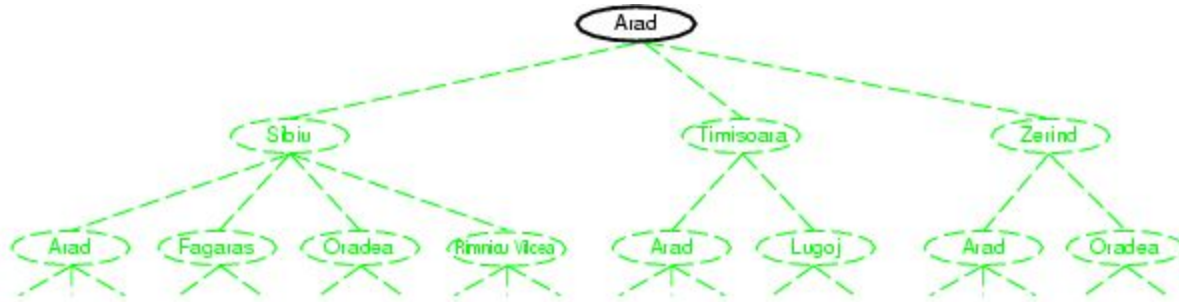


Tree search algorithms

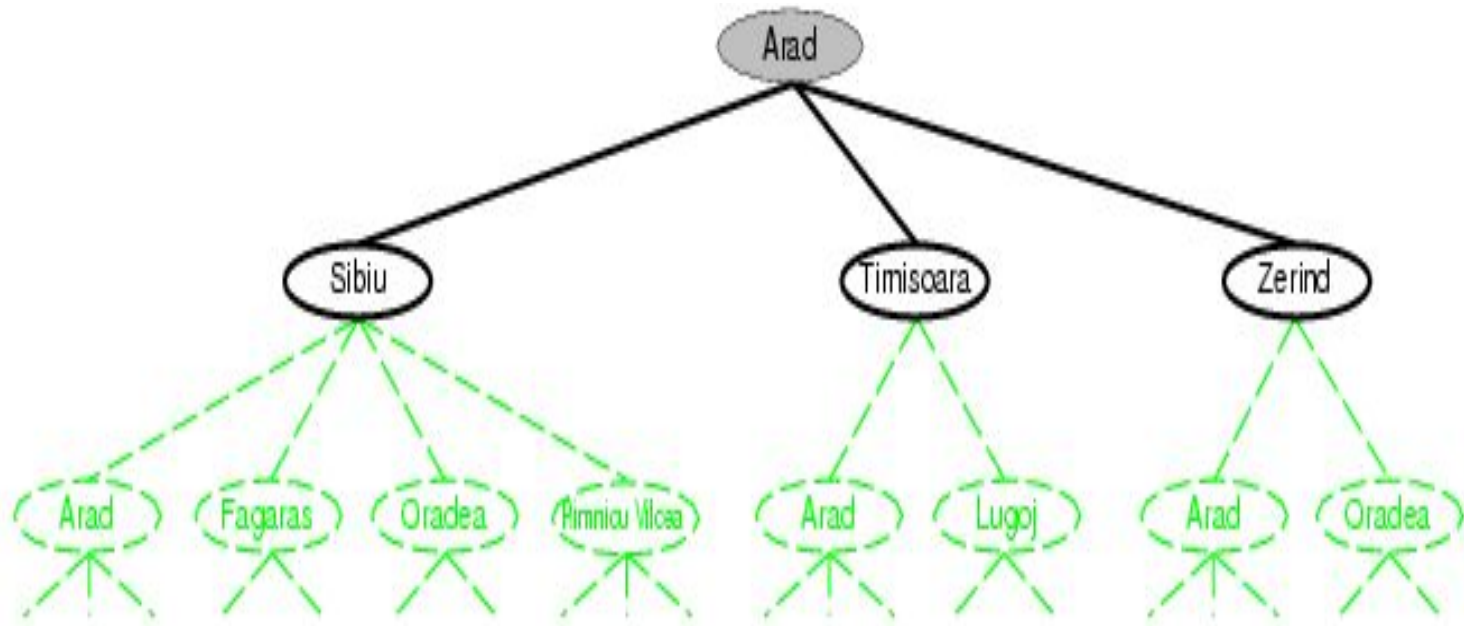
- Basic idea:
 - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. ~expanding states)

```
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
```

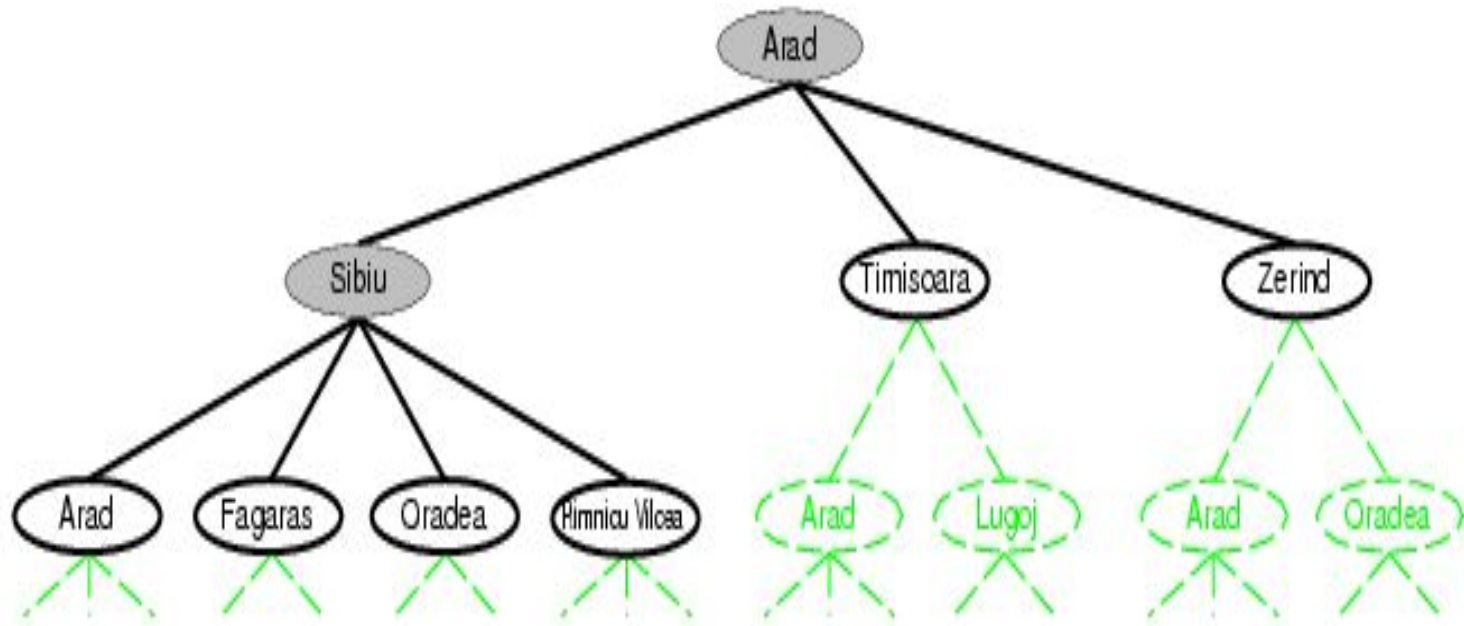
Tree search example



Tree search example



Tree search example



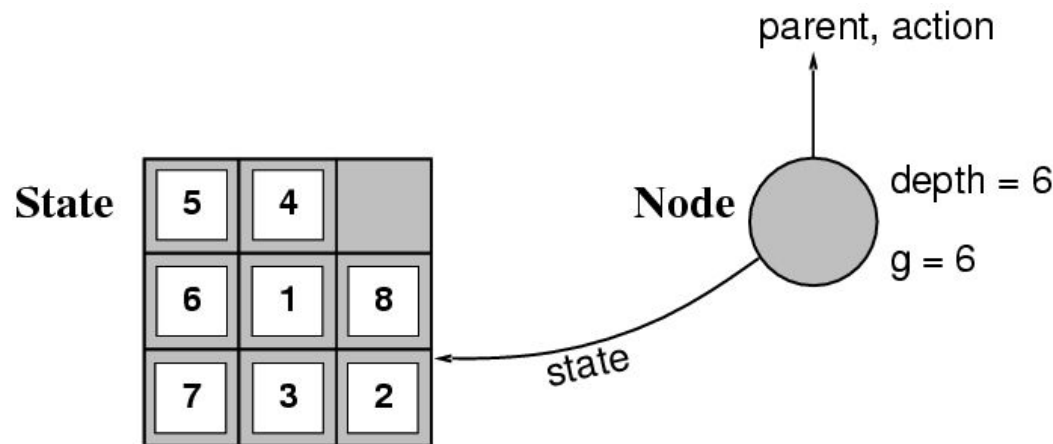
Implementation: general tree search

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

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

Implementation: 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 **state**, **parent node**, **action**, **path cost** $g(x)$, **depth**
- 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 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?
 - **time complexity**: number of nodes generated
 - **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 ∞)

Searching Strategies

- **Blind search** à traversing the search space until the goal nodes is found (might be doing exhaustive search).

- *Techniques* : **Breadth First Uniform Cost ,Depth first, Depth limited, Iterative Deepening search.**

- Guarantees solution.

- **Heuristic search** à search process takes place by traversing search space with applied rules (information).

- *Techniques*: **Greedy Best First Search, A* Algorithm**

- There is no guarantee that solution is found.

Parameters	Informed Search	Uninformed Search
Known as	It is also known as Heuristic Search.	It is also known as Blind Search.
Using Knowledge	It uses knowledge for the searching process.	It doesn't use knowledge for the searching process.
Performance	It finds a solution more quickly.	It finds solution slow as compared to an informed search.
Completion	It may or may not be complete.	It is always complete.
Cost Factor	Cost is low.	Cost is high.
Time	It consumes less time because of quick searching.	It consumes moderate time because of slow searching.
Direction	There is a direction given about the solution.	No suggestion is given regarding the solution in it.
Implementation	It is less lengthy while implemented.	It is more lengthy while implemented.
Efficiency	It is more efficient as efficiency takes into account cost and performance. The incurred cost is less and speed of finding solutions is quick.	It is comparatively less efficient as incurred cost is more and the speed of finding the Breadth-First solution is slow.

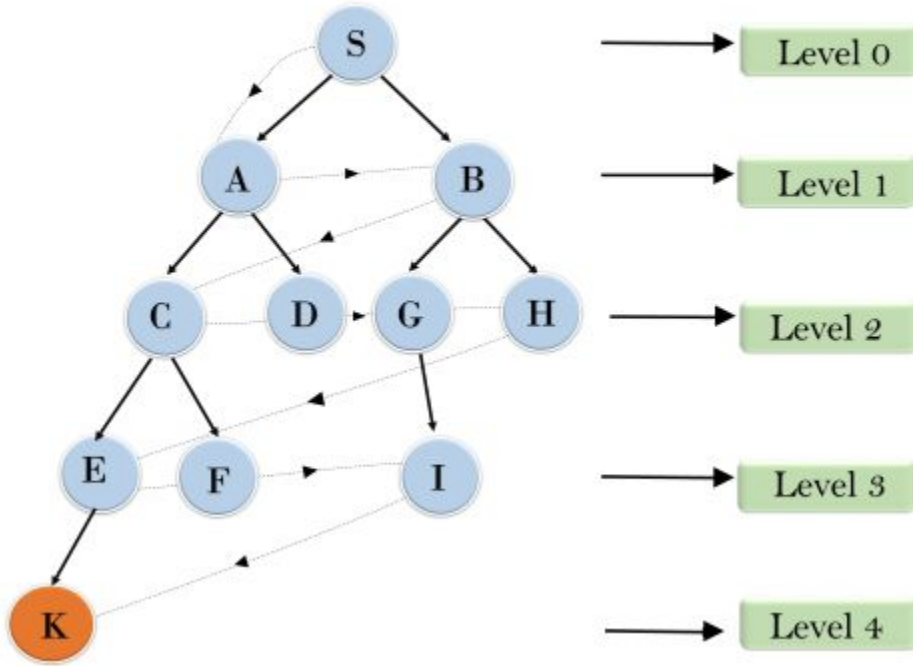
Parameters	Informed Search	UnInformed Search
Computational requirements	Computational requirements are lessened.	Comparatively higher computational requirements.
Size of search problems	Having a wide scope in terms of handling large search problems.	Solving a massive search task is challenging.
Examples of Algorithms	<ul style="list-style-type: none"> • Greedy Search • A* Search • AO* Search • Hill Climbing Algorithm 	<ul style="list-style-type: none"> • Depth First Search (DFS) • Breadth First Search (BFS) • Branch and Bound

Uninformed search strategies

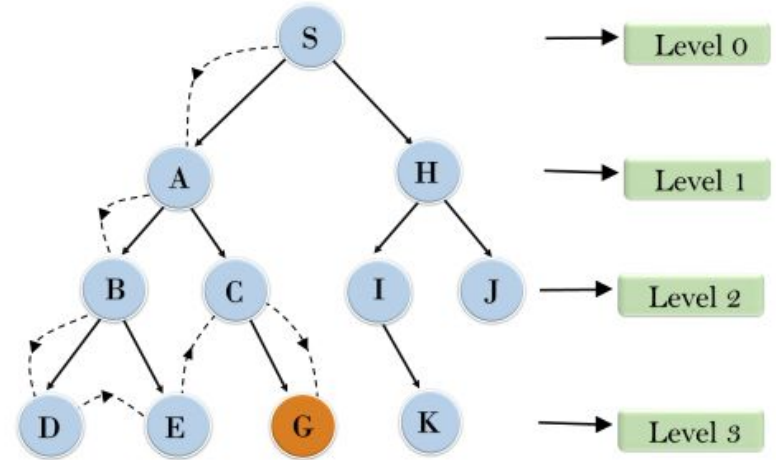
- **Uninformed** search strategies use only the information available in the problem definition
 - Breadth-first search
 - Uniform-cost search
 - Depth-first search
 - Depth-limited search
 - Iterative deepening search
 - Bidirectional Search

