Problem Solving Agent

Chapter#3

Objectives of the Chapter

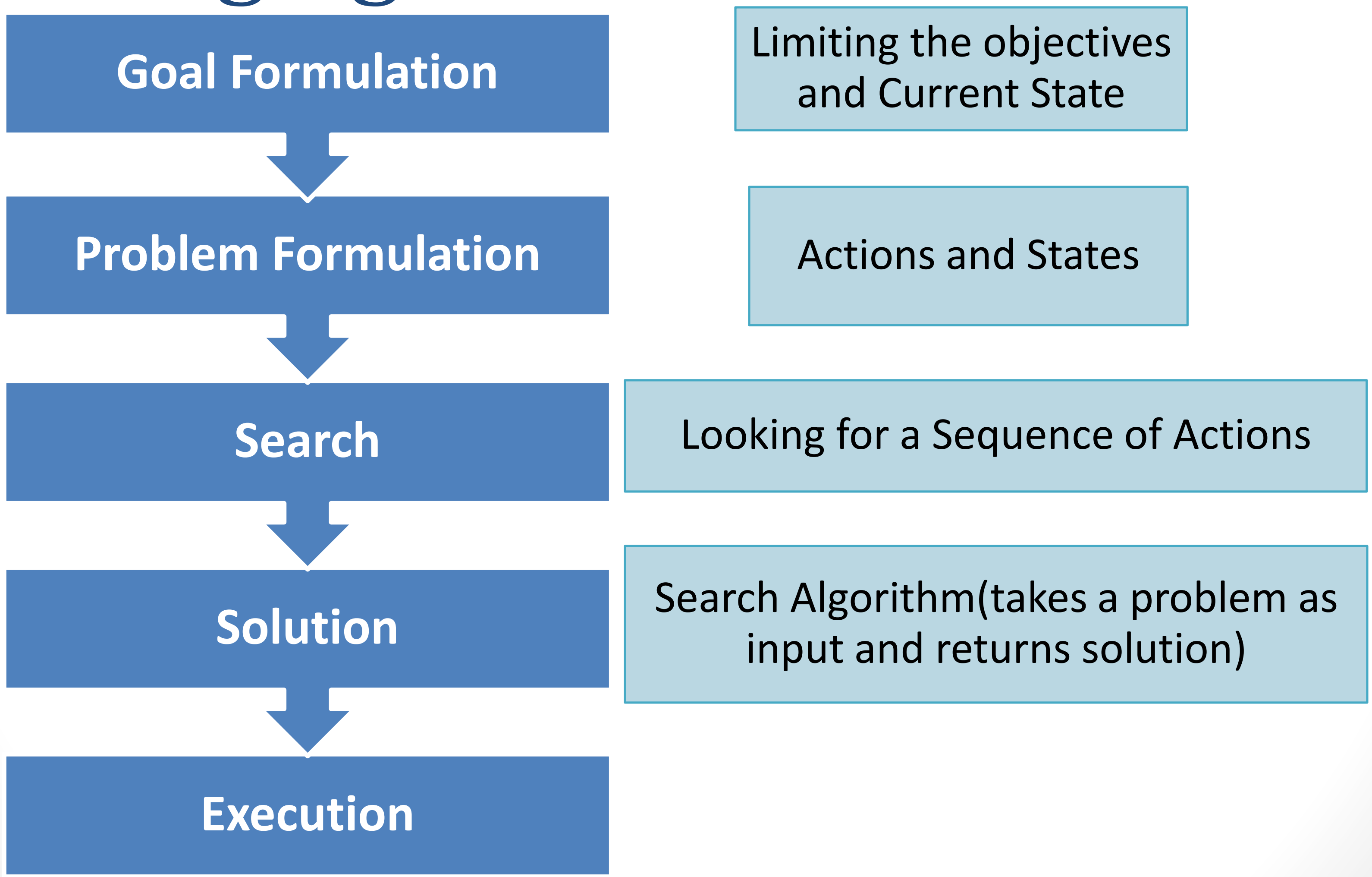
- Problem Solving Agents
- How to formulate a Problem
- Search Strategies
- Informed & Un-informed

Problem Solving Agent

- **Problem Solving agents** are one kind of goal-based agent.

- Problem solving agents use **atomic** representations i.e. states of the world are considered as wholes, with no **internal structure** visible to the problem solving.

Functionality of Problem Solving Agent



Problem Formulation

1.STATE

Problem Formulation

2.INITIAL STATE

Problem Formulation

3.ACTIONS

Problem Formulation

4.TRANSITION MODEL

Problem Formulation

5.GOAL TEST

Problem Formulation

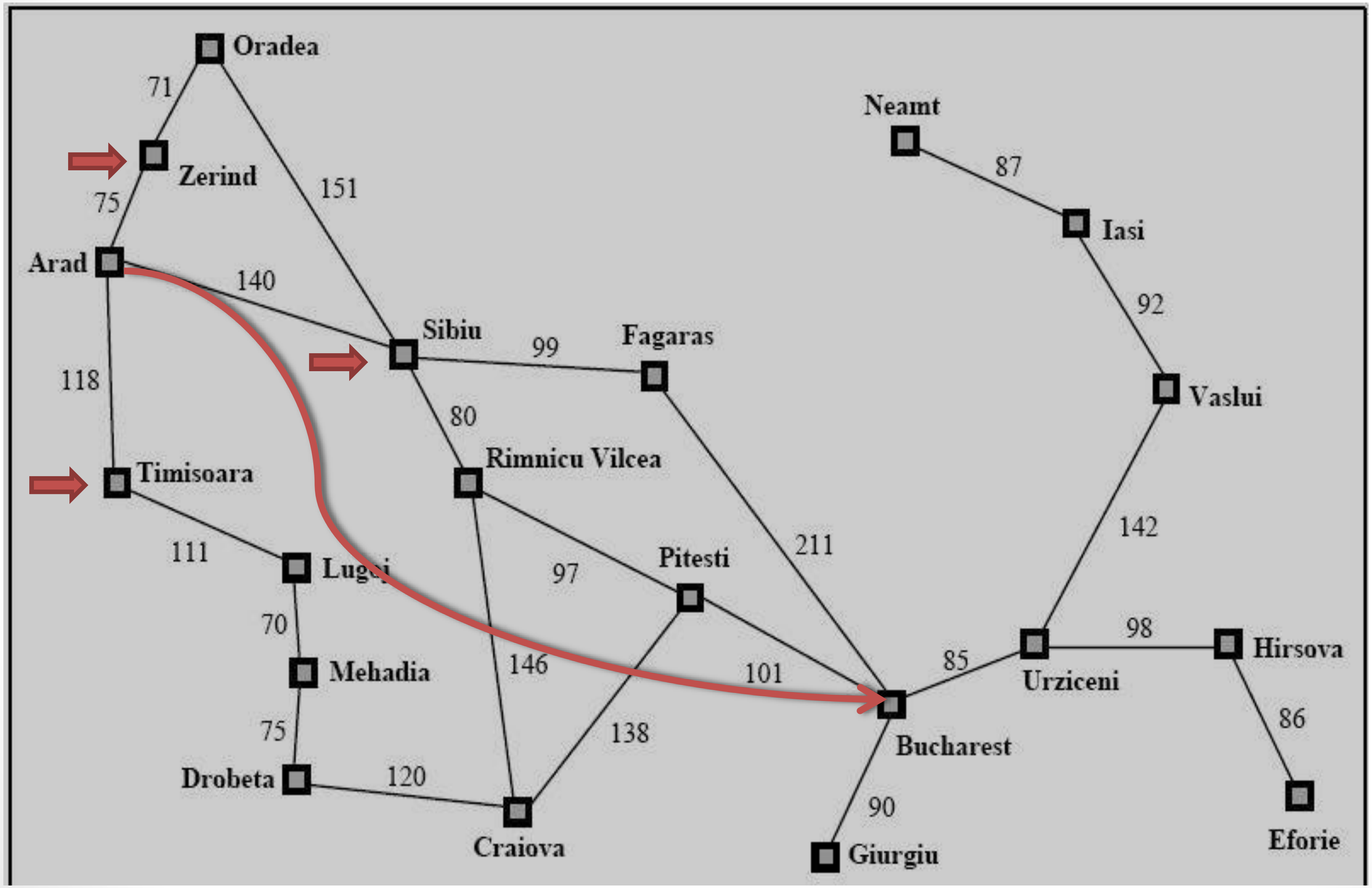
6.PATH COST

A Touring Agent Problem

(SCENARIO)

Imagine an agent in the city of Arad, Romania, enjoying a touring holiday. now, suppose the agent has a non-refundable ticket to fly out of Bucharest the following day. In that case, it makes sense for the agent to adopt the **Goal** of getting to Bucharest

Road Map to Romania



Total Solutions

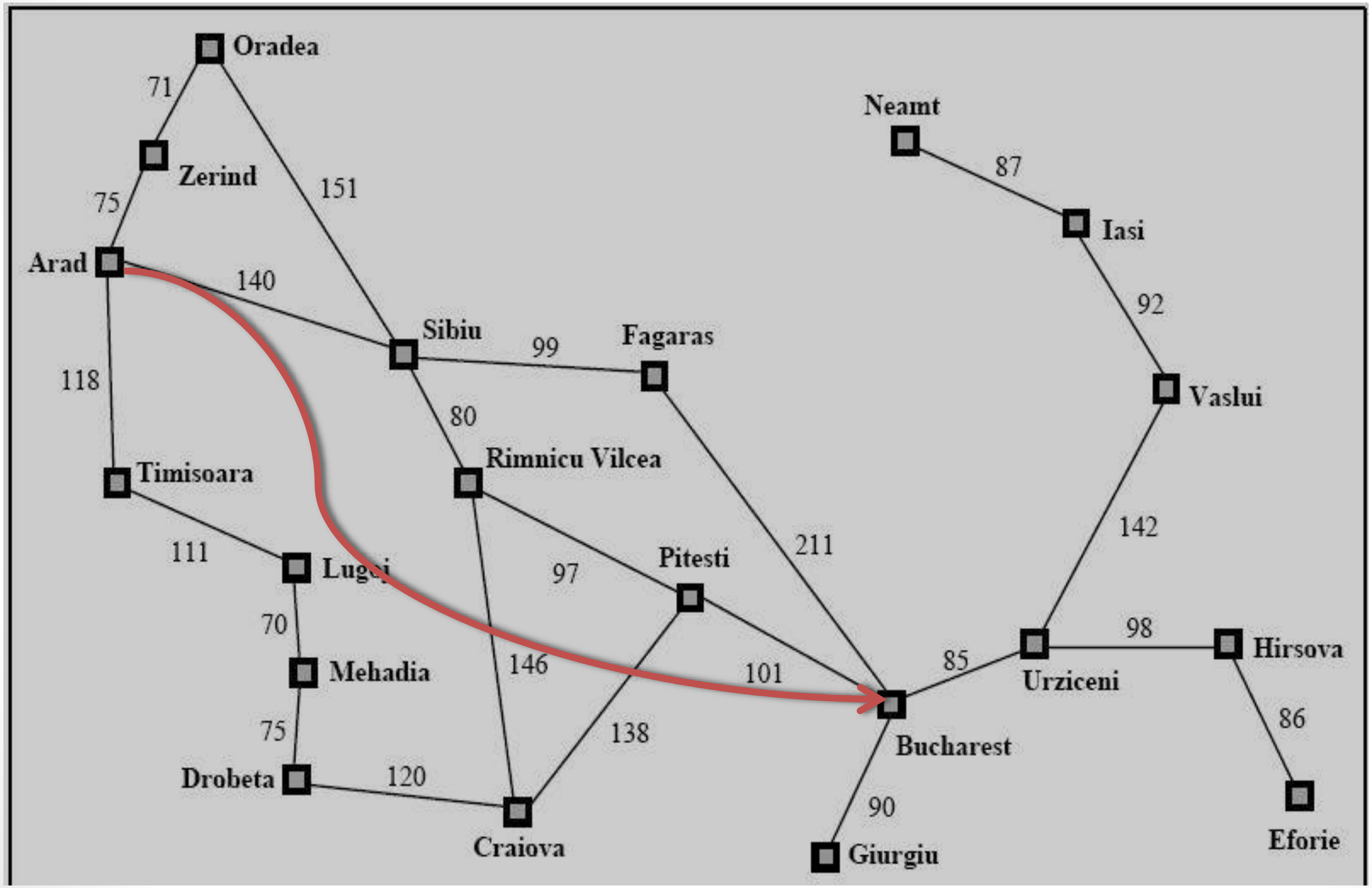
Total Solutions= N x 2^N

- **Example**

  For Romania State N= 20;

  Total Solutions= 20 × 2^{20};

Road Map to Romania



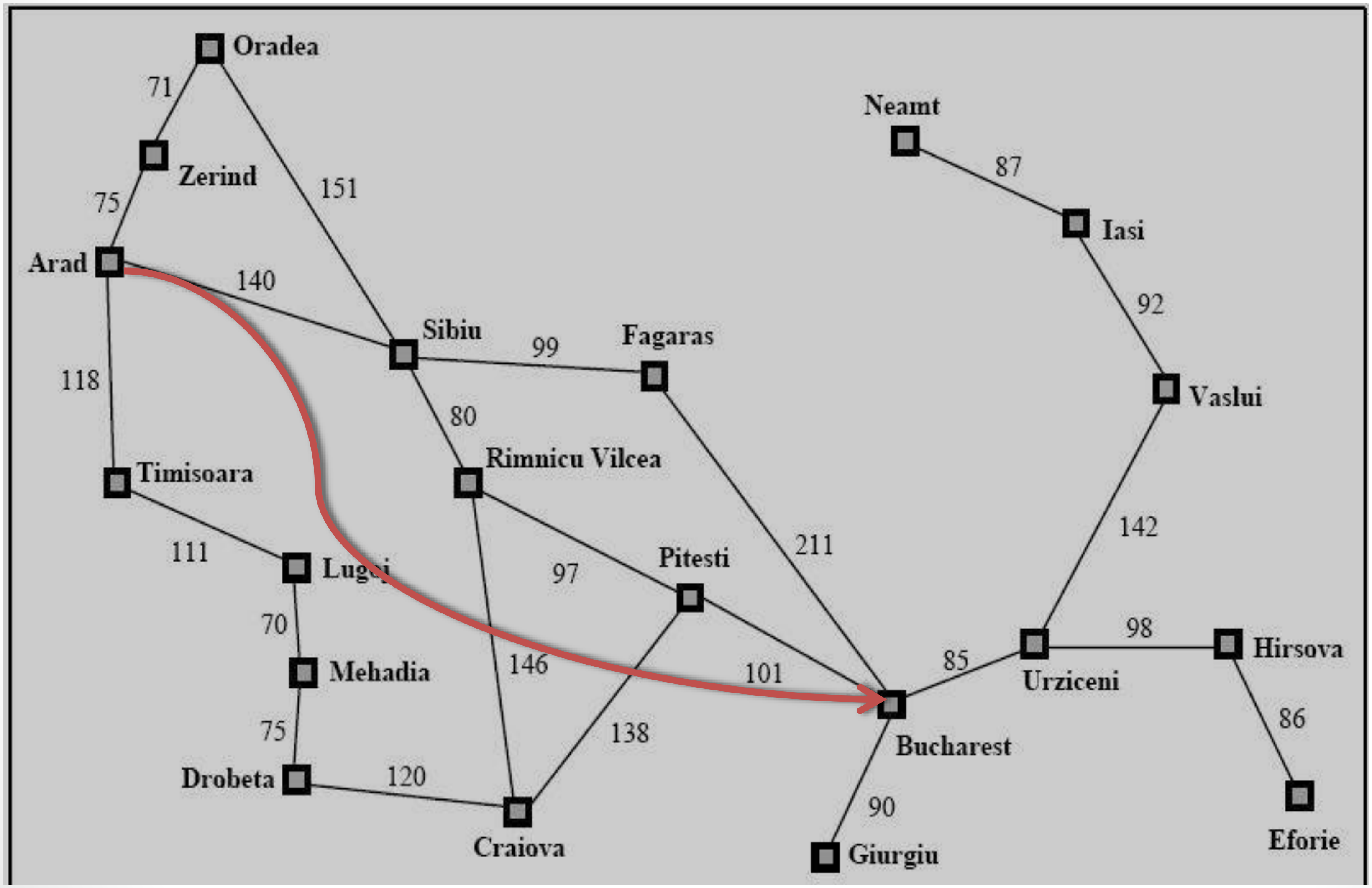
Steps in Problem Formulation

- Initial State

IN (STATE)

