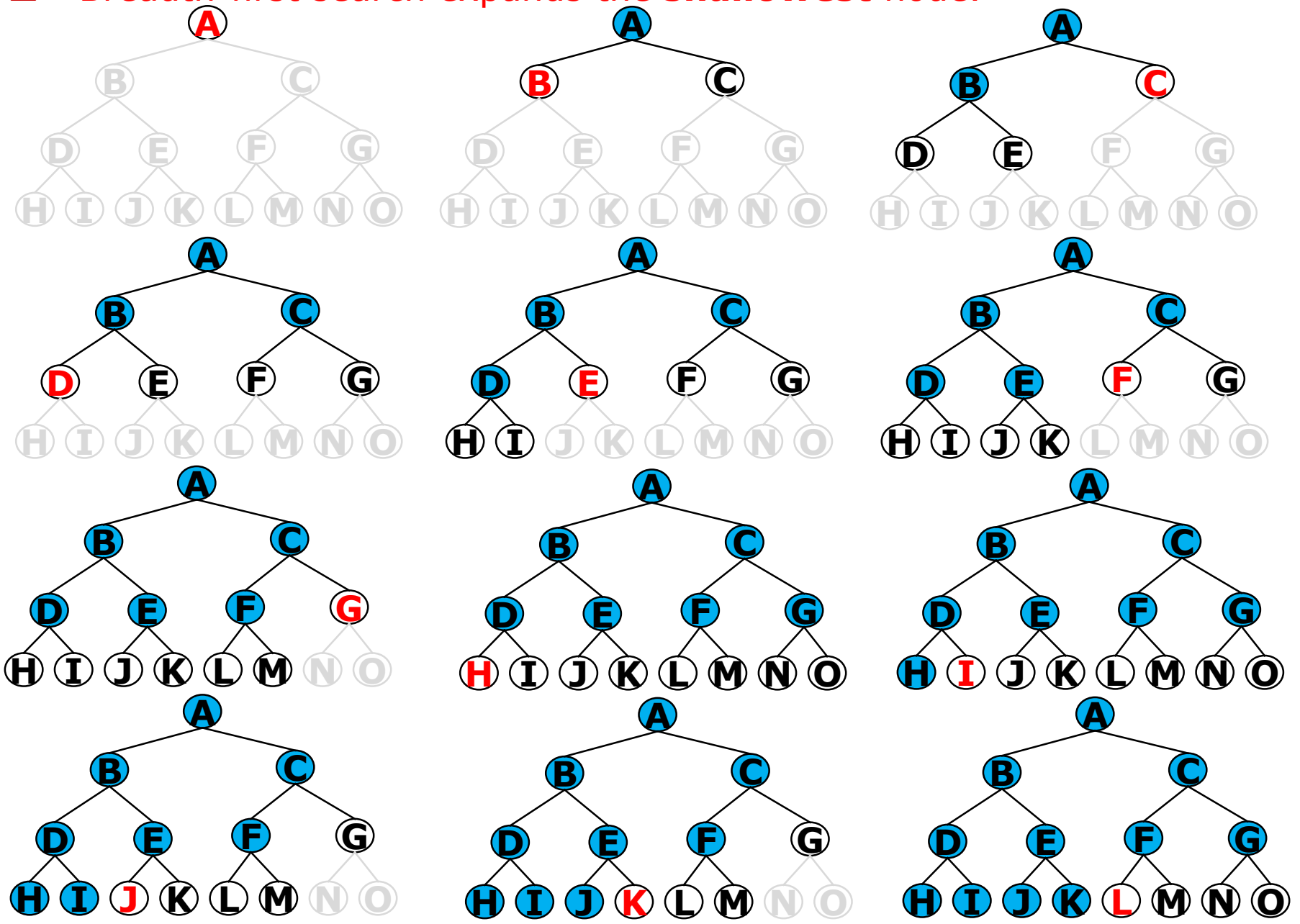


Uniform Cost Search {3.4.2}

- Equivalent to breadth-first if step costs all equal
- It is Optimal
- Complete if the cost of every step is greater than or equal to some small positive constant ε. (如果有0代价行动则可能陷入死循环, 例如程序指令的NoOp行动.)
- Time and Space complexity is $O(b^{1+[c^*/\varepsilon]})$
  - where C* is the cost of the optimal solution
  - the depth of the optimal solution is $[c^*/\varepsilon]$
  - $1+b+\ldots+b^{[c^*/\varepsilon]}+b^{1+[c^*/\varepsilon]}= O(b^{1+[c^*/\varepsilon]})$

