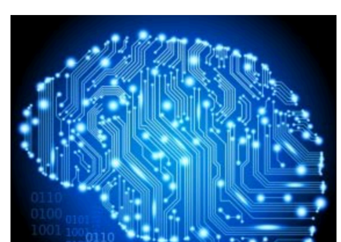
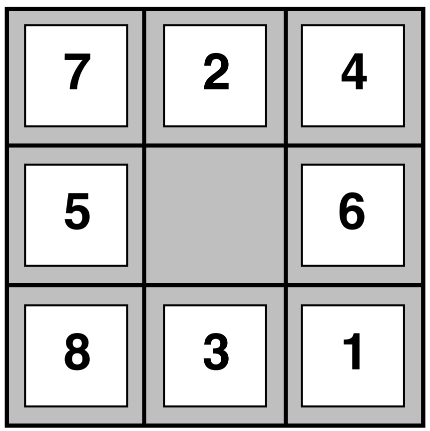
Lecture 2: Search 1

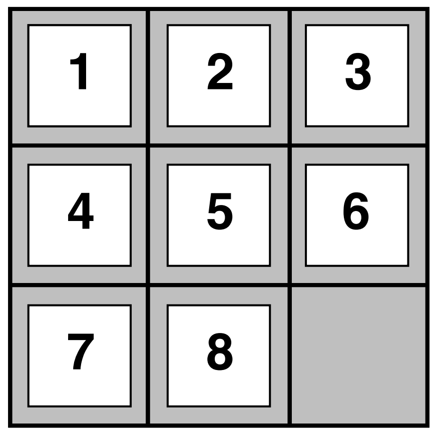
http://lamda.nju.edu.cn/yuy/course_ai17.ashx



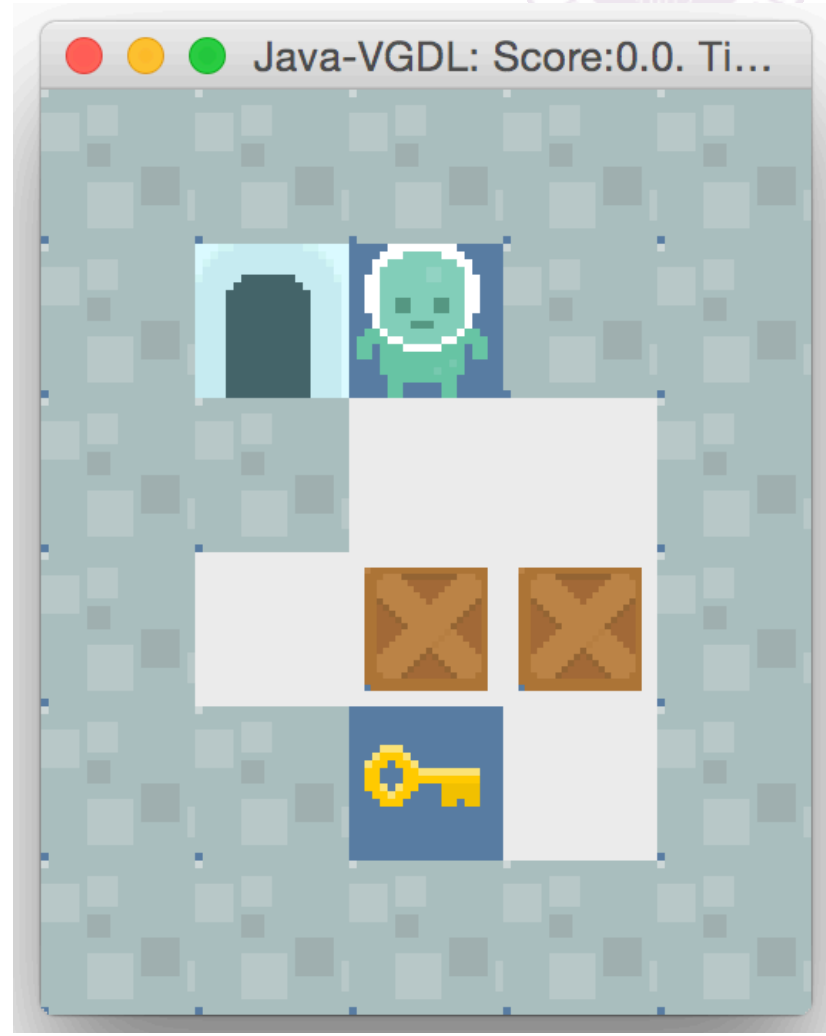
Problem in the lecture



Start State



Goal State



Agent that searches

a *world* is a set of *states*

we first consider a world using atomic representation

atomic representation: state is the basic unit
states that can be factored will be considered later

the big O representation: e.g. O(n)
NP-hardness and NP-completeness

Example: Romania

On holiday in Romania; currently in Arad.
Flight leaves tomorrow from Bucharest

Formulate goal:

be in Bucharest

Formulate problem:

states: various cities

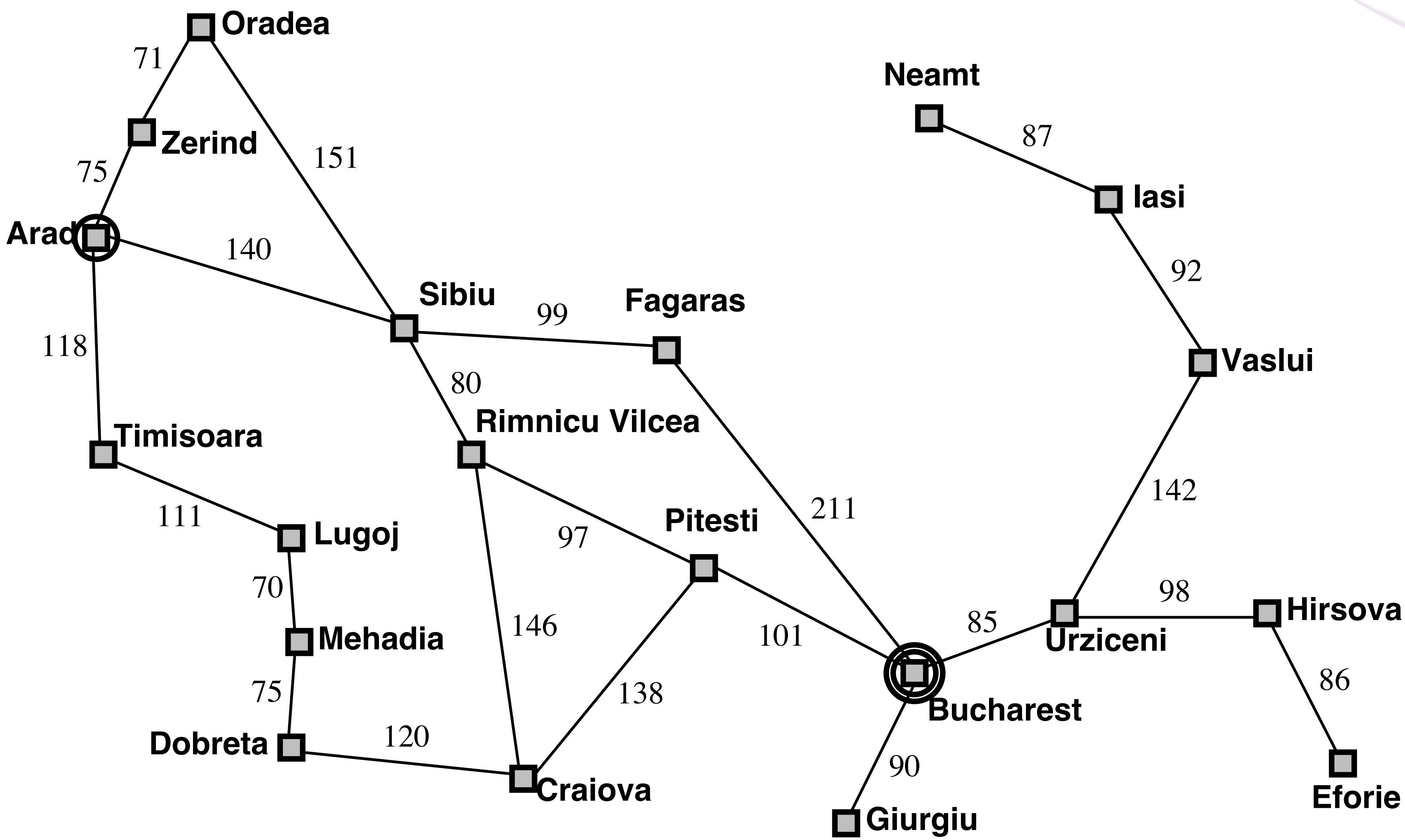
actions: drive between cities

Find solution:

sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania

map:



Problems

A problem is defined by 5 components:

initial state

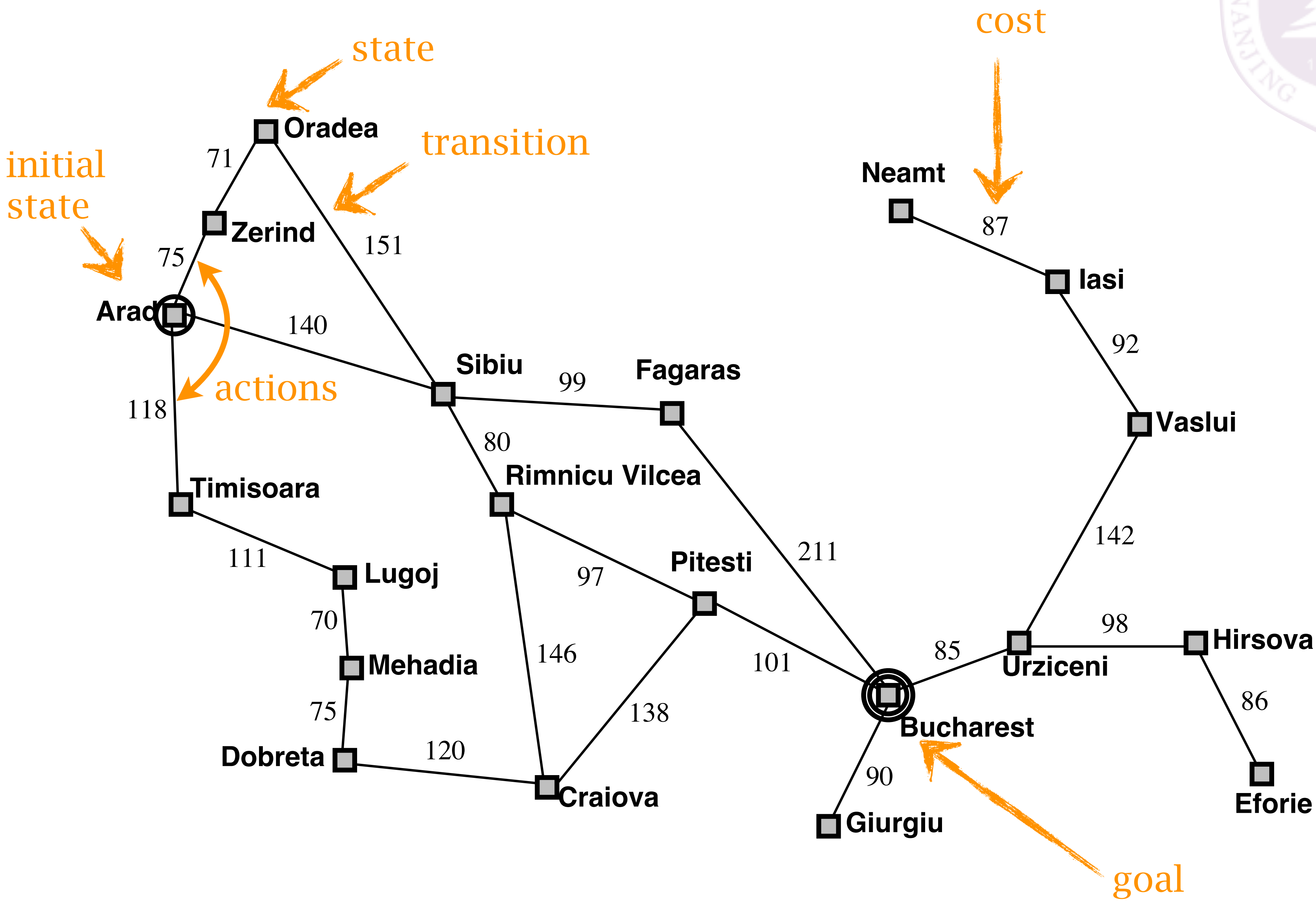
possible actions (and state associated actions)

transition model
    taking an action will cause a state change

goal test
    judge if the goal state is found

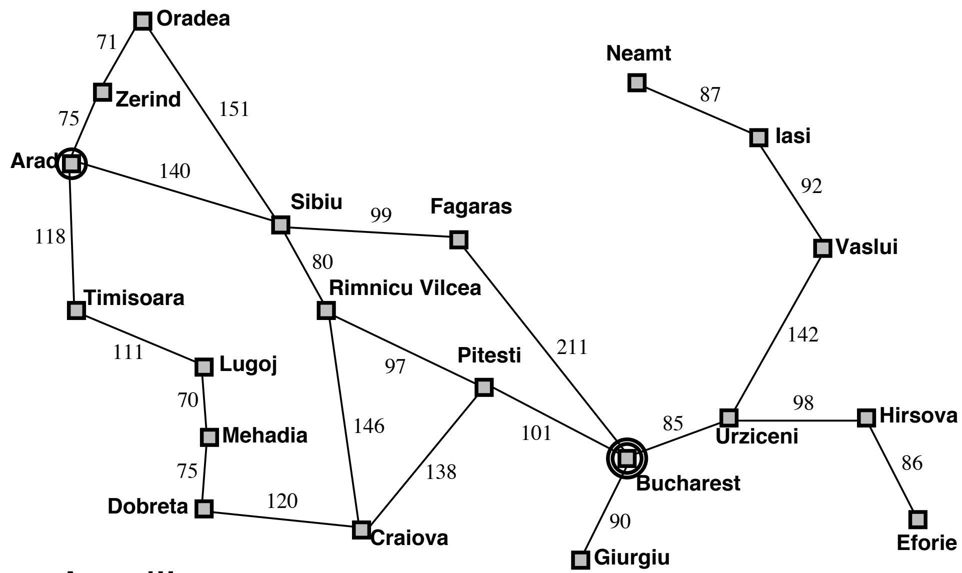
path cost
    constitute the cost of a solution

Problems



state
cost
Oradea
transition
initial state
Neamt
71
87
Zerind
151
75
Iasi
Arad
140
92
Sibiu
99
Fagaras
actions
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
111
211
Pitesti
Lugoj
97
70
98
Hirsova
85
Urziceni
Mehadia
146
101
86
138
Bucharest
75
Dobreta
120
90
Craiova
Giurgiu
Eforie
goal

Problems



initial state   e.g., "at Arad"

successor function $S(x)$ = set of action–state pairs

e.g., $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind \rangle, \ldots\}$

<-- transition

goal test, can be

explicit, e.g., x = "at Bucharest"

implicit, e.g., $NoDirt(x)$

path cost (additive)

e.g., sum of distances, number of actions executed, etc.

$c(x, a, y)$ is the step cost, assumed to be ≥ 0

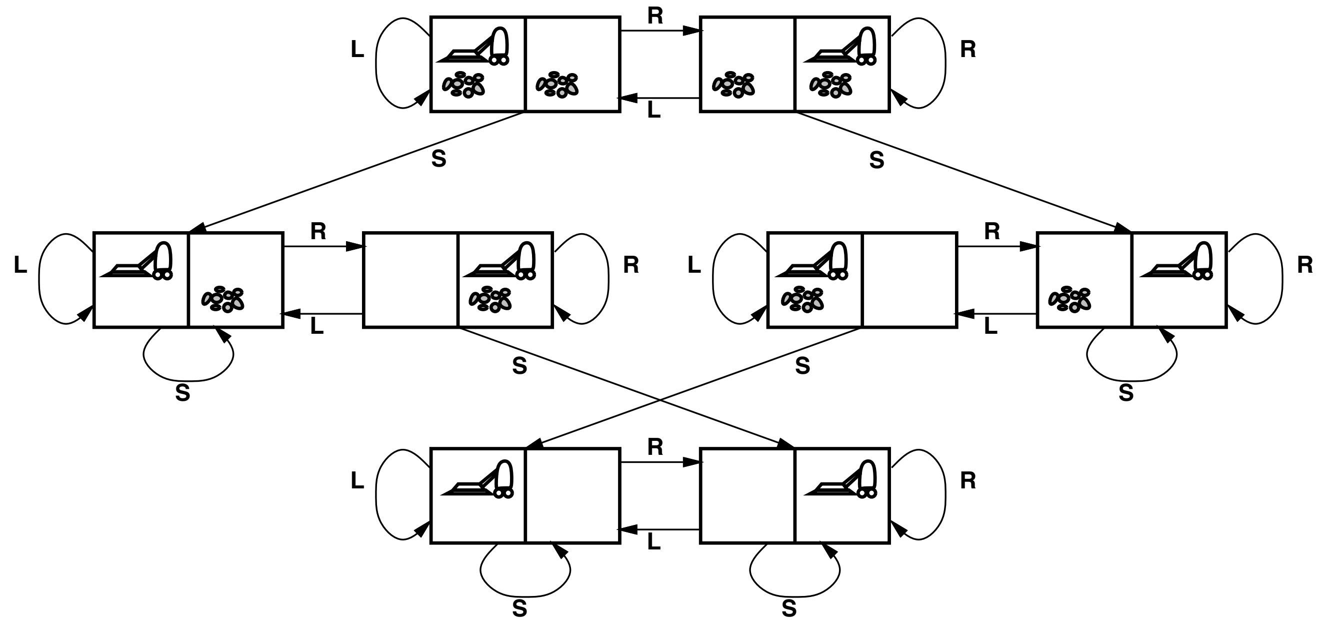
A solution is a sequence of actions
leading from the initial state to a goal state

