Problem solving and search

Outline

- ♢ Problem-solving agents
- ♢ Problem types
- ♢ Problem formulation
- ♢ Example problems
- ♢ Basic search algorithms

Problem-solving agents

Restricted form of general agent:

```
function SIMPLE-PROBLEM-SOLVING-AGENT(p) returns an action
  inputs: p, a percept
  static: s, an action sequence, initially empty
          state, some description of the current world state
          g, a goal, initially null
          problem, a problem formulation

  state ← UPDATE-STATE(state, p)
  if s is empty then
      g ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, g)
      s ← SEARCH(problem)
  action ← RECOMMENDATION(s, state)
  s ← REMAINDER(s, state)
  return action
```

Note: this is *offline* problem solving.
Online problem solving involves acting without complete knowledge of the problem and solution.

Example: Romania

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

Formulate goal:

be in Bucharest

Formulate problem:

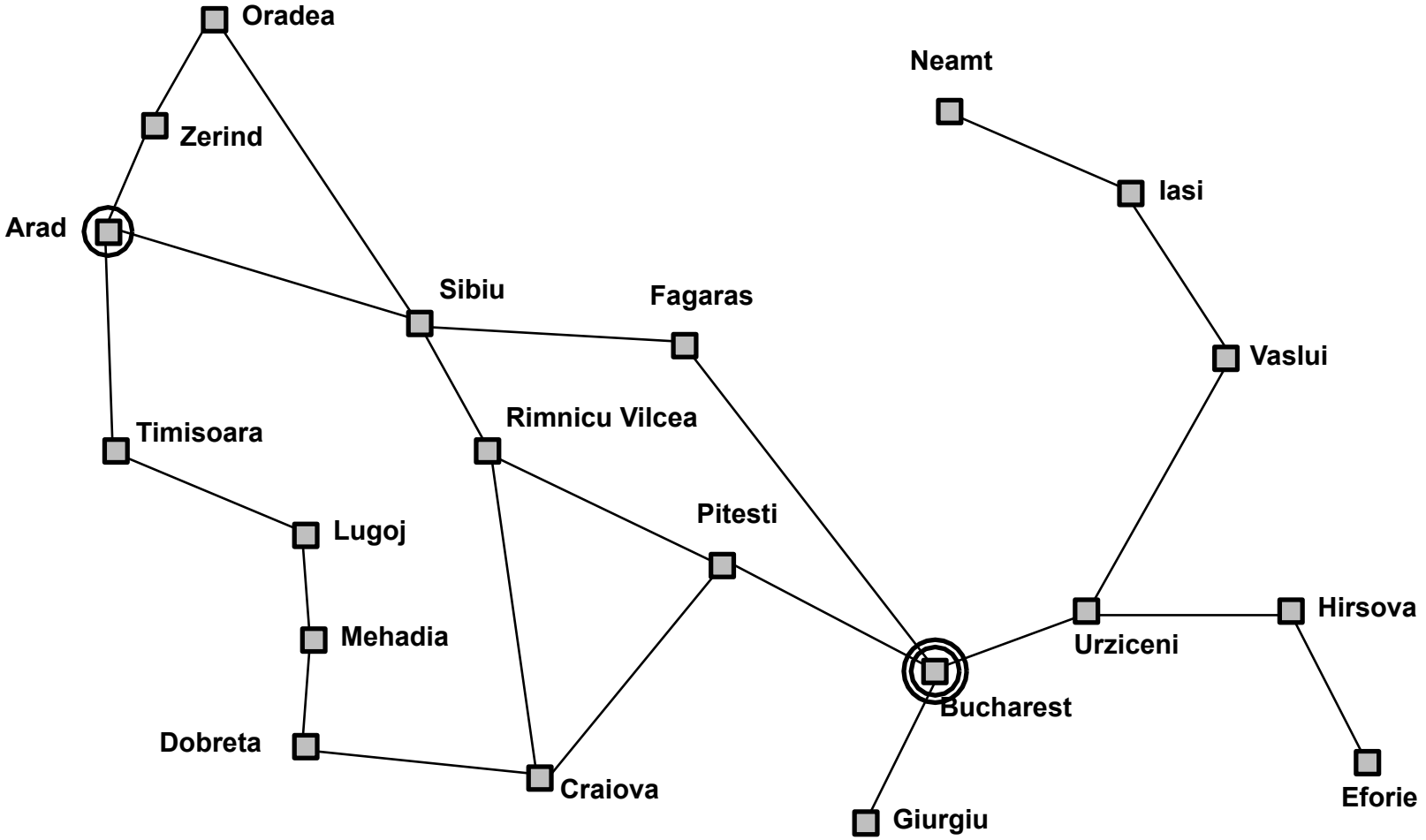
states: various cities

operators: drive between cities

Find solution:

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

Example: Romania



Problem types

Deterministic, accessible $\Longrightarrow$ *single-state problem*
Deterministic, inaccessible $\Longrightarrow$ *multiple-state problem*

Nondeterministic, inaccessible $\Longrightarrow$ *contingency problem*
- must use sensors during execution
- solution is a *tree* or *policy*
- often *interleave* search, execution

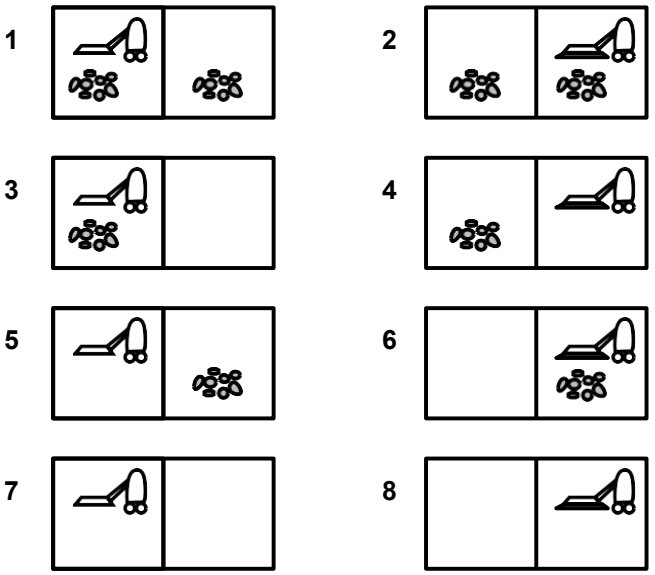
Unknown state space $\Longrightarrow$ *exploration problem* ("online")

Example: vacuum world

Single-state, start in #5. Solution??

Multiple-state, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ goes to $\{2, 4, 6, 8\}$. Solution??

Contingency, start in #5
Murphy's Law: $Suck$ can dirty a clean carpet
Local sensing: dirt, location only.
Solution??



Single-state problem formulation

A *problem* is defined by four items:

initial state e.g., "at Arad"

operators (or *successor function* $S(x)$)
e.g., Arad $\rightarrow$ Zerind Arad $\rightarrow$ Sibiu etc.

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 operators executed, etc.

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

Selecting a state space

Real world is absurdly complex
⇒ state space must be *abstracted* for problem solving

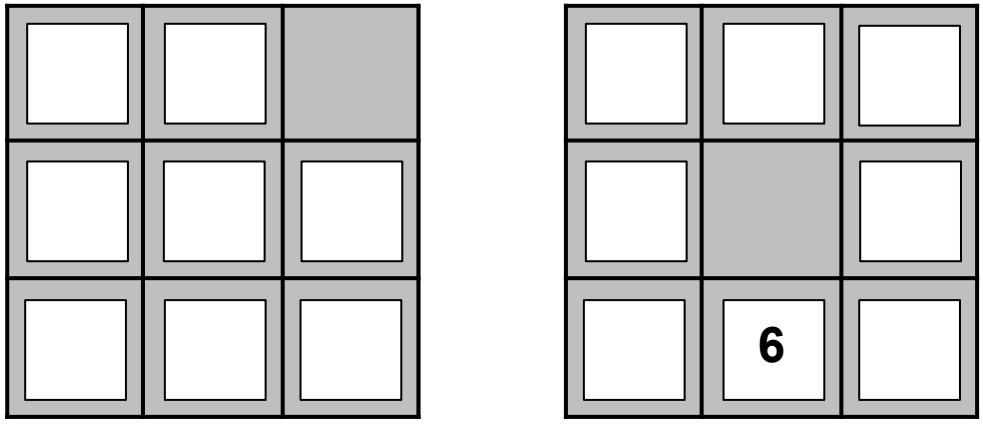
(Abstract) state = set of real states

(Abstract) operator = complex combination of real actions
e.g., "Arad → Zerind" represents a complex set
of possible routes, detours, rest stops, etc.
For guaranteed realizability, any real state "in Arad"
must get to *some* real state "in Zerind"

(Abstract) solution =
set of real paths that are solutions in the real world

Each abstract action should be "easier" than the original problem!

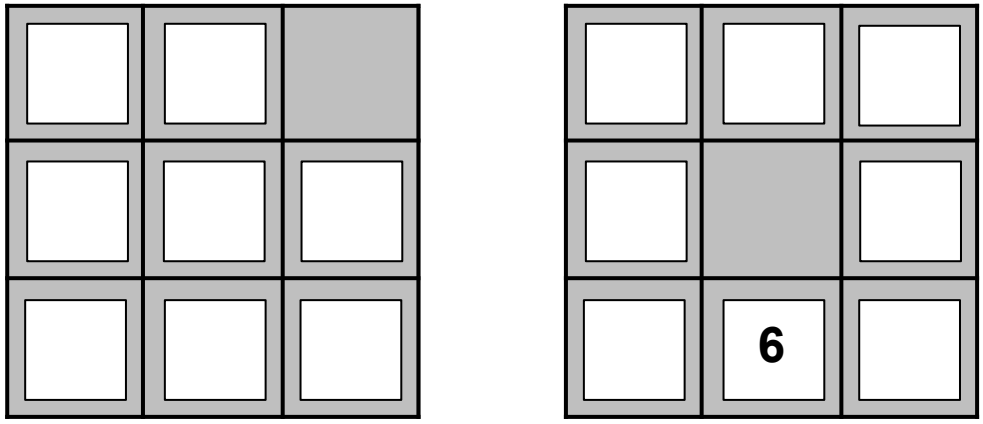
Example: The 8-puzzle



Start State **Goal State**

states??
operators??
goal test??
path cost??

Example: The 8-puzzle

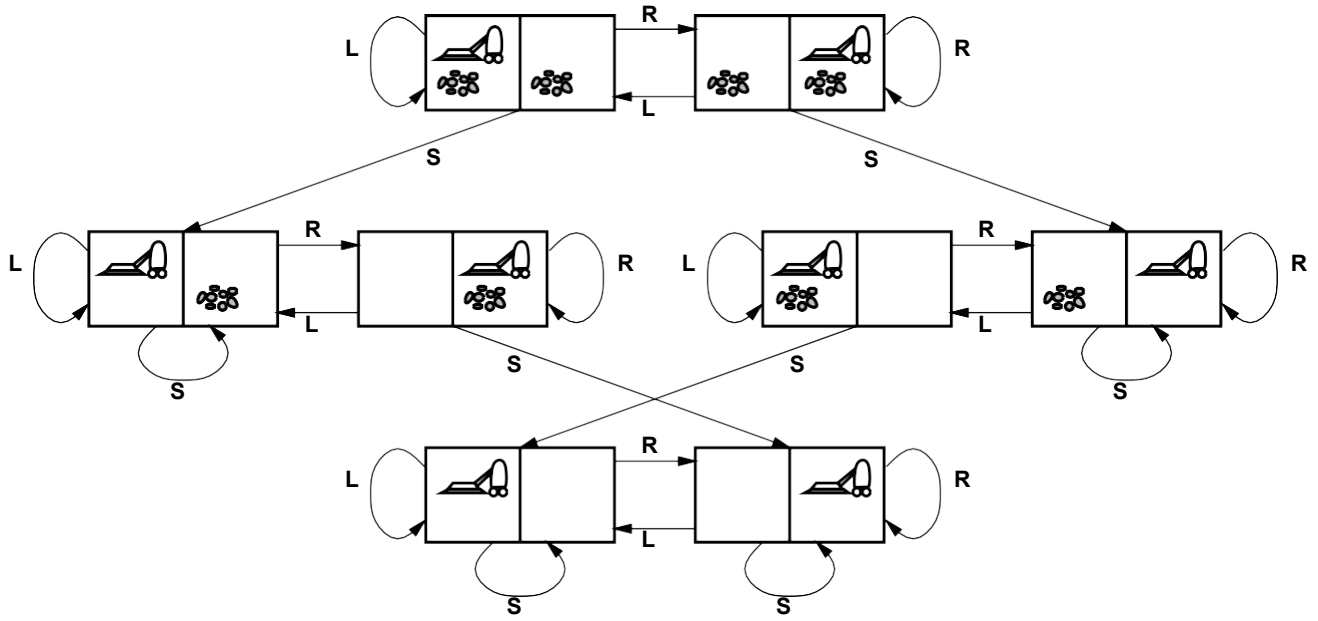


Start State Goal State

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

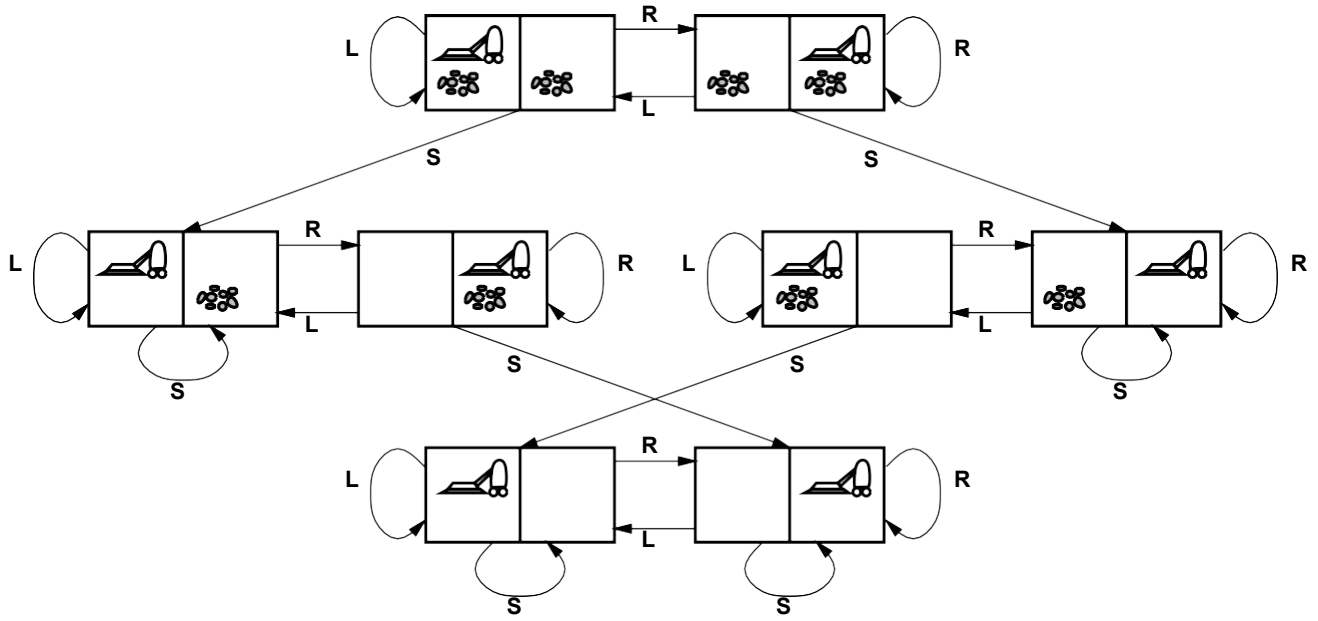
[Note: optimal solution of n-Puzzle family is NP-hard]

Example: vacuum world state space graph



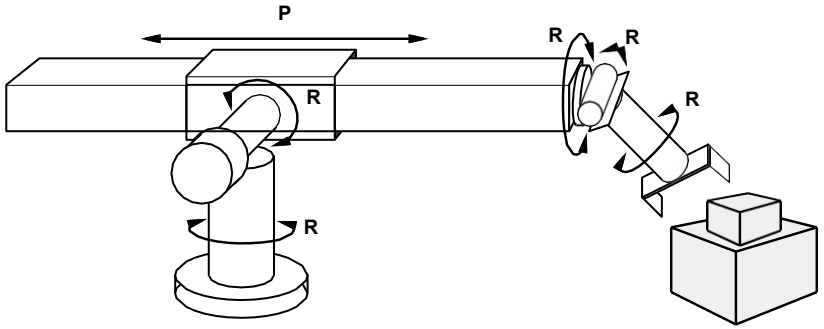
states??
operators??
goal test??
path cost??

