Solving problems by searching

CHAPTER 3

In which we see how an agent can find a sequence of actions that achieves its goals when no single action will do.

1. Well-defined problems and solutions

A problem can be defined formally by five components:

- INITIAL STATE: The initial state that the agent starts in_ For example, the initial state for our agent in Romania might be described as In(Arad).

- ACTIONS:A description of the possible actions available to the agent Given a particular state s, ACTIONS(s)returns the set of actions that can be executed in s.

- TRANSITION MODEL : A description of what each action does; the formal name for this is the transition model.

- GOAL TEST : The goal test, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them.

- A path cost function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure

Example: Romania

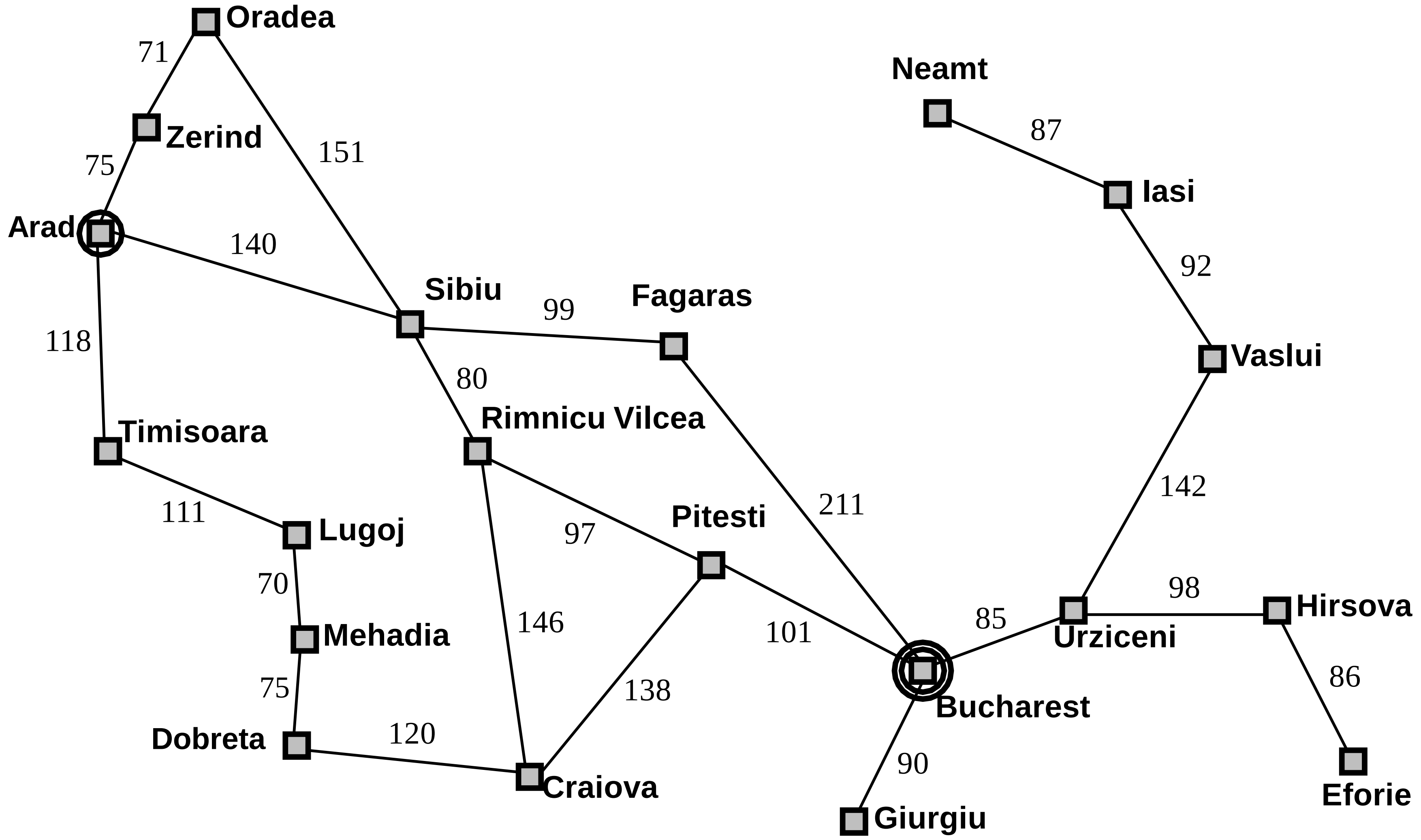
On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest

Formulate goal:
be in Bucharest

Formulate problem: (granularity level) states: various cities
actions: drive between cities

Find solution:
sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



2. Formulating problems

• A formulation of the problem of getting to Bucharest in terms of the initial state, actions, transition model, goal test, and path cost. This formulation seems reasonable, but it is still a model—an abstract mathematical description.

• Compare the simple state description we have chosen, In(Arad), to an actual cross country trip, where the state of the world includes so many things: the traveling companions, the current radio program, the scenery out of the window, the proximity of law enforcement officers, the distance to the next rest stop, the condition of the road, the weather, and so on.

• All these considerations are left out of our state descriptions because they are irrelevant to the problem of finding a route to Bucharest. The process of removing detail from a representation is called abstraction

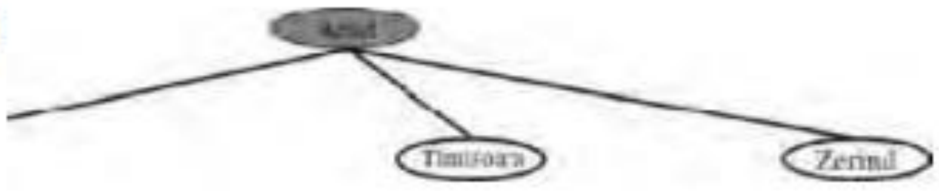
SEARCHING FOR SOLUTIONS

- Having formulated some problems, we now need to solve them. A solution is an action sequence, so search algorithms work by considering various possible action sequences.

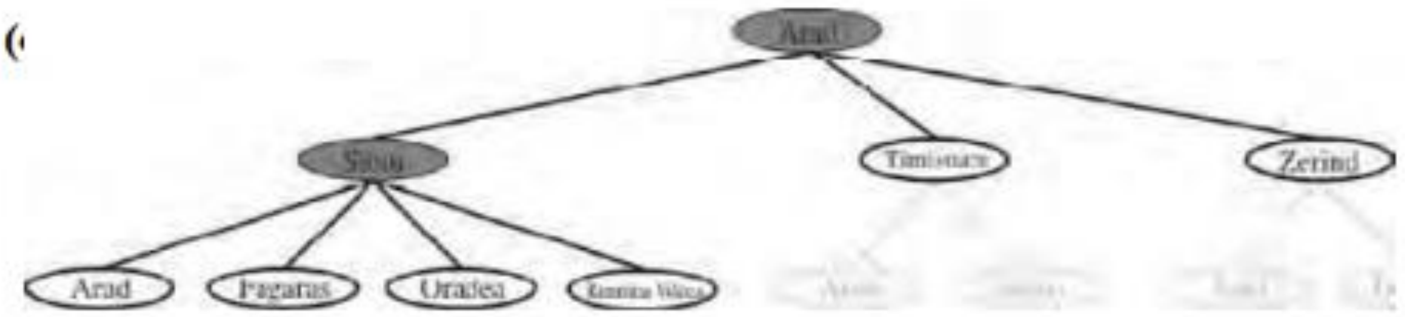
- The possible action sequences starting at the initial state form a search tree with the initial state at the root; the branches are actions and the nodes correspond to states in the state space of the problem

(a) The **initial state**

(b) After expanding Arad

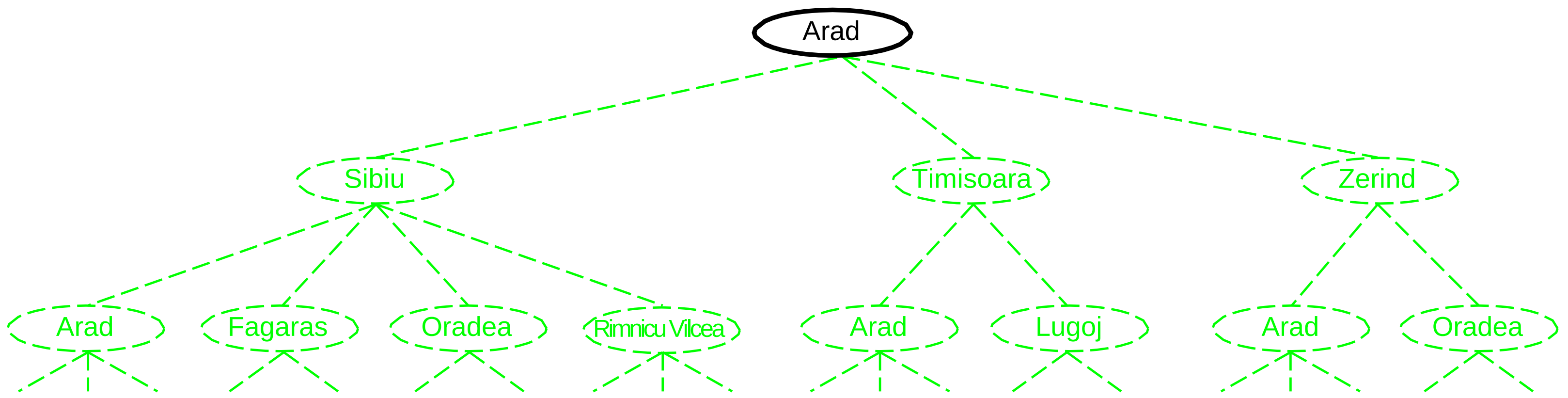


(

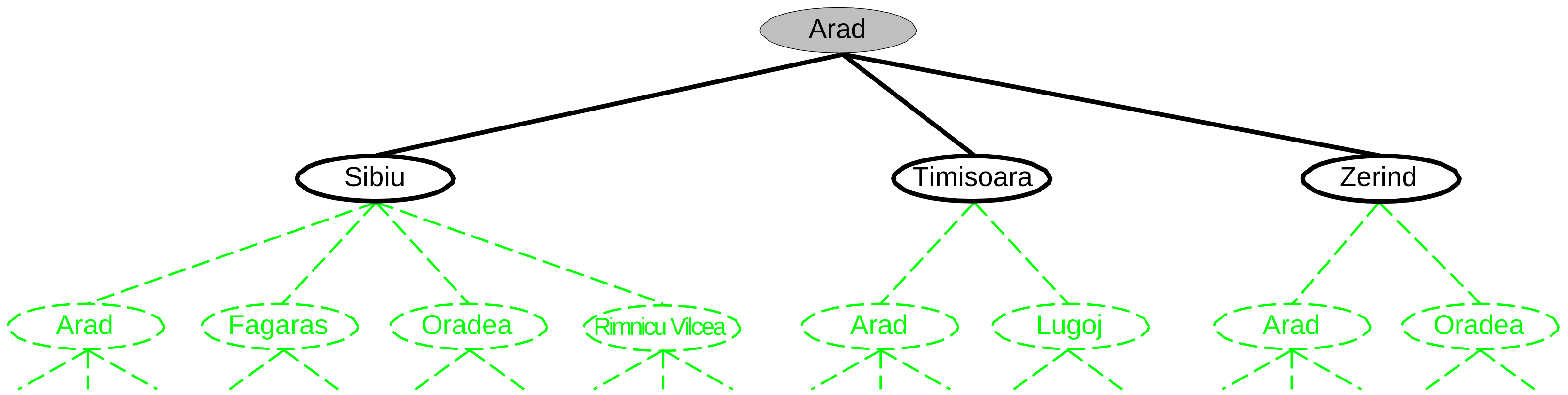


Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

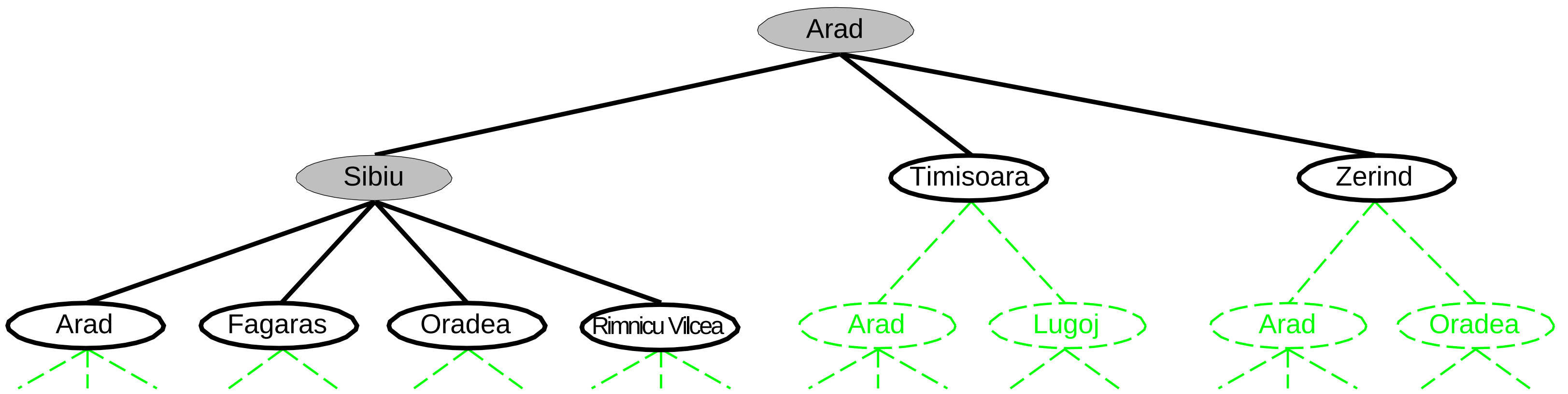
Tree search example



Tree search example



Tree search example



1. Infrastructure for search algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we have a structure that contains four components:

• STATE: the state in the state space to which the node corresponds;

• PARENT: the node in the search tree that generated this node; •

• Action: the action that was applied to the parent to generate the node;

• PATH-COST: the cost, traditionally denoted by g(n),of the path from the initial state to the node, as indicated by the parent pointers