Examples

Breadth First Search

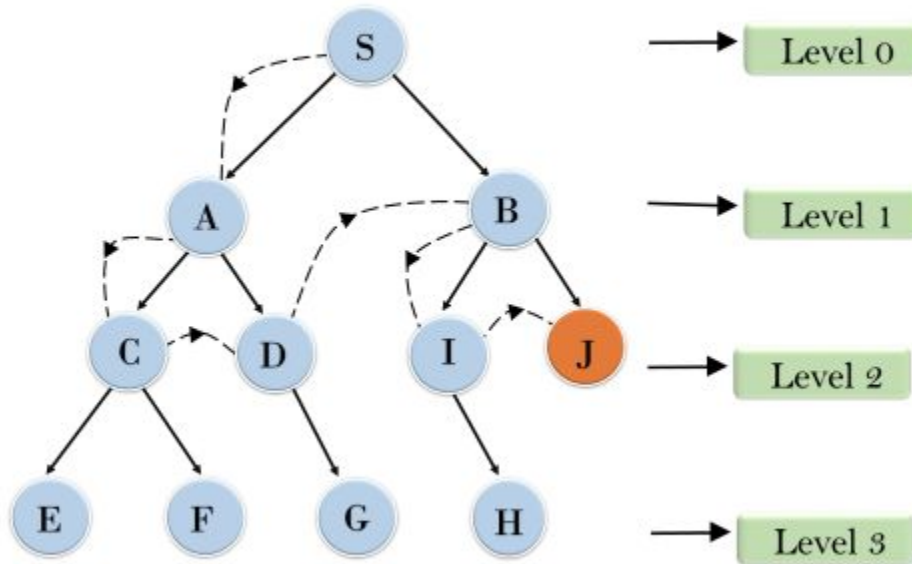


Depth First Search

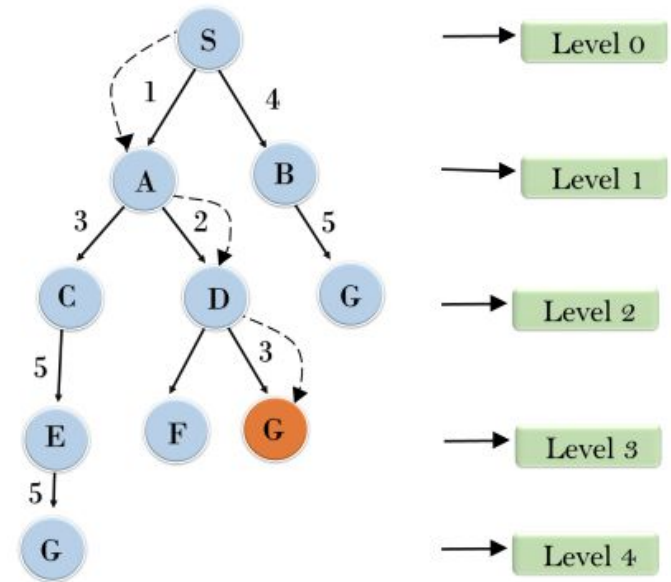


Examples

Depth Limited Search

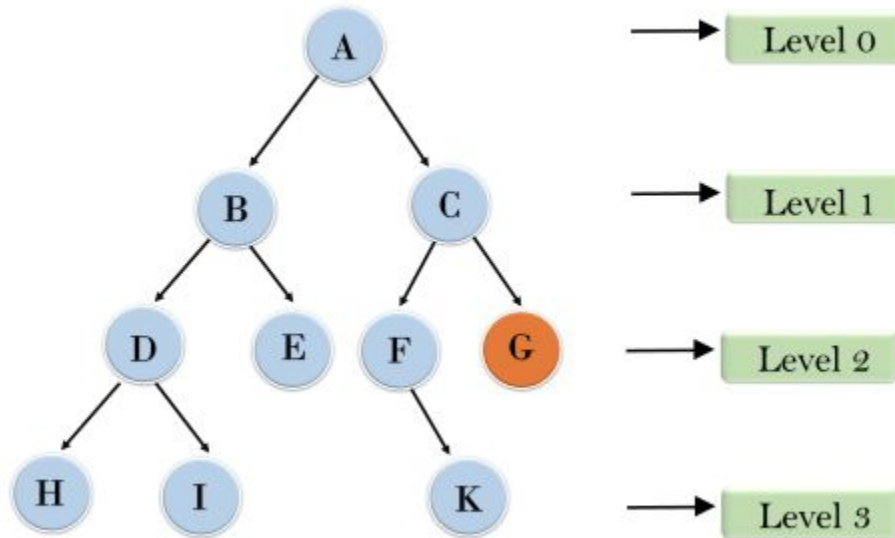


Uniform Cost Search



Examples

Iterative deepening depth first search



1'st Iteration-----> A

2'nd Iteration-----> A, B, C

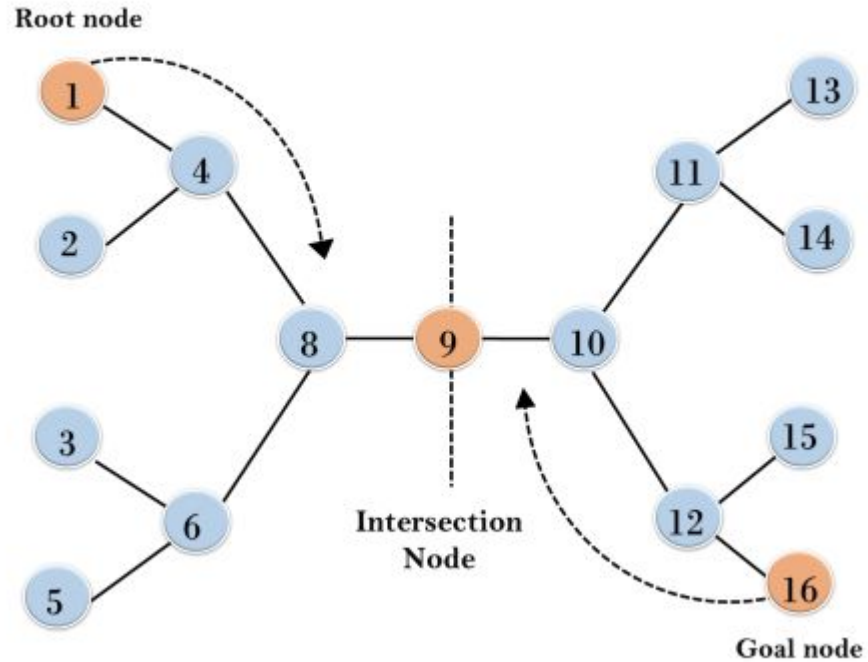
3'rd Iteration----->A, B, D, E, C, F, G

4'th Iteration----->A, B, D, H, I, E, C, F, K, G

In the fourth iteration, the algorithm will find the goal node.

Examples

Bidirectional Search



Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete	Yes ^a	Yes ^{a,b}	No	No	Yes ^a	Yes ^{a,d}
Time Complexity	$O(b^d)$	$O(b^{(b+c/E)})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space Complexity	$O(b^d)$	$O(b^{(b+c/E)})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal	Yes ^c	Yes	No	No	Yes ^c	Yes ^{c,d}

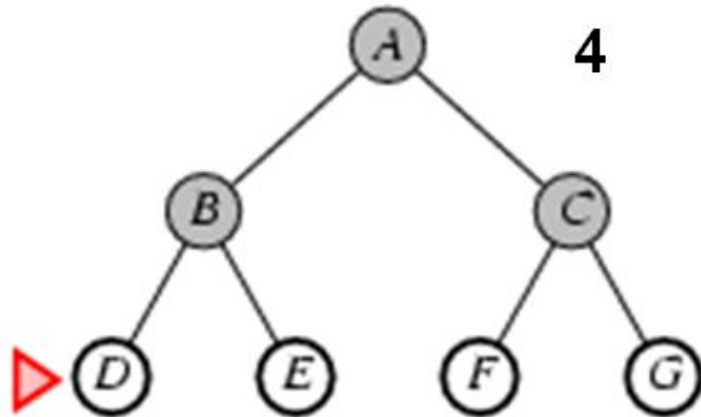
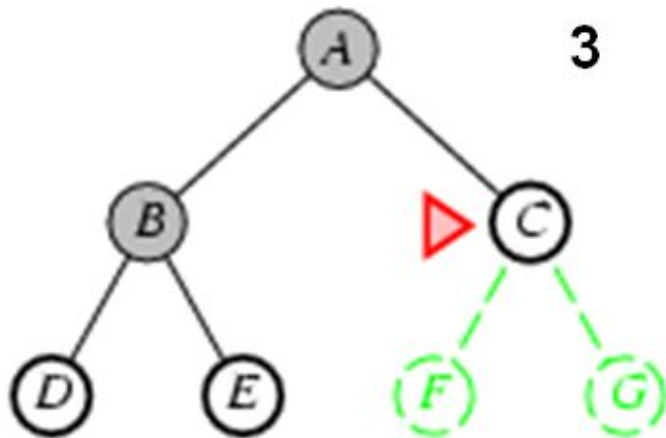
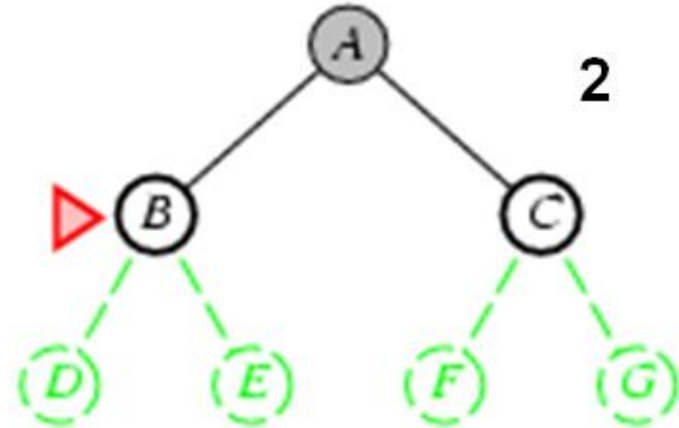
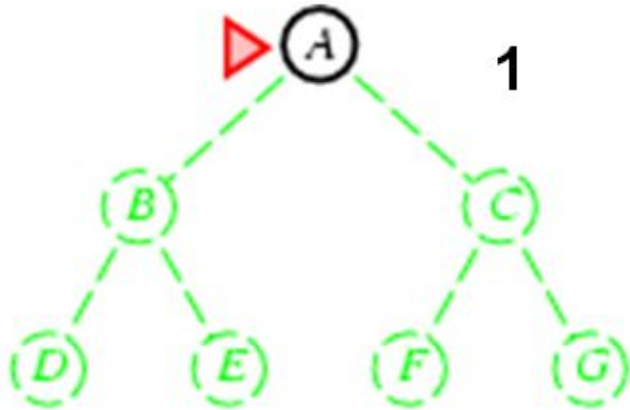
b -> branching factor
 d -> depth of the shallowest solution
 m -> max. depth of search tree
 l -> depth limit

a -> complete if b is finite
 b -> complete if step costs $\geq E$ for $E > 0$
 c -> optimal if step costs are same
 d -> if both directions use BFS

Breadth-first search

- Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.
- **Implementation:**
 - *fringe* is a FIFO queue, i.e., new successors go at end

Blind Search : Breadth First Search



Properties of breadth-first search

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

note:Space is the bigger problem (more than time)

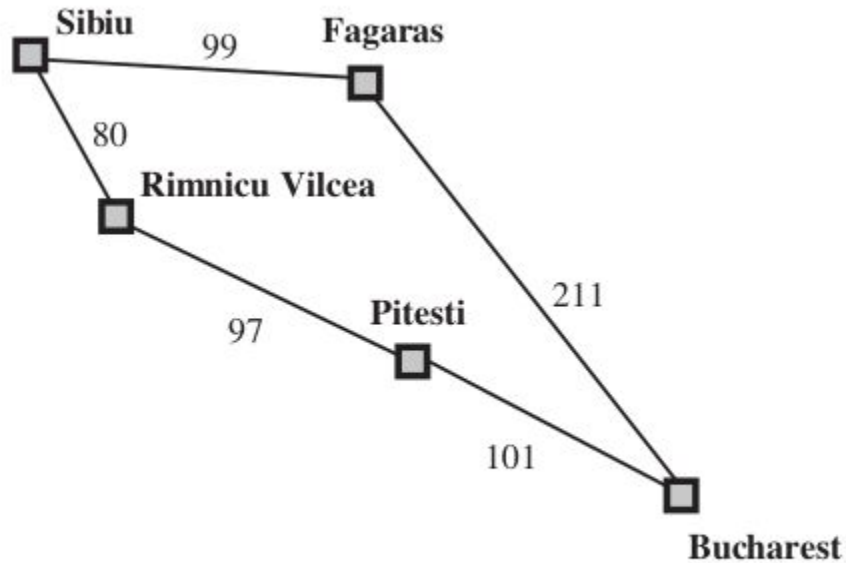
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

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

Uniform-cost search

- When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node.
- what if cost of all possibilities is not same?
- By a simple extension, we can find an algorithm that is optimal
- **uniform-cost search expands the node n with the lowest path cost $g(n)$.**
- Implementation: using priority queue ordered by **g** .

- Uniform-cost search does not care about the number of steps a path has, but only about their total cost. Therefore, it will **get stuck in an infinite loop** if there is a path with an infinite sequence of zero-cost actions—for example, a sequence of NoOp actions.
- Completeness is guaranteed provided the cost of every step exceeds some small positive constant ϵ .
- Uniform-cost search is guided by path costs rather than depths



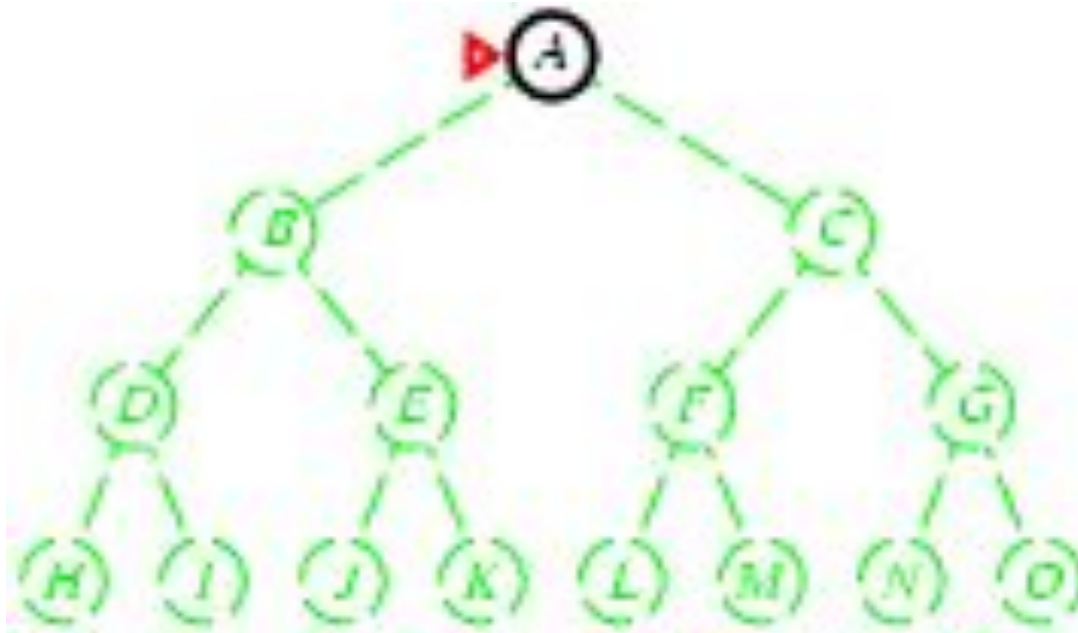
5 Part of the Romania state space, selected to illustrate uniform-cost search.

Uniform-cost search

- Complete? Yes, if step cost $\geq \epsilon$
- Time? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\text{ceiling}(C^*/\epsilon)})$ where C^* is the cost of the optimal solution
- Space? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\text{ceiling}(C^*/\epsilon)})$
- Optimal? Yes – nodes expanded in increasing order of $g(n)$

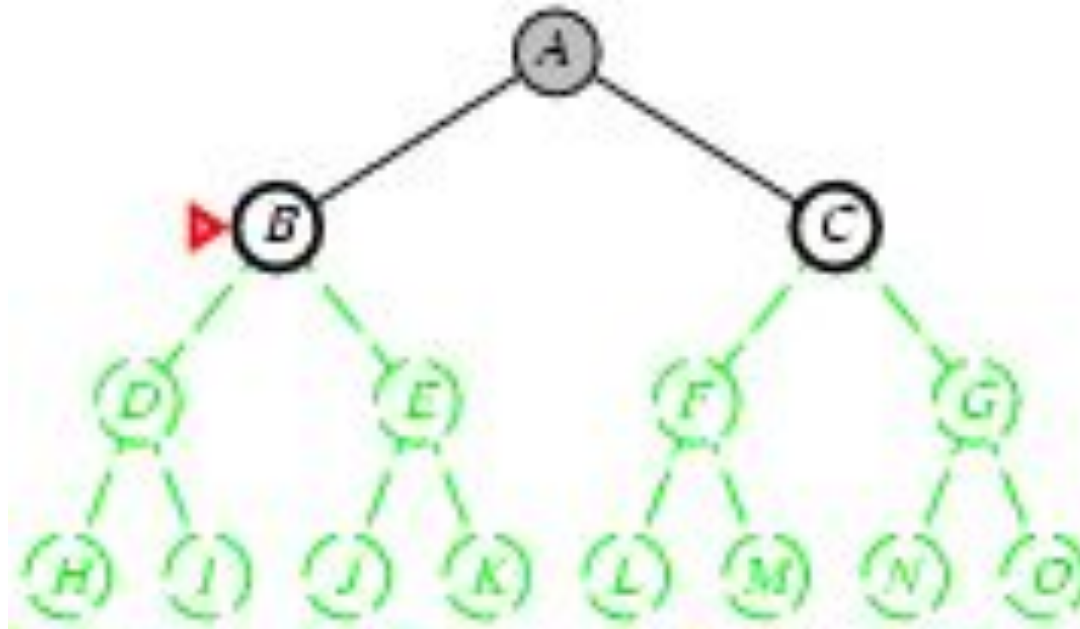
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at front