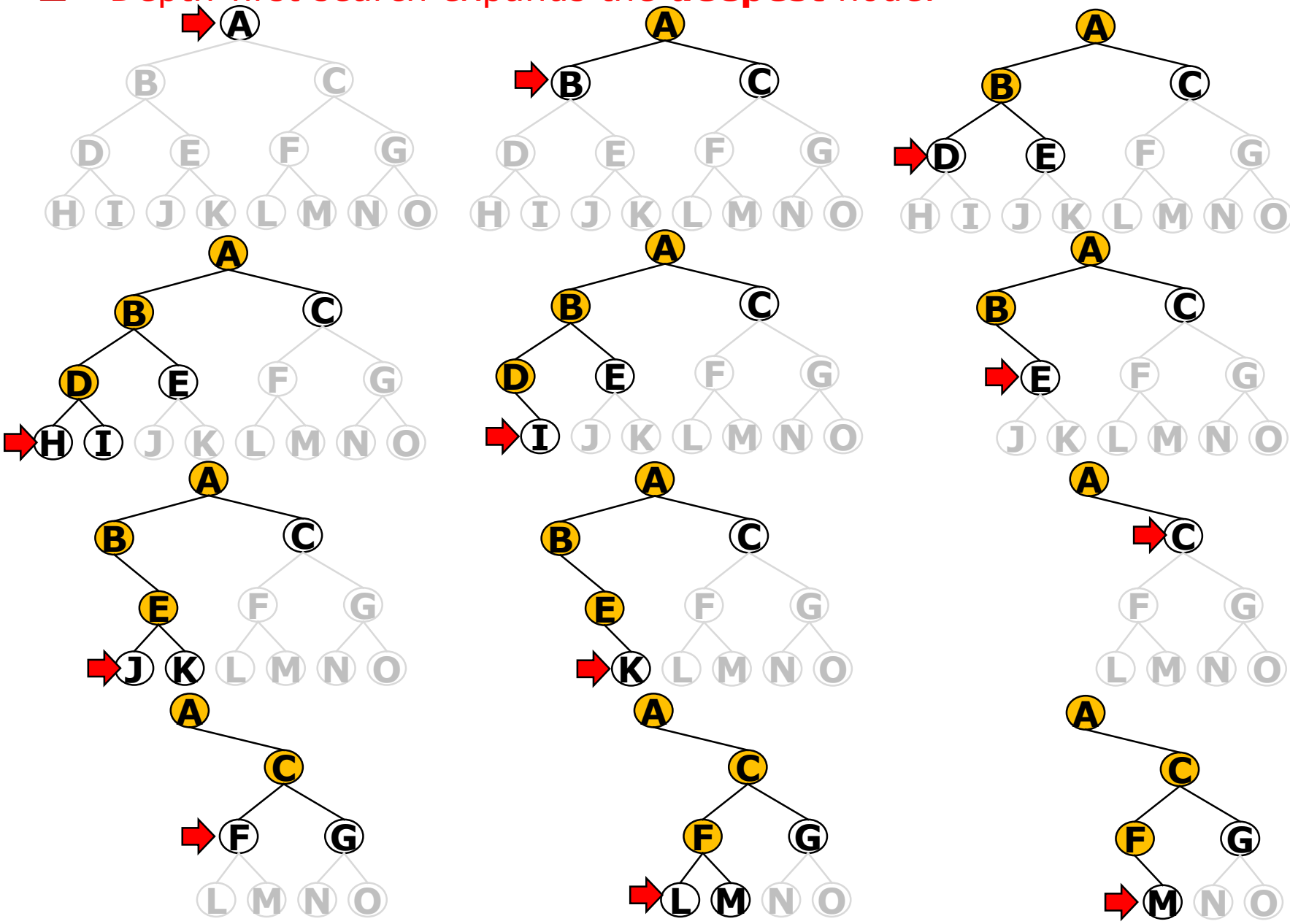
- Breadth-first search requires too large memory space! $O(b^d)$
- How to decrease the space complexity?