Example: vacuum world state space graph



states??: integer dirt and robot locations (ignore dirt *amounts*)
operators??: *Left*, *Right*, *Suck*
goal test??: no dirt
path cost??: 1 per operator

Example: robotic assembly



states??: real-valued coordinates of
robot joint angles
parts of the object to be assembled

operators??: continuous motions of robot joints

goal test??: complete assembly *with no robot included!*

path cost??: time to execute

Search algorithms

Basic idea:

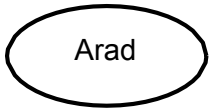
offline, simulated exploration of state space

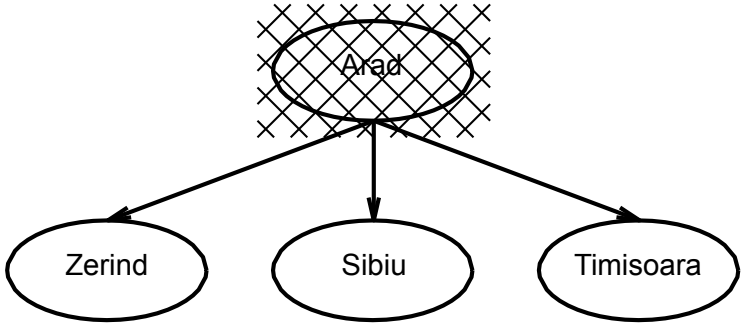
by generating successors of already-explored states

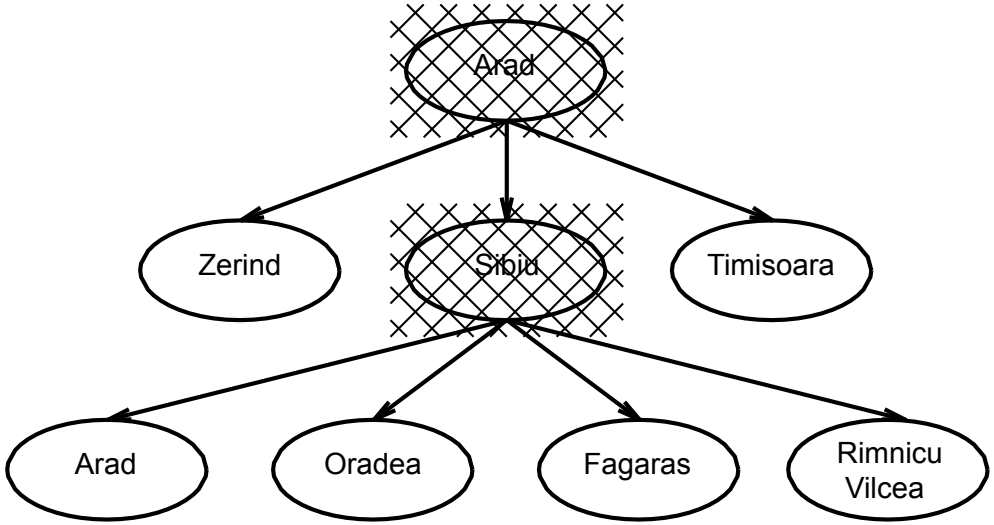
(a.k.a. *expanding* states)

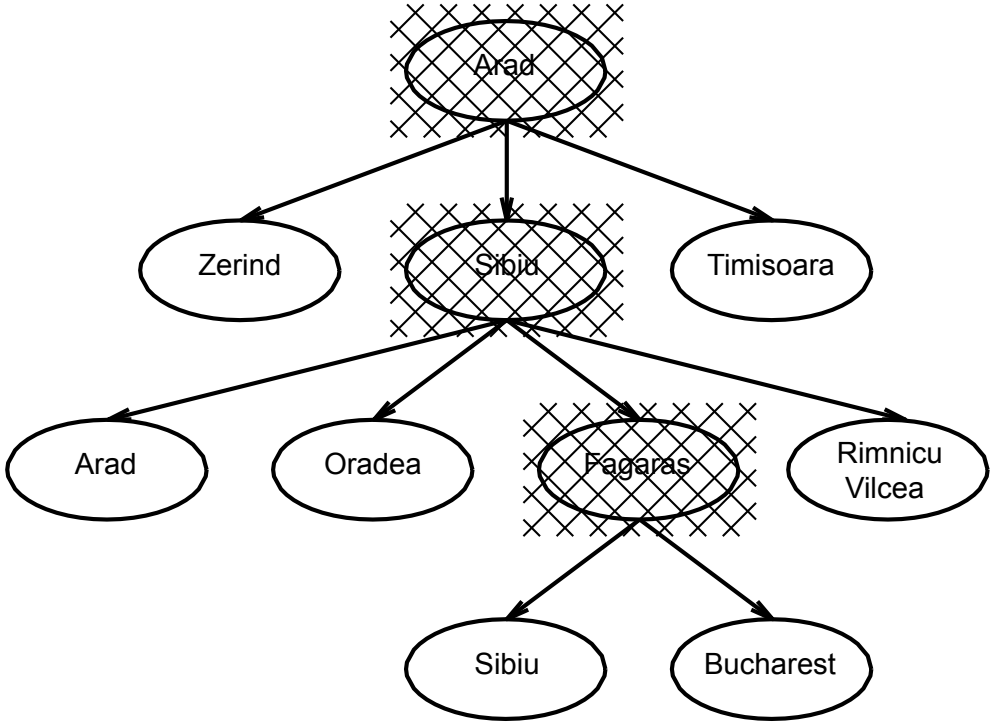
```
function GENERAL-SEARCH( problem, strategy) returns a solution, or failure
   initialize the search tree using the initial state of problem
   loop do
      if there are no candidates for expansion then return failure
      choose a leaf node for expansion according to strategy
      if the node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

General search example




Arad
Zerind
Sibiu
Timisoara


Arad
Zerind
Sibiu
Timisoara
Arad
Oradea
Fagaras
Rimnicu
Vilcea

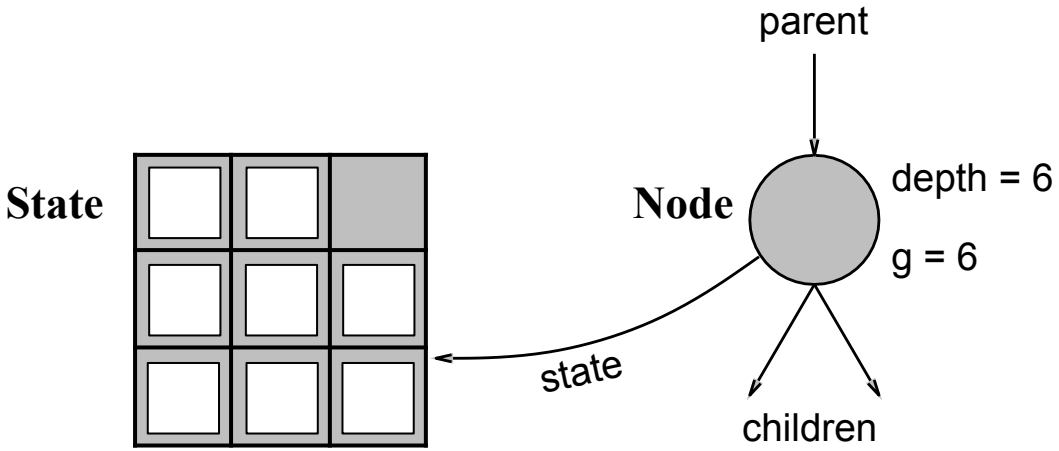

Arad
Zerind
Sibiu
Timisoara
Arad
Oradea
Fagaras
Rimnicu
Vilcea
Sibiu
Bucharest

Implementation of search algorithms