IN (ARAD)

Road Map to Romania



Action

ACTION (STATE) → GO(STATES)

ACTION (ARAD) → GO (ZERIND)

GO (SIBIU)

GO (TIMISOARA)

Transition model

RESULT (s, a) ⟶ IN (x) (Transition Model)next state

- **Current state= s**
- **Action = a**

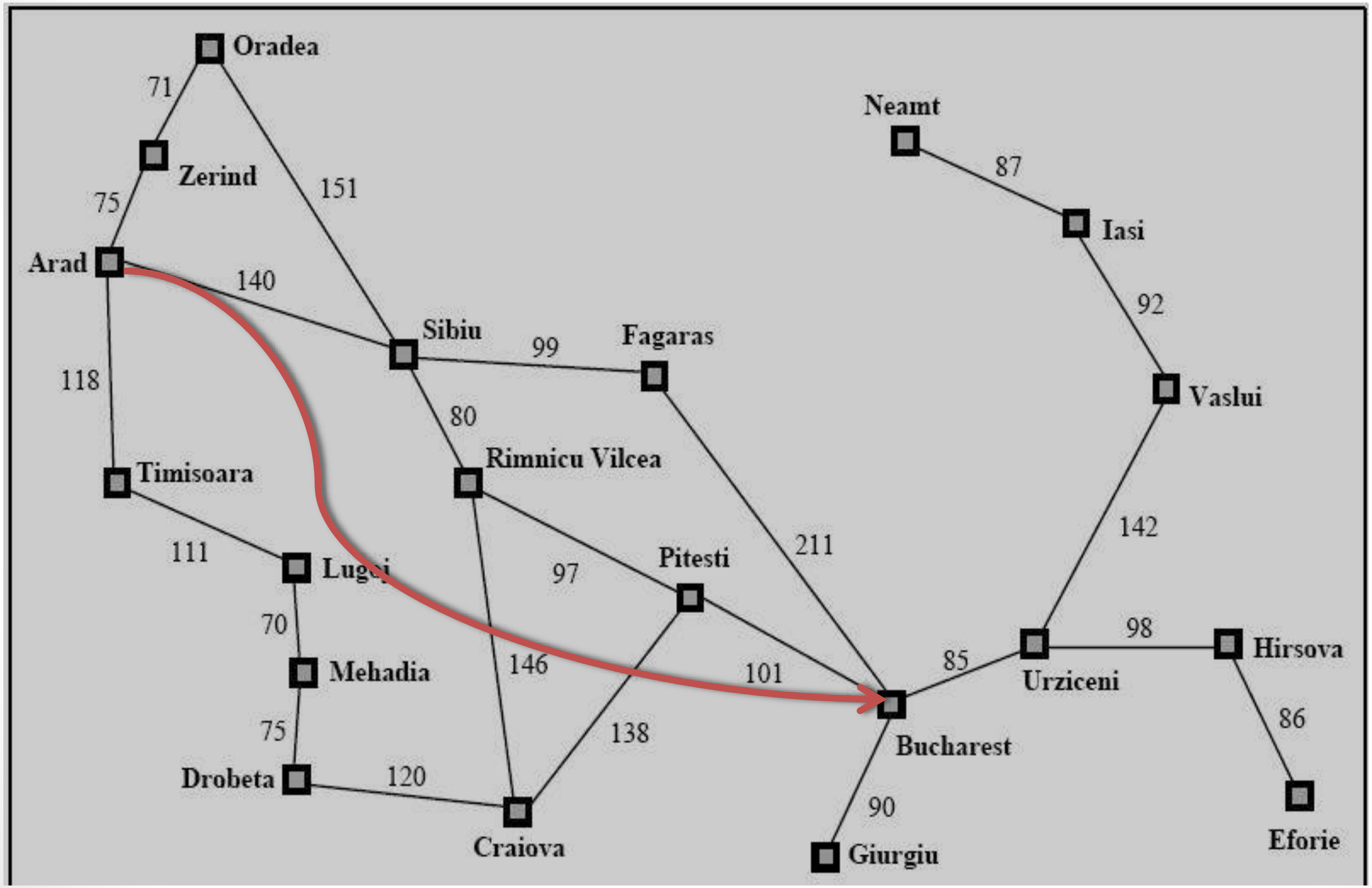
- **RESULT(IN(ARAD), GO (ZERIND)= IN (ZERIND)**

Goal Test

- **IN (x) = = {IN(g)};**

- **IN (x) = = {in(BUCHAREST)};**

Road Map to Romania



Path Cost

- **C (s, a, x) = = p**

- **C(IN (ARAD), GO(ZERIND), IN (ZERIND)) = = 75**

NOTE

Solution Quality is measured by the **Path cost function**, and an **Optimal solution** has the **lowest path cost** among all solutions

Problems

- Vacuum Cleaner
- 8-Puzzle
- 8-Queens Problem
- Robotic Assembly
- VLSI Layout

Problem types:

Single State:	Accessible and Deterministic Environment
Multiple State:	Inaccessible and Deterministic Environment
Contingency:	Inaccessible and Nondeterministic Environment
Exploration:	Unknown State-space

Finding a solution

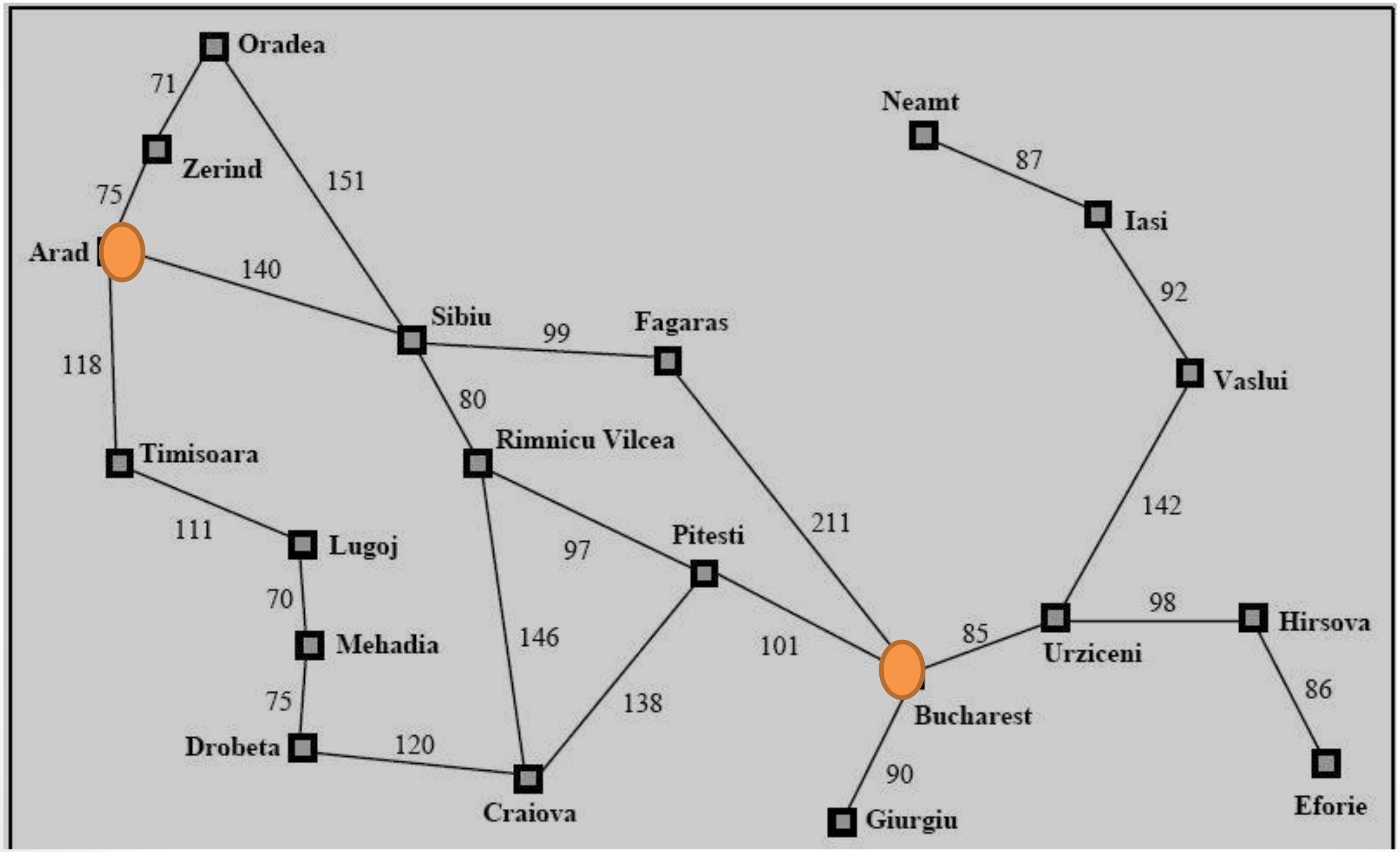
Solution: is a sequence of operators that bring you from current state to the goal state

Basic idea: offline, systematic exploration of simulated state-space by generating successors of explored states (expanding)

Function General-Search(*problem*, *strategy*) returns a *solution*, or failure
  initialize the search tree using the initial state 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 resulting nodes to the search tree
End

- **Strategy:** The search strategy is determined by <u>the **order** in which the nodes are expanded.</u>

Example: Traveling from Arad To Bucharest

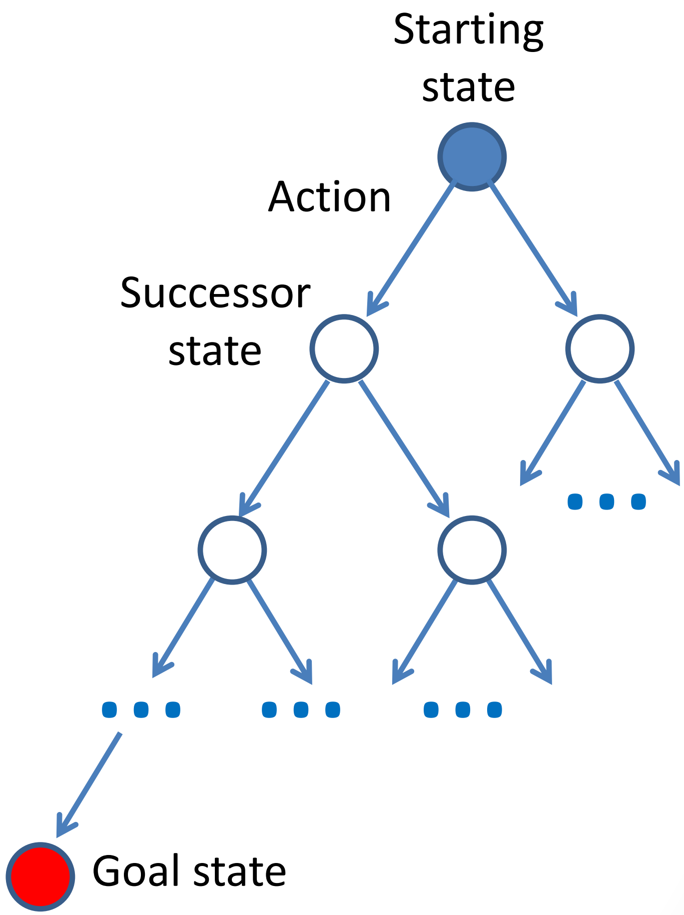


Tree Search

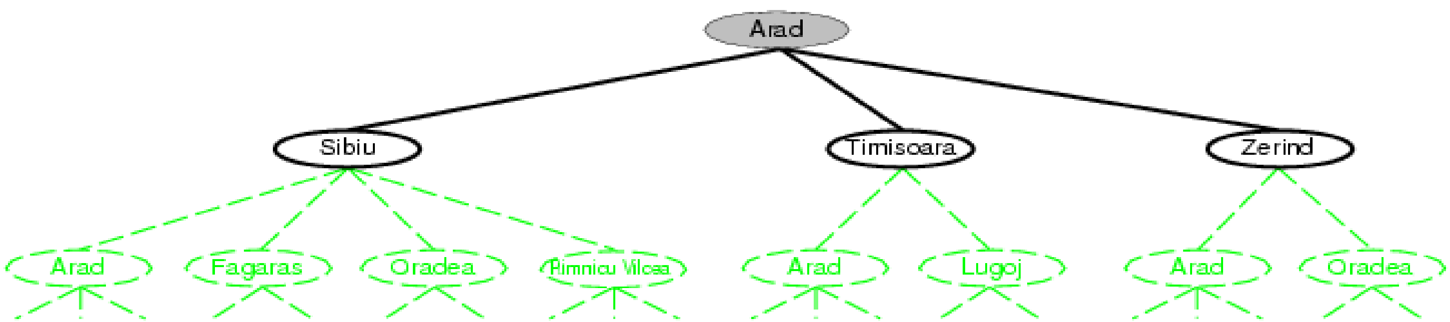
- The root node corresponds to the starting state(Parent Node)
- The children of a node correspond to the **successor states** of that node's state(Child Node)
- A path through the tree corresponds to a sequence of actions

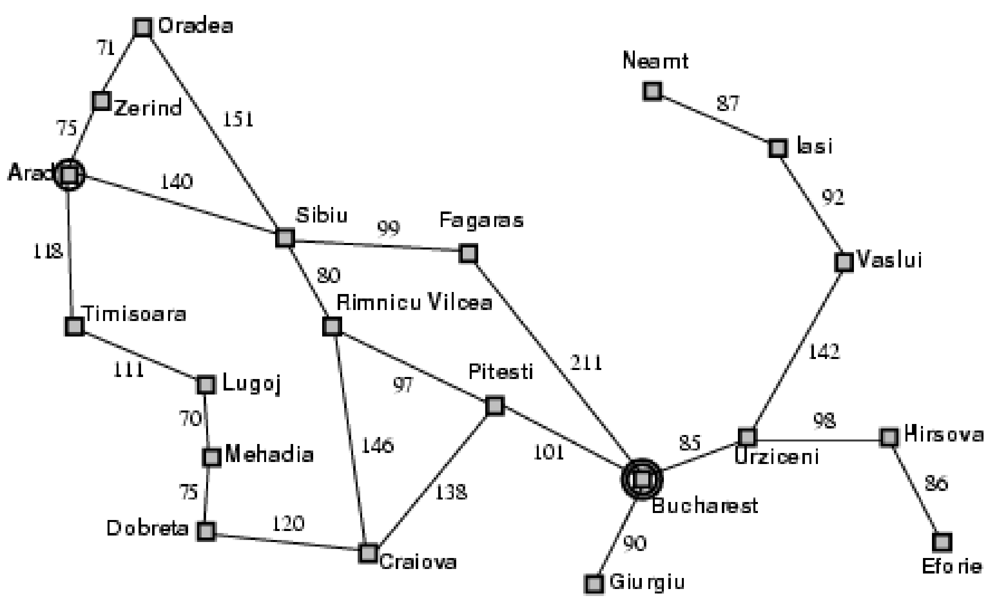
  -A solution is a path ending in the goal state
- Nodes vs. states

  -A state is a representation of a physical configuration, while a node is a data structure that is part of the search tree

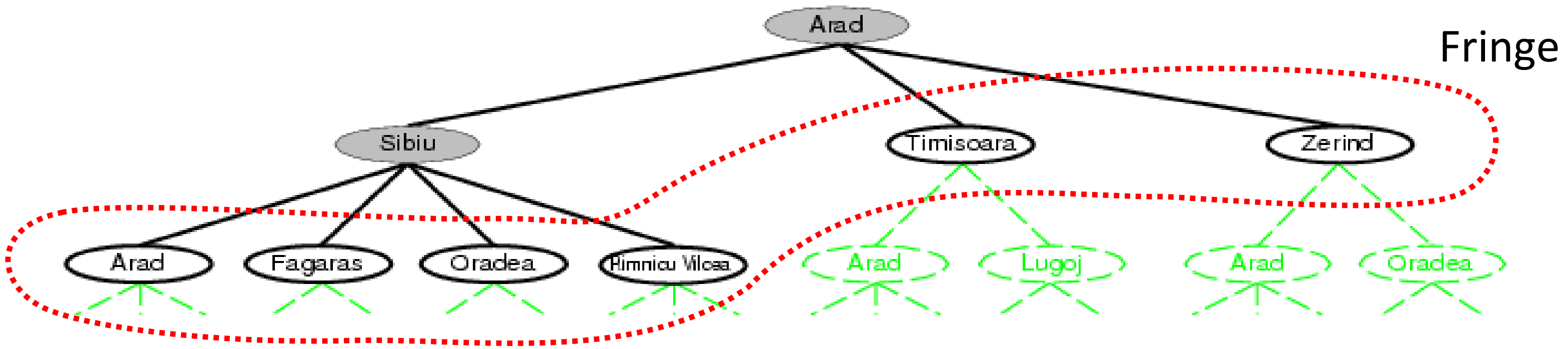


Tree search example





Tree search example



- We can then choose any of these four or go back and choose Timisoara or Zerind. Each of these six nodes is a **leaf node**, that is, a node with no children in the tree.
- The set of all leaf nodes available for expansion at any given point is called the **frontier**.