Problems

we assume

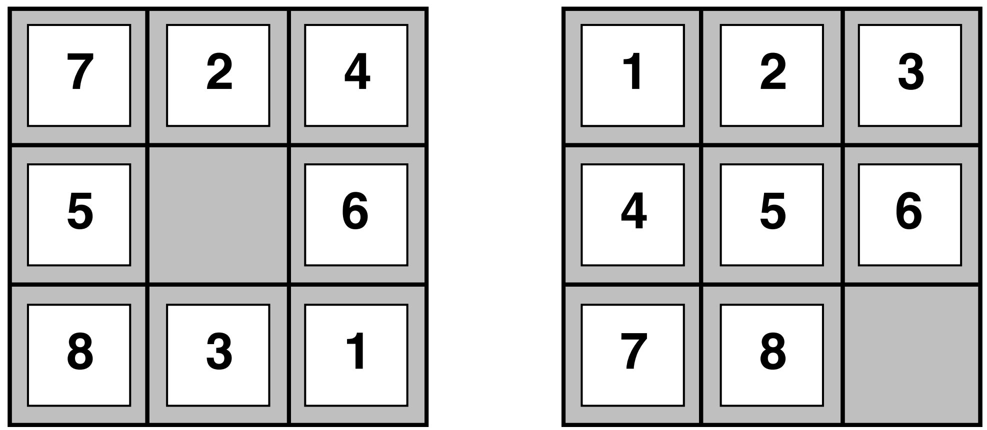
- observable states (a seen state is accurate)
    - in partial observable case, states are not accurate
- discrete states
    - there are also continuous state spaces
- deterministic transition
    - there could be stochastic transitions

Example: vacuum world



states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: $Left$, $Right$, $Suck$, $NoOp$
goal test??: no dirt
path cost??: 1 per action (0 for $NoOp$)

Example: 8-puzzle



Start State **Goal State**

states??: integer locations of tiles (ignore intermediate positions)
actions??: move blank left, right, up, down (ignore unjamming etc.)
goal test??: $=$ goal state (given)
path cost??: 1 per move

[Note: optimal solution of n-Puzzle family is NP-hard]

Search Algorithms on Graphs

Tree search

1. start from the initial state
2. expand the current state

essence of search: following up one option now and putting the others aside

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

all search algorithms share this tree search structure
they vary primarily according to how they choose which state to expand --- the so-called search strategy

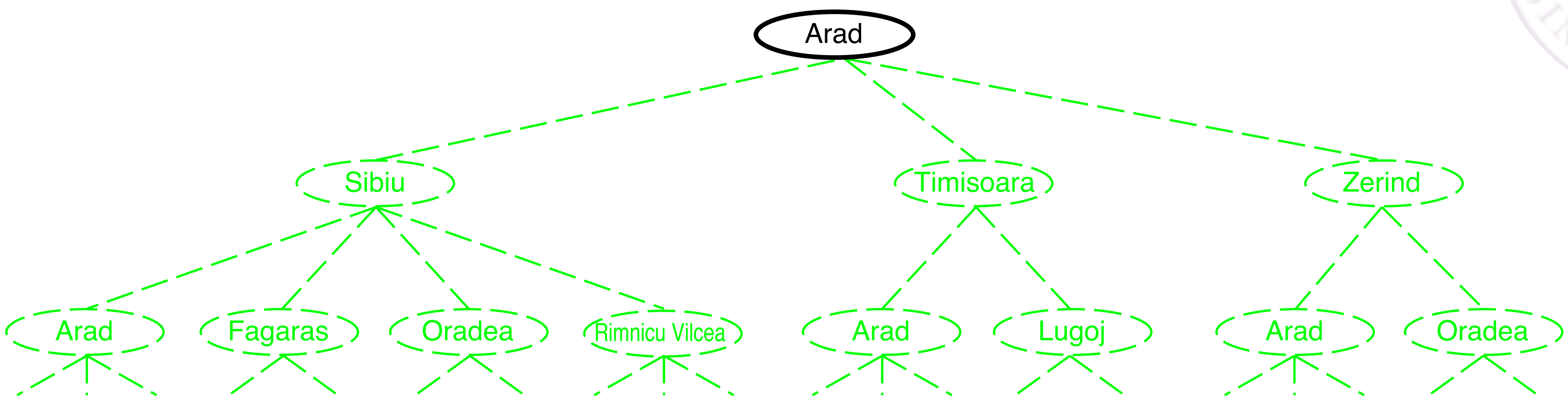
General tree search

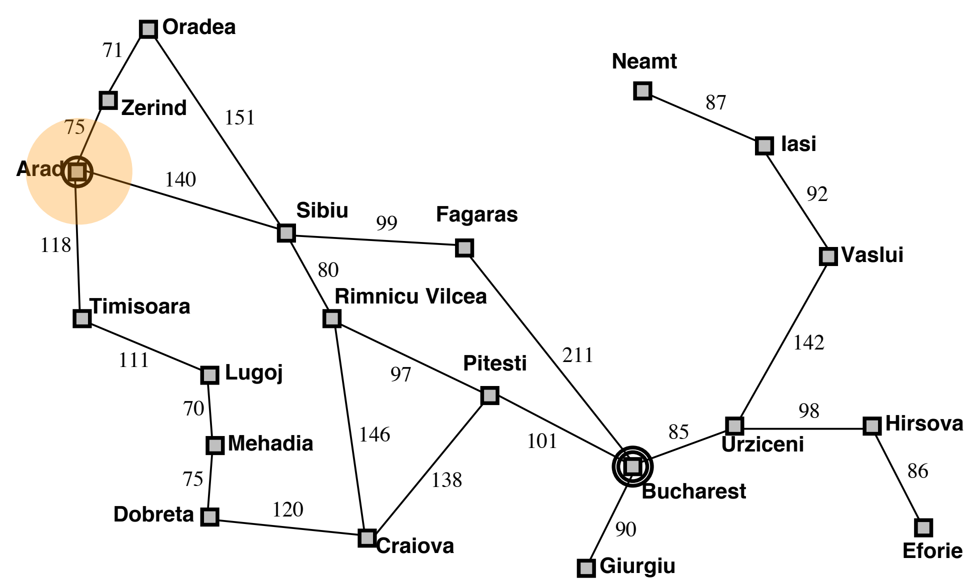
```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE(node)) then return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

note the time of goal-test: expanding time not generating time

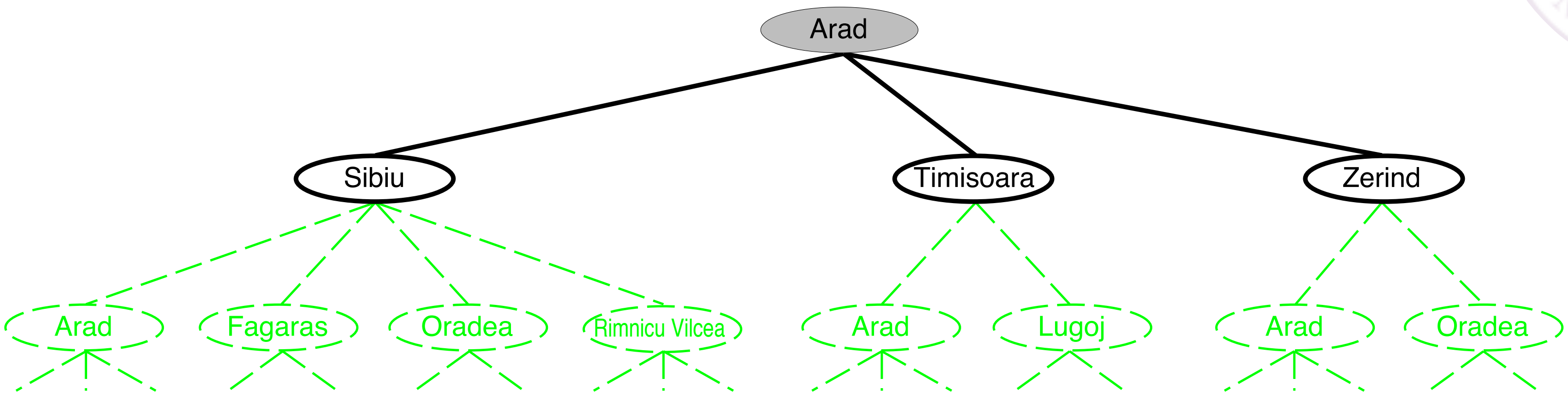
```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

