Unit - II

Outline

Solving Problems by Searching:

- Problem-Solving Agents
- Example Problems
- Searching for Solutions
- Uninformed Search Strategies
- Informed (Heuristic) Search Strategies
- Heuristic Functions

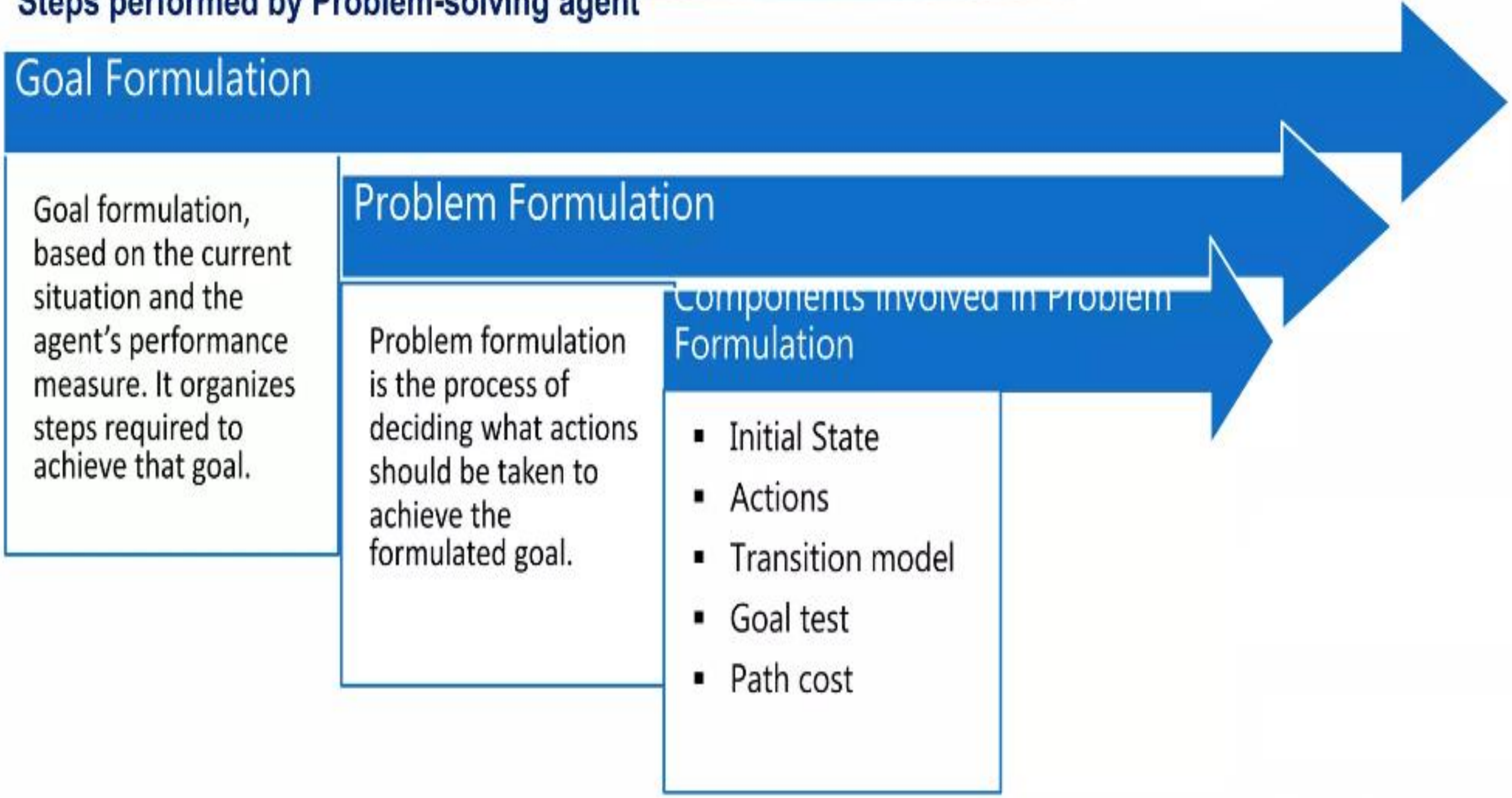
Problems Solving by search

- An important application of Artificial Intelligence is Problem Solving
- Define the Problem statement first
- Generating the solution by keeping the different condition in mind
- Searching is the most commonly used technique of problem solving in artificial intelligence.

Problem Solving Agents:

A problem-solving agent is a **goal-driven agent** and focuses on satisfying the goal.

Steps performed by Problem-solving agent



Goal Formulation

Goal formulation, based on the current situation and the agent's performance measure. It organizes steps required to achieve that goal.

Problem Formulation

Problem formulation is the process of deciding what actions should be taken to achieve the formulated goal.

Components involved in Problem Formulation

- Initial State
- Actions
- Transition model
- Goal test
- Path cost

Problem-Solving Agents

- Intelligent agents are supposed to maximize their performance measure
- An agent can adopt a **goal and aim at** satisfying it.

  Example :

  Imagine an agent in the city of Arad, Romania, enjoying a touring holiday. Now, suppose Agent has a nonrefundable ticket to fly out of Bucharest. Courses of action that don’t reach Bucharest on time can be rejected without further consideration
- **Goal formulation :**
  - Goals help organize behavior by limiting the objectives that the agent is trying to achieve
  - Goal formulation, based on the current situation and the agent’s performance measure, is the first step in problem solving

- The agent’s task is to find out which sequence of actions will get it to a goal state.

→it needs to decide (or we need to decide on its behalf) what sorts of actions and states it should consider

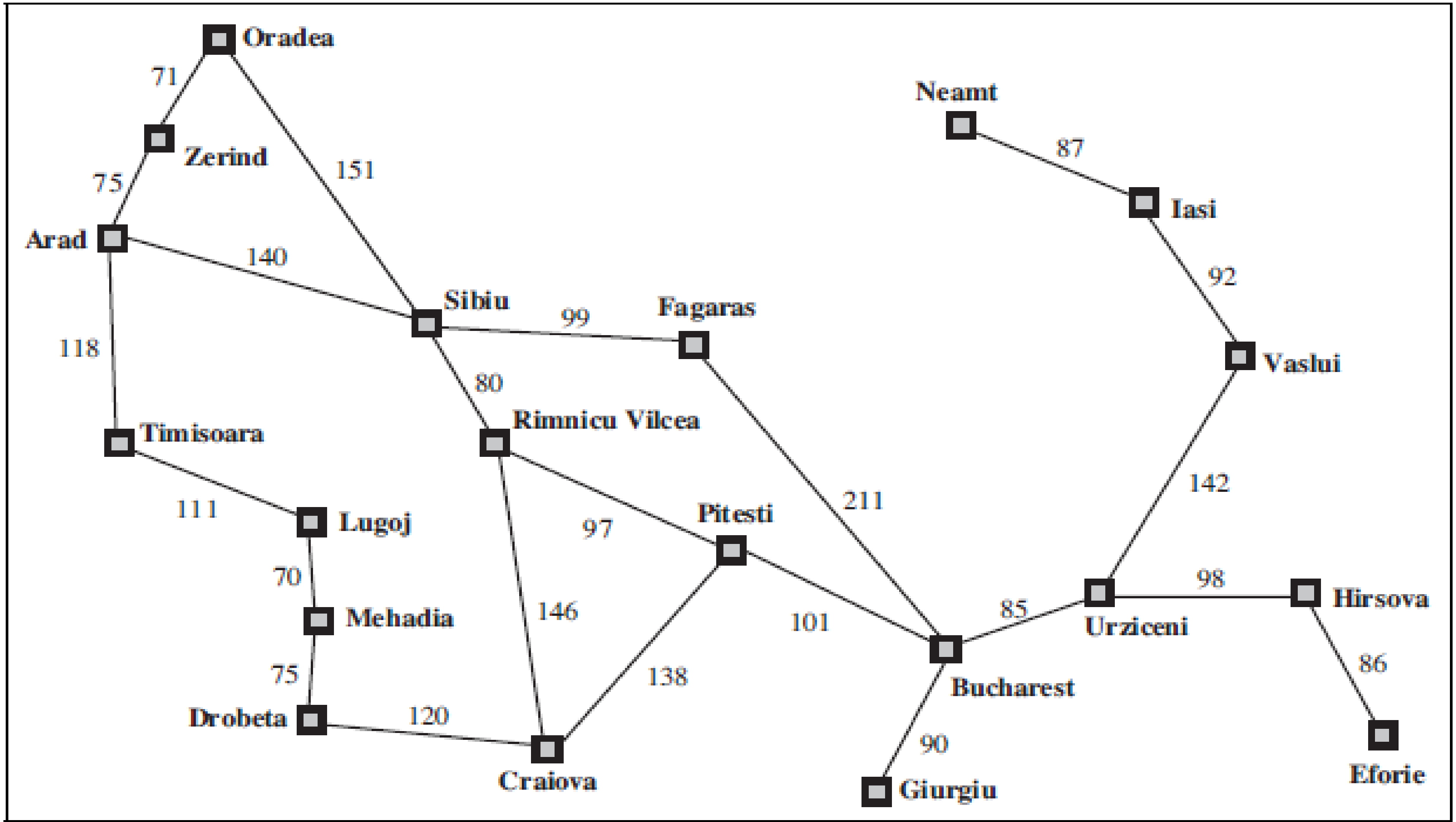


Figure : A simplified road map of part of Romania.

➢ Problem formulation

- Problem formulation is the process of deciding what actions and states to consider, given a goal.

example : let us assume that the agent will consider actions at the level of driving from one major town to another.

- Our agent has now adopted the goal of driving to Bucharest and is considering where to go from Arad. Three roads lead out of Arad, one toward Sibiu, one to Timisoara, and one to Zerind.
- None of these achieves the goal
- In other words, the agent will not know which of its possible actions is best, because it does not yet know enough about the state that results from taking each action. If the agent has no additional knowledge, then it is stuck.
- The best it can do is choose one of the action at random

- But suppose the agent has a map of Romania. either on paper or in his memory. The point of a map is to provide the agent with information about the states it might get itself into and the actions it can take.

- The agent can use this information to consider subsequent stages of a hypothetical journey via each of the three towns, trying to find a journey that eventually gets to Bucharest.

- Once it has found a path on the map from Arad to Bucharest, it can achieve its goal

- In general, an agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value.

- The process of looking for a sequence of actions that reaches the goal is called **search.** A search algorithm takes a problem as input and returns a **solution in the form of an action** sequence. Once a solution is found, the actions it recommends can be carried out. This is called the **execution phase.**
- Thus, we have a simple “formulate, search, execute” design