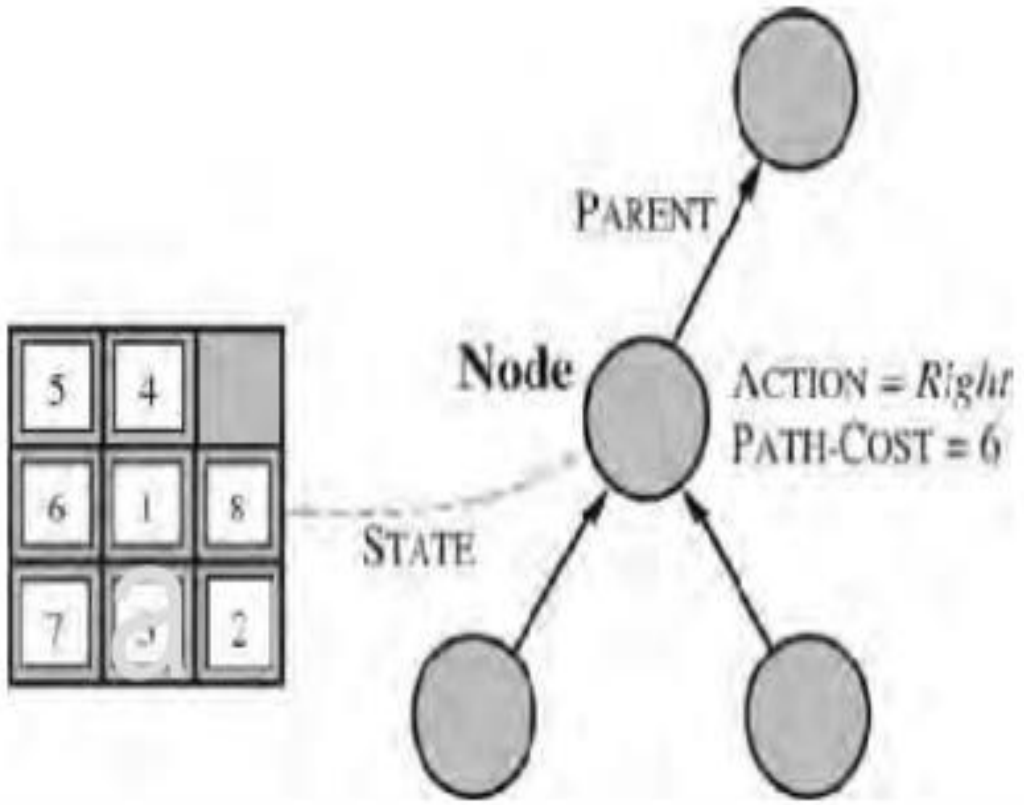


Figure Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

2. Measuring problem-solving performance

Before we get into the design of specific search algorithms, we need to consider the criteria that might be used to choose among them. We can evaluate an algorithm's performance in four ways:

• Completeness: Is the algorithm guaranteed to find a solution when there is one?

• Optimality: Does the strategy find the optimal solution, as defined on page 68?

• Time complexity: How long does it take to find a solution?

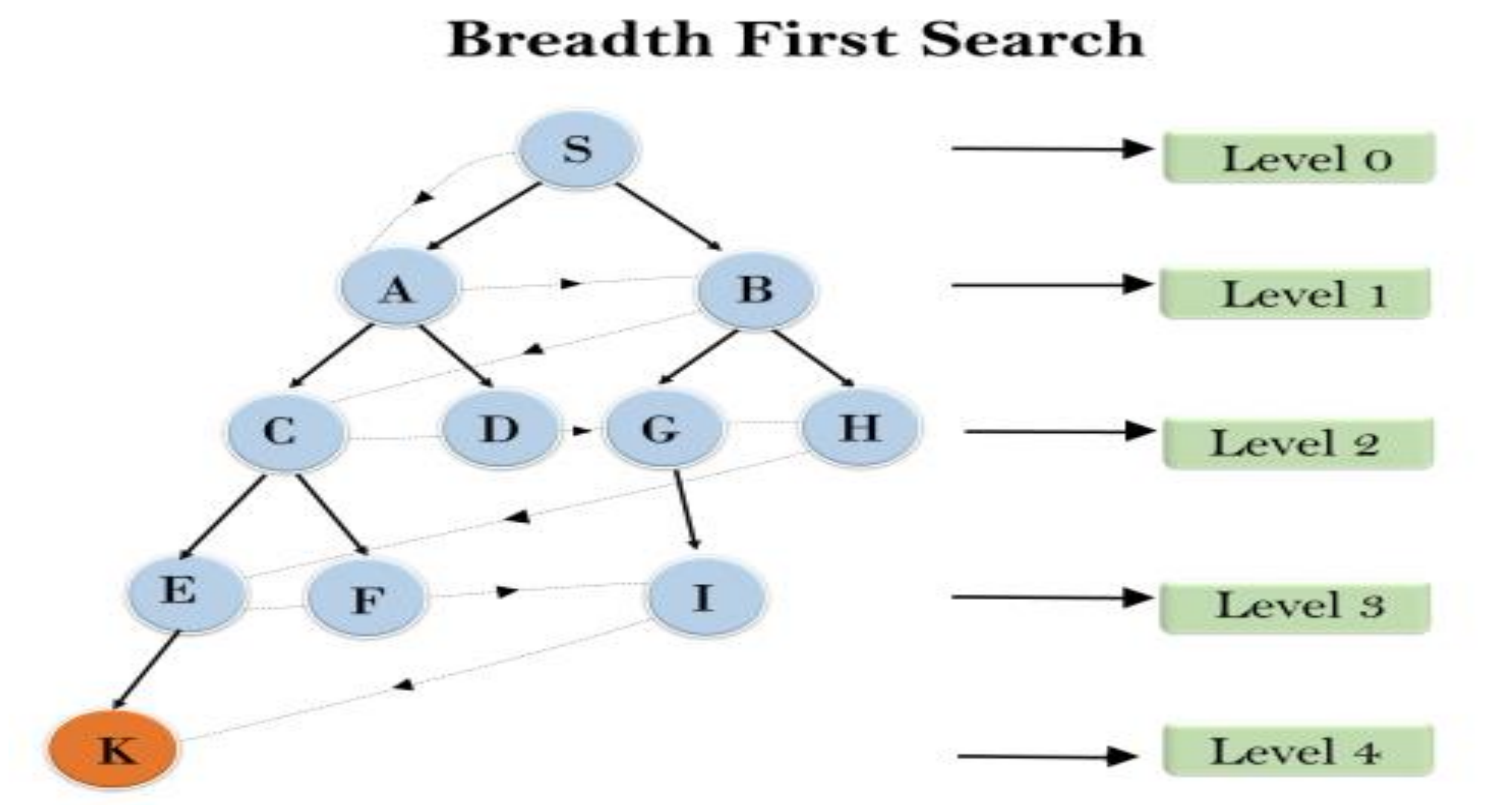
• Space complexity: How much memory is needed to perform the search?

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

Breadth-first search, Uniform-cost search, Depth-first search, Depth-limited search,

Iterative deepening search



Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

T (b) = $1+b^2+b^3+.......+ b^d= O (b^d)$

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

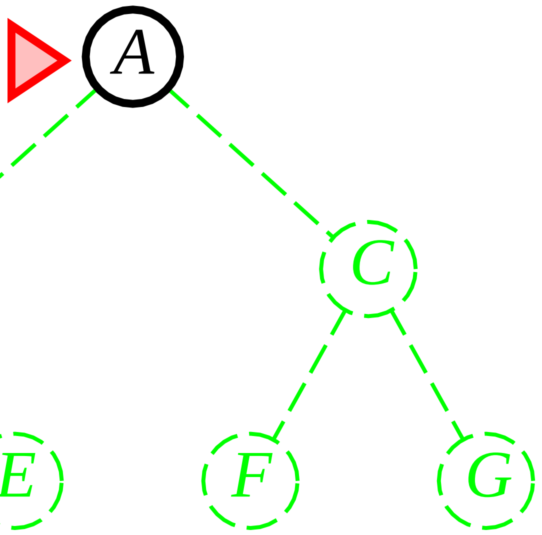
- **Advantages:**
- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.
- **Disadvantages:**
- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

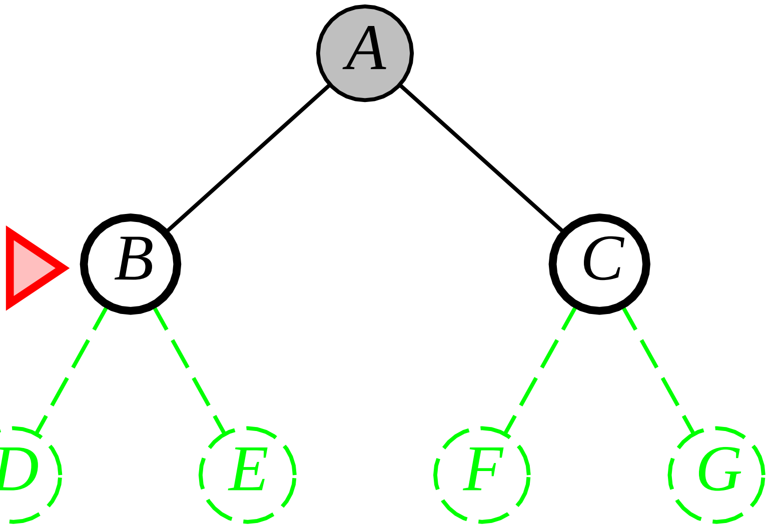


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

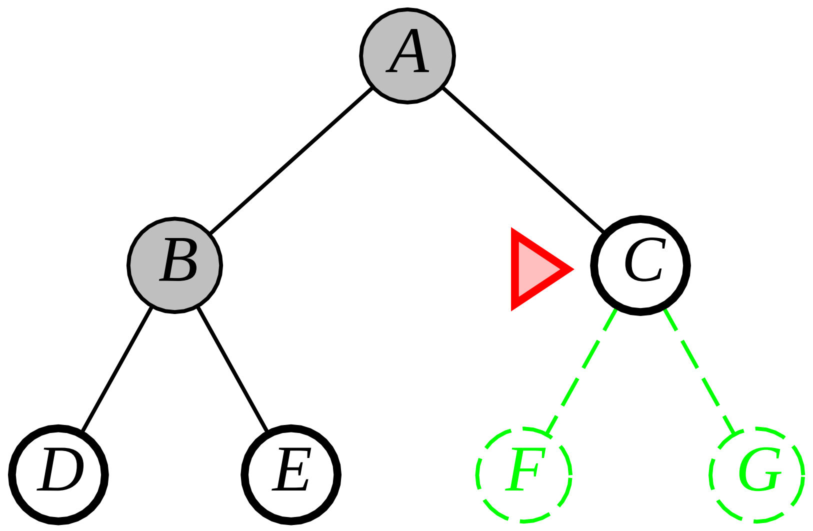


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

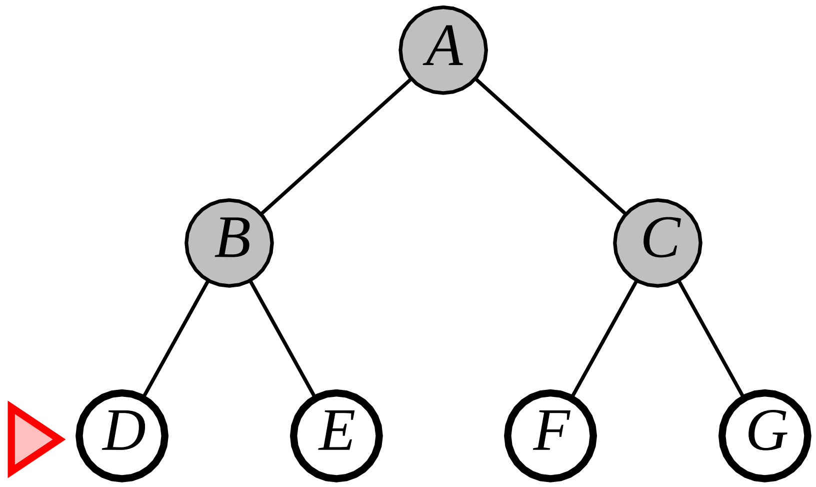


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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