```
function General-Search( problem, Queuing-Fn) returns a solution, or failure
    nodes ← Make-Queue(Make-Node(Initial-State[problem]))
    loop do
        if nodes is empty then return failure
        node ← Remove-Front(nodes)
        if Goal-Test[problem] applied to State(node) succeeds then return node
        nodes ← Queuing-Fn(nodes, Expand(node, Operators[problem]))
    end
```

Implementation contd: states vs. 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 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 ∞)

Measuring problem-solving performance

- Graph is an explicit data structure that is input to the search program.
- The typical measure is the size of the state space graph, |V | + |E|
- where V is the set of vertices (nodes) of the graph
- E is the set of edges (links).
- In AI, the graph is often represented implicitly by the initial state, actions, and transition model

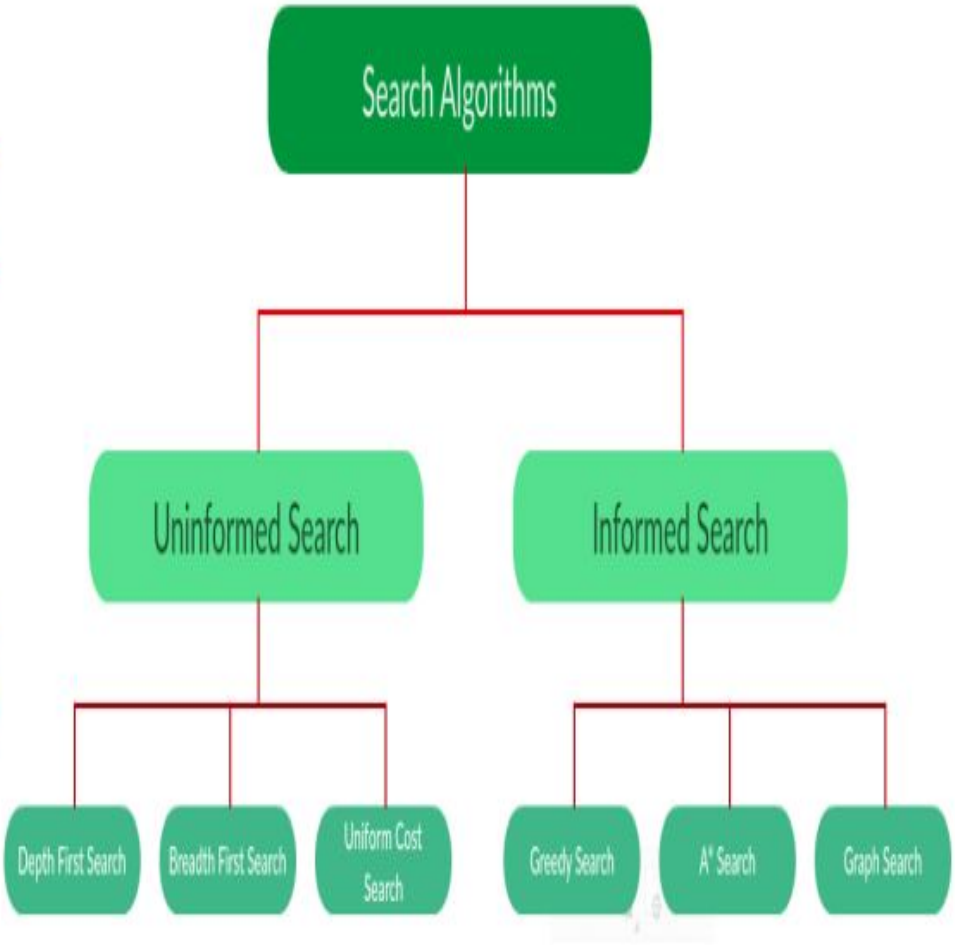
Search Strategies

Uninformed search (Blind search)

- No information about the number of steps or the path cost from the current state to the goal
- Search the state space **blindly**

Informed search, or heuristic search

- A cleverer strategy that searches toward the goal, based on the information from the current state so far



Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

Breadth-first search

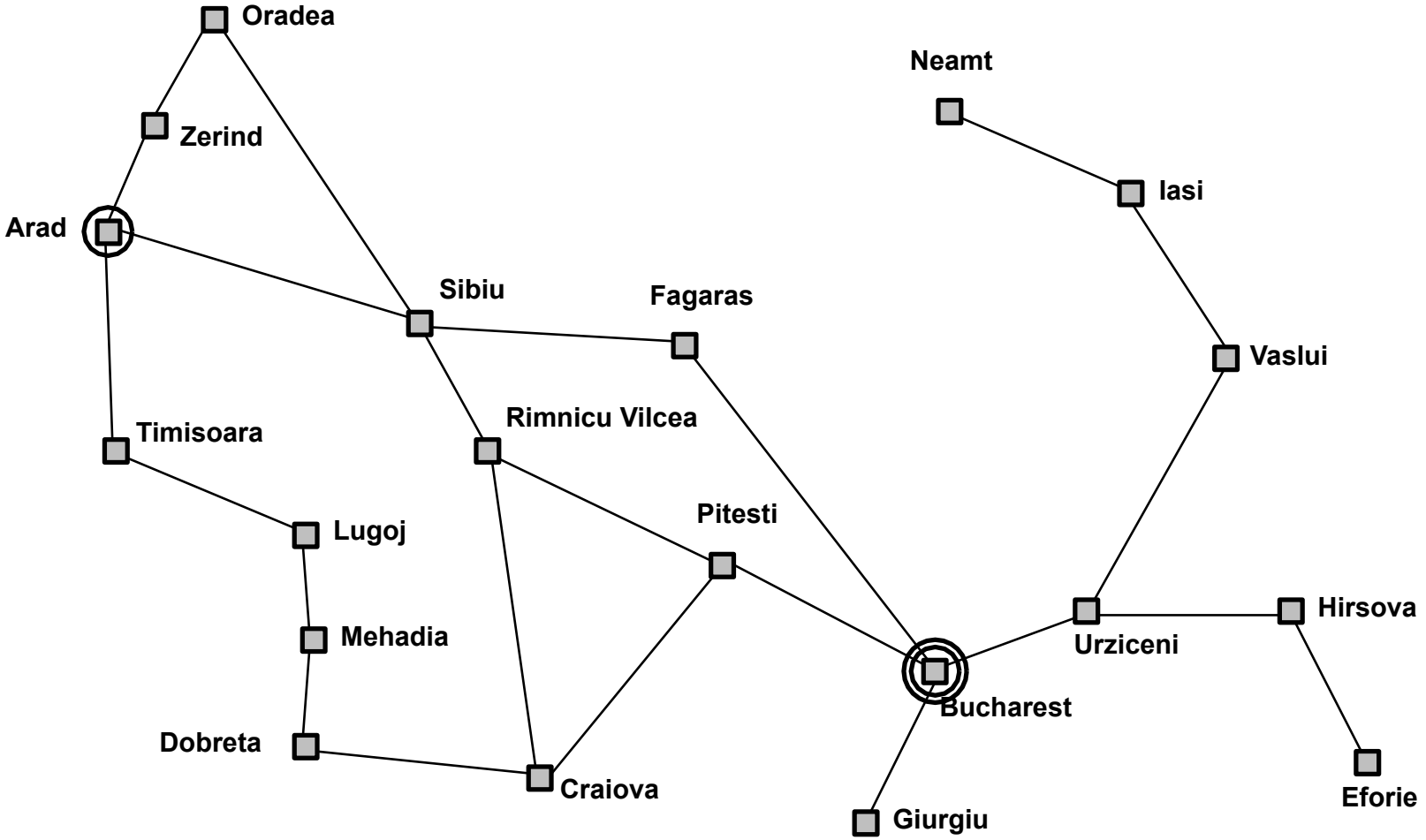
Uniform-cost search **Or Dijkstra's algorithm**

Depth-first search

Depth-limited search

Iterative deepening search

Example: Romania



Breadth-first search

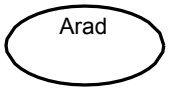
Expand shallowest unexpanded node : **"In which the root node is expanded first, then all the successors of the root node are expanded next"**

Implementation:

QUEUEINGFN = put successors at end of queue (faster than a priority queue)

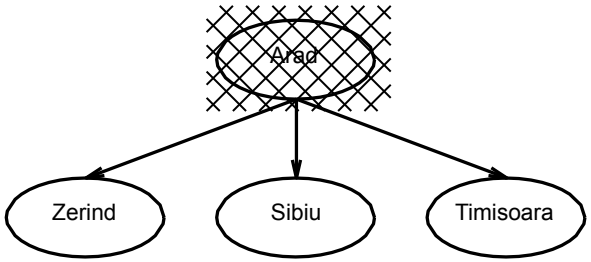
will give us the correct order of nodes

Step 1

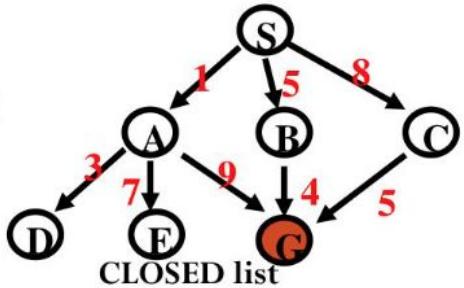


- Function f(n) is the depth of the node, that is, the number of actions it takes to reach the node

Step 2



Breadth-First Search

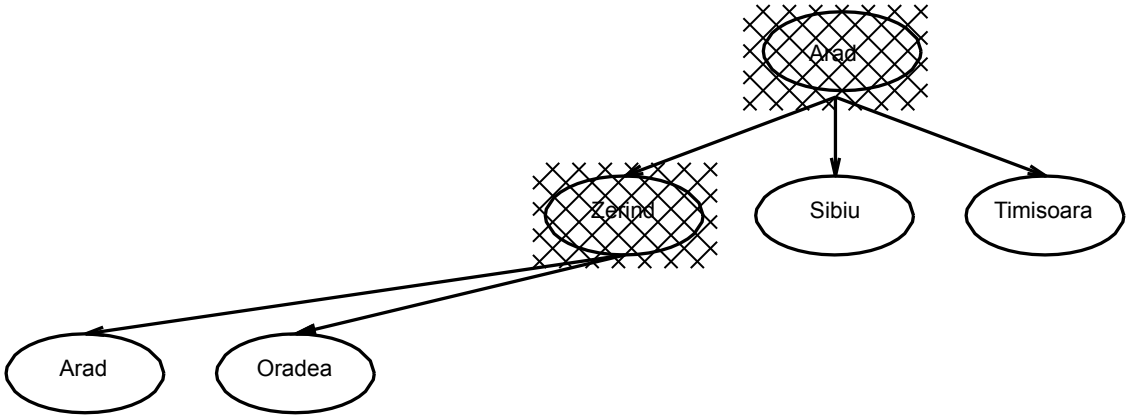


exp. node	OPEN list	CLOSED list
	{S}	{}
S	{A B C}	{S}
A	{B C D E G}	{S A}
B	{C D E G G'}	{S A B}
C	{D E G G' G"}	{S A B C}
D	{E G G' G"}	{S A B C D}
E	{G G' G"}	{S A B C D E}
G	{G' G"}	{S A B C D E}

Solution path found is S A G <-- this G also has cost 10

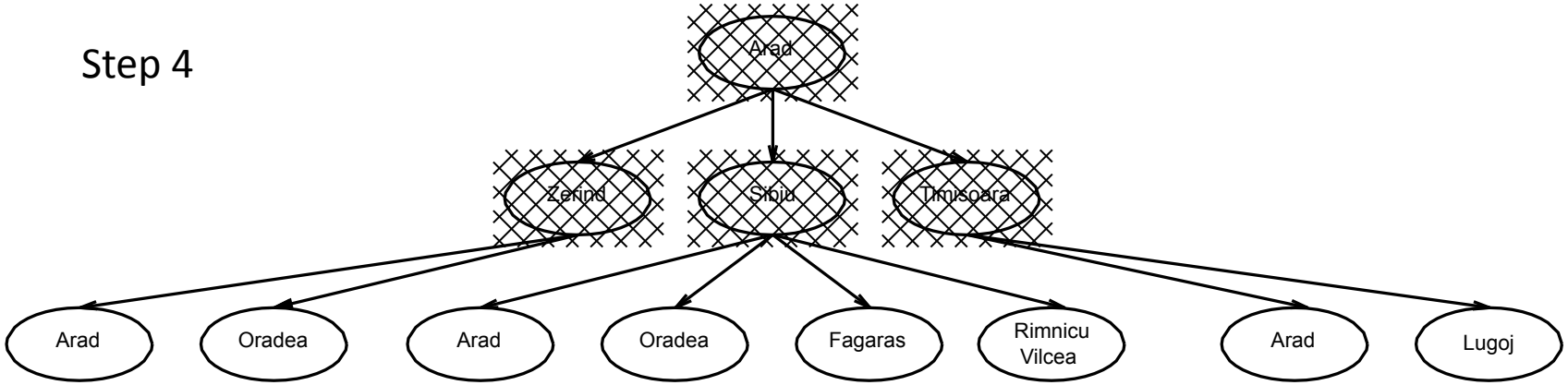
Number of nodes expanded (including goal node) = 7

Step 3



once we've reached a state, Early goal test we can never find a better path to the state

Step 4



Properties of breadth-first search

Complete??

Time??

Space??

Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

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

Space?? $O(b^d)$ (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 1MB/sec
so 24hrs = 86GB.

The memory requirements are a bigger problem for breadth-first than the execution time.

But time is still an important factor. At depth d = 14, even with search infinite memory, the search would take 3.5 years.

In general, problems cannot be solved by uninformed search for any but the exponential-complexity search problems cannot be solved by uninformed search for any but the smallest instances.

Breadth-first search

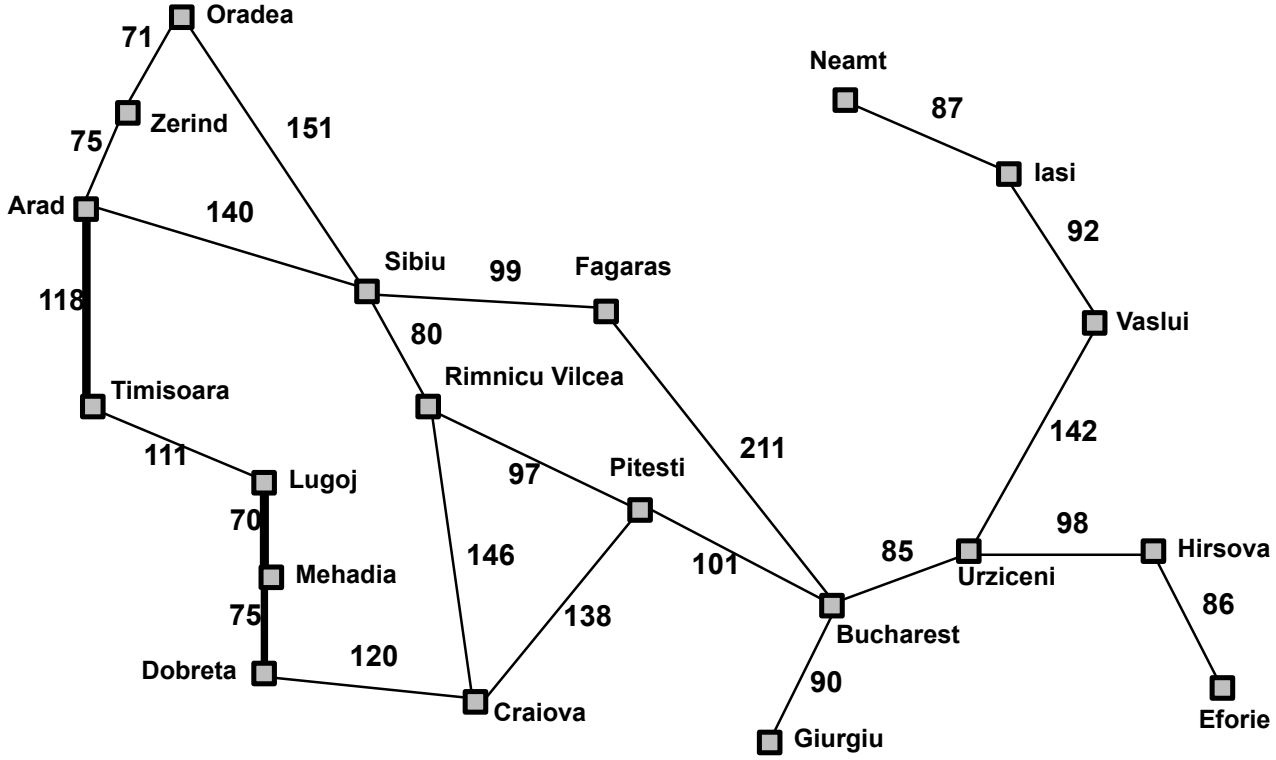
Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

Disadvantage

- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

Romania with step costs in km



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

Dijkstra's algorithm or Uniform cost search

When actions have different costs, a choice is to use best-first search where the evaluation function is the cost of the path from the root to the current node.

This is called Dijkstra's algorithm by the theoretical **computer science community**, and uniform-cost search by the **AI community.**

Breadth-first search spreads out in waves of uniform depth—first depth 1, then depth 2, and so on—uniform-cost search spreads out in waves of uniform path-cost