  for the agent.
- After formulating a goal and a problem to solve, the agent calls a search procedure to solve it.
- It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do
- typically, the first action of the sequence—and then removing that step from the sequence. Once the solution has been executed, the agent will formulate a new goal.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept ) returns an action
 persistent: seq, an action sequence, initially empty
        state, some description of the current world state
        goal , a goal, initially null
        problem, a problem formulation
state←UPDATE-STATE(state, percept )
if seq is empty then
        goal ←FORMULATE-GOAL(state)
        problem ←FORMULATE-PROBLEM(state, goal )
        seq ←SEARCH(problem)
        if seq = failure then return a null action
action ←FIRST(seq)
seq ←REST(seq)
return action
```

Figure   : A simple problem-solving agent

➢ Well defined problems and solutions

- A problem can be defined by four components

  1) Initial state

  2) Actions

  3) Goal test

  4) Path cost

1) Initial state : The initial state that the agent starts in

for example, the initial state for our agent is Romania might be described as In(Arad).

- A description of the possible actions available to the agent. The most common formulation uses a successor function

for Eaxample, from the state In(Arad),the successor function for the Romania problem would return

(Go(Sibiu), In(Sibiu)), (Go(Timisoara), In(Timisoara)) (Go(Zerind), In(Zerind))

Together, the initial state, actions, and transition model implicitly define the **state space** of the problem

A **path in the state space is a sequence** of states connected by a sequence of actions.

- ✓ The **goal test,** which determines whether a given state is a goal state.

  The agent’s goal in Romania is the singleton set {In(Bucharest)}.

- ✓ A **path cost function** that assigns a numeric cost to each path.

  the agent trying to get to Bucharest, time is of the essence, so the cost of a path might be its length in kilometers.

- • **The step cost** function that assigns a numeric cost to each path.
- ✓ **solution to a problem is an** sequence of **action** that leads from the initial state to a goal state.
- ✓ Solution quality is measured by the path cost function, and an **optimal solution has the lowest path cost among all solutions.**

Example Problems

- vacuum cleaner world problem
- 8 puzzle problem
- 8-queens problem

vacuum cleaner world

- States : The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. thus there are $2 \times 2^2 = 8$ possible world states. A larger environment with n locations has $n.2^n$ states.

- Initial state : any state can be designated as the initial state.

- Successor function : In this simple environment, each state has just three actions: *Left*, *Right*, *and Suck*. *Larger* environments might also include *Up and Down*.

- Goal test : This checks whether all the squares are clean.

- Path cost : Each step costs 1, so the path cost is the number of steps in the path

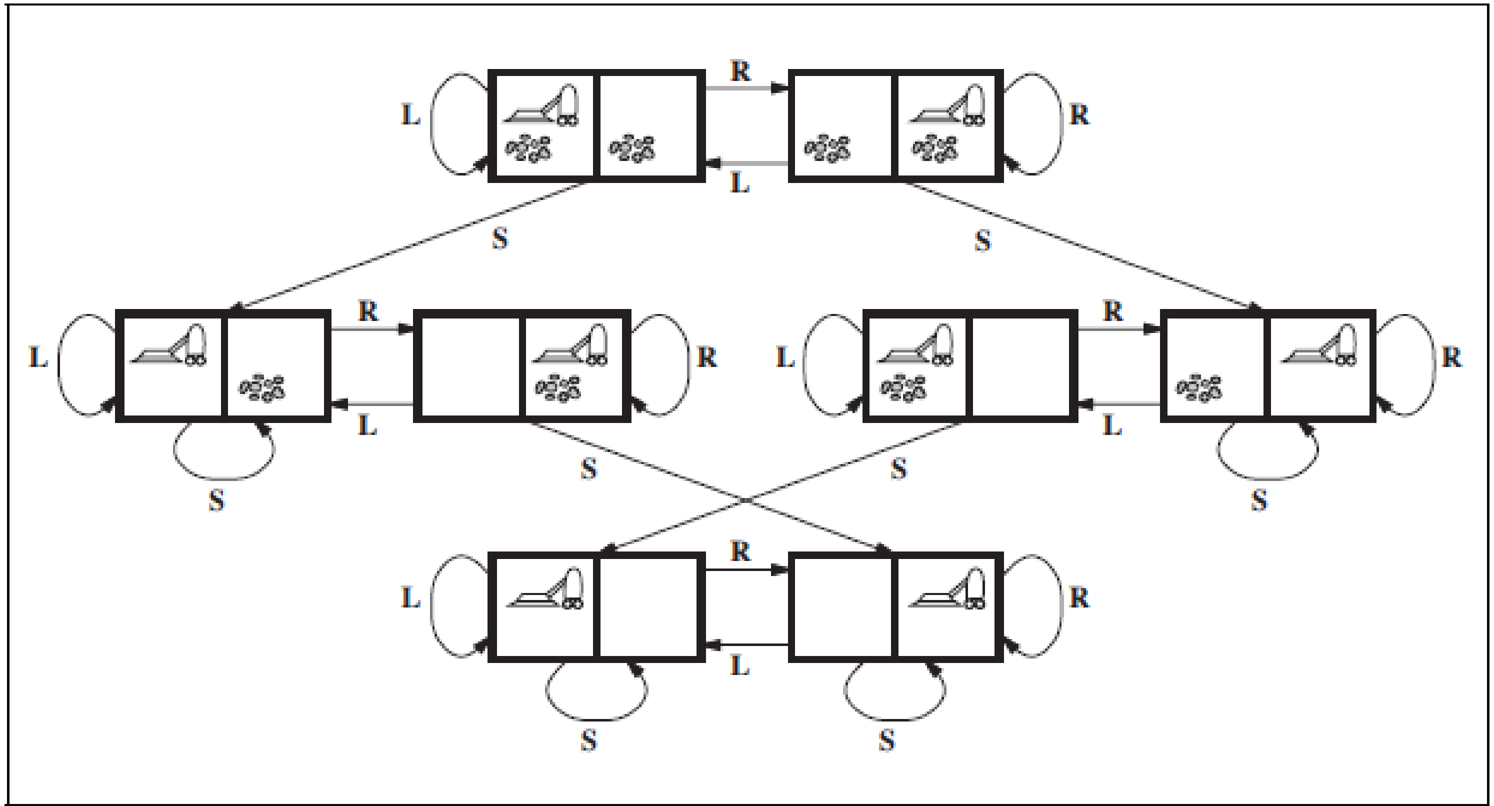


Figure The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

states ?? : integer dirt and robot locations

actions?? : Left, Right, Suck, NoOp

goal test?? : no dirty

path cost ?? : 1 per action (0 for NoOP)

Compared with the real world, this toy problem has discrete locations, discrete dirt, reliable cleaning, and it never gets any dirtier.

8 puzzle problem

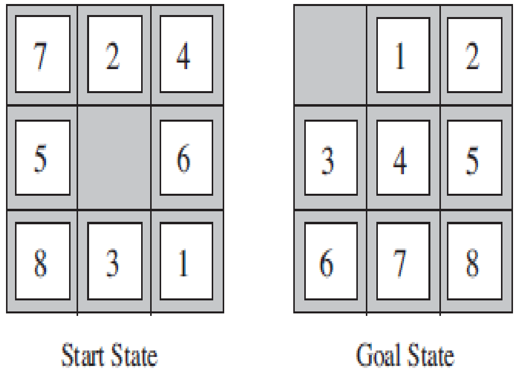


Figure A typical instance of the 8-puzzle.

states ?? : integer location of tile (ignore intermediate positions)
actions?? : Left, Right, up, down tile (ignore intermediate positions)
goal test?? : goal state
path cost ?? : 1 per move

- **Let us consider a 3×3 Board with eight numbered tiles and a blank space**

States: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.

- **Initial state: Any state can be designated as the initial state**.
- **Actions: The simplest formulation defines the actions as movements of the blank space** *Left, Right, Up, or Down. Different subsets of these are possible depending on where* the blank is.
- **Transition model: Given a state and action, this returns the resulting state;**

for example, if we apply *Left to the start state in above Figure , the resulting state has the 5 and the blank* switched.

- **Goal test: This checks whether the state matches the goal configuration shown in Figure** (Other goal configurations are possible.)
- **Path cost: Each step costs 1, so the path cost is the number of steps in the path.**

8-queens problem

- The goal of 8-queens problem is placing the 8 queens on chessboard such that no queens attacks any other.

✓ States : Any arrangement of 0 to 8 queens on the board is a state.

✓ Initial state : No queens on the board.

✓ Actions : Add a queen to any empty square.

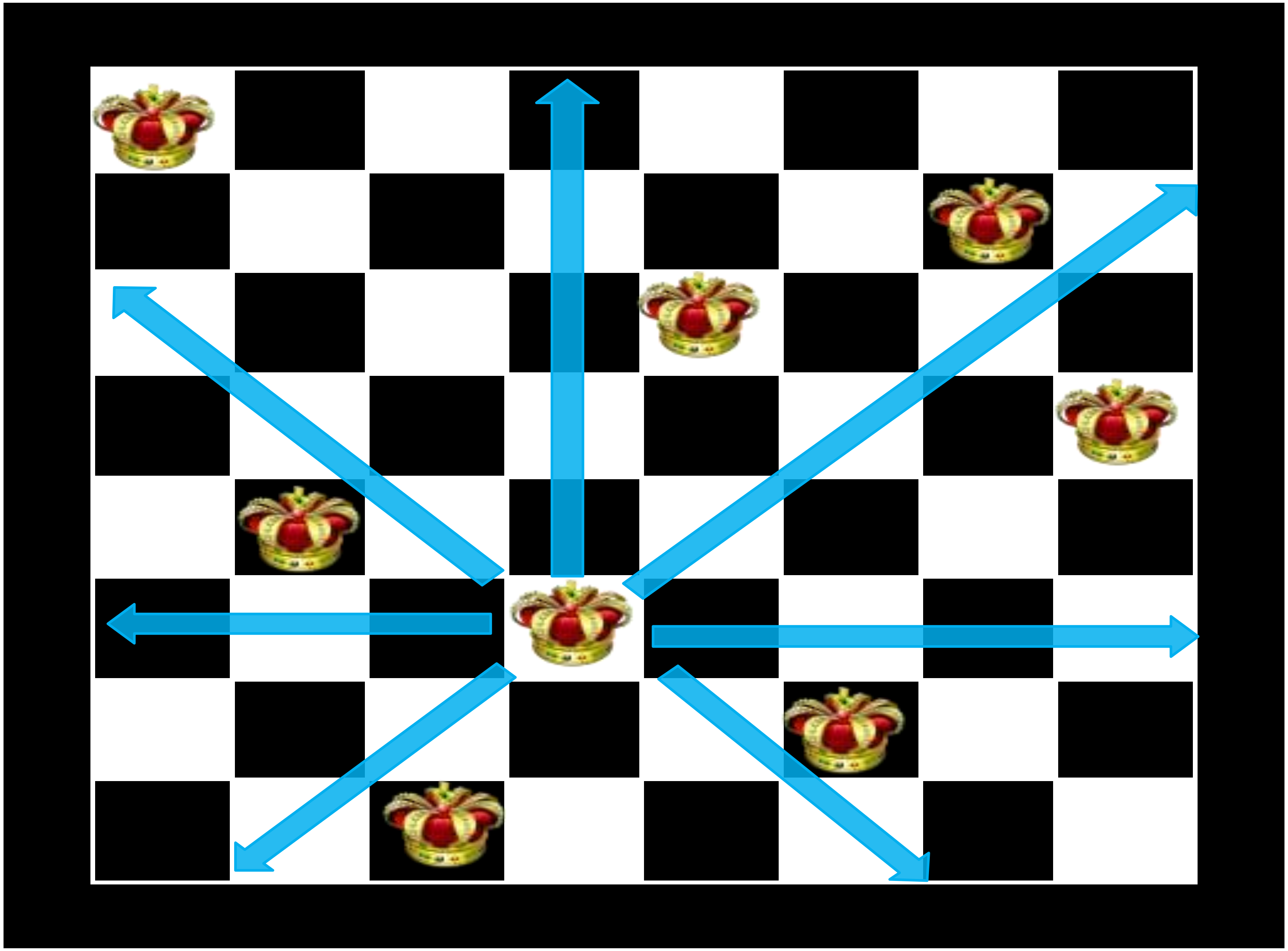
✓ Transition model : Returns the board with a queen added to the specified square.

✓ Goal test : 8 queens are on the board, none attacked.

In this formulation, we have 64 · 63 · · · 57 ≈ 1.8×1014 possible sequences to investigate. A better formulation would prohibit placing a queen in any square that is already attacked:

✓ **States: All possible arrangements of n queens (0 ≤ n ≤ 8), one per column in the** leftmost n columns, with no queen attacking another.

✓ **Actions: Add a queen to any square in the leftmost empty column such that it is not** attacked by any other queen.



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**.**

- Then we need to consider taking various actions. We do this by **expanding the current state; that is,** applying each legal action to the current state, thereby **generating a new set of states. In** this case, we add three branches from the **parent node** ***In(Arad) leading to three new child*** **nodes:** ***In(Sibiu), In(Timisoara), and In(Zerind)*****.** ***Now we must choose which of these three*** possibilities to consider further.

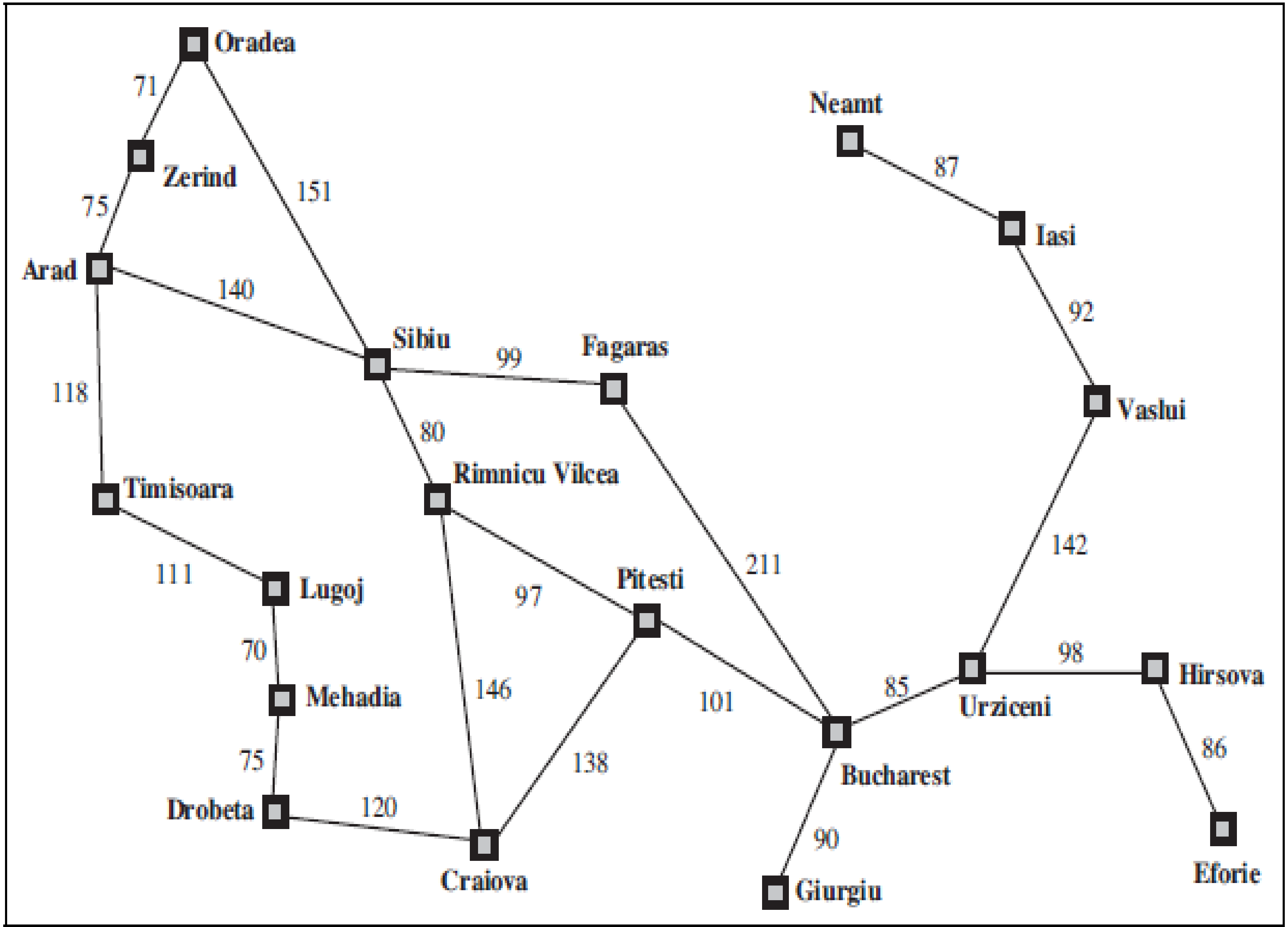
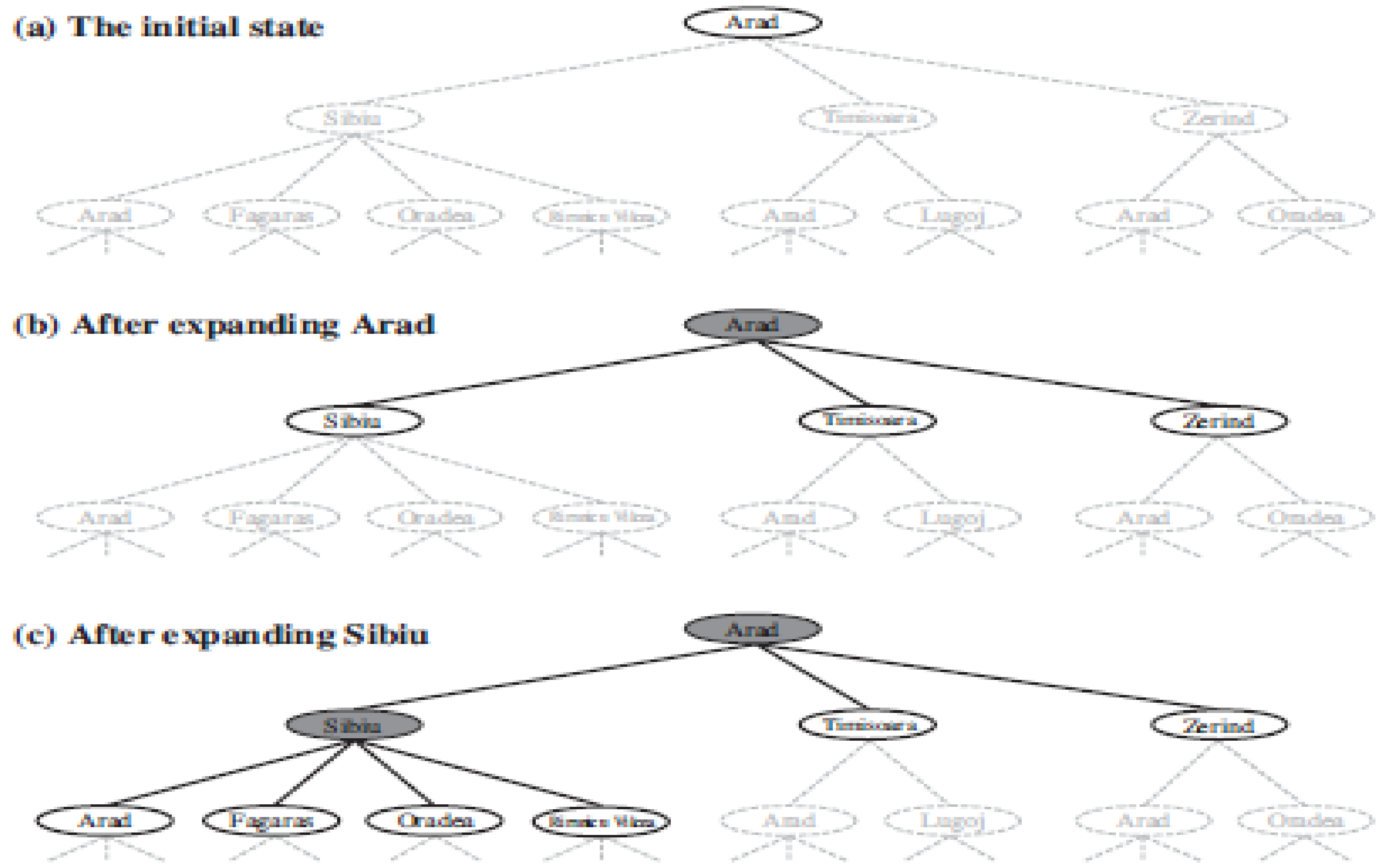


Figure : A simplified road map of part of Romania.



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

SEARCHING FOR SOLUTIONS

- This is the essence of search-following up one option now and putting the others aside for later, in case the first choice does not lead to a solution.
- Each of these six nodes is a leaf node(i.e a node with no children in the tree). The set of all available for expansion at any given point is called the frontier.

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
    only if not in the frontier or explored set
```