Breadth-first search on a graph

```
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)
```

```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
     STATE = problem.RESULT(parent.STATE, action),
     PARENT = parent, ACTION = action,
     PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

Properties of breadth-first search

Complete??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time??

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., exp. in d

Space??

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., exp. in d

Space?? $O(b^d)$ (keeps every node from the frontier $O(b^d)$ and explored set $O(b^{d-1})$ in memory)

Optimal??

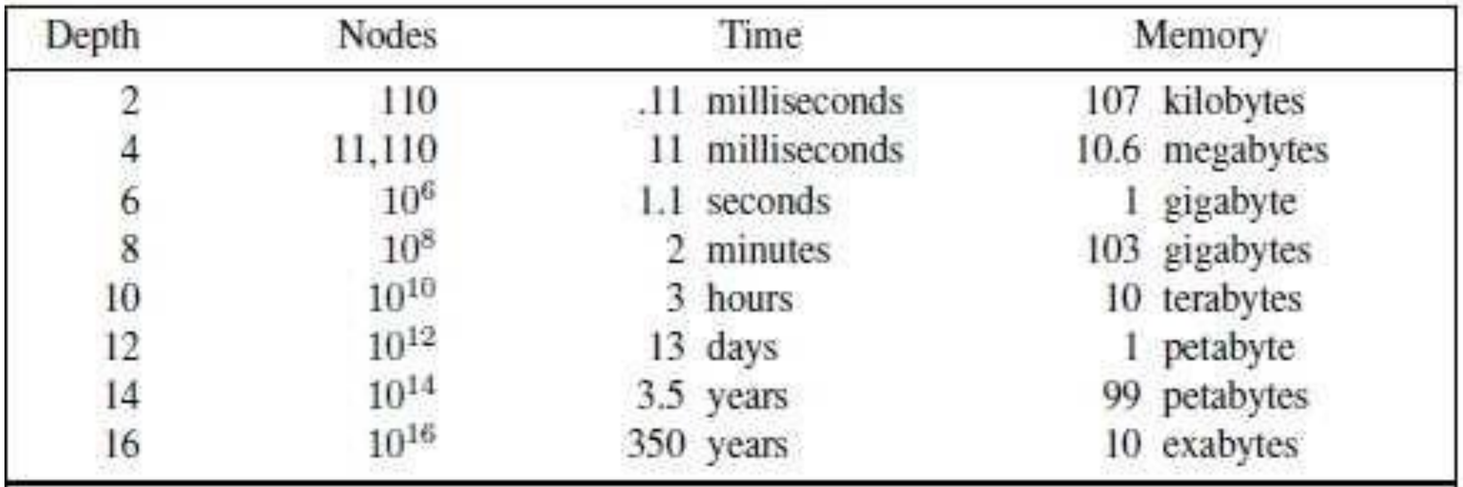
Properties of breadth-first search

Complete?? Yes (if b is finite)

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

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

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

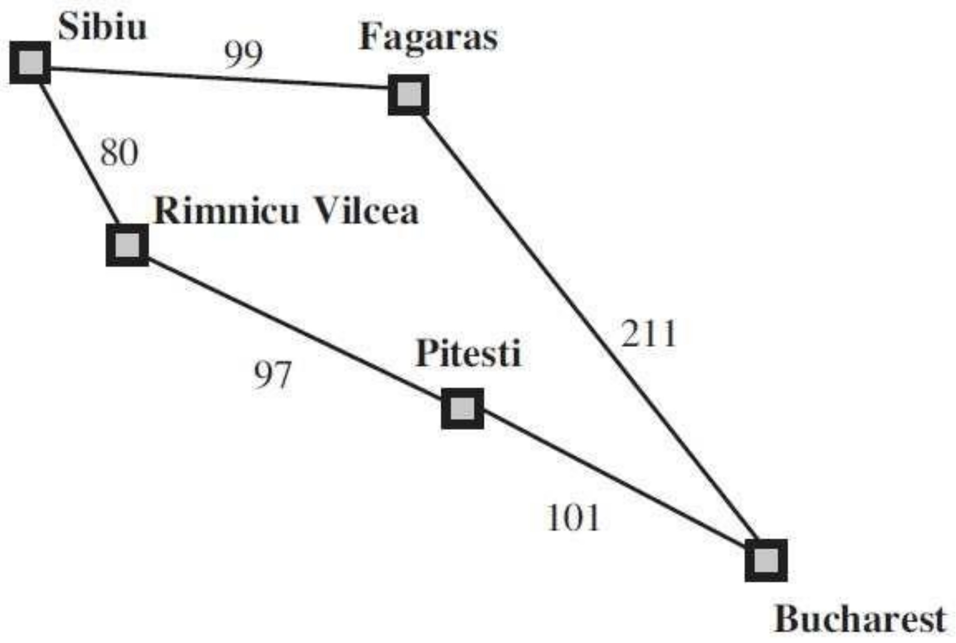


Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

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

Expand the node with the lowest path cost g(n)



Implementation:

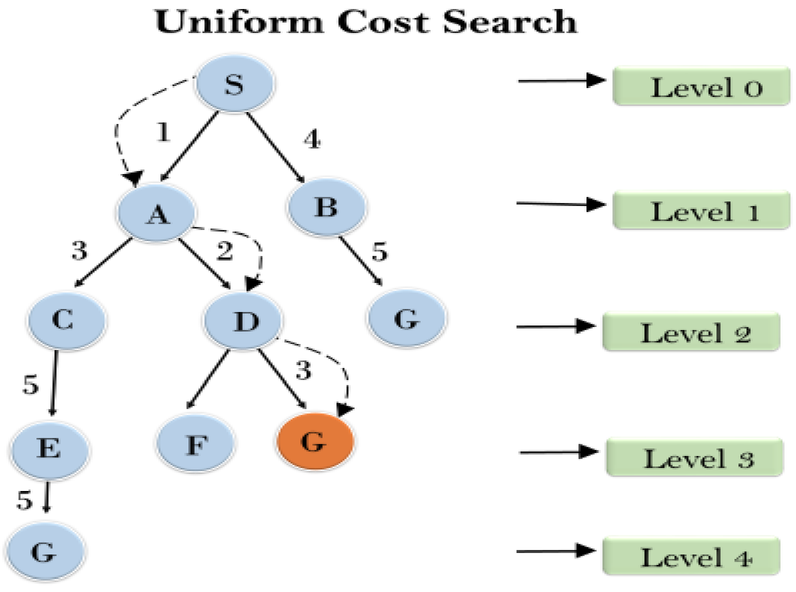
fringe = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

- Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs form the root node.
- A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

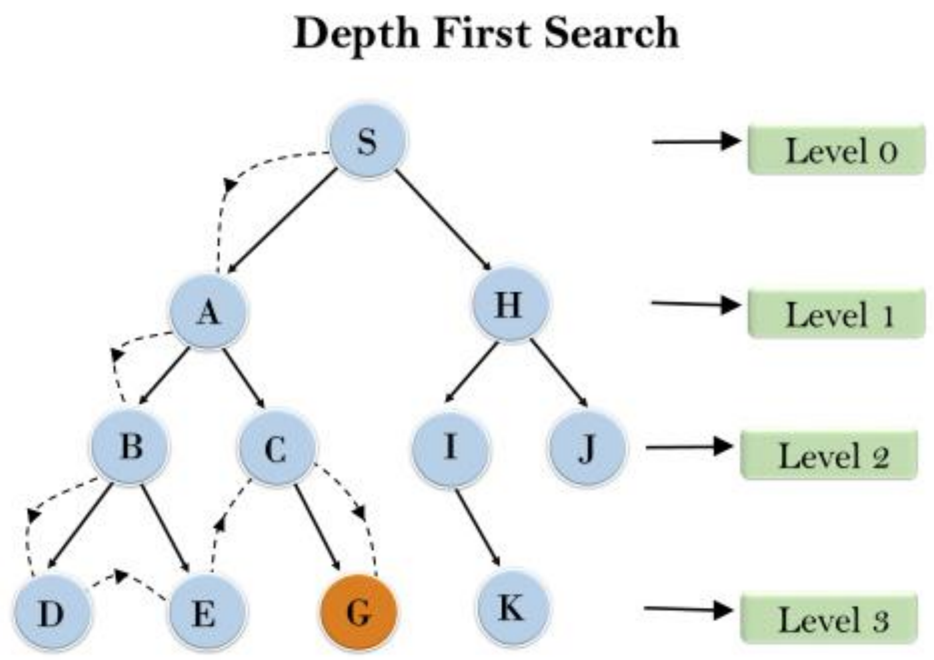
- **Advantages:**
- Uniform cost search is optimal because at every state the path with the least cost is chosen.
- **Disadvantages:**
- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Example:



- **Completeness:**
- Uniform-cost search is complete, such as if there is a solution, UCS will find it.
- **Time Complexity:**
- Let C* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.
- Hence, the worst-case time complexity of Uniform-cost search is$\mathbf{O(b^{1 + [C^*/\varepsilon]})/}$.
- **Space Complexity:**
- The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $\mathbf{O(b^{1 + [C^*/\varepsilon]})}$.

- **Optimal:**
- Uniform-cost search is always optimal as it only selects a path with the lowest path cost.



- **Completeness:** DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.
- **Time Complexity:** Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

- **$T(n)= 1+ n^2+ n^3 +.........+ n^m=O(n^m)$**
- **Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)**
- **Space Complexity:** DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.
- **Optimal:** DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

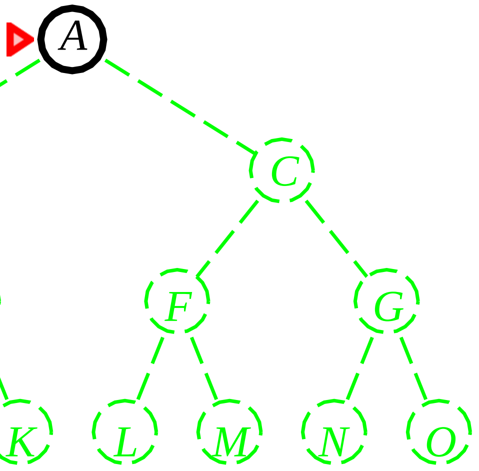
- **Advantage:**
- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).
- **Disadvantage:**
