Last class

- What is search
- Problem complexity: P, NP, NP-hard, NP-complete
- Tree-search and graph-search
- Performance evaluation criteria
- Asymptotic notations

Heuristic Search and Evolutionary Algorithms

Lecture 2: Uninformed Search

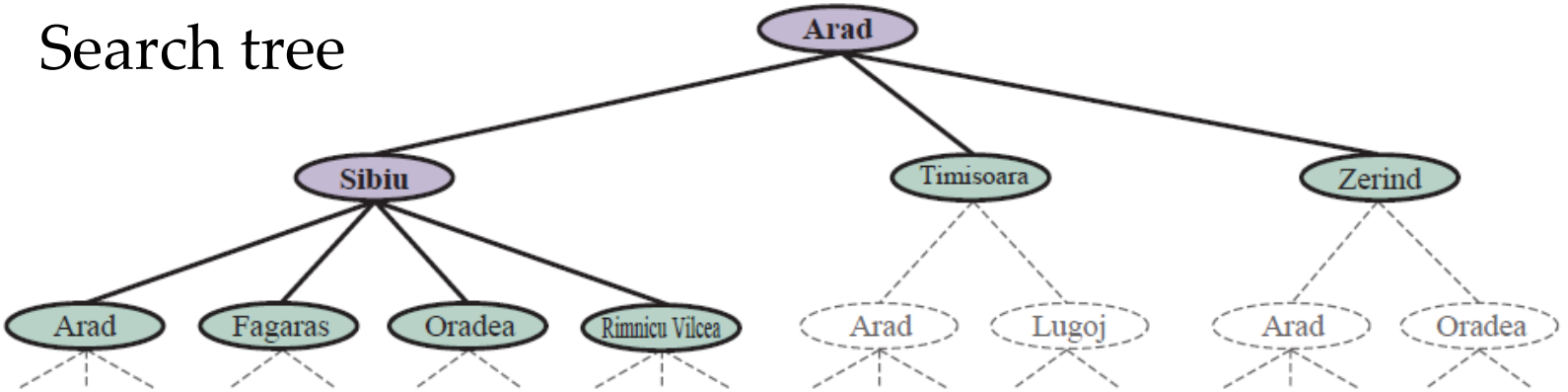
Chao Qian （钱超）

Associate Professor, Nanjing University, China

Email: qianc@nju.edu.cn
Homepage: http://www.lamda.nju.edu.cn/qianc/

Search algorithms

Search tree



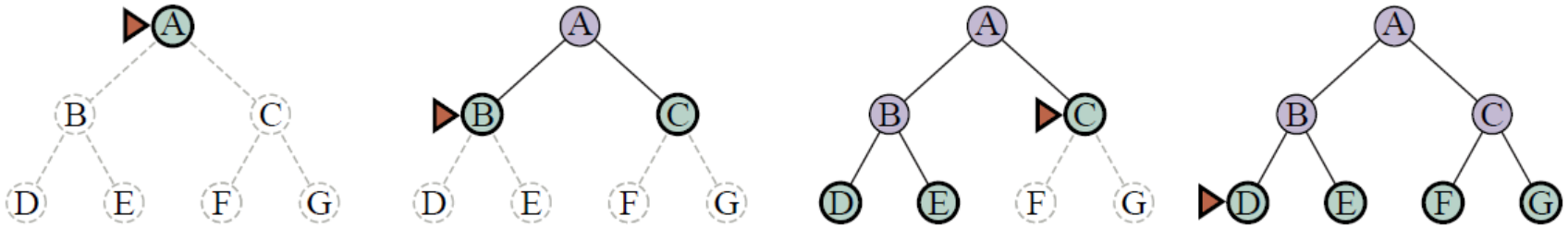
Explored set: Arad, Sibiu

Frontier: Fagaras, Oradea, Rimnicu Vilcea, Timisoara, Zerind

How to select one node from the frontier for expansion? ⇨ Different search algorithms

Breadth-first search

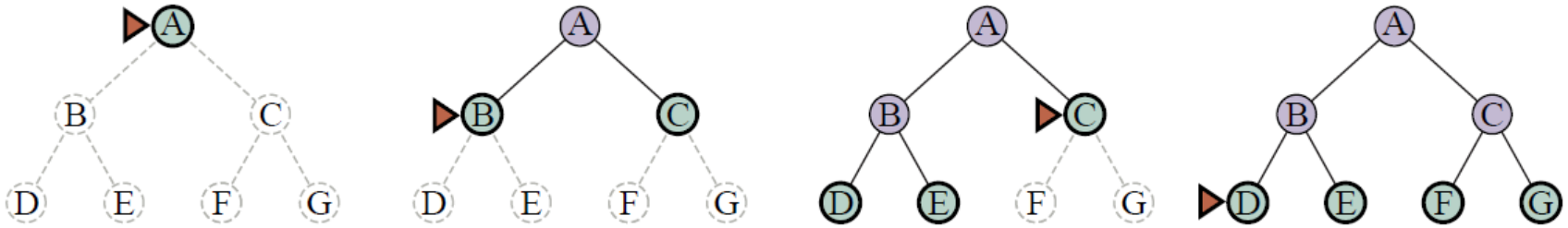
Main idea: expand the root node first, then all the successors of the root node, then their successors, and so on




```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

Breadth-first search

Main idea: expand the root node first, then all the successors of the root node, then their successors, and so on