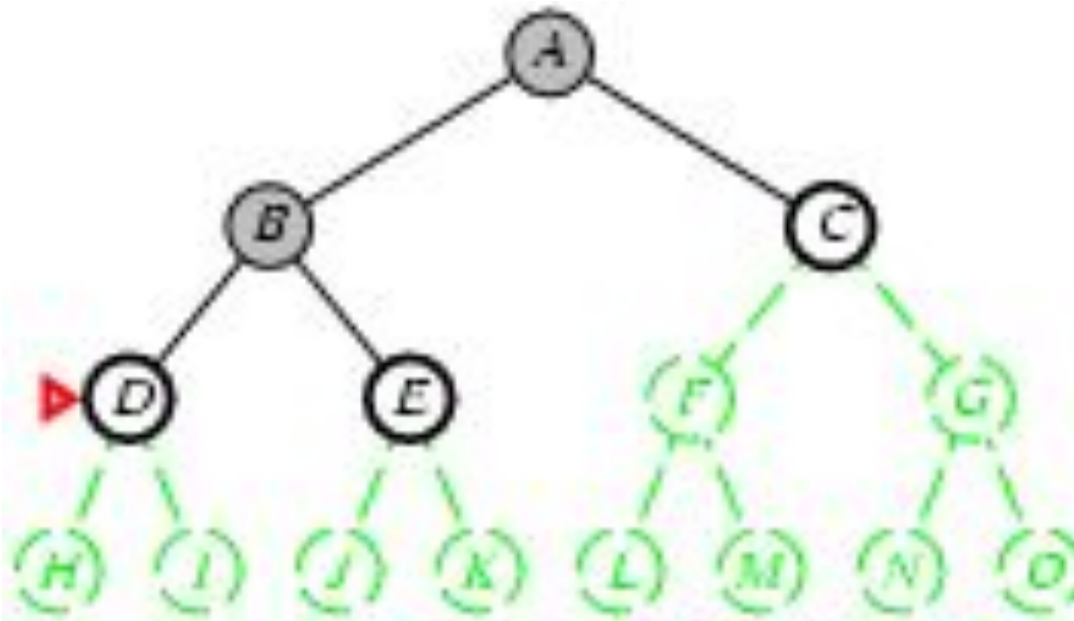
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at *front*



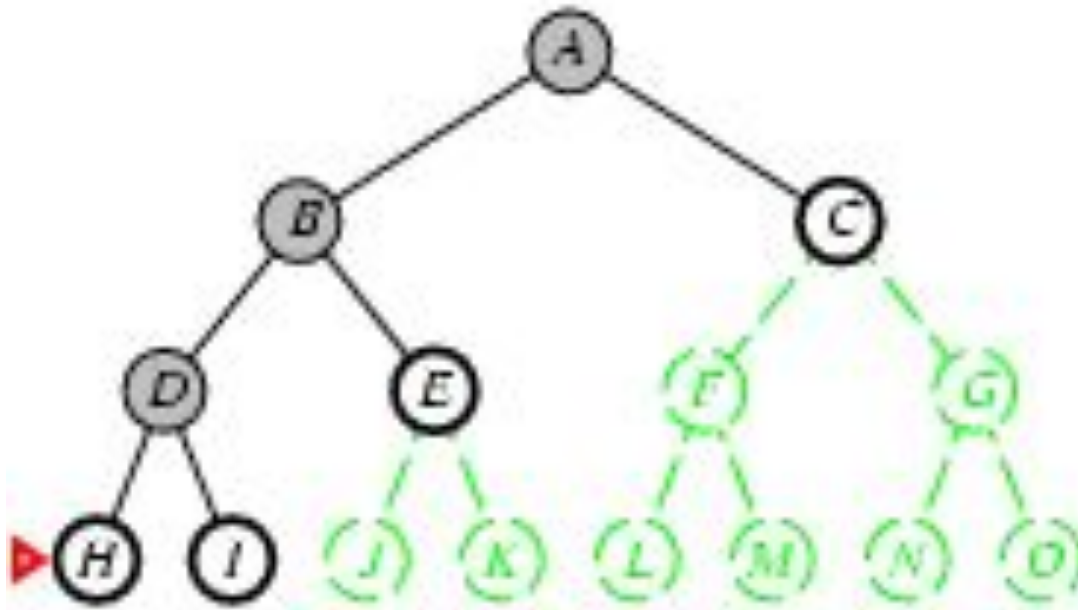
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at front



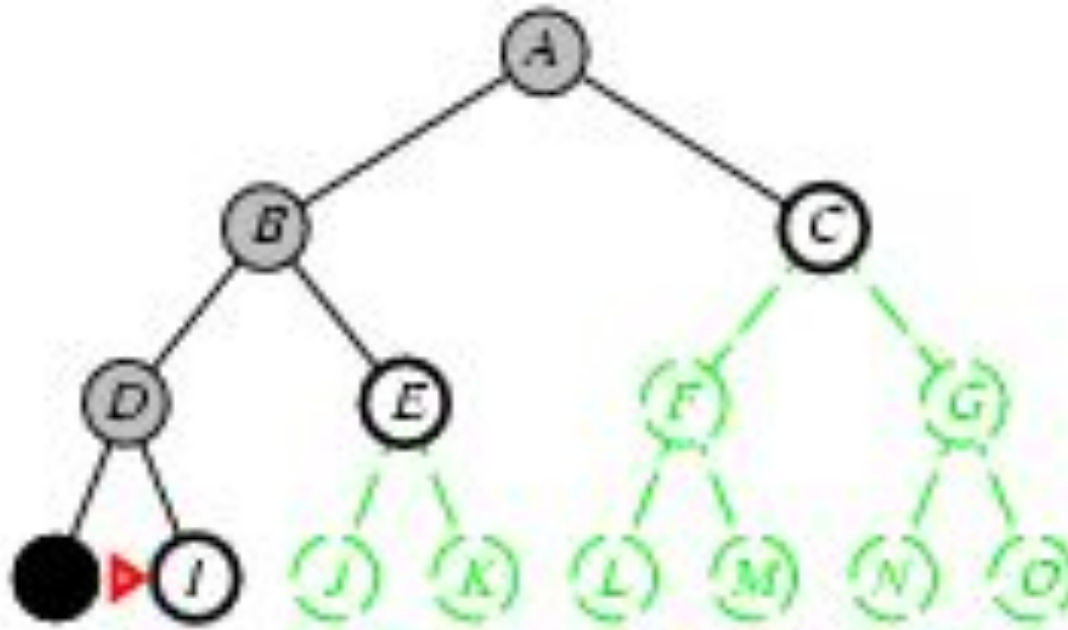
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at *front*



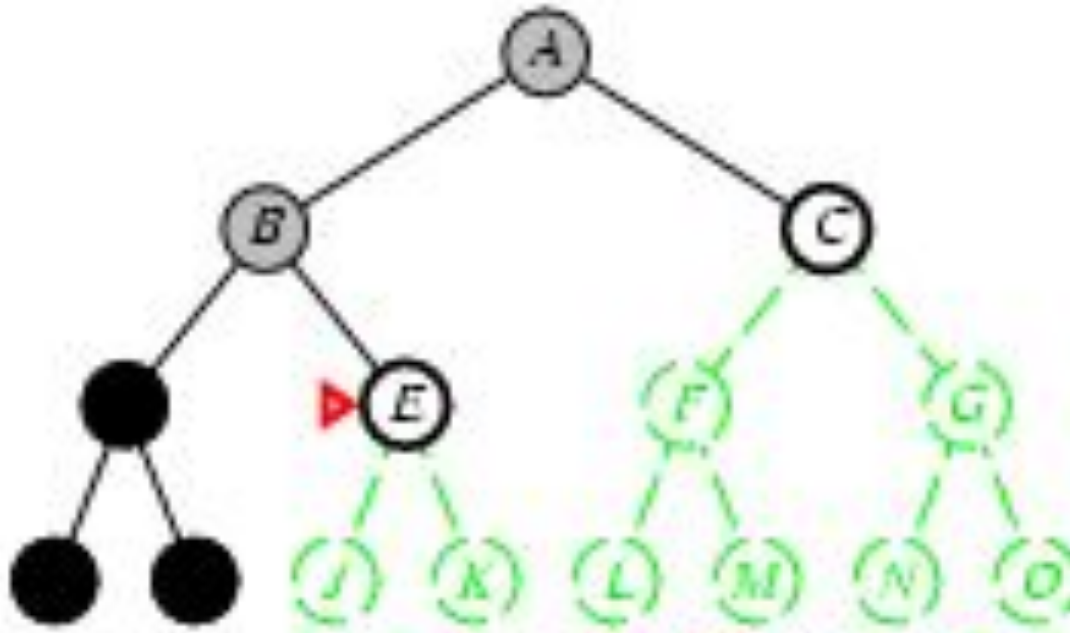
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at *front*



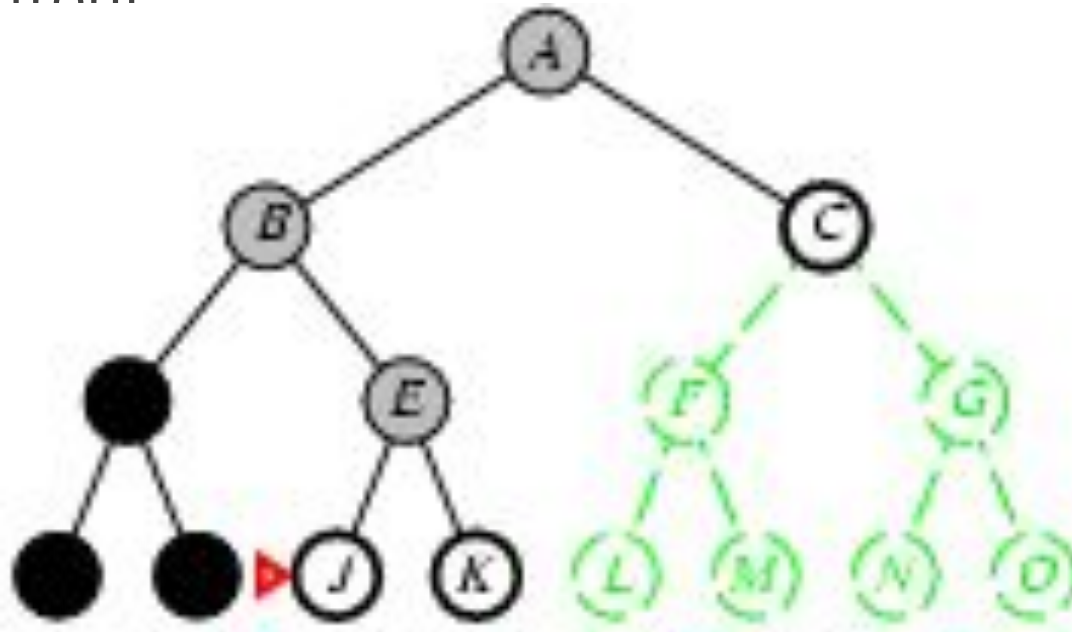
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at *front*



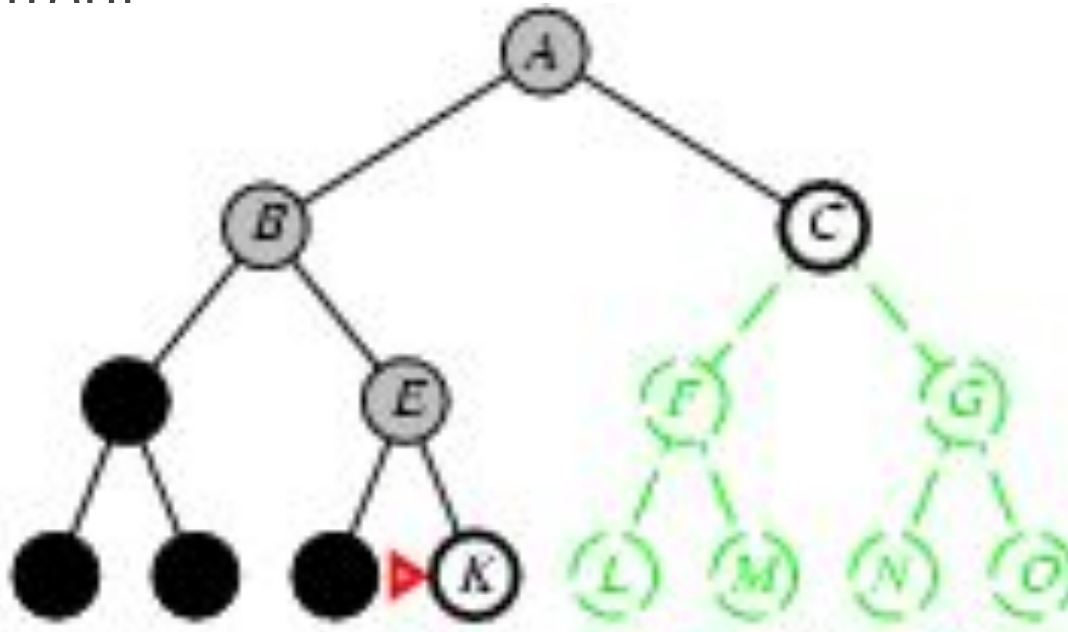
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at *front*



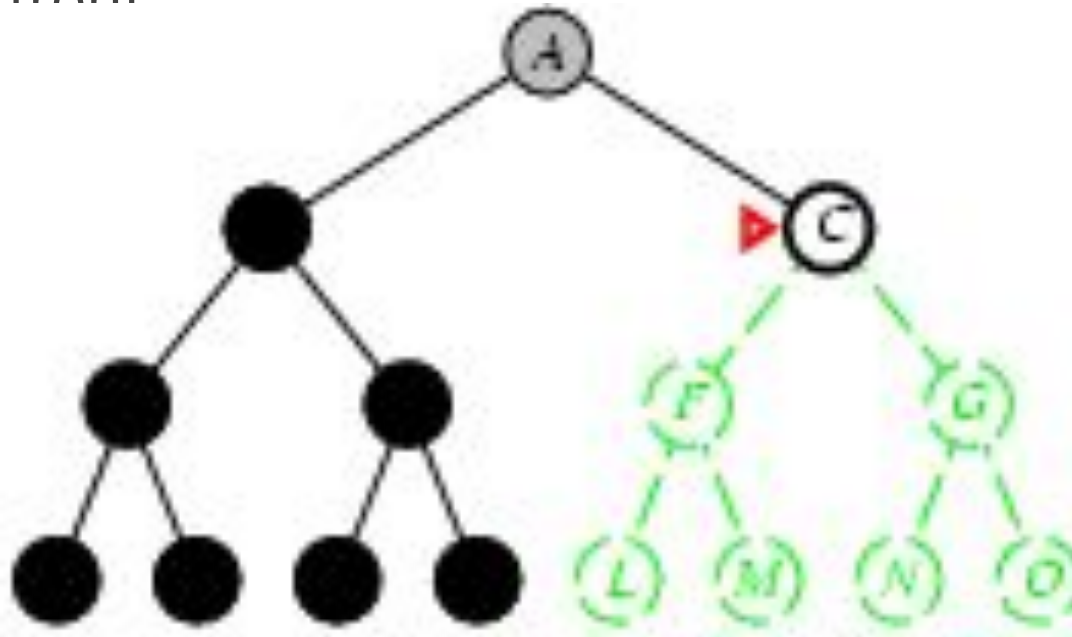
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at *front*



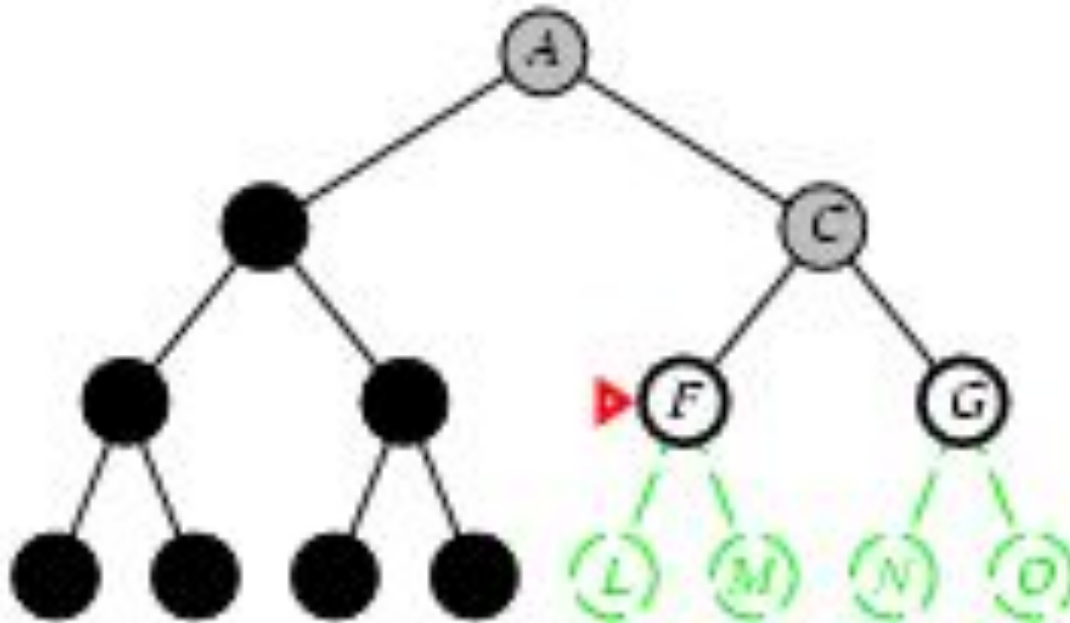
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at *front*