Figure : An informal description of the general tree-search and graph-search algorithms

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:
  - **n.STATE:** the state in the state space to which the node corresponds;
  - **n.PARENT:** the node in the search tree that generated this node;
  - **n.ACTION:** the action that was applied to the parent to generate the node;
  - **n.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.

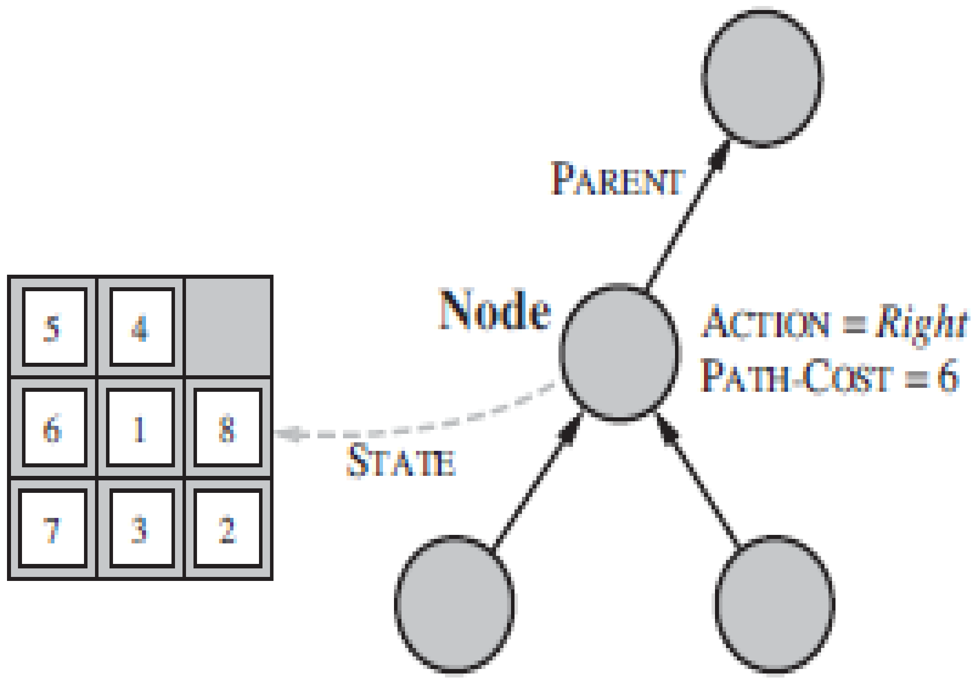


Figure : Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

➢ Measuring problem-solving performance

- **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?

➢ Time and space complexity are always considered with respect to some measure of the problem difficulty.

Types of search algorithms

- ➢ Based on the search problems we can classify the search algorithm into two types
  1. Uninformed Search Algorithm (Blind Search)
  2. Informed Search Algorithm (Heuristic Search)
- ➢ Uninformed Search Algorithm (Blind Search) :
  - The Uninformed Search Algorithm does not contain any domain knowledge such as closeness, the location of the goal.
  - it operates in a brute force way, as it only includes information about how to traverse the tree & how to identify leaf & goal nodes.
  - Uninformed Search Algorithm applies a way in which search tree is searched without any information about search space like initial state, test for the goal, so it is called blind search.

Uninformed Search Strategies

- *Artificial Intelligence* is the study of building agents that act rationally. Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.
- A search problem consists of:
  - A State Space. Set of all possible states where you can be.
  - A Start State. The state from where the search begins.
  - A Goal Test. A function that looks at the current state returns whether or not it is the goal state.
- The Solution to a search problem is a sequence of actions, called the plan that transforms the start state to the goal state.
- This plan is achieved through search algorithms.

Types of search algorithms

Uninformed Search Algorithm	Informed Search Algorithm
1. Breadth First Search 2. Uniform Cost Search 3. Depth First Search 4. Depth Limited Search 5. Iterative Deeping Depth First Search	1.Best First Search 2. A* Search 3.AO* Algorithm 4. Problem Reduction 5. Hill Climbing

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.*
- Breadth-first search is an instance of the general graph-search algorithm , in which the *shallowest unexpanded node is chosen for expansion*
- This is achieved very simply by using a FIFO queue for the frontier
- the goal test is applied to each node when it is *generated rather than when it is selected for* expansion
- breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node
- The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level. Each of *these generates b more nodes, yielding* b^3 nodes at the third level, and so on.

- the total number of nodes generated is

$$b + b^2 + b^3 + \cdot \quad \cdot \quad \cdot + b^d = O(b^d)$$

- Time complexity = O(V+E)

$$O(b^{d+1})$$

Here b = Branch Factor $\longrightarrow$ The branching factor is property of tree that defines the number of children at each node.

d= level of depth $\longrightarrow$ Depth. In a tree, number of edges from the root node to the particular node are called the depth of the tree.

- space complexity is $O(b^d)$

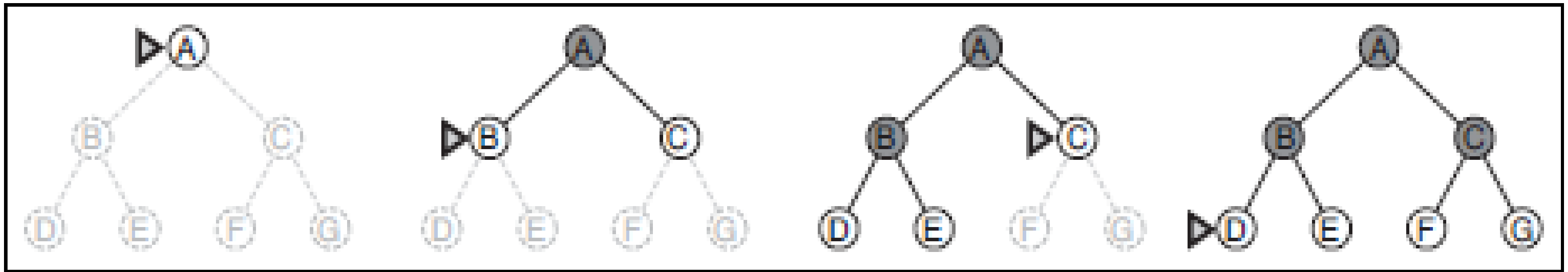


Figure Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

```
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)
```

Breadth-first search on a graph.

Depth-first search

- Depth-first search always expands the *deepest node in the current frontier of the search tree*
- The search proceeds immediately to the deepest level of the search tree.
- breadth-first-search uses a FIFO queue, depth-first search uses a LIFO STACK.
- A LIFO STACK means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected.