```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

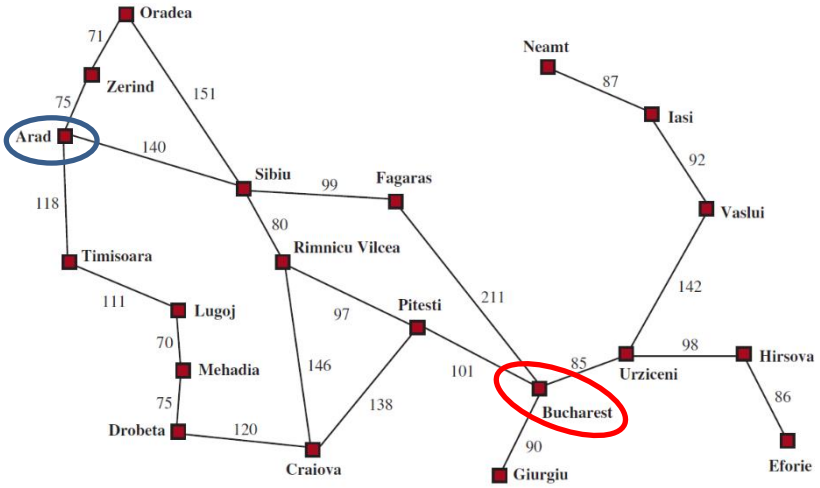
Expand the shallowest node

Goal test is applied when a node is generated rather than when it is expanded

Graph-search

The union of explored set and frontier

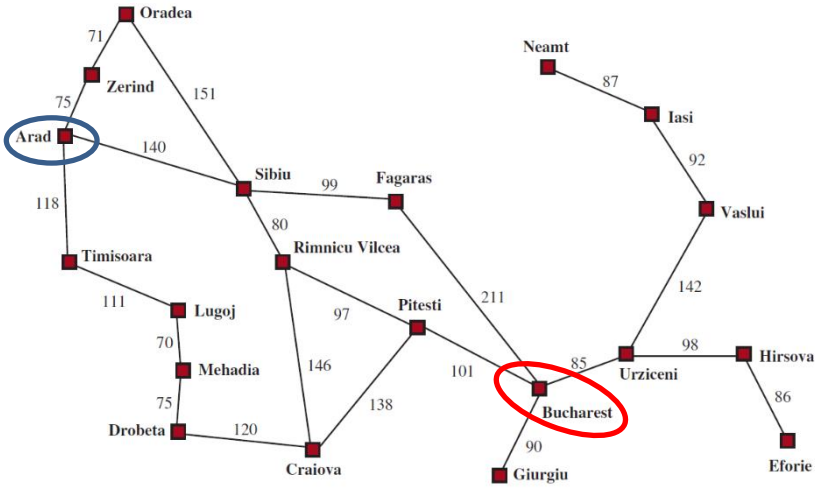
Breadth-first search - example



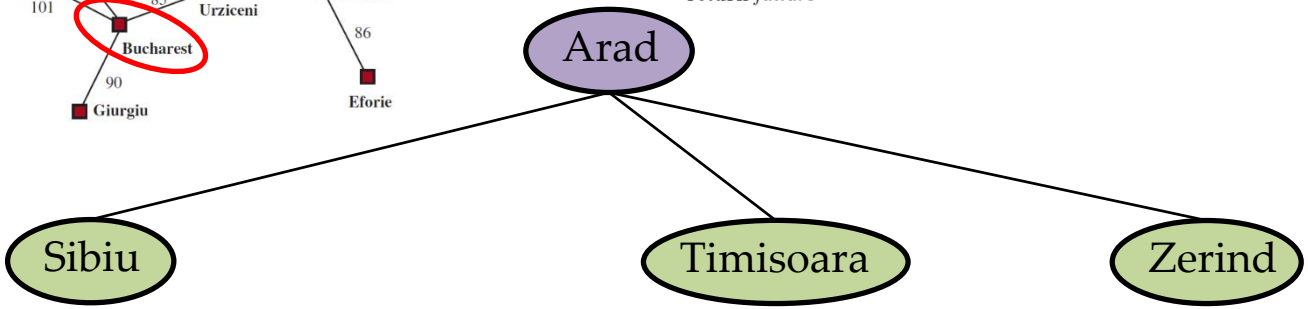

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

Arad

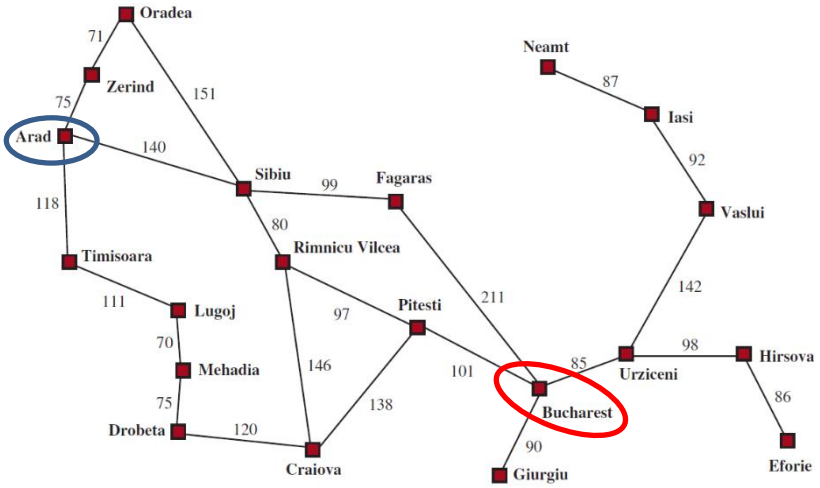
Breadth-first search - example



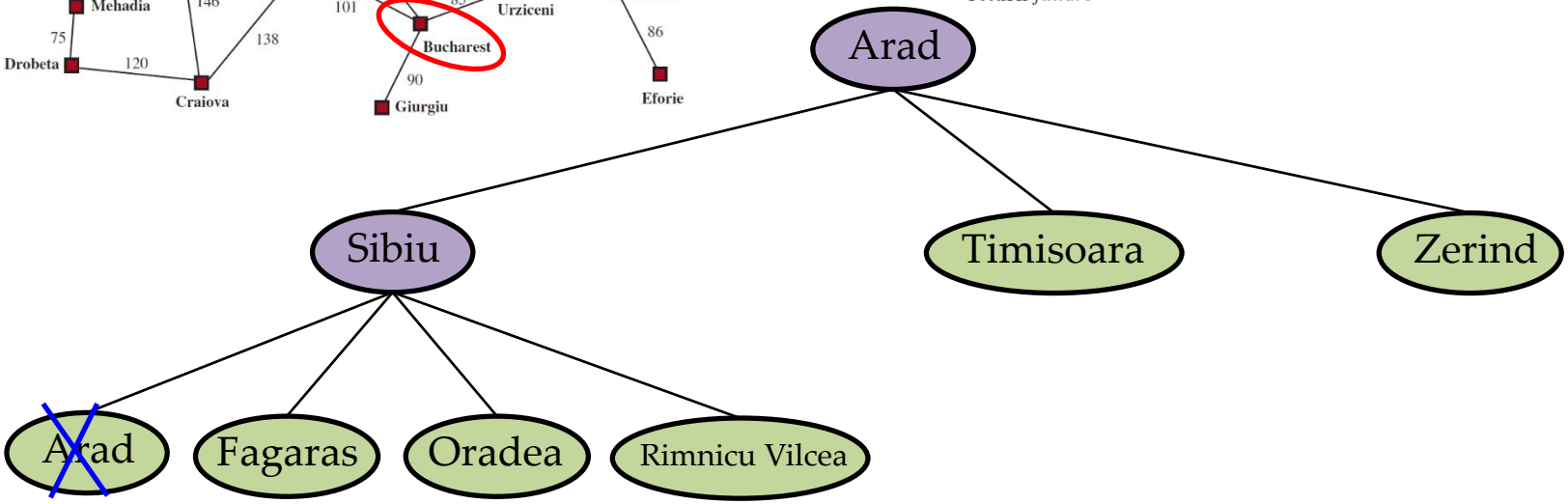

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```



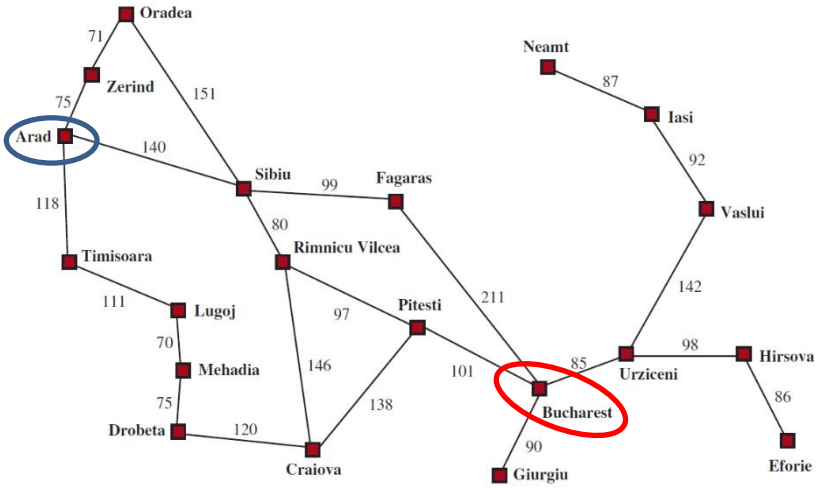
Breadth-first search - example



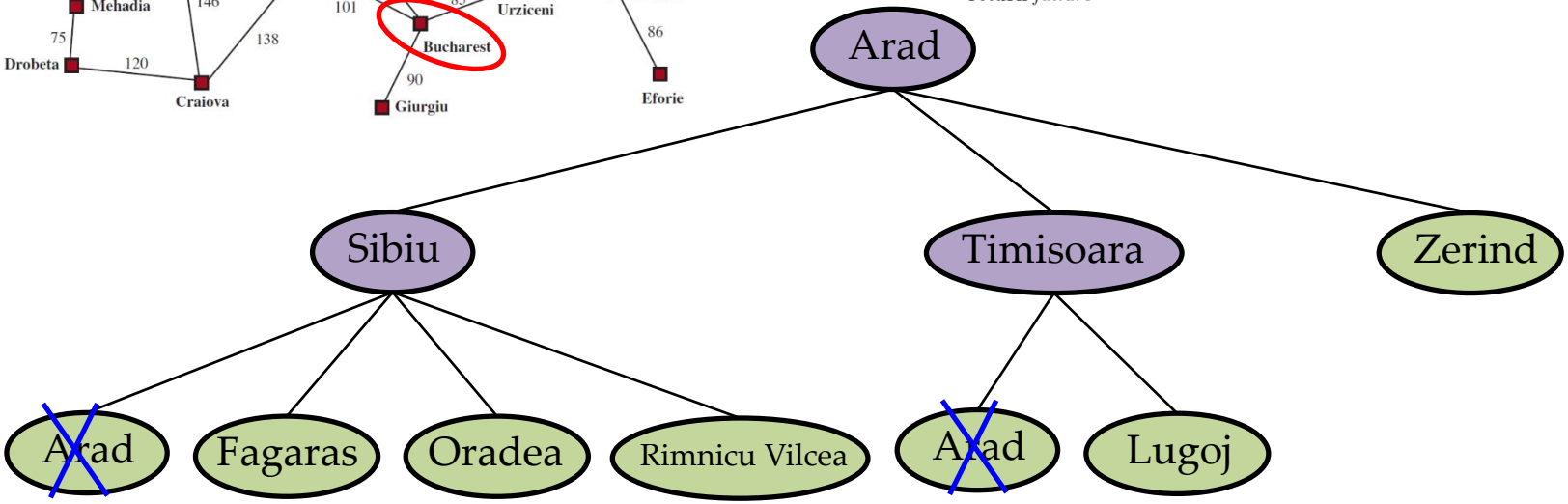
function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution node or *failure*
  node ← NODE(*problem*.INITIAL)
  if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
  frontier ← a FIFO queue, with *node* as an element
  reached ← {*problem*.INITIAL}
  while not IS-EMPTY(*frontier*) **do**
    node ← POP(*frontier*)
    for each *child* **in** EXPAND(*problem*, *node*) **do**
      s ← *child*.STATE
      if *problem*.IS-GOAL(*s*) **then return** *child*
      if *s* is not in *reached* **then**
        add *s* to *reached*
        add *child* to *frontier*
  return *failure*



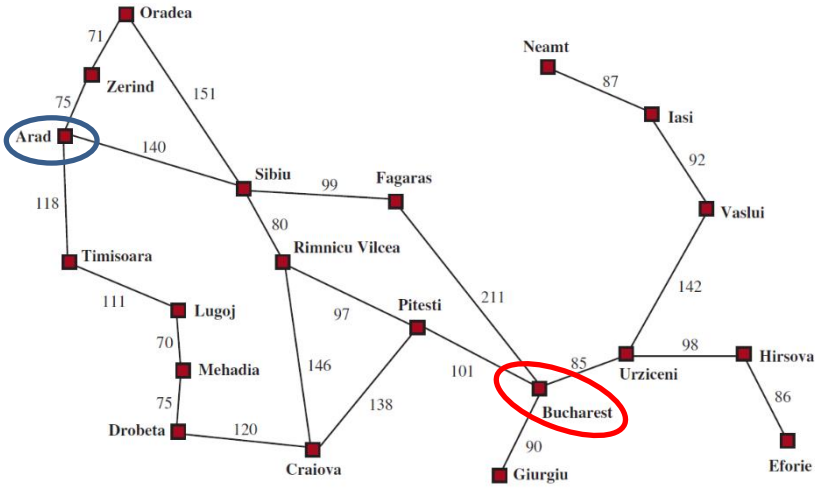
Breadth-first search - example



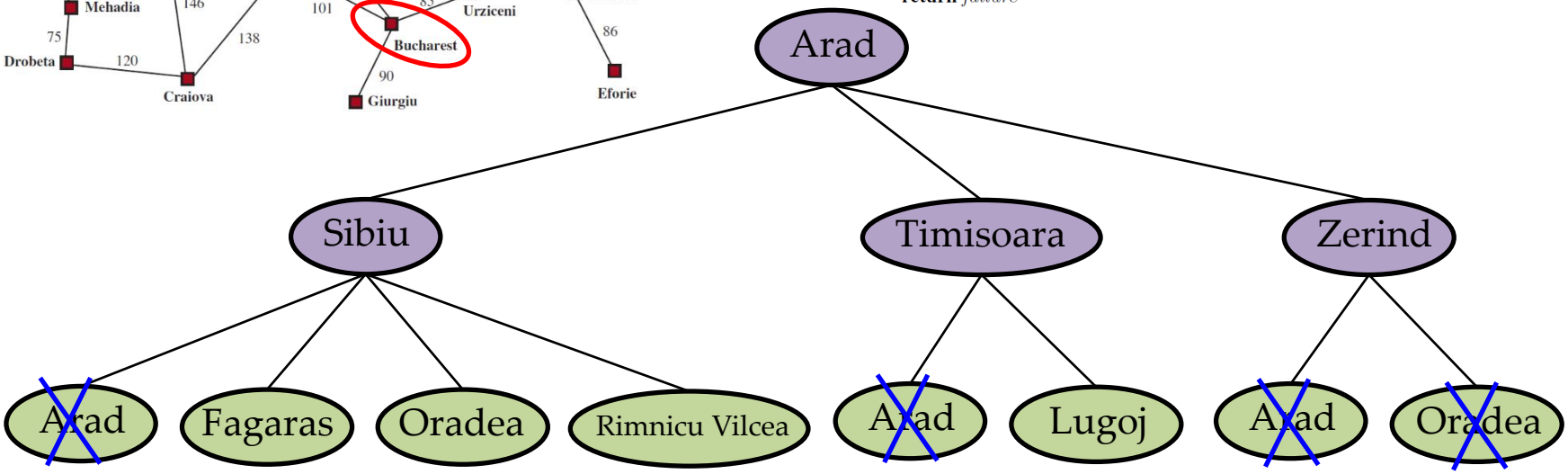

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```



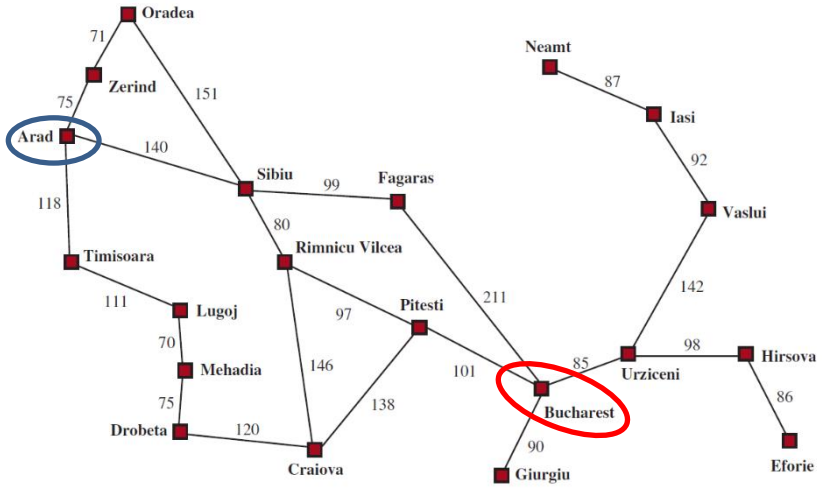
Breadth-first search - example



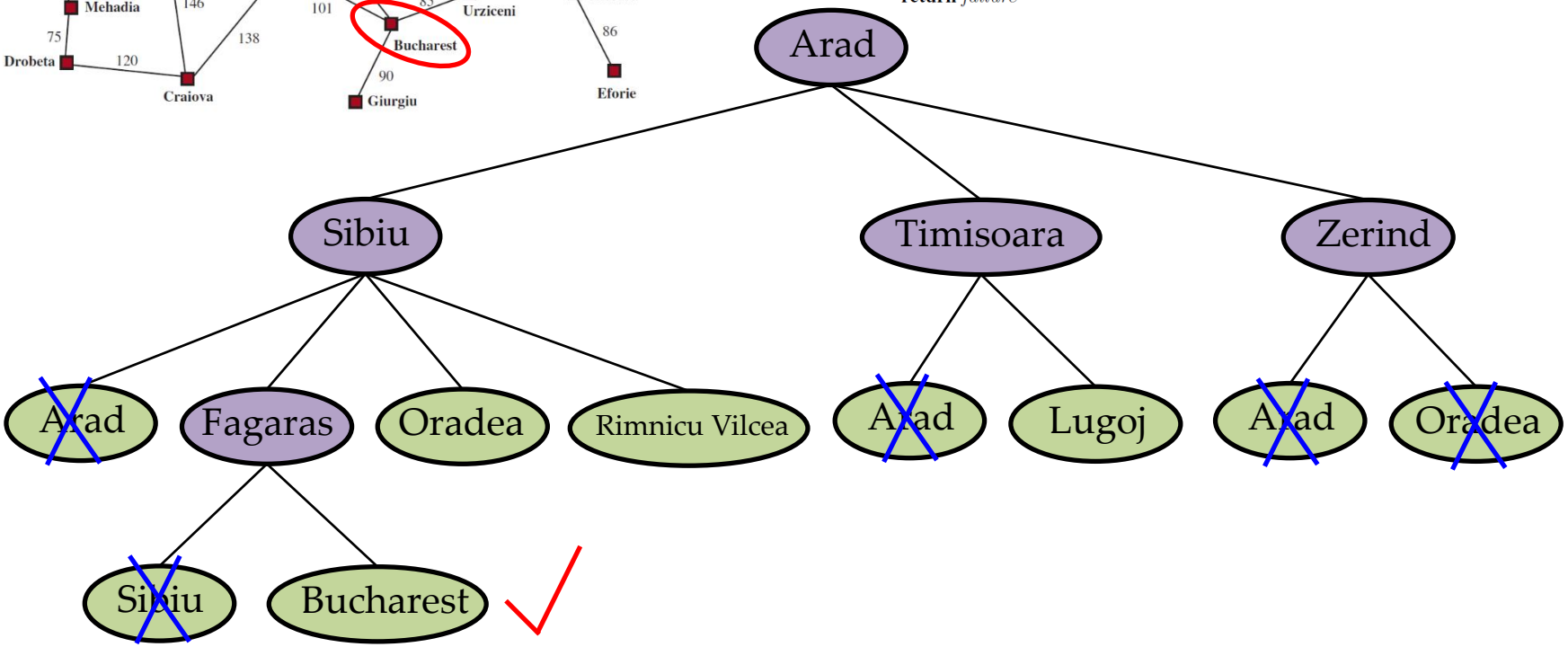

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```



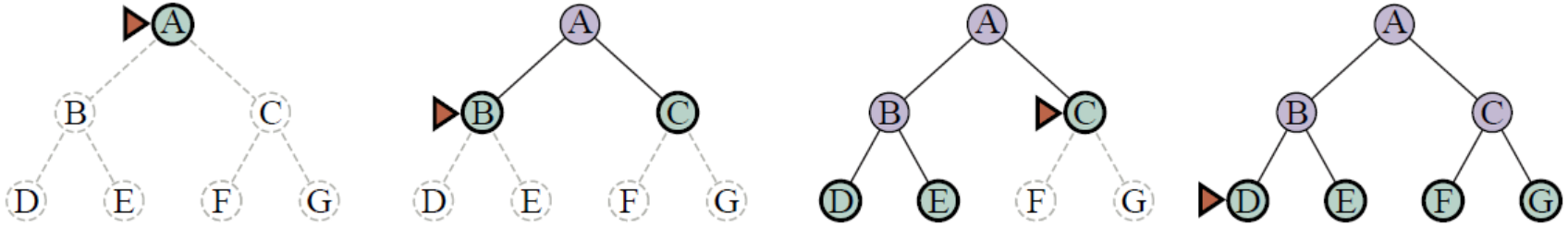
Breadth-first search - example



function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution node or *failure*
  node ← NODE(*problem*.INITIAL)
  if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
  frontier ← a FIFO queue, with *node* as an element
  reached ← {*problem*.INITIAL}
  while not IS-EMPTY(*frontier*) **do**
    node ← POP(*frontier*)
    for each *child* **in** EXPAND(*problem*, *node*) **do**
      s ← *child*.STATE
      if *problem*.IS-GOAL(*s*) **then return** *child*
      if *s* is not in *reached* **then**
        add *s* to *reached*
        add *child* to *frontier*
  return *failure*