- There is the possibility that many 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.

Depth-first search

Expand deepest unexpanded node

Implementation:

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

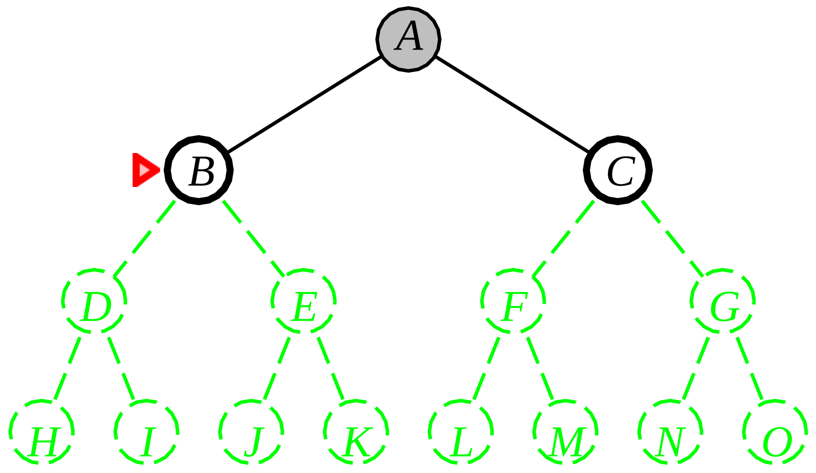


Depth-first search

Expand deepest unexpanded node

Implementation:

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

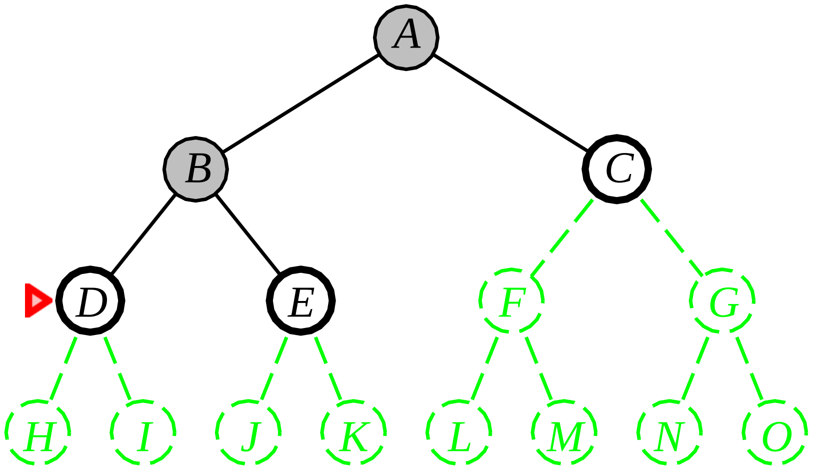


Depth-first search

Expand deepest unexpanded node

Implementation:

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

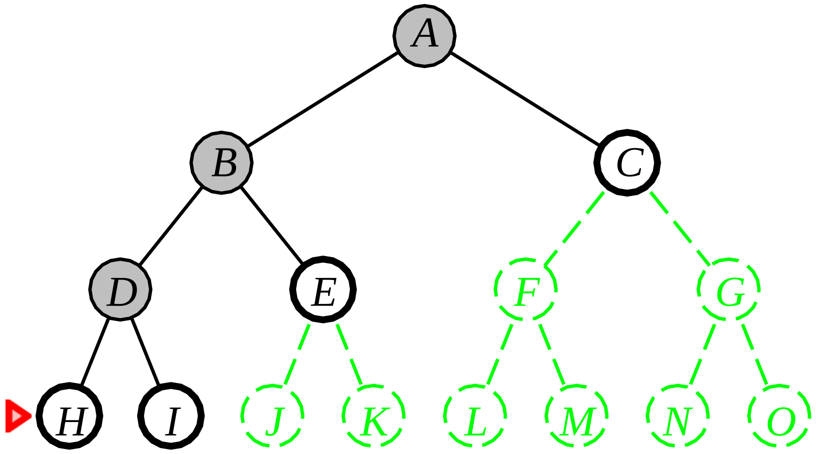


Depth-first search

Expand deepest unexpanded node

Implementation:

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

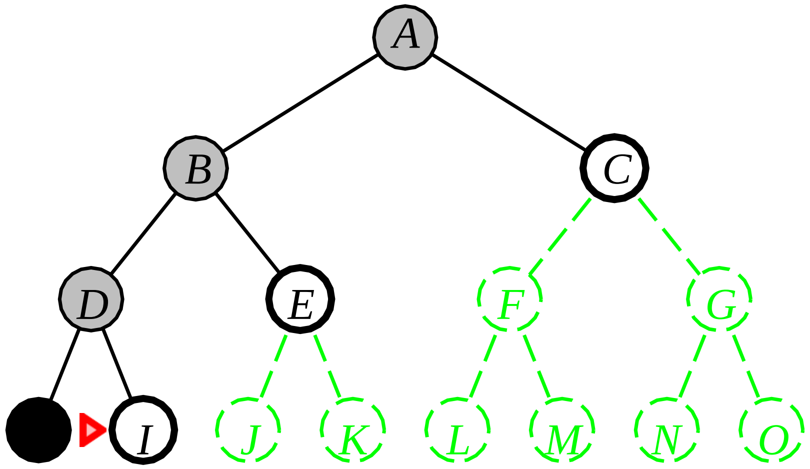


Depth-first search

Expand deepest unexpanded node

Implementation:

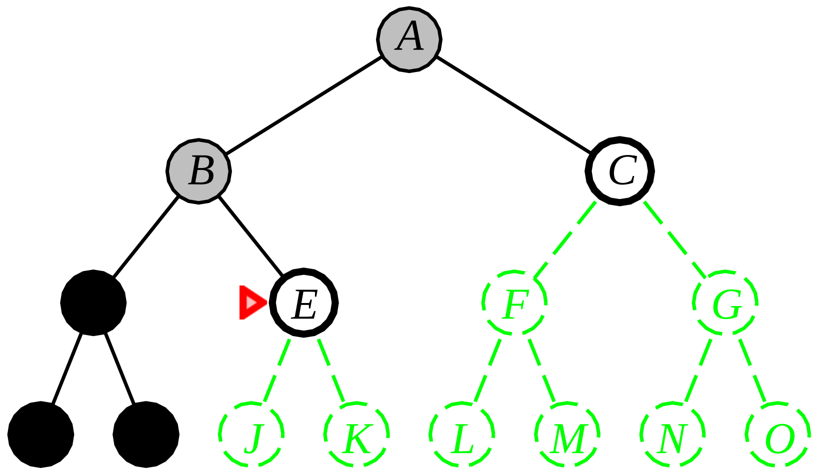
fringe = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

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

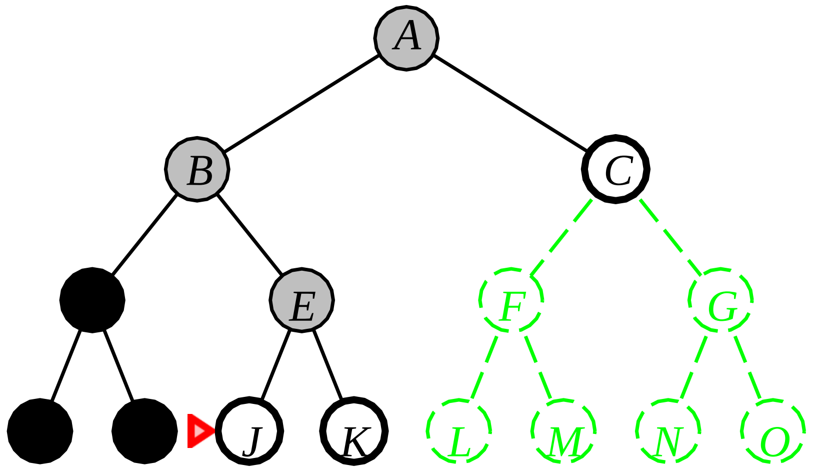


Depth-first search

Expand deepest unexpanded node

Implementation:

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

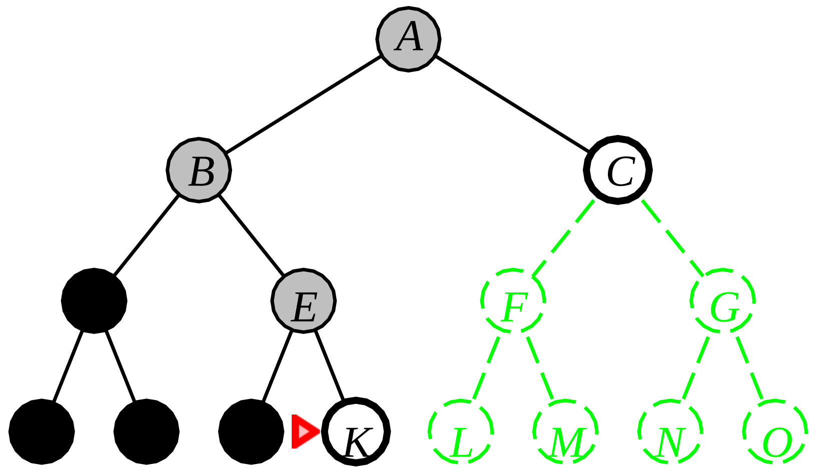


Depth-first search

Expand deepest unexpanded node

Implementation:

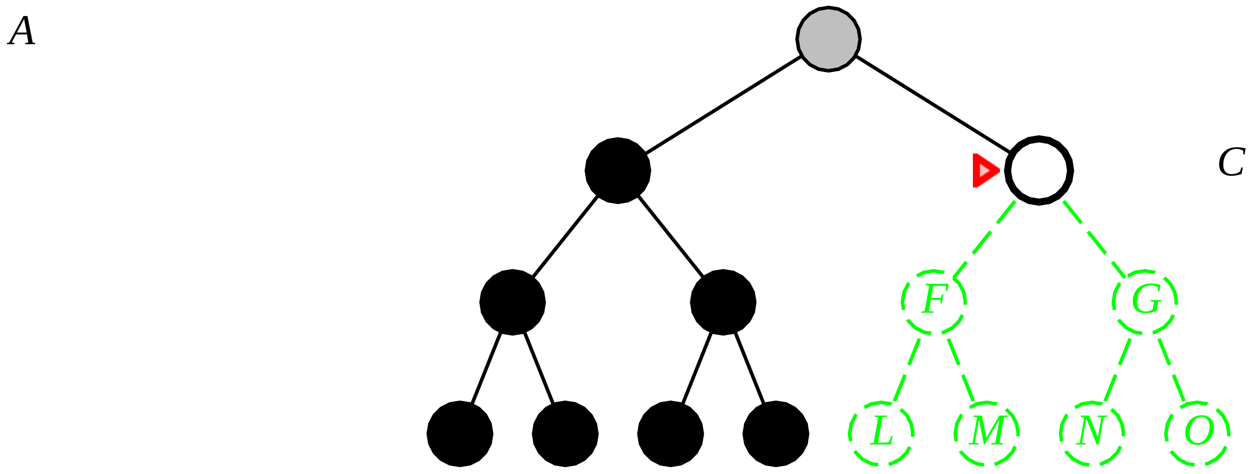
fringe = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

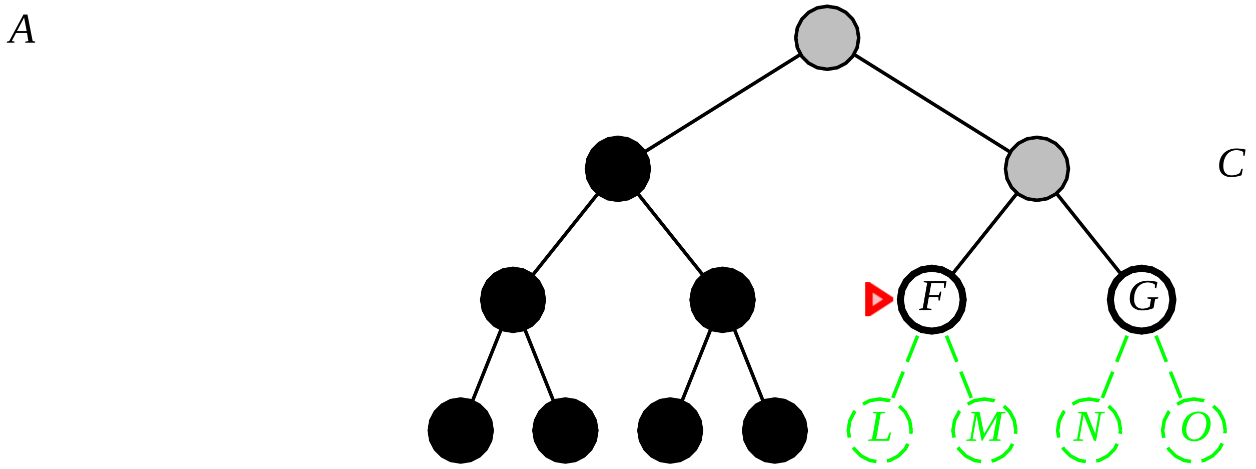
Implementation:
fringe = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

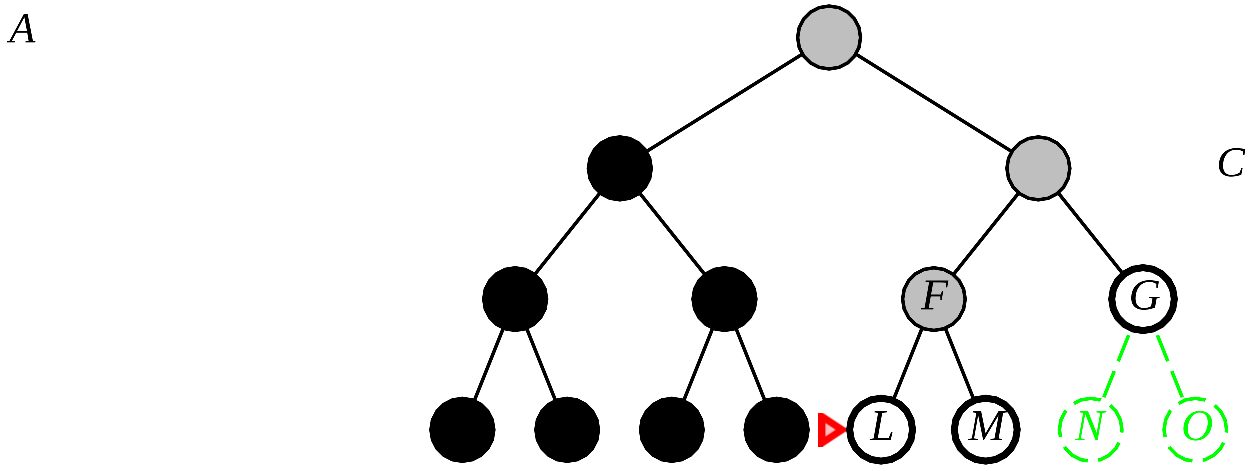
Implementation:
fringe = LIFO queue, i.e., put successors at front



Depth-first search

Expand deepest unexpanded node

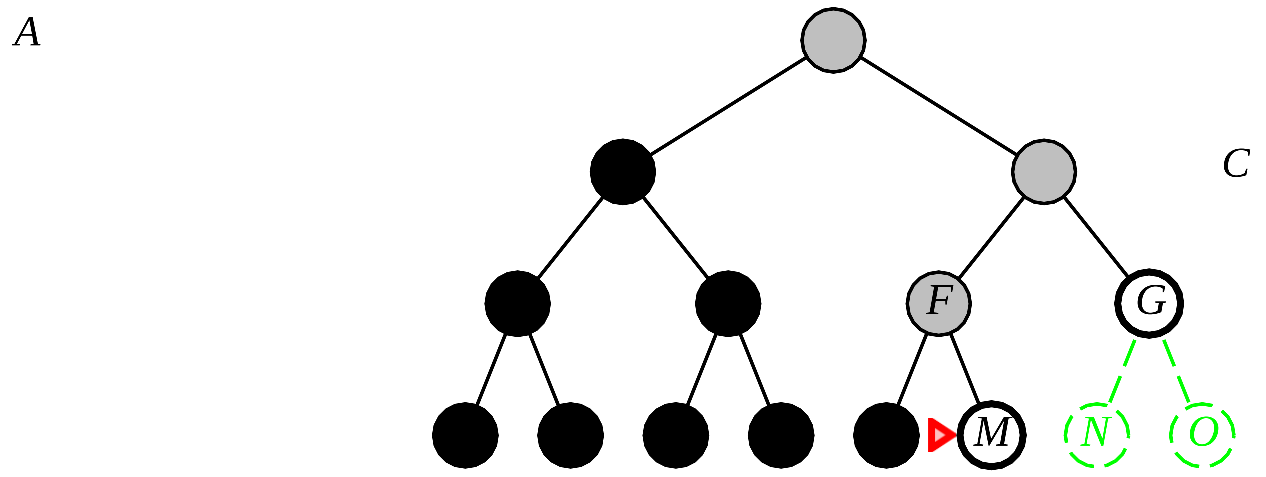
Implementation:
fringe = LIFO queue, i.e., put successors at front



Depth-first search

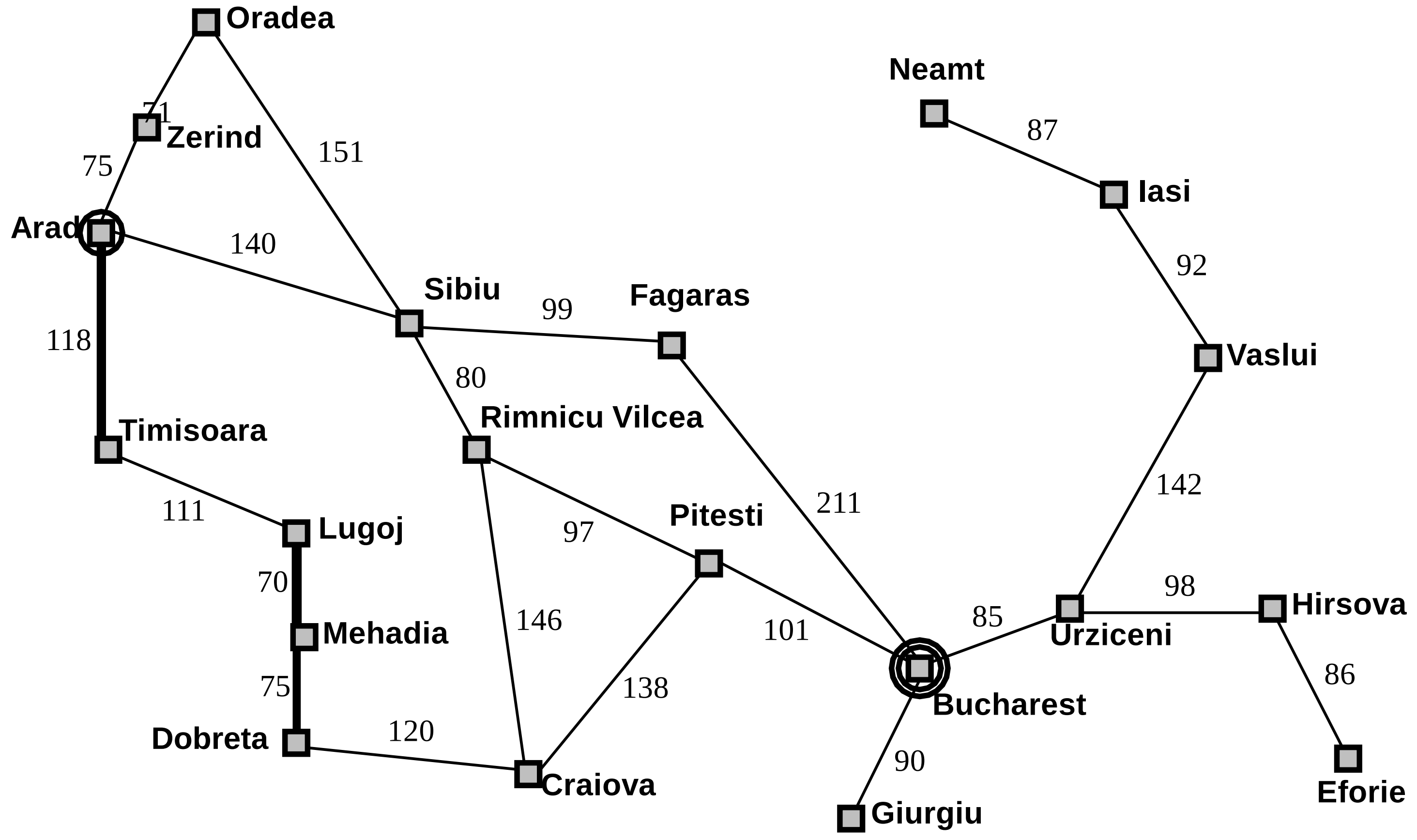
Expand deepest unexpanded node

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



Properties of depth-first search

Complete??



Properties of depth-first search

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

Time??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than breadth-first

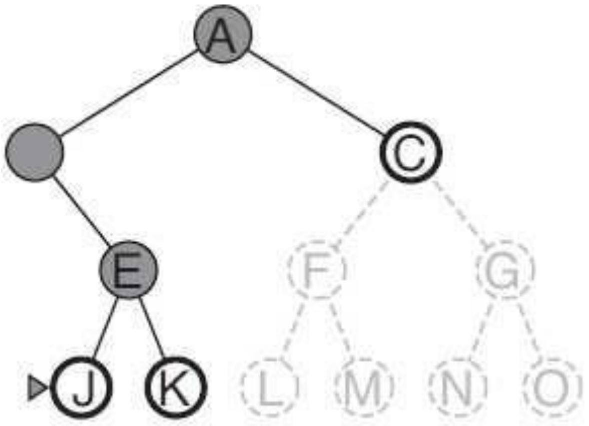
Space??

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?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than breadth-first

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



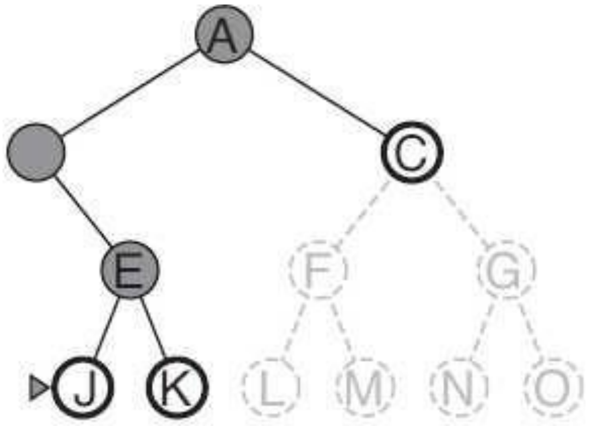
Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than breadth-first

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

Optimal?? No (if J and C are goal nodes, the algorithm returns J)



Depth-limited search

= depth-first search with depth limit *l*,i.e., nodes at depth *l* have no succes- sors