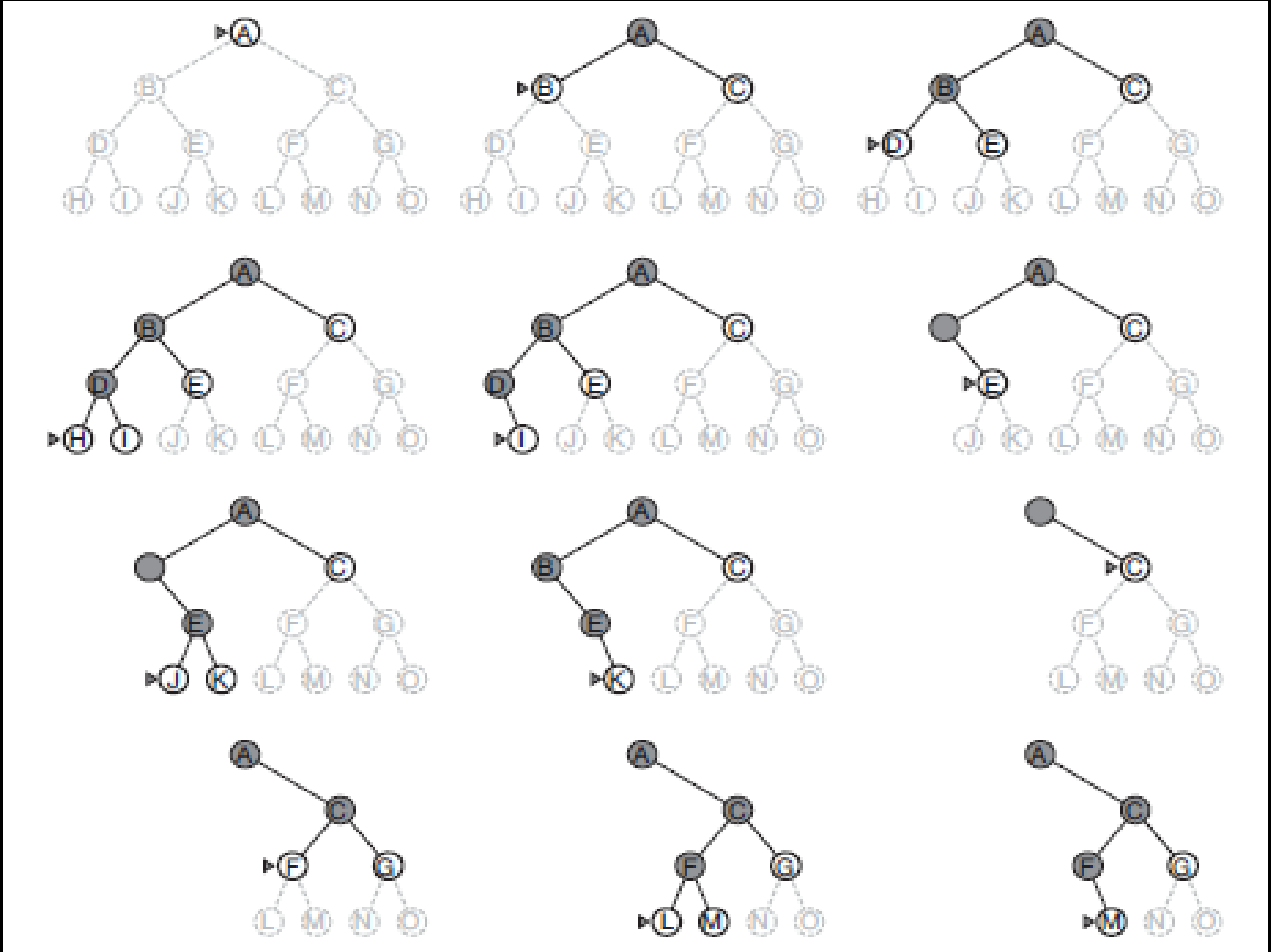


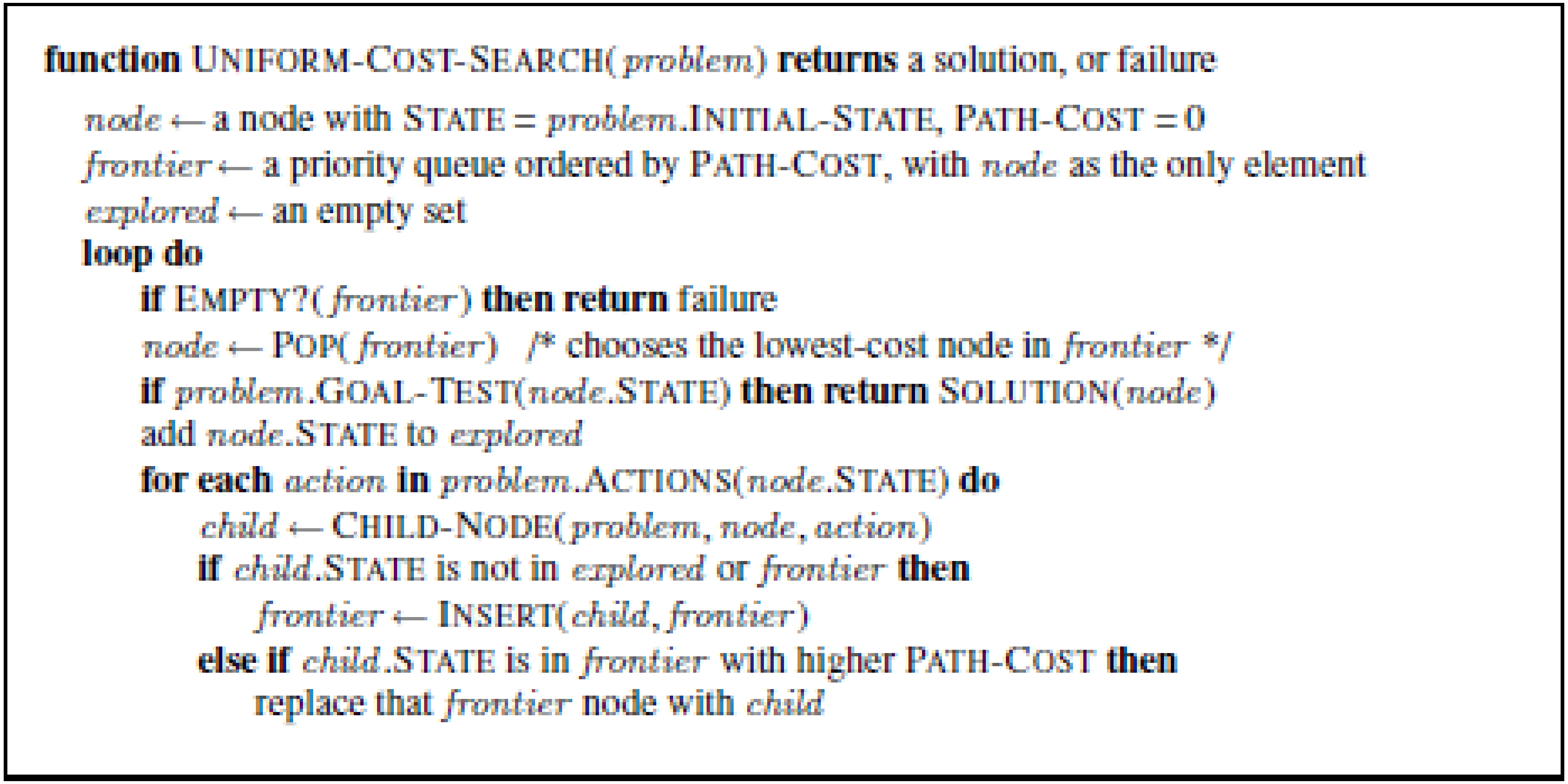
Figure Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and *M* is the only goal node.

- The properties of depth-first search depend strongly on whether the graph-search or tree-search version is used.
- The graph-search version, which avoids repeated states and redundant paths, is complete in finite state spaces because it will eventually The tree-search version, on the other hand, is *not complete E*xpand every node.
- depth-first search called **backtracking search**
- The time complexity of depth-first graph search is bounded by the size of the state space
- A depth-first tree search, on the other hand, may generate all of the O(b^m) nodes in the search tree, where m is the maximum depth of any node

2) Uniform-cost search

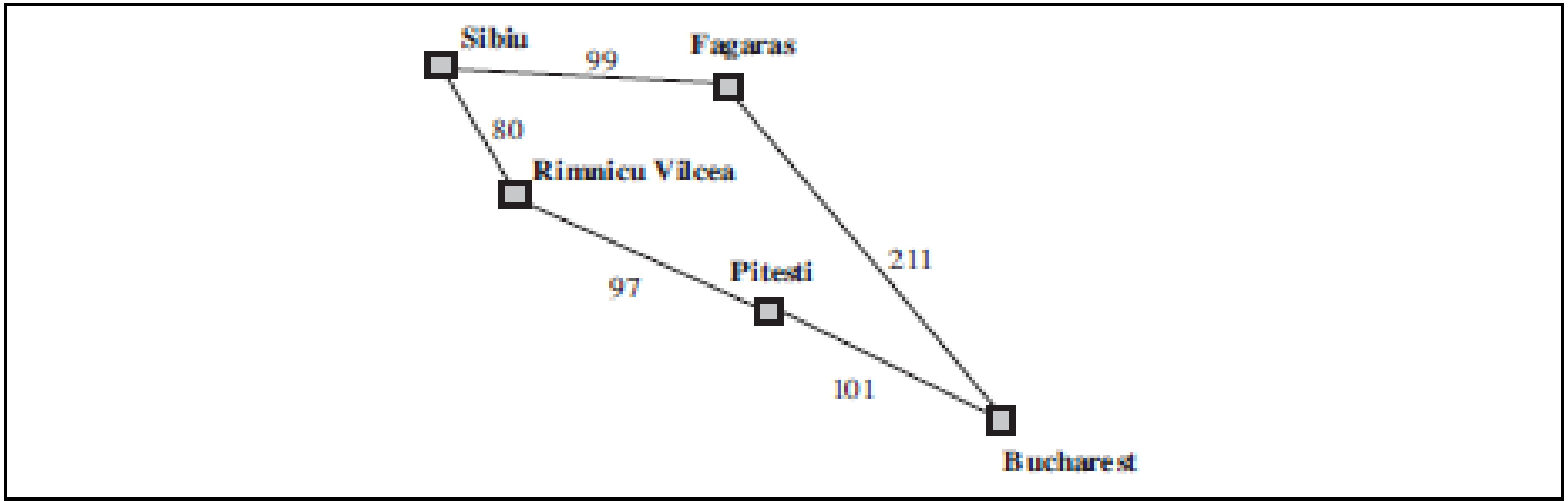
- When all step costs are equal, breadth-first search is optimal because it always expands the *shallowest unexpanded node*.
- Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the *lowest path cost g(n)*
- **uniform-cost search implemented by** priority queue ordered by g.
- In addition to the ordering of the queue by path cost, there are two other significant differences from breadth-first search.

    1. The first is that the goal test is applied to a node when it is *selected for expansion* rather than when it is first generated. The reason is that the first goal node that is *generated* may be on a suboptimal path
    2. The second difference is that a test is added in case a better path is found to a node currently on the frontier.



```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier ← a priority queue ordered by PATH-COST, with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        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
                frontier ← INSERT(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

Figure Uniform-cost search on a graph. The algorithm is identical to the general
graph search algorithm, except for the use of a priority queue and the addition
of an extra check in case a shorter path to a frontier state is discovered. The data structure for
frontier needs to support efficient membership testing, so it should combine the capabilities
of a priority queue and a hash table.



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

➢the problem is to get from Sibiu to Bucharest.

➢The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively.

➢The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97=177. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99+211=310. Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path. to Bucharest with cost 80+97+101= 278.

➢if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.

- *uniform-cost search expands nodes in order of their optimal path cost. Hence, the first goal node selected for expansion must be the optimal solution.*
- Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d.
- Instead, let C^* be the cost of the optimal solution, and assume that every action costs at least Ɛ. Then the algorithm’s worst-case time and space complexity is $O(b^{1+[C^*/Ɛ]})$, which can be much greater than b^d.

Depth Limited Search

- ➢ The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit . That is, nodes at depth are treated as if they have no successors. This approach is called **depth-limited search.**
- ✓ depth limit solves the infinite-path problem(DFS).
- ➢ it also introduces an additional source of incompleteness if we choose $l < d$, that is, the shallowest goal is beyond the depth limit.
- ➢ Depth-limited search will also be nonoptimal if we choose $l > d$. Its time complexity is $O(b^l)$ and its space complexity is $O(bl)$.
- ➢ Depth-first search can be viewed as a special case of depth-limited search with $l=\infty$.

- Sometimes, depth limits can be based on knowledge of the problem.

  Example :

  on the map of Romania there are 20 cities. Therefore, we know that if there is a solution, it must be of length 19 at the longest, so l = 19 is a possible choice

- on the map of Romania , any city can be reached from any other city in at most 9 steps. This number, known as the **diameter of the state space, gives us a better depth limit,** which leads to a more efficient depth-limited search.

- Depth-limited search can be implemented as a simple modification to the general tree or graph-search algorithm.

depth-limited search can terminate with two kinds of failure:

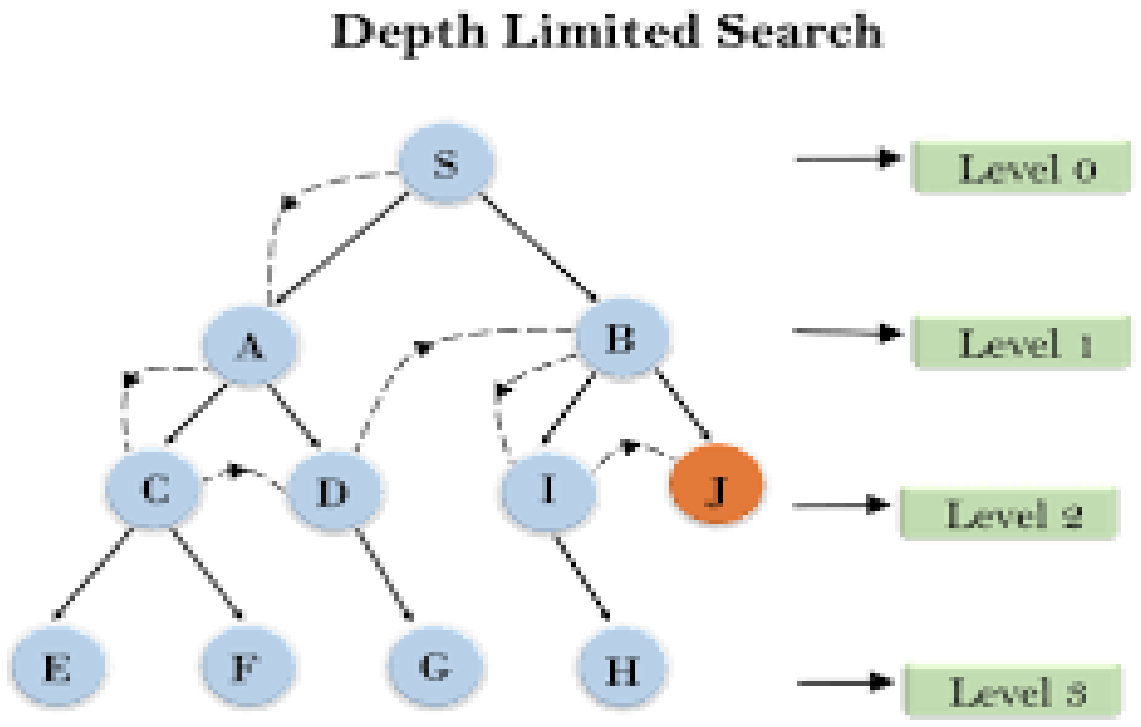
- ✓ the standard failure value indicates no solution
- ✓ the cutoff value indicates no solution within the depth limit.

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure A recursive implementation of depth-limited tree search.

Example :


Depth Limited Search
S
A
B
C
D
I
J
E
F
G
H
Level 0
Level 1
Level 2
Level 3

Iterative deepening depth-first search

- ➢ **Iterative deepening search (or iterative deepening depth-first search) is a general strategy,** 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.
- ➢ Iterative deepening combines the benefits of depth-first and breadth-first search.
- ✓ Like depth-first search, its memory requirements are modest: O(bd) to be precise.
- ✓ Like breadth-first search, it is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node

- Iterative deepening search may seem wasteful because states are generated multiple times. It turns out this is not too costly.
- The reason is that in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times.
- In an iterative deepening search, the nodes on the bottom level (depth d) are generated once, those on the next-to-bottom level are generated twice, and so on, up to the children of the root, which are generated d times. So the total number of nodes generated in the worst case is

$$N(IDS) = (d)b + (d - 1)b^2 + \cdot \quad \cdot \quad \cdot + (1)b^d$$

which gives a time complexity of $O(b^d)$