Breadth-first search - performance



- Complete

  if the depth d of the shallowest goal node is finite

- Optimal

  if all actions have the same cost

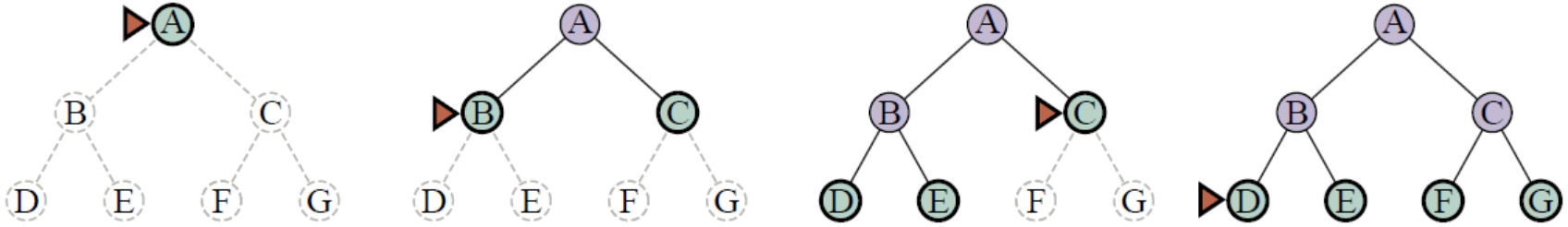
- Time complexity

  $b + b^2 + b^3 + \cdots + b^d = O(b^d)$, *where b is the branching factor*

  If goal test is applied when a node is expanded

  $$b + b^2 + b^3 + \cdots + b^d + b^{d+1} = O(b^{d+1})$$

Breadth-first search - performance



- Time complexity

  $b + b^2 + b^3 + \cdots + b^d = O(b^d)$, *where* b *is the branching factor*

  If goal test is applied when a node is expanded

  $$b + b^2 + b^3 + \cdots + b^d + b^{d+1} = O(b^{d+1})$$

- Space complexity

  $b + b^2 + b^3 + \cdots + b^{d-1} = O(b^{d-1})$ *in the explored set*

  b^d *in the frontier*

  $\Big\}\ O(b^d)$

  If using tree-search, $O(b^d)$

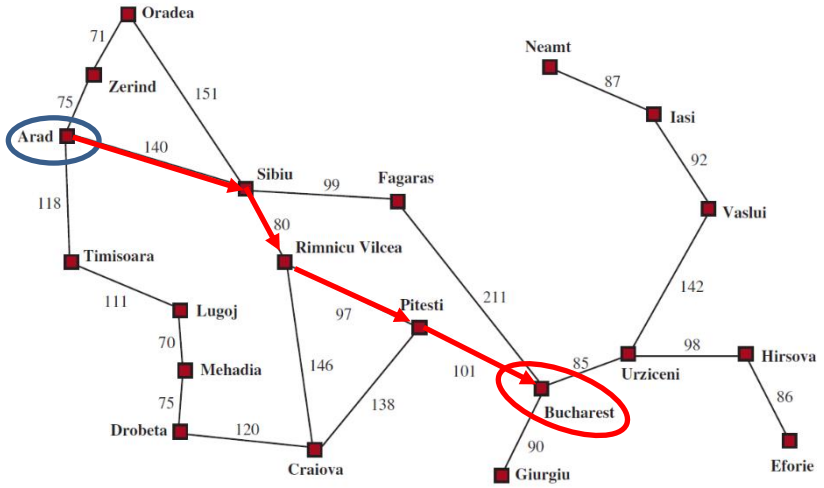
Breadth-first search - performance

$b = 10$ 1 million nodes/second 1000 bytes/node

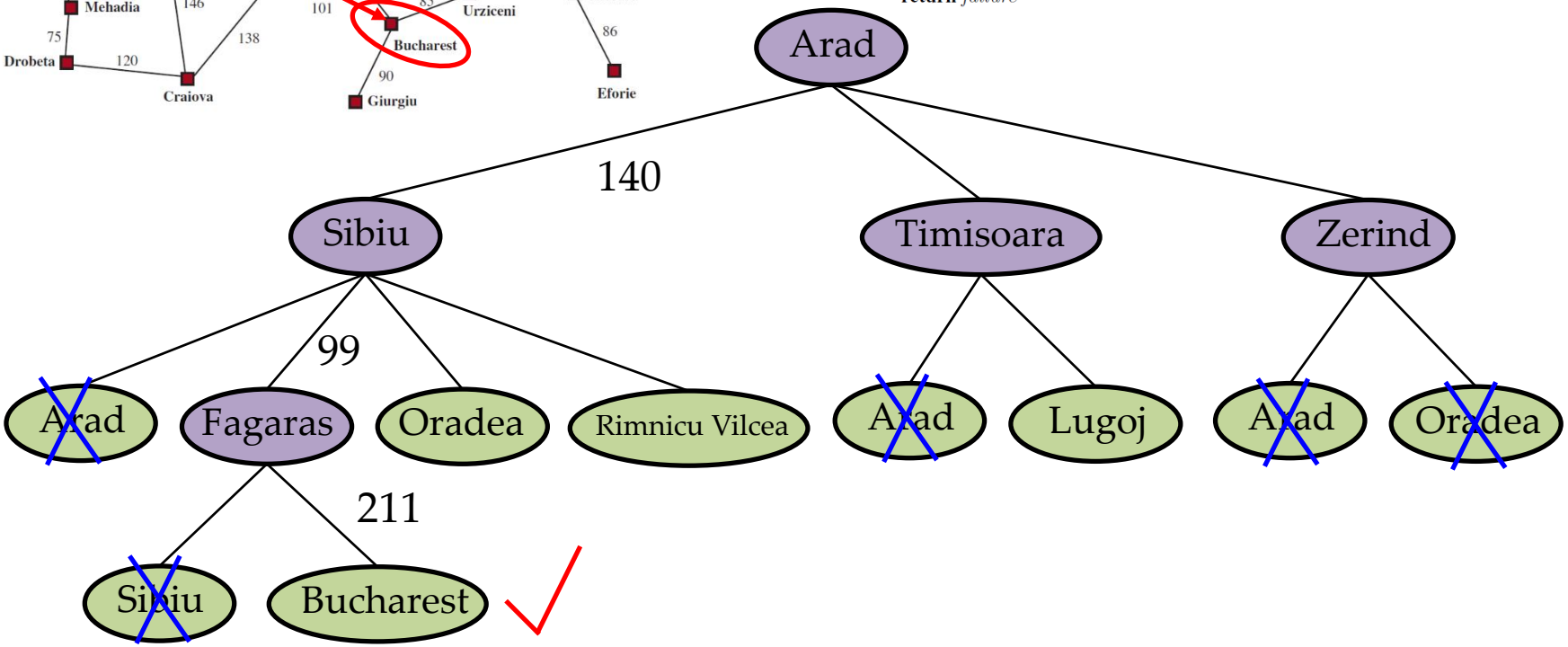
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

The time and space complexity of BFS: not acceptable

Breadth-first search - example




```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```



Uniform-cost search

Main idea: expand the node n with the lowest cost $g(n)$

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

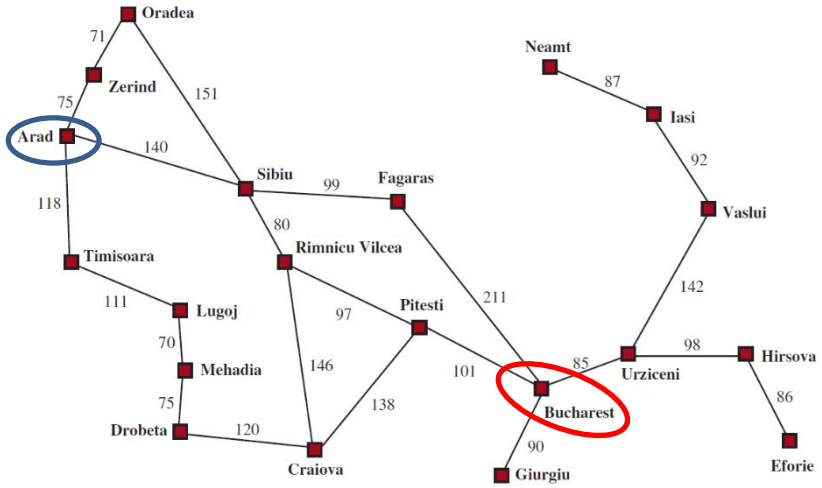
Path cost $g(n)$ from the initial node to the current node

Goal test is applied when a node is expanded rather than when it is generated

The union of explored set and frontier → Graph-search

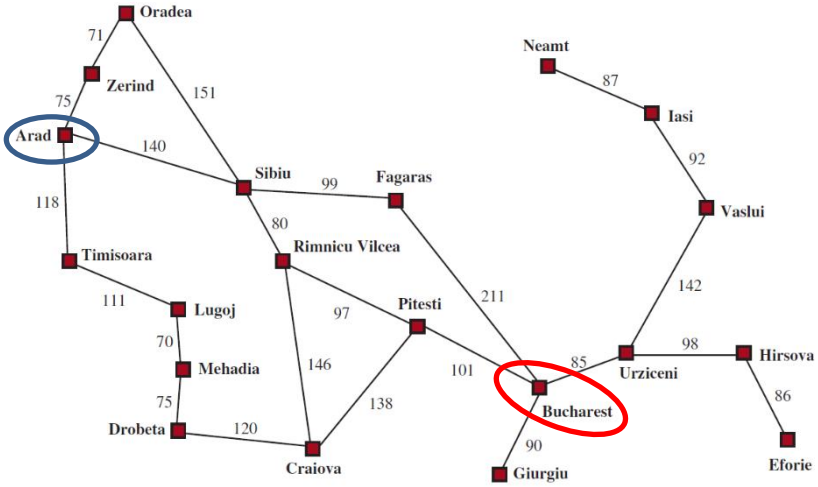
```
function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