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

---

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

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

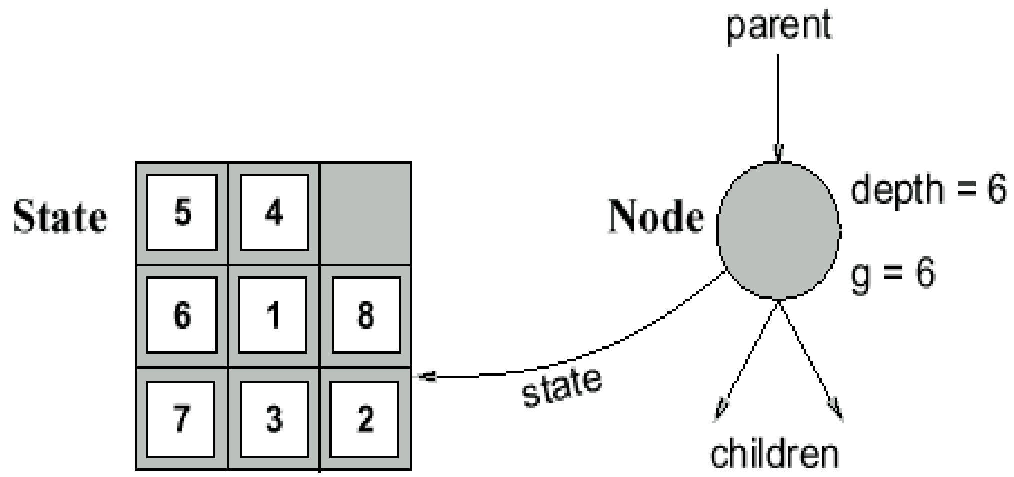
Encapsulating *state* information in *nodes*

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 OPERATORS (or SUCCESSORFN) of the problem to create the corresponding states.

Search strategies

- A **search 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?
  - **Optimality:** does it always find a least-cost solution?
  - **Time complexity:** number of nodes generated
  - **Space complexity:** maximum number of nodes in memory
- 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 length of any path in the state space (may be infinite)

Uninformed search strategies

- Uninformed search strategies use only the information available in the problem definition

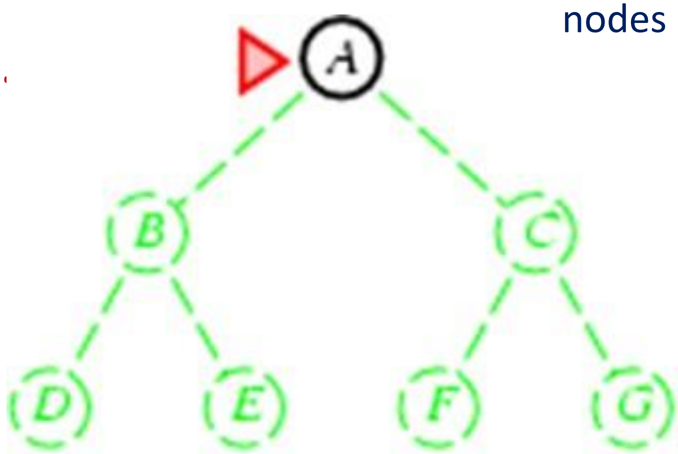
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening depth-first search
  - Bidirectional search
  - Comparing uninformed search strategies

Breadth-first search

- Breadth-first search Expand shallowest unexpanded node
  Frontier (or fringe): nodes in queue to be explored
- Frontier is a first-in-first-out (FIFO) queue, i.e., new successors go at end of the queue.
- Goal-Test when inserted.

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Initial state = A
Is A a goal state?
Put A at end of queue.
frontier = [A]

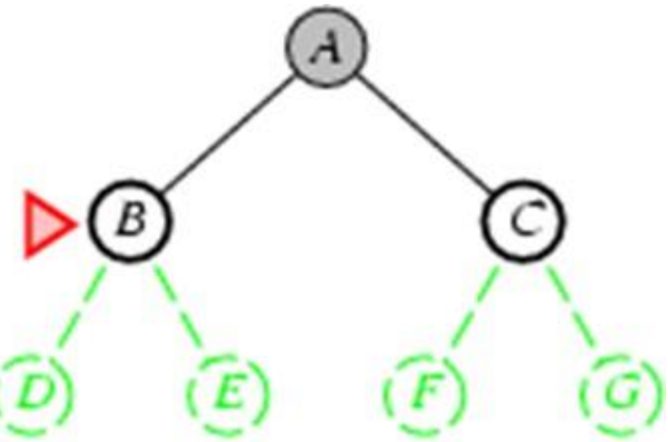


Breadth-first search

- Breadth-first search Expand shallowest unexpanded node
  Frontier (or fringe): nodes in queue to be explored
- Frontier is a first-in-first-out (FIFO) queue, i.e., new successors go at end of the queue.
- Goal-Test when inserted.

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Expand A to B, C.
Is B or C a goal state?
Put B, C at end of queue
frontier = [B,C]



Breadth-first search

- Breadth-first search Expand shallowest unexpanded node
  Frontier (or fringe): nodes in queue to be explored
- Frontier is a first-in-first-out (FIFO) queue, i.e., new successors go at end of the queue.

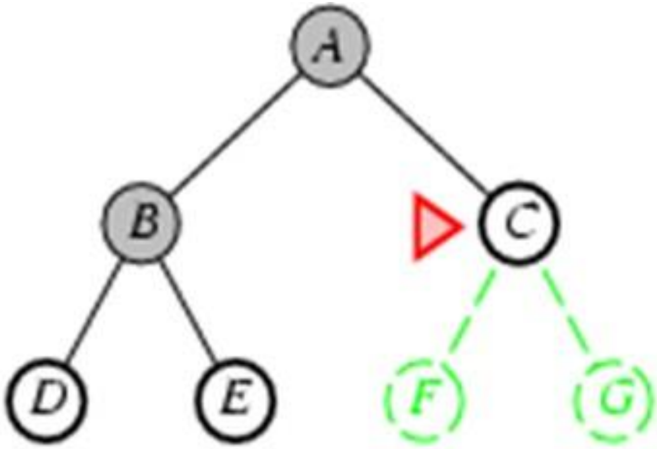
Expand B to D, E

Is D or E a goal state?

Put D, E at end of queue

frontier=[C,D,E]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes



Breadth-first search

- Breadth-first search Expand shallowest unexpanded node Frontier (or fringe): nodes in queue to be explored
- Frontier is a first-in-first-out (FIFO) queue, i.e., new successors go at end of the queue.
- Goal-Test when inserted.

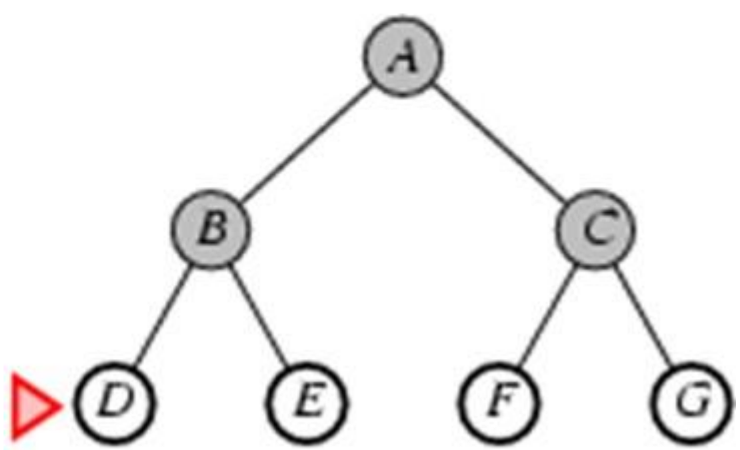
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Expand C to F, G.

Is F or G a goal state?

Put F, G at end of queue.

frontier = [D,E,F,G]



Breadth-first search

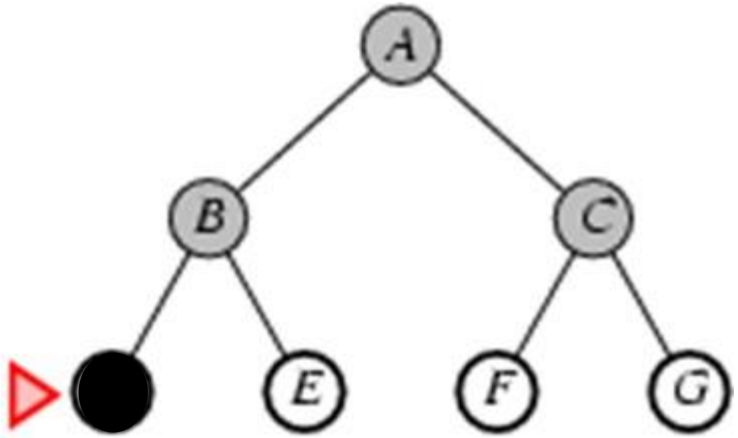
- Breadth-first search Expand shallowest unexpanded node Frontier (or fringe): nodes in queue to be explored
- Frontier is a first-in-first-out (FIFO) queue, i.e., new successors go at end of the queue.
- Goal-Test when inserted.

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Expand D to no children.

Forget D.

frontier = [E,F,G]



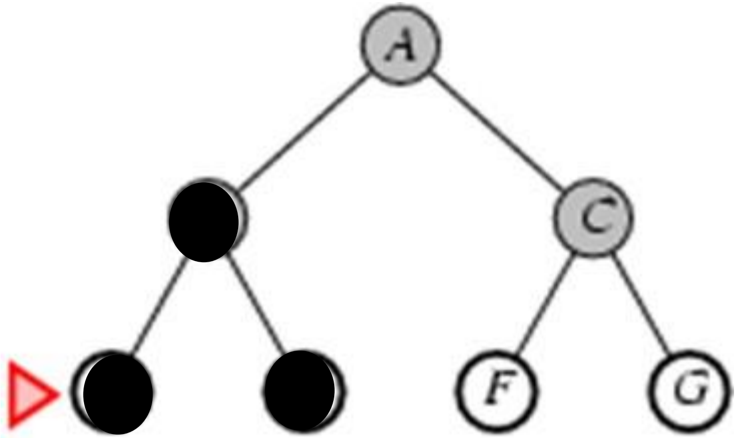
Breadth-first search

- Breadth-first search Expand shallowest unexpanded node Frontier (or fringe): nodes in queue to be explored
- Frontier is a first-in-first-out (FIFO) queue, i.e., new successors go at end of the queue.

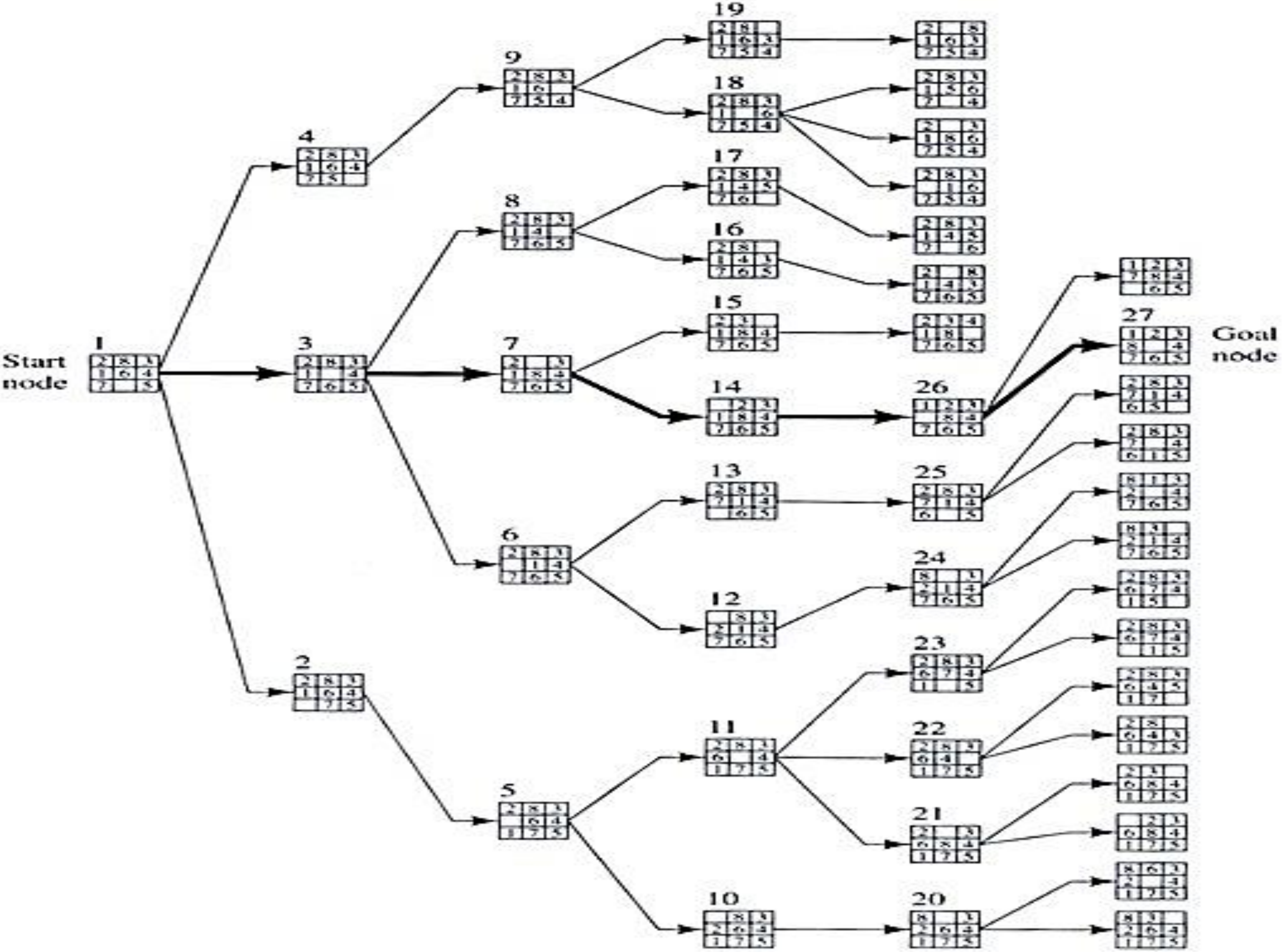
Expand E to no children.

Forget B,E.

frontier = [F,G]



Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes



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

```
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 ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses 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)
```

Figure 3.11 Breadth-first search on a graph.

Properties of Breadth First Search

- Complete? Yes (if b is finite)
- Time? $1+b+b^2+b^3+\ldots+b^d+b(b^d-1) = O(b^{d+1})$
- Space? $O(b^{d+1})$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)
- Space is the bigger problem (more than time)

Uniform Cost

Expand node with smallest path cost g(n).