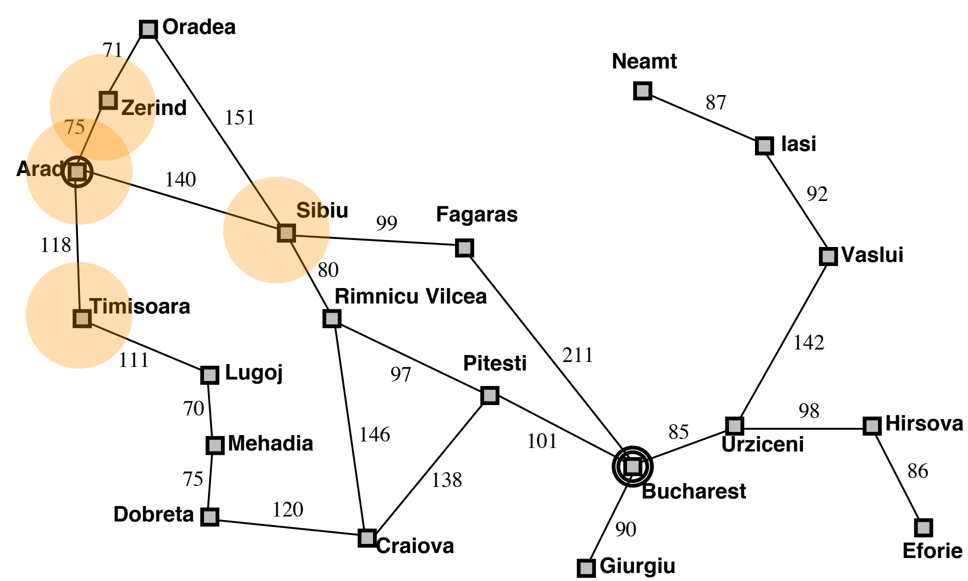
Example


Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

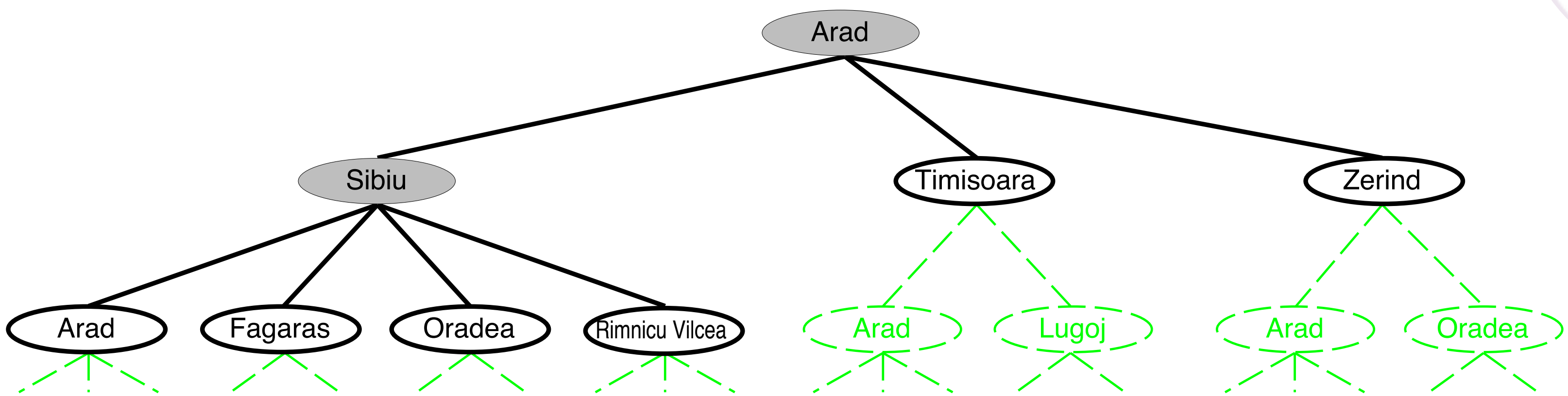

Oradea
Zerind
Arad
Timisoara
Lugoj
Mehadia
Dobreta
Sibiu
Fagaras
Rimnicu Vilcea
Pitesti
Craiova
Bucharest
Giurgiu
Urziceni
Hirsova
Eforie
Vaslui
Iasi
Neamt
71
75
151
140
118
111
70
75
120
99
80
97
146
138
101
211
85
90
98
86
142
92
87

Example

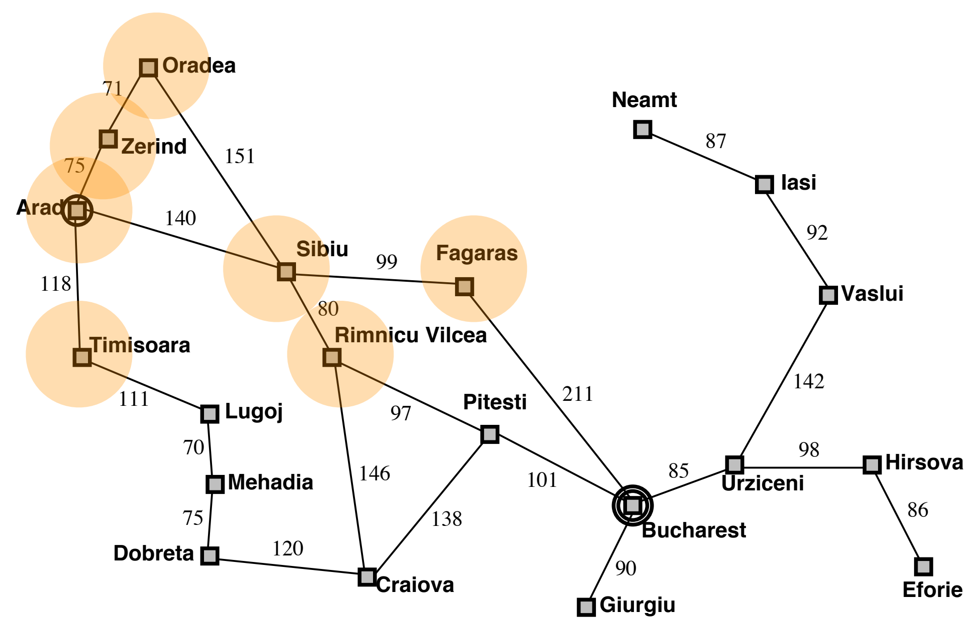

Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea


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

Example

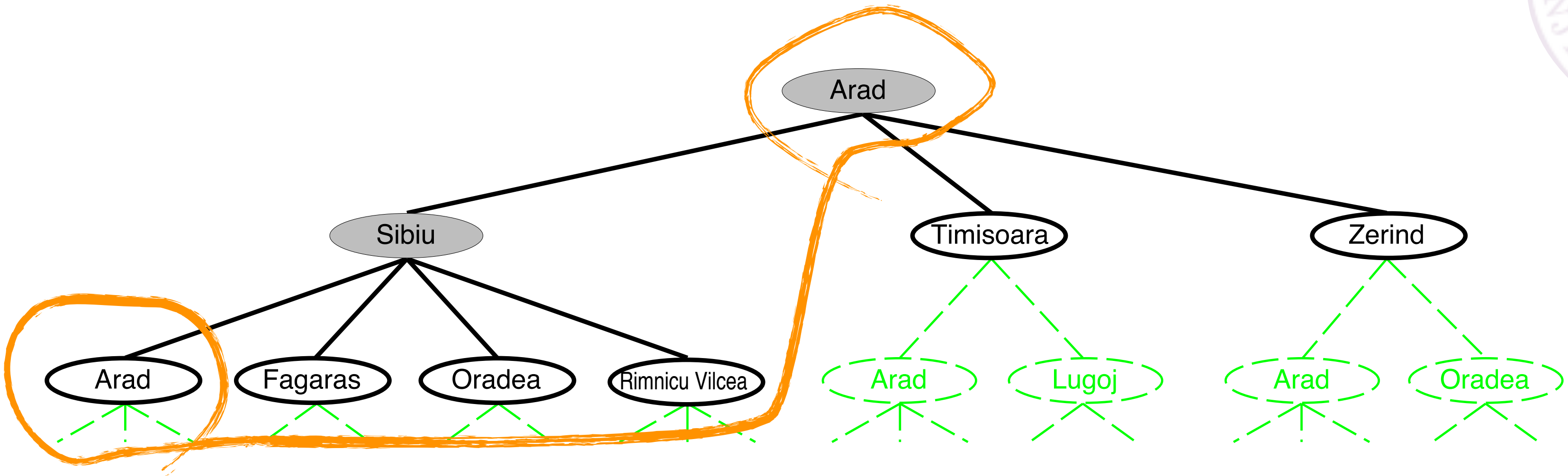


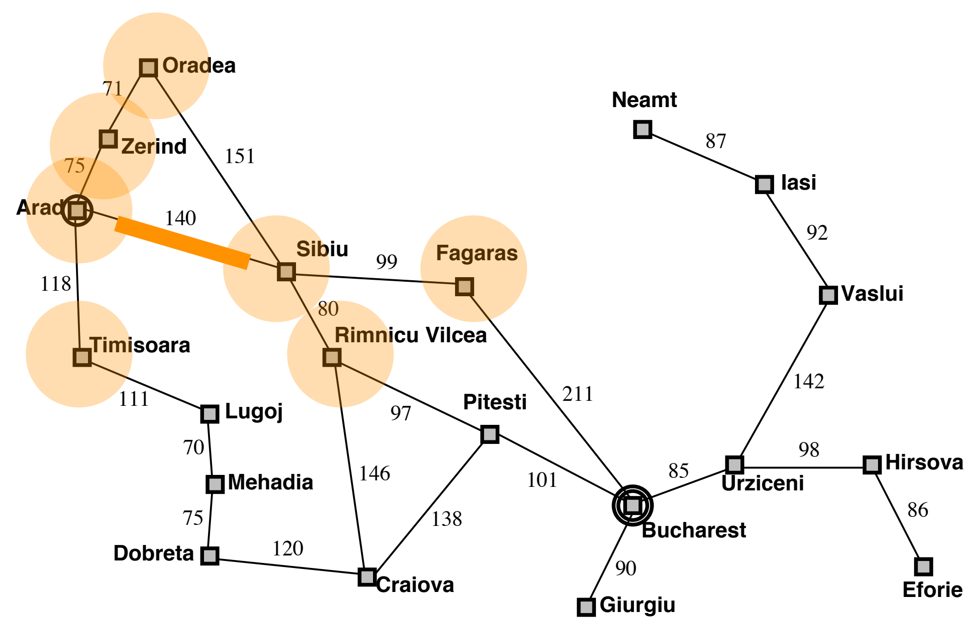
Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea



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

Example


Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea


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

Graph search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    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
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