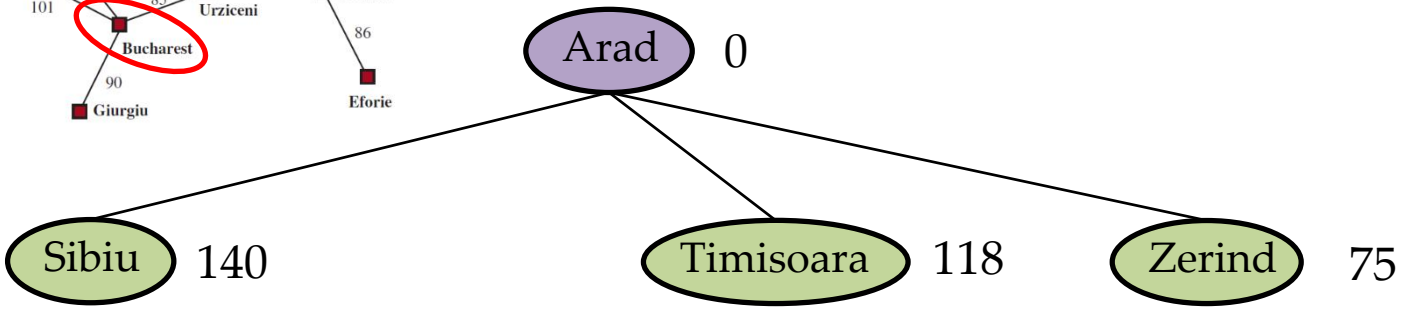
Uniform-cost search - example



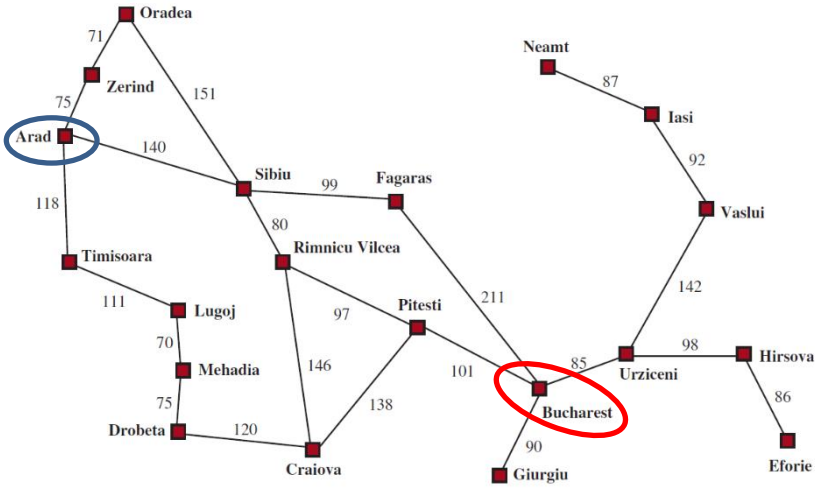
Arad 0

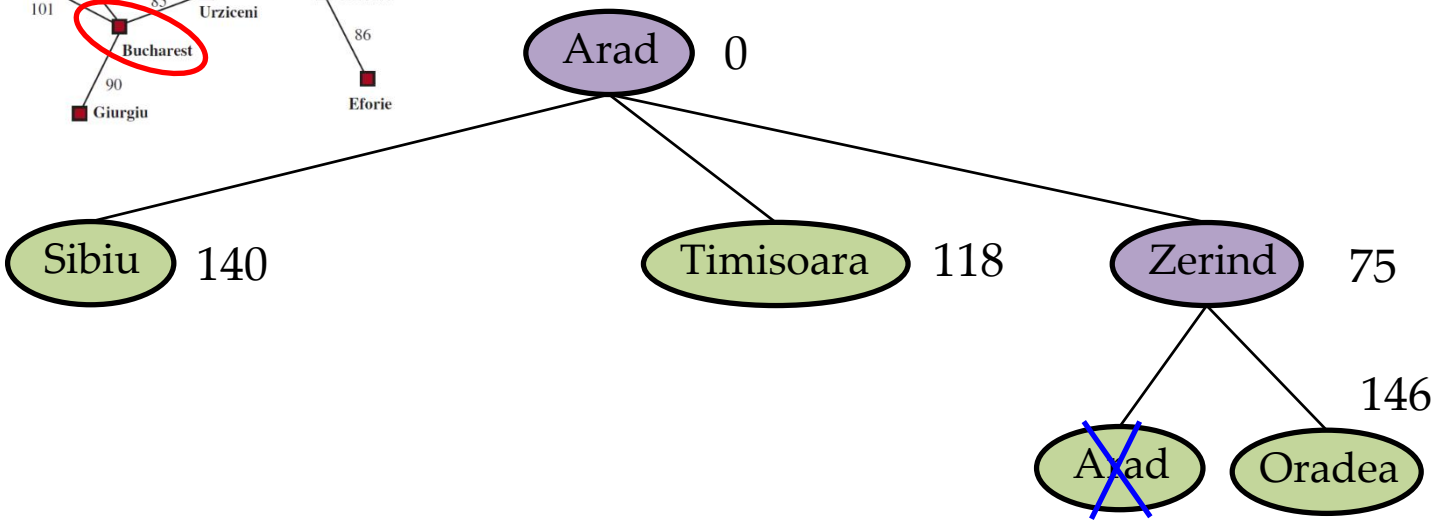
Uniform-cost search - example



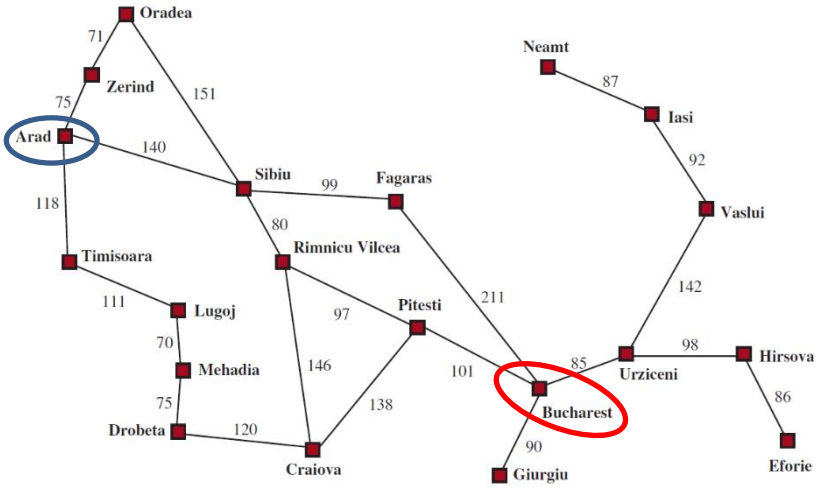


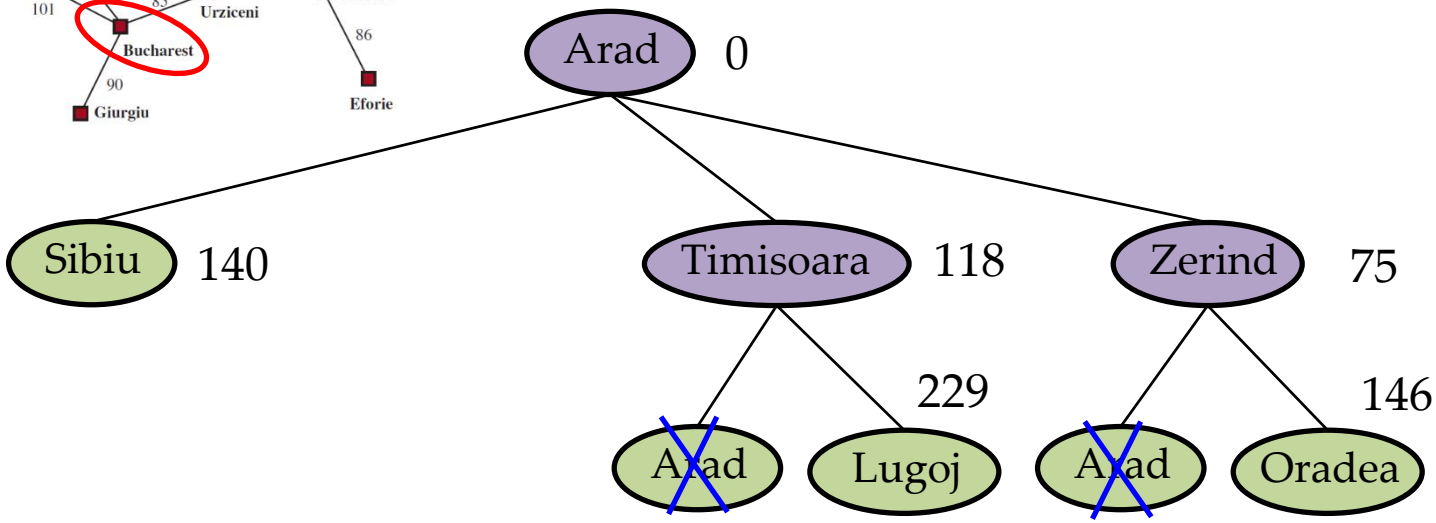
Uniform-cost search - example



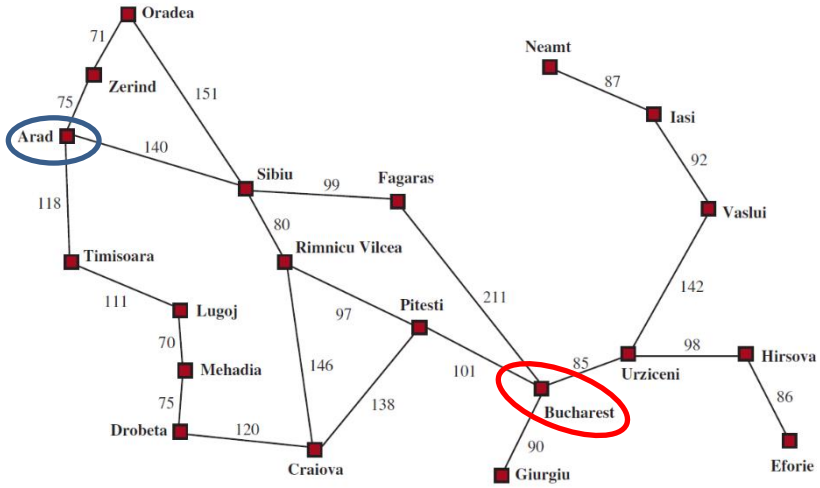


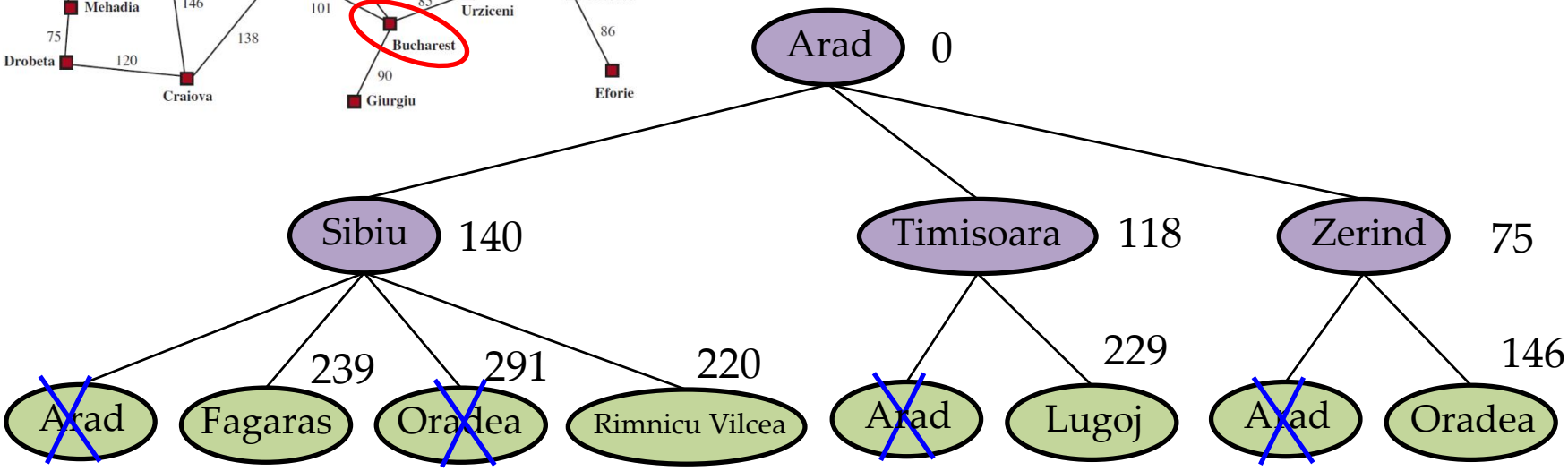
Uniform-cost search - example



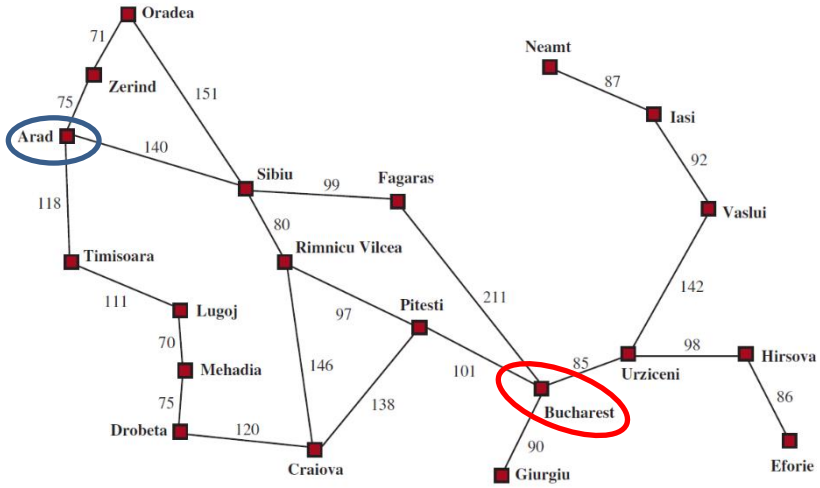


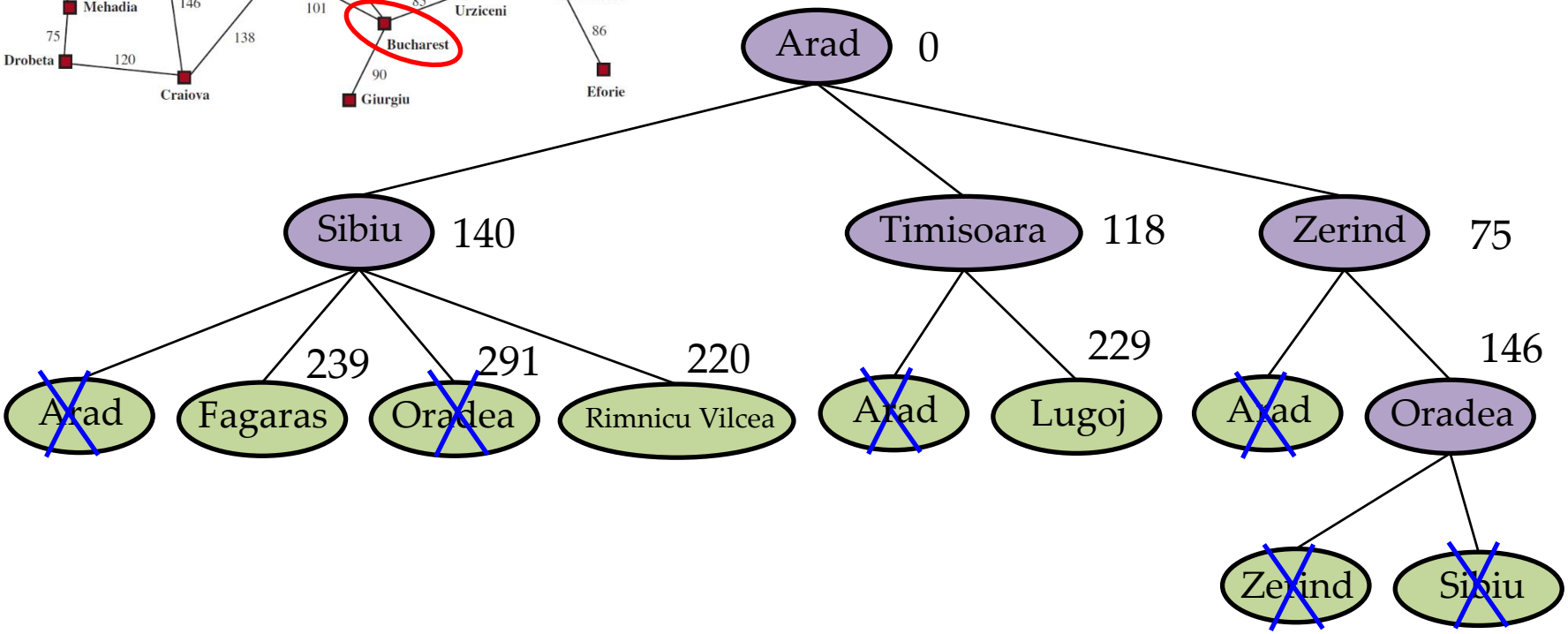
Uniform-cost search - example



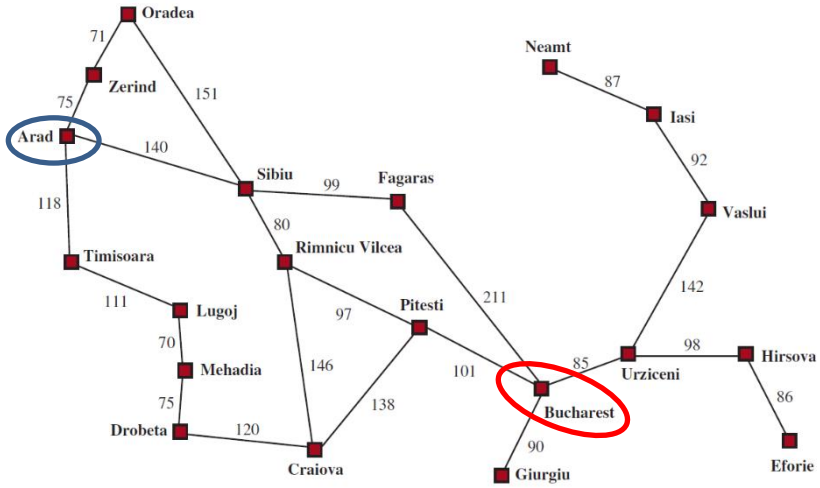


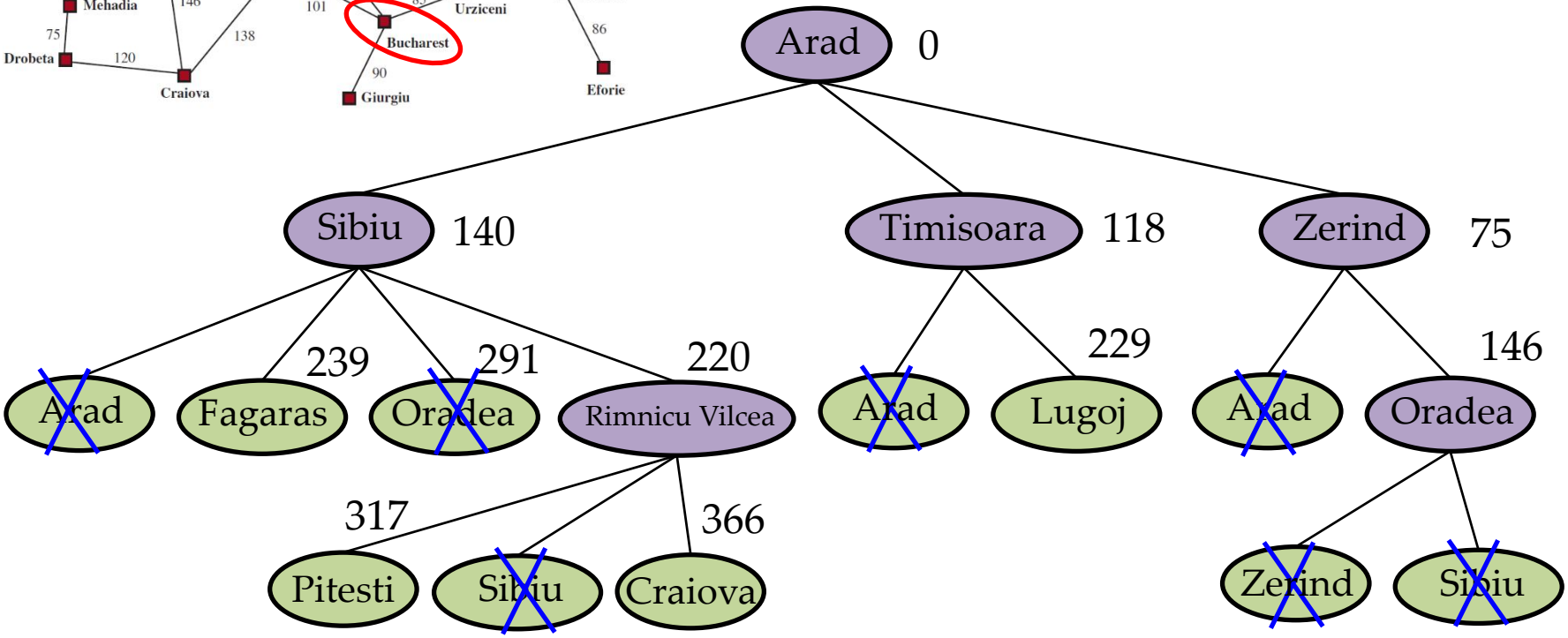
Uniform-cost search - example



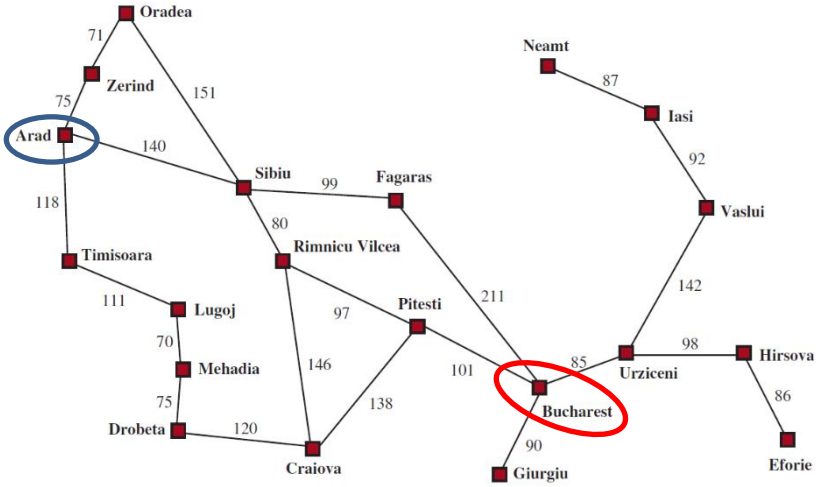


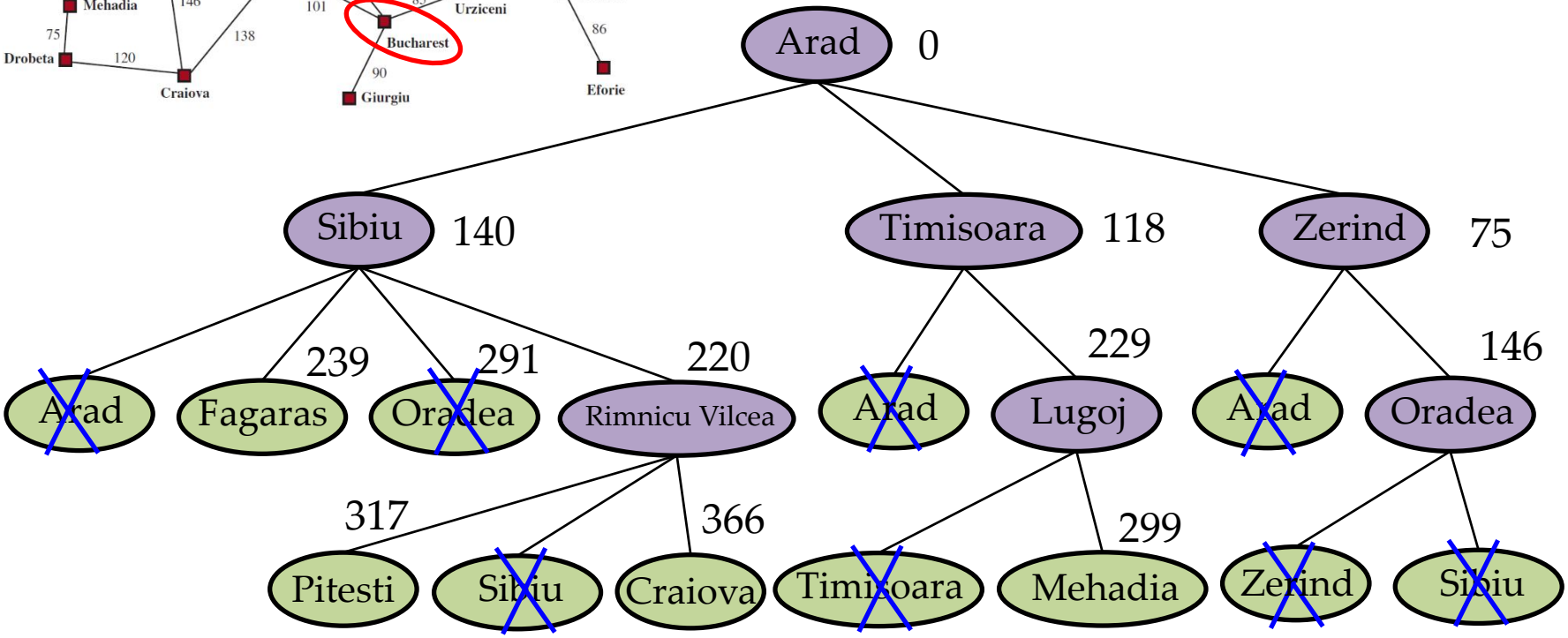
Uniform-cost search - example



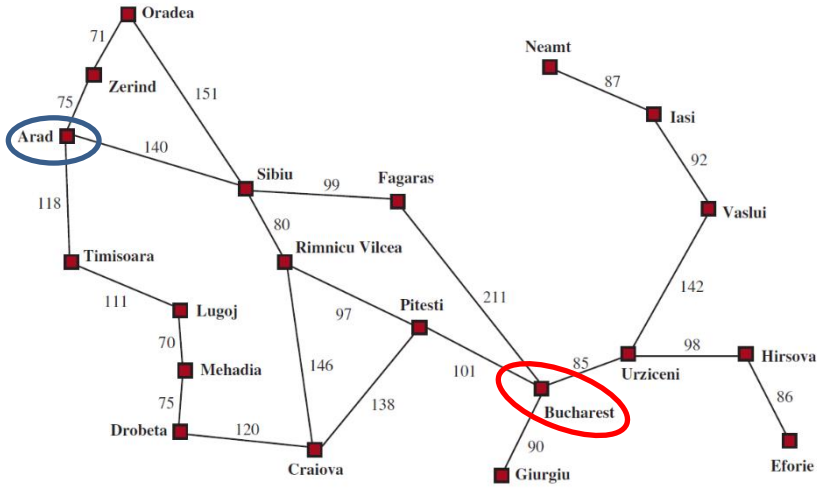