Uniform-cost search

Breadth-first finds the shallowest goal state

- but not necessarily be the least-cost solution
- work only if all step costs are equal

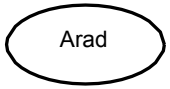
Uniform cost search

- modifies breadth-first strategy by always expanding the lowest-cost node
- The lowest-cost node is measured by the path cost *g(n)*

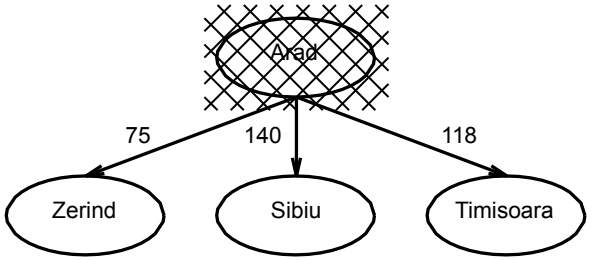
Expand least-cost unexpanded node

Implementation:

$\text{QUEUEINGFN} =$ insert in order of increasing path cost



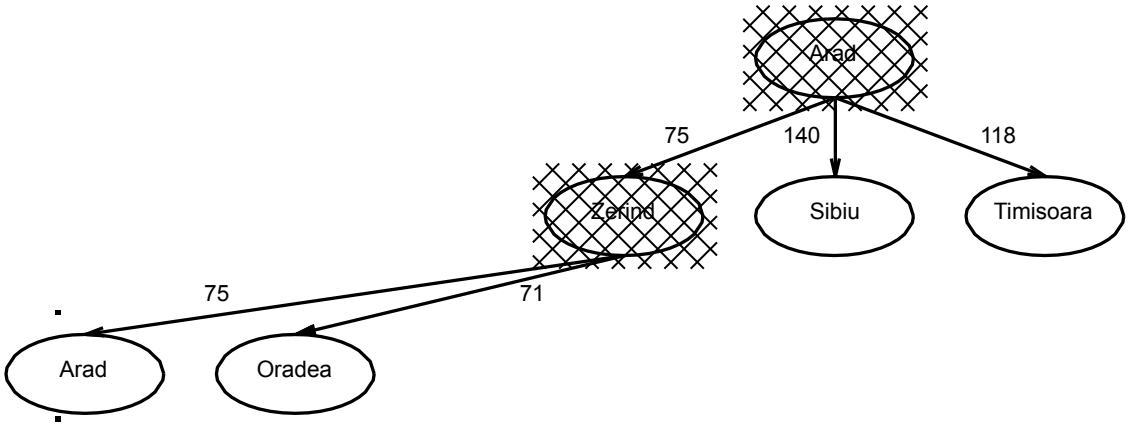
Example:

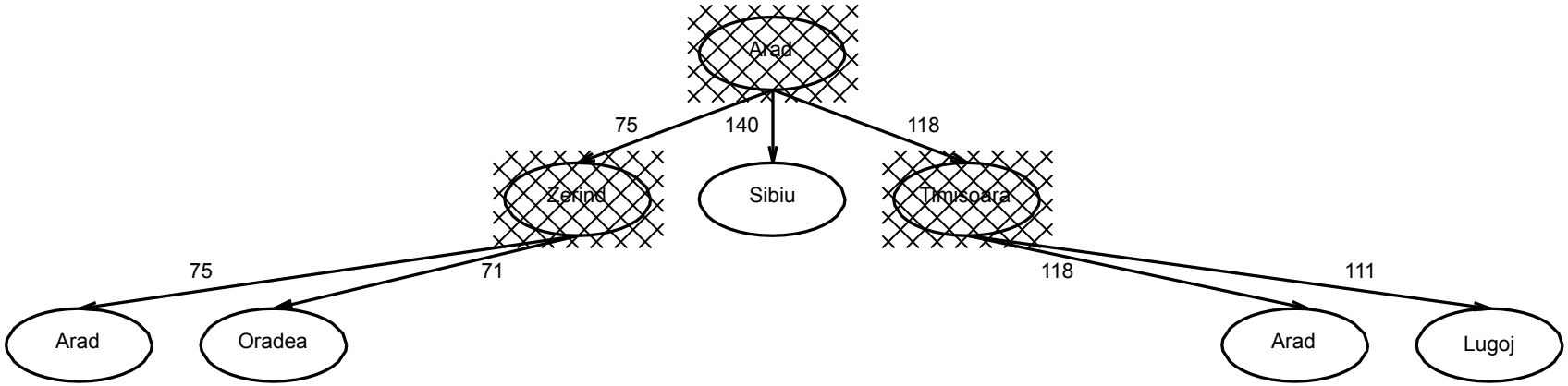


if we had checked for **a goal** ,

generating a node rather than expanding the lowest-cost node,

then we would have returned a higher-cost path





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?**

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

When all action costs are equal and uniform-cost search is similar to breadth-first search

Properties of uniform-cost search

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

Time?? # of nodes with $g \leq$ cost of optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution

Optimal?? Yes

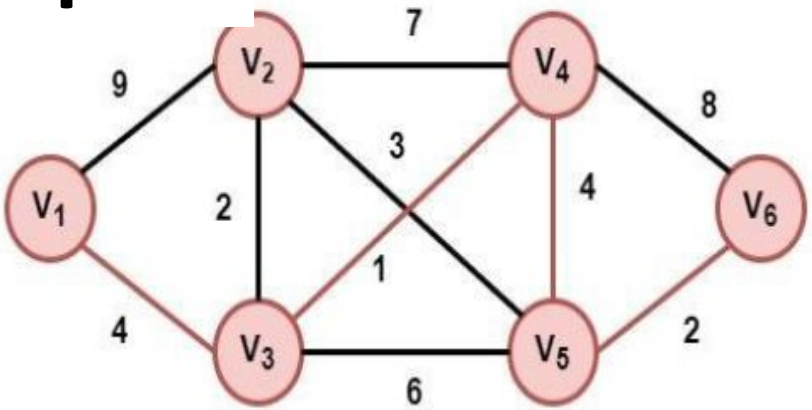
Advantages:

- Uniform cost search is optimal because at every state the path with the least cost is chosen.

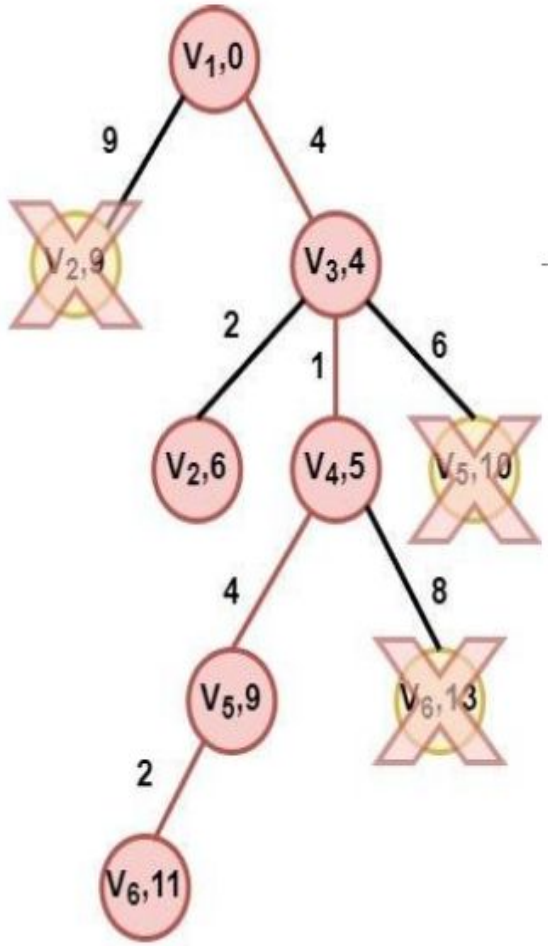
Disadvantages:

- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Example



Pace	Opened	Closed
0	$\{(V_1,0)\}$	{-}
1	$\{(V_2,9),(V_3,4)\}$	$\{(V_1,0)\}$
2	$\{(V_2,6),(V_4,5),(V_5,10)\}$	$\{(V_1,0),(V_3,4)\}$
3	$\{(V_2,6),(V_6,13),(V_5,9)\}$	$\{(V_1,0),(V_3,4),(V_4,5)\}$
4	$\{(V_6,13),(V_5,9)\}$	$\{(V_1,0),(V_3,4),(V_4,5),(V_2,6)\}$
5	$\{(V_6,11)\}$	$\{(V_1,0),(V_3,4),(V_4,5),(V_2,6),(V_5,9)\}$
6	{-}	$\{(V_1,0),(V_3,4),(V_4,5),(V_2,6),(V_5,9)\}$



Path: V_1 - V_3 - V_4 - V_5 - V_6

Total Cost: 11

Depth-first search

Expand deepest unexpanded node

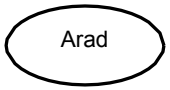
- Depth-first search is a recursive algorithm for traversing a tree or graph data structure
- Because it starts from the root node and follows each path to its greatest depth node before moving to the next path

- DFS uses a STACK

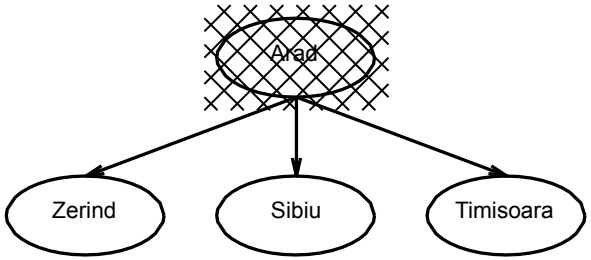
memory complexity of only **O(bm)**,
Where,

b □ branching factor
m □maximum depth of the tree

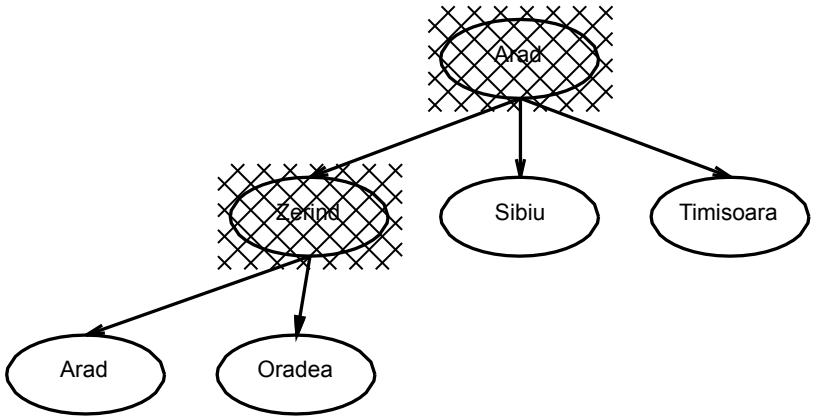
Example:



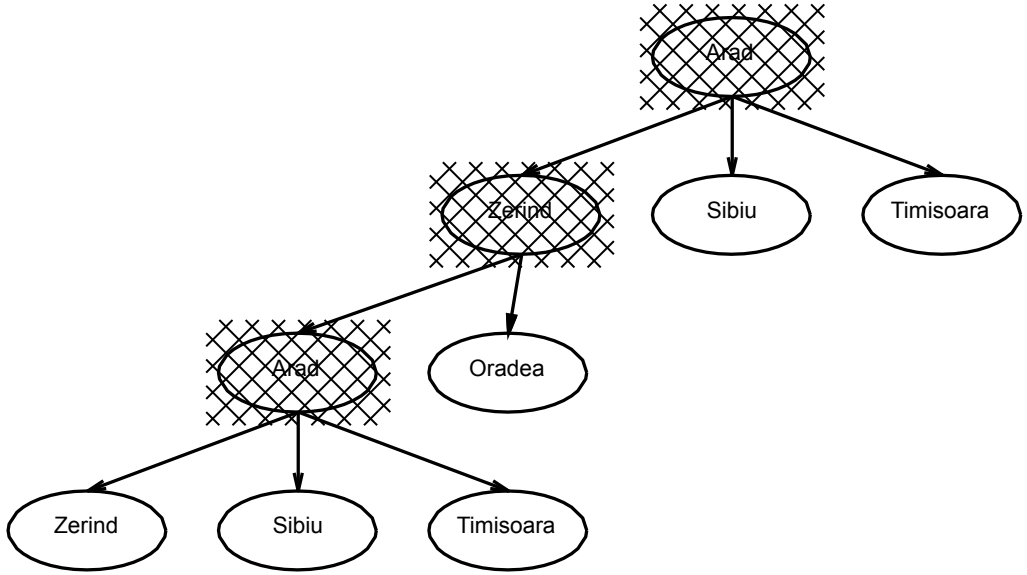
Step 1



Step 2



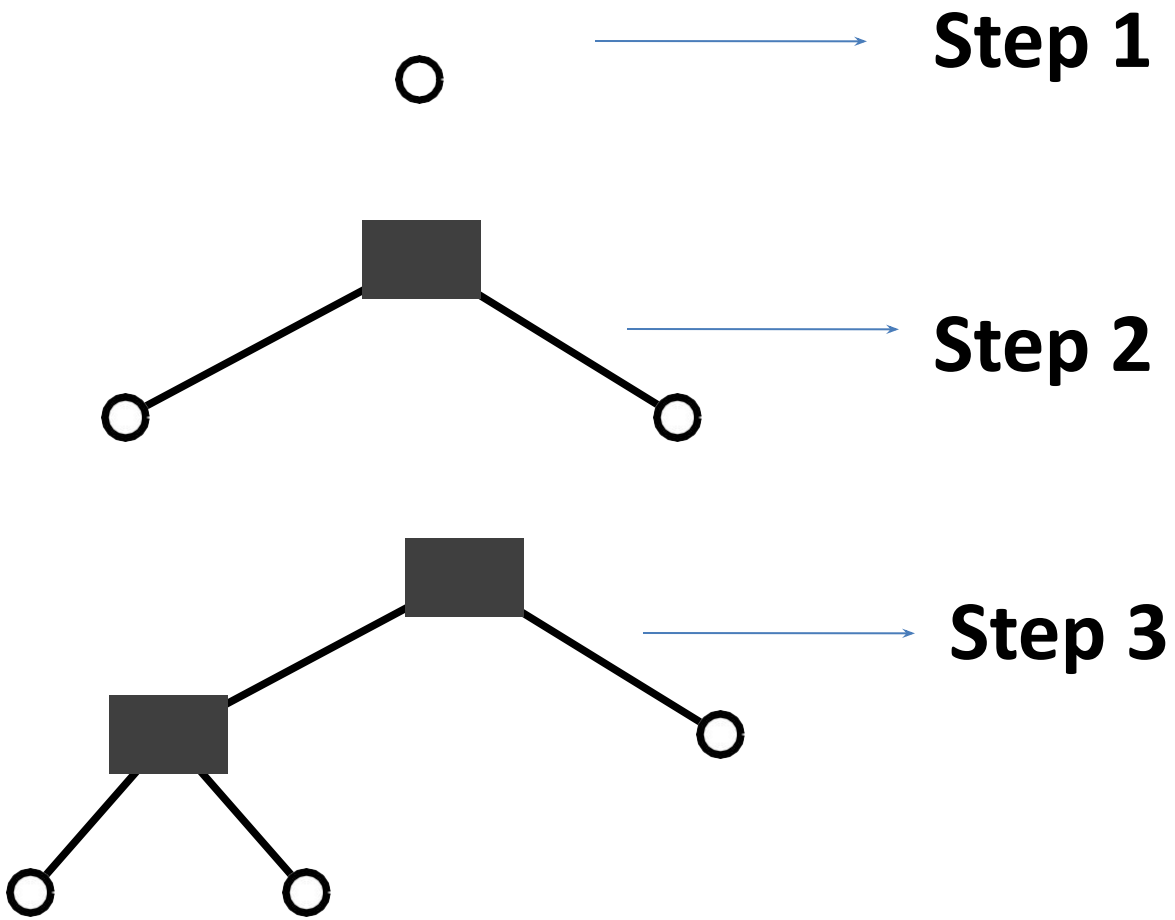
Step 3

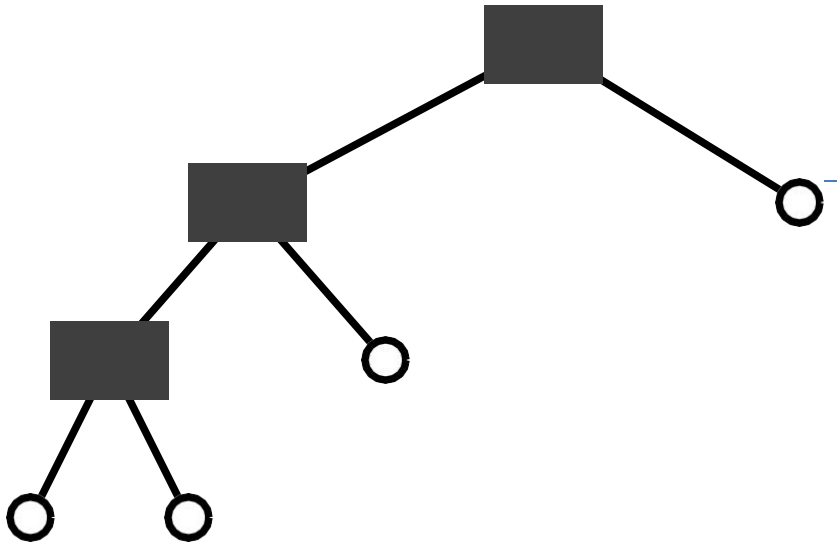


Step 4

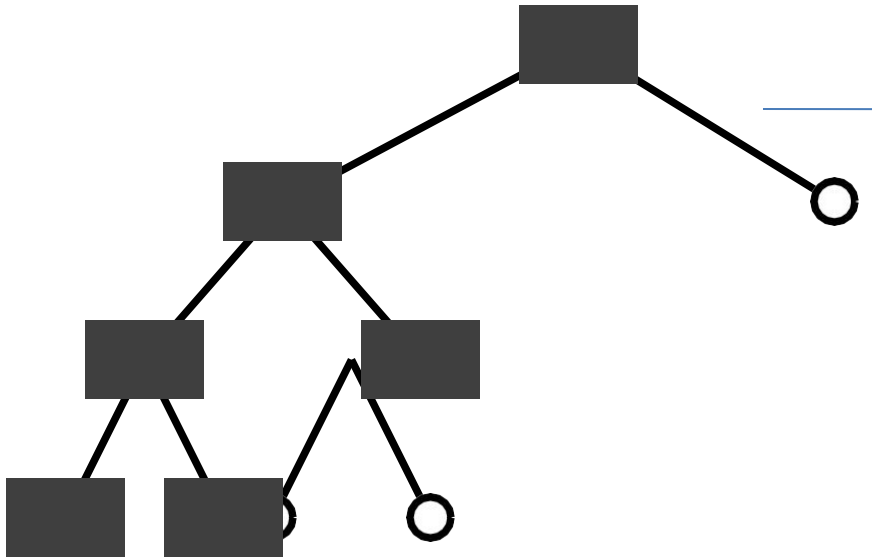
I.e., depth-first search can perform infinite cyclic excursions
Need a finite, non-cyclic search space (or repeated-state checking)

DFS on a depth-3 binary tree

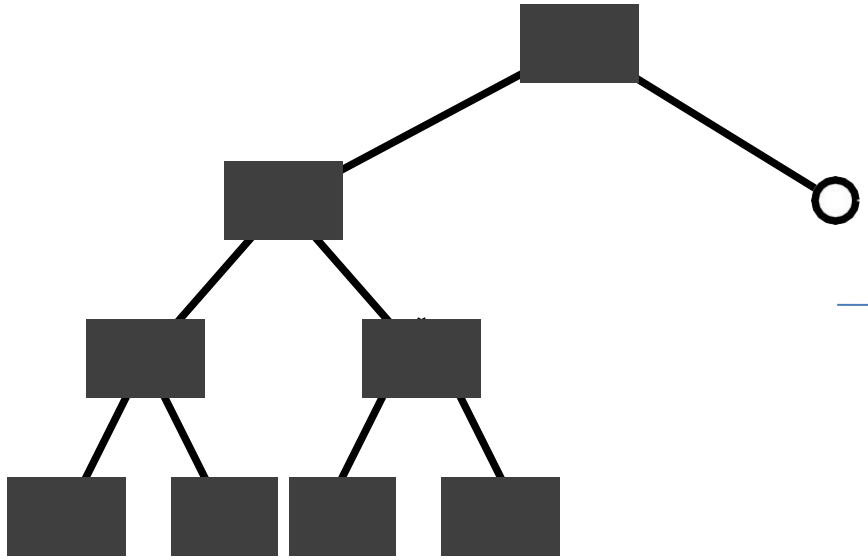




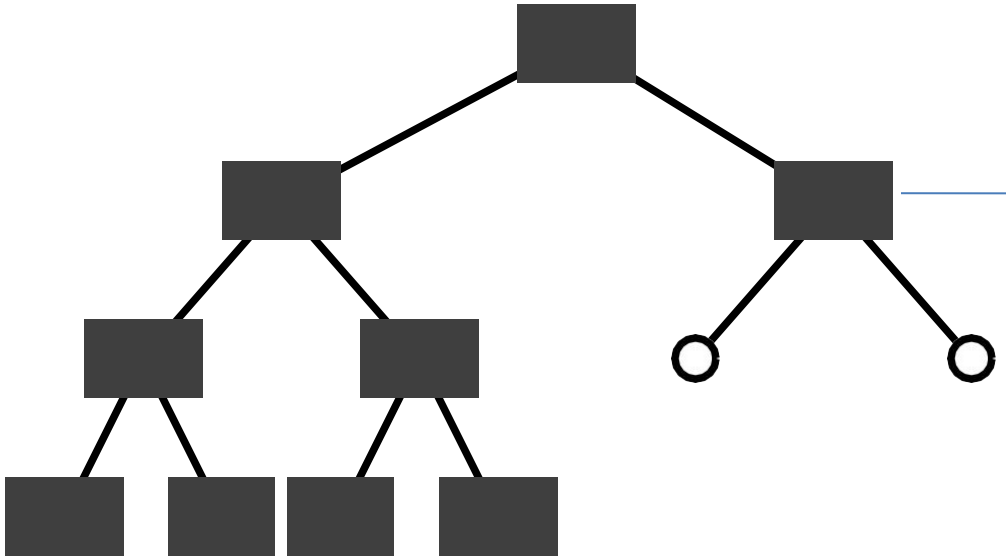
Step 4



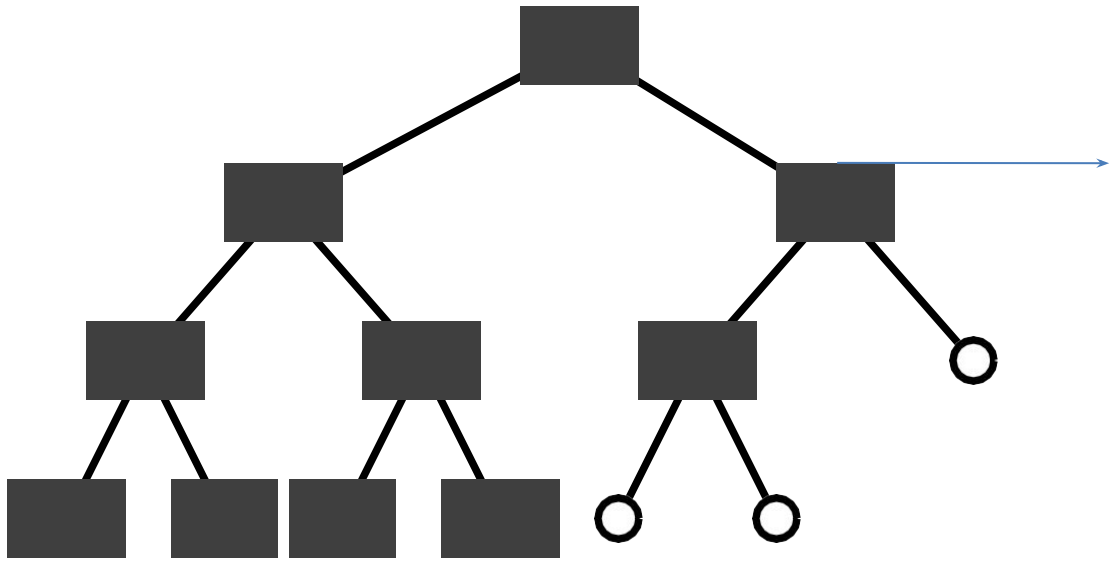
Step 5



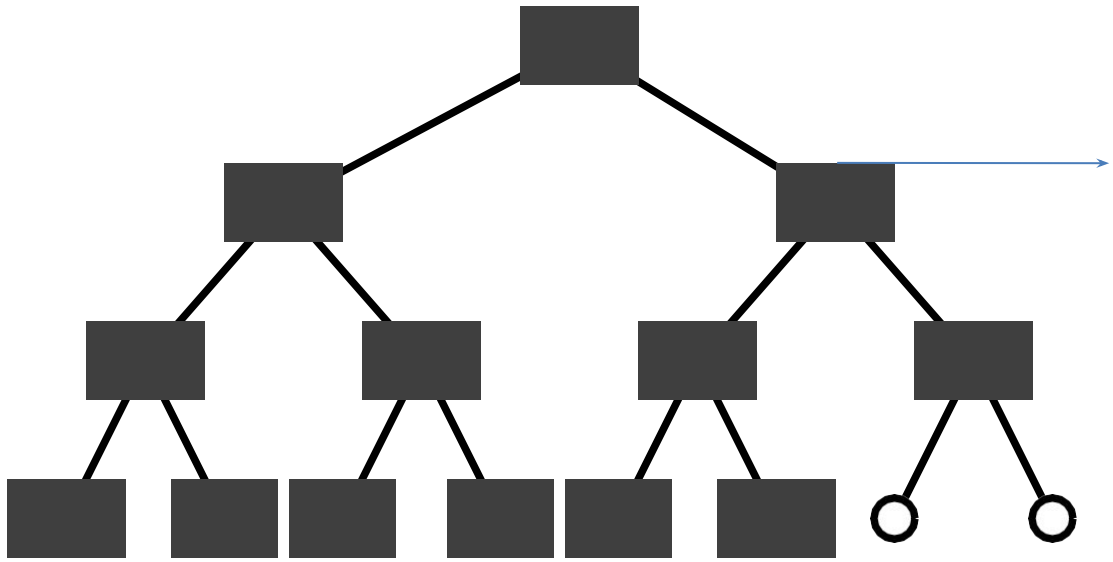
Step 6



Step 7



Step 8



Step 7

Properties of depth-first search

Completeness: Is complete within finite state space as it will expand every node within a limited search tree.

Optimal: Is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by: $O(b^m)$

Where, m= maximum depth of any node and this can be much larger than d

Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the O(bm).

Advantage:

DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.

It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.

DFS algorithm goes for deep down searching and sometime it may go to the infinite loop

Comparison	BFS	DFS
Concept	Vertex-based algorithm	Edge Based Algorithm
Data structure used to store the nodes	Queue	Stack
Memory consumption	Inefficient	Efficient
Built tree structure	Wide and short	Narrow and long
Going through mode	The oldest unvisited vertices are explored at the beginning.	The vertices along the edge are explored at the beginning.
Optimality	Optimal for finding the shortest distance	Not optimal