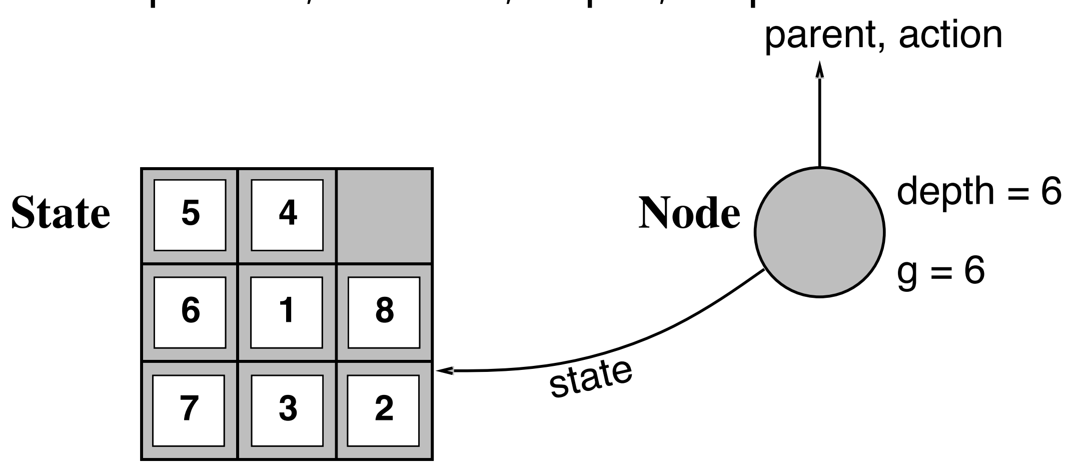
```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node])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

Graph separation property

the frontier (expandable leaf nodes) separates the visited and the unexplored nodes

State v.s. node

A state is a (representation of) a physical configuration
A node is a data structure constituting part of a search tree
includes parent, children, depth, path cost $g(x)$
States do not have parents, children, depth, or path cost!



The EXPAND function creates new nodes, filling in the various fields and using the SUCCESSORFN of the problem to create the corresponding states.

Search strategies

A strategy is defined by picking the **order of node expansion**

Strategies are evaluated along the following dimensions:

completeness—does it always find a solution if one exists?
time complexity—number of nodes generated/expanded
space complexity—maximum number of nodes in memory
optimality—does it always find a least-cost solution?

Time and space complexity are measured in terms of

b—maximum branching factor of the search tree
d—depth of the least-cost solution
m—maximum depth of the state space (may be ∞)

Uninformed Search Strategies

Uninformed strategies use only the information available in the problem definition

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

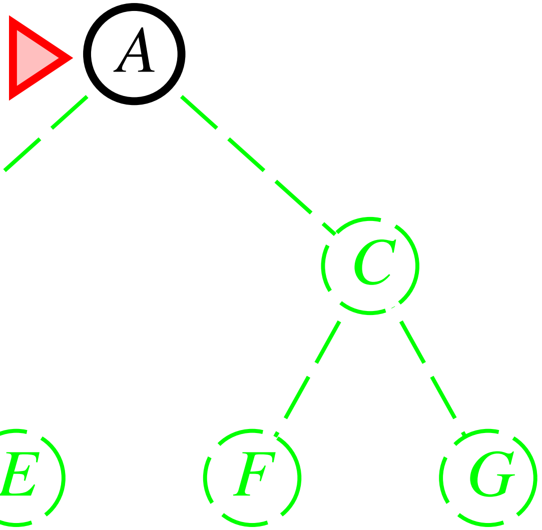
Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

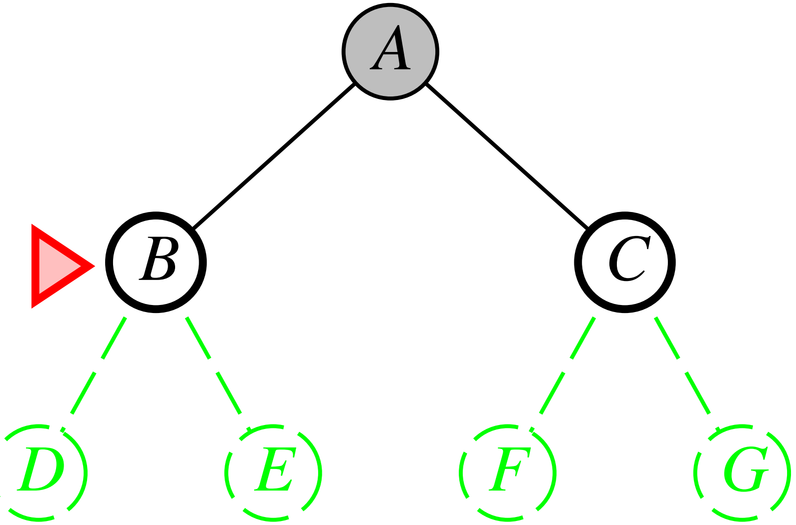


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