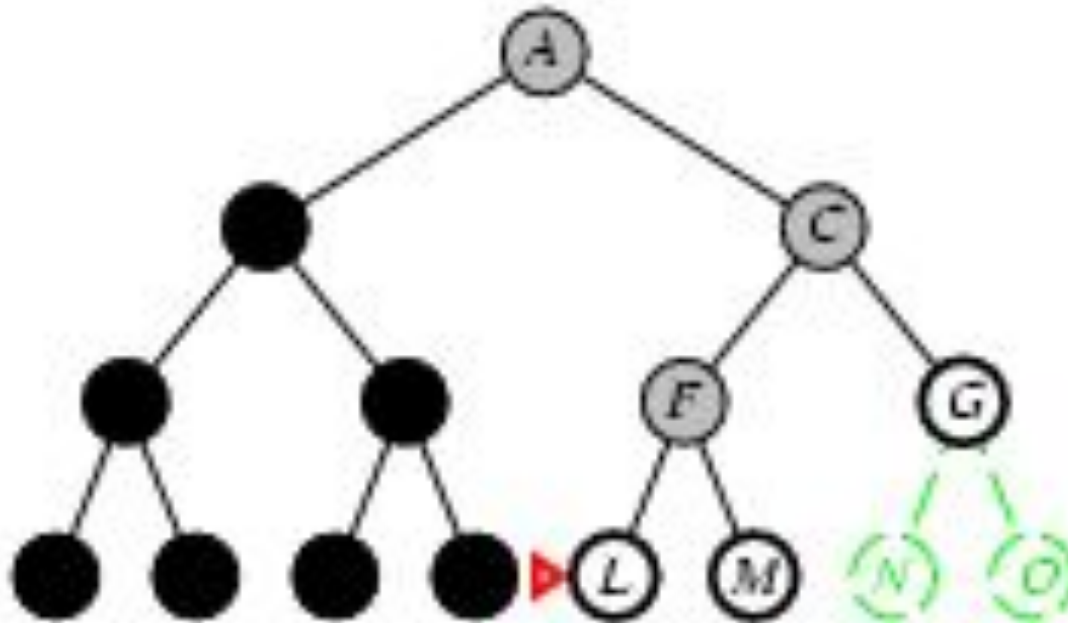
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at *front*



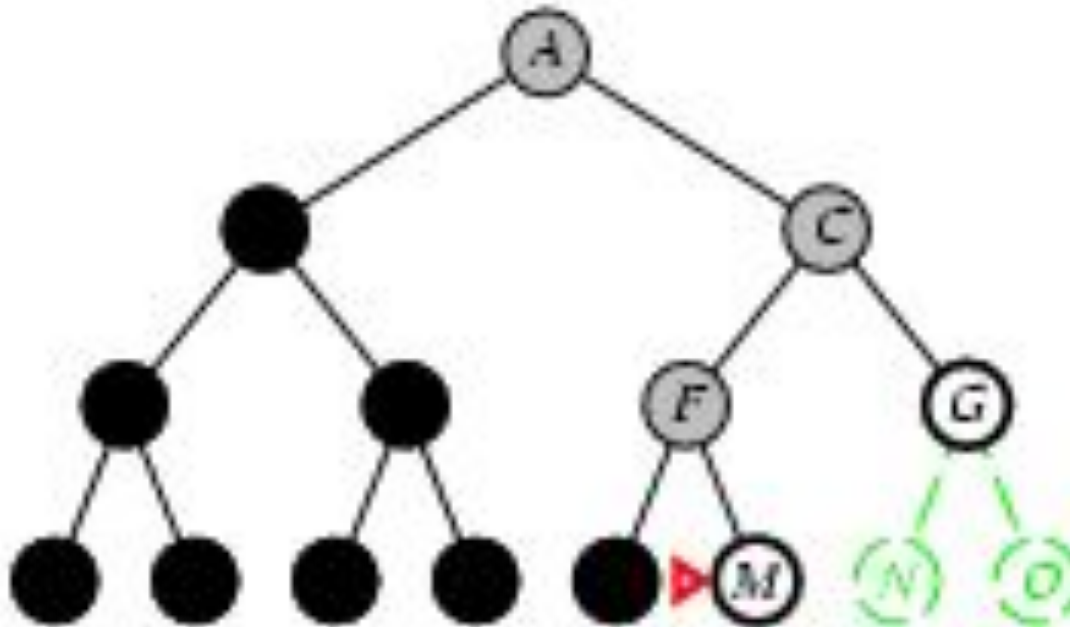
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at *front*

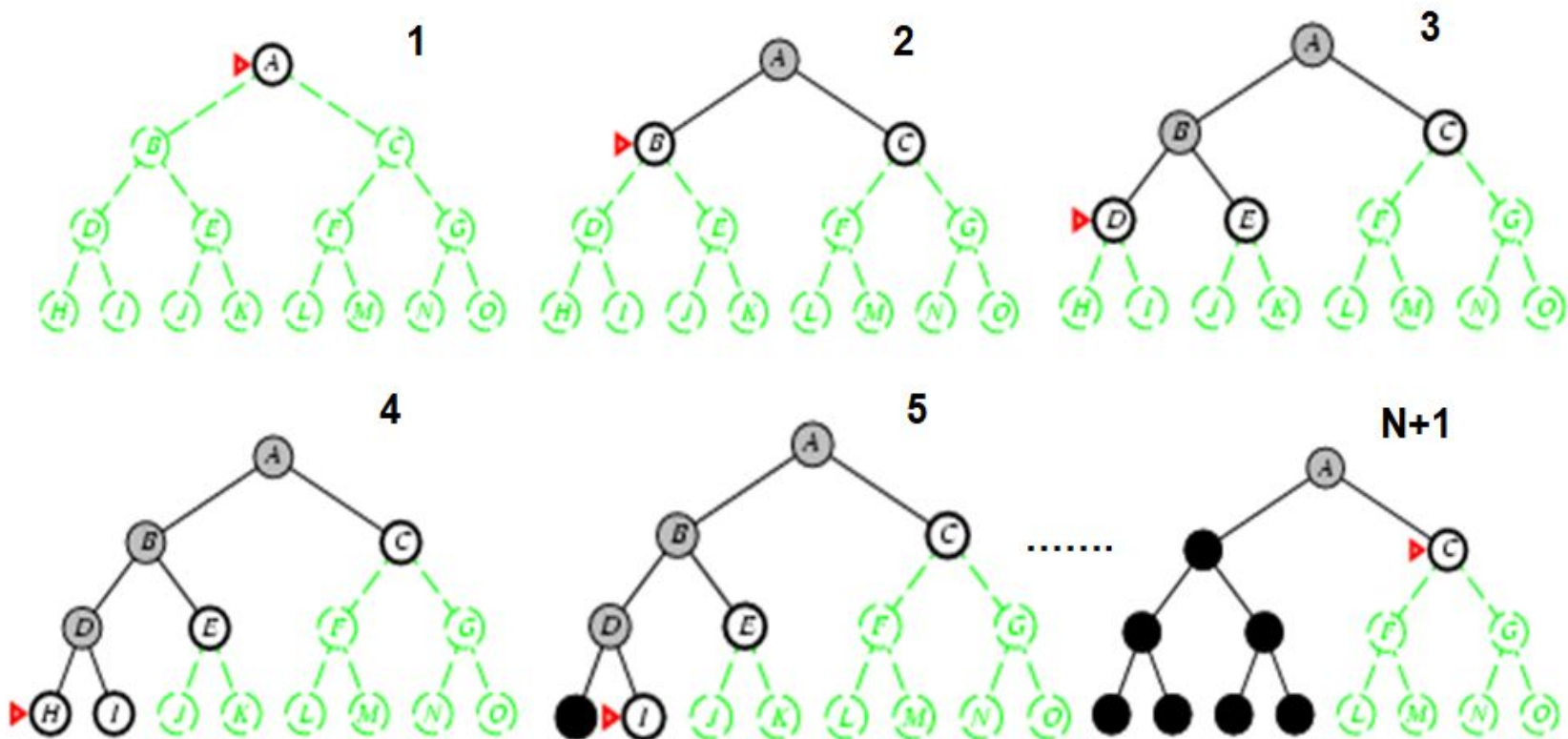


Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
 - *fringe* = LIFO queue, i.e., put successors at *front*



Blind Search : Depth First Search (DFS)



Properties of depth-first search

- **Complete?** No: fails in infinite-depth spaces, spaces with loops
- Modify to avoid repeated states along path
complete in finite spaces
- **Time?** $O(bm)$: 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

Depth-limited search(DLS)

- The embarrassing failure of depth-first search in **infinite state spaces** can be alleviated by supplying depth-first search with a **predetermined depth limit l** .
- Nodes at depth **l** are treated as if they have no successors.
- The depth limit solves the infinite-path problem.

Depth-limited search(DLS)

- It also introduces an additional source of **incompleteness if we choose $l < d$** , that is, the shallowest goal is beyond the depth limit.
(This is likely when d is unknown.)
- Depth-limited search will also be **nonoptimal if we choose $l > d$** .
- **Time complexity : $O(bl)$**
- **Space complexity: $O(bl)$.**
- Depth-first search can be viewed as a **special case of depth-limited search with $l = \infty$** .

Iterative deepening search

- Iterative deepening search (or iterative deepening depth-first search) is often used in combination with depth-first tree search, that finds the best depth limit.
- It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found.
- This will occur when the depth limit reaches d , the depth of the shallowest goal node.
- Iterative deepening combines the benefits of depth-first and breadth-first search. Like depth-first search, its memory requirements are modest: $O(bd)$

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or fail-  
ure  
  inputs: problem, a problem  
  for depth  $\leftarrow$  0 to  $\infty$  do  
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)  
    if result  $\neq$  cutoff then return result
```

Iterative deepening search / =0



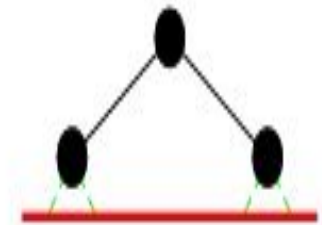
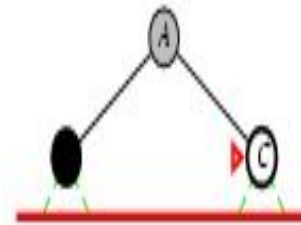
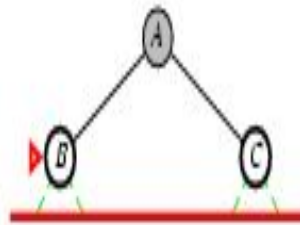
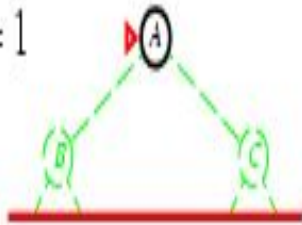
Limit = 0



Iterative deepening search / =1

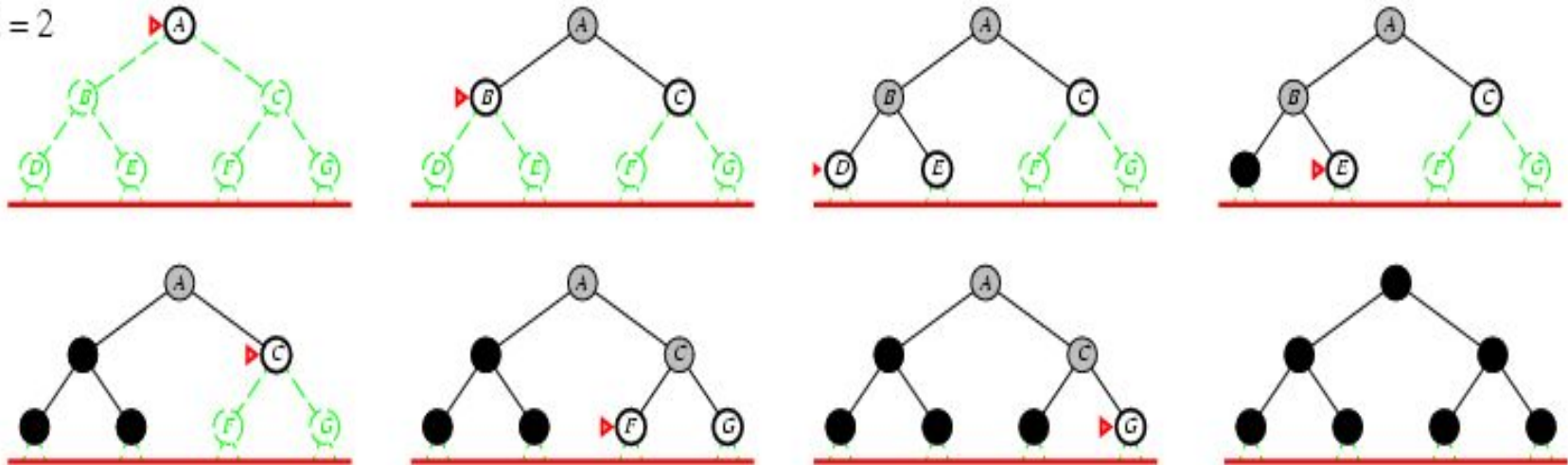


Limit = 1



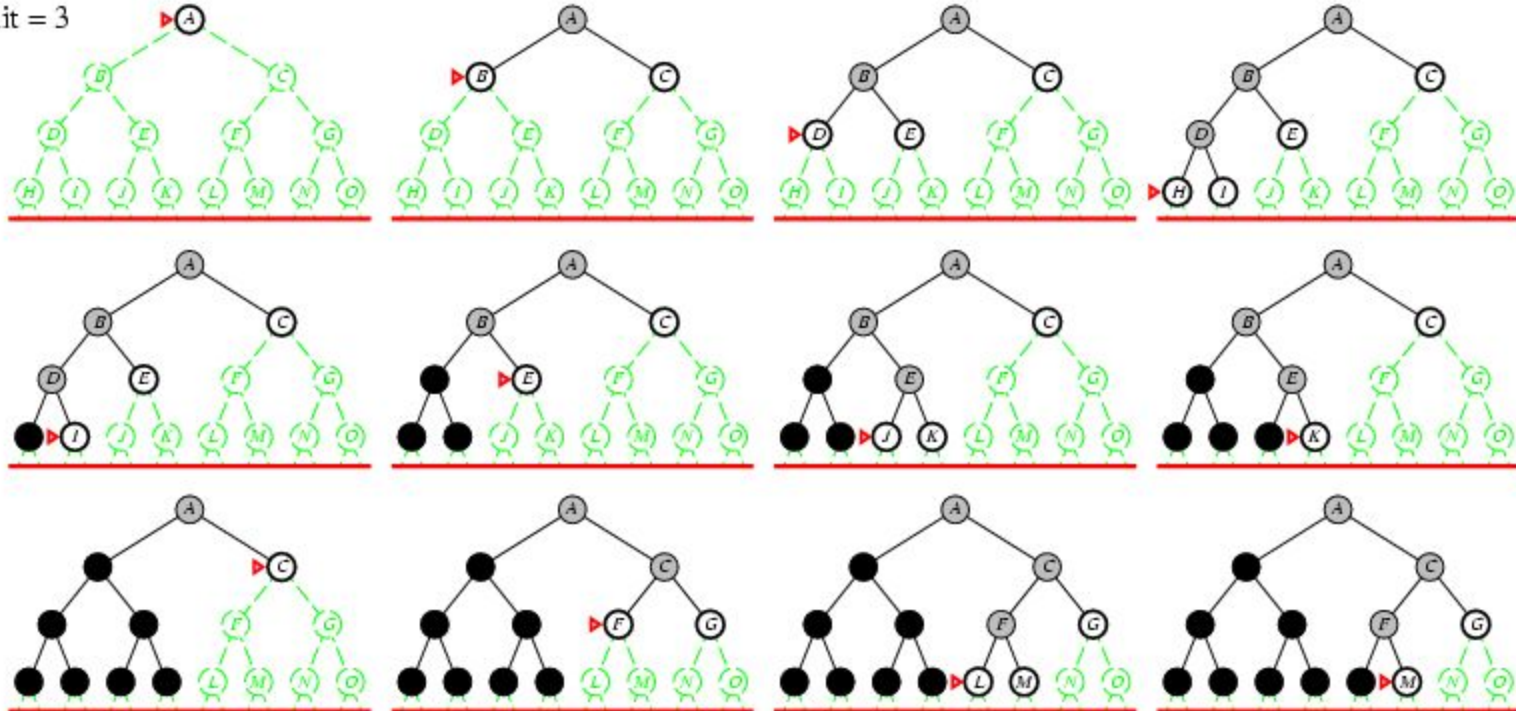
Iterative deepening search / =2

Limit = 2



Iterative deepening search / =3

Limit = 3



Properties of iterative deepening search

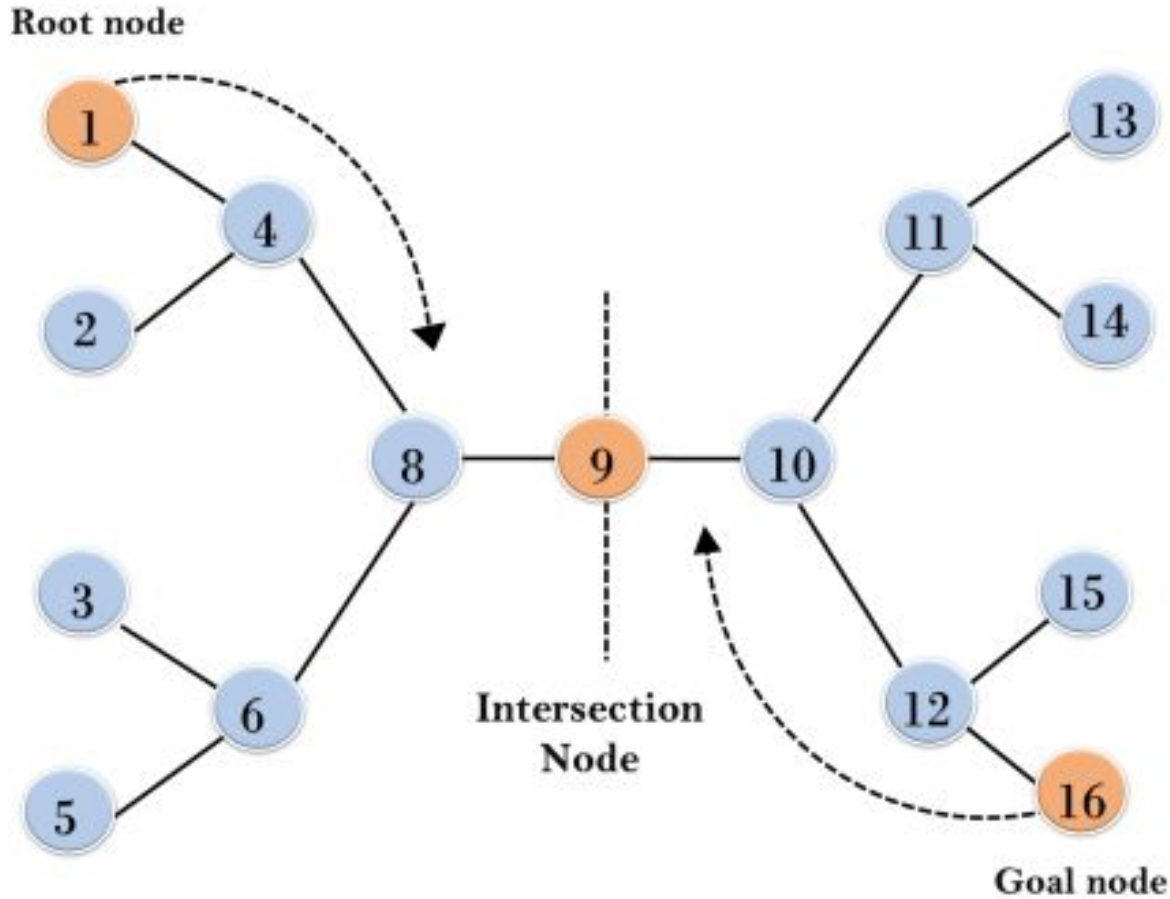
- Complete? Yes
- Time? $(d+1)b^0 + d b^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

Iterative deepening search

- Number of nodes generated in a depth-limited search to depth d with branching factor b :
 - $N_{DLS} = b^0 + b^1 + b^2 + \dots + b^{d-2} + b^{d-1} + b^d$
- Number of nodes generated in an iterative deepening search to depth d with branching factor b :
 - $N_{IDS} = (d+1)b^0 + d b^1 + (d-1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + 1b^d$
- For $b = 10, d = 5$,
 - $N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$
 - $N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$
- Overhead = $(123,456 - 111,111)/111,111 = 11\%$

Bidirectional Search

Bidirectional Search



Bidirectional Search

Completeness: Bidirectional Search is complete if we use BFS in both searches.

Time Complexity: Time complexity of bidirectional search using BFS is $O(b^{d/2})$.

Space Complexity: Space complexity of bidirectional search is $O(b^{d/2})$.

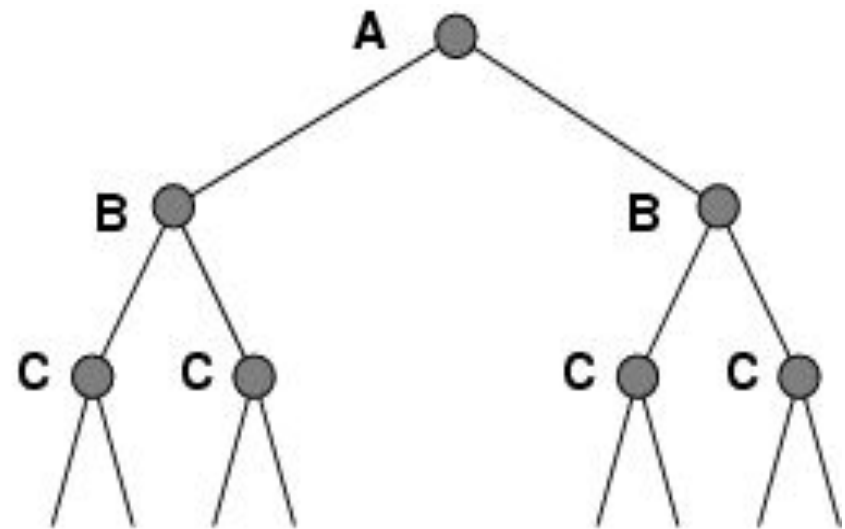
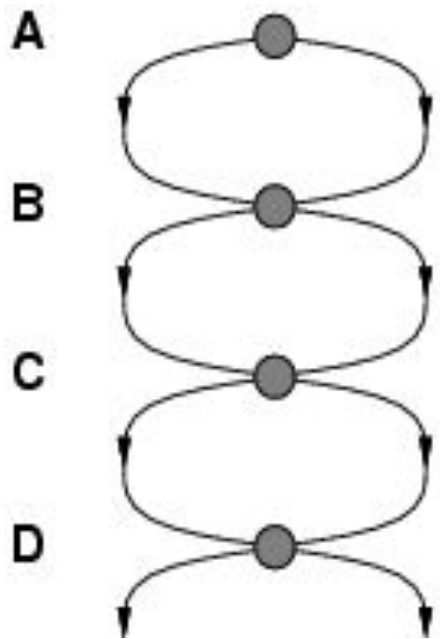
Optimal: Bidirectional search is Optimal.

Summary of algorithms

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

Repeated states

Failure to detect repeated states can turn a linear problem into an exponential one!



Graph search