```
➢ function ITERATIVE-DEEPENING-SEARCH(problem)
  returns a solution, or failure
  for depth = 0 to∞ do
  result ←DEPTH-LIMITED-SEARCH(problem, depth)
  if result = cutoff then return result
```

Figure : The iterative deepening search algorithm, which repeatedly applies depth limited search with increasing limits. It terminates when a solution is found or if the depthlimited search returns failure, meaning that no solution exists.

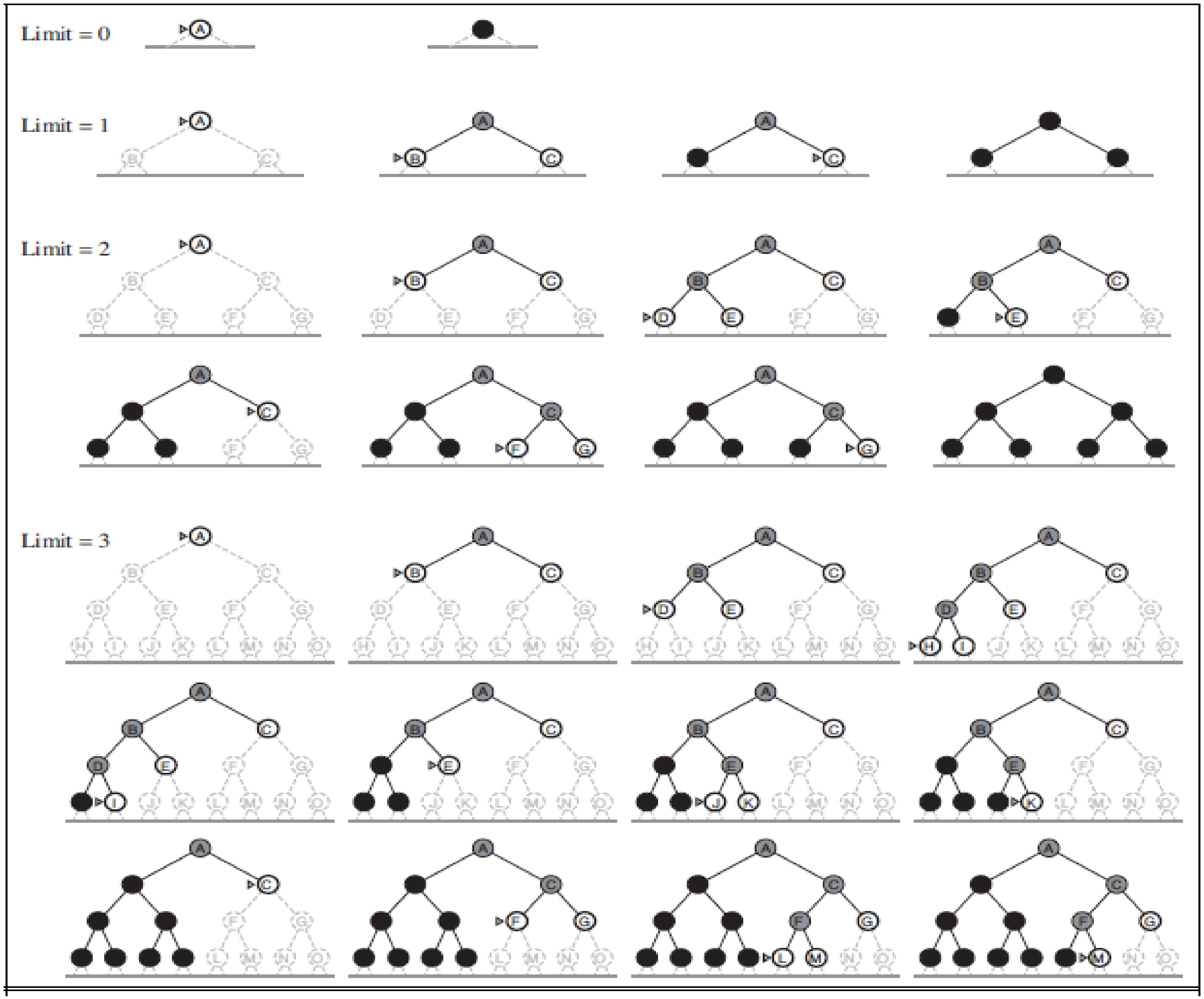


Figure : Four iterations of iterative deepening search on a binary tree.

informed Search Algorithm

- informed Search Algorithm uses domain knowledge, the problem information is available which can guide the search.
- Informed Search strategies can find a solution more efficient than an uninformed . Informed Search is also called as Heuristic search
- A Heuristic search is a way which might not always be guaranteed the best solution but guaranteed to find a good solution in reasonable time
- Heuristic search solve the complex problems which can not be solved in another way.

Informed Search Algorithm

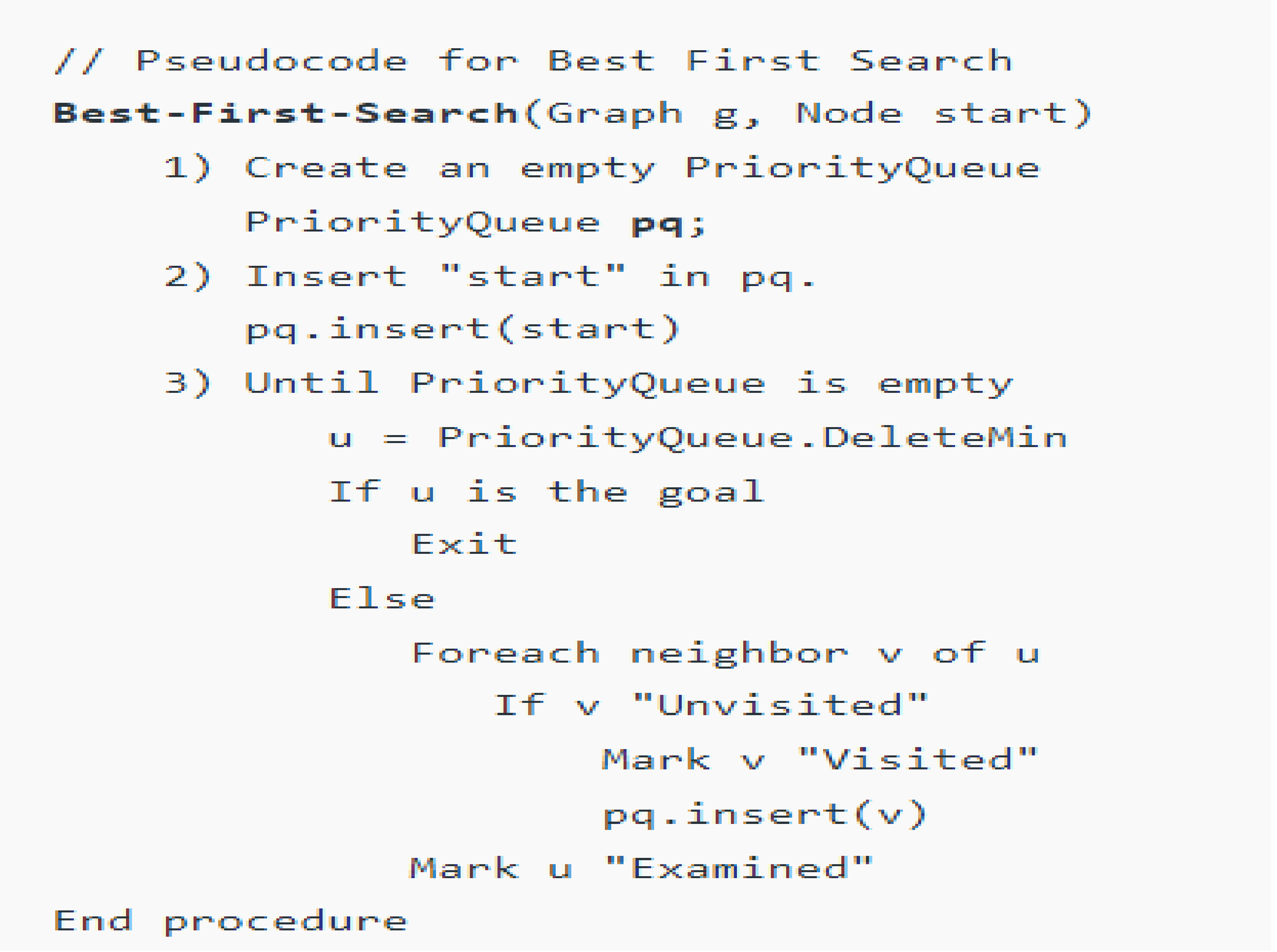
1. Best First Search
2. A* Search
3. AO* Algorithm
4. Problem Reduction
5. Hill Climbing

1.Best First Search

- In BFS and DFS, when we are at a node, we can consider any of the adjacent as the next node. So both BFS and DFS blindly explore paths without considering any cost function.

➢ **Implementation of Best First Search:**

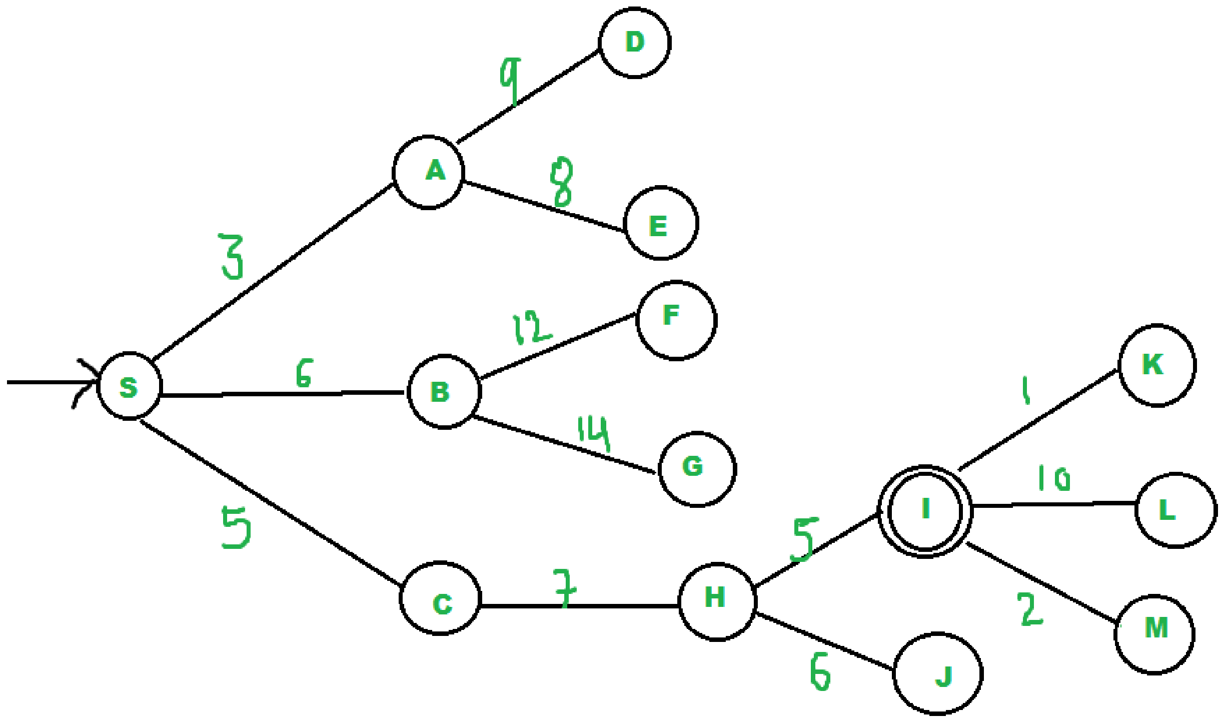
- We use a priority queue or heap to store the costs of nodes that have the lowest evaluation function value. So the implementation is a variation of BFS, we just need to change Queue to PriorityQueue.