a new source of incompleteness ($l < d$) and nonoptimality ($l > d$)

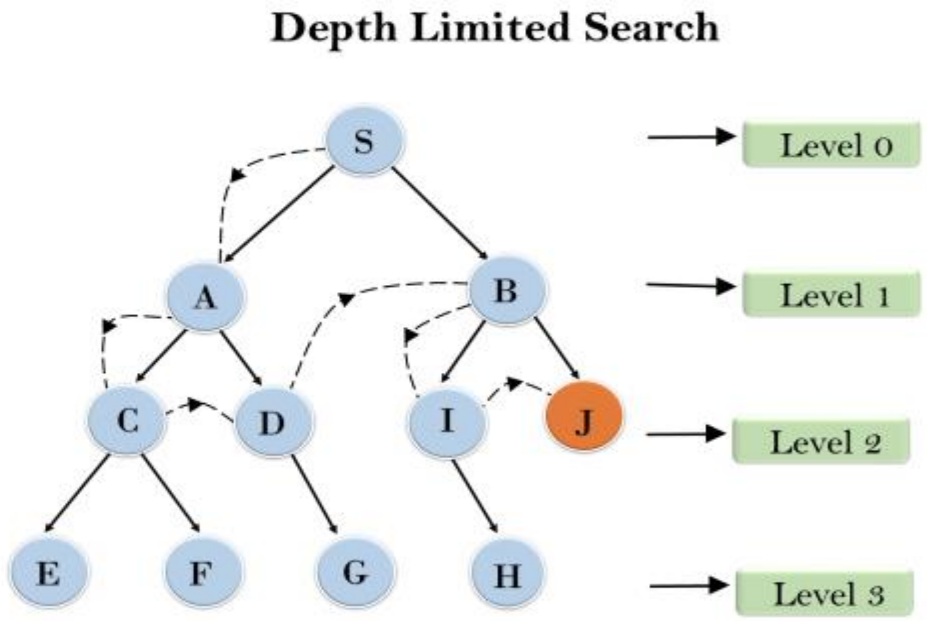
Recursive implementation:

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

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

- A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.
- Depth-limited search can be terminated with two Conditions of failure:
- Standard failure value: It indicates that problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.

- **Advantages:**
- Depth-limited search is Memory efficient.
- **Disadvantages:**
- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.



- **Completeness:** DLS search algorithm is complete if the solution is above the depth-limit.
- **Time Complexity:** Time complexity of DLS algorithm is **$O(b^{\ell})$**.
- **Space Complexity:** Space complexity of DLS algorithm is O**$(b\times\ell)$**.
- **Optimal:** Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $\ell>d$.

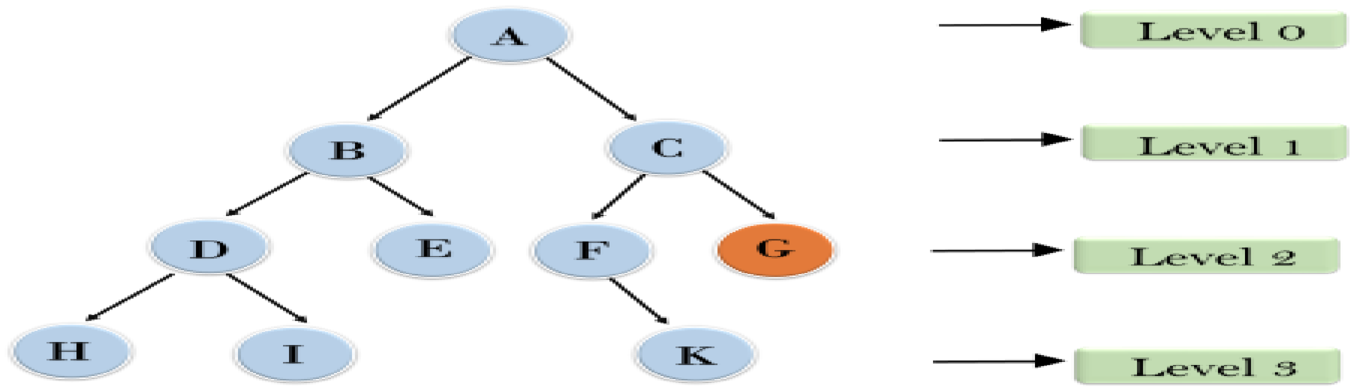
Iterative deepening search

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

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

- The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.
- This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.
- This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

- **Advantages:**
- Itcombines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.
- **Disadvantages:**
- The main drawback of IDDFS is that it repeats all the work of the previous phase.



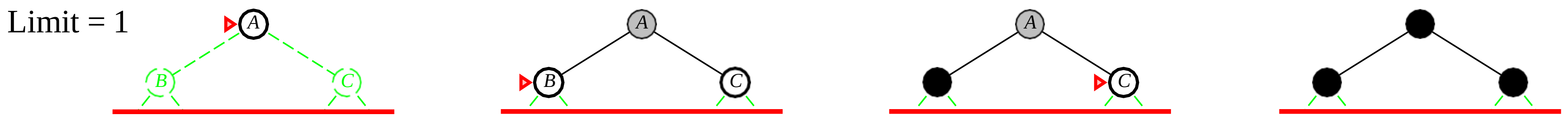
1'st Iteration-----> A
2'nd Iteration----> A, B, C
3'rd Iteration------>A, B, D, E, C, F, G
4'th Iteration------>A, B, D, H, I, E, C, F, K, G
In the fourth iteration, the algorithm will find the goal node.

- **Completeness:**
- This algorithm is complete is ifthe branching factor is finite.
- **Time Complexity:**
- Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **O(b^d)**.
- **Space Complexity:**
- The space complexity of IDDFS will be **O(bd)**.
- **Optimal:**
- IDDFS algorithm is optimal if path cost is a non-decreasing function of the depth of the node.

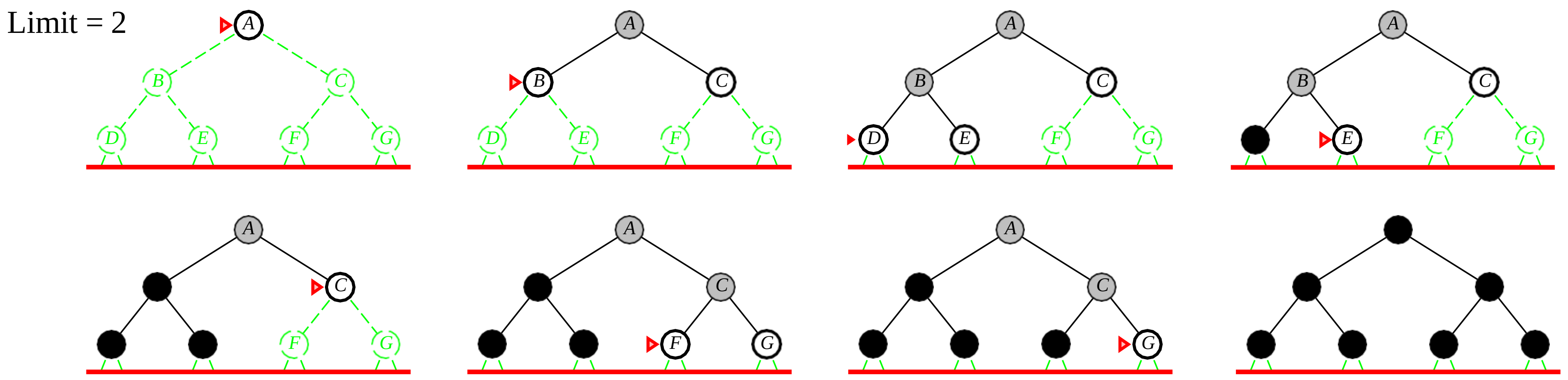
Iterative deepening search $l = 0$

Limit = 0

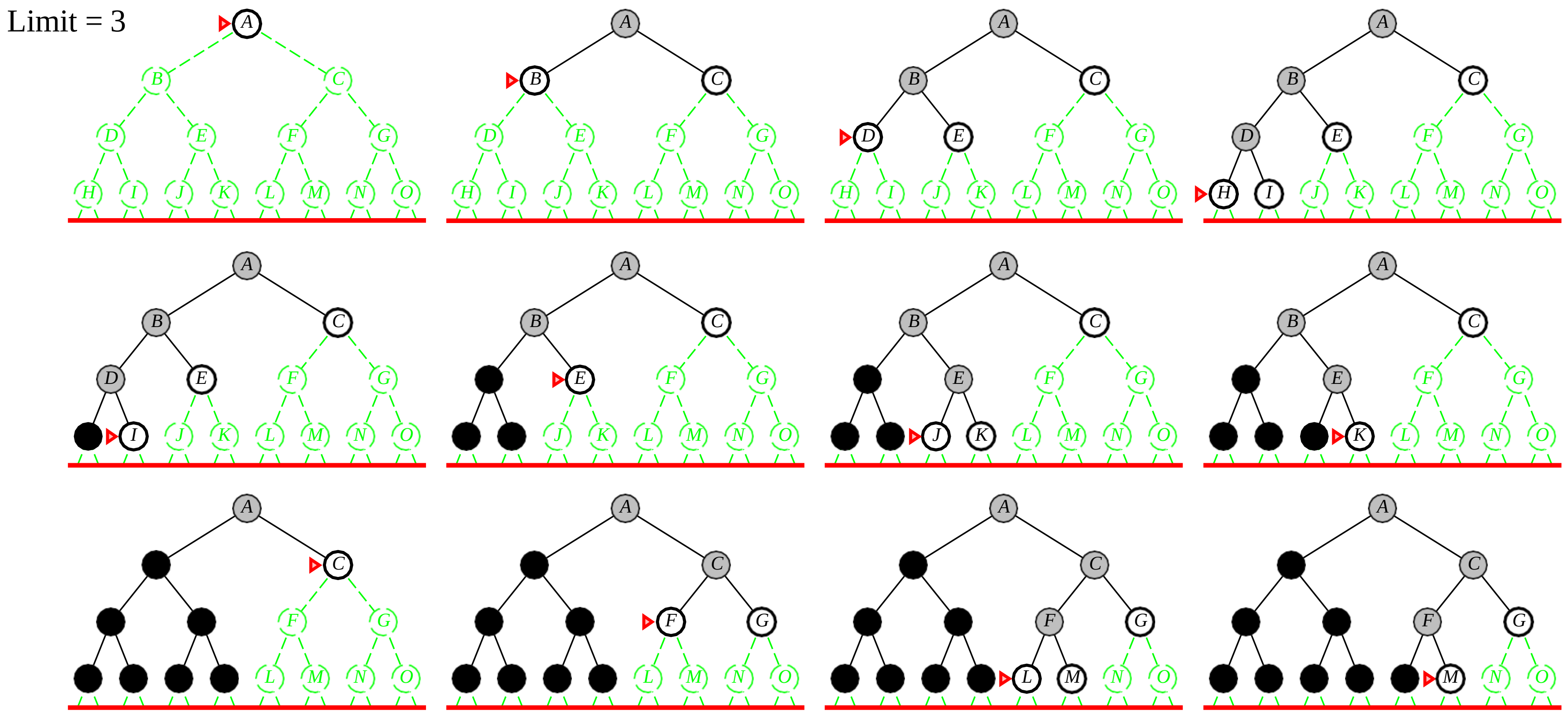
Iterative deepening search $l = 1$



Iterative deepening search *l* = 2



Iterative deepening search *l* = 3



Properties of iterative deepening search

Complete??

Properties of iterative deepening search

Complete?? Yes

Time??

Properties of iterative deepening search

Complete?? Yes

Time?? - nodes generated $N(IDS)(d)b^1 + (d - 1)b^2 + . . . + (1)b^d$
recall $N(BFS) = b^1 + b^2 + ... + b^d$) b=10, d=4

Space??

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??