- *Frontier* is a priority queue, i.e., new successors are merged into the queue sorted by g(n).
  - Can remove successors already on queue w/higher g(n).
    - Saves memory, costs time; another space-time trade-off.
- *Goal-Test* when node is popped off queue.

Uniform Cost

Expand node with smallest path cost g(n).

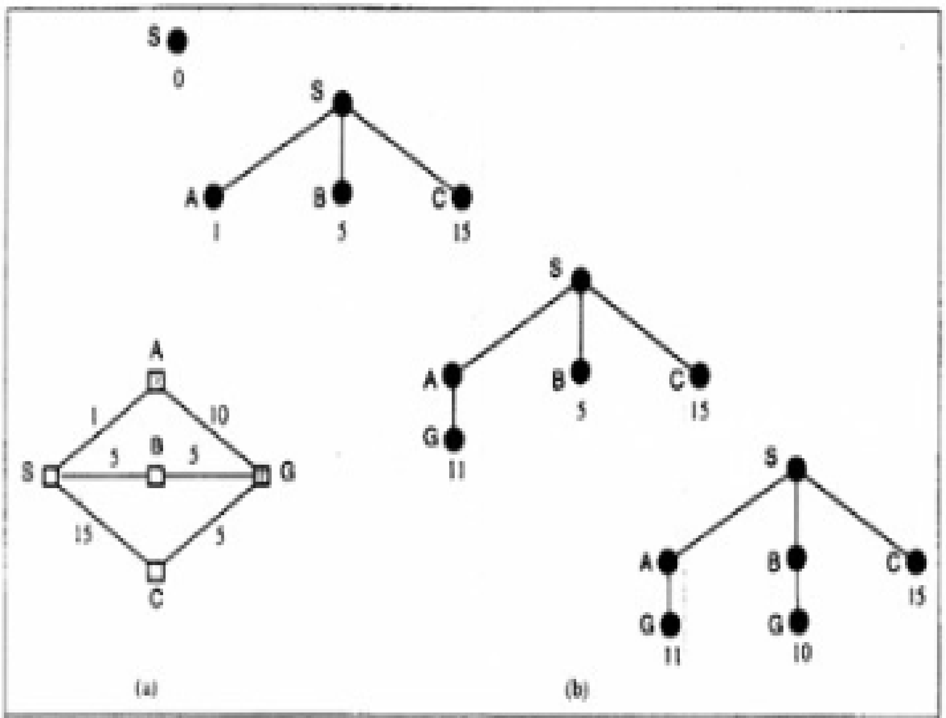


Figure 3.13 A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with $g(n)$. At the next step, the goal node with $g = 10$ will be selected.

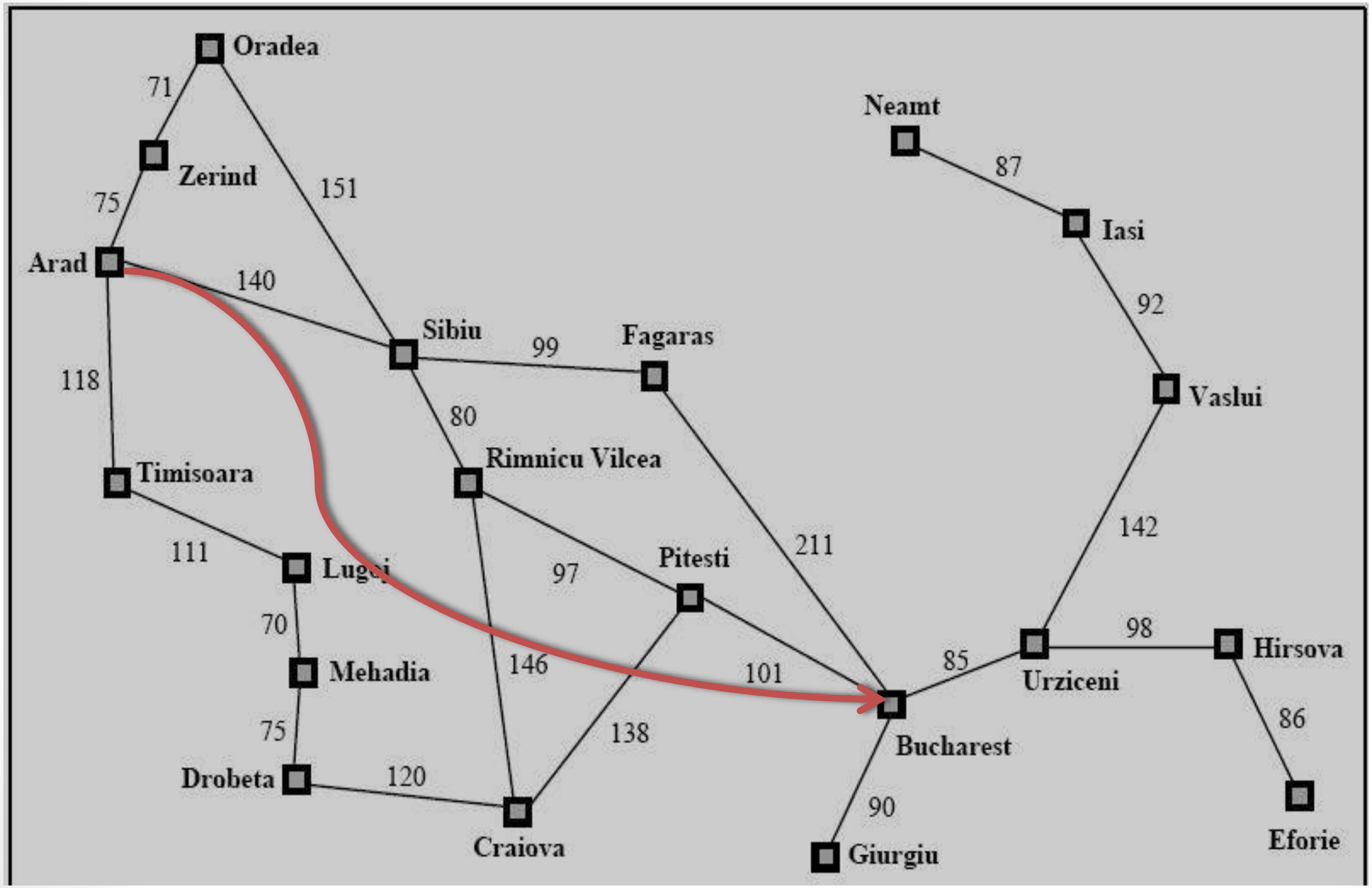
Proof of Completeness:

Given that every step will cost more than 0, and assuming a finite branching factor, there is a finite number of expansions required before the total path cost is equal to the path cost of the goal state. Hence, we will reach it.

Proof of optimality given completeness:

Assume UCS is not optimal.
Then there must be an (optimal) goal state with path cost smaller than the found (suboptimal) goal state (invoking completeness).
However, this is impossible because UCS would have expanded that node first by definition.
Contradiction.

Road Map to Romania



Uniform Cost Example

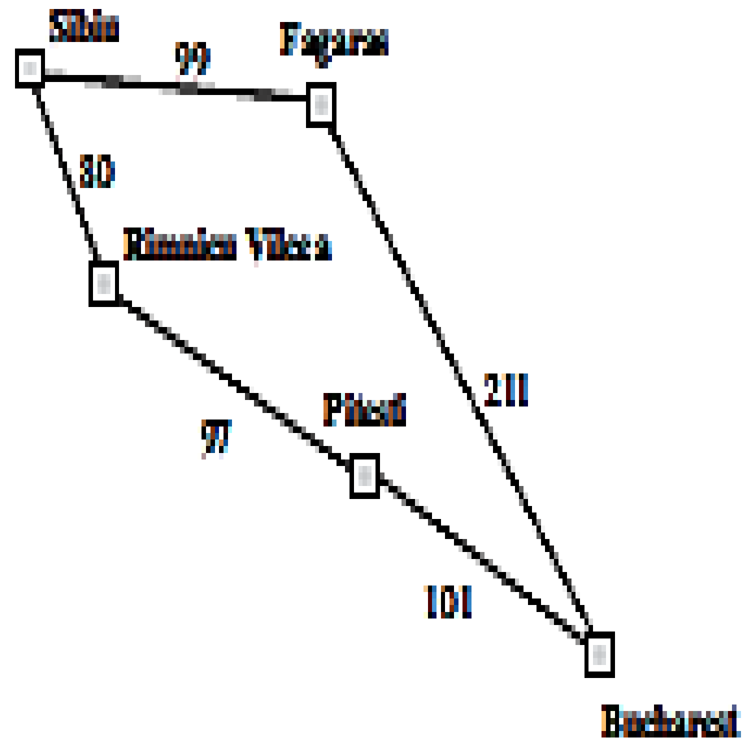


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

Properties of Uniform Cost Search

- **Complete?**

  Yes, if step cost is greater than some positive constant ε (we don't want infinite sequences of steps that have a finite total cost)

- **Optimal?**

  Yes – nodes expanded in increasing order of path cost

- **Time?**

  Number of nodes with path cost ≤ cost of optimal solution (C^*), $O(b^{C^*/\varepsilon})$

  This can be greater than $O(b^d)$: the search can explore long paths consisting of small steps before exploring shorter paths consisting of larger steps

- **Space?**

  $O(b^{C^*/\varepsilon})$

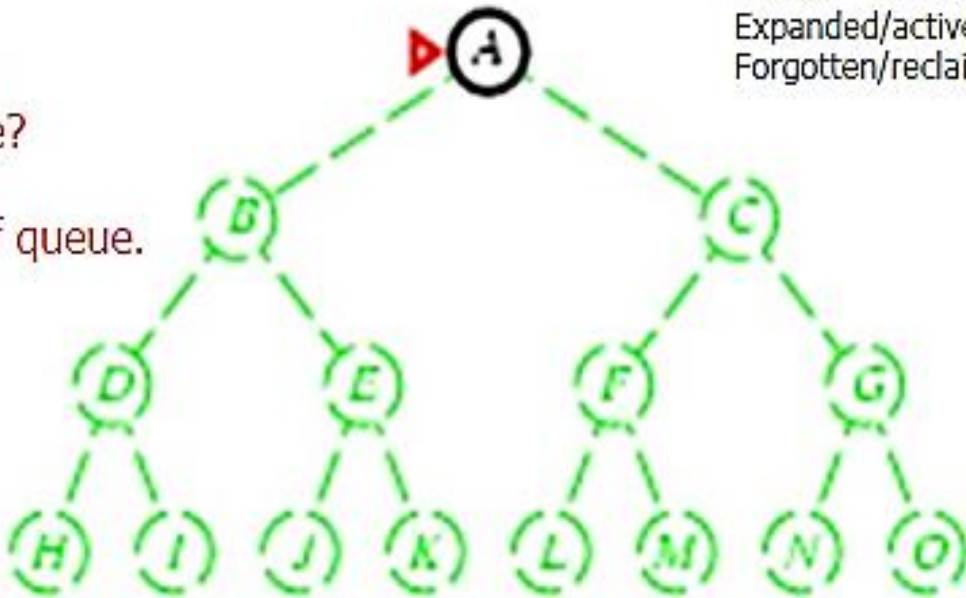
Depth First Search

- Expand *deepest* unexpanded node
- *Frontier* = Last In First Out (LIFO) queue, i.e., new successors go at the front of the queue.
- *Goal-Test* when inserted.

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Initial state = A
Is A a goal state?

Put A at front of queue.
frontier = [A]



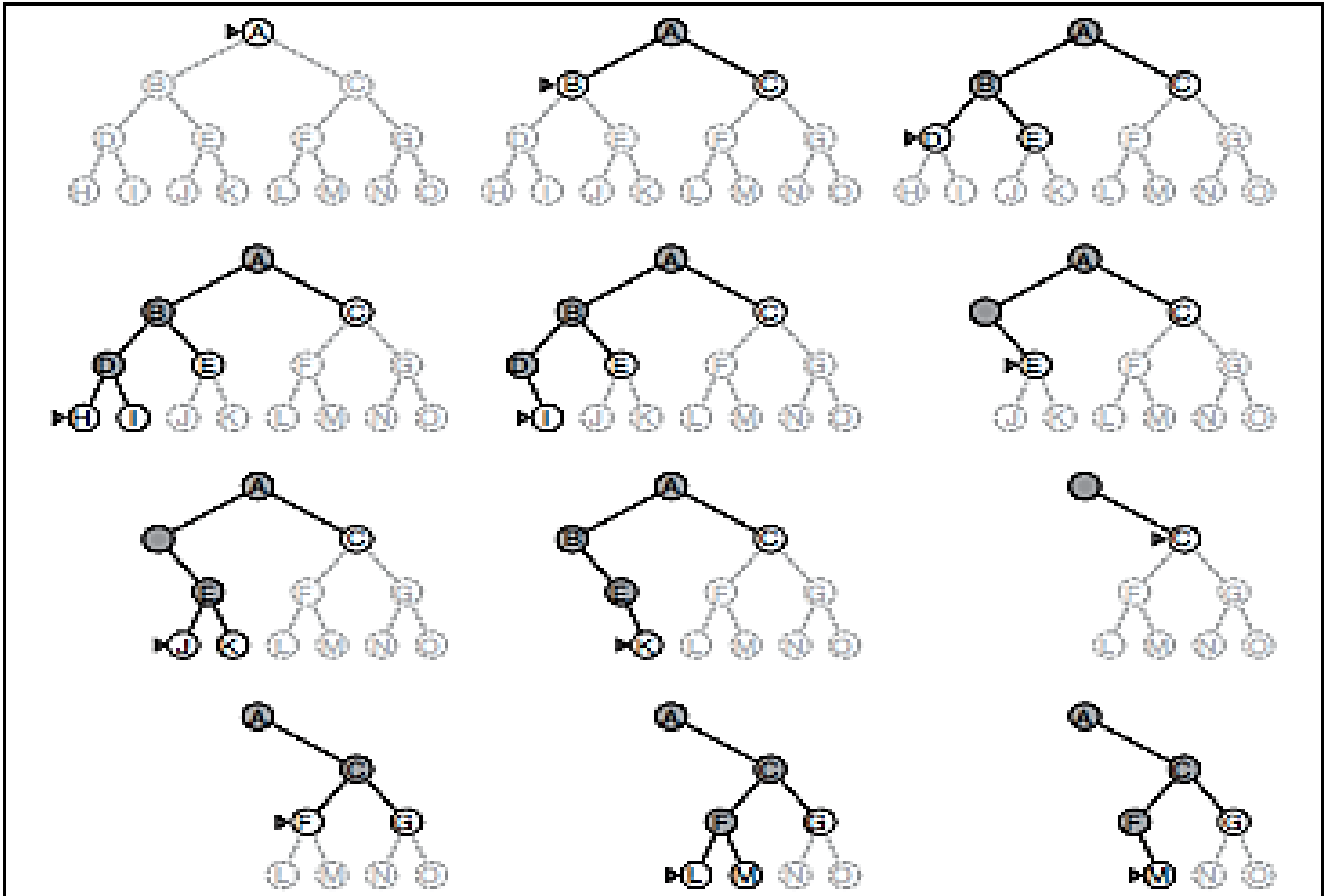


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

Properties of Depth First Search

- Complete? No: fails in loops/infinite-depth spaces
  - Can modify to avoid loops/repeated states along path
    - check if current nodes occurred before on path to root
  - Can use graph search (remember all nodes ever seen)
    - problem with graph search: space is exponential, not linear
  - Still fails in infinite-depth spaces (may miss goal entirely)
- Time? $O(b^m)$ with m =maximum depth of space
  - Terrible if m is much larger than d
  - If solutions are dense, may be much faster than BFS
- Space? $O(bm)$, i.e., linear space!
  - Remember a single path + expanded unexplored nodes
- Optimal? No: It may find a non-optimal goal first

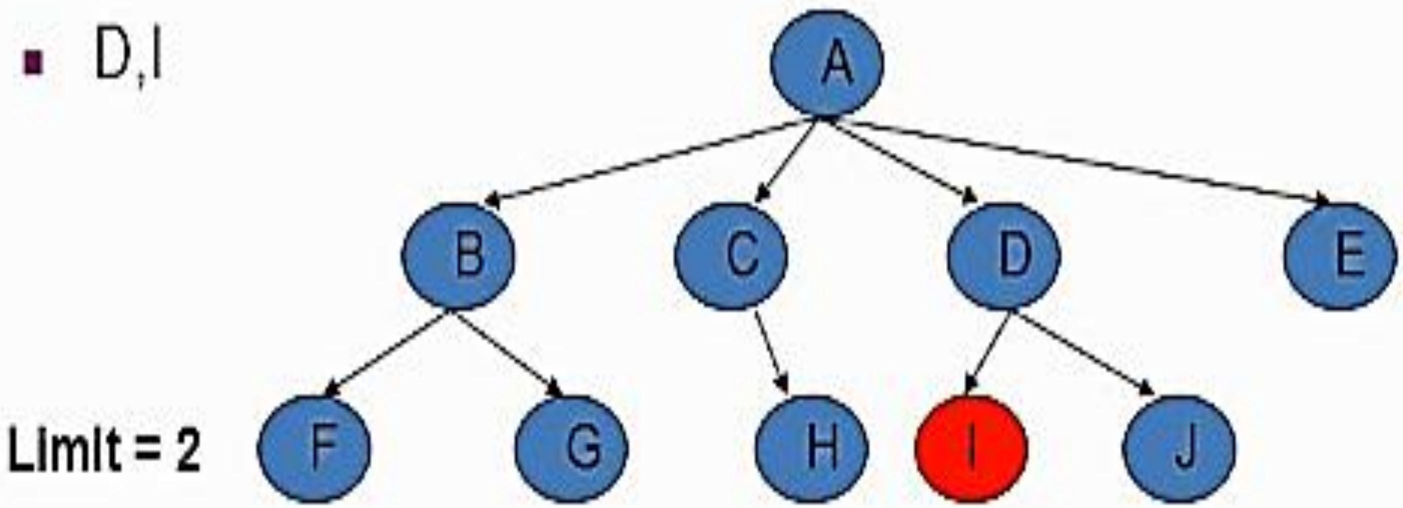
Depth Limited Search

- To avoid the infinite depth problem of DFS, only search until depth L, i.e., we don't expand nodes beyond depth L.