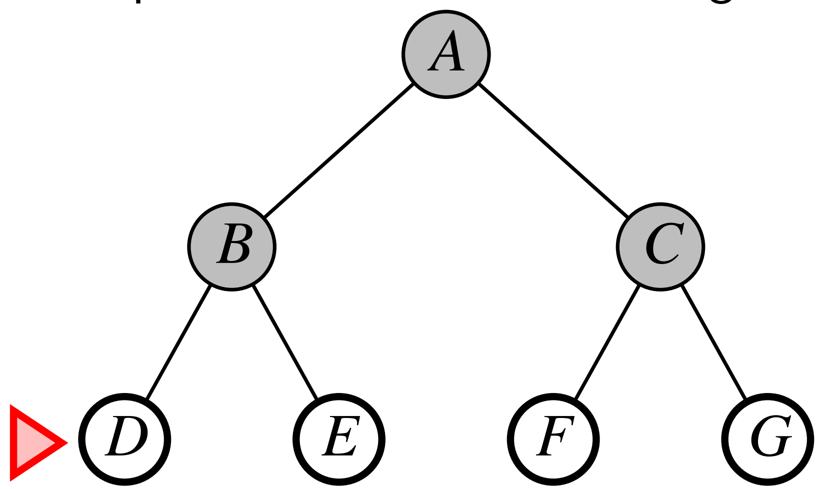


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end



Properties

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? Yes (if cost $= 1$ per step); not optimal in general

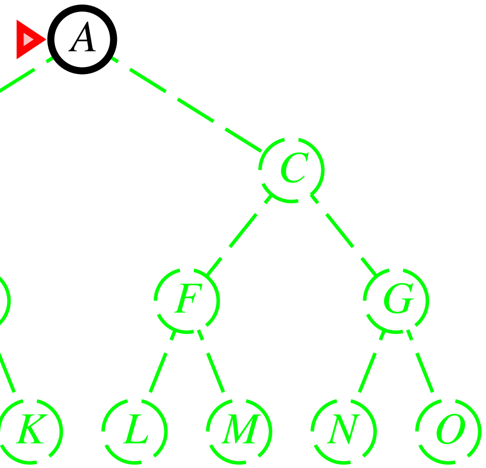
Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs = 8640GB.

Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

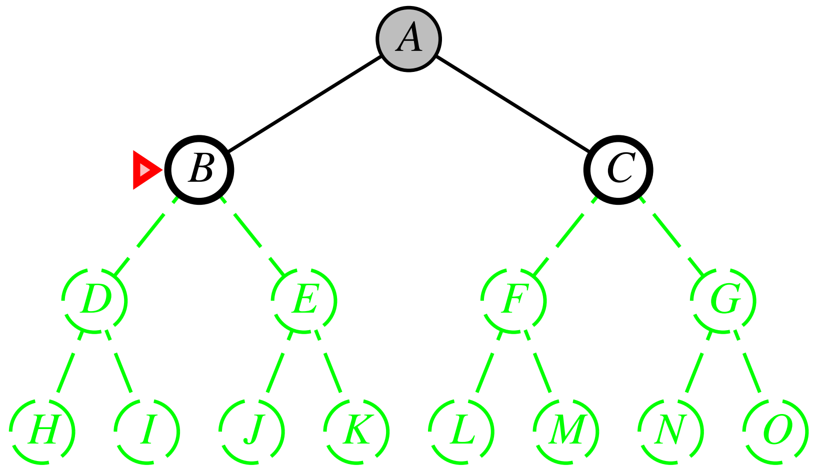


Depth-first search

Expand deepest unexpanded node

Implementation:

$fringe$ = LIFO queue, i.e., put successors at front

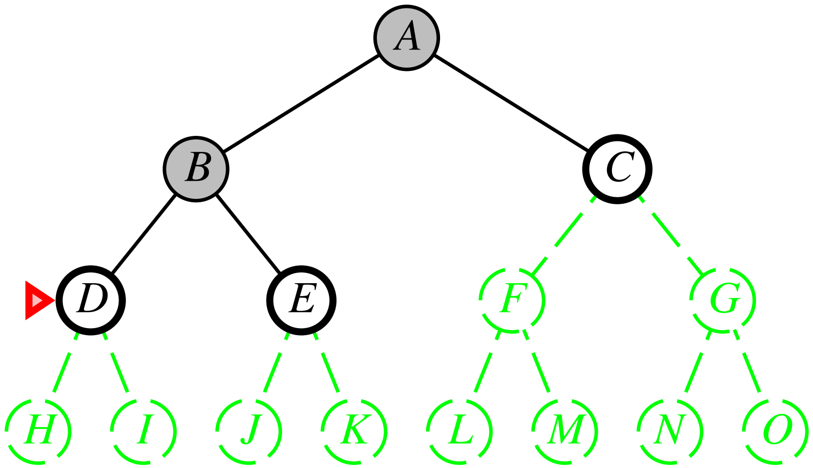


Depth-first search

Expand deepest unexpanded node

Implementation:

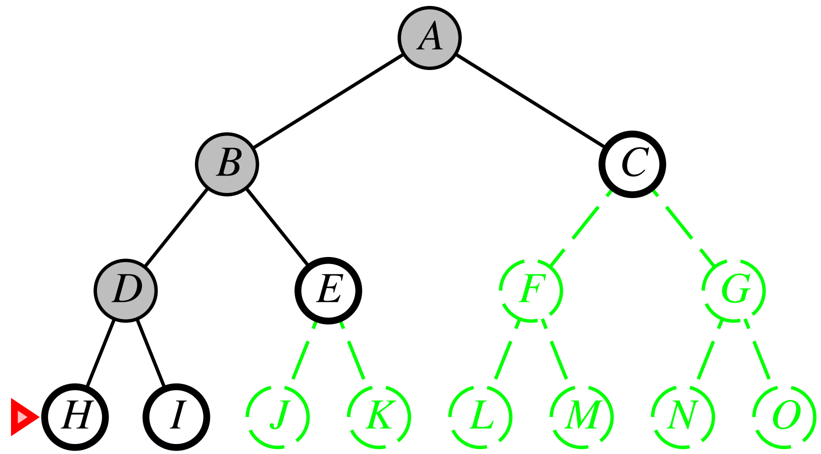
$fringe$ = LIFO queue, i.e., put successors at front



Depth-first search

Implementation:

$fringe$ = LIFO queue, i.e., put successors at front

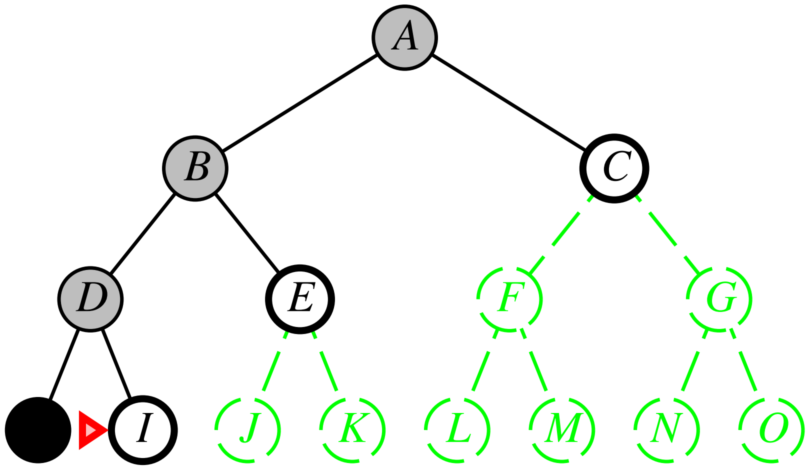


Depth-first search

Expand deepest unexpanded node

Implementation:

$fringe$ = LIFO queue, i.e., put successors at front

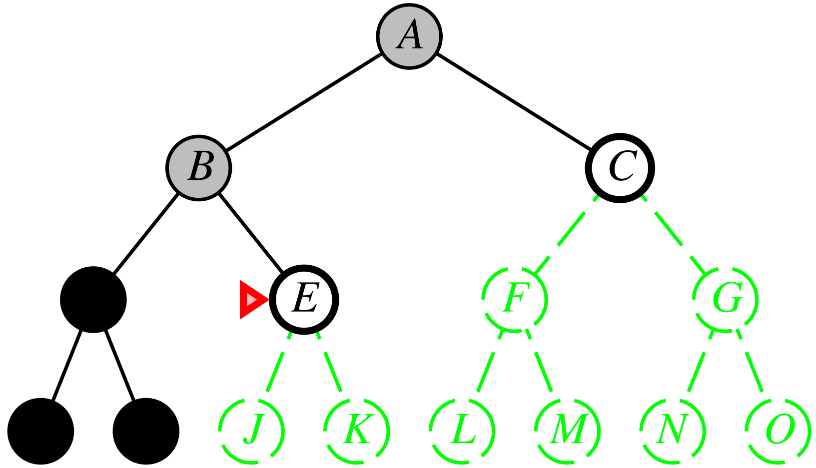


Depth-first search

Expand deepest unexpanded node

Implementation:

$fringe$ = LIFO queue, i.e., put successors at front

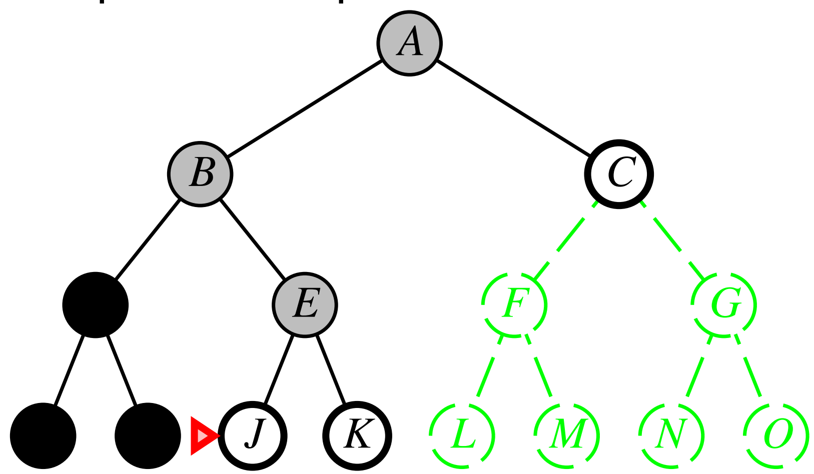


Depth-first search

Expand deepest unexpanded node

Implementation:

$fringe$ = LIFO queue, i.e., put successors at front

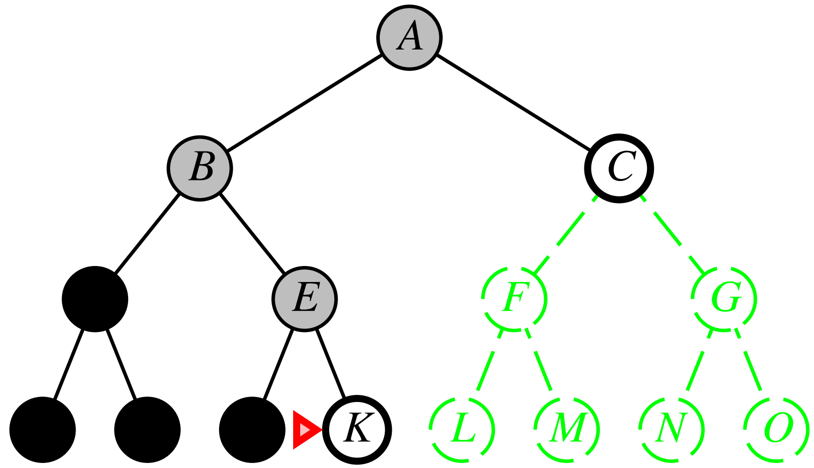


Depth-first search

Expand deepest unexpanded node

Implementation:

$fringe$ = LIFO queue, i.e., put successors at front

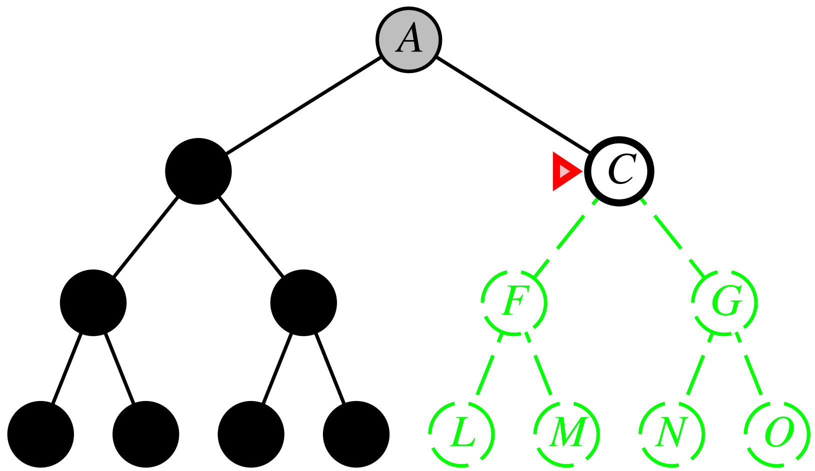