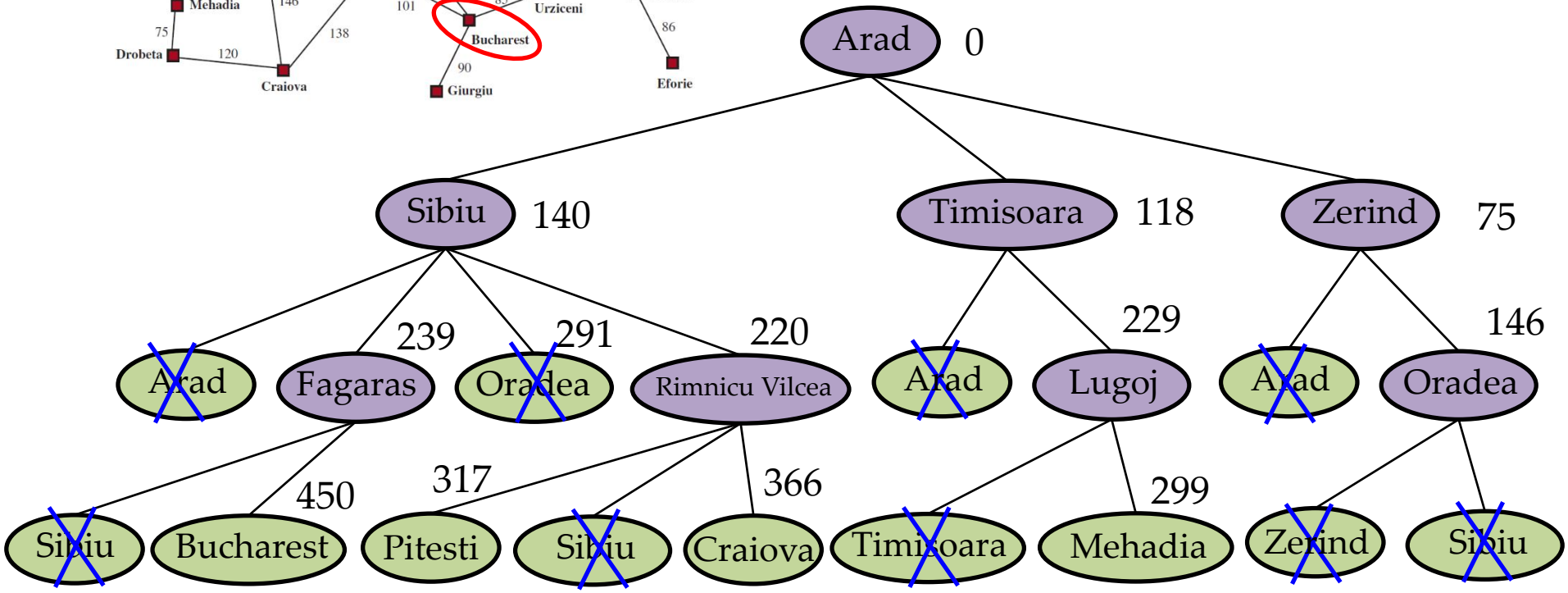
Uniform-cost search - example



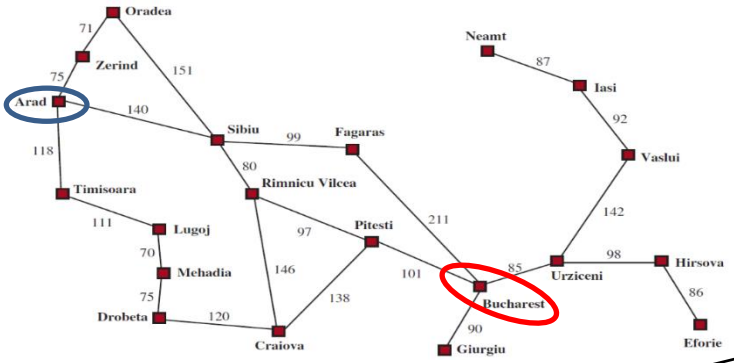


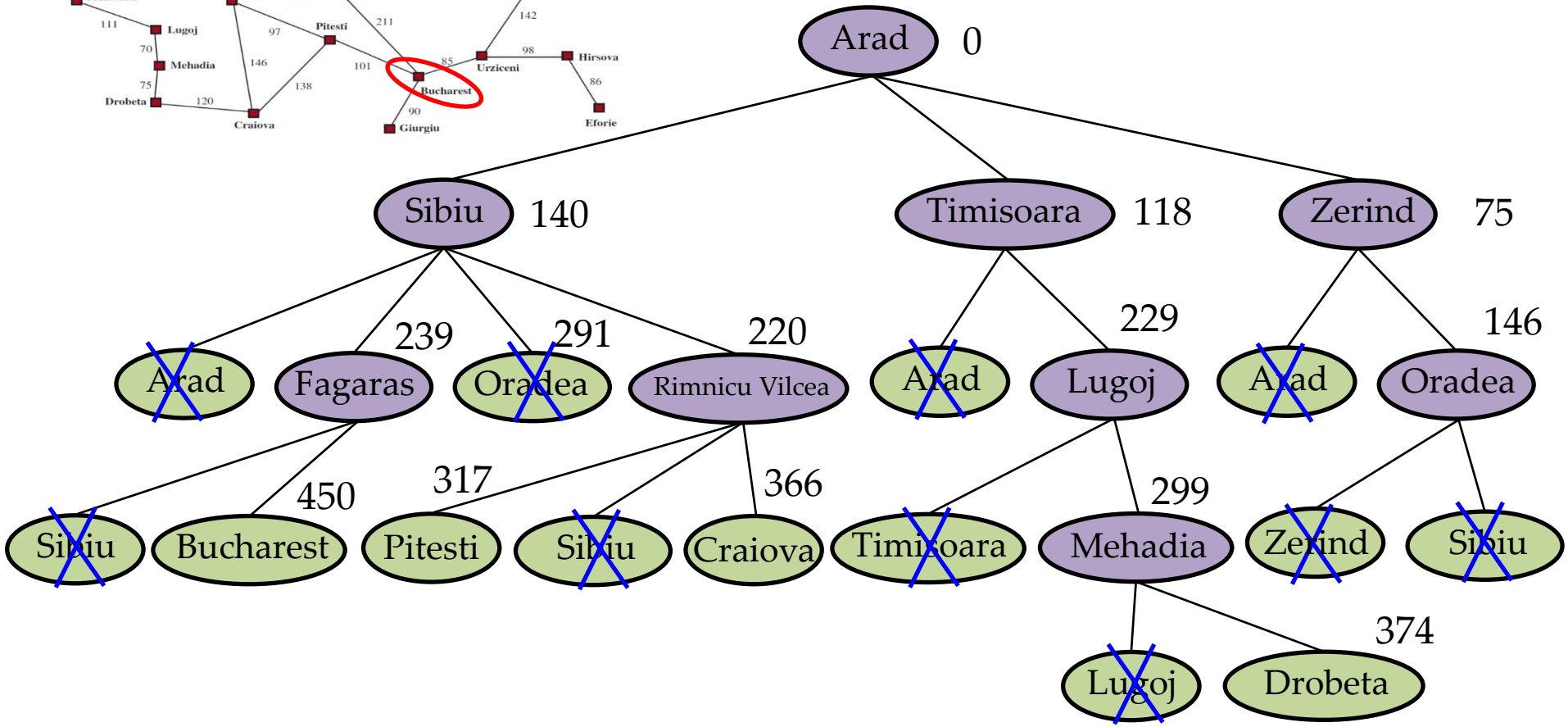
Uniform-cost search - example



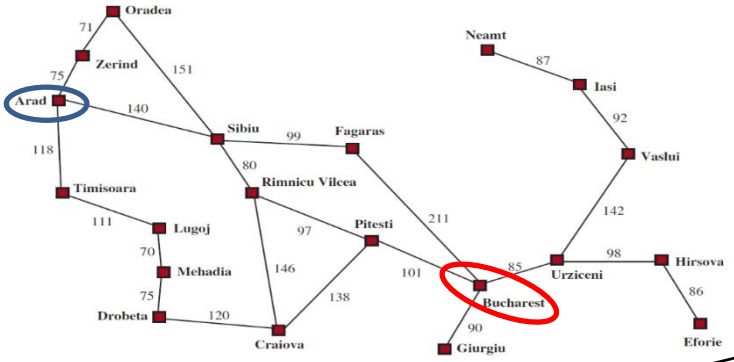


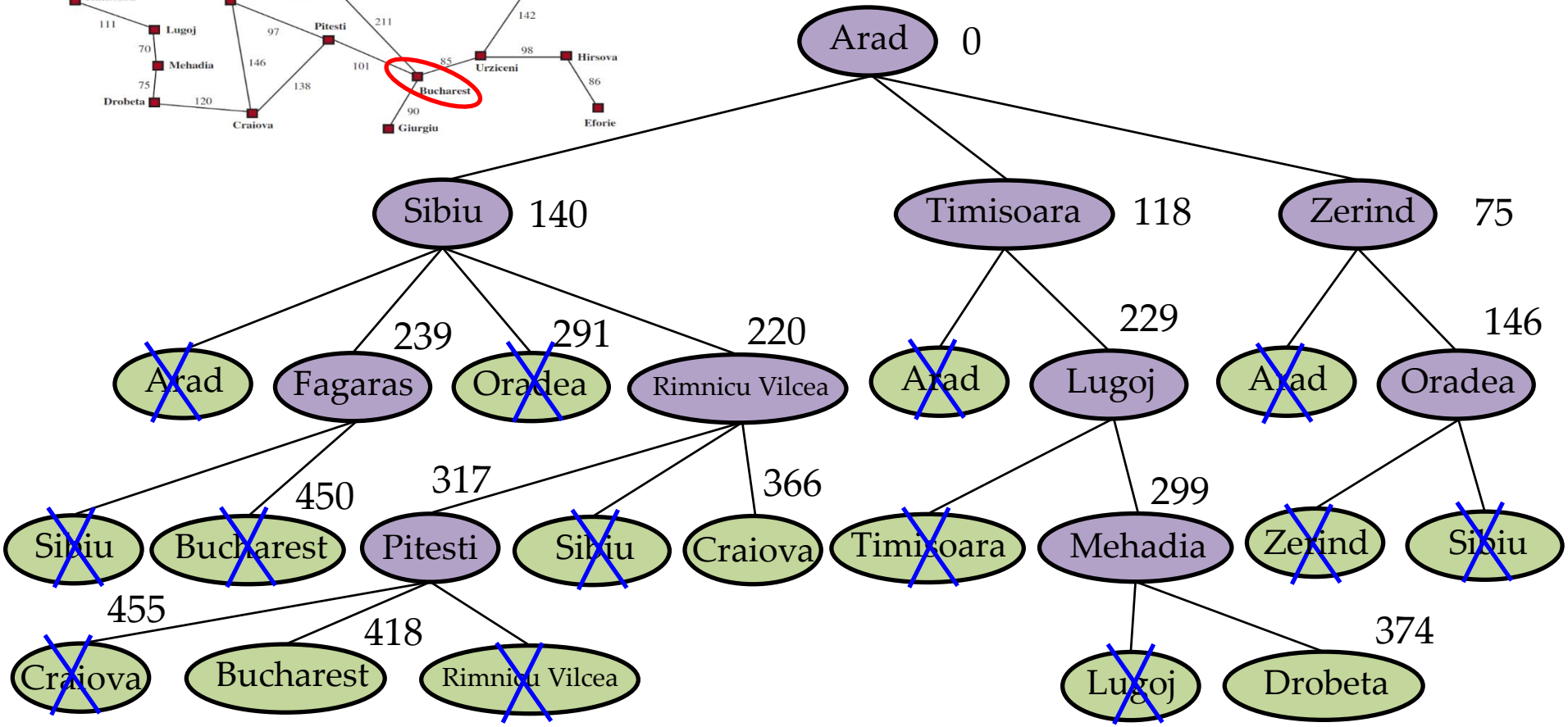
Uniform-cost search - example



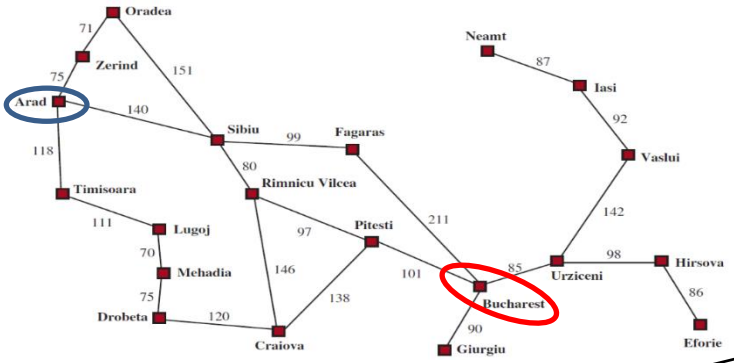


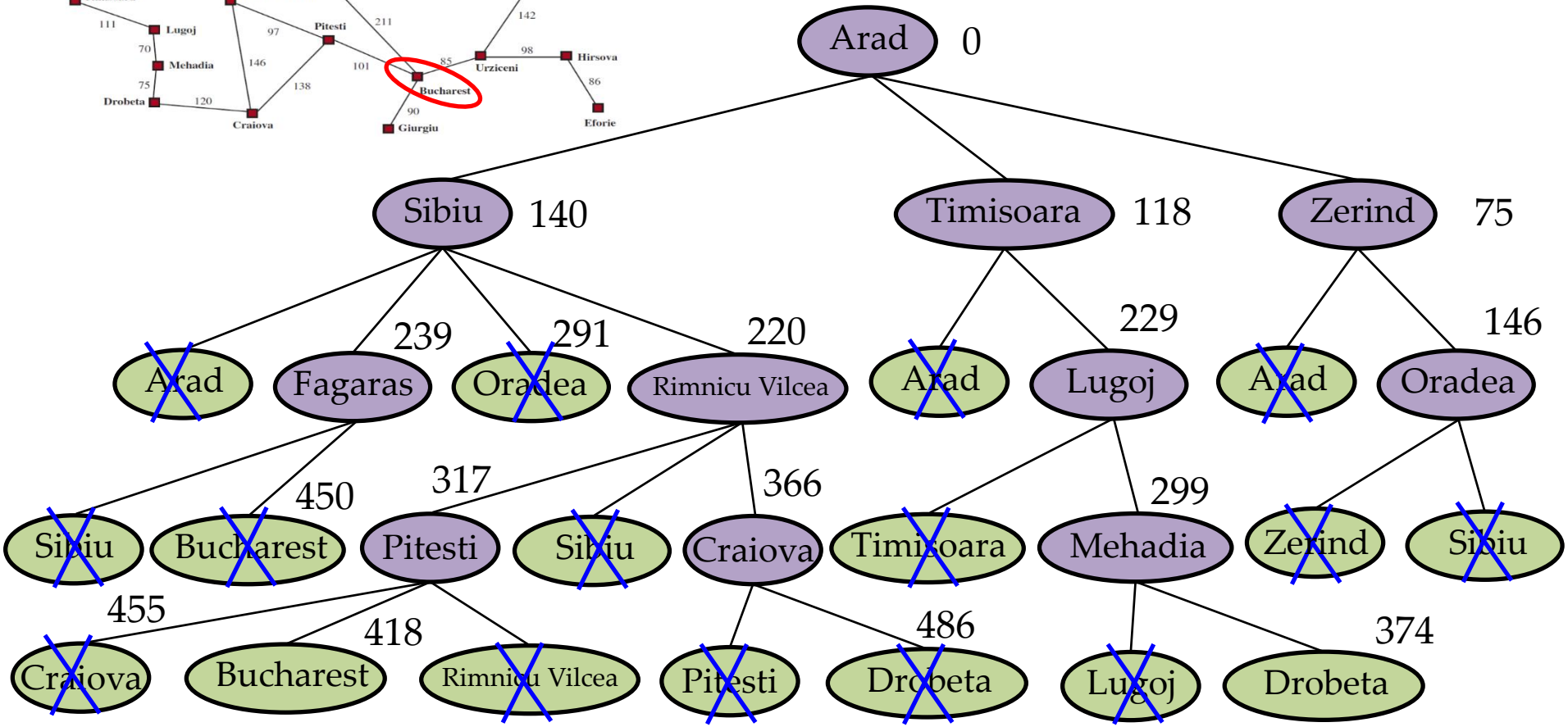
Uniform-cost search - example



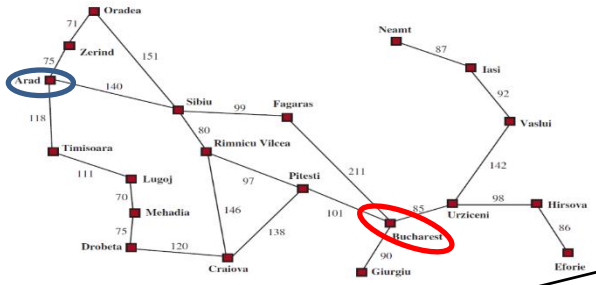


Uniform-cost search - example

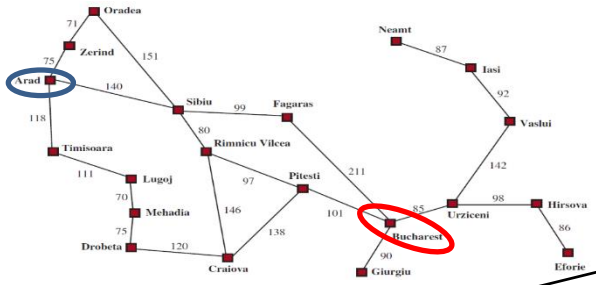




Uniform-cost search - example



Uniform-cost search - example



Dijstra's Algorithm for single source shortest path

Uniform-cost search - performance

Main idea: expand the node n with the lowest cost $g(n)$

- Complete

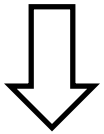