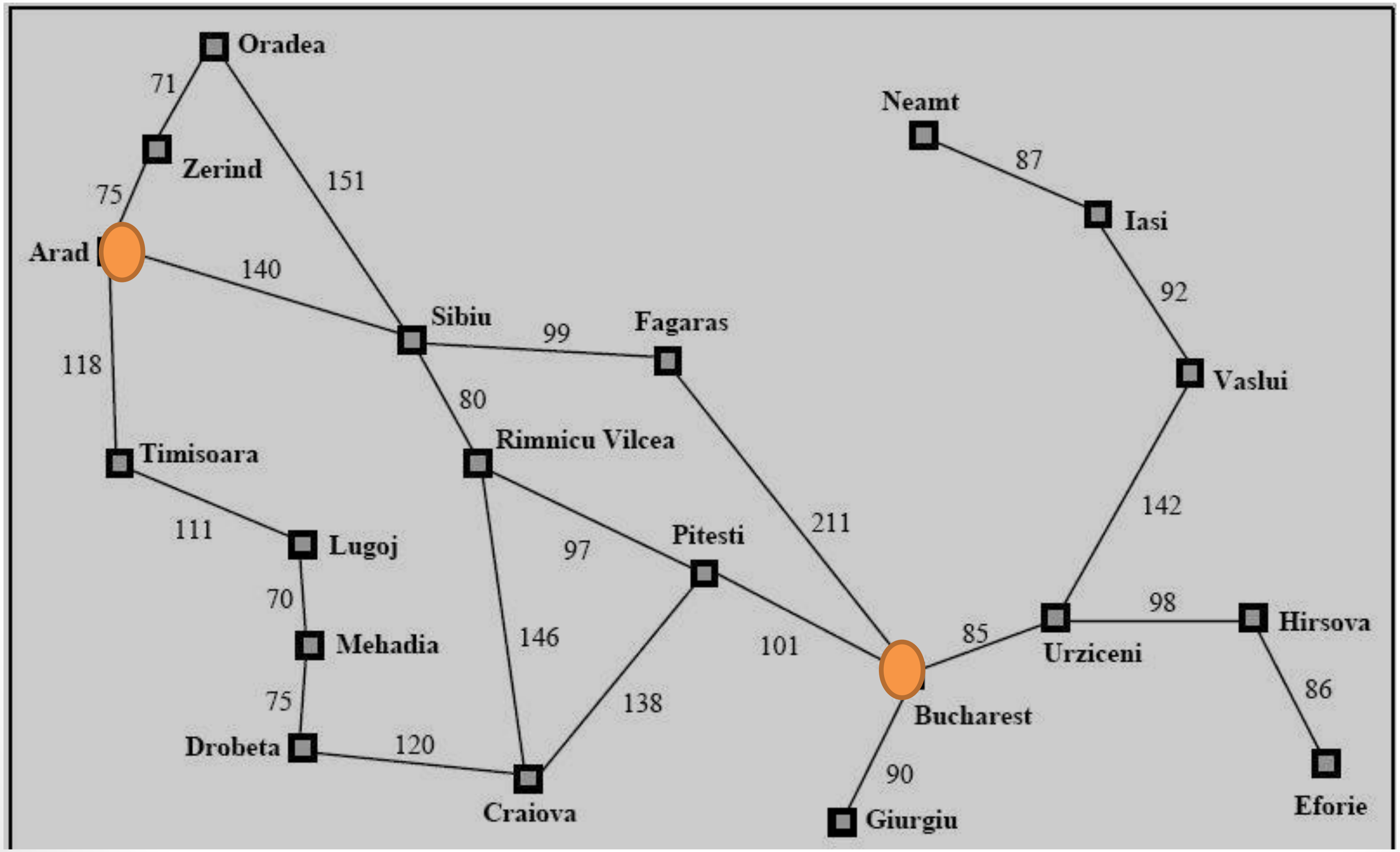
→ Depth-Limited Search

Example

- A,B,F,
- G,
- C,H,
- D,I

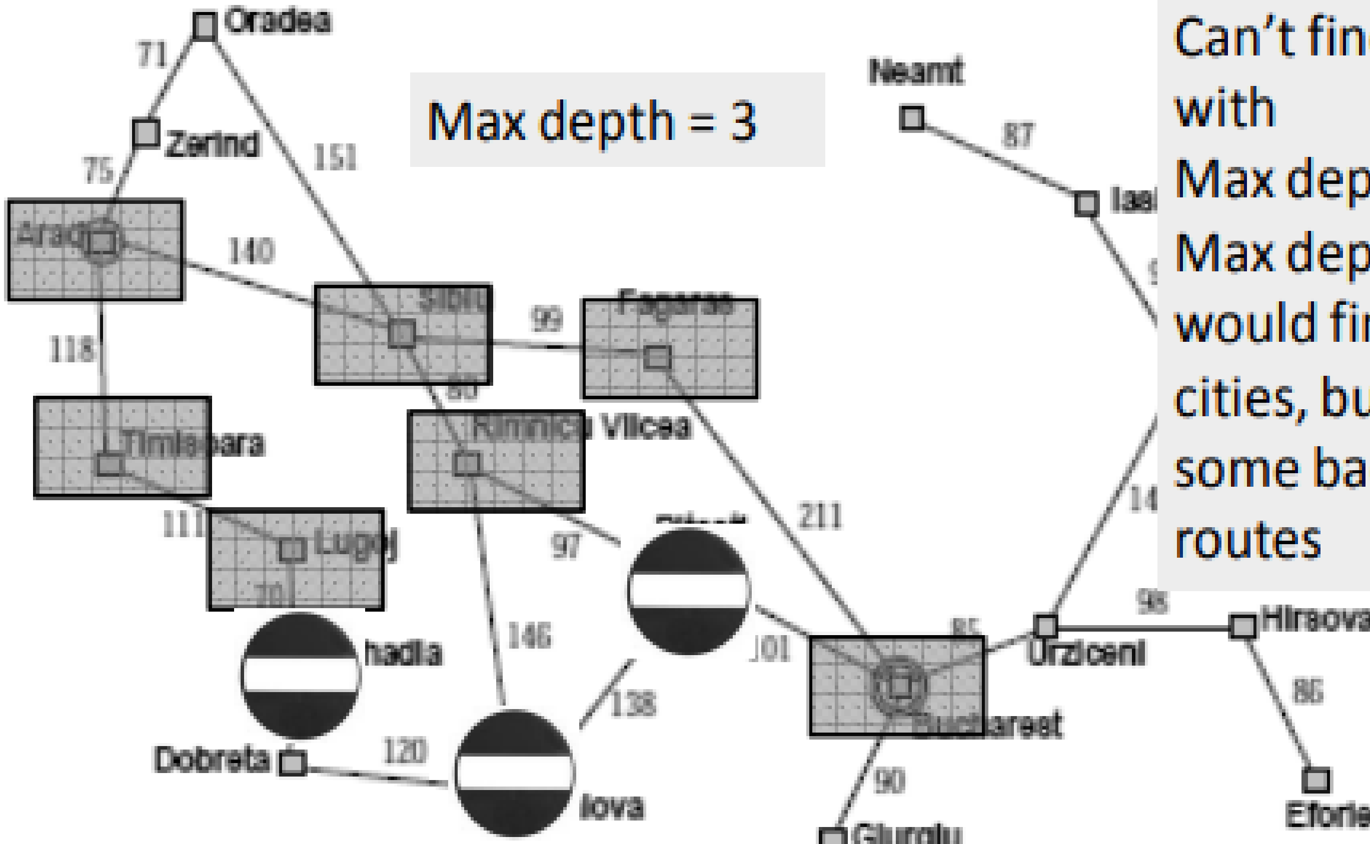


Example: Traveling from Arad To Bucharest



Example: Romania Problem

Only 20 cities on the map so no path longer than 19
In fact any city can reach any other in no more than 10 steps.



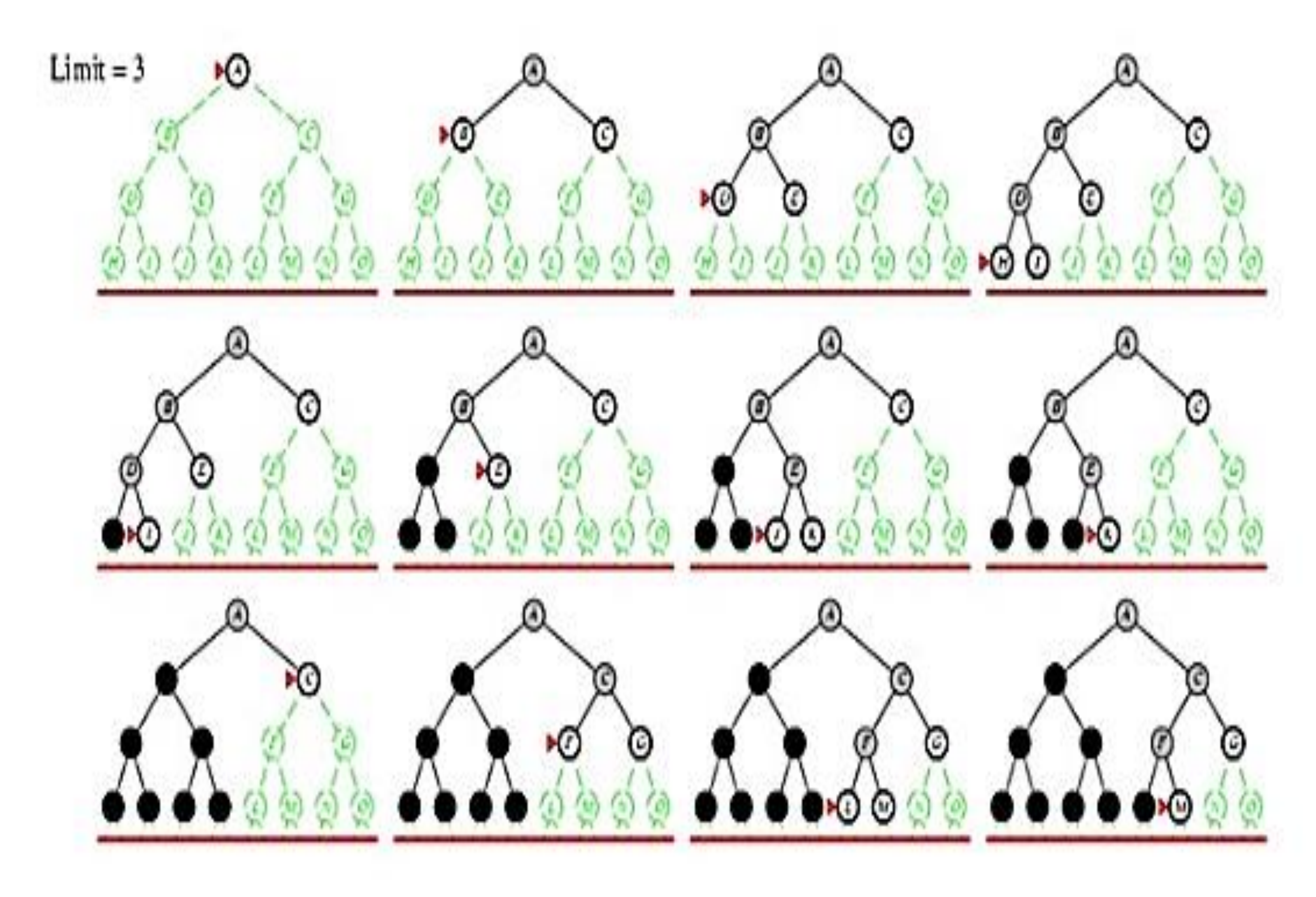
Max depth = 3

Can't find Eforie with
Max depth = 3;
Max depth = 9 would find all cities, but use some bad routes

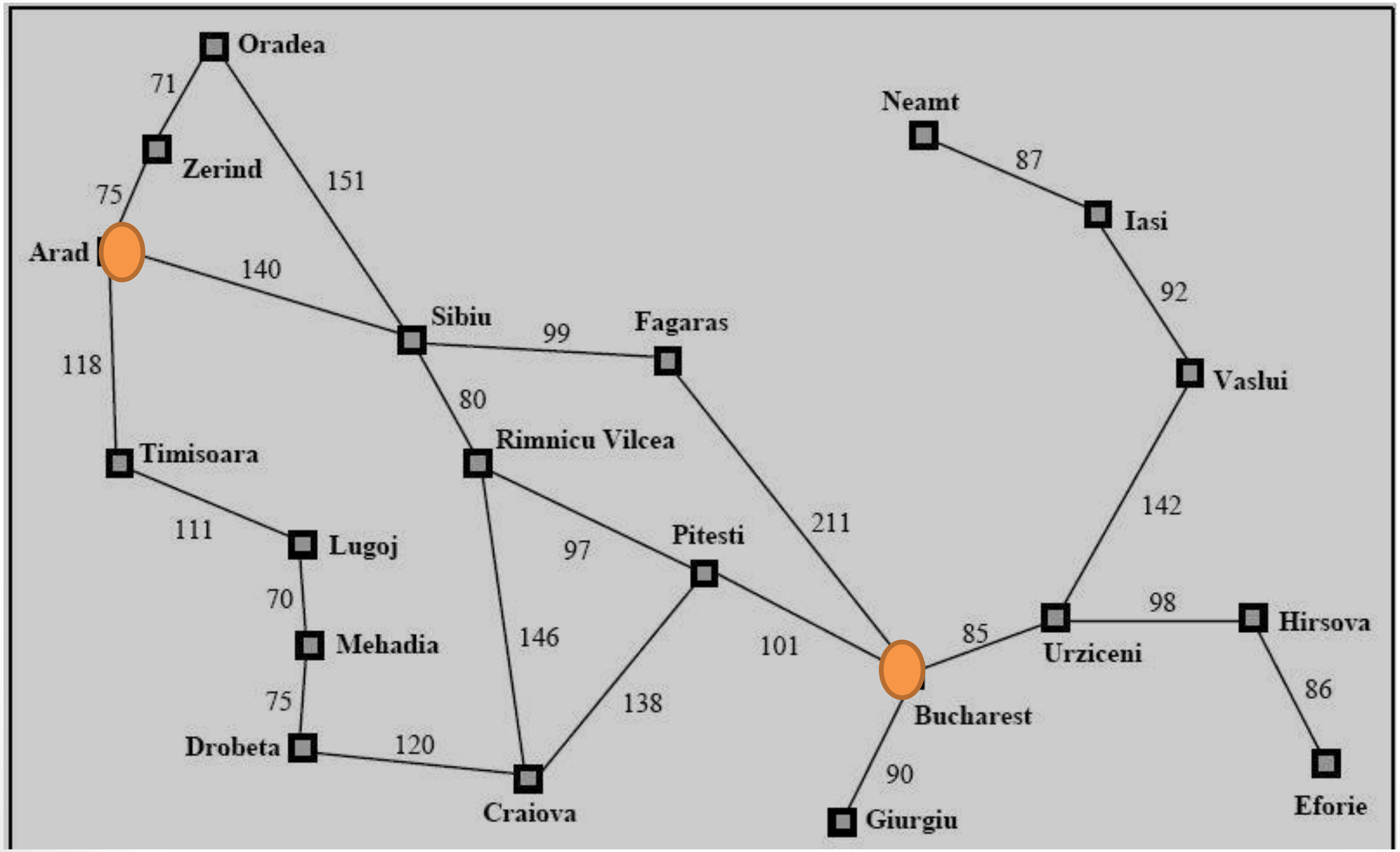
Iterative deepening depth first search

- What if solution is deeper than L? → Increase L iteratively.
  → Iterative Deepening Search
- This inherits the memory advantage of Depth-first search
- Better in terms of space complexity than Breadth-first search.
- Basic idea is:
  - do d.l.s. for depth n = 0; if solution found, return it;
  - otherwise do d.l.s. for depth n = n + 1; if solution found, return it, etc;
  - So we repeat d.l.s. for all depths until solution found.
- Useful if the search space is large and the maximum depth of the solution is not known.