```
// Pseudocode for Best First Search
Best-First-Search(Graph g, Node start)
    1) Create an empty PriorityQueue
       PriorityQueue pq;
    2) Insert "start" in pq.
       pq.insert(start)
    3) Until PriorityQueue is empty
          u = PriorityQueue.DeleteMin
          If u is the goal
             Exit
          Else
             Foreach neighbor v of u
                If v "Unvisited"
                    Mark v "Visited"
                    pq.insert(v)
             Mark u "Examined"
End procedure
```

- We use a priority queue or heap to store the costs of nodes that have the lowest evaluation function value. So the implementation is a variation of BFS, we just need to change Queue to Priority Queue.
- Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function, f(n).**
- The evaluation function is construed as a cost estimate, so the node with the *lowest evaluation is expanded first.*

node	H(n)
S→I	41
A→L	21
B→L	26
C→L	24
D→L	30
E→L	28
F→L	32
G→L	35
H→L	27
I→I	0
J→	
K-→	
L→	
M→	
N→	



- *We start from source "S" and search for goal "I" using given costs and Best First search.*

- priority queue *initially contains S*
  - *We remove S from* priority queue *and process unvisited neighbors of S to* priority queue *.*
  - priority queue *now contains {A, C, B} (C is put before B because C has lesser cost)*

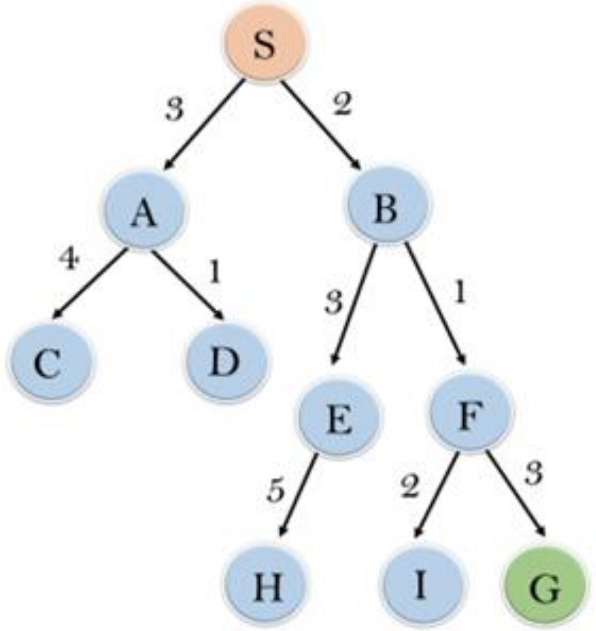
- *We remove A from* priority queue *and process unvisited neighbors of A to* priority queue *.* priority queue *now contains {C, B, E, D}*

- *We remove C from* priority queue *and process unvisited neighbors of C to* priority queue *.*
  - priority queue *now contains {B, H, E, D}*

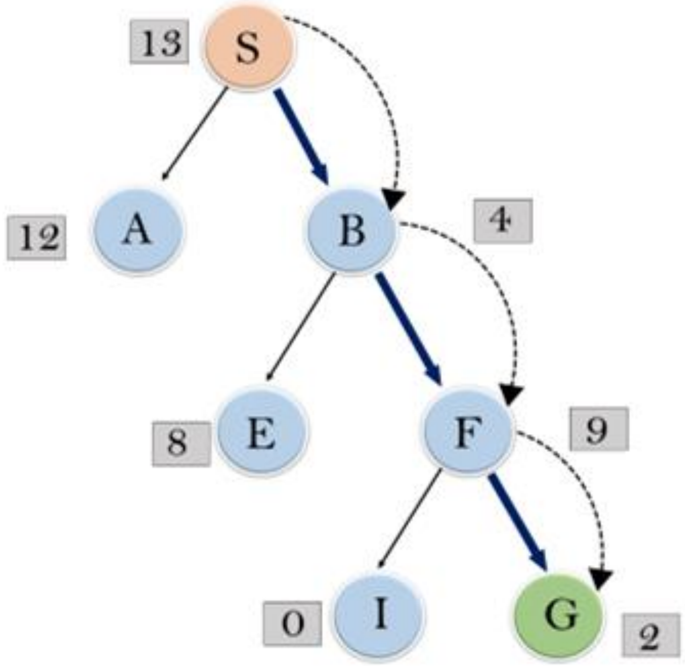
- *We remove B from* priority queue *and process unvisited neighbors of B to* priority queue *.*
  - priority queue *now contains {H, E, D, F, G}*
- *We remove H from* priority queue *.*
- *Since our goal "I" is a neighbor of H, we return.*

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.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.



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



Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.

- **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.
- Com**plete: Gr**eedy best-first search is also incomplete, even if the given state space is finite.
- Opt**imal: Gr**eedy best first search algorithm is not optimal.

A* Algorithm

- A* Algorithm is one of the best and popular techniques used for path finding and graph traversals.
- A lot of games and web-based maps use this algorithm for finding the shortest path efficiently.

➢ **Working-**

A* Algorithm works as-

- It maintains a tree of paths originating at the start node.
- It extends those paths one edge at a time.
- It continues until its termination criterion is satisfied.

A* Algorithm extends the path that minimizes the following function-

$$\mathbf{f(n) = g(n) + h(n)}$$

Here,

‘n’ is the last node on the path

g(n) is the cost of the path from start node to node ‘n’

h(n) is a heuristic function that estimates cost of the cheapest path from node ‘n’ to the goal node

Algorithm-

- The implementation of A* Algorithm involves maintaining two lists- OPEN and CLOSED.
- OPEN contains those nodes that have been evaluated by the heuristic function but have not been expanded into successors yet.
- CLOSED contains those nodes that have already been visited.

Step-01:

- Define a list OPEN.
- Initially, OPEN consists of a single node, the start node S.

Step-02:

- If the list is empty, return failure and exit.

Step-03:

- Remove node n with the smallest value of f(n) from OPEN and move it to list CLOSED.
- If node n is a goal state, return success and exit.

Step-04:

- Expand node n.

Step-05:

- If any successor to n is the goal node, return success and the solution by tracing the path from goal node to S.
- Otherwise, go to Step-06.

Step-06:

- For each successor node,
- Apply the evaluation function f to the node.
- If the node has not been in either list, add it to OPEN.

Step-07:

- Go back to Step-02.

➢ Problem-01:

Given an initial state of a 8-puzzle problem and final state to be reached?

2	8	3
1	6	4
7		5

Initial State

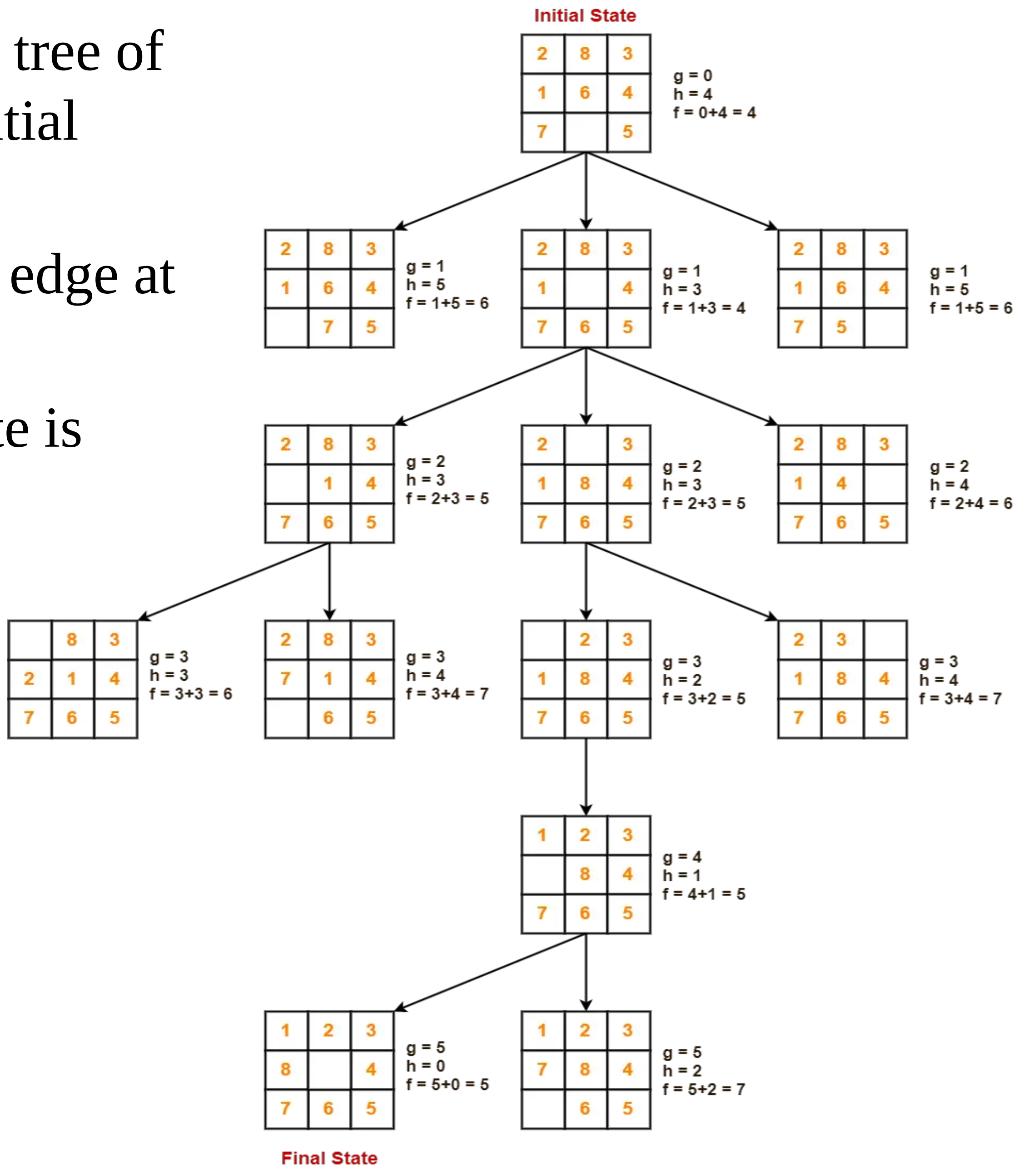
1	2	3
8		4
7	6	5

Final State