Depth-first search

Expand deepest unexpanded node

Implementation:

$fringe$ = LIFO queue, i.e., put successors at front



Properties

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces with repeated states avoid

Time?? $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, may be much faster than breadth-first

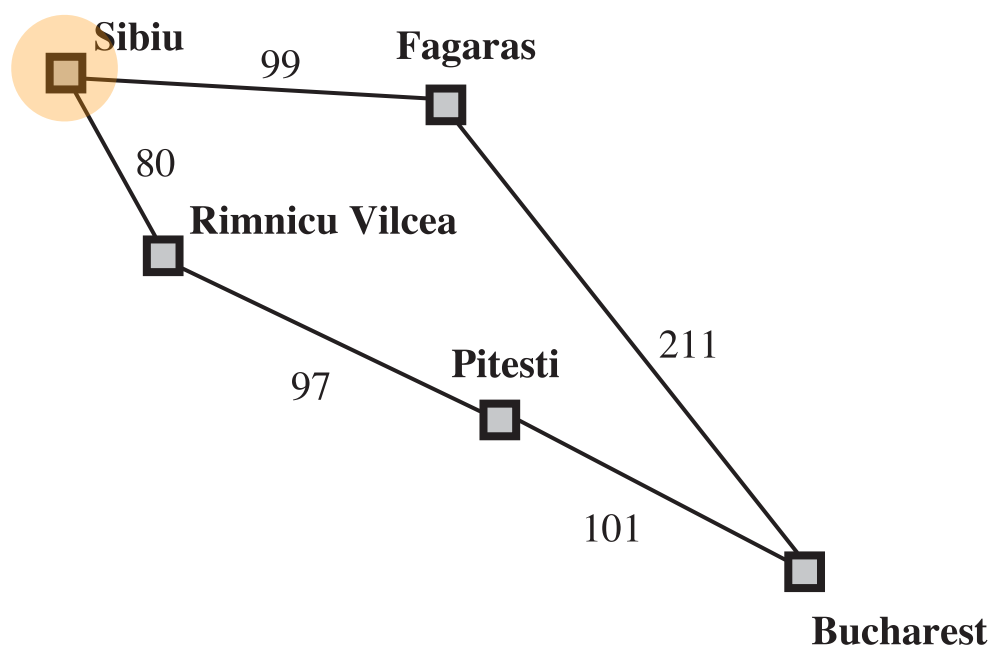
Space?? $O(bm)$, i.e., linear space!

Optimal?? No

Uniform-cost search

Breadth-first search: First In First Out queue
Depth-first search: Last In First Out queue (stack)
Uniform-cost search: Priority queue (least cost out)

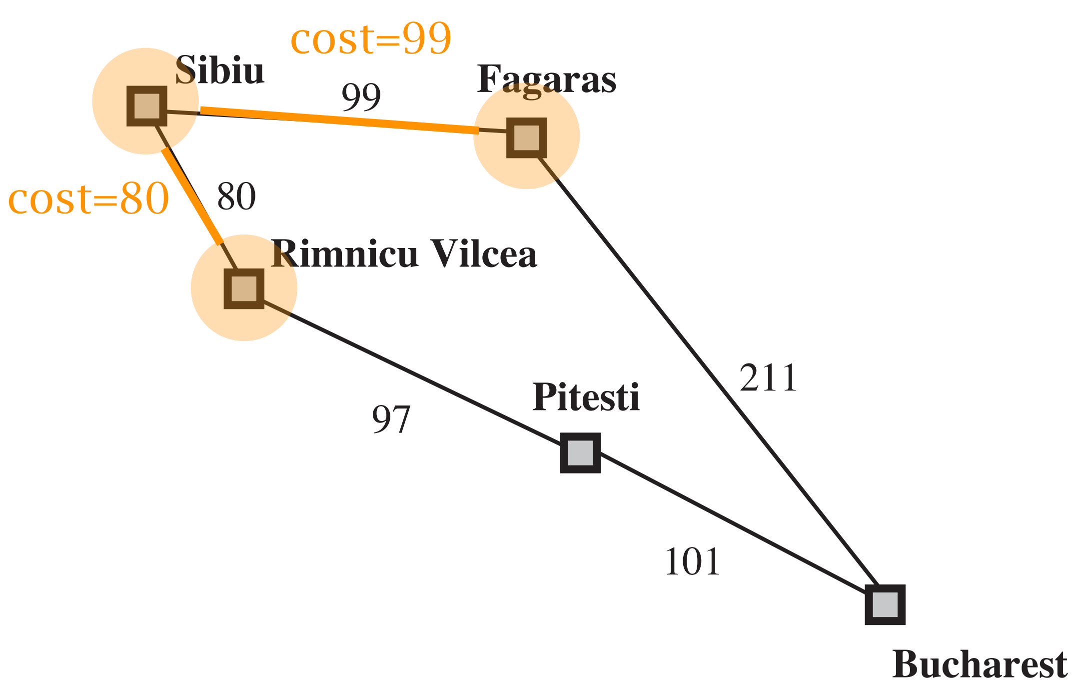
part of the map



Uniform-cost search

Breadth-first search: First In First Out queue
Depth-first search: Last In First Out queue (stack)
Uniform-cost search: Priority queue (least cost out)

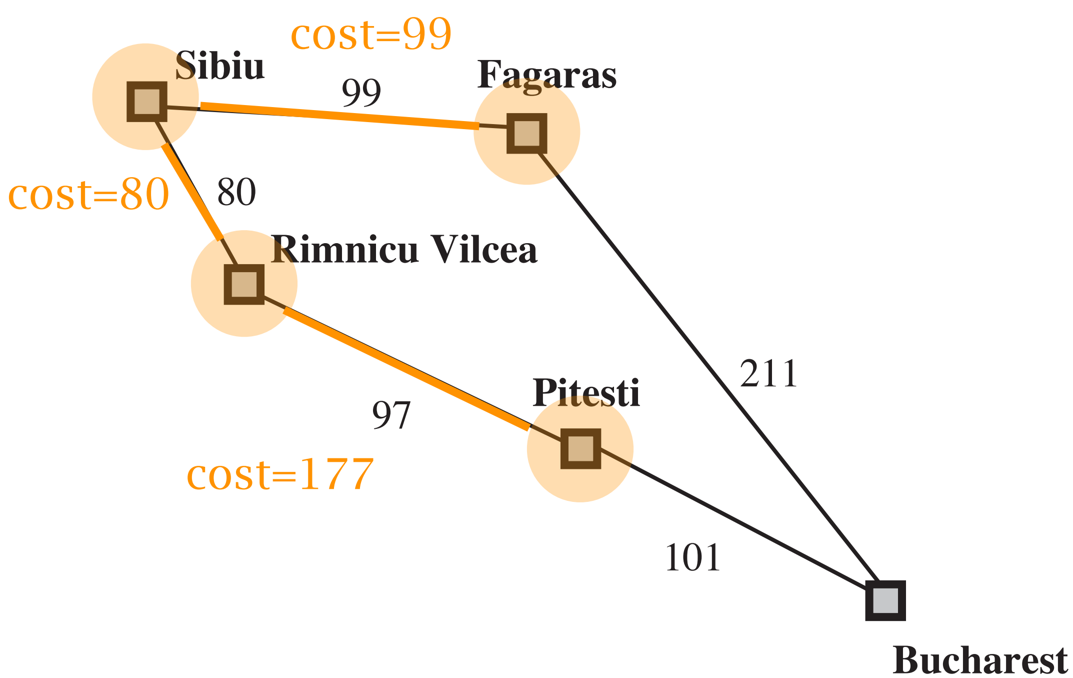
part of the map



Uniform-cost search

Breadth-first search: First In First Out queue
Depth-first search: Last In First Out queue (stack)
Uniform-cost search: Priority queue (least cost out)

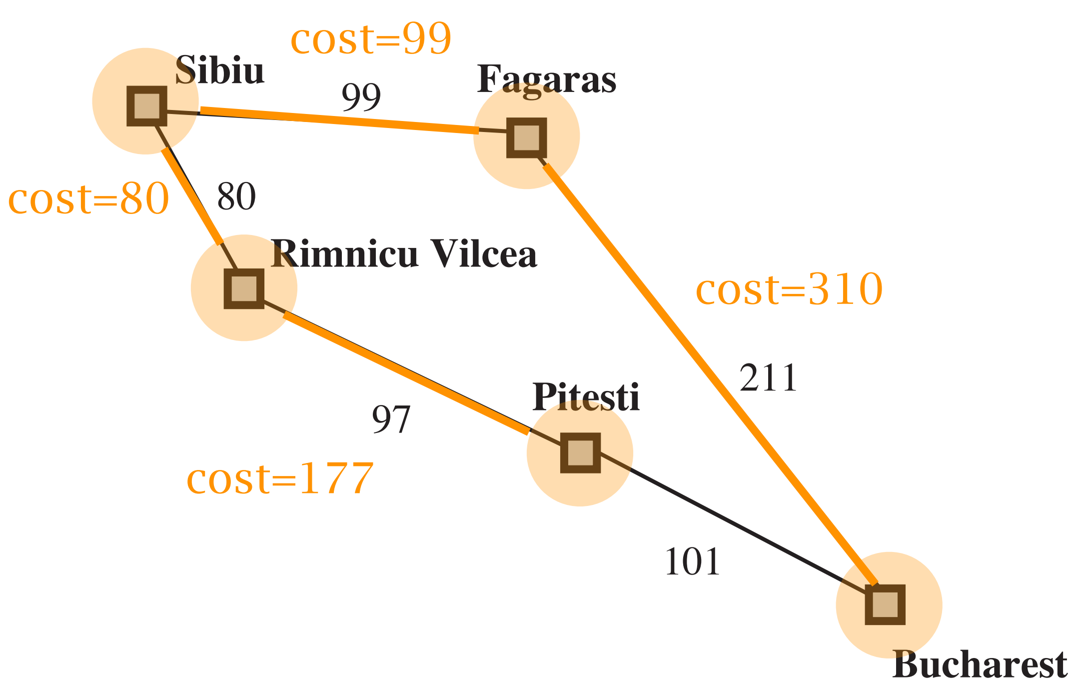
part of the map



Uniform-cost search

Breadth-first search: First In First Out queue
Depth-first search: Last In First Out queue (stack)
Uniform-cost search: Priority queue (least cost out)

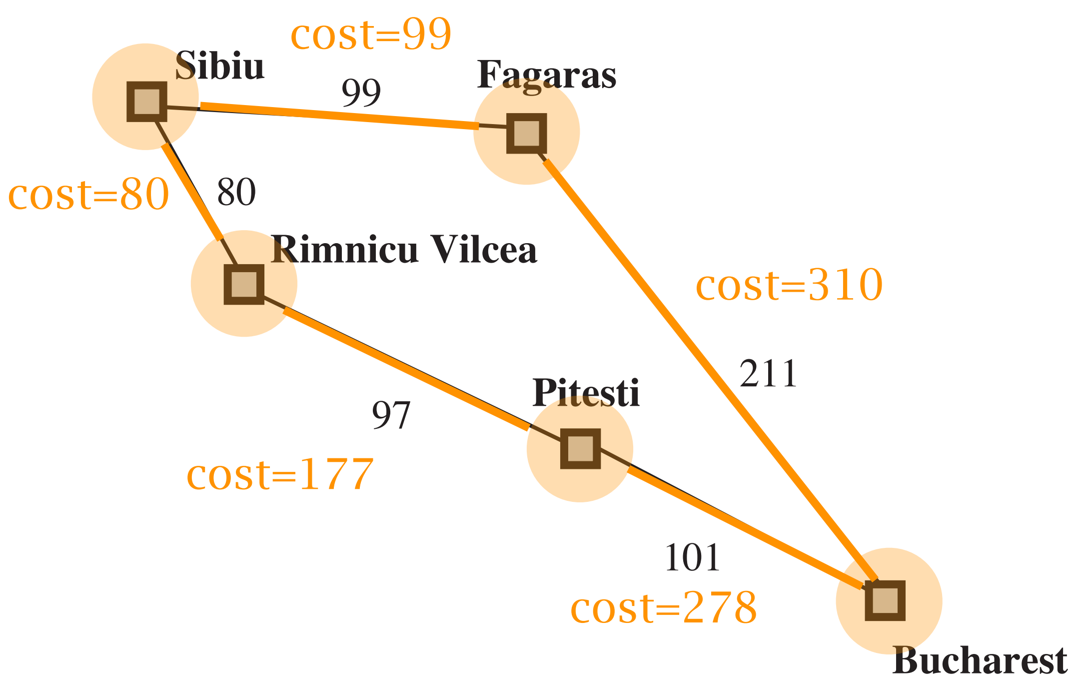
part of the map



Uniform-cost search

Breadth-first search: First In First Out queue
Depth-first search: Last In First Out queue (stack)
Uniform-cost search: Priority queue (least cost out)

part of the map