- Breadth-first search expands the **shallowest** node.



space complexity is $O(b^d)$, b is the branching factor, d is the goal depth.

- Depth-first search expands the **deepest** node.



space complexity is O(bm), b is the branching factor, m is the largest depth

complete? optimal? Complexity. {3.4.3}

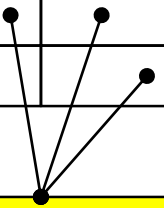
- Complete?

	finite space	infinite space
depth first tree search	no	no
depth first graph search	complete	no

- not Optimal
- b: branching factor
- d: depth of shallowest goal node
- m: maximal depth of a leaf node
- Number of nodes generated (worst case):
  $1 + b + b^2 + \ldots + b^m = O(b^m)$
- Time complexity is $O(b^m)$
- Space complexity is $O(bm)$

complete?

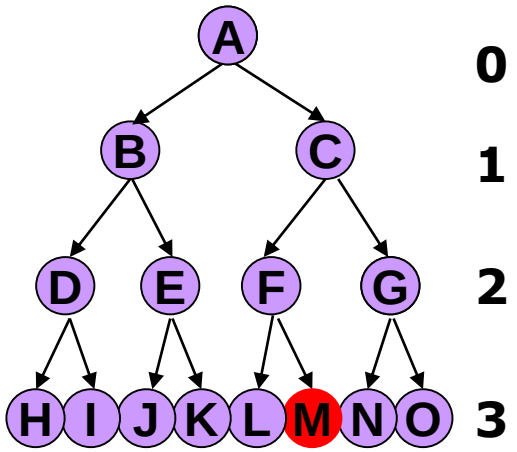
	finite space	infinite space
depth first tree search	no	no
depth first graph search	complete	no



Resulting an infinite search tree, and thus not complete.(可能搜索无限深)
How to deal with the infinite search tree???

Depth-Limited Search {3.4.4}

- Depth-first with cutoff k depth below which nodes are not expanded
- Recursive-DLS(*node*, *problem*, *limit*) returns:
  - Solution
  - Failure (no solution)
  - Cutoff (no solution within cutoff)



Recursive-DLS(A,problem,2) returns cutoff
Recursive-DLS(A,problem,3) returns solution
Recursive-DLS(B,problem,2) returns cutoff
Recursive-DLS(B,problem,3) returns failure

Depth-limited search {3.4.4}

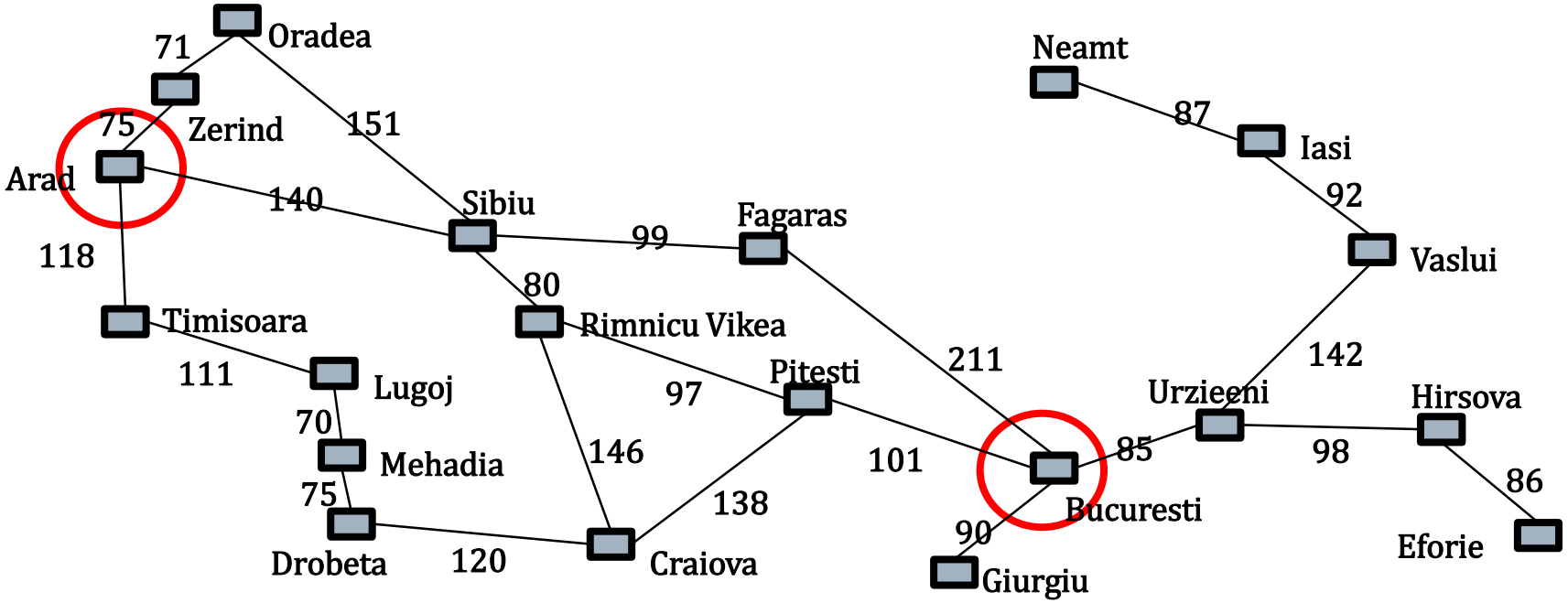
```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

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
```