Example

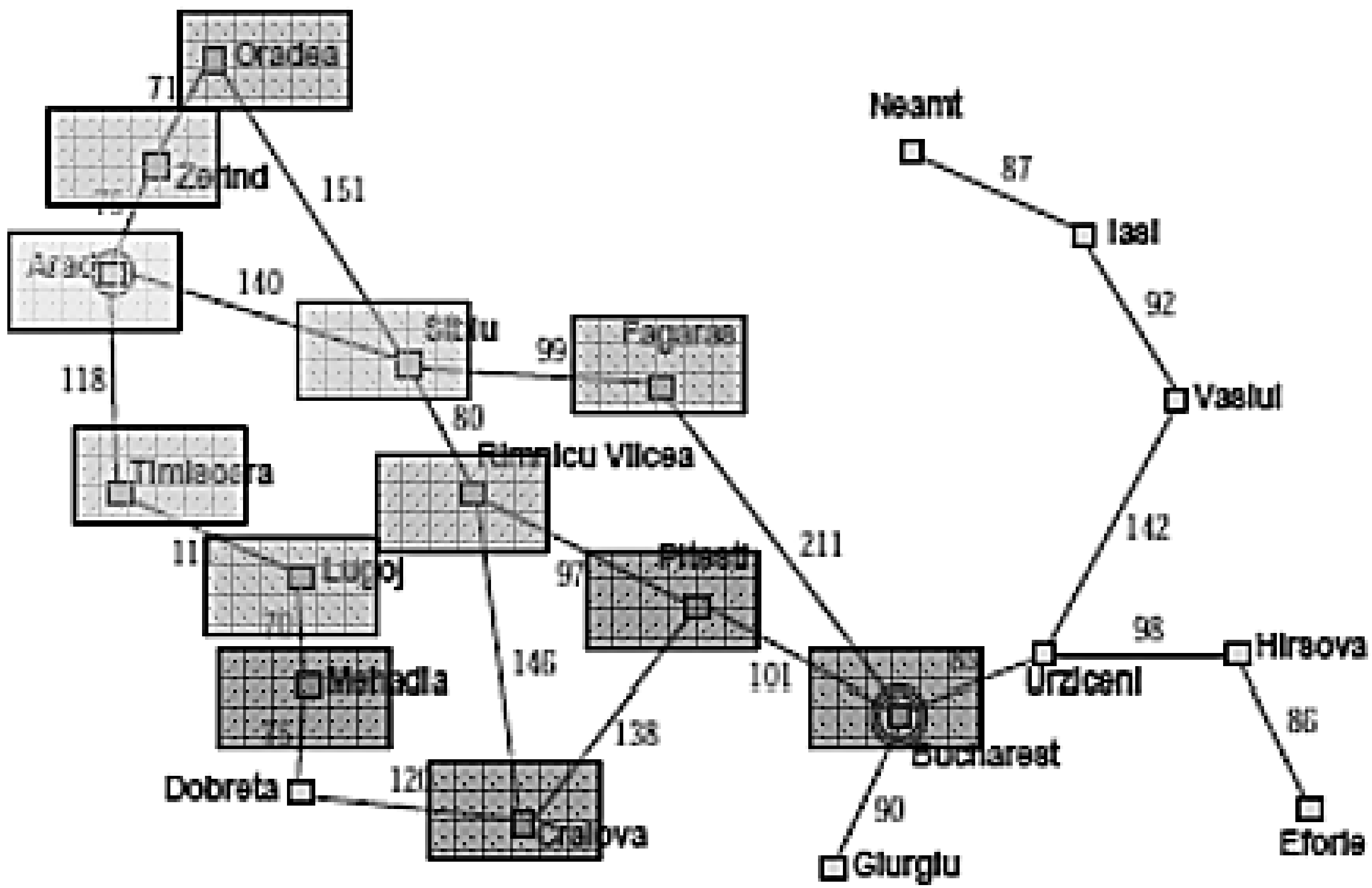


Example: Traveling from Arad To Bucharest



Example: Romania Problem

D =0 D =1 D =2 D =3



Properties of IDFS

- Complete? Yes
- Time? $O(b^d)$
- Space? $O(bd)$
- Optimal? No, for general cost functions.
  Yes, if cost is a non-decreasing function only of depth.

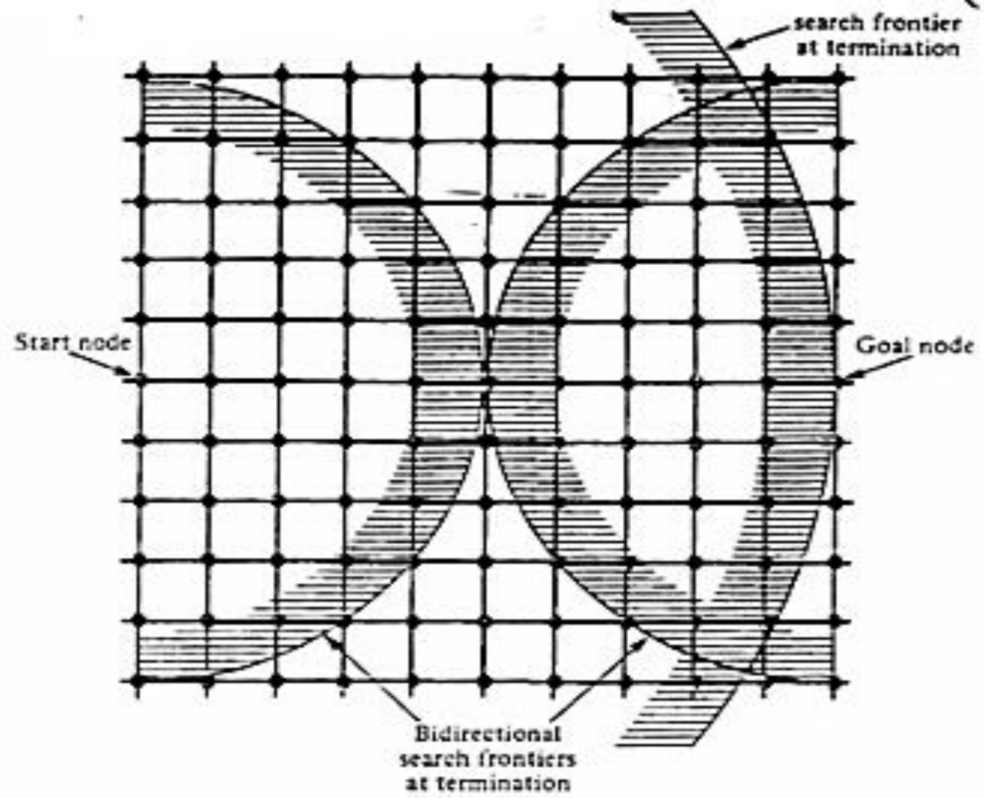
Generally the preferred uninformed search strategy.

Bi-directional Search

- Idea
  - simultaneously search forward from S and backwards from G
  - stop when both "meet in the middle"
  - need to keep track of the intersection of 2 open sets of nodes
- What does searching backwards from G mean
  - need a way to specify the predecessors of G
    - this can be difficult,
    - e.g., predecessors of checkmate in chess?
  - which to take if there are multiple goal states?
  - where to start if there is only a goal test, no explicit list?

Bi-directional Search

Complexity: time and space complexity are: $O(b^{d/2})$



Comparison of Uninformed Search

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

There are a number of footnotes, caveats, and assumptions.

[a] complete if b is finite
[b] complete if step costs $\geq \varepsilon > 0$
[c] optimal if step costs are all identical
(also if path cost non-decreasing function of depth only)
[d] if both directions use breadth-first search
(also if both directions use uniform-cost search with step costs $\geq \varepsilon > 0$)

Generally the preferred uninformed search strategy

Informed Search Strategies

- Heuristics
- Greedy best-first search
- A^* search
- Proof of A*

Informed Search Strategy

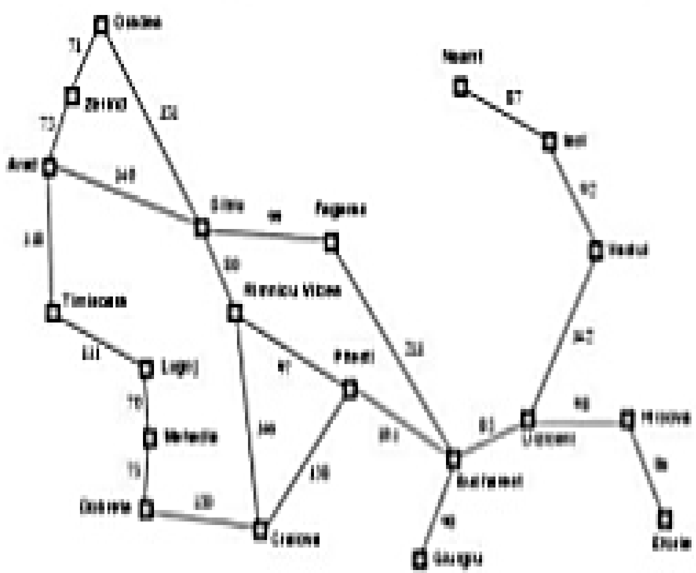
- We now consider informed search that uses problem-specific knowledge beyond the definition of the problem itself.
- This information helps to find solutions more efficiently than an uninformed strategy
- The information concerns the regularities of the state space
- An evaluation function *f(n)* determines how promising a node n in the search tree appears to be for the task of reaching the goal
- Traditionally, one aims at minimizing the value of function f

Heuristic Function

- A key component of an evaluation function is a heuristic function h(n)= which estimates the cost of the cheapest path from node n to a goal node
- Goal states are nevertheless identified: in a corresponding node n it is required that h(n) = 0
- If n is goal then h(n)=0
- Heuristic functions are the most common form in which additional knowledge is imported to the search algorithm

Romania with step costs in km

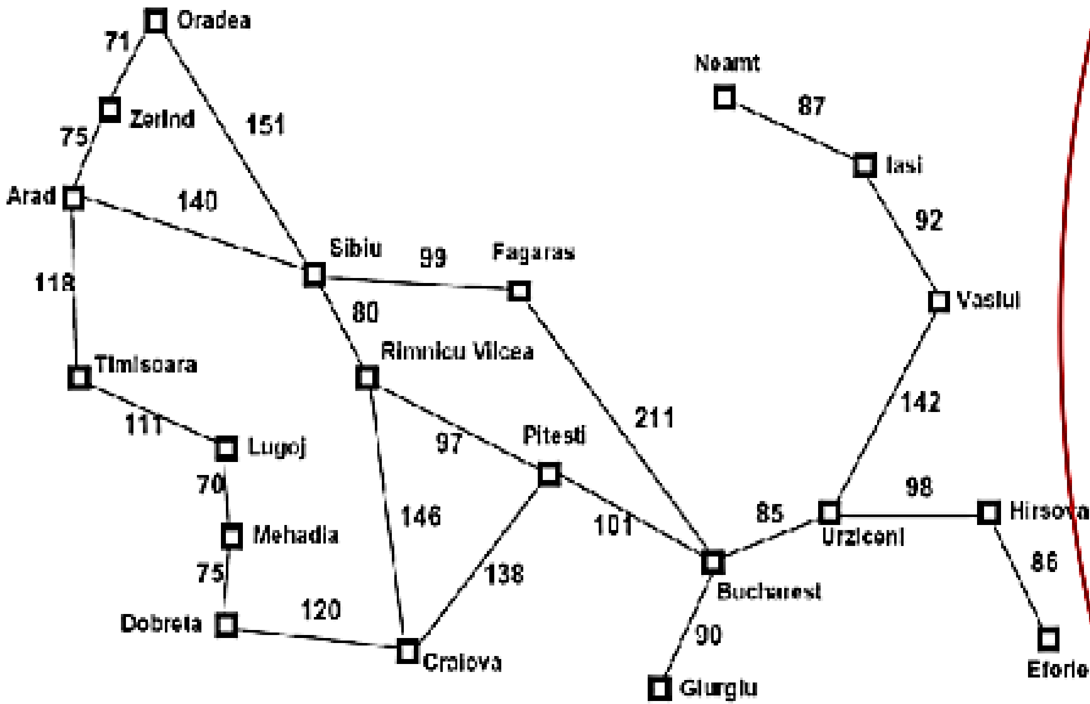
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374



- h_{SLD}=straight-line distance heuristic.
- h_{SLD} can **NOT** be computed from the problem description itself
- In this example $f(n)=h(n)$
  - **Expand node that is closest to goal**