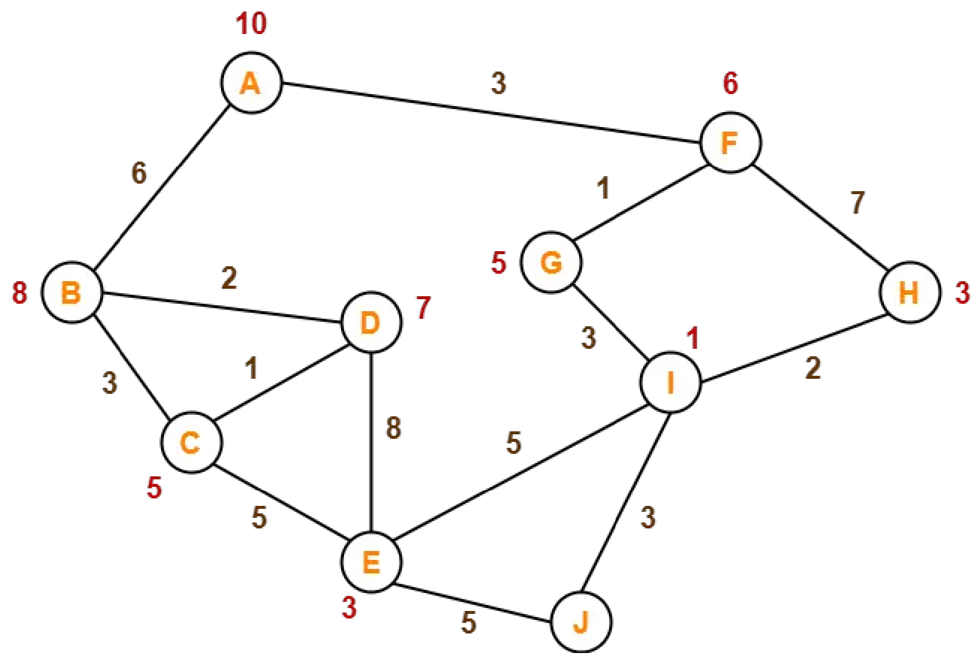
Find the most cost-effective path to reach the final state from initial state using A* Algorithm.
Consider g(n) = Depth of node and h(n) = Number of misplaced tiles.

Solution :

- A* Algorithm maintains a tree of paths originating at the initial state.
- It extends those paths one edge at a time.
- It continues until final state is reached.



Problem-02:
Consider the following graph-



The numbers written on edges represent the distance between the nodes.
The numbers written on nodes represent the heuristic value.
Find the most cost-effective path to reach from start state A to final state J using A* Algorithm.

- Solution -

Step-01:

We start with node A.

Node B and Node F can be reached from node A.

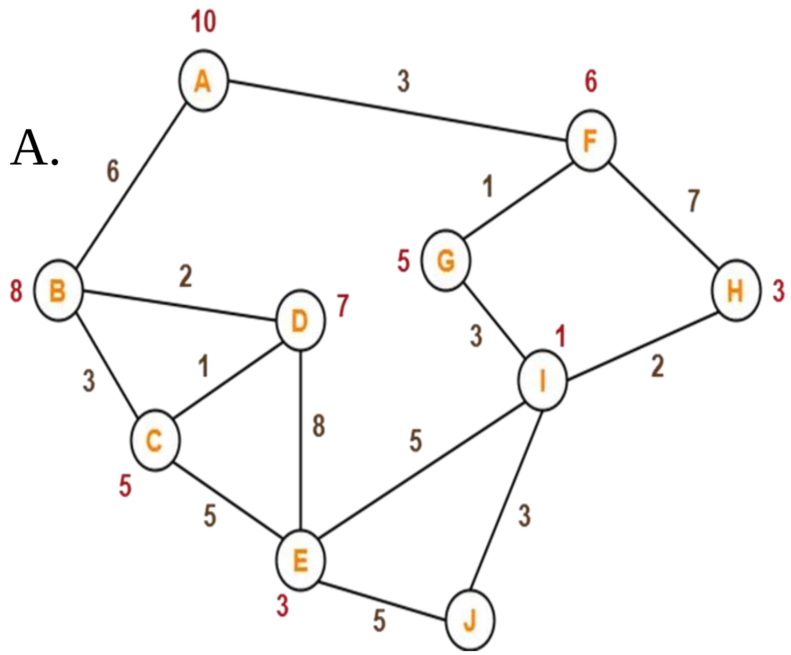
A* Algorithm calculates f(B) and f(F).

f(B) = 6 + 8 = 14

f(F) = 3 + 6 = 9

Since f(F) < f(B), so it decides to go to node F.

Path - A → F



Step-02:

Node G and Node H can be reached from node F.

A* Algorithm calculates f(G) and f(H).

f(G) = (3+1) + 5 = 9

f(H) = (3+7) + 3 = 13

Since f(G) < f(H), so it decides to go to node G.

- **Path- A → F → G**

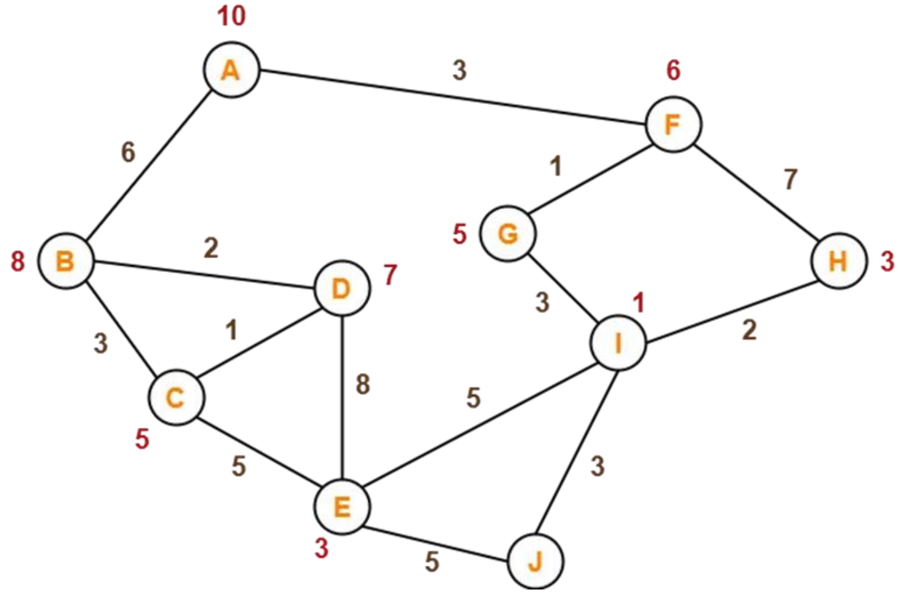
- **Step-03:**

  Node I can be reached from node G.

  A* Algorithm calculates f(I).

  f(I) = (3+1+3) + 1 = 8

  It decides to go to node I.
- **Path- A → F → G → I**



Step-04:

Node E, Node H and Node J can be reached from node I.

A* Algorithm calculates f(E), f(H) and f(J).

f(E) = (3+1+3+5) + 3 = 15

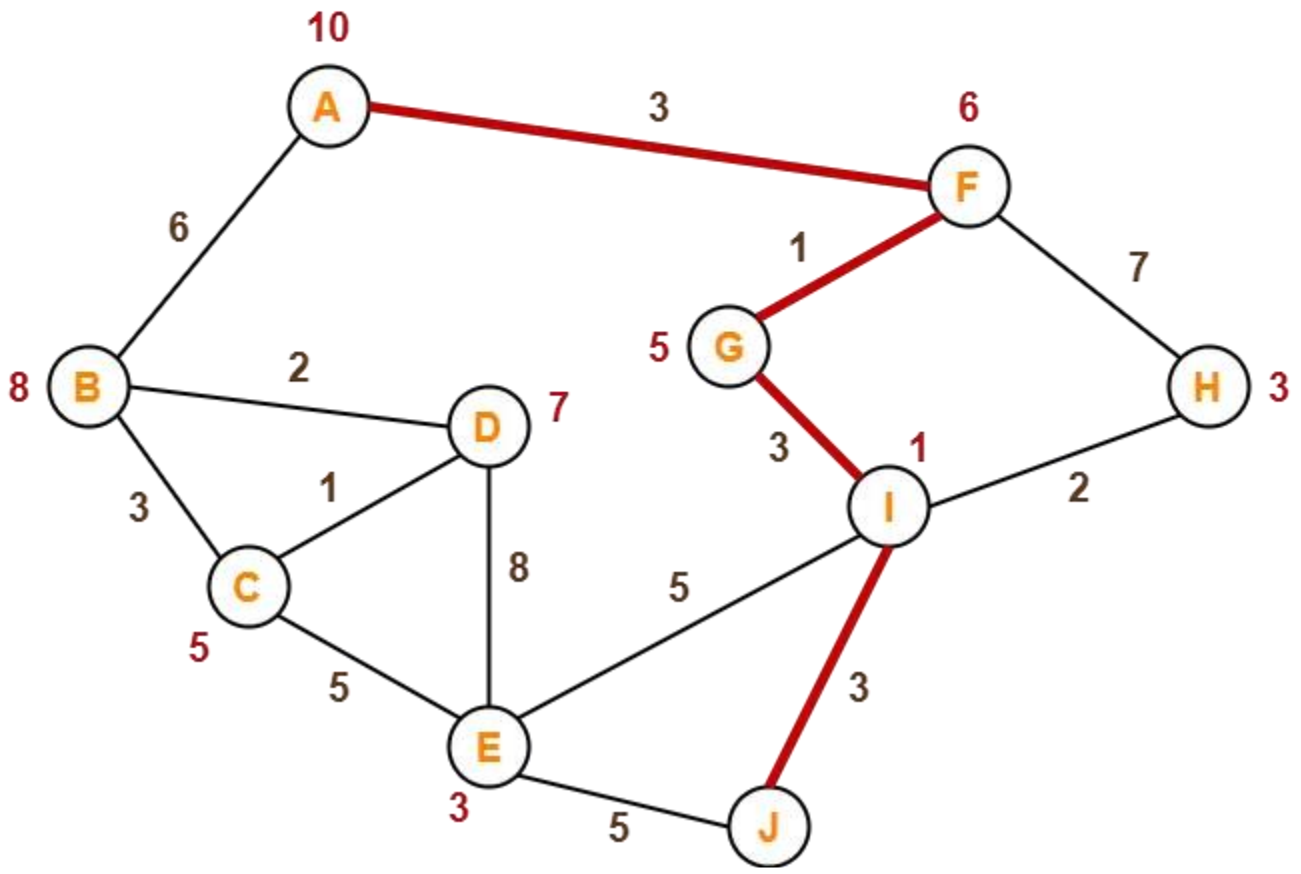
f(H) = (3+1+3+2) + 3 = 12

f(J) = (3+1+3+3) + 0 = 10

Since f(J) is least, so it decides to go to node J.

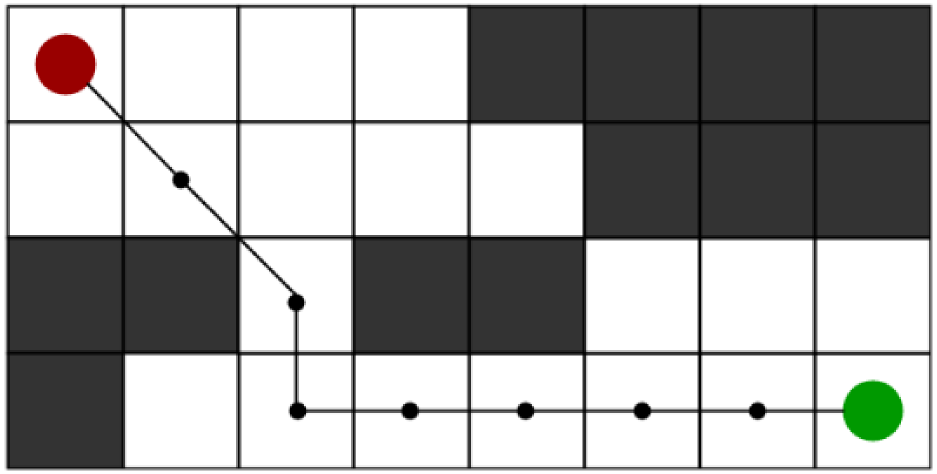
- **Path- A → F → G → I → J**

- This is the required shortest path from node A to node J.



A* Search Algorithm

- We can consider a 2D Grid having several obstacles and we start from a source cell (colored red below) to reach towards a goal cell (colored green below)



- A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.
- it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently
- Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible.

- A* Search Algorithm does is that at each step it picks the node according to a value-‘**f**’ , which is a parameter equal to the sum of two other parameters – ‘**g**’ and ‘**h**’.

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

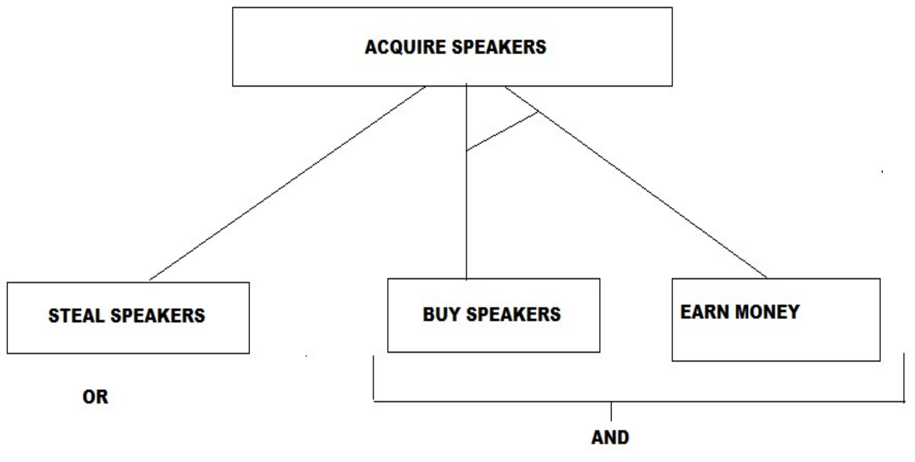
and process that node/cell. We define ‘**g**’ and ‘**h**’ as simply as possible below:

g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.

h = the estimated movement cost to move from that given square on the grid to the final destination.

AO* algorithm

- The AO* algorithm is a knowledge-based search technique, meaning the start state and the goal state is already defined , and the best path is found using heuristics.
- The time complexity of the algorithm is significantly reduced due to the informed search technique. Compared to the A* algorithm , AO* algorithm is very efficient in searching the AND-OR trees very efficiently.
- AO* algorithm is a best first search algorithm.
- AO* algorithm uses the concept of AND-OR graphs to decompose any complex problem given into smaller set of problems which are further solved.



A SIMPLE AND-OR GRAPH

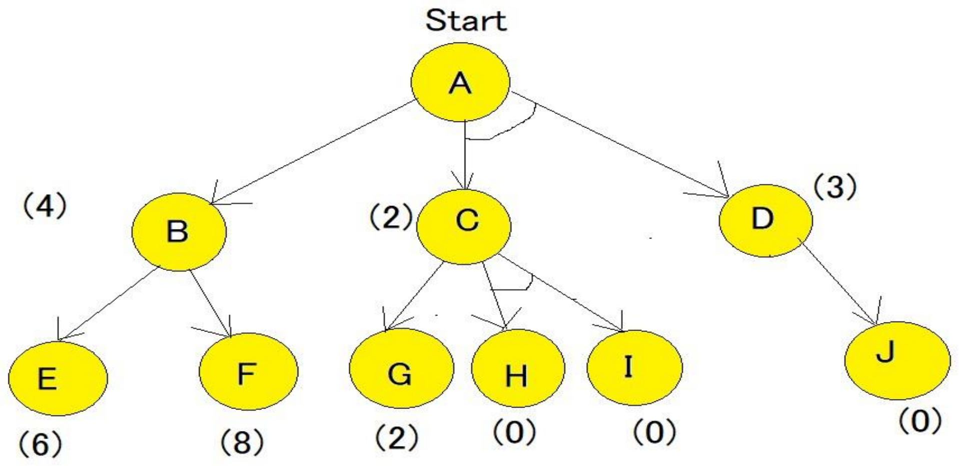
Working of AO algorithm:

The AO* algorithm works on the formula given below :

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

where,

- g(n): The actual cost of traversal from initial state to the current state.
- h(n): The estimated cost of traversal from the current state to the goal state.
- f(n): The actual cost of traversal from the initial state to the goal state.

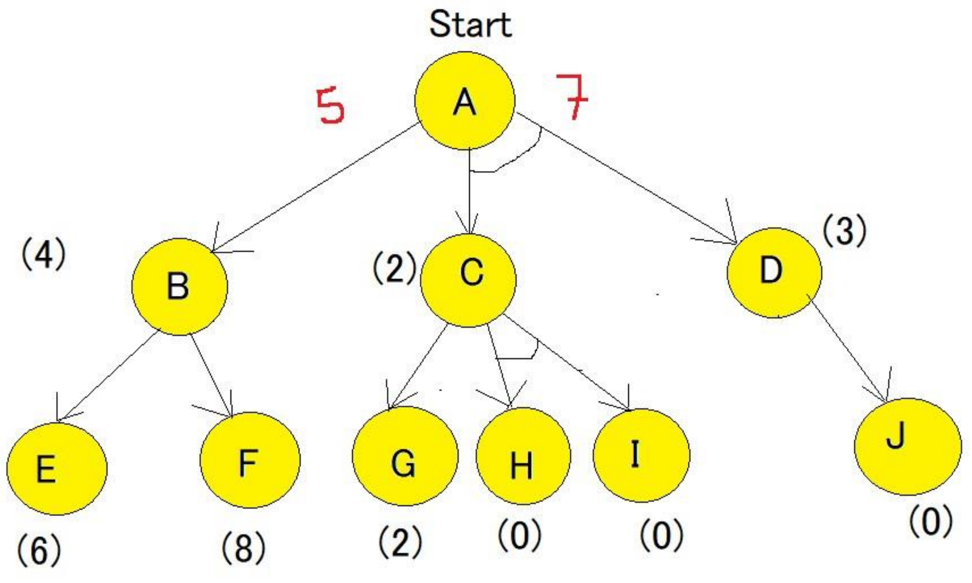


Here, in the above example all numbers in brackets are the heuristic value i.e h(n). Each edge is considered to have a value of 1 by default.

- **Step-1**
  Starting from node A, we first calculate the best path.
  f(A-B) = g(B) + h(B) = 1+4= 5 ,

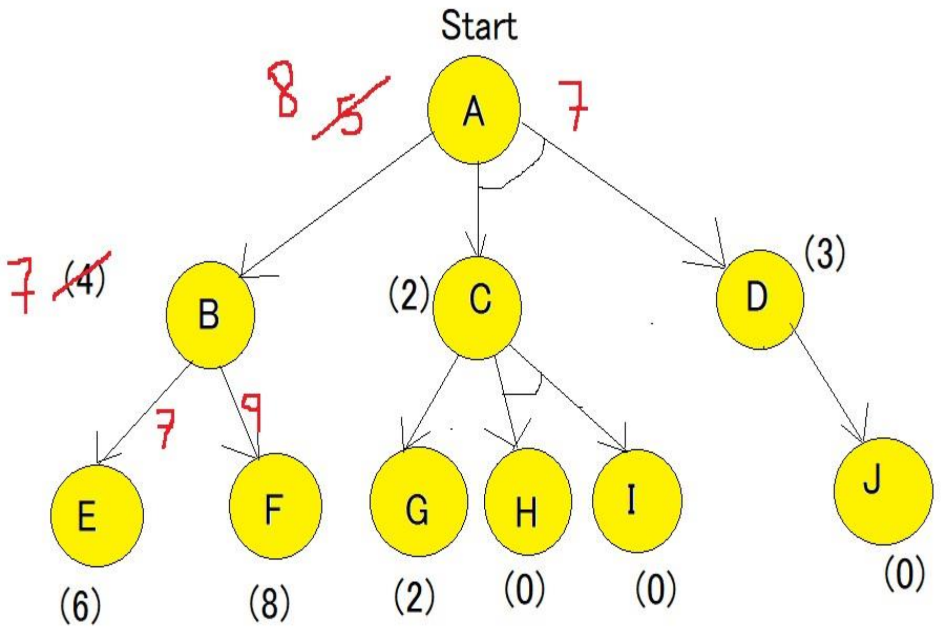
  where 1 is the default cost value of travelling from A to B and 4 is the estimated cost from B to Goal state.
- f(A-C-D) = g(C) + h(C) + g(D) + h(D) = 1+2+1+3 = 7 ,