Figure 3.17 A recursive implementation of depth-limited tree search.

how to determine depth limit?

Determine the depth limit? Example: Romania Problem



- Only 20 cities on the map
- Any city can be reached from any other city in less than 10 steps

complete?optimal?complexity

- not complete
- not optimal
- L: depth limit
- time complexity $O(b^L)$,
  the number of nodes generated.
- space complexity $O(bL)$

Depth Limited Search is of low space complexity and not necessarily to find the shallowest solution.
Breadth First Search is of high space complexity and is to find the shallowest solution.

low space complexity + shallowest solution, possible ???

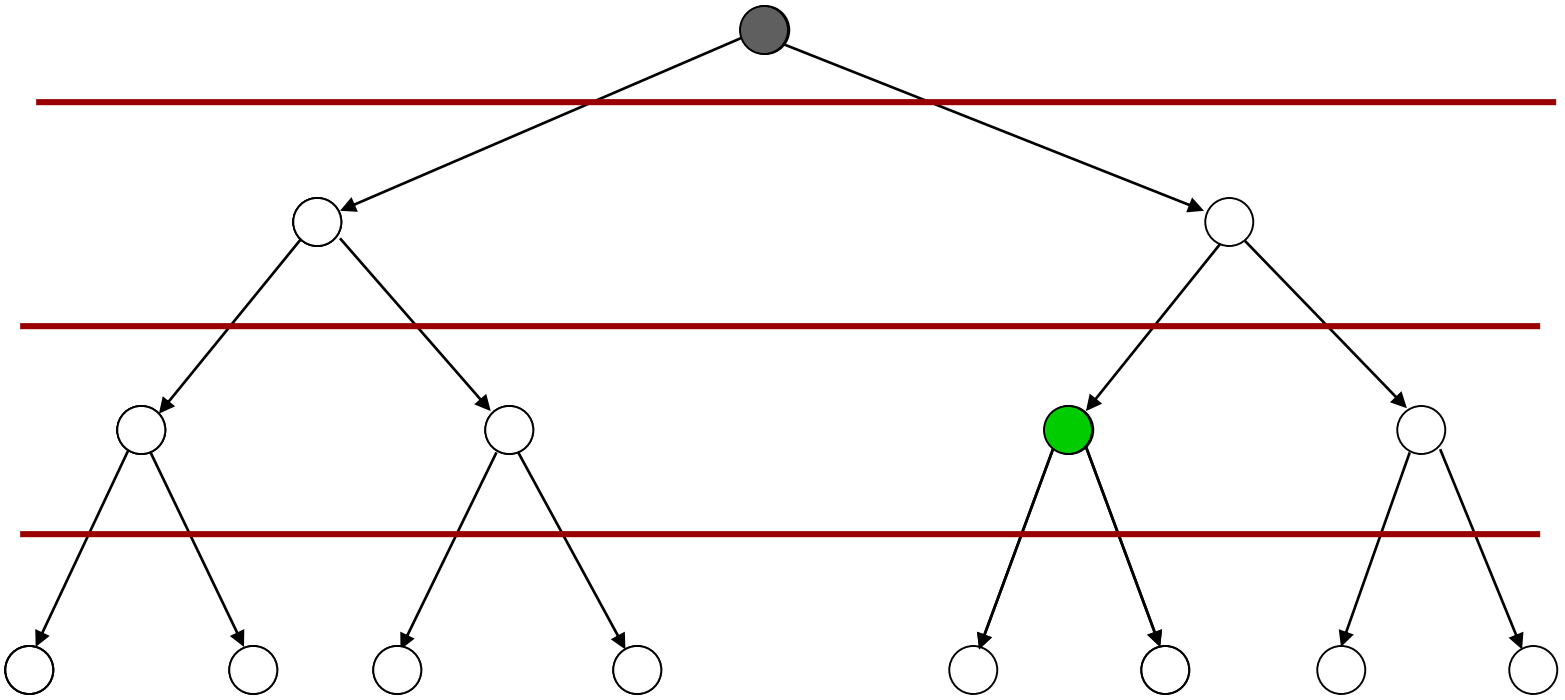
Depth-Limited-Search(problem, limit=0)
Depth-Limited-Search(problem, limit=1)
Depth-Limited-Search(problem, limit=2)
Depth-Limited-Search(problem, limit=3)
… …