Request	Examine the bipartite chart, the connected component and the shortest path present in a chart.	Examine the connected graph of two edges, the strongly connected graph, the acyclic graph and the topological order.

BASIS FOR COMPARISON	BFS	DFS
Basic	Vertex-based algorithm	Edge-based algorithm
Data structure used to store the nodes	Queue	Stack
Memory consumption	Inefficient	Efficient
Structure of the constructed tree	Wide and short	Narrow and long
Traversing fashion	Oldest unvisited vertices are explored at first.	Vertices along the edge are explored in the beginning.
Optimality	Optimal for finding the shortest distance, not in cost.	Not optimal
Application	Examines bipartite graph, connected component and shortest path present in a graph.	Examines two-edge connected graph, strongly connected graph, acyclic graph and topological order.

Depth-limited search

depth-first search with depth limit l

Implementation:

Nodes at depth l have no successors

Depth limit is ℓ, assign all nodes at depth

a poor choice for ℓ the algorithm will fail to reach the solution(incomplete)

Diameter of the state-space graph, gives us a better depth limit, which leads to a more efficient depth-limited search.

The time complexity is $O(b^{\ell})$

The space complexity is $O(b\ell)$

EX: the map of Romania there are 20 cities. Therefore, ℓ=19 is a valid limit

Iterative deepening search

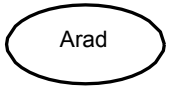
Solves the problem of picking a good value for ℓ by trying all values:

first 0, then 1, then 2, and so on—until either a solution is found, or the depth limited search returns the *failure* value rather than the *cutoff* value.

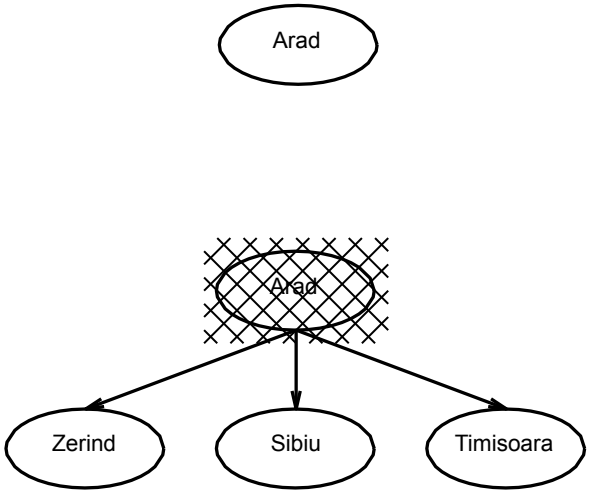
Iterative deepening combines many of the benefits of depth-first and breadth-first search.

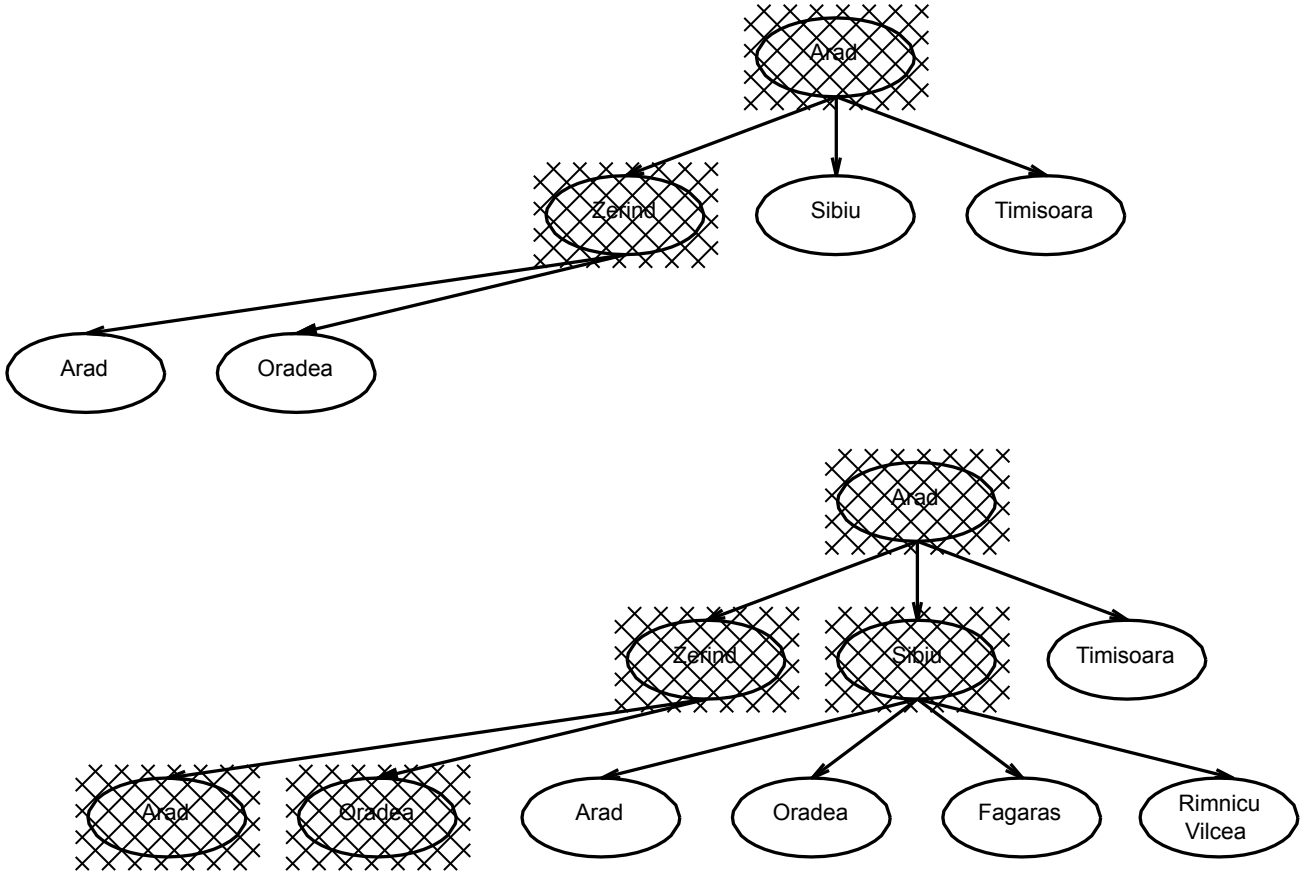
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution sequence
    inputs: problem, a problem

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

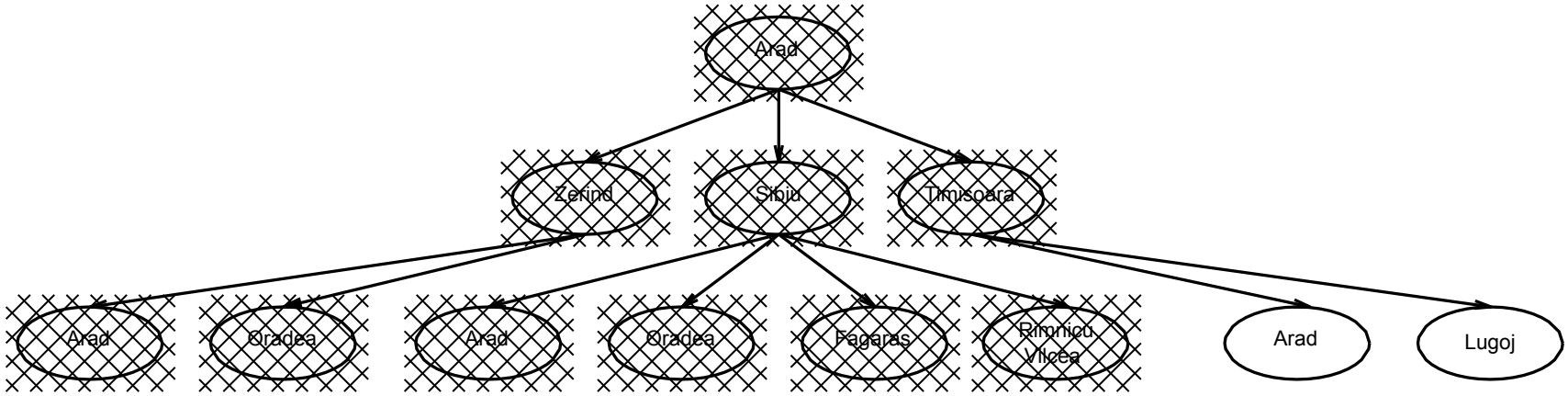
Iterative deepening search $l = 0$



Iterative deepening search $l = 1$




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



- It generates a new level, in the same way that breadth first search does.
- But breadth-first does this by storing all nodes in memory, while iterative deepening does it by repeating the previous levels.

The total number of nodes generated in the worst case is

$$N(\text{IDS}) = (d)b^1 + (d-1)b^2 + (d-2)b^3 \cdots + b^d$$

For example, if $b = 10$ and $d = 5$, the numbers are

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110.$$

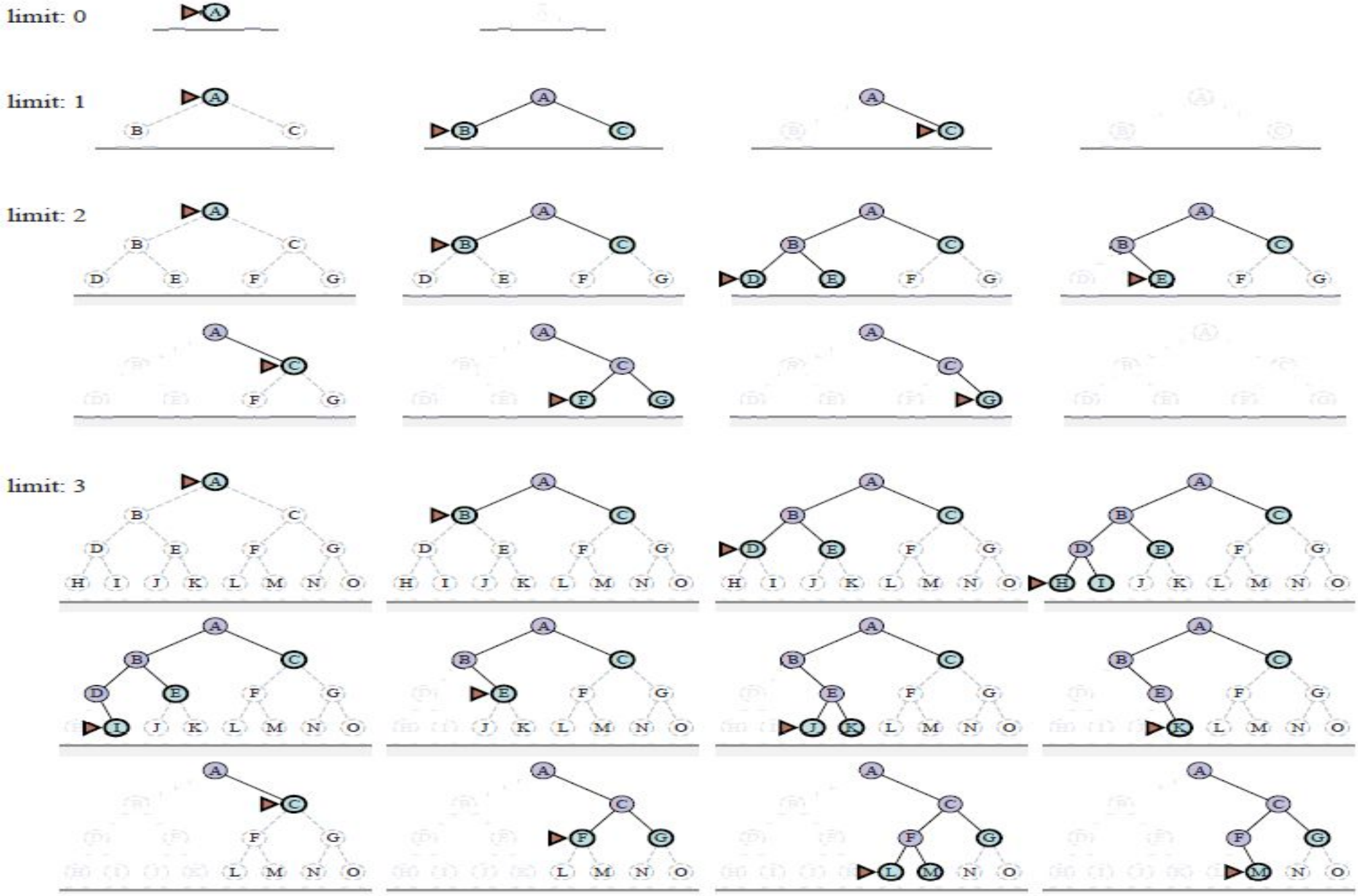


Figure 3.13 Four iterations of iterative deepening search for goal M on a binary tree, with the depth limit varying from 0 to 3. Note the interior nodes form a single path. The triangle marks the node to expand next; green nodes with dark outlines are on the frontier; the very faint nodes provably can't be part of a solution with this depth limit.

Properties of iterative deepening search

Complete?? Yes

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

Space?? $O(bd)$

Optimal?? Yes, if step cost $= 1$

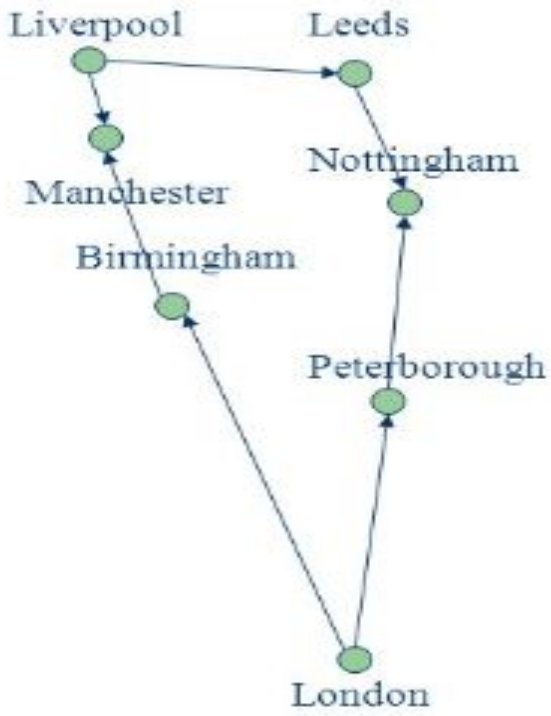
Can be modified to explore uniform-cost tree

Feature	Depth-Limited Search (DLS)	(IDDFS)
Definition	A depth-first search (DFS) with a predefined depth limit to prevent infinite loops.	A repeated depth-limited search with increasing depth limits until the solution is found.
Purpose	Used when the depth of the solution is known or to prevent excessive recursion depth.	Used when the depth of the solution is unknown but completeness and optimality are required.
Depth Restriction	Fixed depth limit (`L`). If the solution is beyond this limit, it won't be found.	Increases depth limit incrementally (`0, 1, 2, …`) until the solution is located.
Completeness	Incomplete if the depth limit is too small.	Complete if the branching factor is finite (will always find a solution if one exists).
Optimality	Not optimal, as it may not find the shallowest solution.	Optimal in uniform-cost cases (where all step costs are equal).