= Greedy best-first search

Example: Heuristic Function



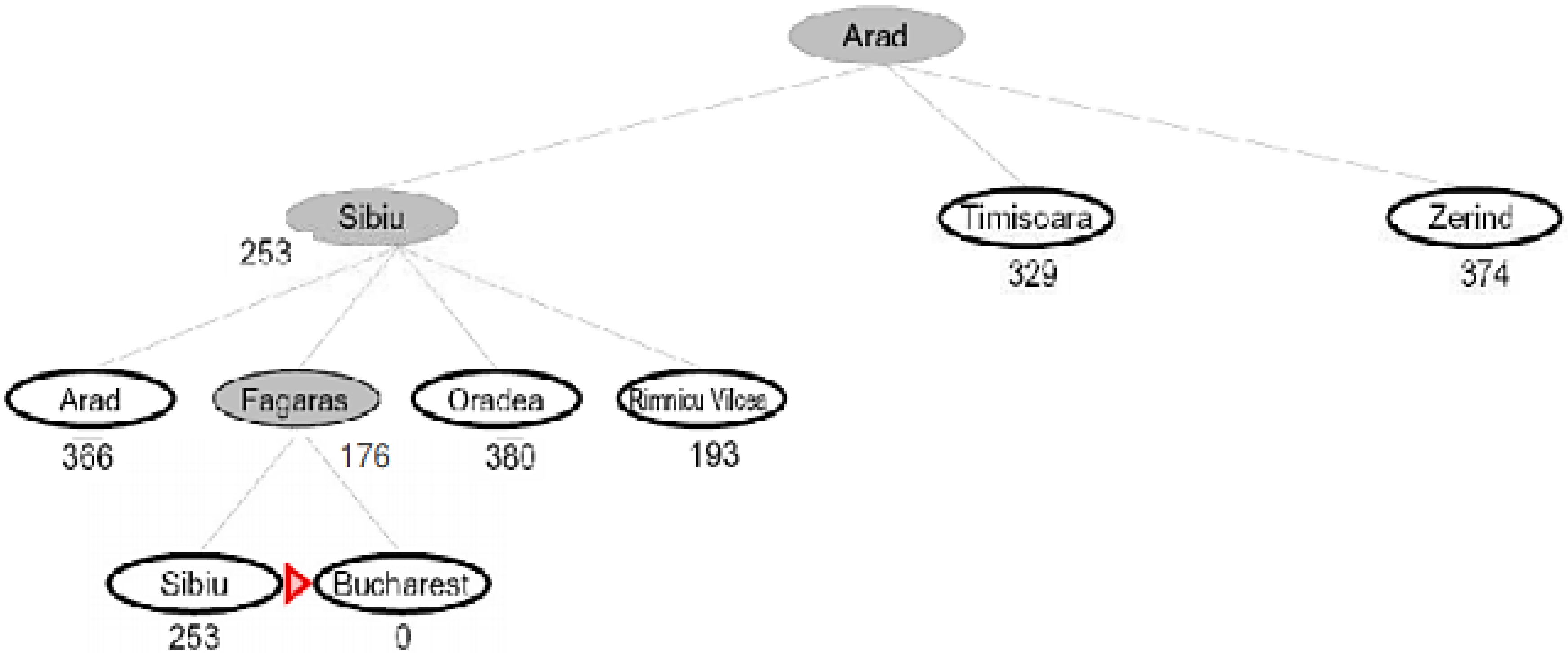
Straight−line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

$h(x)$

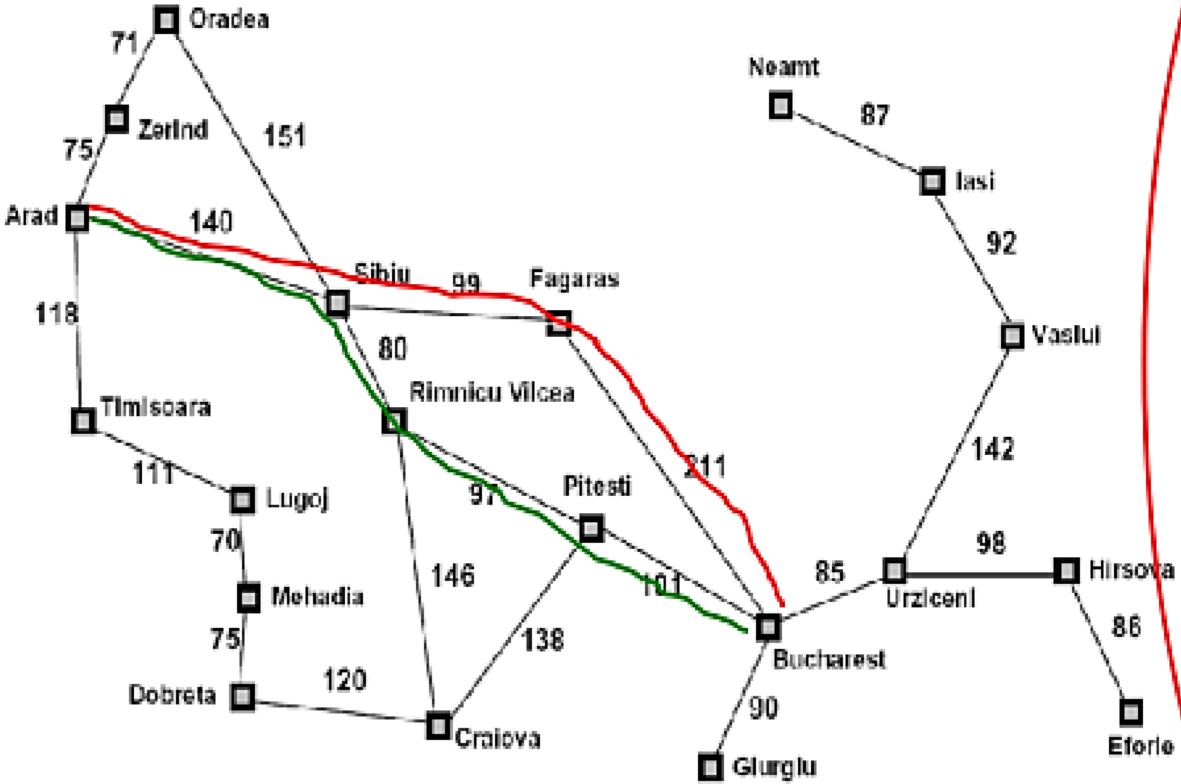
Greedy Best-First Search

- Strategy: expand a node that you think is closest to a goal state
  - Heuristic: estimate of distance to nearest goal for each state



- What can go wrong?

Example: Heuristic Function



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

$h(x)$

Red Path (GBF) = 450

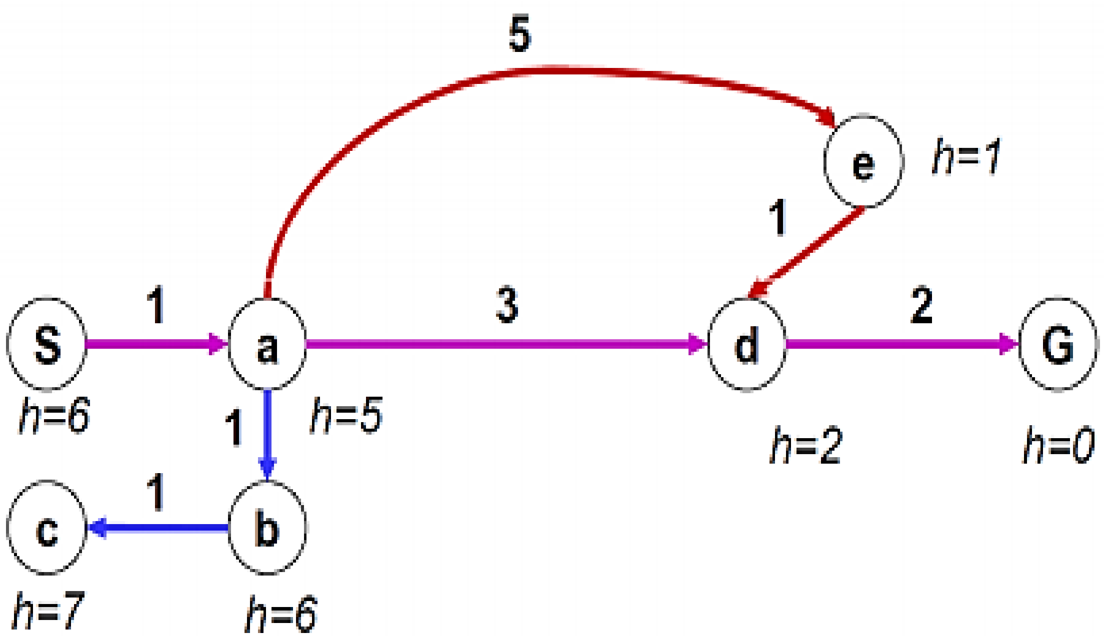
Green path(UCS)=418

GBFS Evaluation

- Completeness: NO (cfr. DF-search)
- Time complexity: $O(b^m)$
- Space complexity: $O(b^m)$
- Optimality? NO
  - **Same as DF-search**

Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* $g(n)$
- Greedy orders by goal proximity, or *forward cost* $h(n)$



UCS

s 0
s → a 1
s → a → b 2
s → a → d 4
s → a → e 6

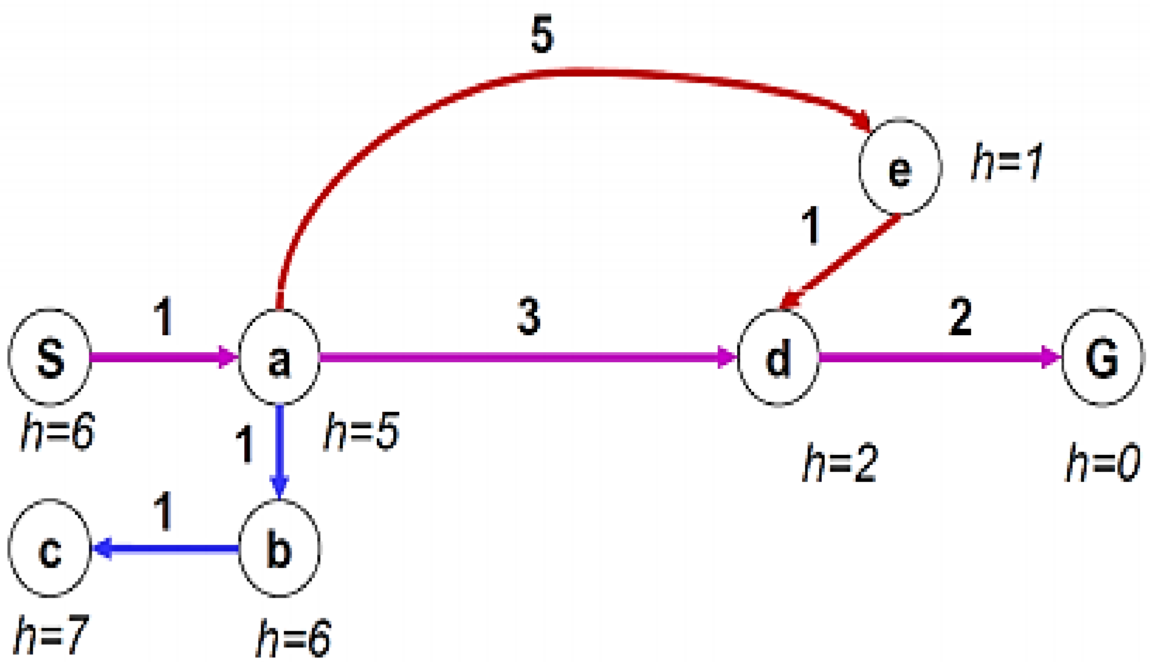
GBF

s 6
s → a 5
s → a → b 6
s → a → d 2
s → a → e 1
s → a → e → d 2

- A* Search orders by the sum: $f(n) = g(n) + h(n)$

Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* $g(n)$
- Greedy orders by goal proximity, or *forward cost* $h(n)$



A*

s 0+6

s → a 1+5

s → a → b 2+6

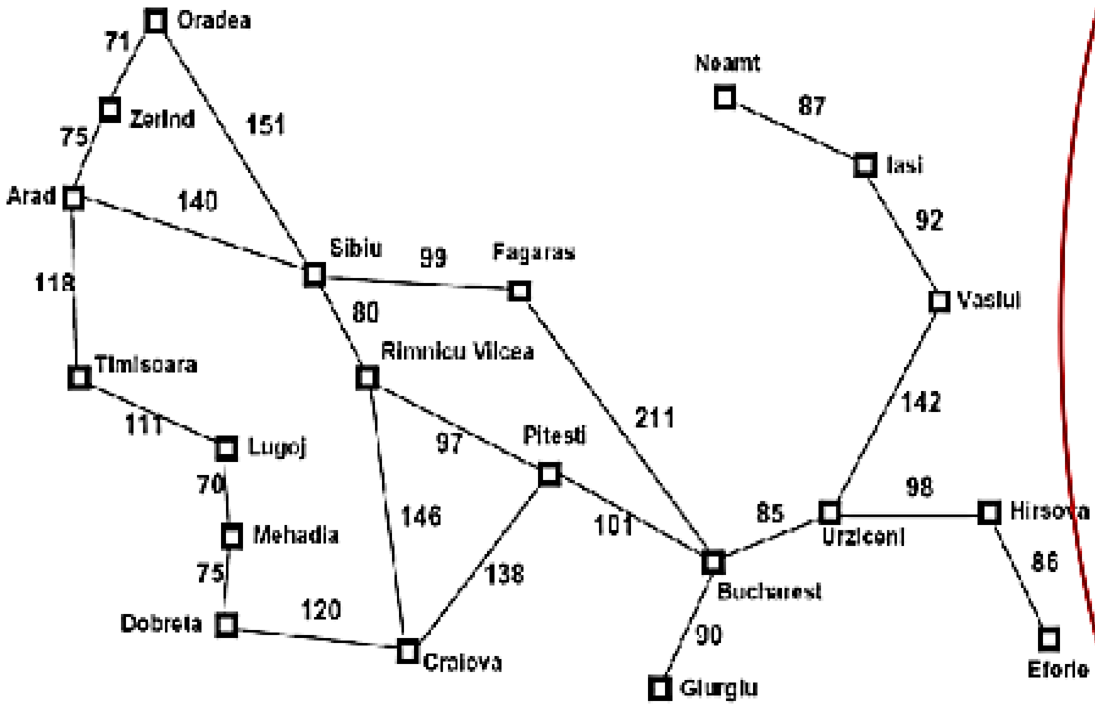
s → a → d 4+2

s → a → e 6+1

s → a → d → G 6+0

- A* Search orders by the sum: $f(n) = g(n) + h(n)$

Road Map to Romania g(n) and h(n)



Straight−line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

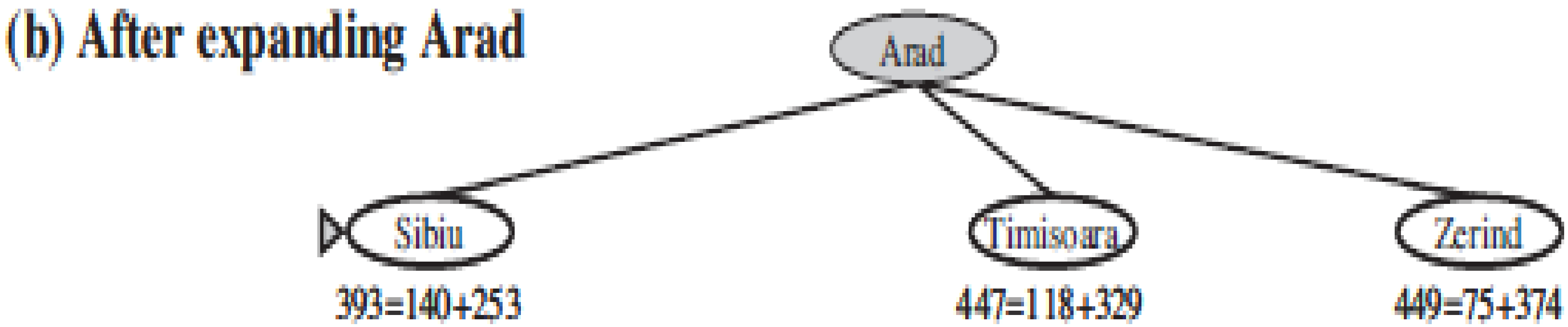
$h(x)$

A* search example

(a) The initial state

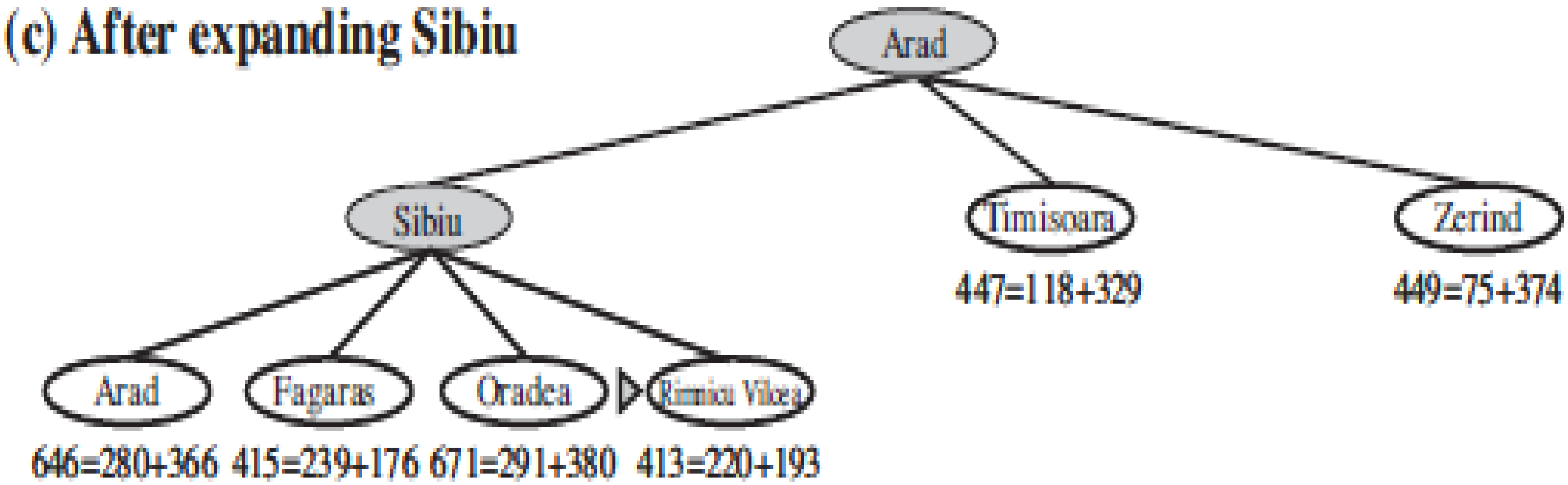
- Find Bucharest starting at Arad
  - **f(Arad) = c(??,Arad)+h(Arad)=0+366=366**