  if the cost of every step exceeds some small positive constant ϵ

- Optimal

  when a node n *is expanded, the optimal path to* n*.state has been found*

  Otherwise, there is another frontier node n' *on the optimal path to* n*.state, whose corresponding node is denoted as* n''

$$g(n') \leq g(n'') < g(n)$$



n' will be expanded before n, making a contradiction

Uniform-cost search - performance

Main idea: expand the node n with the lowest cost $g(n)$

- Complete

  if the cost of every step exceeds some small positive constant ϵ

- Optimal

  when a node n *is expanded, the optimal path to* n*.state has been found*

- Time complexity

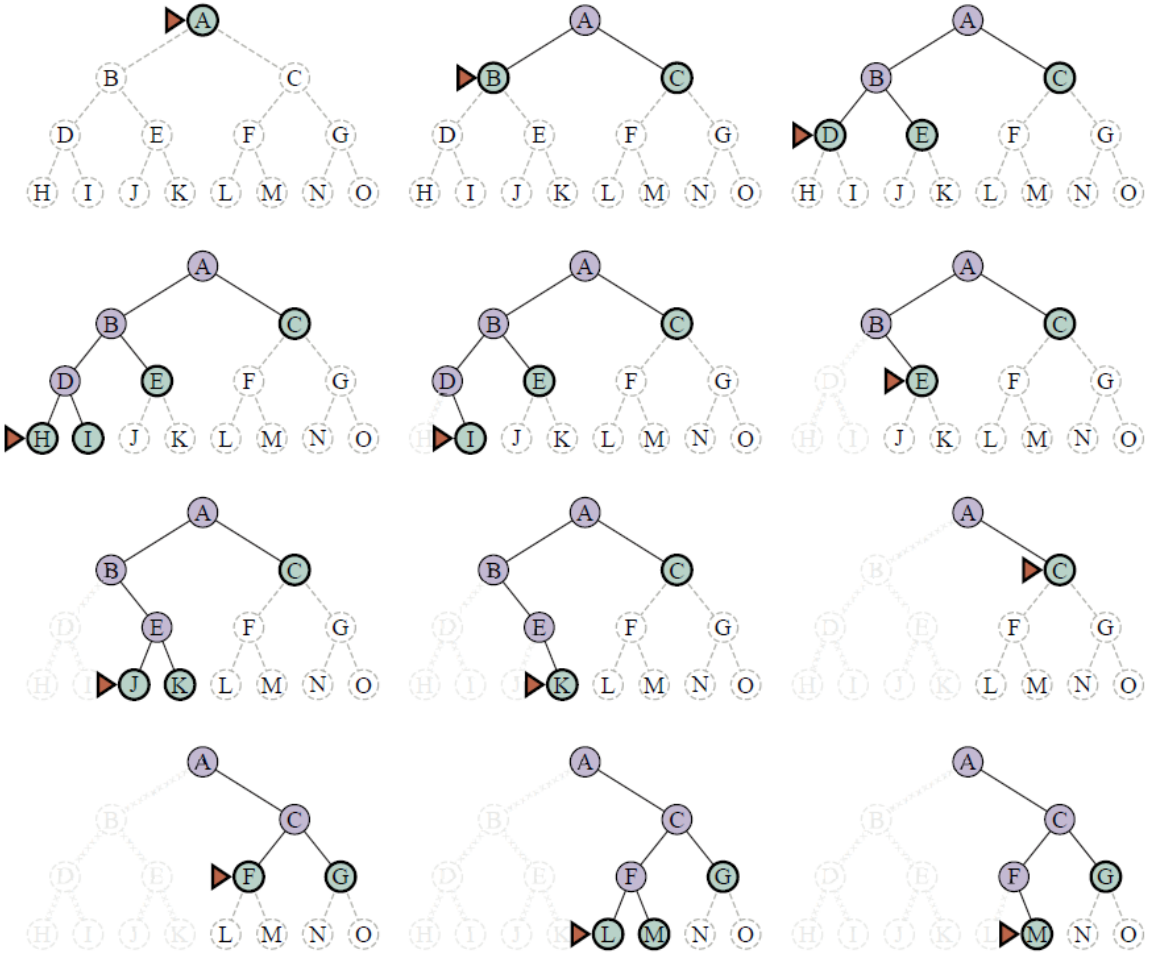
  $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, *where* C^* *is the cost of the optimal path*