```
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 SOLUTION(node)
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Avoiding Repeated States

- **Do not return to the parent state** (e.g., in 8 puzzle problem, do not allow the Up move right after a Down move)
- **Do not create solution paths with cycles**
- **Do not generate any repeated states** (need to store and check a potentially large number of states)
- This is done by keeping a list of "**expanded states**" i.e., states whose daughters have already been put on the enqueued list.
- This entails removing states from the "enqueued list" and placing them on an "expanded list"

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

- It uses **problem-specific knowledge** beyond the definition of the problem itself
- can **find solutions more efficiently** than can an uninformed strategy.
- It uses **heuristic cost** .
- Important aspect: formation of **heuristic function $(h(n))$** .
- Heuristic function : **additional knowledge** to guide searching strategy (short cut).

Cost???

$g(n)$ = the cost of getting from the initial node to n .

$h(n)$ = the estimate, according to the heuristic function, of the cost of getting from n to the goal node.

$f(n) = g(n) + h(n)$; intuitively, this is the estimate of the best solution that goes through n .

OPEN AND CLOSED LIST

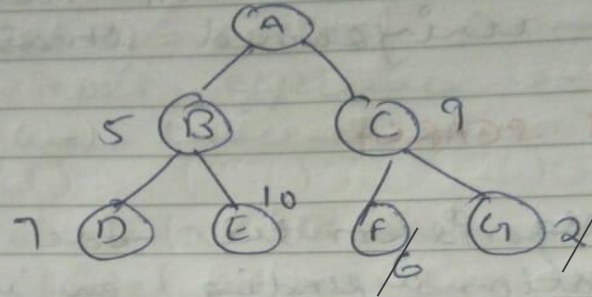
- INFORMED SEARCH uses two lists:
 - the OPEN list - track **nodes that need be examined**,
 - the CLOSED list - track of **nodes that are already examined**.
- Initially, the OPEN list contains just the initial node, and the CLOSED list is empty.
- **Each node n in the graph maintains** the following additional information:
 - $g(n)$ = the cost of getting from the initial node to n.
 - $h(n)$ = heuristic cost from n to the goal node.
 - $f(n) = g(n) + h(n)$ total cost

Best-first search

- In Best-first search a node is selected for expansion based on an **evaluation function, $f(n)$** . and here **$f(n)=g(n)$**
- The evaluation function is construed as a cost estimate, so the node with the **lowest evaluation is expanded first**.
- The implementation of best-first graph search is identical to that for uniform-cost search except for the use of f instead of g to order the priority queue.

Example: best first search

Ex:

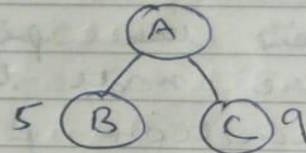


I



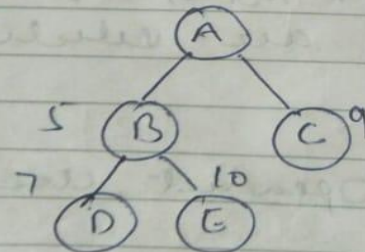
$OL = \{A\}$
 $CL = \{A\}$
 $EM = \{A\}$

II



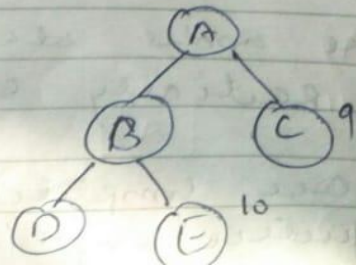
$OL = \{B, C\}$
 $CL = \{A\}$
 $EM = \{B\}$

III



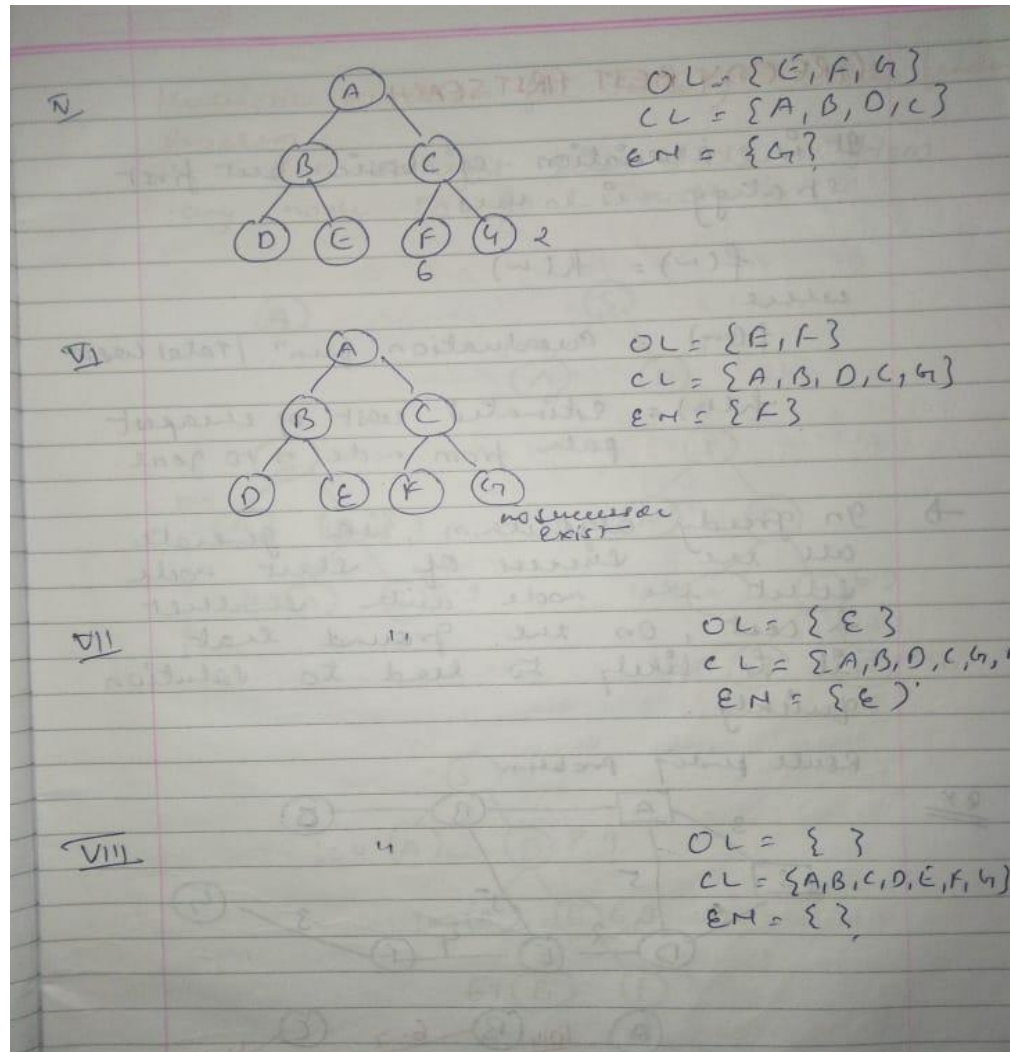
$OL = \{C, D, E\}$
 $CL = \{A, B\}$
 $EM = \{D\}$

IV



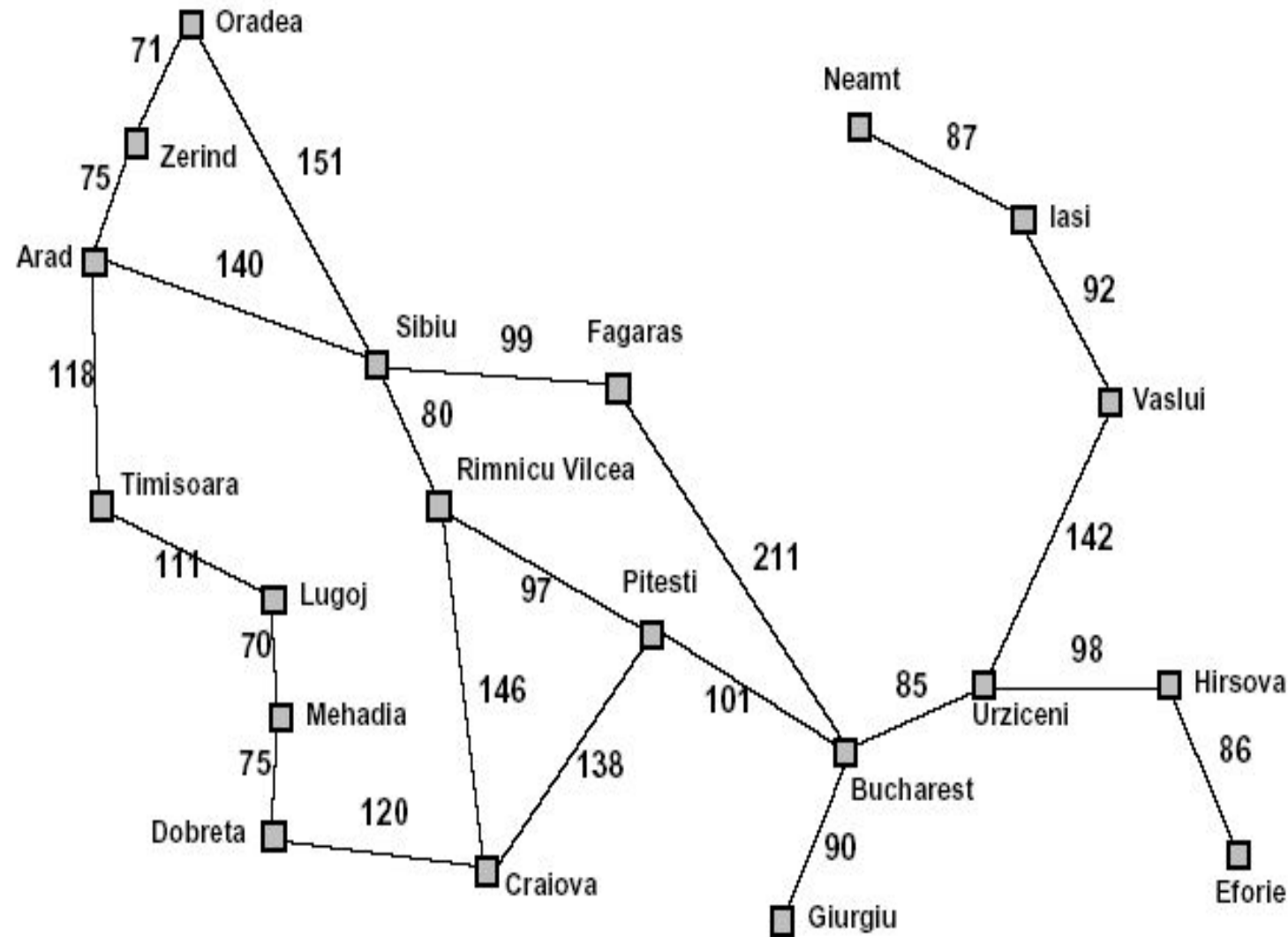
$OL = \{C, E\}$
 $CL = \{A, B, D\}$
 $EM = \{C\}$