Iterative Deepening Search {3.4.5}

- Iterative Deepening Search(IDS)
  - Provides the best of both breadth-first and depth-first search

```
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
```

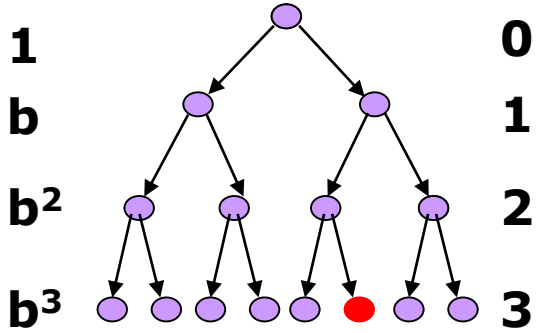
Iterative Deepening {3.4.5}



complete? optimal? {3.4.5}

- complete if the branching factor is finite.
- optimal if the path cost is a nondecreasing function of the depth of the node

- d: the depth of the shallowest goal node.
- space complexity: O(bd)
- how many nodes are going to be generated?

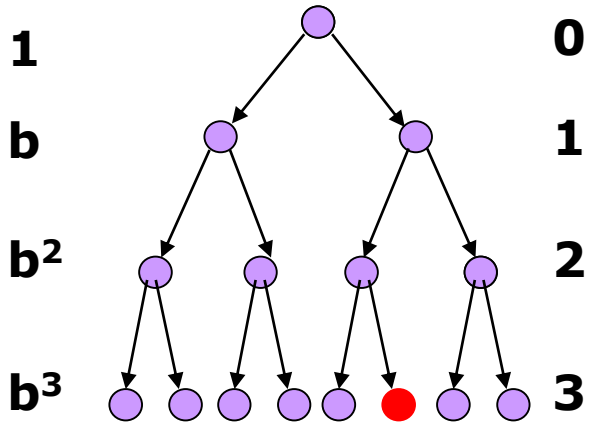


the number of nodes generated{3.4.5}

- suppose d=3
- Depth Limited Search

  $1+b+b^2+b^3$

  in general: $N_{DLS}=1+b+b^2+\ldots+b^d=O(b^d)$



- Iterative Deepening Search

  1 limit=0

  1+b limit=1

  $1+b+b^2$ limit=2