A* search example



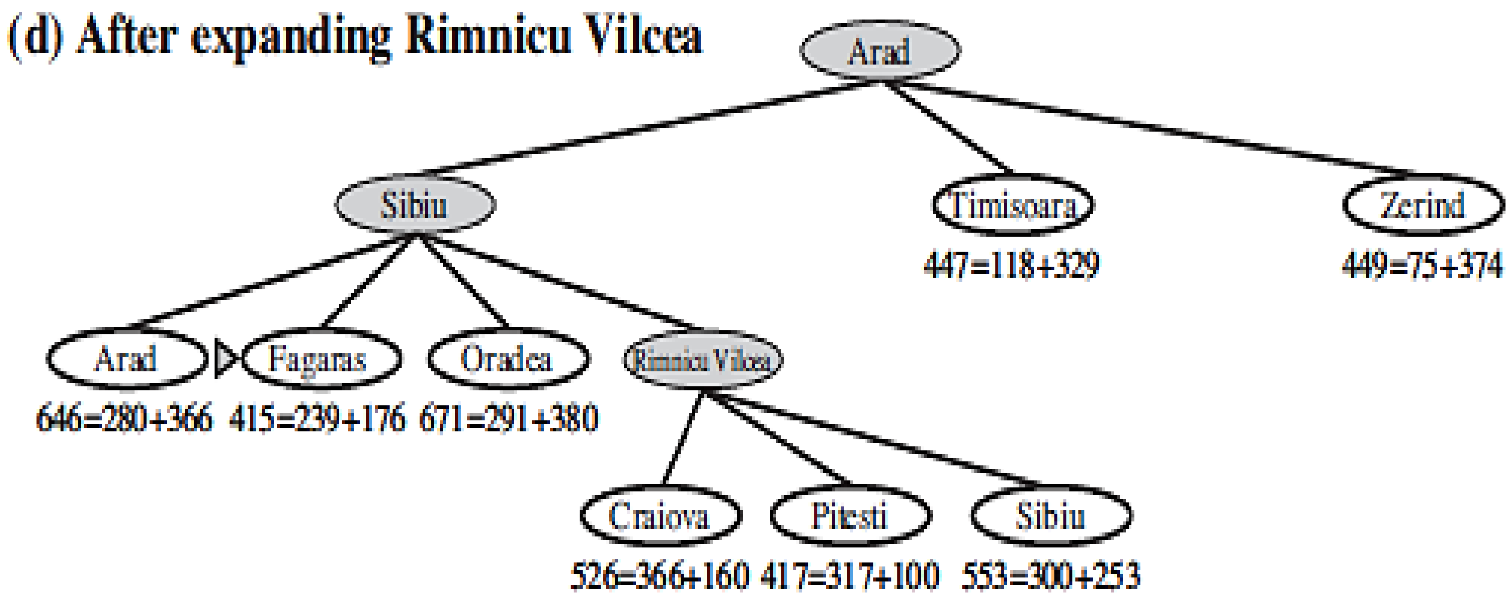
- Expand Arrad and determine *f(n)* for each node
  - **f(Sibiu)=c(Arad,Sibiu)+h(Sibiu)=140+253=393**
  - **f(Timisoara)=c(Arad,Timisoara)+h(Timisoara)=118+329=447**
  - **f(Zerind)=c(Arad,Zerind)+h(Zerind)=75+374=449**
- Best choice is Sibiu

A* search example



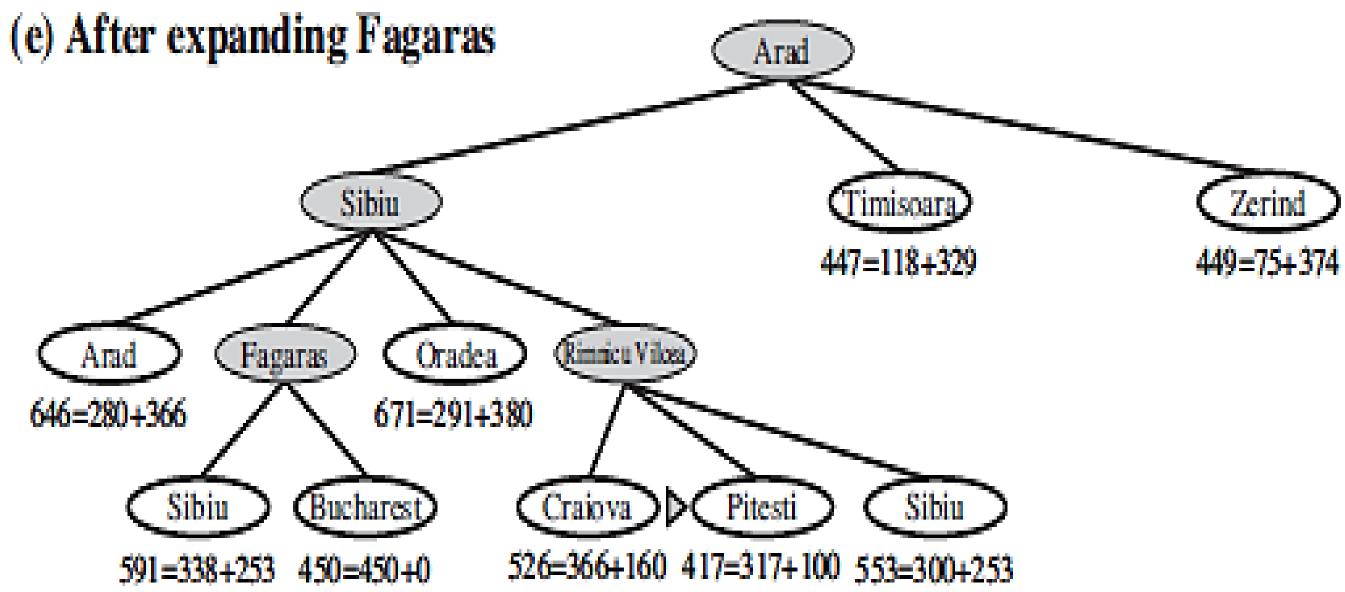
- Expand Sibiu and determine *f(n)* for each node
  - **f(Arad)=c(Sibiu,Arad)+h(Arad)=280+366=646**
  - **f(Fagaras)=c(Sibiu,Fagaras)+h(Fagaras)=239+179=415**
  - **f(Oradea)=c(Sibiu,Oradea)+h(Oradea)=291+380=671**
  - **f(Rimnicu Vilcea)=c(Sibiu,Rimnicu Vilcea)+ h(Rimnicu Vilcea)=220+192=413**
- Best choice is Rimnicu Vilcea

A* search example



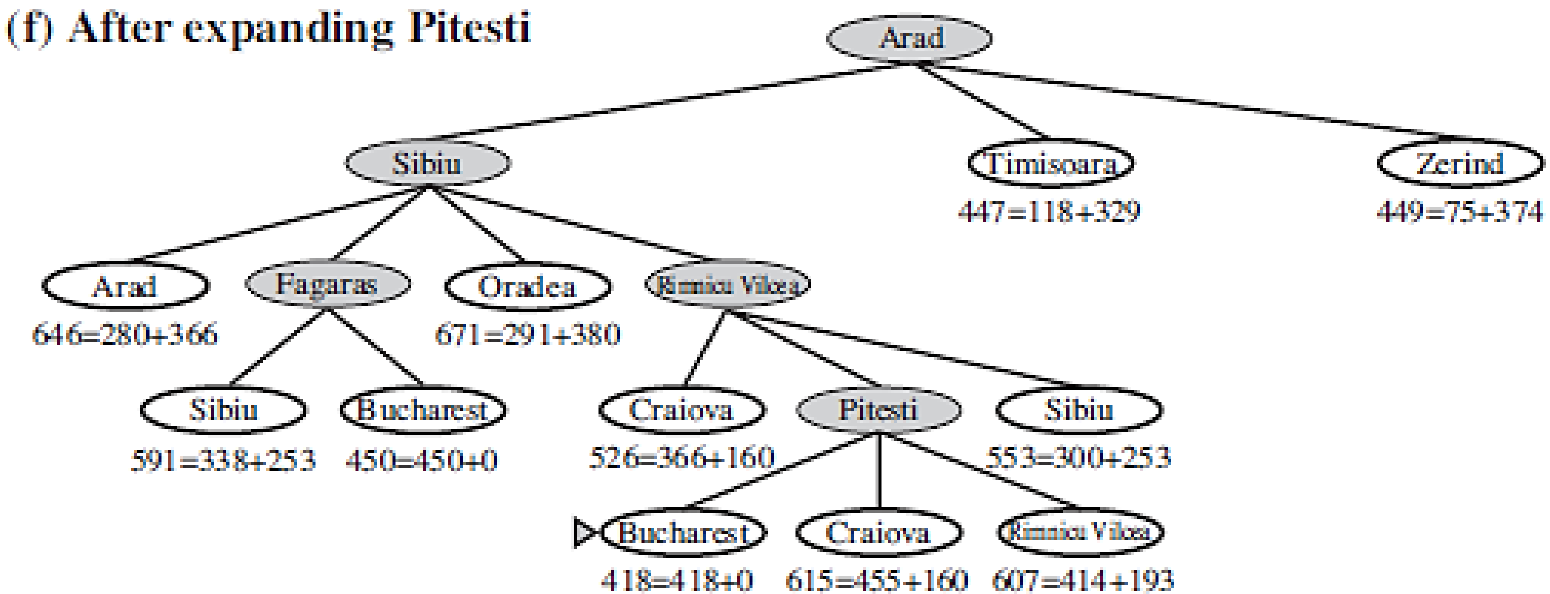
- Expand Rimnicu Vilcea and determine *f(n)* for each node
  - **f(Craiova)=c(Rimnicu Vilcea, Craiova)+h(Craiova)=360+160=526**
  - **f(Pitesti)=c(Rimnicu Vilcea, Pitesti)+h(Pitesti)=317+100=417**
  - **f(Sibiu)=c(Rimnicu Vilcea,Sibiu)+h(Sibiu)=300+253=553**
- Best choice is Fagaras

A* search example



- Expand Fagaras and determine *f(n)* for each node
  - **f(Sibiu)=c(Fagaras, Sibiu)+h(Sibiu)=338+253=591**
  - ***f(Bucharest)=c(Fagaras,Bucharest)+h(Bucharest)=450+0=450***
- Best choice is Pitesti !!!

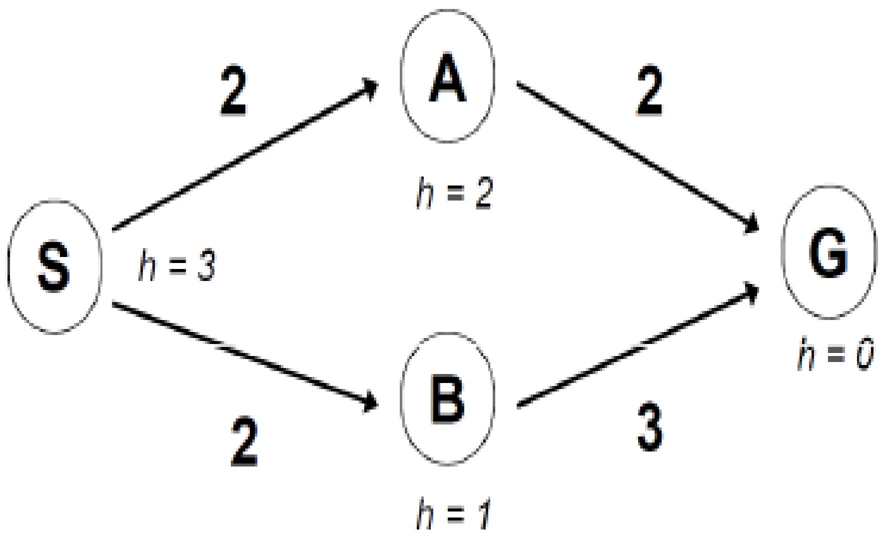
A* search example



- Expand Pitesti and determine *f(n)* for each node
  - ***f(Bucharest)=c(Pitesti,Bucharest)+h(Bucharest)=418+0=418***
- Best choice is Bucharest !!!
  - **Optimal solution (only if *h(n)* is admissable)**
- Note values along optimal path !!

When should A* terminate?

- Should we stop when we enter a goal in the frontier?



A*
S 0+3
S → A 2+2
S → B 2+1
S → B → G 5+0
S → A → G 4+0

- No: only stop when we select a goal for expansion

Admissible heuristics

- A heuristic $h(n)$ is *admissible* if it *never overestimates* the cost to reach the goal; i.e. it is *optimistic*
  - Formally: $\forall n$, n a node:
    1. $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from n
    2. $h(n) \geq 0$ so $h(G)=0$ for any goal G.

- *Example:* $h_{SLD}(n)$ never overestimates the actual road distance

Theorem: If $h(n)$ is *admissible*, A^* using Tree Search is *optimal*

Consistency

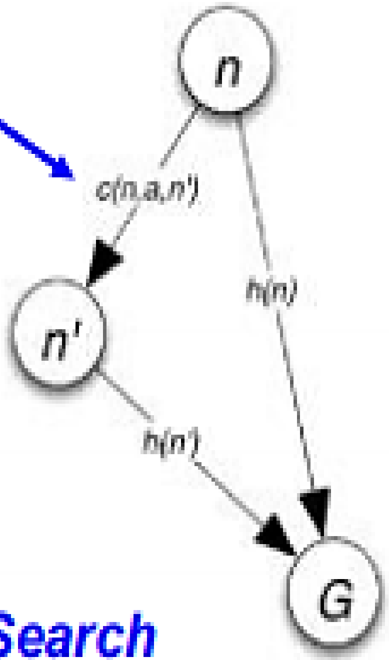
- A heuristic is ***consistent*** if

$$h(n) \le c(n,a,n') + h(n')$$

Cost of getting from n to n' by any action a

- Lemma: If h is consistent,

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n,a,n') + h(n') \\ &\ge g(n) + h(n) = f(n) \end{aligned}$$



i.e. f(n) is *nondecreasing* along any path.

Theorem: if h(n) is consistent, *A* using Graph-Search is optimal*

A* search, evaluation

- **Completeness: YES**
- **Time complexity: (exponential with path length)**
- **Space complexity:(all nodes are stored)**
- **Optimality: YES**
  - Cannot expand f_{i+1} until f_i is finished.
  - A* expands all nodes with $f(n)< f(G)$
  - A* expands one node with $f(n)=f(G)$
  - A* expands no nodes with $f(n)>f(G)$

Also *optimally efficient* (not including ties)