example best first search



Heuristic Search : Heuristic Function

- Heuristic F_n , $f(n) = g(n)$ = distance between cities



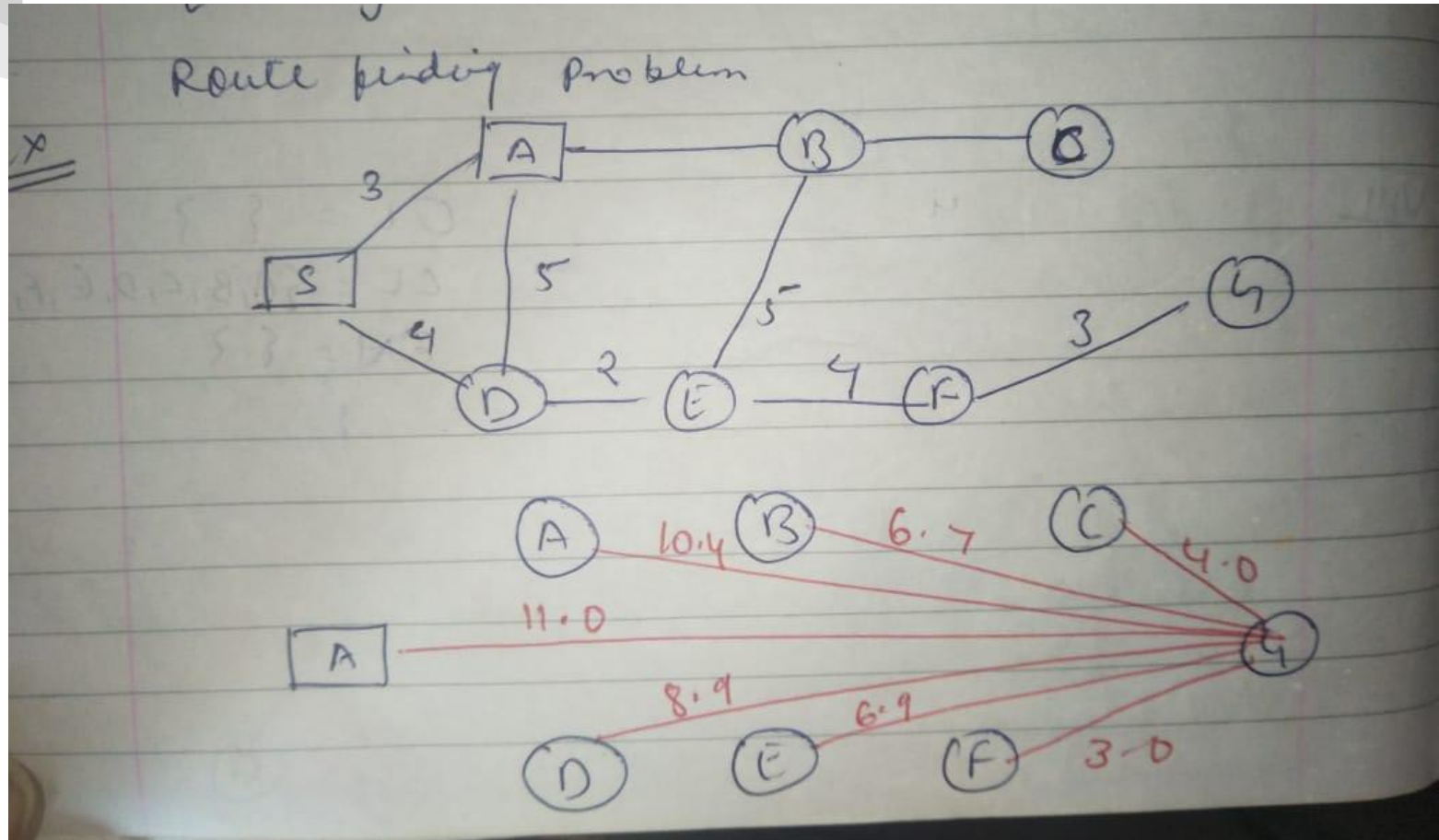
Straight-line distance
to Bucharest

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

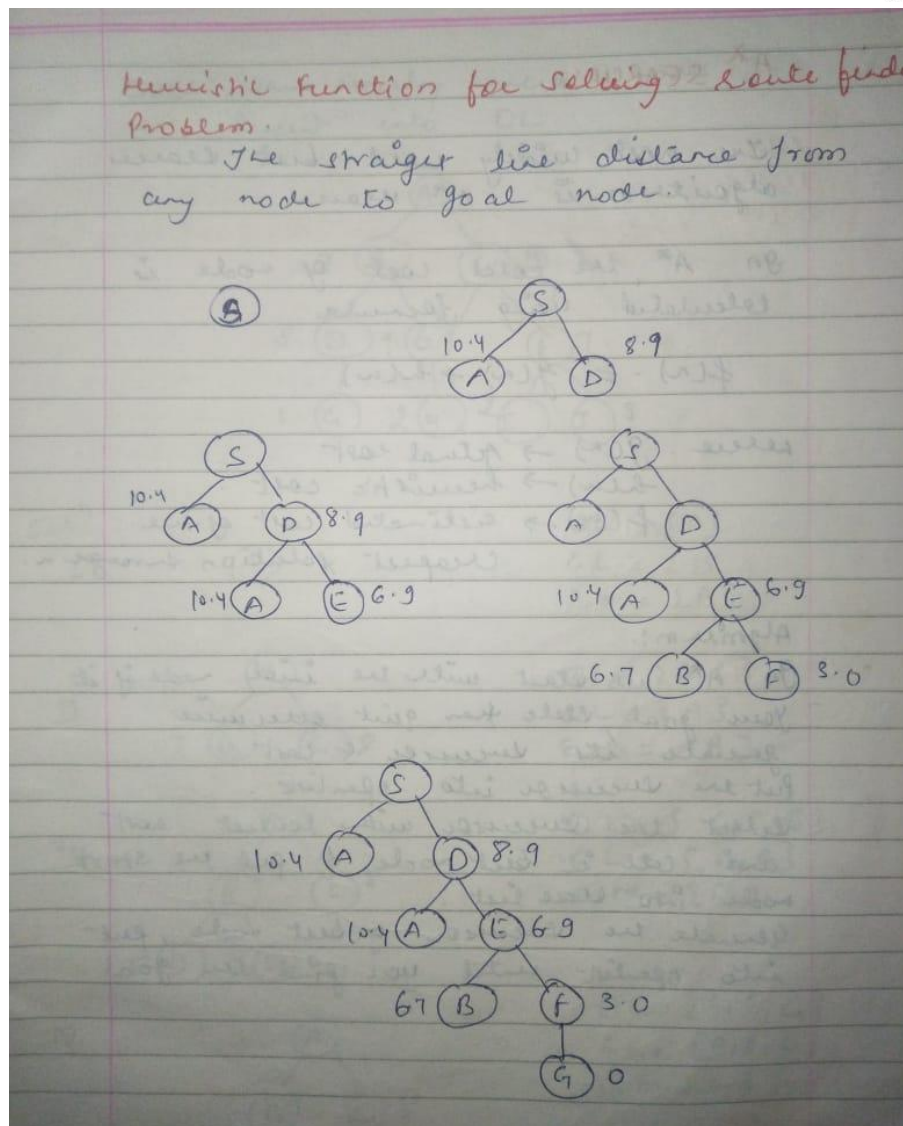
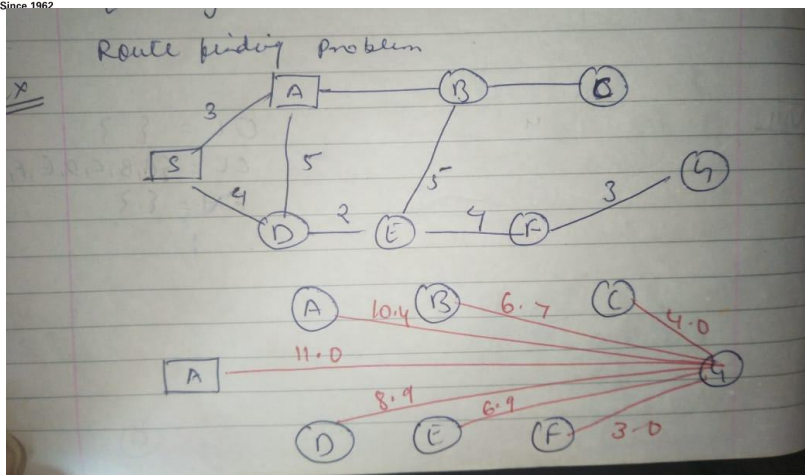
Heuristic Search : Greedy-Best Search

- Tries to expand the node that is closest to the goal.
- Evaluates using only heuristic function : $f(n) = h(n)$
- Possibly lead to the solution very fast.
- Problem ? ~ can end up in sub-optimal solutions (doesn't take notice of the distance it travels).
-

Example: Greedy best first Search

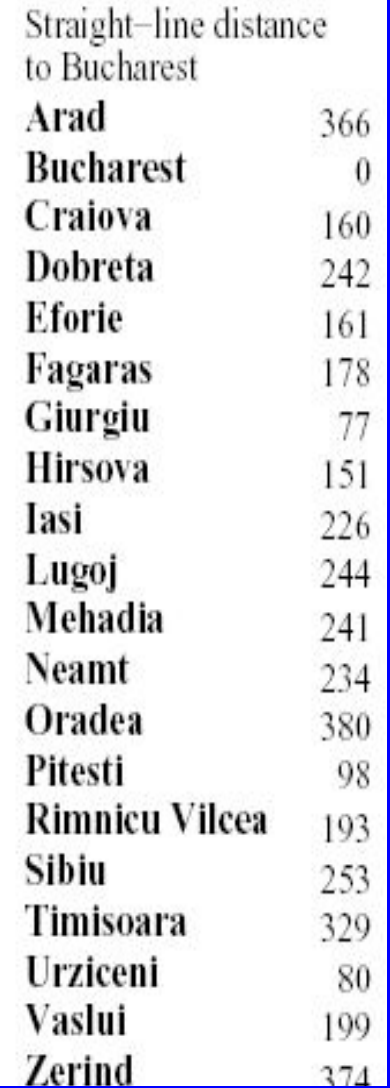


Example: Greedy best first Search



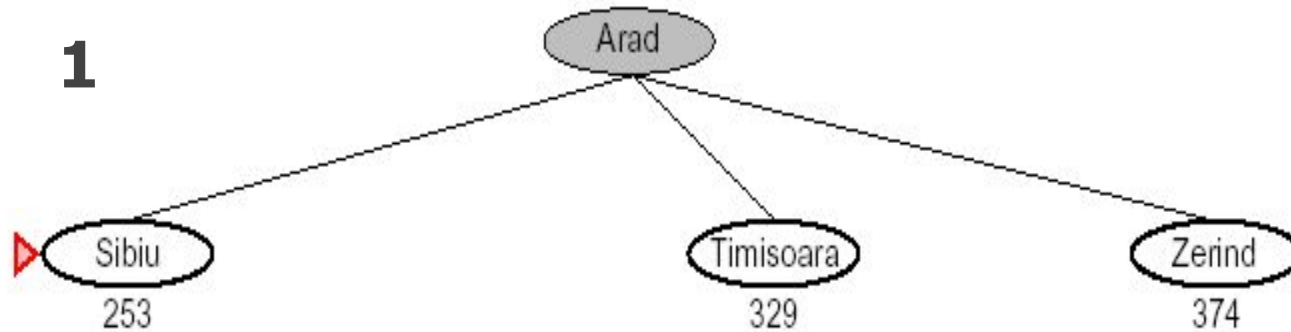


- Distance: heuristic function can be straight line distance (SLD)