Properties of iterative deepening search

Complete?? Yes

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

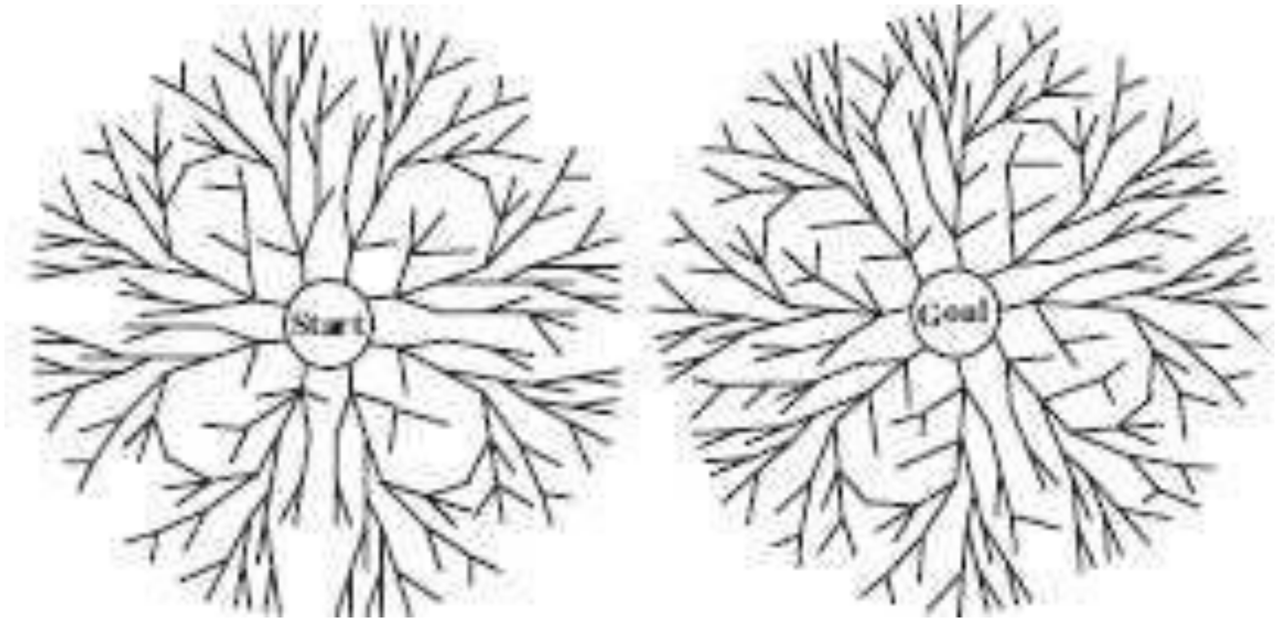
Space?? $O(bd)$

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

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

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

Bidirectional search



$b^{d/2} + b^{d/2} < b^d$

Replacing the goal test: check wheather the frontiers

intersect Solution is not optimal

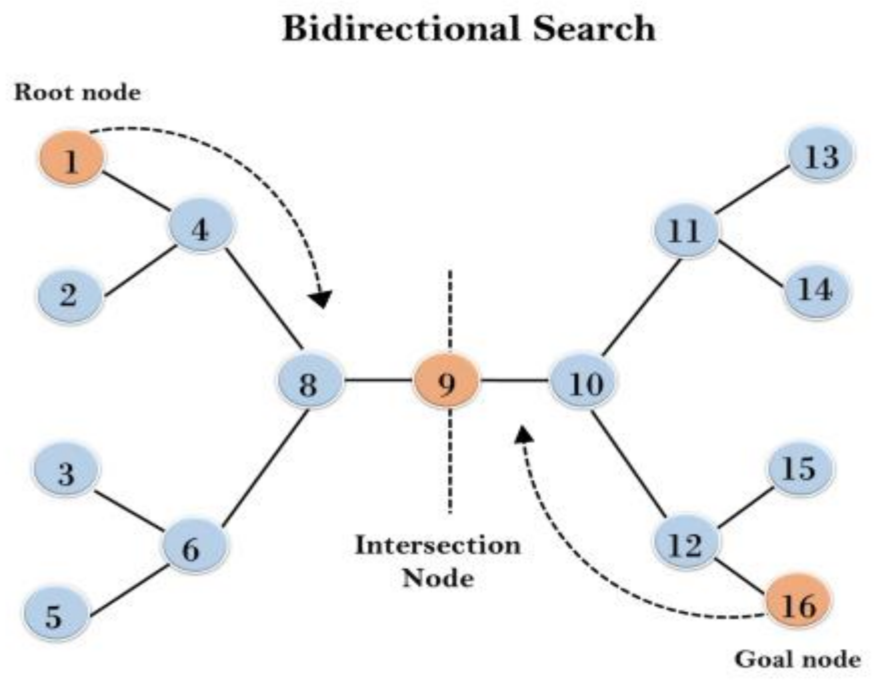
What if the goal is an abstract description: queen”?

”no queen attacks another

- Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node. Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.
- Bidirectional search can use search techniques such as BFS, DFS, etc.
- Advantages:
- Bidirectional search is fast.
- Bidirectional search requires less memory

- **Disadvantages:**
- Implementation of the bidirectional search tree is difficult.
- **In bidirectional search, one should know the goal state in advance.**
- In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.
- The algorithm terminates at node 9 where two searches meet.

- **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)**.
- **Space Complexity:** Space complexity of bidirectional search is **O(b^d)**.
- **Optimal:** Bidirectional search is Optimal.



Outline

- Problem-solving agents
- Problem formulation
- Basic search algorithms
- Informed (heuristic) search strategies

Greedy best-first search algorithm:

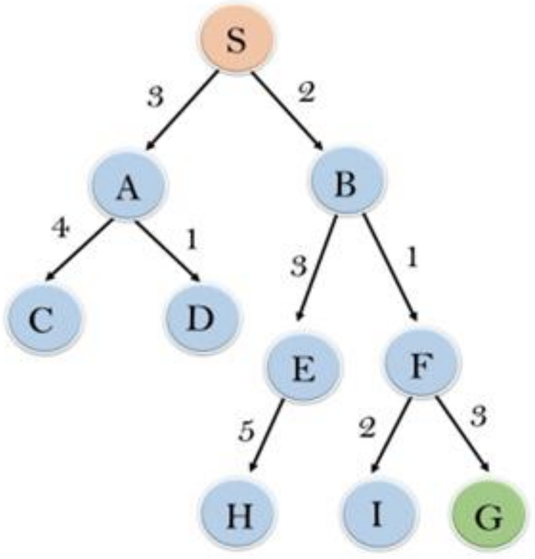
It always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e. $f(n)= g(n)$.

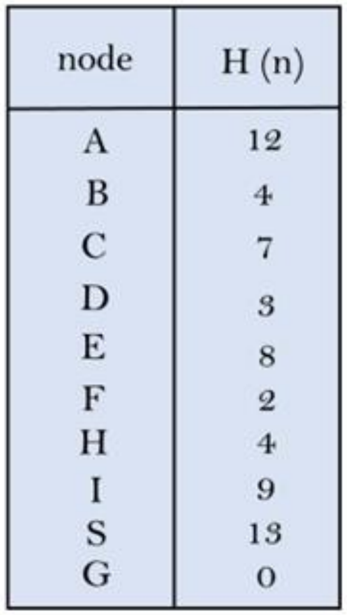
- Were, $h(n)=$ estimated cost from node n to the goal.
- The greedy best first algorithm is implemented by the priority queue.

- Best first search algorithm:
- **Step 1:** Place the starting node into the OPEN list.
- **Step 2:** If the OPEN list is empty, Stop and return failure.
- **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
- **Step 4:** Expand the node n, and generate the successors of node n.
- **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.

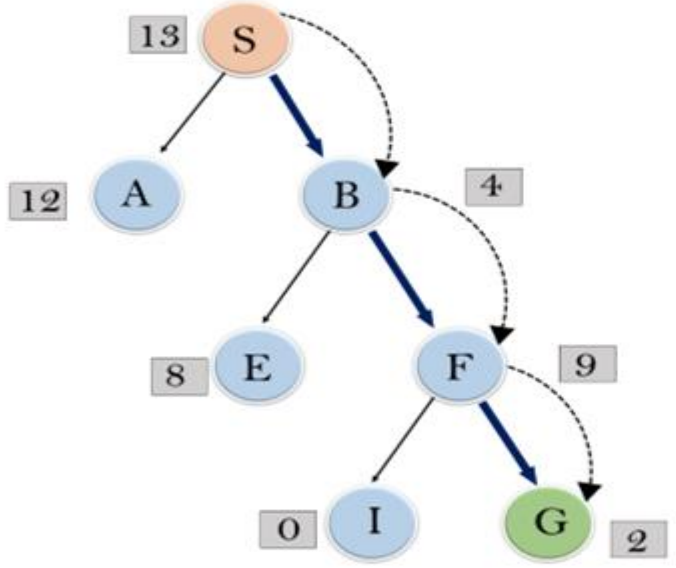
- **Step 6:** For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
- **Step 7:** Return to Step 2.
- Advantages:
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.
- Disadvantages:
- It can behave as an unguided depth-first search in the worst case scenario.
- this algorithm is not optimal .

In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists. Following are the iteration for traversing the above example.





node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0



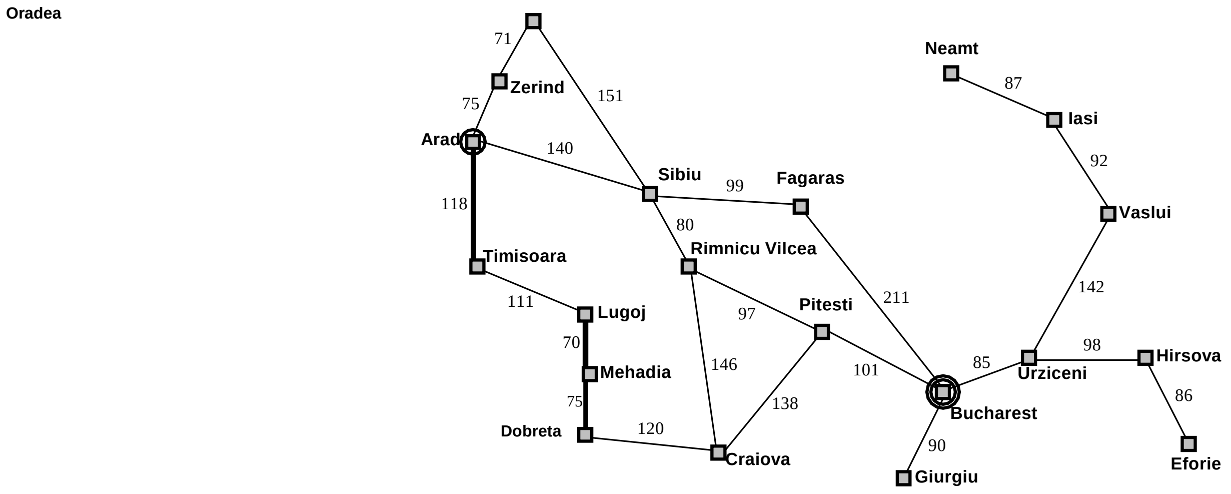