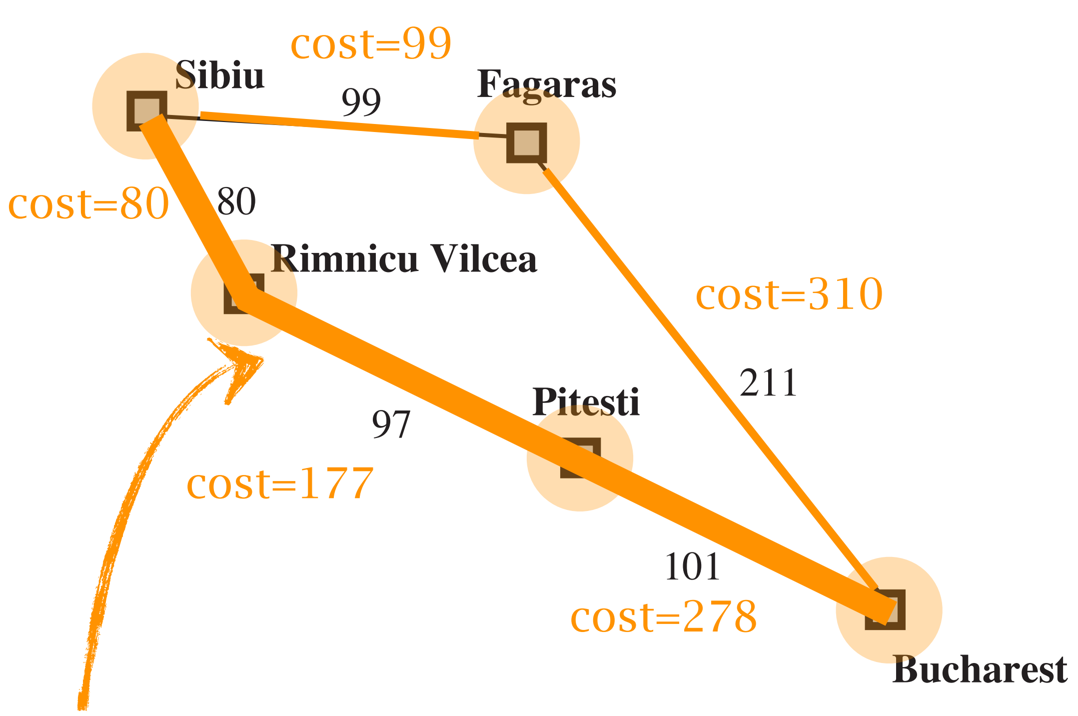


Uniform-cost search

Breadth-first search: First In First Out queue
Depth-first search: Last In First Out queue (stack)
Uniform-cost search: Priority queue (least cost out)

Equivalent to breadth-first if step costs all equal

part of the map



best path from Sibiu to Bucharest

Properties

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal?? Yes—

Question: why it is optimal?

Breadth-first v.s. depth-first

Breadth-first: faster, larger memory
Depth-first: less memory, slower

Question: how to better balance time and space?

Depth-limited search

limit the maximum depth of the depth-first search

i.e., nodes at depth *l* have no successors

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST(problem, STATE[node]) then return node
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Iterative deepening search

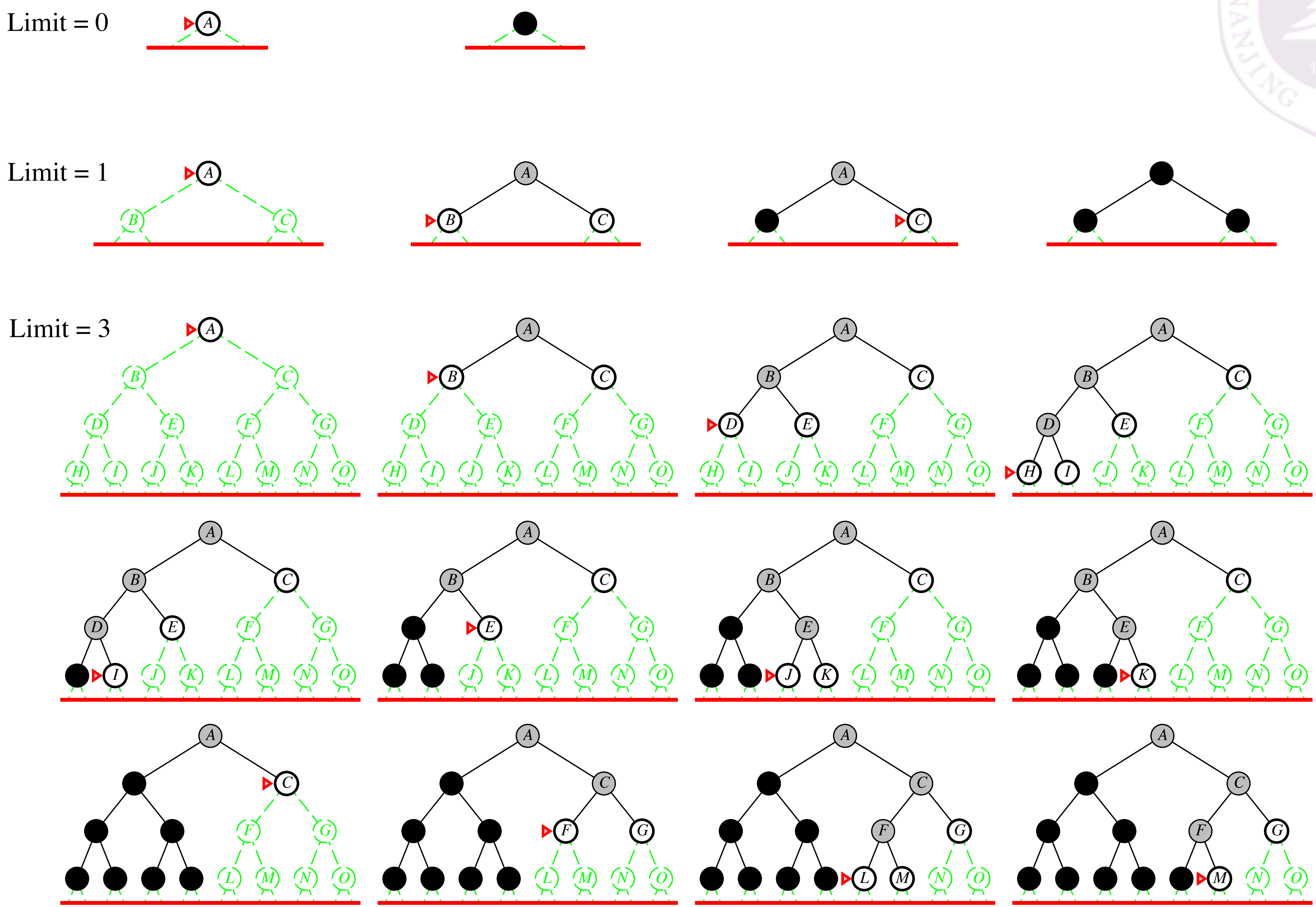
try depth-limited search with increasing limit

restart the search at each time

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
    end
```

Example



wasteful searching the beginning nodes many times?

Properties

in the same order as the breadth-first search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

small space as depth-first search

Space?? $O(bd)$

Optimal?? Yes, if step cost = 1
Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

IDS does better because other nodes at depth d are not expanded

BFS can be modified to apply goal test when a node is **generated**

Summary

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*