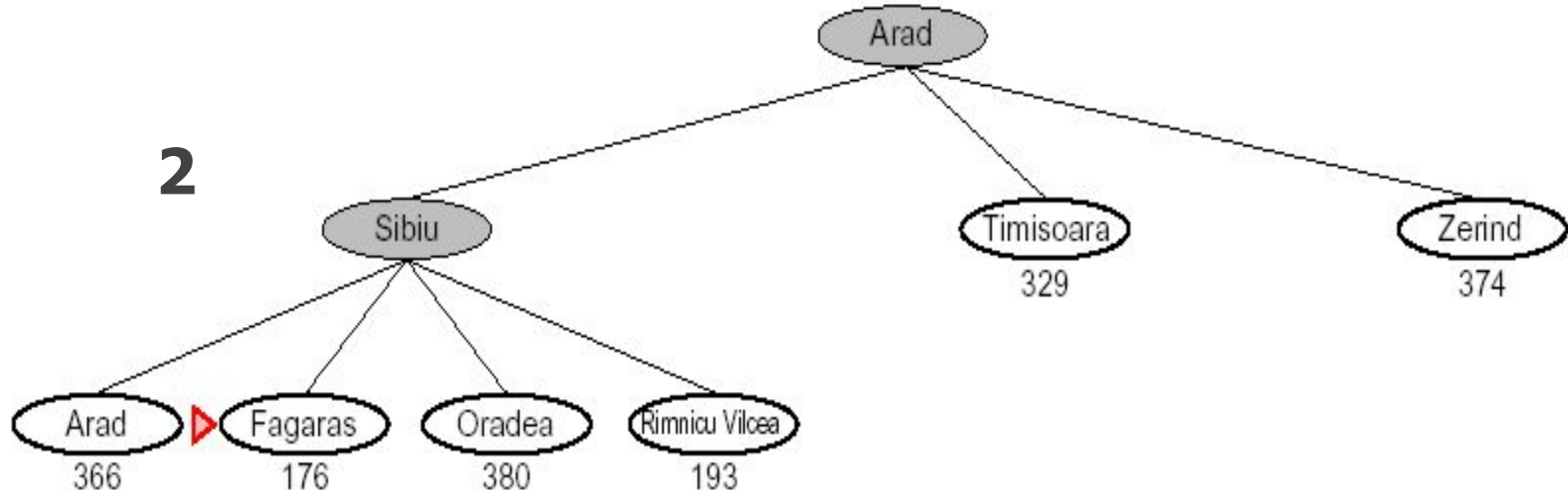


Heuristic Search : Greedy-Best Search

1

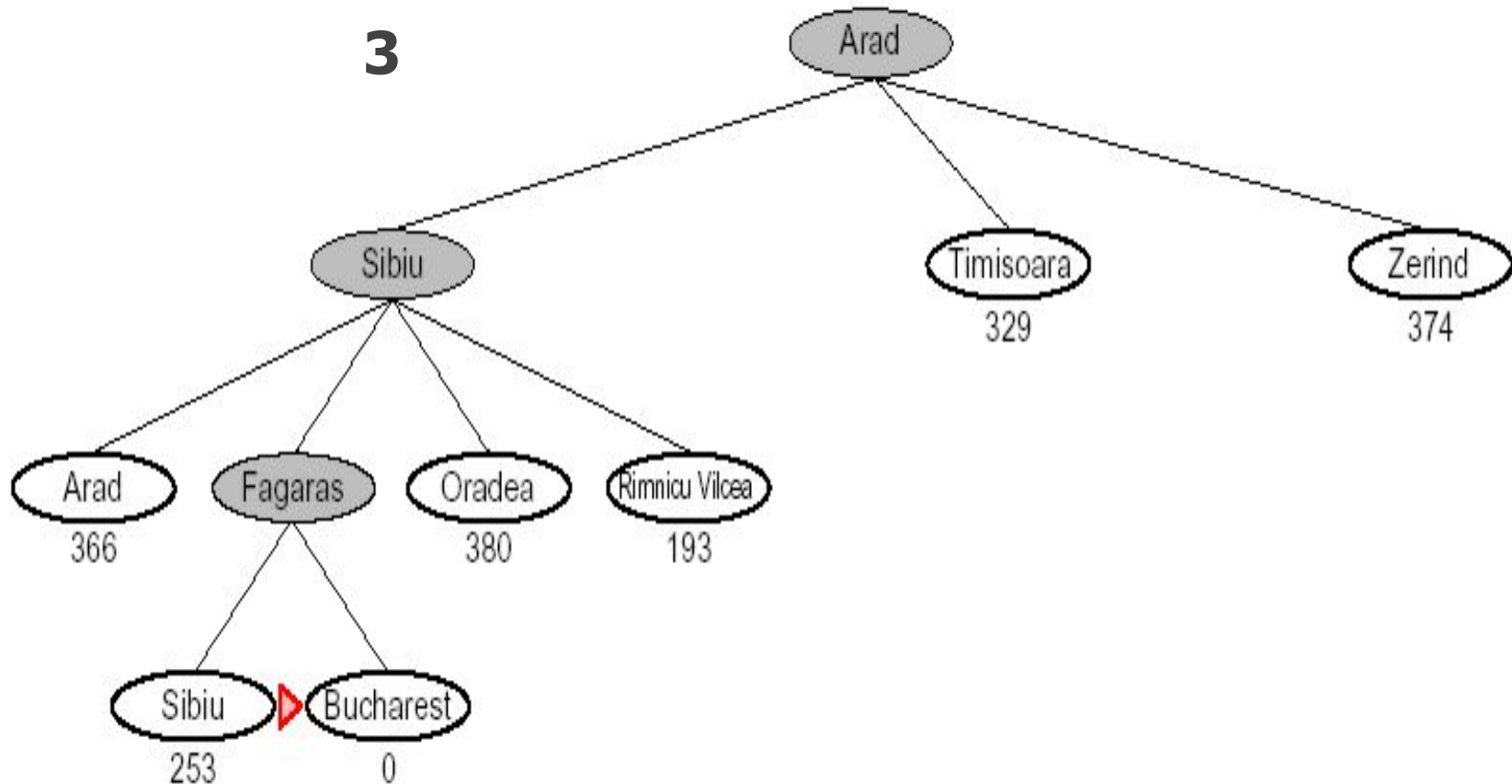


2



Heuristic Search : Greedy-Best Search

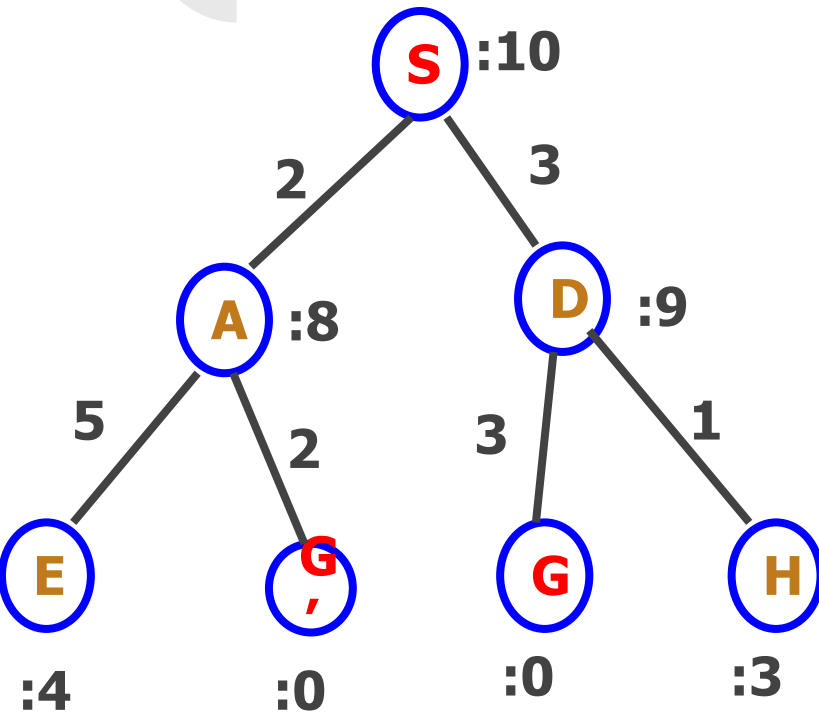
3



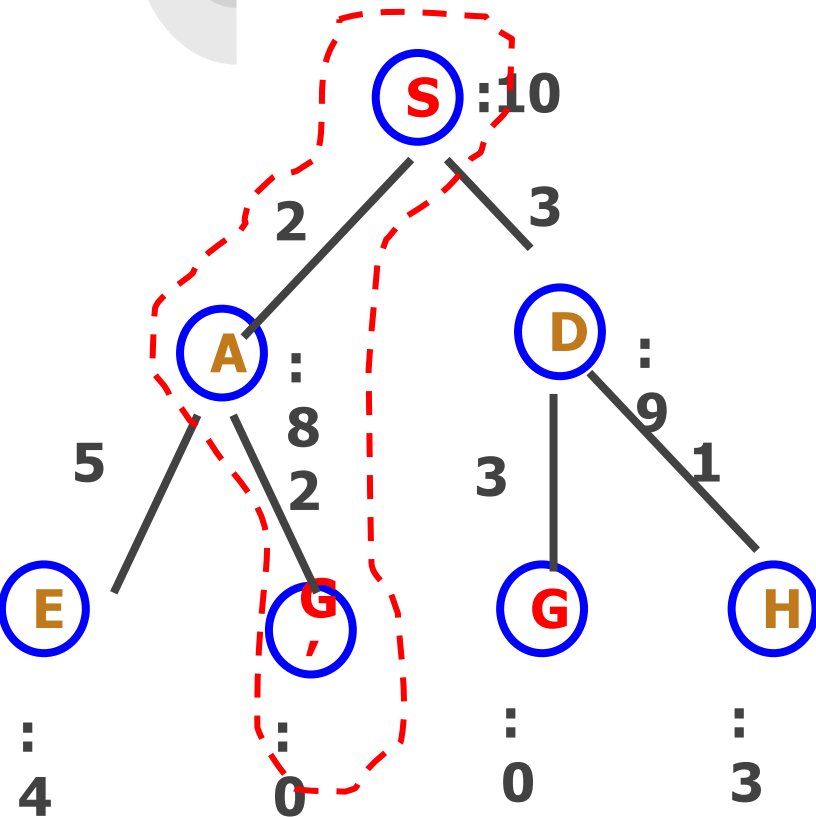
Heuristic Search : A* Algorithm

- Widely known algorithm – (pronounced as “A star” search).
- Evaluates nodes by combining $g(n)$ “cost to reach the node” and $h(n)$ “cost to get to the goal”
- $f(n) = g(n) + h(n)$,
- $f(n)$ □ estimated cost of the cheapest solution.
- Complete and optimal ~ since evaluates all paths.
- Time ? ~ a bit time consuming
- Space ? ~ lot of it!

Heuristic Search : A* Algorithm



Heuristic Search : A* Algorithm



* Path S-A-G is chosen
= Lowest cost

Path cost for S-D-G

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

$$= 0 + 10 \square 10$$

$$f(D) = (0+3) + 9 \square 12$$

$$f(G) = (0+3+3) + 0 \square 6$$

$$\text{Total path cost} = f(S) + f(D) + f(G) \square 28$$

Path cost for S-A-G'

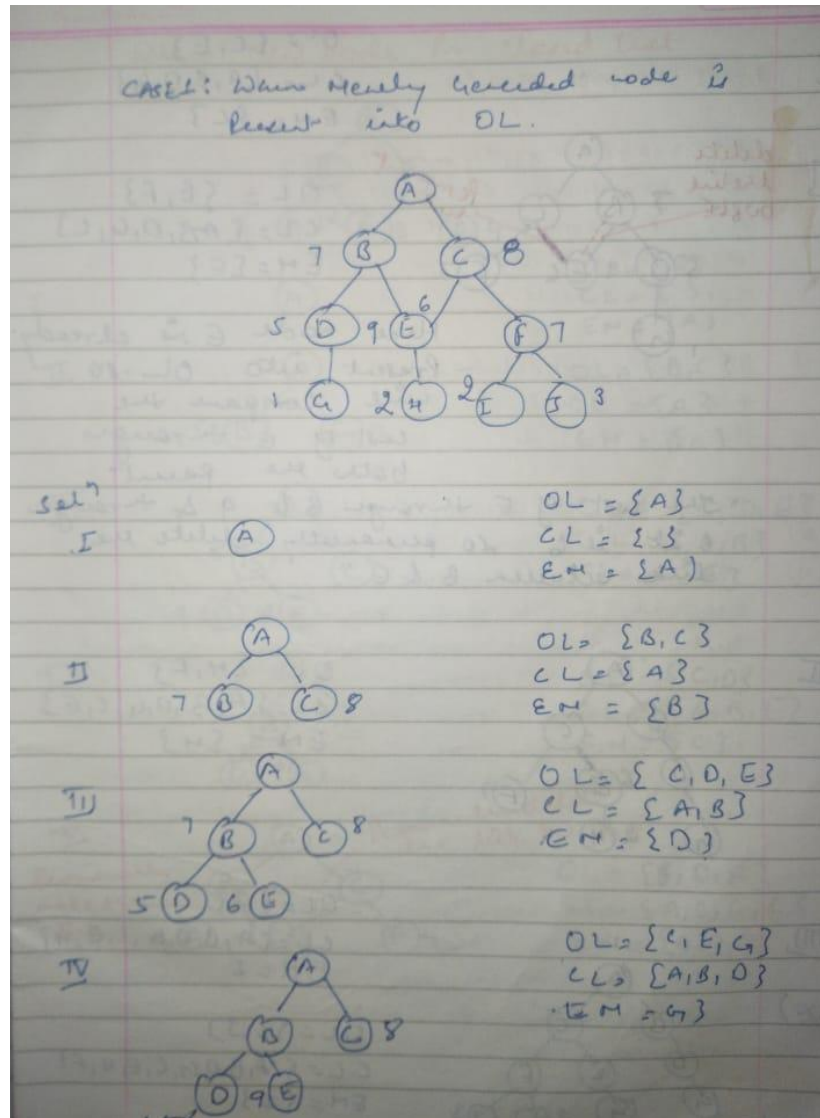
$$f(S) = 0 + 10 \square 10$$

$$f(A) = (0+2) + 8 \square 10$$

$$f(G') = (0+2+2) + 0 \square 4$$

$$\text{Total path cost} = f(S) + f(A) + f(G') \square 24$$

Example: A* Search (new node is in open list)



Example: A* Search (new node is in open list)

V

VI

delete the link
bugle

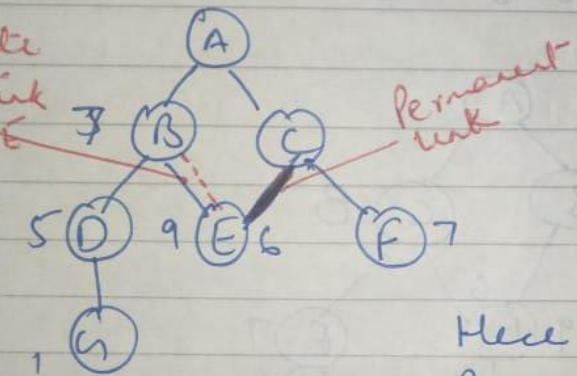
Permanent link

$D^L = \{C, E\}$
 $CL = [A, B, D, G]$
 $EM = \{C\}$

$OL = \{E, F\}$
 $CL = [A, B, D, G, C]$
 $EM = \{E\}$

Here node E is already present into OL, so will compare the cost of E through both the parent

"The cost of E through B is 9 & through C it is 6, so permanently delete the link between B & E"

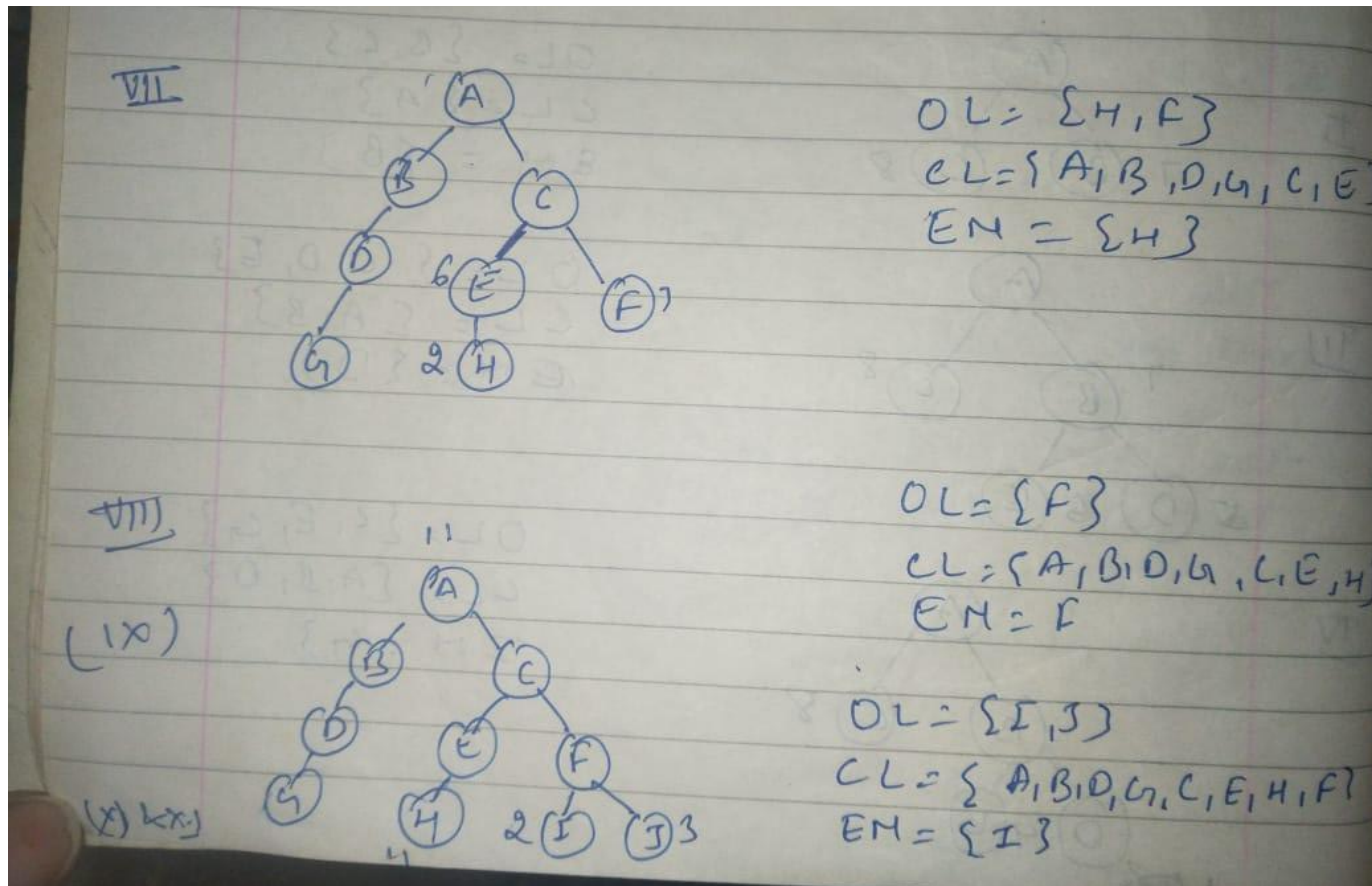


```

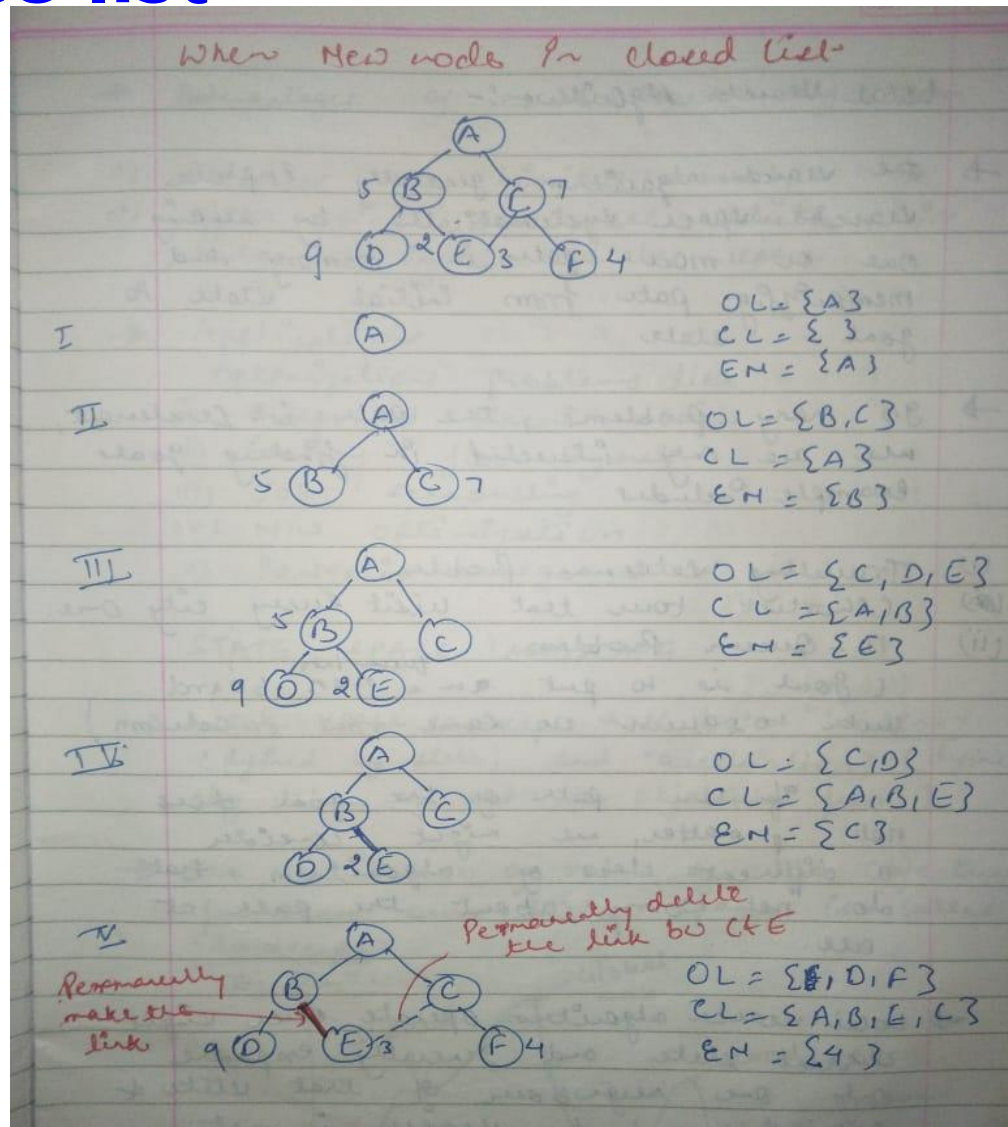
graph TD
    A((A)) --- B((B))
    A --- C((C))
    B --- D((D))
    B --- E((E))
    C --- F((F))
    D --- G((G))
    E -.- B
    E -.- C
    style E stroke-dasharray: 5 5
    style B stroke-dasharray: 5 5
    style C stroke-dasharray: 5 5
    
```