Memory Usage	O(d), where `d` is the depth limit.	O(d), same as DLS, as it only keeps track of the current path.
Time Complexity	O(b^L), where `b` is the branching factor and `L` is the limit.	O(b^d), where `d` is the depth of the solution. Re-explores nodes but remains efficient.
Redundancy	Searches only once up to the depth limit.	Searches nodes multiple times at increasing depths.
Best Use Case	When the maximum search depth is known in advance.	When the depth of the solution is unknown and completeness is required.
Example Scenario	Searching for a goal node within a known depth range in a decision tree.	Solving puzzles like the 8-puzzle or chess, where the depth of the solution is unknown.

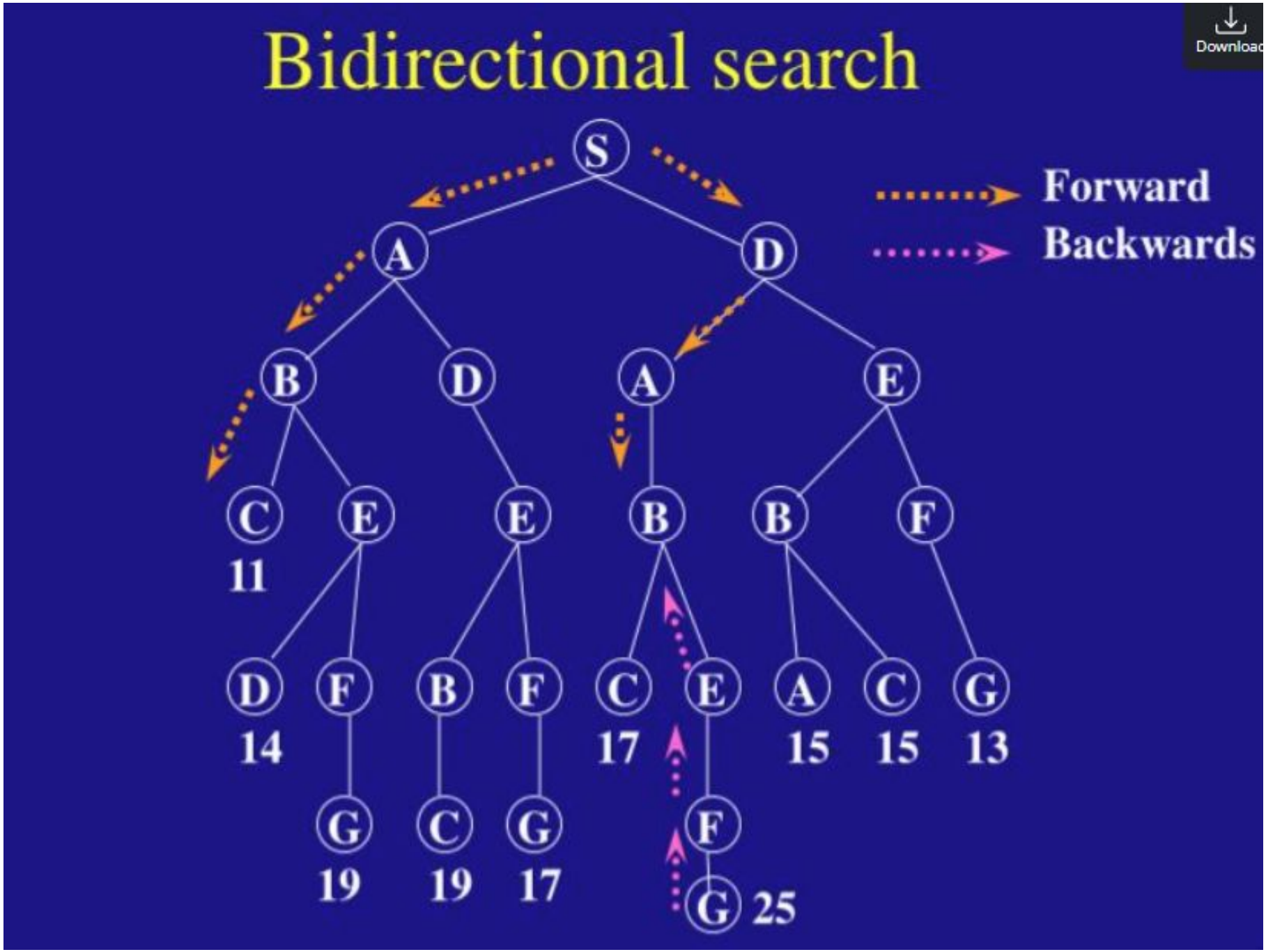
Bidirectional Search

- If you know the solution state
  - Work forwards and backwards
  - Look to meet in middle
- Only need to go to half depth

- Difficulties
  - Do you really know solution? Unique?
  - Must be able to reverse operators
  - Record all paths to check they meet
    - Memory intensive



It need to keep track **of two frontiers** and two tables of reached states, and we need to be **able to reason backwards**: if state s′ is a successor of s in the **forward direction**, then we need to know that s is a successor of s′ in the backward direction


Bidirectional search
Download
S
A
D
B
D
A
E
C
E
E
B
B
F
11
D
F
B
F
C
E
A
C
G
14
17
15
15
13
G
C
G
F
19
19
17
G
25
Forward
Backwards

Bi-directional search

- Much more efficient.
  - If branching factor = b in each direction, with solution at depth d
  - Rather than doing one search of b^d, we do *two* $b^{d/2}$ searches.
  - $b^{d/2} + b^{d/2}$ is much less than b^d

Example:

- Suppose b = 10, d = 6. Suppose each direction runs BFS
- In the worst case, two searches meet when each search has generated all of the nodes at depth 3.
- Breadth first search will examine 11, 11, 111 nodes.
- Bidirectional search will examine 2,220 nodes.

Comparing uninformed search algorithms

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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

S No.	Parameters	Depth First Search(DFS)	Depth Limited Search(DLS)	Breadth First Search(BFS)
1.	*Tree Traversal*	Depth-Wise	Depth-Wise	Level-Wise
2.	*Implementation*	Stack LIFO(Last-in First-out)	Stack LIFO(Last-in First-out)	Queue FIFO(First-in First-out)
3.	*Memory Required*	Lower	Lower	Higher
4.	*Backtracking*	Allowed	Allowed	Not Allowed
5.	*Infinite Loops*	Yes	No	No
6.	*Time Complexity*	$O(b^m)$ m=maximum depth of search tree	$O(b^l)$ l=specified depth limit	$O(b^d)$ d=depth of shallowest tree
7.	*Space Complexity*	$O(b^m)$	O(bxl)	$O(b^d)$
8.	*Optimal*	No	No	Yes
9.	*Completeness*	No	Yes, when d<l (depth-limit)	Yes
10.	*Applications*	1.Finding connected nodes or components 2.Used for detecting cycles	1.Solving Puzzles with only one solution 2.Toplogical sorting	1.Finding the shortest path between two nodes 2.Examining the bipartite graph

Summary

Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored

Variety of uninformed search strategies

Iterative deepening search uses only linear space
and not much more time than other uninformed algorithms

INFORMED (HEURISTIC) SEARCH STRATEGIES

- Informed search strategy : It uses domain-specific hints about the location of goals

- It can find solutions more efficiently than an uninformed strategy.

- The hints in the form of a **heuristic function**. Its denoted as **h(n)**

  "It is evaluation function helps to select the optimal move in the every traversals"

h(n) = estimated cost of the cheapest path from the state at node n to a goal state

Example : Straight line distance on the map between the two points.

Greedy best-first search

It is a form of best-first search that expands first the node with the Greedy best-first search

lowest h(n) value—the node that appears to be closest to the goal

It leads to a solution quickly.

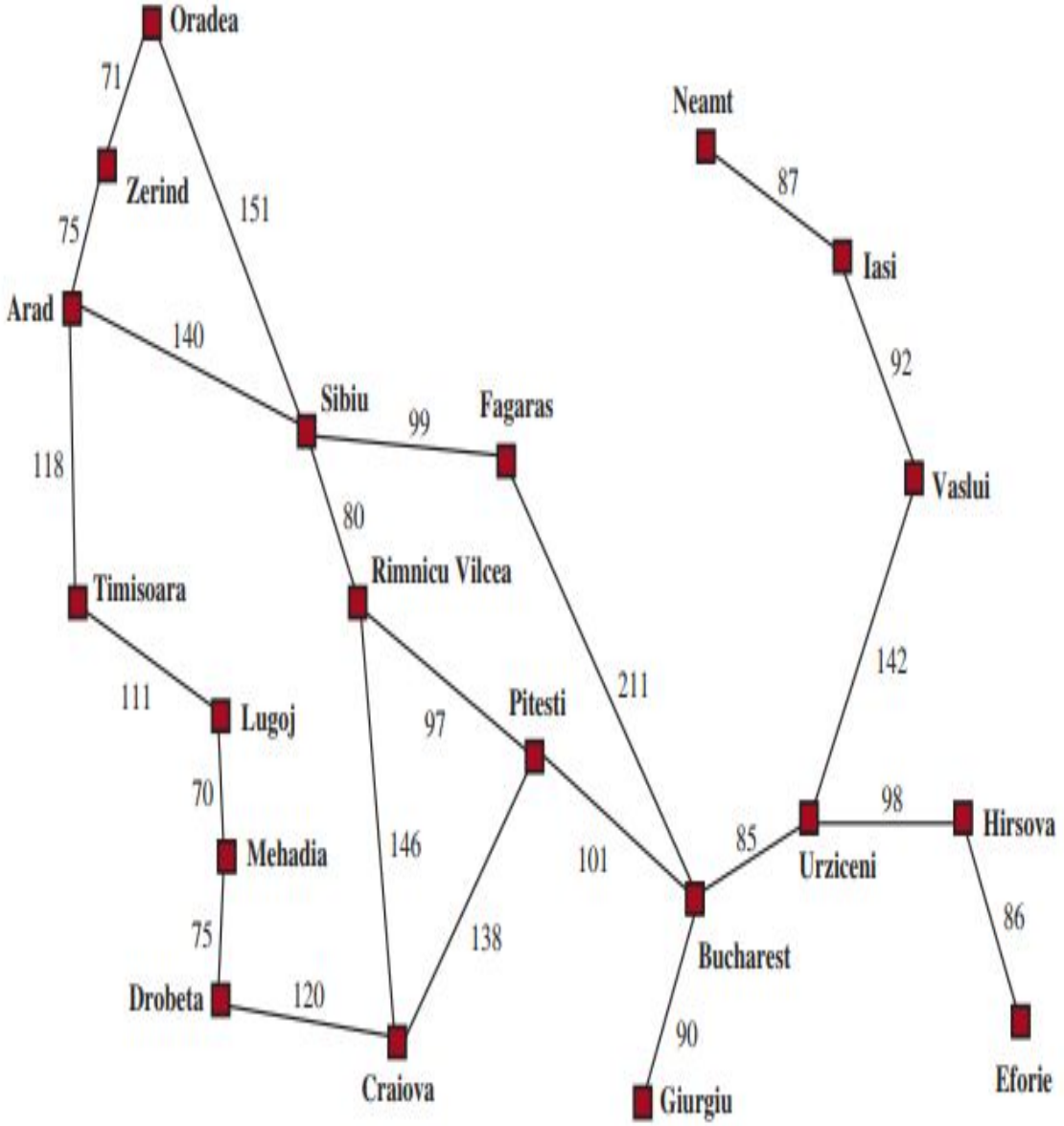
the evaluation function f (n) = h(n) straight-line distance heuristic

For example:

$\mathbf{h_{SLD}(Arad)=366}$. Using h_{SLD} to find a path from Arad to Bucharest

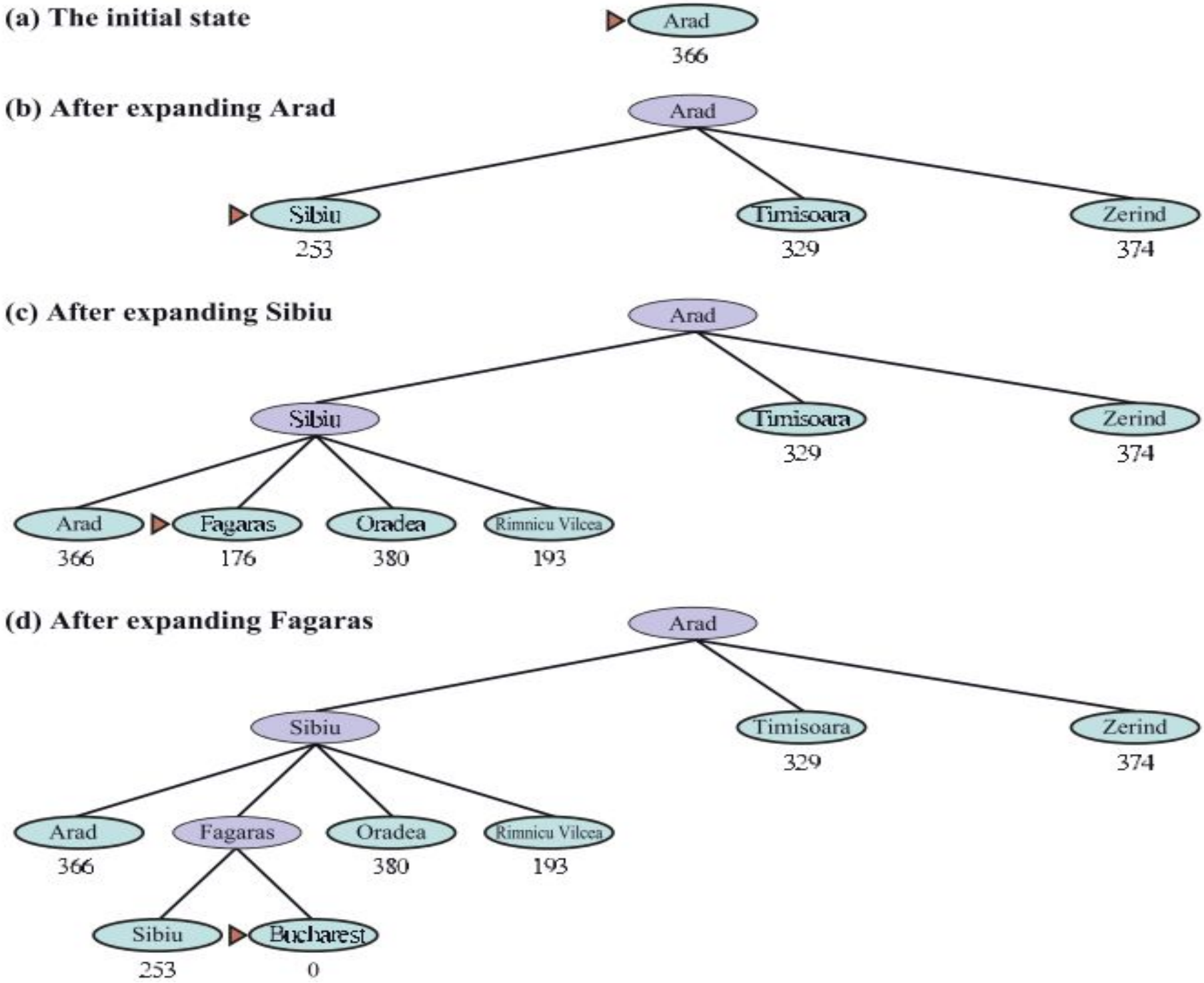
This is why the algorithm is called "greedy"—on each iteration it tries to get as close to a goal

- The worst-case time and space complexity is O(|V |).
- With a good heuristic function, however, the complexity can be reduced substantially, on certain problems reaching O(bm).



Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	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

Values of h_{SLD}—straight-line distances to Bucharest.


(a) The initial state
Arad
366
(b) After expanding Arad
Arad
Sibiu
253
Timisoara
329
Zerind
374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
176
Oradea
380
Rimnicu Vilcea
193
(d) After expanding Fagaras
Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
Oradea
380
Rimnicu Vilcea
193
Sibiu
253
Bucharest
0

A* search algorithm

- The most common informed search algorithm is A* search

- It uses the evaluation function

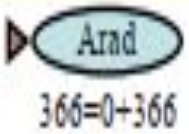
$$f(n) = g(n) + h(n)$$

g(n) --> path cost from the initial state to node n

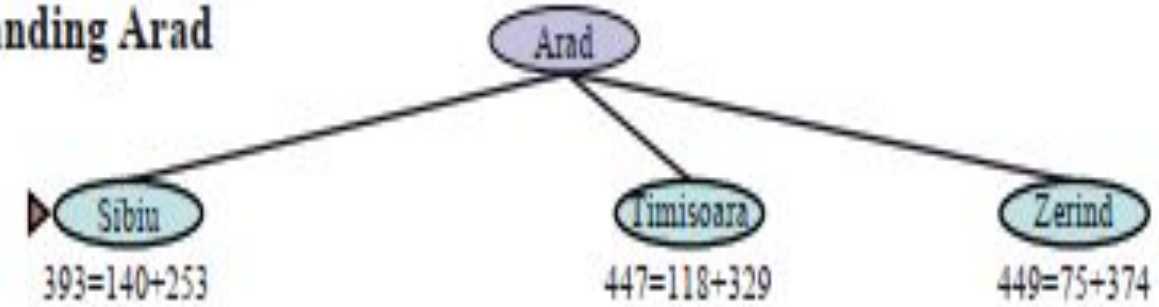
h(n) □ estimated cost of the shortest path from n to a goal state

f (n) □ estimated cost of the best path that continues from n to a goal.

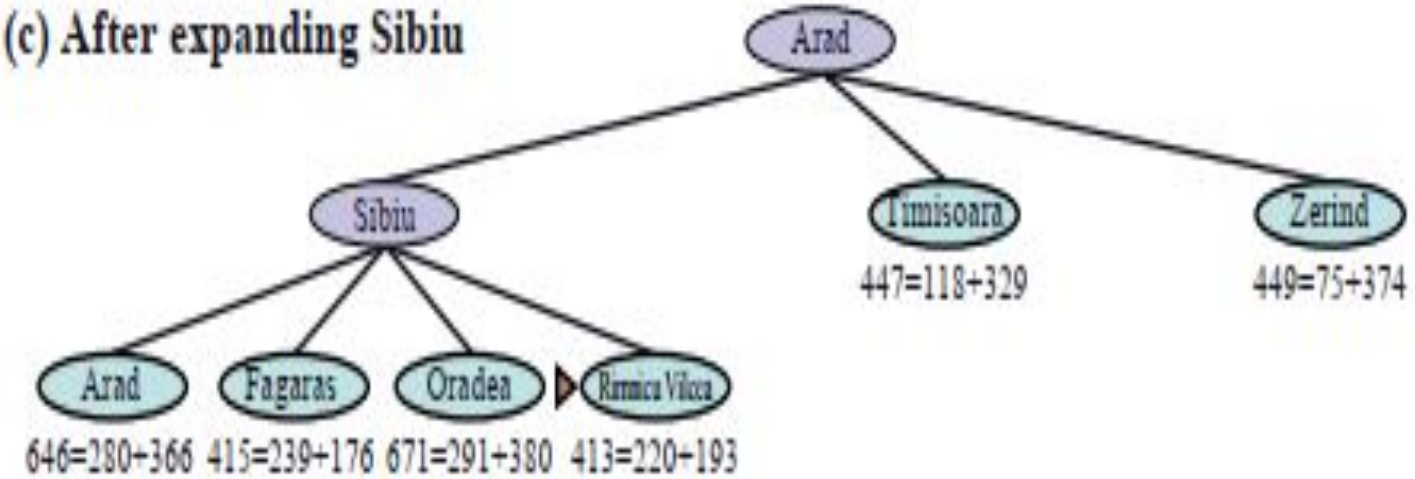
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu



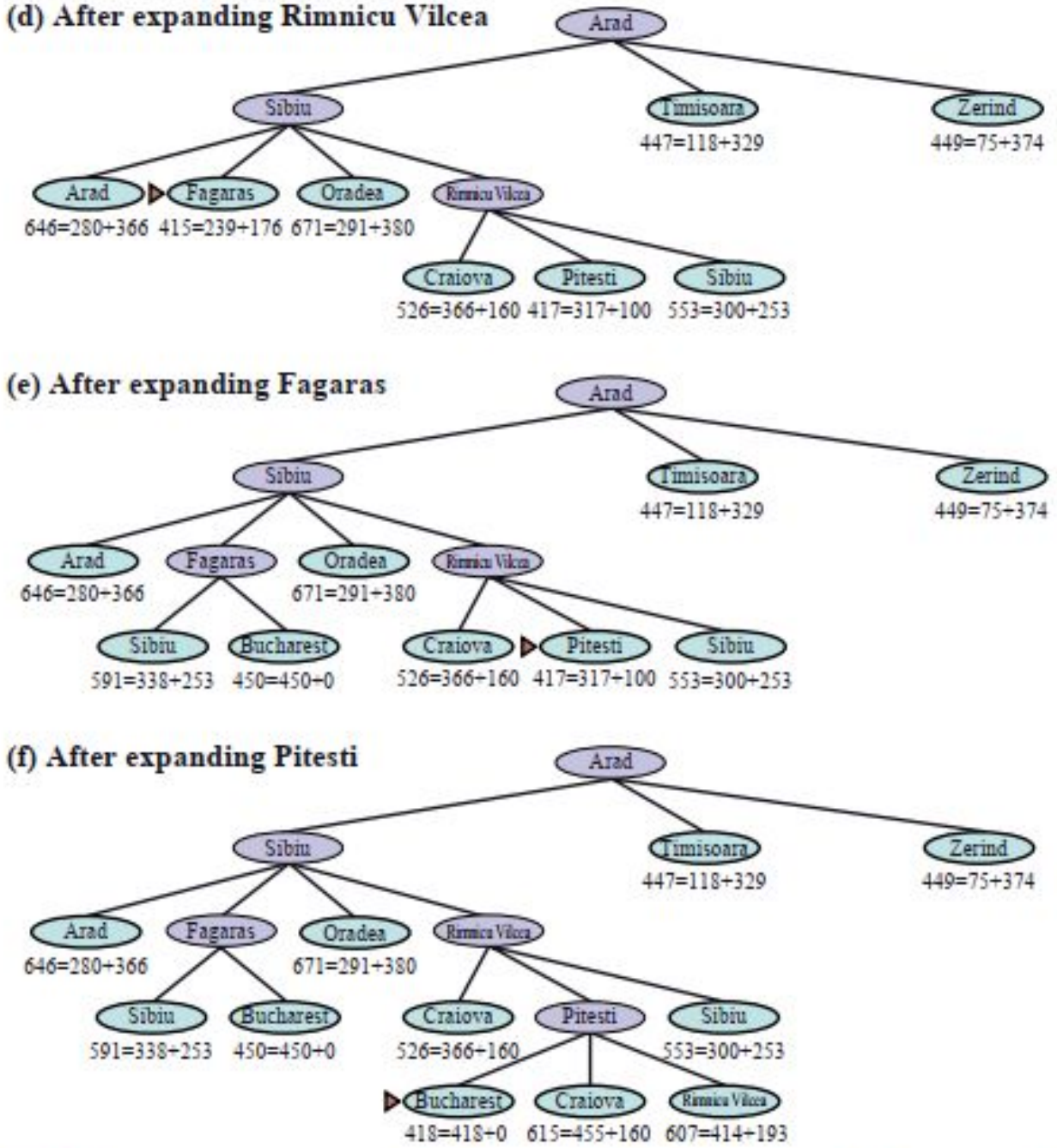


Figure 3.18 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.16.

Properties of A* search

Complete: A* search is complete.

Whether A* is cost-optimal depends on certain properties of the heuristic.

Optimal: A* search algorithm is optimal if it follows below two conditions:

1. **Admissible:** an admissible heuristic is one that never overestimates the cost to reach a goal

2. Consistency :

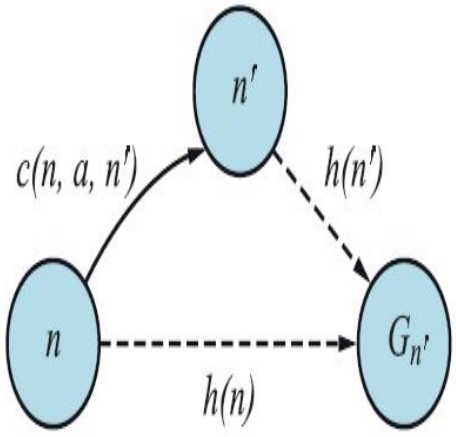
h(Consistency *n*) is consistent if, for every node *n* and every successor *n'* of *n* generated by an action *a*,

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

"a side of a triangle cannot be longer than the sum of the other two sides"

Time Complexity: O(b^d), where b is the branching factor.

Space Complexity: O(b^d)



A* search

Advantages

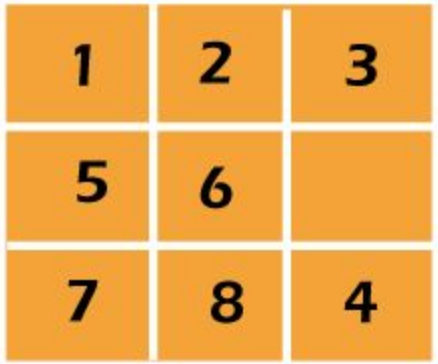
- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

Disadvantages

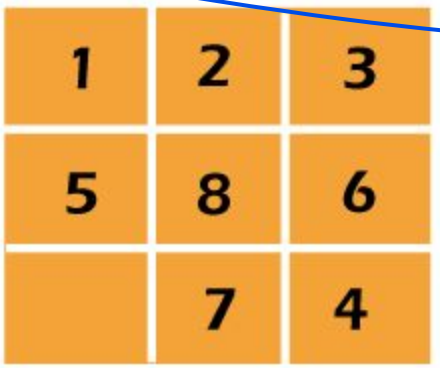
- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

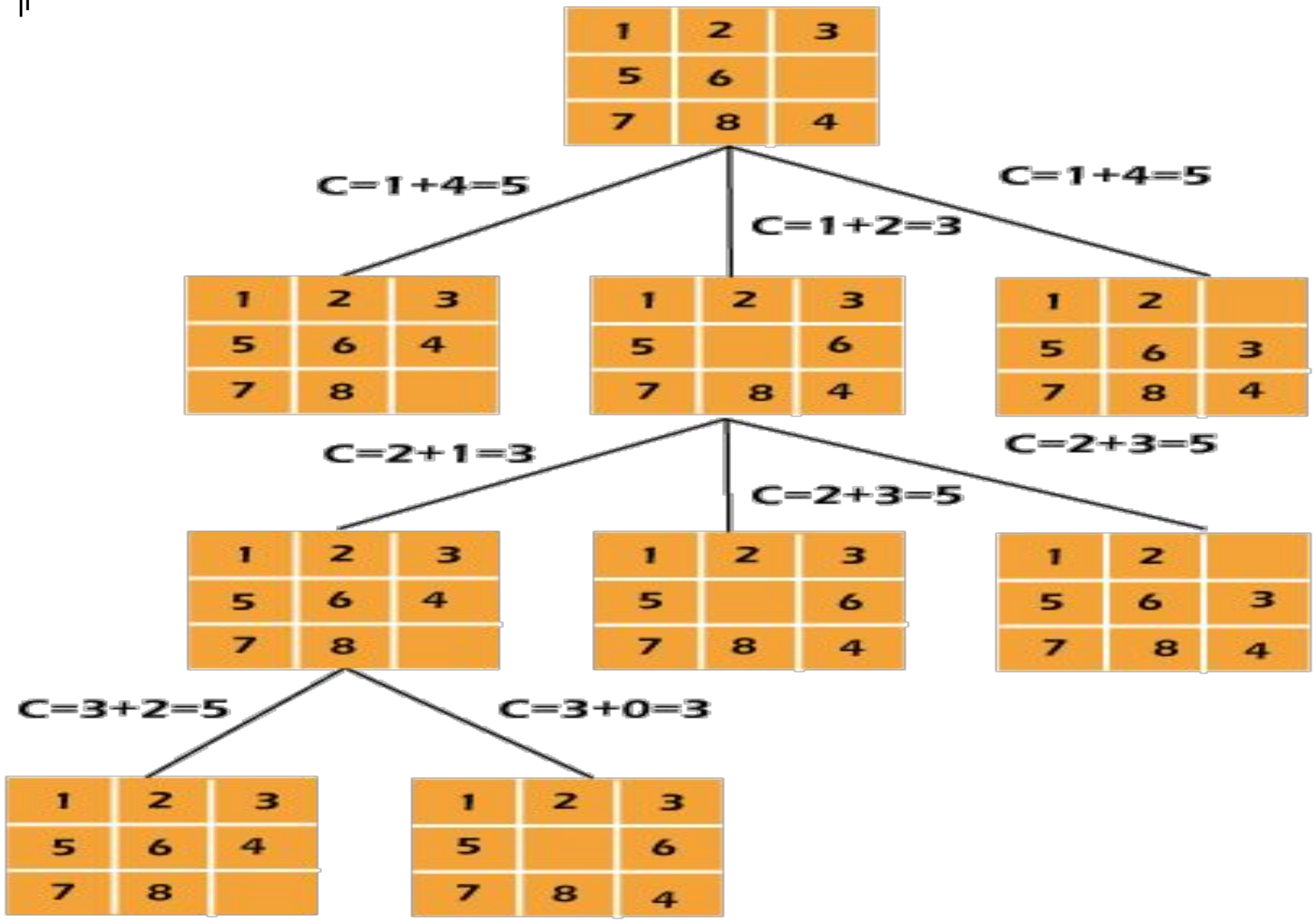
A* Algorithm for 8 Puzzle Problem

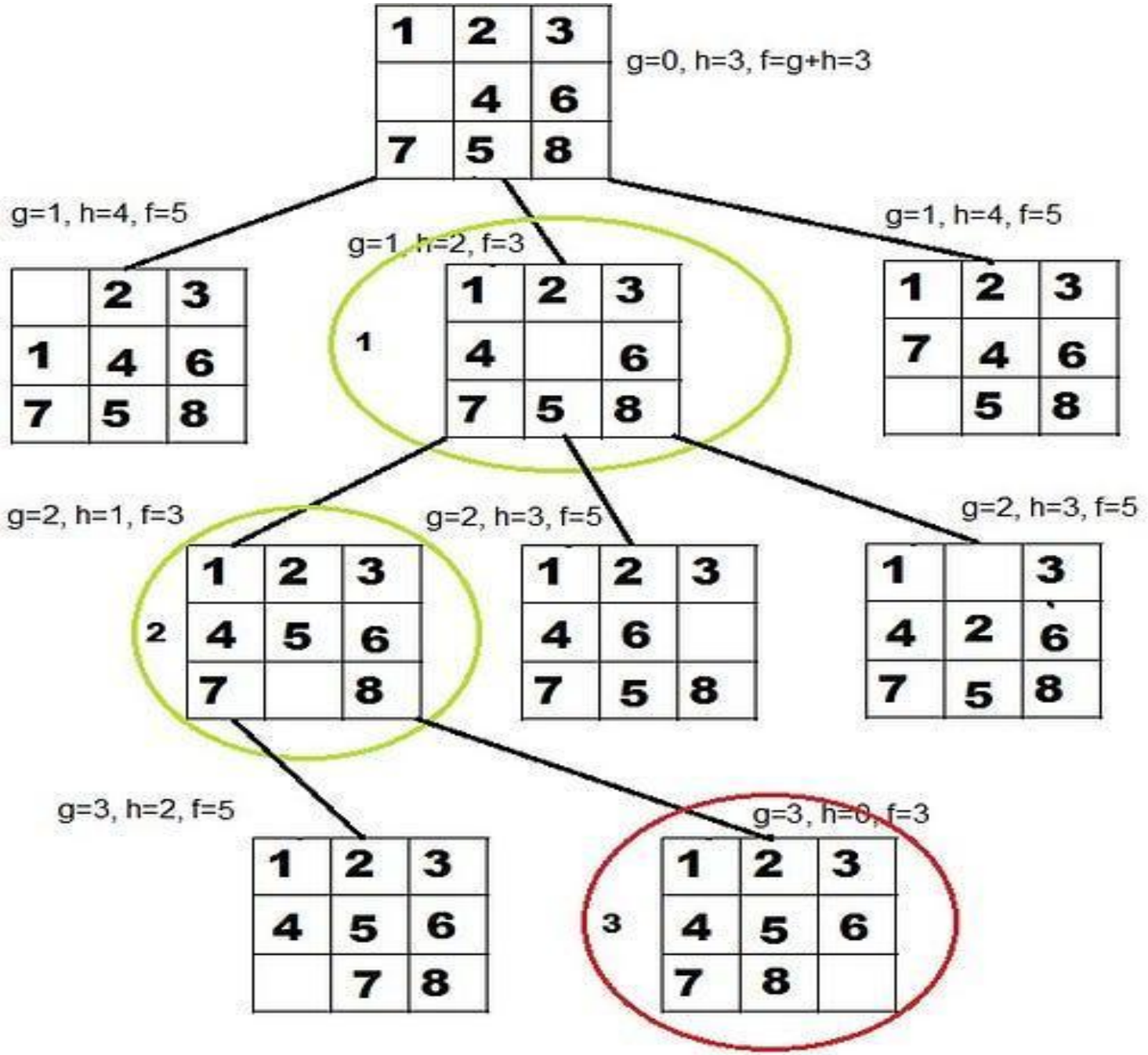
Initial Configuration

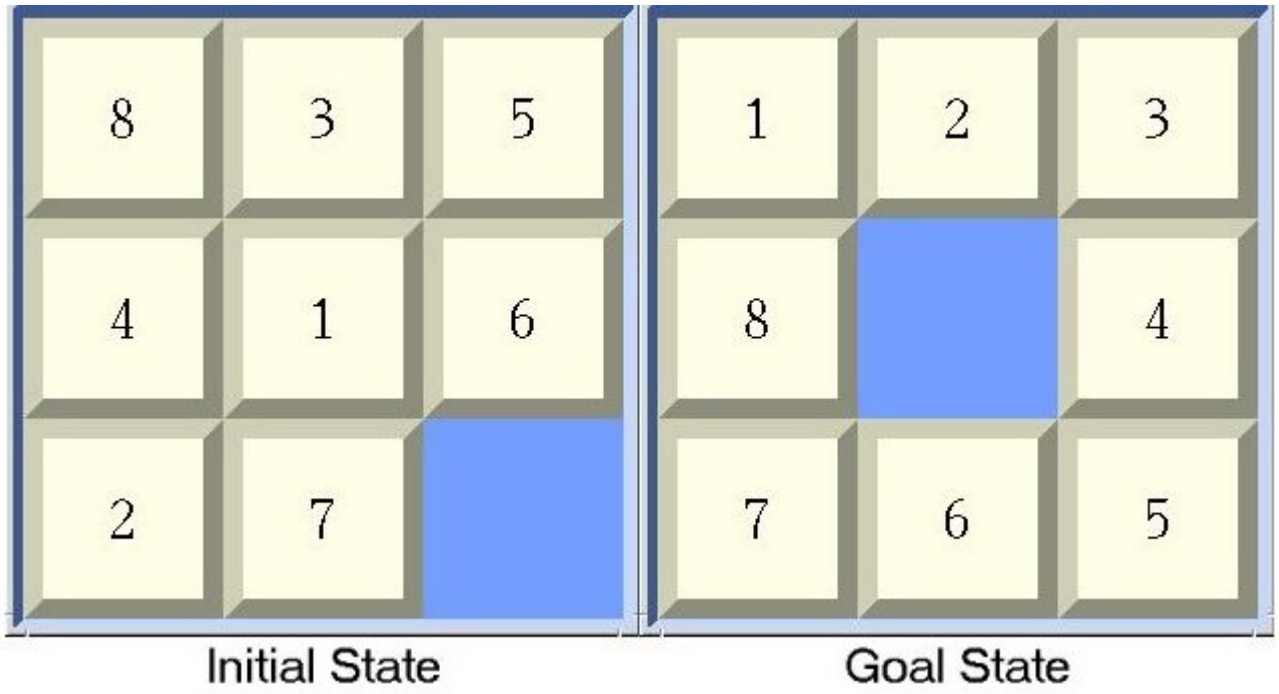


Final Configuration




1 2 3
5 6
7 8 4
C=1+4=5
C=1+2=3
C=1+4=5
1 2 3
5 6 4
7 8
1 2 3
5 6
7 8 4
1 2
5 6 3
7 8 4
C=2+1=3
C=2+3=5
C=2+3=5
1 2 3
5 6 4
7 8
1 2 3
5 6
7 8 4
1 2
5 6 3
7 8 4
C=3+2=5
C=3+0=3
1 2 3
5 6 4
7 8
1 2 3
5 6
7 8 4


1 2 3
4 6
7 5 8
g=0, h=3, f=g+h=3
g=1, h=4, f=5
2 3
1 4 6
7 5 8
g=1, h=2, f=3
1
1 2 3
4 6
7 5 8
g=1, h=4, f=5
1 2 3
7 4 6
5 8
g=2, h=1, f=3
2
1 2 3
4 5 6
7 8
g=2, h=3, f=5
1 2 3
4 6
7 5 8
g=2, h=3, f=5
1 3
4 2 6
7 5 8
g=3, h=2, f=5
1 2 3
4 5 6
7 8
g=3, h=0, f=3
3
1 2 3
4 5 6
7 8


8 3 5
4 1 6
2 7
1 2 3
8 4
7 6 5
Initial State
Goal State

Heuristic Search

- Heuristic Technique is a criterion for determining most effective to achieve some goal among several alternative way

- Increase the efficiency of a search process by sacrificing claims of systematic and completeness

- Two types :
    - General Purpose – **Ex :** nearest neighbor algorithm
    - Special purpose (domain specific)- Ex: control system
    - Example Algorithms : Branch and bound(Uniform cost search)
      Hill climbing, Best first, A*

Hill climbing Search

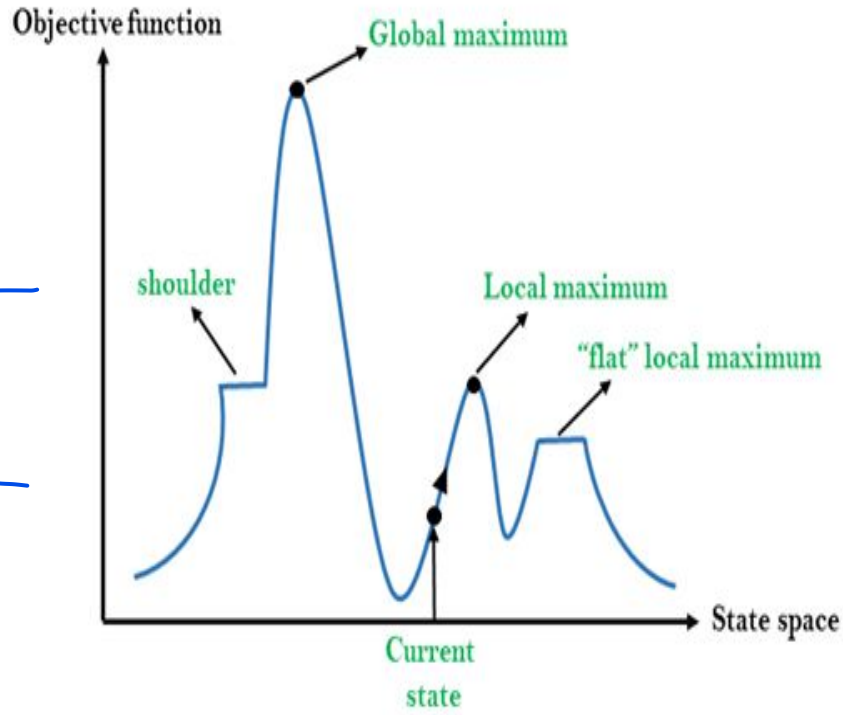
•Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem.

• It terminates when it reaches a peak value where no neighbor has a higher value.

•examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.

• Two components : state and value.

Regions in the Hill climbing



What is,