- Space complexity

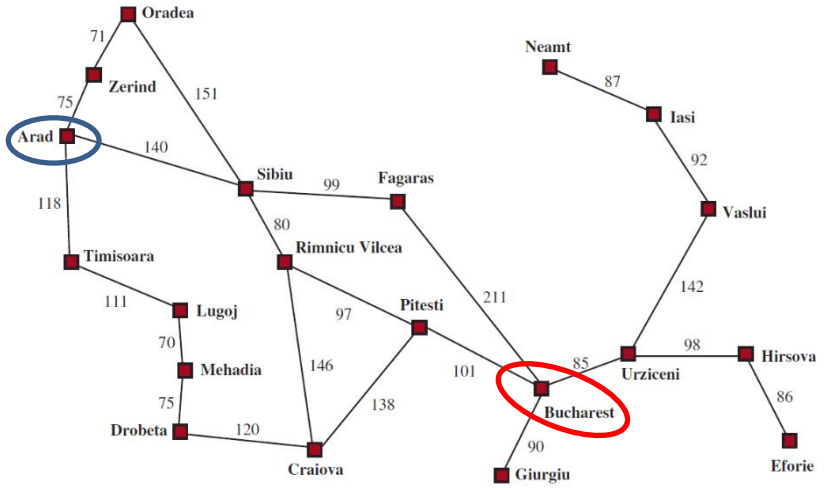
  $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, *where* C^* *is the cost of the optimal path*

Depth-first search

Main idea: expand the deepest node in the frontier

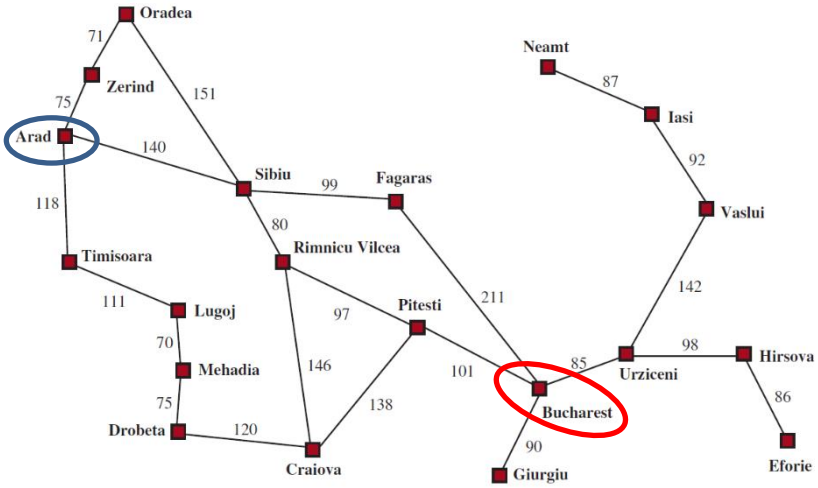


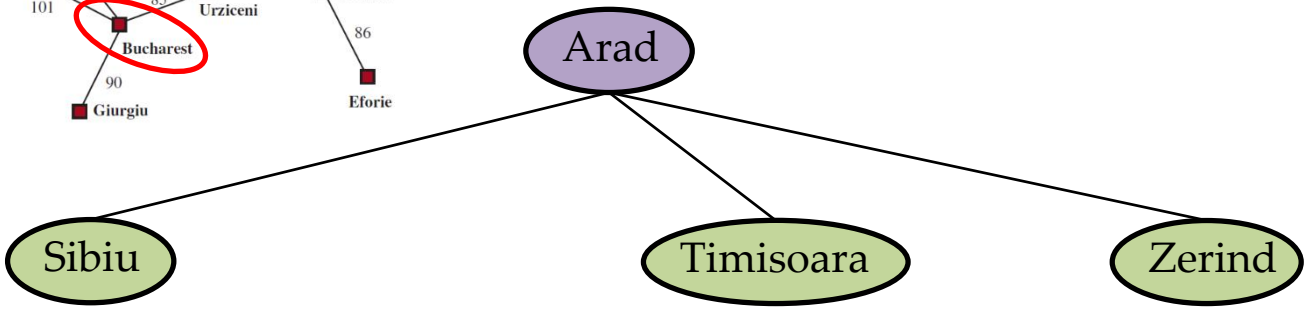
Depth-first search - example



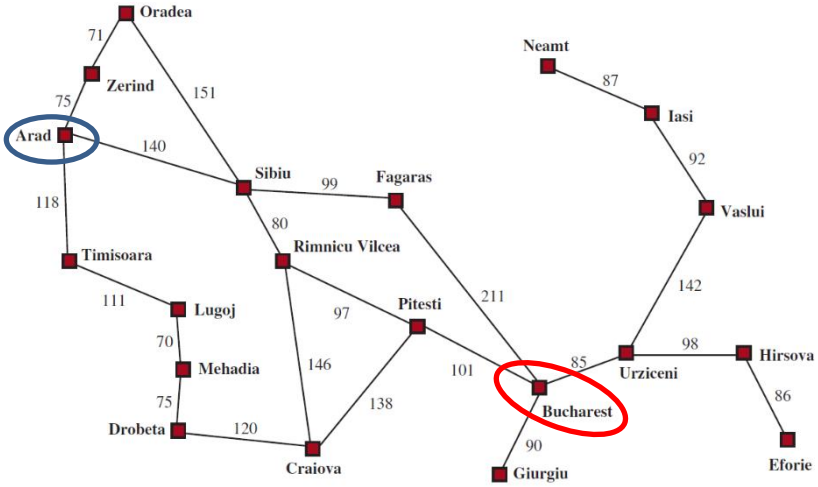
Arad

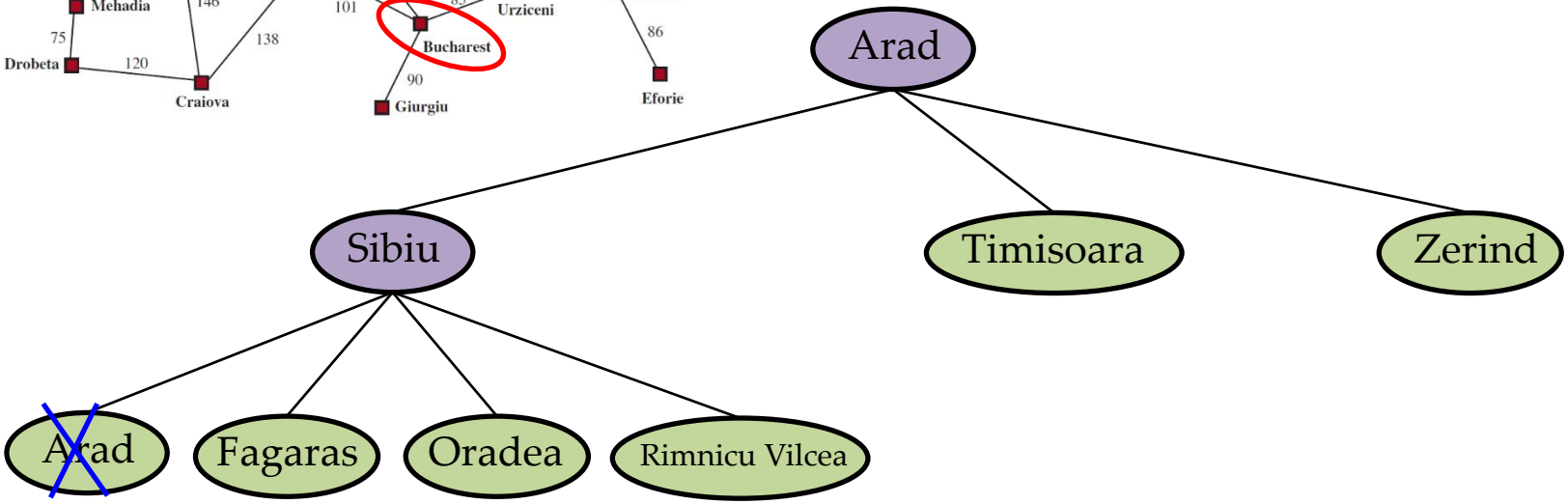
Depth-first search - example



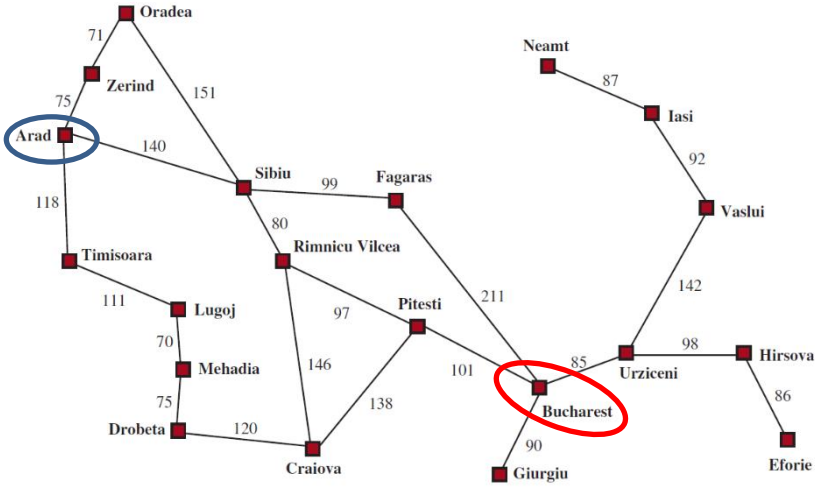


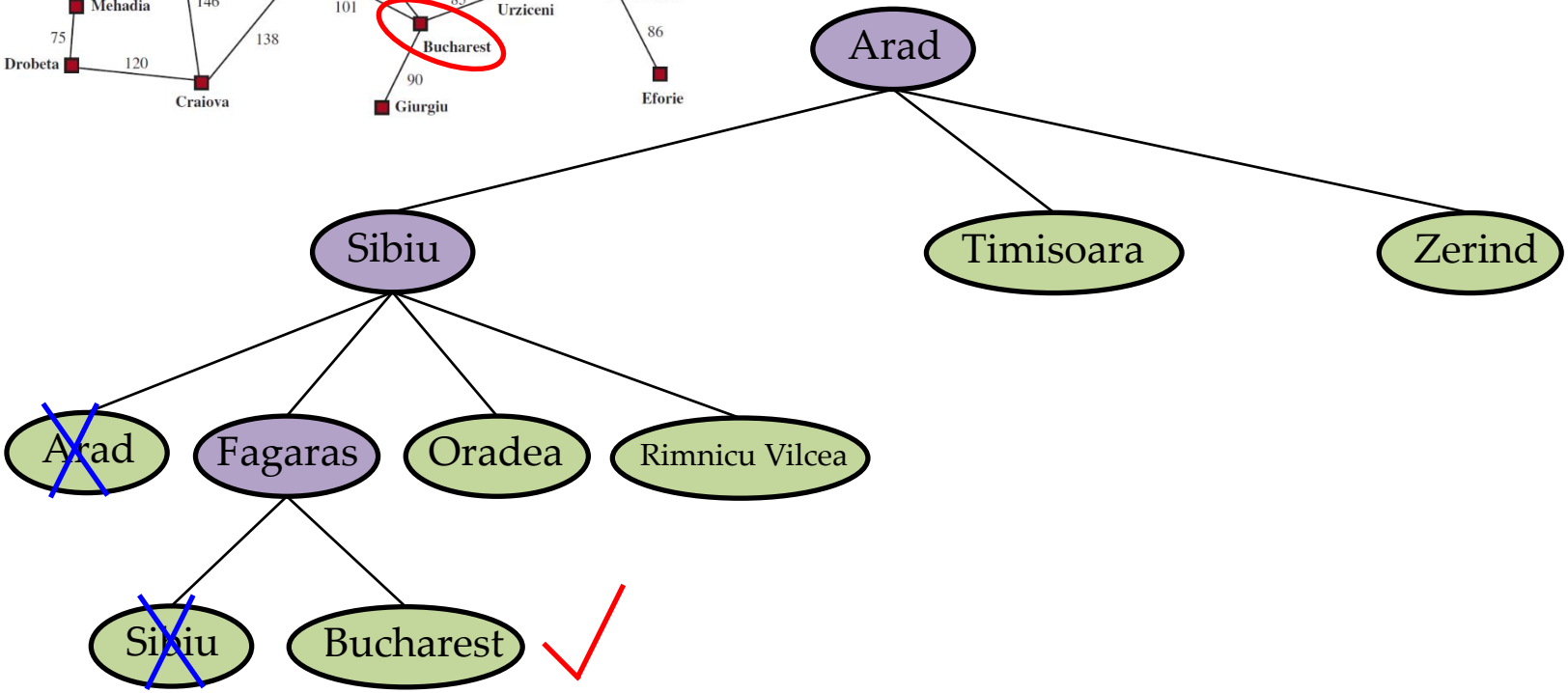
Depth-first search - example





Depth-first search - example





Depth-first search - performance

Main idea: expand the deepest node in the frontier

- Complete

  if using graph-search and the state space is finite

- Optimal ✗

- Time complexity

  $O(b^m)$, *where* m *is the maximum depth of any node, if using tree-search*

  the size of the state space, if using graph-search

- Space complexity

  $O(bm)$, *if using tree-search*

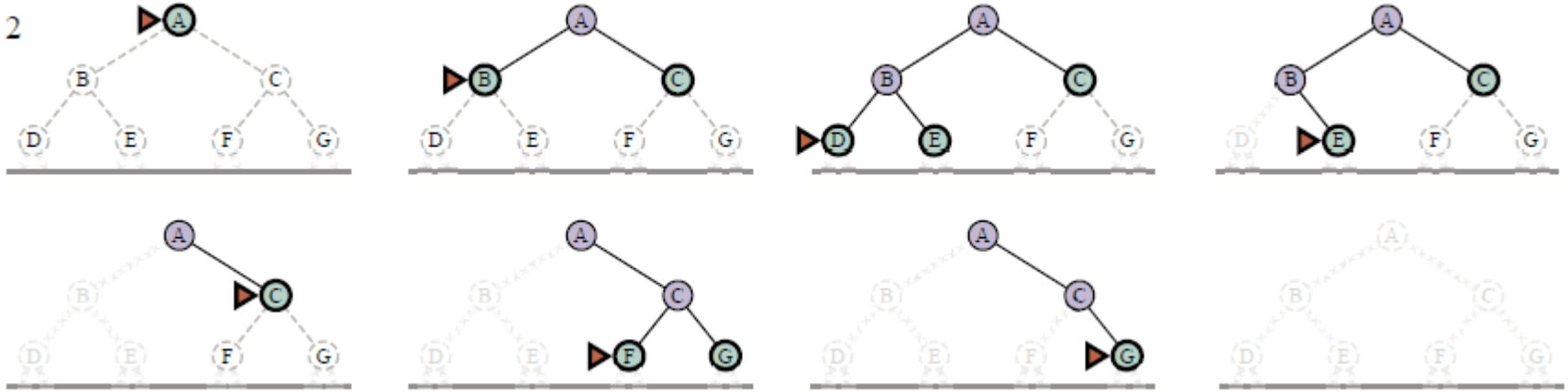
  the size of the state space, if using graph-search