Costs: D=5, E=9, F=7, G=1

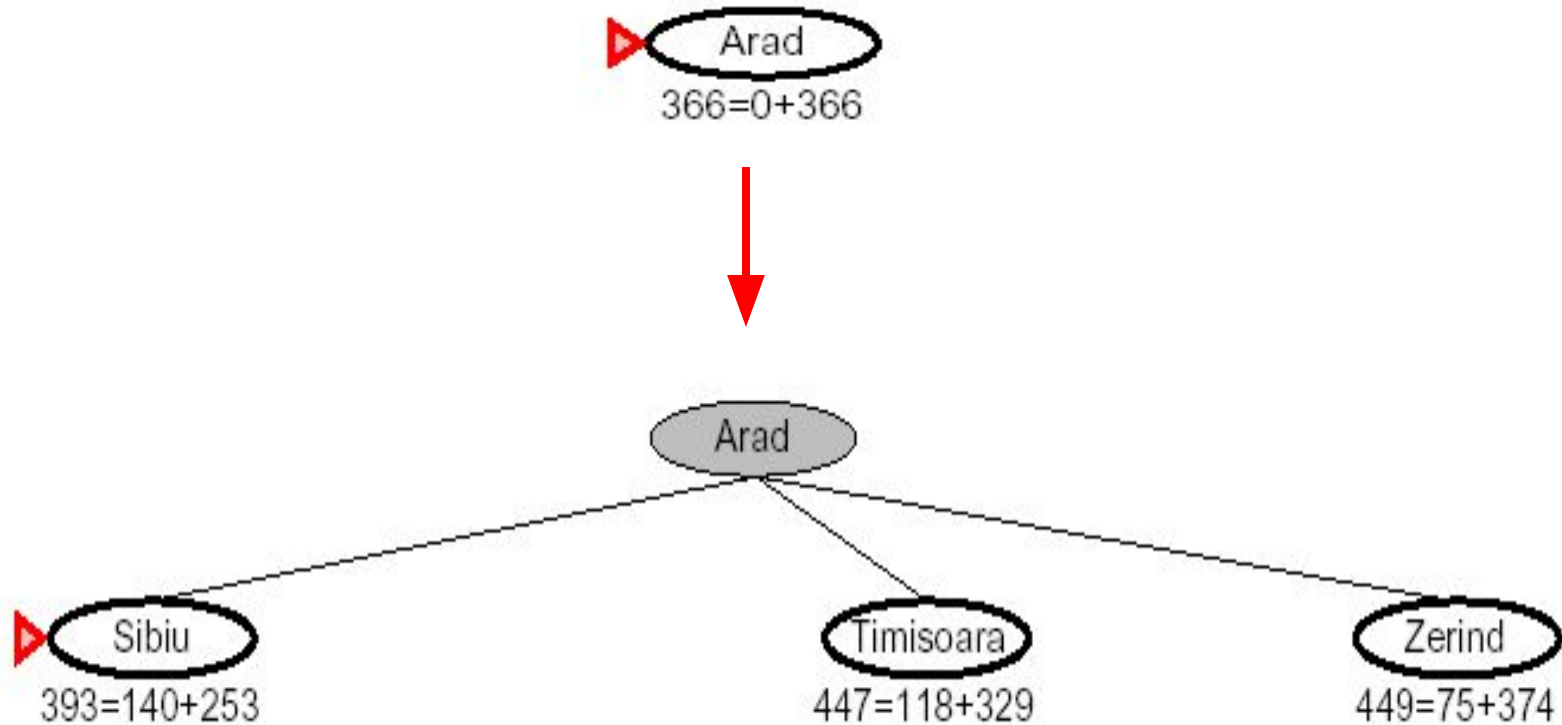
Example: A* Search (new node is in open list)



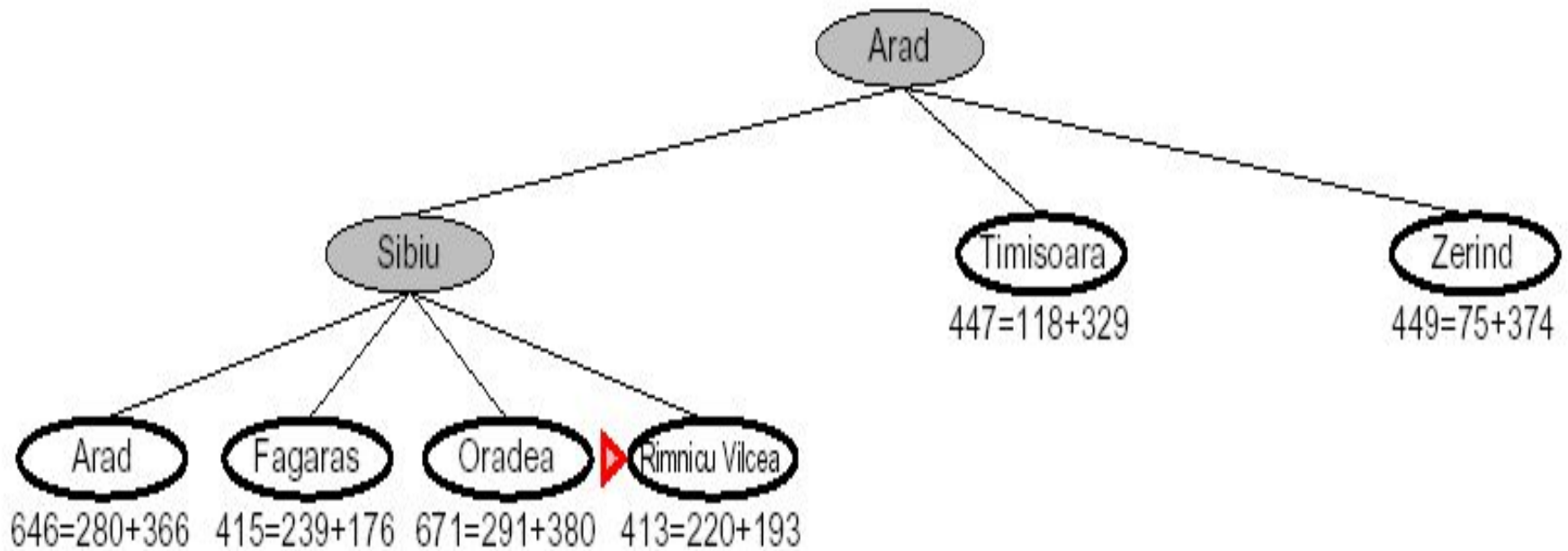
Example: A* Search (new node is in closed list)



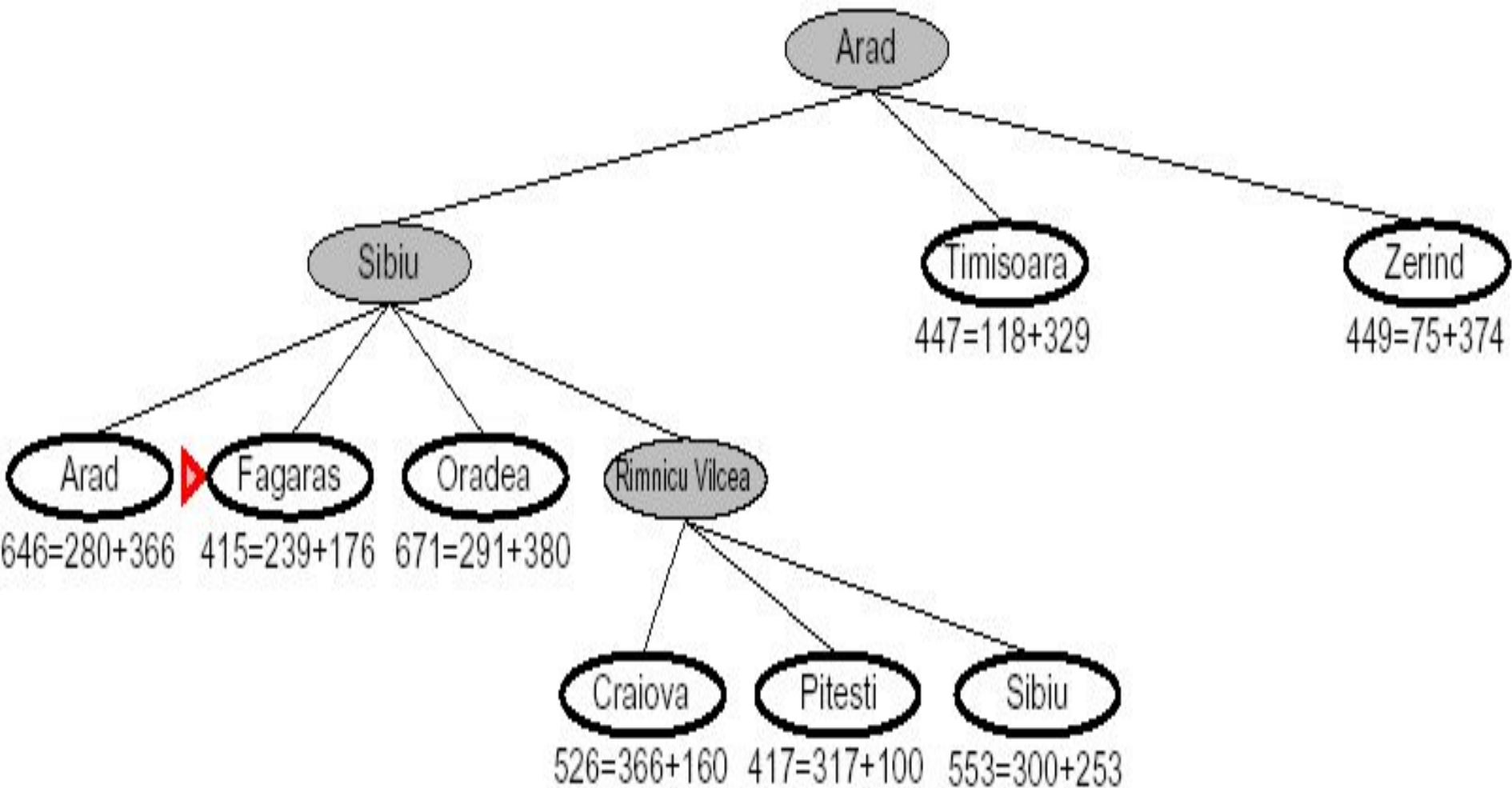
Heuristic Search : A* Algorithm



Heuristic Search : A* Algorithm

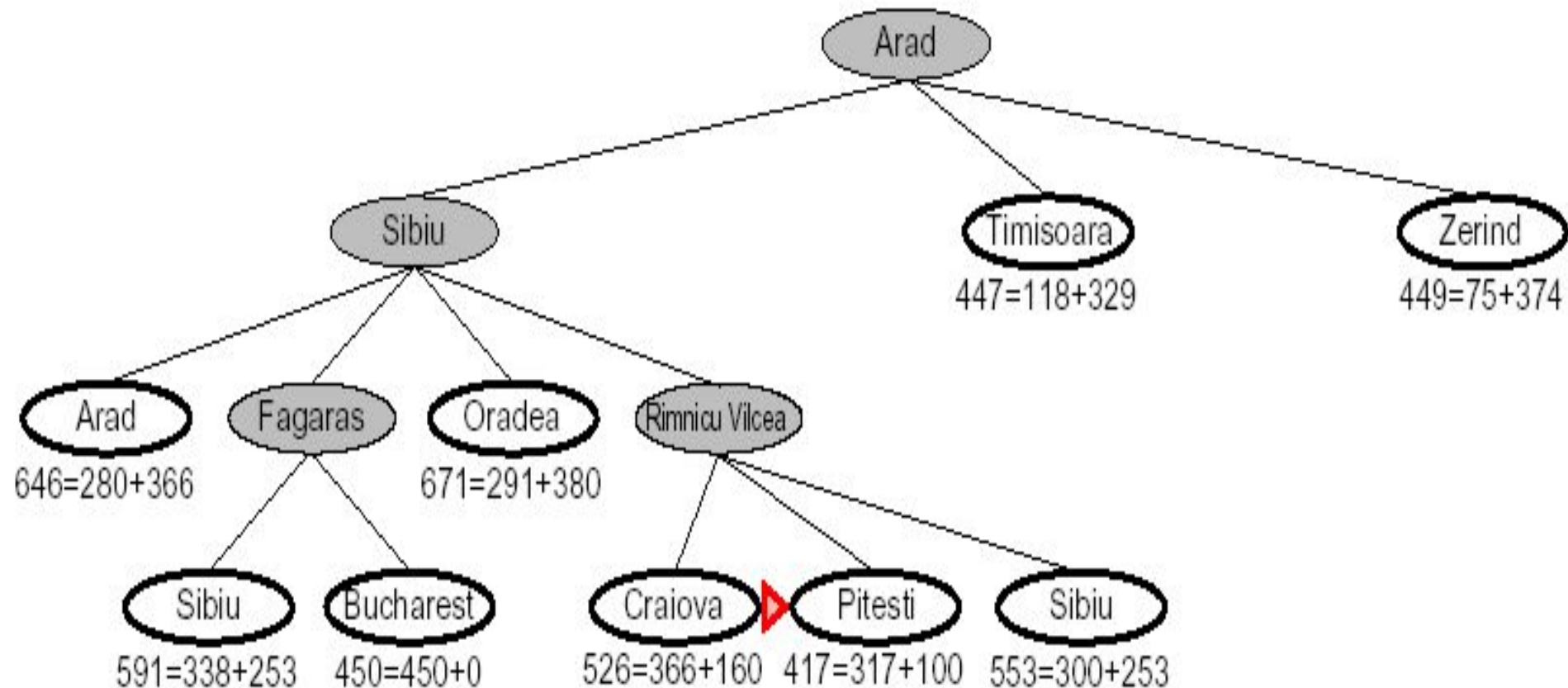


Heuristic Search : A* Algorithm



Heuristic Search : A*

Algorithm





References

- <https://slideplayer.com/slide/9787248/>
- <https://www.slideshare.net/RenasRekany/ai-local-search>
- <https://slideplayer.com/slide/8514257/>
-