- **State and State space**: state represents a specific configuration or condition of the problem.
- **Example:** The 8-Puzzle - Each possible arrangement of the tiles is a state in the state space.

- **Objective function and Value:** mathematical representation that guides the learning and adaptation processes of AI algorithms by quantifying(**Value**) the difference between model predictions and actual outcomes.

Features :

- **Generate and Test variant:** The variant of generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

Deterministic Nature: Hill Climbing is a deterministic optimization algorithm, which means that given the same initial conditions and the same problem, it will always produce the same result. There is no randomness or uncertainty in its operation.

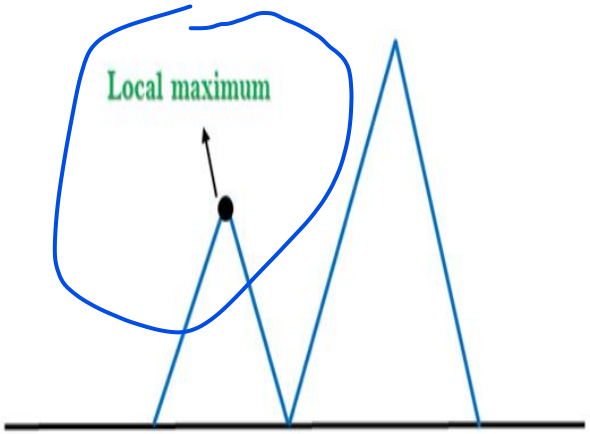
Local Neighborhood: Hill Climbing is a technique that operates within a small area around the current solution. It explores solutions that are closely related to the current state by making small, gradual changes.

This approach allows it to find a solution that is better than the current one although it may not be the global optimum.

Problems in Hill Climbing Algorithm

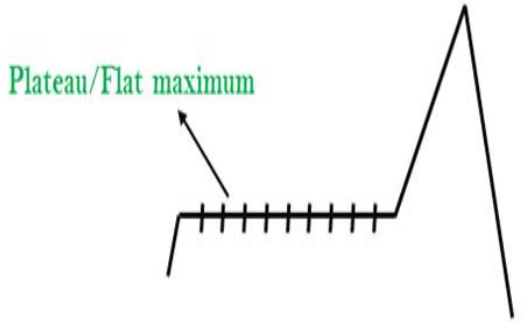
1. **Local Maximum**: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

   Sol : Backtracking technique



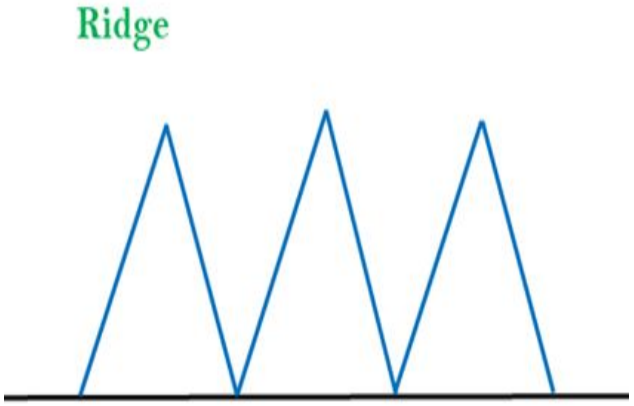
2. **Plateau:** A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move.

   Sol: take big steps or very little steps



3. **Ridges:** It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.

   Sol : use of bidirectional search or moving in different directions



Types of Hill Climbing Algorithm:

- **Simple hill Climbing: It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state**

- **Steepest-Ascent hill-climbing: It examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state.**

- **Stochastic hill Climbing: It uses search algorithm selects one neighbor node at random and decides whether to choose it as a current state or examine another state.**

Applications of Hill Climbing Algorithm:

- **Machine Learning-** Fine tuning of machine learning models by hyper parameter optimization
- **Robotics** : Its path is adjusted before arriving at the destination
- **Network Design** :the efficiency of the networks are increased and adjusting their configurations for reliability.
- **Game playing** : Helps to get the maximum scores.
- **Natural language processing**: summarizing text, translating languages and recognizing speech

Iterative Deepening A* algorithm (IDA*)

It is a kind of **iterative deepening depth-first search** that adopts the **A* search algorithm's** idea of using a heuristic function to assess the remaining cost to reach the goal.

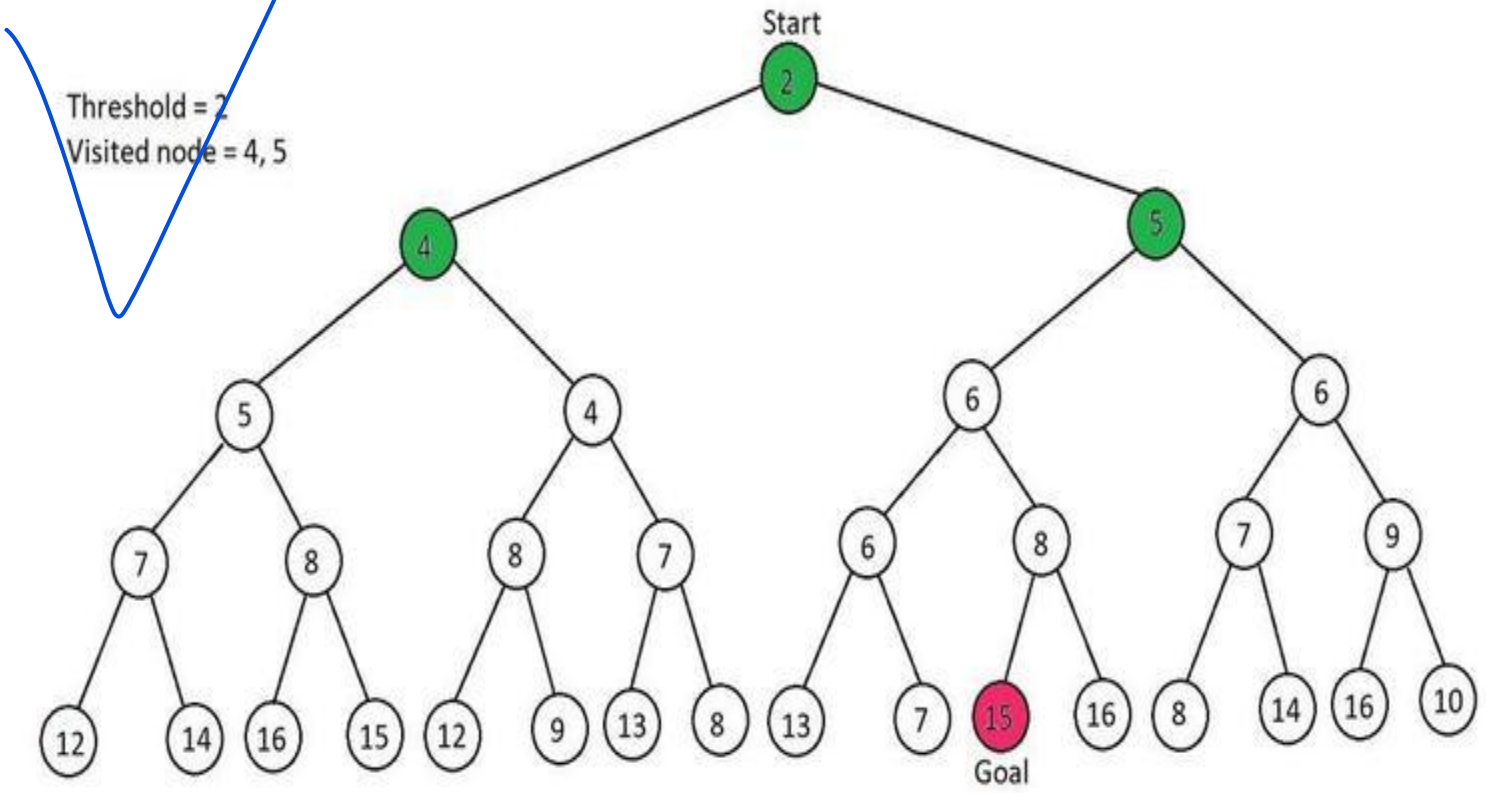
Iterative Deepening A* uses a **heuristic** to choose which nodes to explore and at which depth to stop.

Its opposed to **Iterative Deepening DFS:-** which utilizes simple depth to determine when to end the current iteration and continue with a higher depth.

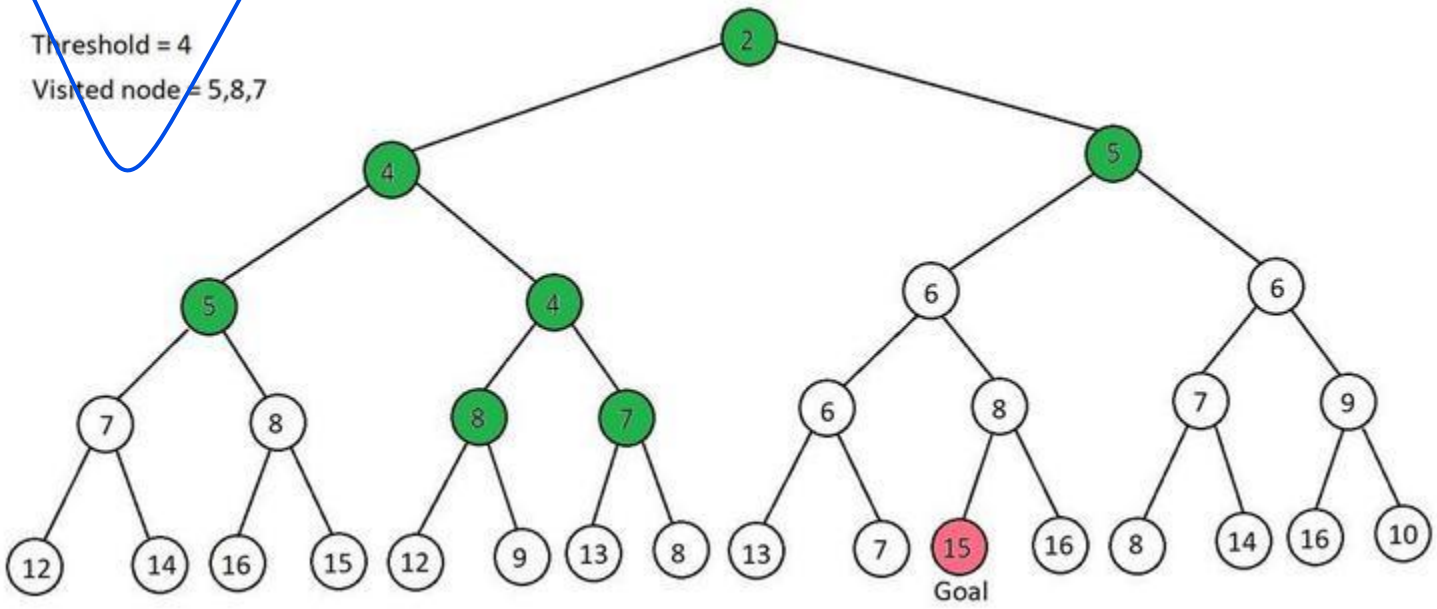
memory-limited version of **A*** is called **IDA***.

IDA* algorithm uses **F-score as a heuristic function** that is used to estimate the cost of reaching the goal state from a given state.

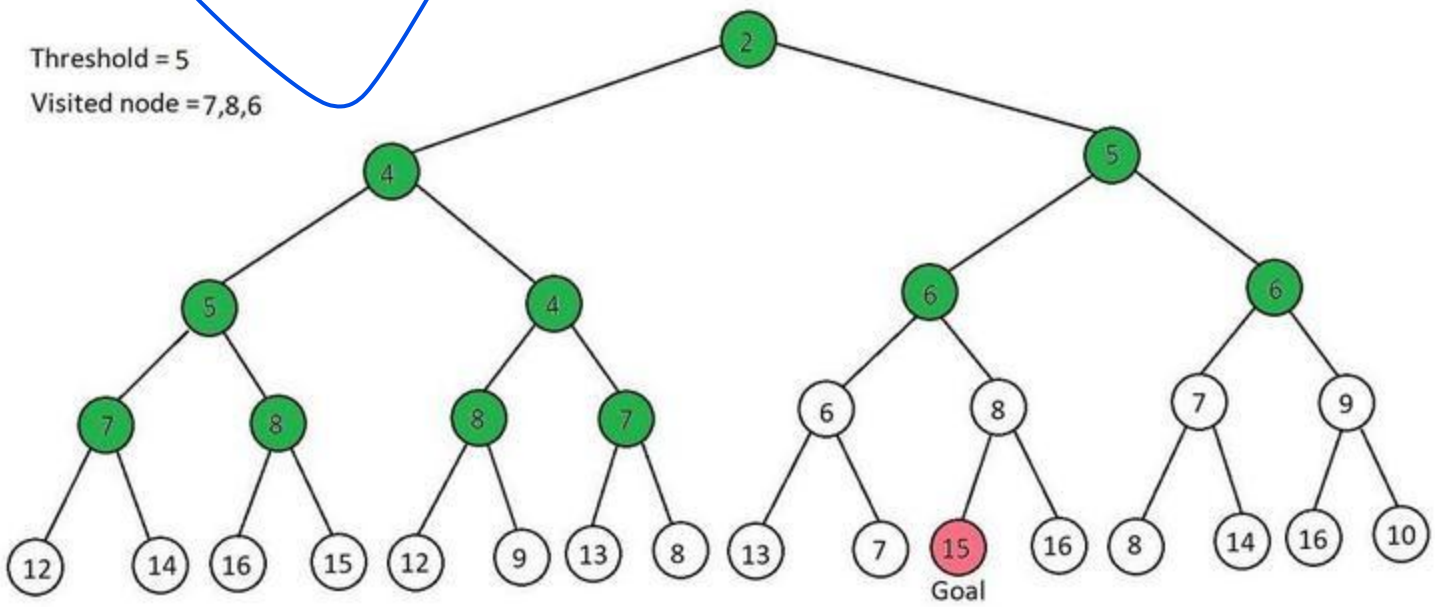
Example : **Iteration 1**



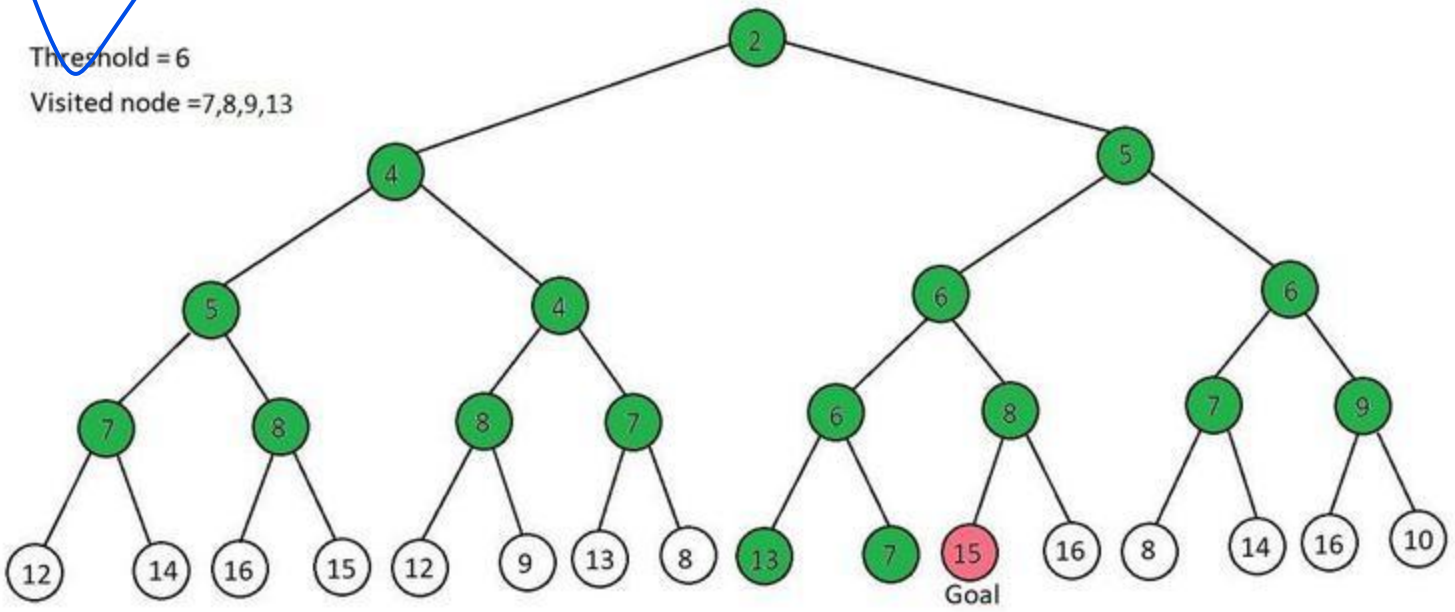
Iteration 2


Threshold = 4
Visited node = 5,8,7
2
4
5
5
4
6
6
7
8
8
7
6
8
7
9
12
14
16
15
12
9
13
8
13
7
15
16
8
14
16
10
Goal

Iteration 3

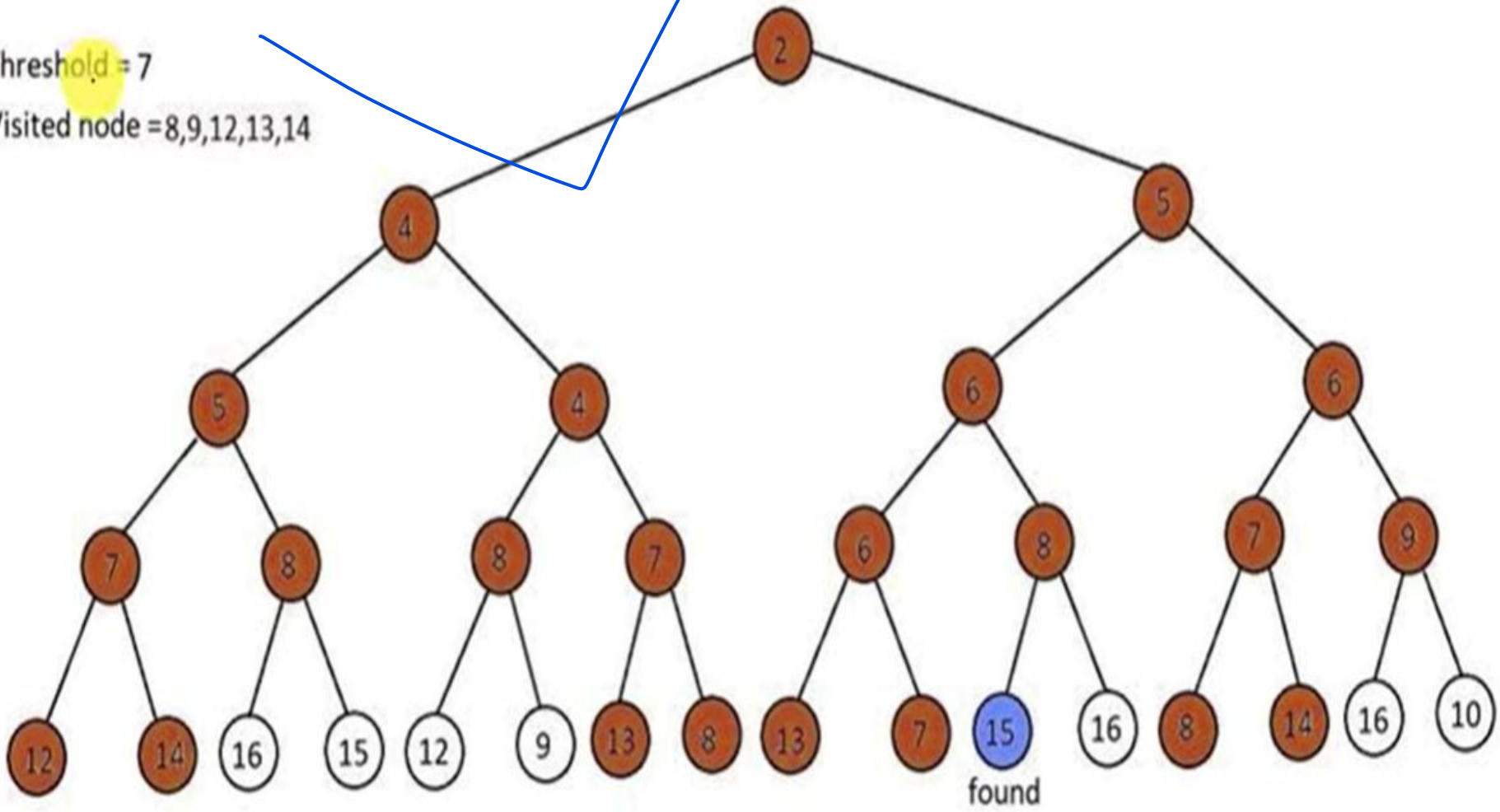

Threshold = 5
Visited node = 7,8,6
2
4
5
5
4
6
6
7
8
8
7
6
8
7
9
12
14
16
15
12
9
13
8
13
7
15
Goal
16
8
14
16
10

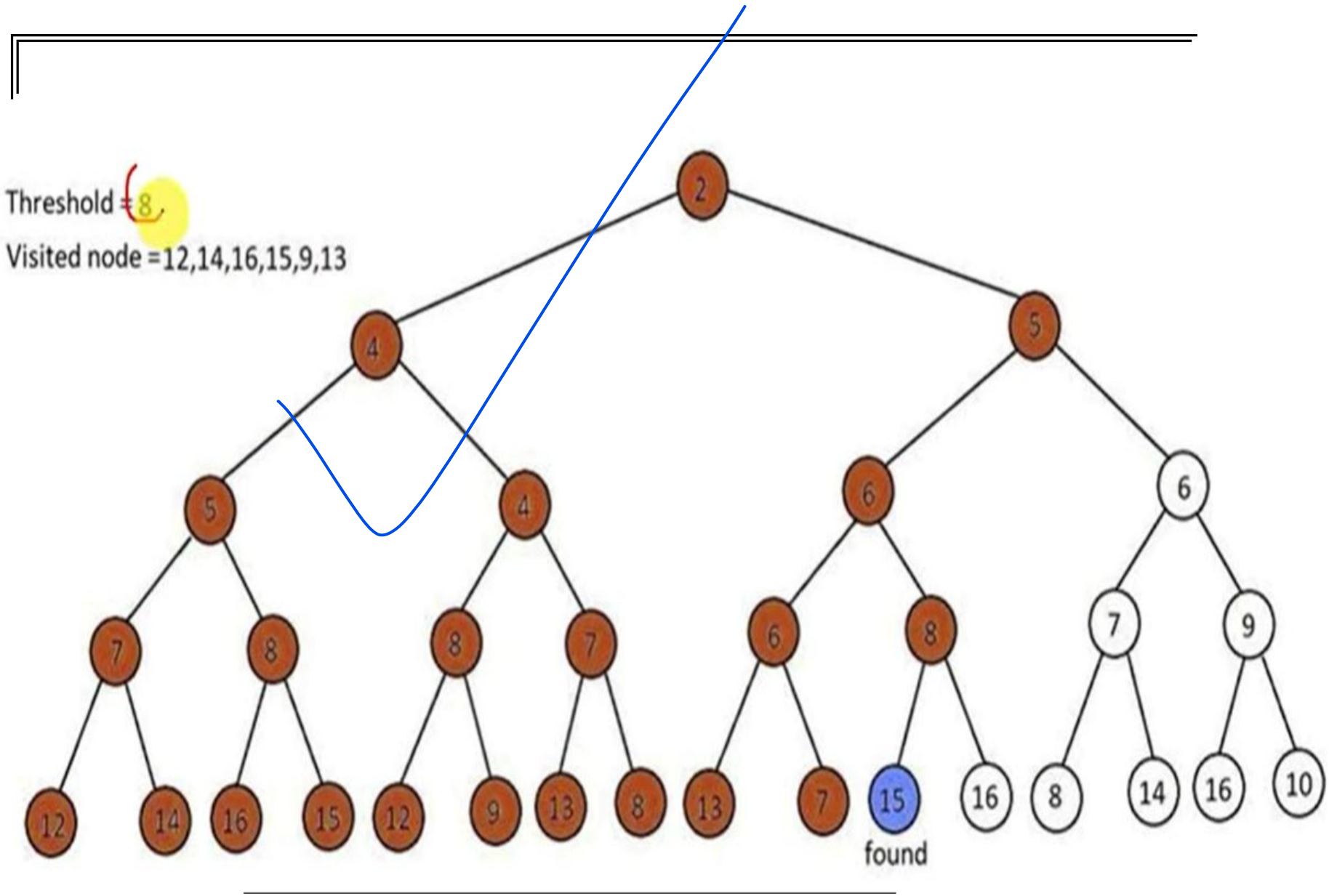
Iteration 4


Threshold = 6
Visited node =7,8,9,13
2
4
5
5
4
6
6
7
8
8
7
6
8
7
9
12
14
16
15
12
9
13
8
13
7
15
Goal
16
8
14
16
10

Threshold = 7

Visited node =8,9,12,13,14




Threshold = 8
Visited node =12,14,16,15,9,13
2
4
5
5
4
6
6
7
8
8
7
6
8
7
9
12
14
16
15
12
9
13
8
13
7
15
16
8
14
16
10
found