Expand the nodes of S and put in the CLOSED list
Initialization: Open [A, B], Closed [S]
Iteration 1: Open [A], Closed [S, B]
Iteration 2: Open [E, F, A], Closed [S, B]
: Open [E, A], Closed [S, B, F]
Iteration 3: Open [I, G, E, A], Closed [S, B, F]
: Open [I, E, A], Closed [S, B, F, G]
Hence the final solution path will be: **S----> B----->F----> G**

Time Complexity: The worst case time complexity of Greedy best first search is O(b^m).

- **Space Complexity:** The worst case space complexity of Greedy best first search is O(b^m). Where, m is the maximum depth of the search space.
- **Complete:** Greedy best-first search is also incomplete, even if the given state space is finite.
- **Optimal:** Greedy best first search algorithm is not optimal.

Greedy best first search

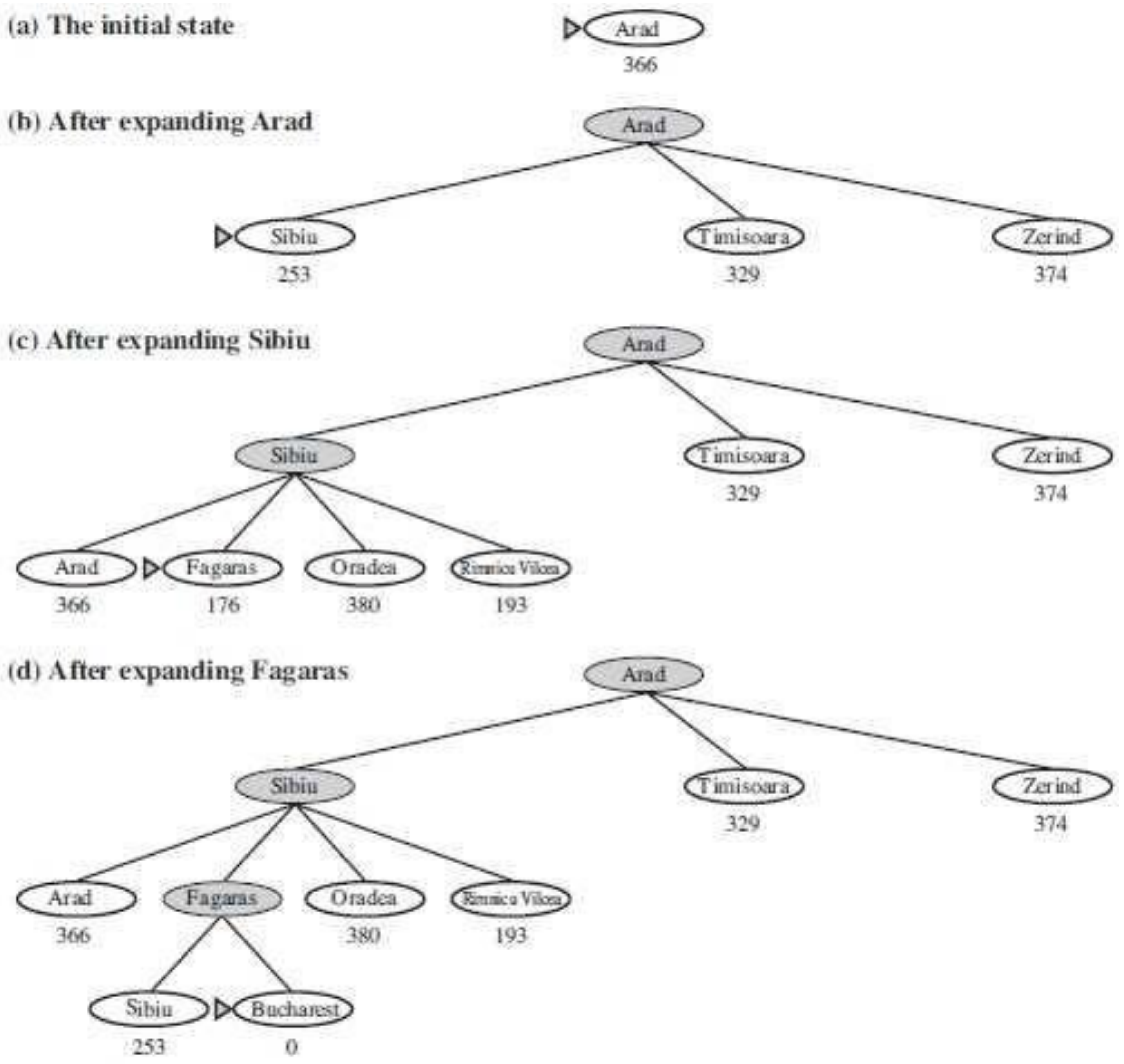
Heuristic function h(n)= estimated cost of the cheapest path from the state at node n to a goal state, e.g., h(n)=straight line distance



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

h_{sld} cannot be computed from the problem description itself

Search cost is minimal: without expanding a node not on the solution path. Is it optimal? Is it complete? (Iasi-Fagaras)

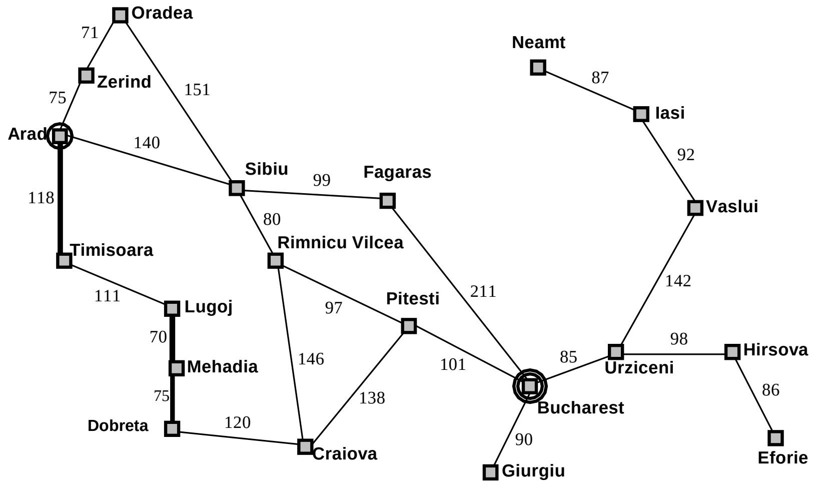

(a) The initial state
Arad
366
(b) After expanding Arad
Arad
Sibiu
253
Timisoara
329
Zerind
374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
176
Oradea
380
Rimnicu Vilcea
193
(d) After expanding Fagaras
Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
Oradea
380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

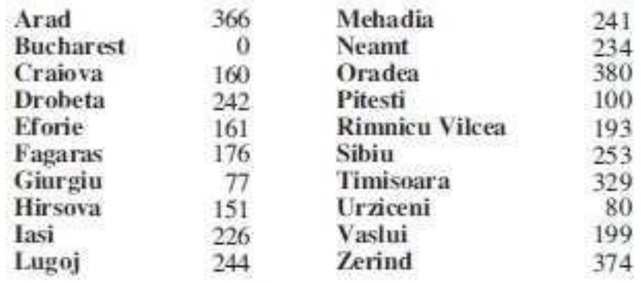
A * search: minimising the total estimated solution cost

g(n) - the cost to reach the solution

h(n) - the cost to get from the node to the

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

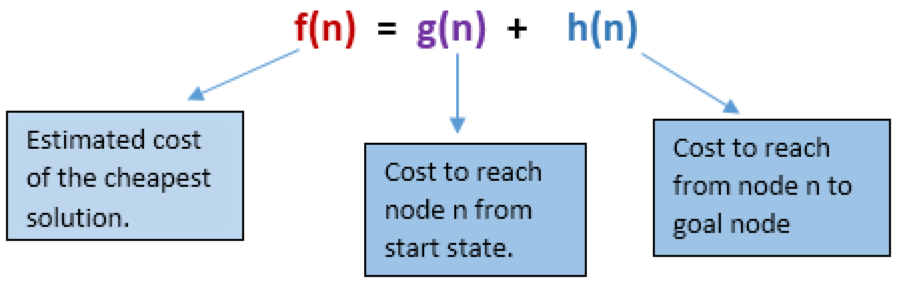




Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

A* search: minimising the total estimated solution cost

A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a **fitness number**.


f(n) = g(n) + h(n)
Estimated cost of the cheapest solution.
Cost to reach node n from start state.
Cost to reach from node n to goal node

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.

Algorithm of A* search:

~~**Step1:** Place the starting node in the OPEN list.~~

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Return to **Step 2**.

The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.

Points to remember:

A* algorithm returns the path which occurred first, and it does not search ~~for all remaining paths.~~

The efficiency of A* algorithm depends on the quality of heuristic.

Complete: A* algorithm is complete as long as:

Branching factor is finite and Cost at every action is fixed.

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

Admissible: the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.

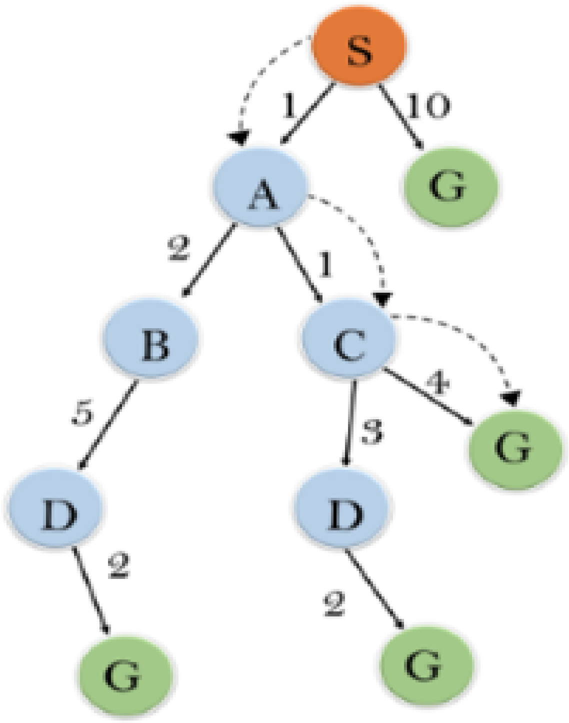
Consistency: Second required condition is consistency for only A* graph-search.

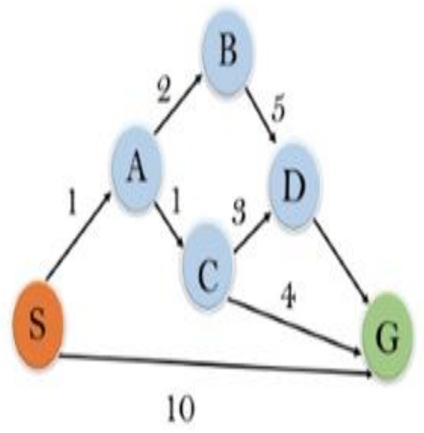
If the heuristic function is admissible, then A* tree search will always find the least cost path.

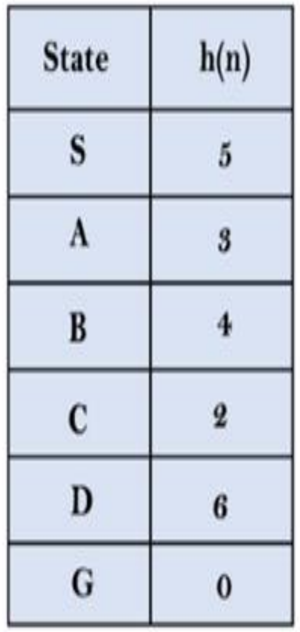
Time Complexity: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is **O(b^d)**

Solution:







State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

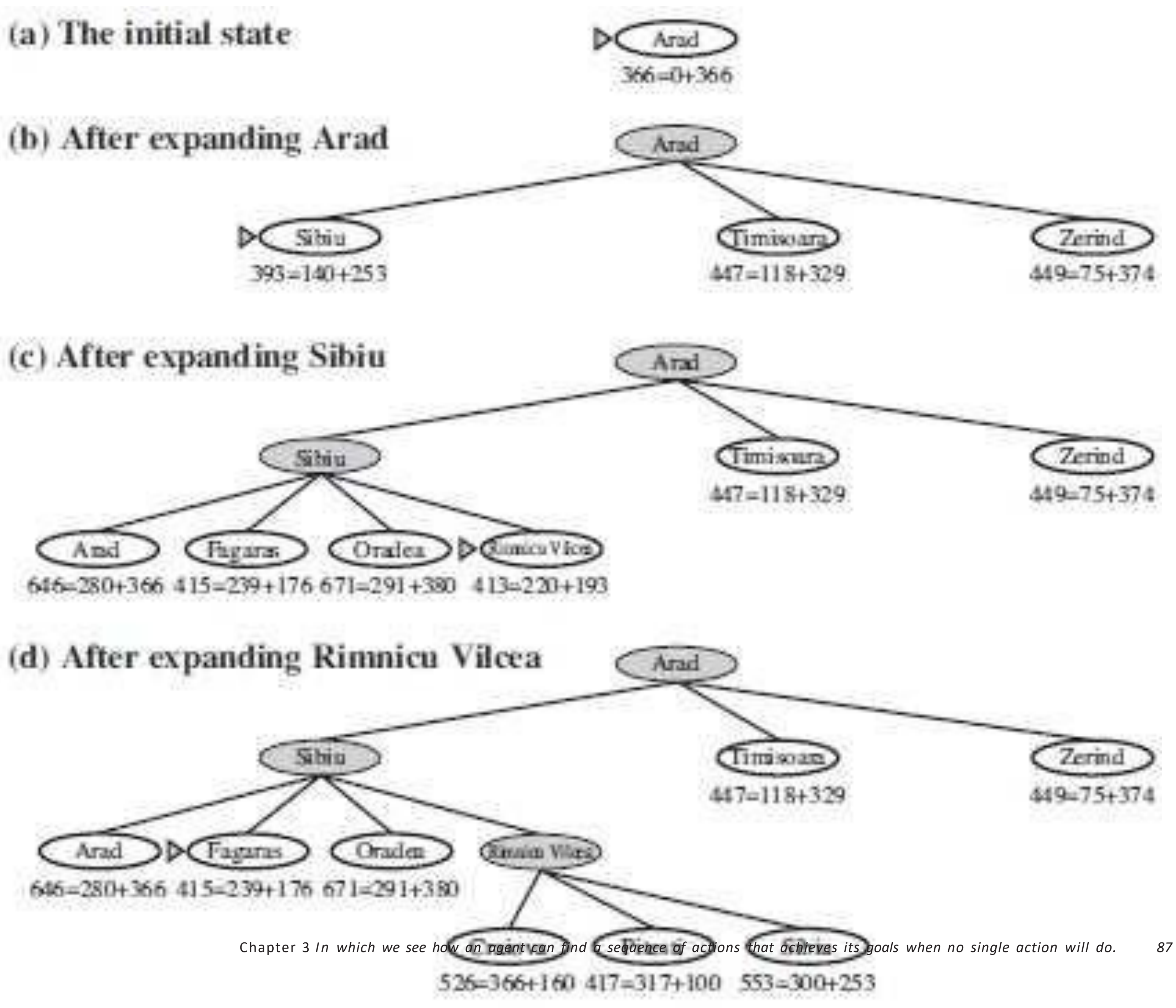
Initialization: {(S, 5)}

Iteration1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

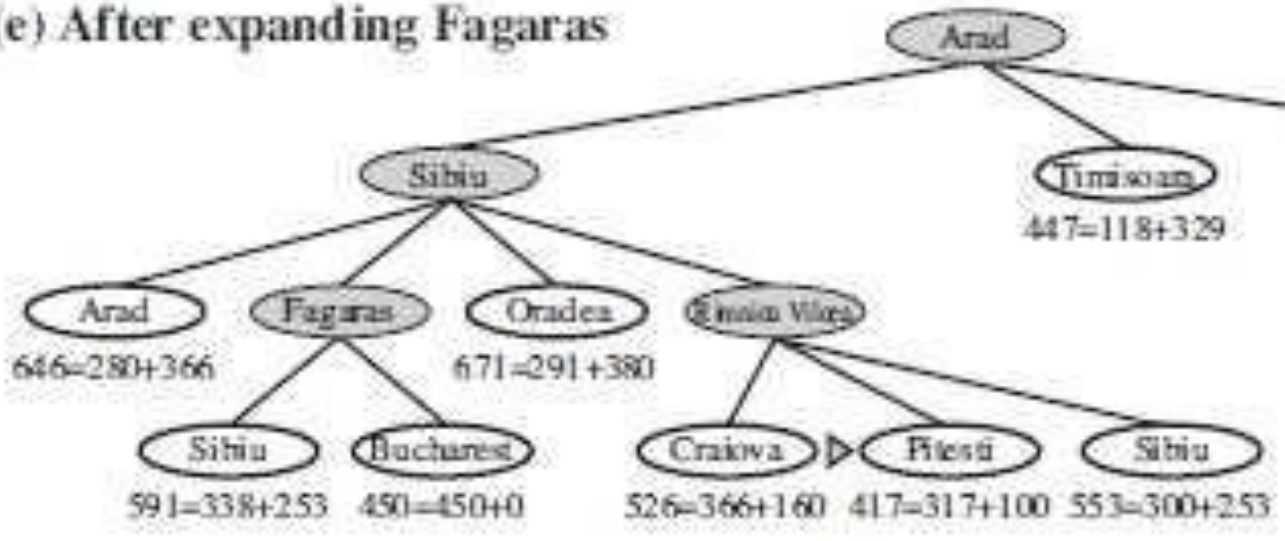
Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Iteration 4 will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.

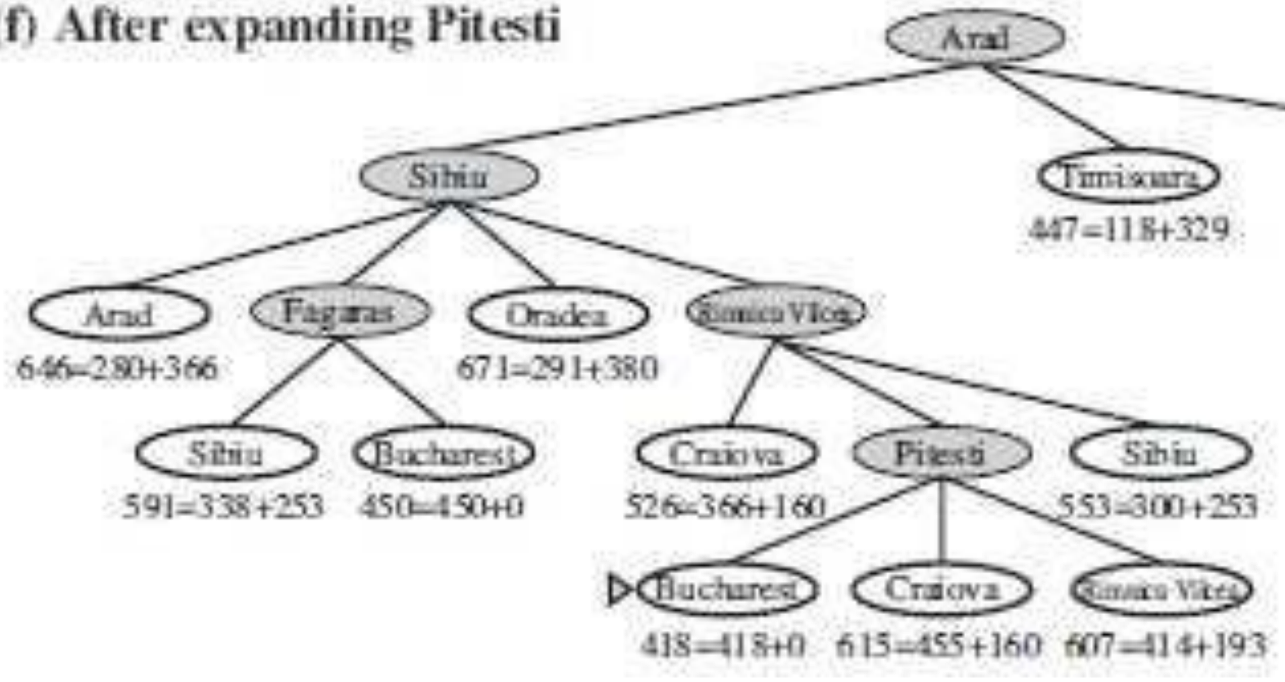

(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193
(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
526=366+160
417=317+100
553=300+253

A* search: minimising the total estimated solution cost

(e) After expanding Fagaras

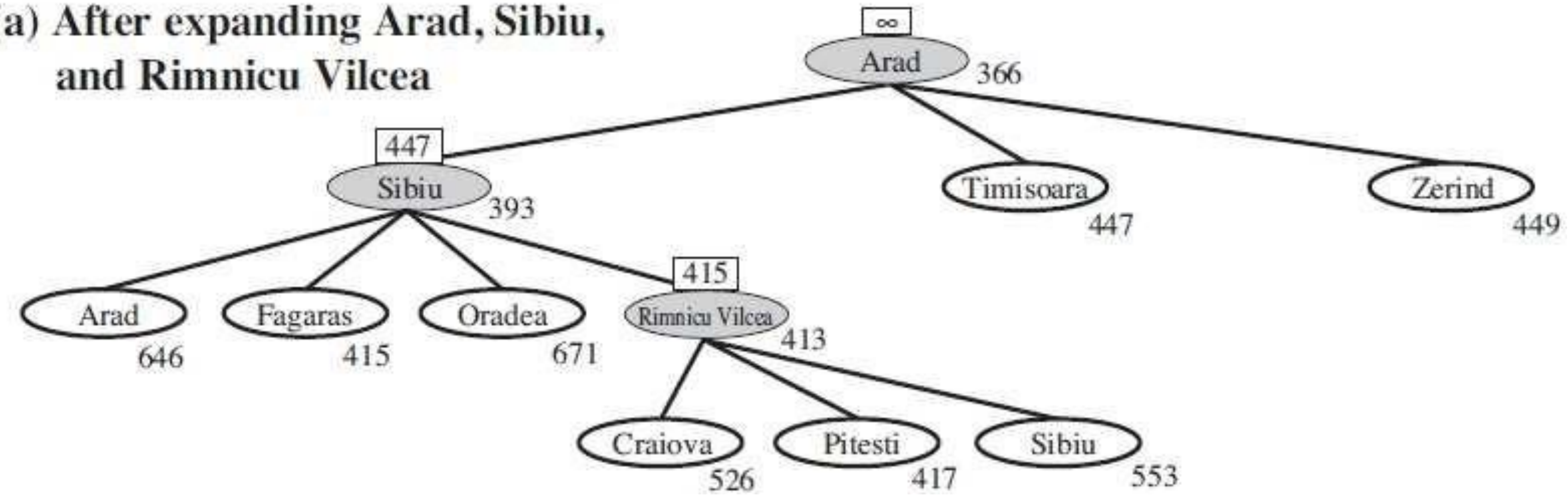


(f) After expanding Pitesti

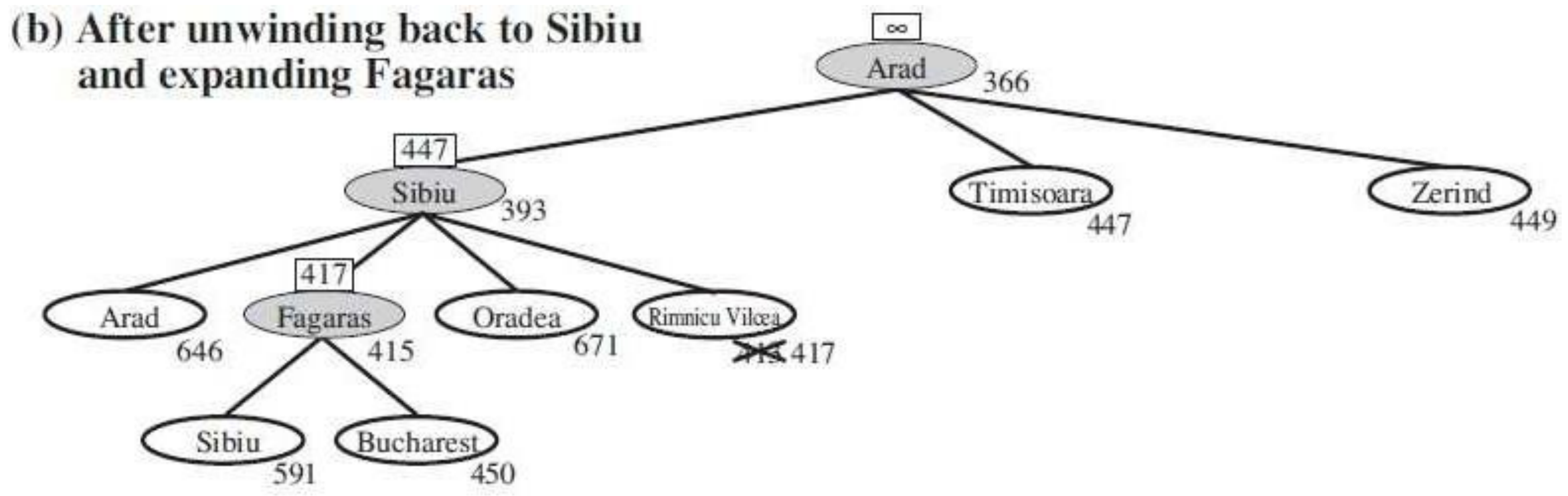


Memory-bounded heuristic search 1

(a) After expanding Arad, Sibiu, and Rimnicu Vilcea

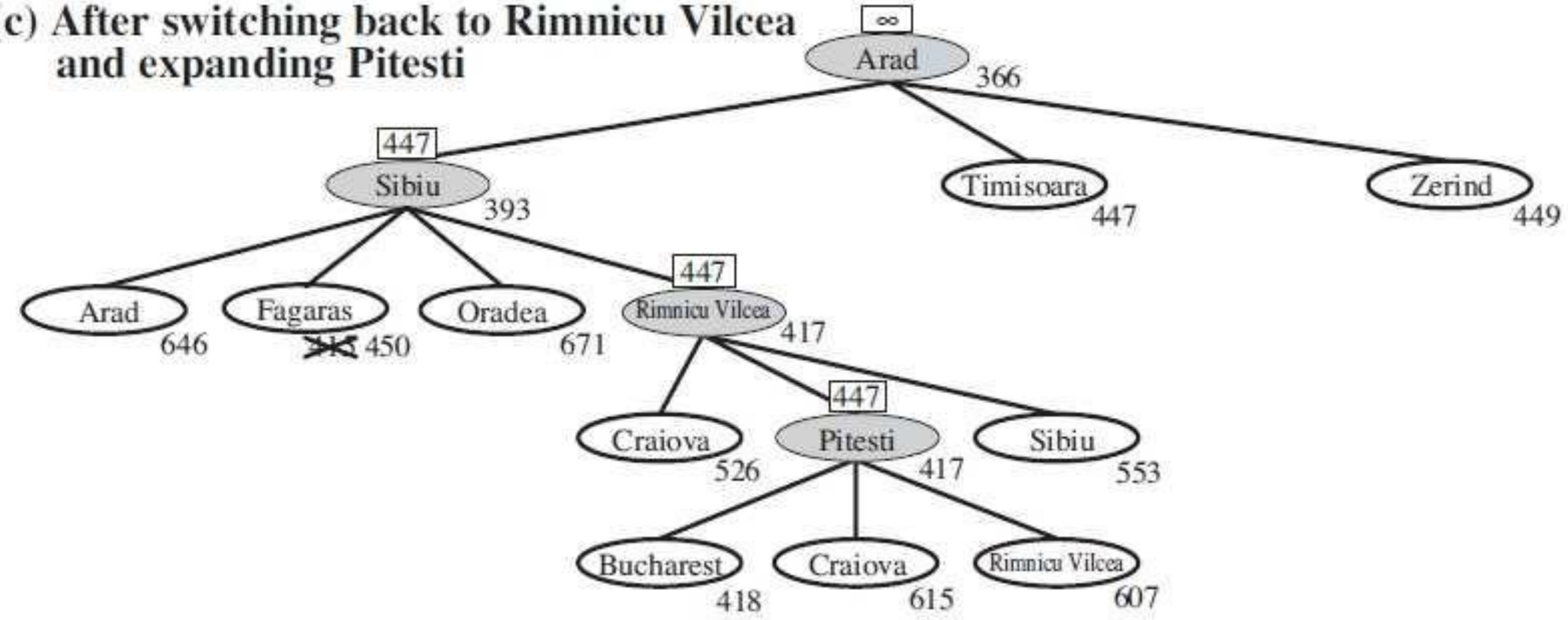


Memory-bounded heuristic search 2

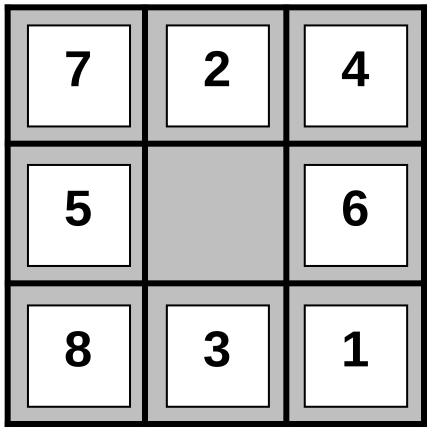


Memory-bounded heuristic search 3

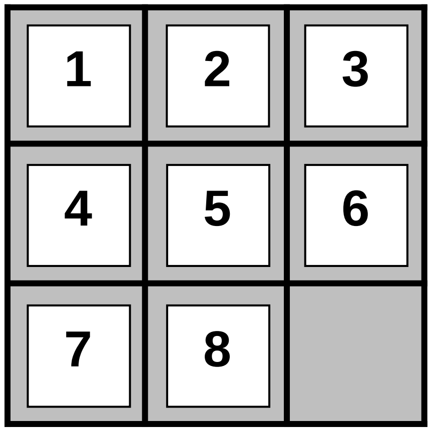
(c) After switching back to Rimnicu Vilcea and expanding Pitesti



Heuristic functions



Start State



Goal State

h_1 - the number of misplaced tiles

h_2 - the sum of the distances of the tiles from their goal positions

Is it possible for a computer to invent such a heuristic mechanically?

Summary

◆Methods that an agent can use to select actions in environments that are deterministic, observable, static, and completely known. In such cases, the agent can construct sequences of actions that achieve its goals - search.

◆Before an agent can start searching for solutions, a goal must be identified and a welldefined problem must be formulated.

◆A problem: 1) initial state, 2) a set of actions, 3) a transition model describing the results of those actions, 4) a goal test function, and 5) a path cost function.

◆Search algorithms: completeness, optimality, time complexity, and space complexity.

◆Uninformed search: breadth-first, uniform cost, depth-first, iterative deep- ening, bidirectional search

◆Informed (heuristic) search: greedy-best first, $A*$, memory bounded