  $1+b+b^2+b^3$ limit=3

  total in general: $N_{IDS}=(d+1)1+db+(d-1)b^2+\ldots+1b^d = O(b^d)$

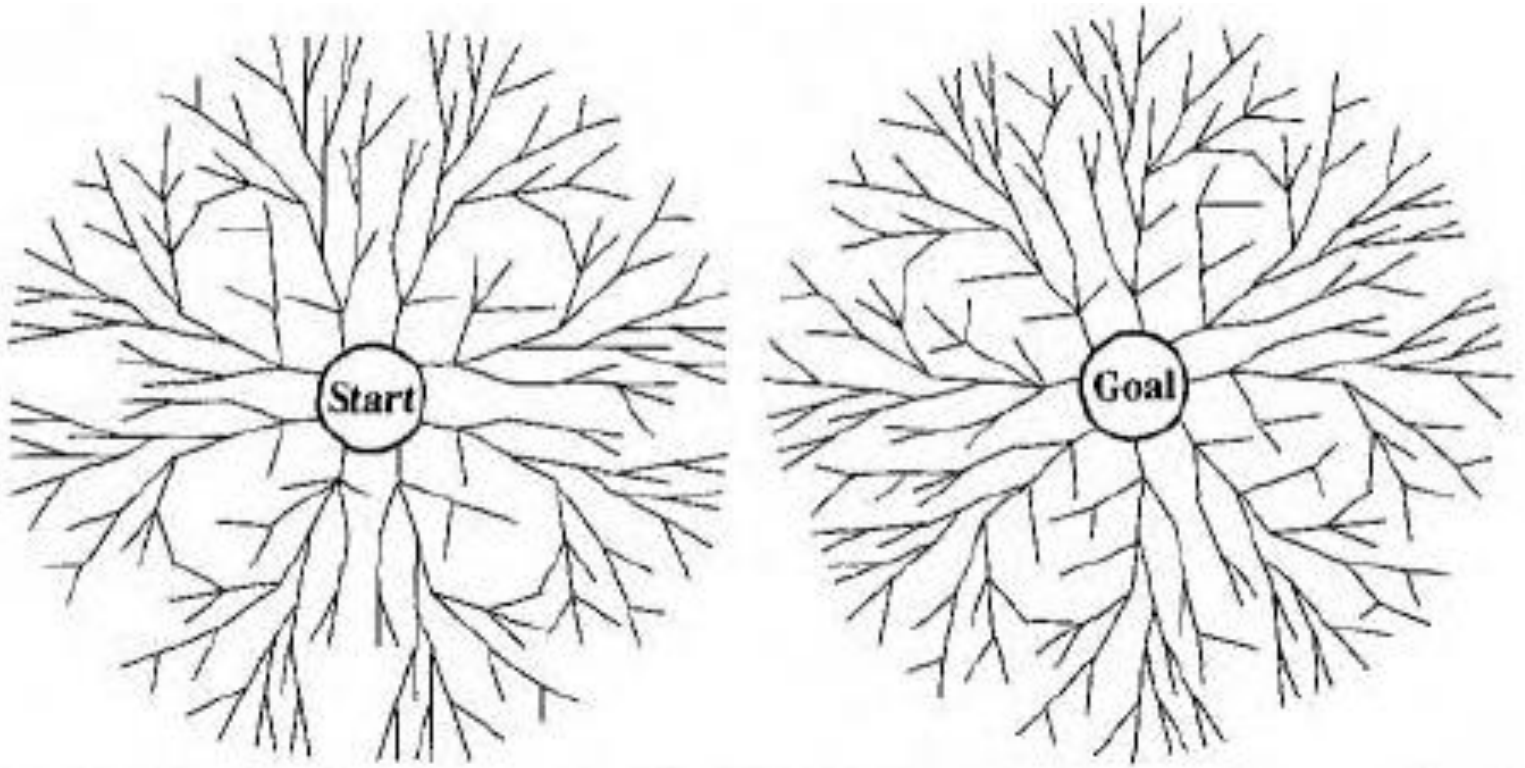
the number of nodes generated {3.4.5}

- For *b = 10*, *d = 5*,
  - N_{DLS} = 1+10+100+1,000+10,000+100,000 = 111,111
  - N_{IDS} =6+50+400+3,000+20,000+100,000 = 123,456
- Overhead
  - (123,456-111,111)/111,111 = 11%

In general, iterative deepening is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.

Bi-directional search {3.4.6}

- $b^{d/2} + b^{d/2}$ is much less than b^d



祝恩

summary

- 搜索问题形式化
- 树搜索，图搜索，及算法评估
- 策略：宽度优先，一致代价，深度优先，深度受限，迭代加深，双向搜索

祝恩

all concepts

搜索问题形式化，初始状态，行动Actions(s)，转移模型Result(s,a)，目标测试Goal-Test(s)，路径代价Path-Cost/g(n)。

盲目搜索，启发式搜索。树搜索，图搜索，frontier，explored，结点与状态。

完备性，最优性，时间/空间复杂性。

宽度优先搜索，一致代价搜索，深度优先搜索，深度受限搜索，迭代加深搜索，双向搜索。