Depth-limited search

Main idea: depth-first search with a depth limit l

limit: 1

limit: 2



Depth-limited search - performance

Main idea: depth-first search with a depth limit l

- Complete

  if $l \geq d$, the depth of the shallowest goal node

- Optimal

  if $l > d$, no; if $l = d$ and all actions have the same cost, yes

- Time complexity

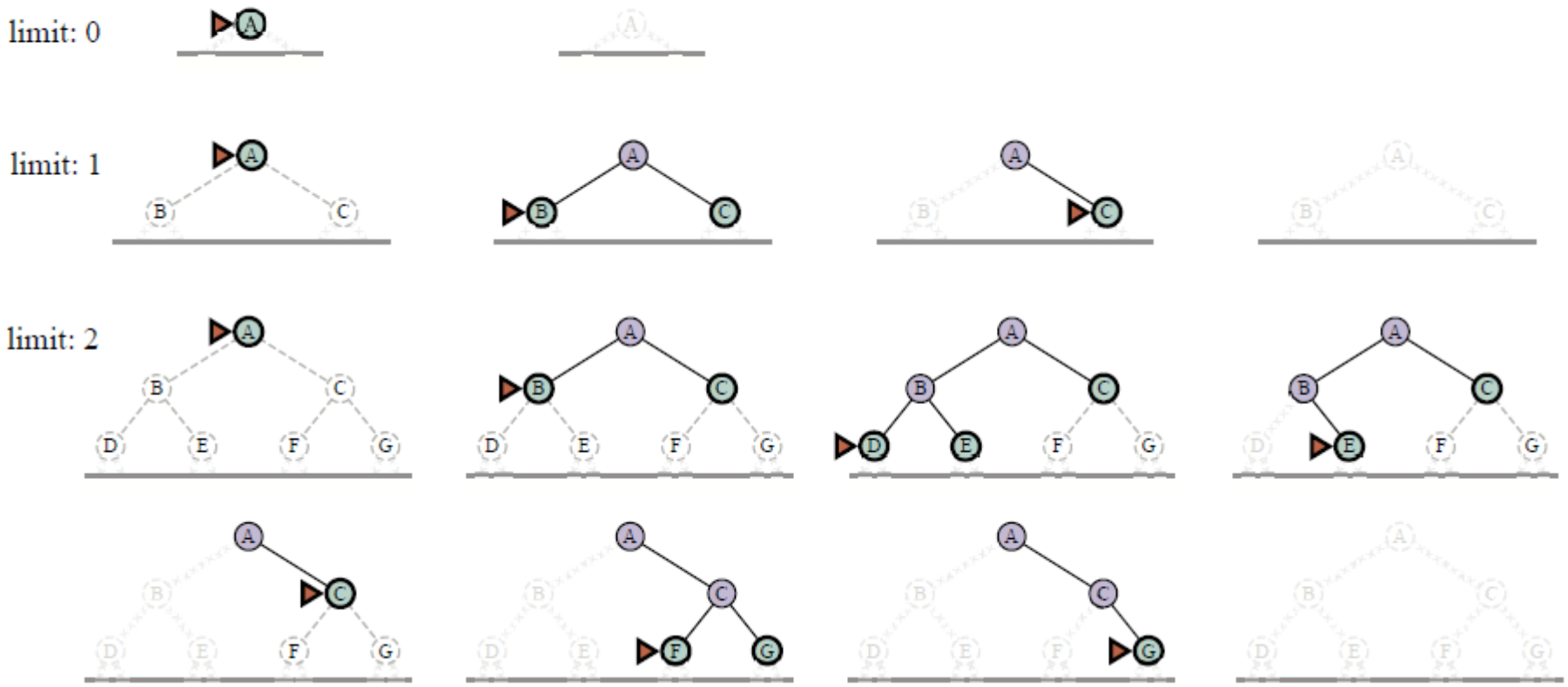
  $O(b^l)$

- Space complexity

  $O(bl)$, *if using tree-search*

Iterative deepening depth-first search

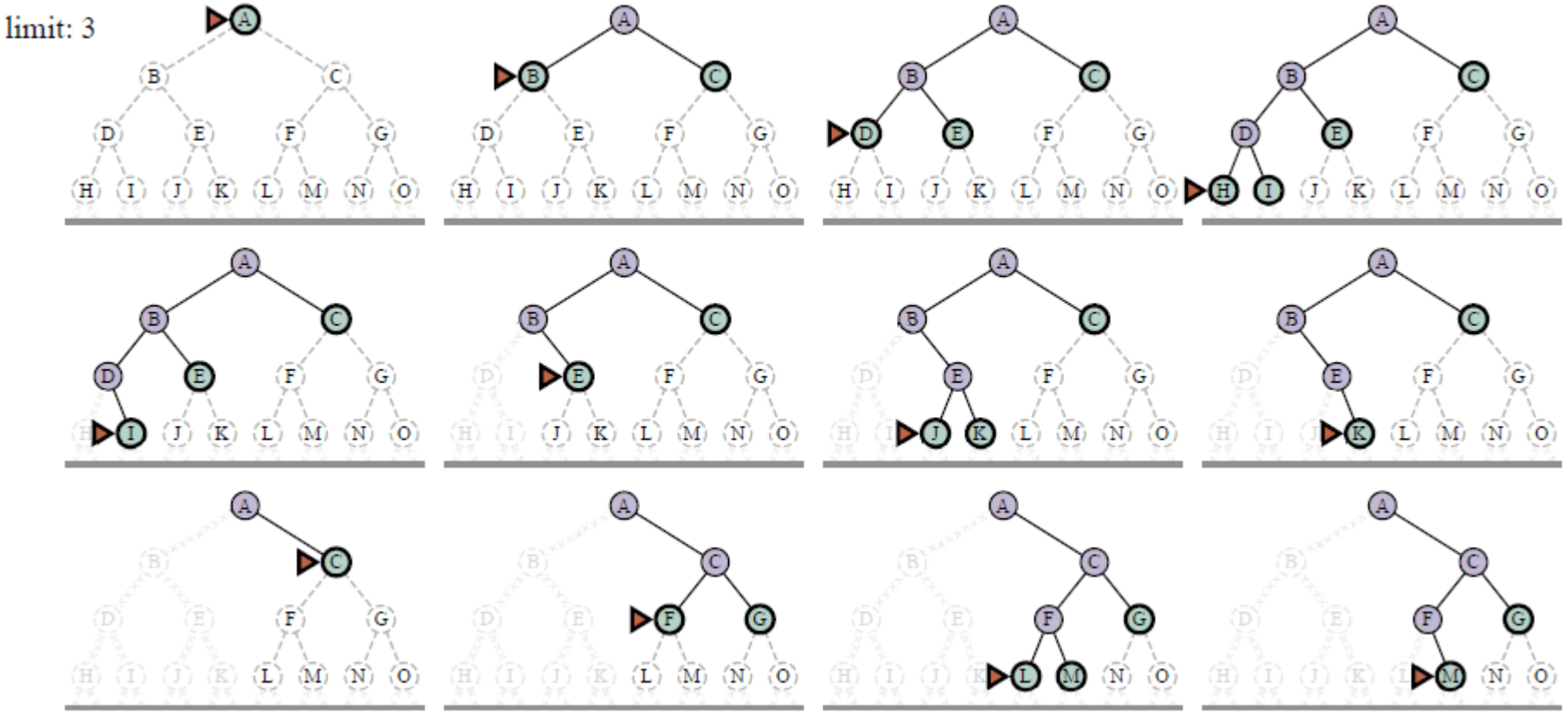
Main idea: iteratively apply depth-limited search with gradually increasing depth limit l



Iterative deepening depth-first search

Main idea: iteratively apply depth-limited search with gradually increasing depth limit l

Stop until $l = d$



Iterative deepening depth-first search – performance

Main idea: iteratively apply depth-limited search with gradually increasing depth limit l

Stop until $l = d$

- Complete

  if the depth d of the shallowest goal node is finite

- Optimal

  if all actions have the same cost

Combine the benefit of breadth-first search and depth-first search

- Time complexity

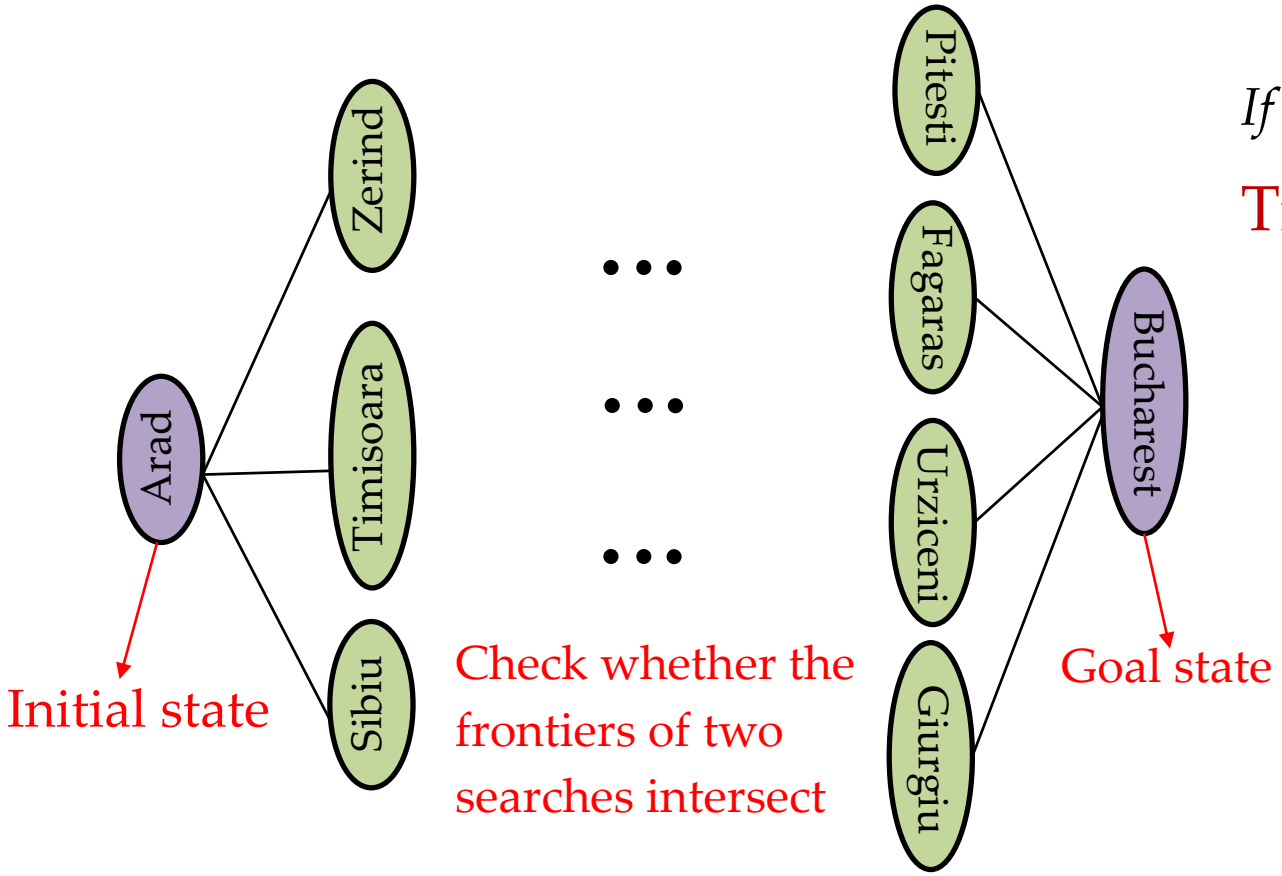
  $db + (d-1)b^2 + (d-2)b^3 + \cdots + (1)b^d = O(b^d)$

- Space complexity

  $O(bd)$, *if using tree-search*

Bidirectional search

Main idea: run two simultaneous searches, one forward from the initial state and the other backward from the goal



If using breadth-first search

Time complexity

$O(b^d)$

$O(b^{d/2})$

Not easy to use

Performance comparison

Tree-search versions

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

Summary

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening depth-first search
- Bidirectional search

Uninformed search

No additional information beyond the problem definition

References

- S. J. Russell and P. Norvig. Artificial Intelligence: A Modern Approach. Chapter 3.4, Third edition.