  here we are calculating the path cost as both C and D because they have the AND-Arc. The default cost value of travelling from A-C is 1, and from A-D is 1, but the heuristic value given for C and D are 2 and 3 respectively hence making the cost as 7

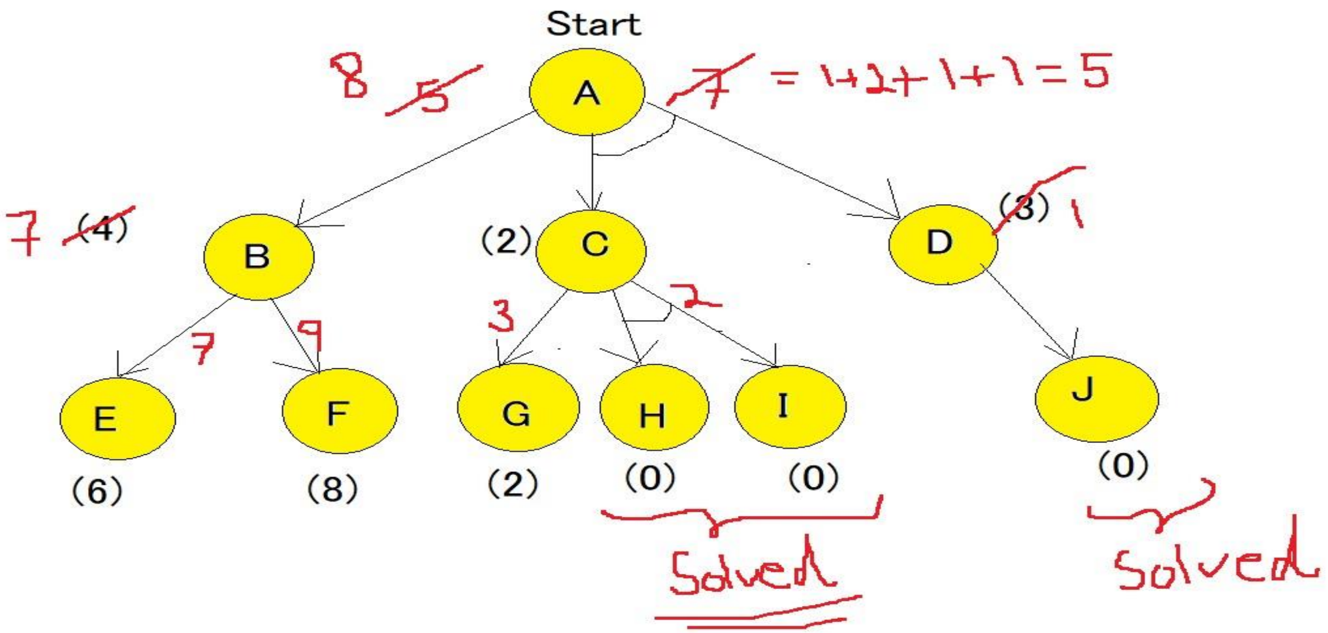


The minimum cost path is chosen i.e A-B.

- **Step-2**
  Using the same formula as step-1, the path is now calculated from the B node,
  f(B-E) = 1 + 6 = 7.
  f(B-F) = 1 + 8 = 9
  Hence, the B-E path has lesser cost. Now the heuristics have to be updated since there is a difference between actual and heuristic value of B. The minimum cost path is chosen and is updated as the heuristic , in our case the value is 7. And because of change in heuristic of B there is also change in heuristic of A which is to be calculated again.
  f(A-B) = g(B) + updated((h(B)) = 1+7=8



- **Step-3**
  Comparing path of f(A-B) and f(A-C-D) it is seen that f(A-C-D) is smaller. Hence f(A-C-D) needs to be explored.
  Now the current node becomes C node and the cost of the path is calculated,
  f(C-G) = 1+2 = 3
  f(C-H-I) = 1+0+1+0 = 2
  f(C-H-I) is chosen as minimum cost path, also there is no change in heuristic since it matches the actual cost. Heuristic of path of H and I are 0 and hence they are solved, but Path A-D also needs to be calculated , since it has an AND-arc.
  f(D-J) = 1+0 = 1, hence heuristic of D needs to be updated to 1. And finally the f(A-C-D) needs to be updated.
  f(A-C-D) = g(C) + h(C) + g(D) + updated((h(D)) = 1+2+1+1 =5.



As we can see that the solved path is f(A-C-D).

- •A* algorithm provides with the optimal solution, whereas AO* stops when it finds any solution.
- •AO* algorithm requires lesser memory compared to A* algorithm.
- •AO* algorithm doesn't go into infinite loop whereas the A* algorithm can go into an infinite loop.

Problem Reduction

- A solution to a problem can be obtained by decomposing it into smaller sub-problems. Each of this sub-problem can then be solved to get its sub-solution. These sub-solutions can then be recombined to get a solution as a whole. That is called is Problem Reduction.
- This method generates arc which is called as AND arcs. One AND arc may point to any number of successor nodes, all of which must be solved for an arc to point to a solution.

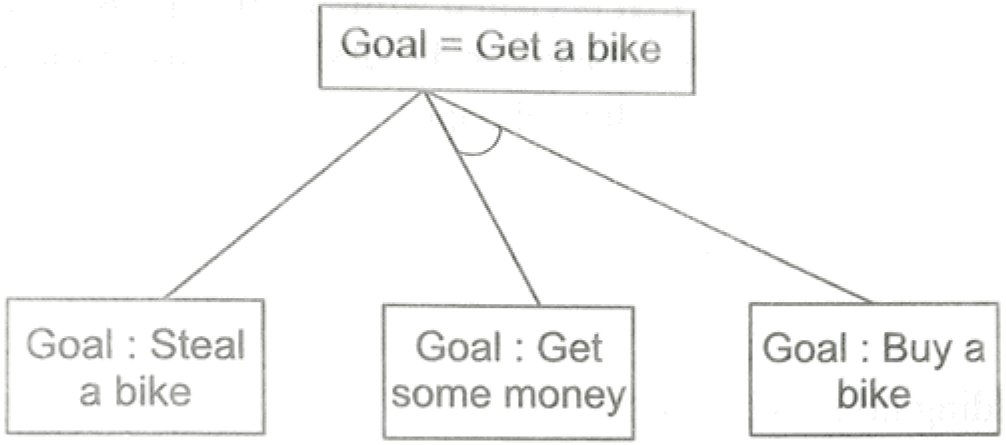


Fig: AND / OR Graph

Hill Climbing

- Hill Climbing (local search, greed approach, no back tracking) is a heuristic search used for mathematical optimization problems in the field of Artificial Intelligence
- **Types of Hill Climbing**

1. Simple Hill climbing

It examines the neighboring nodes one by one and selects the first neighboring node which optimizes the current cost as the next node.

- **Algorithm for Simple Hill climbing :**

- Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make the initial state as the current state.
- Loop until the solution state is found or there are no new operators present which can be applied to the current state.
- Select a state that has not been yet applied to the current state and apply it to produce a new state.
- Perform these to evaluate the new state.
- If the current state is a goal state, then stop and return success.
- If it is better than the current state, then make it the current state and proceed further.
- If it is not better than the current state, then continue in the loop until a solution is found.
- Exit from the function.

Steepest-Ascent Hill climbing

It first examines all the neighboring nodes and then selects the node closest to the solution state as of the next node.

Algorithm for Steepest Ascent Hill climbing :

- Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make the initial state as the current state.
- Repeat these steps until a solution is found or the current state does not change
- Select a state that has not been yet applied to the current state.
- Initialize a new 'best state' equal to the current state and apply it to produce a new state.
- Perform these to evaluate the new state
- If the current state is a goal state, then stop and return success.
- If it is better than the best state, then make it the best state else continue the loop with another new state.
- Make the best state as the current state and go to Step 2 of the second point.
- Exit from the function.

Stochastic hill climbing

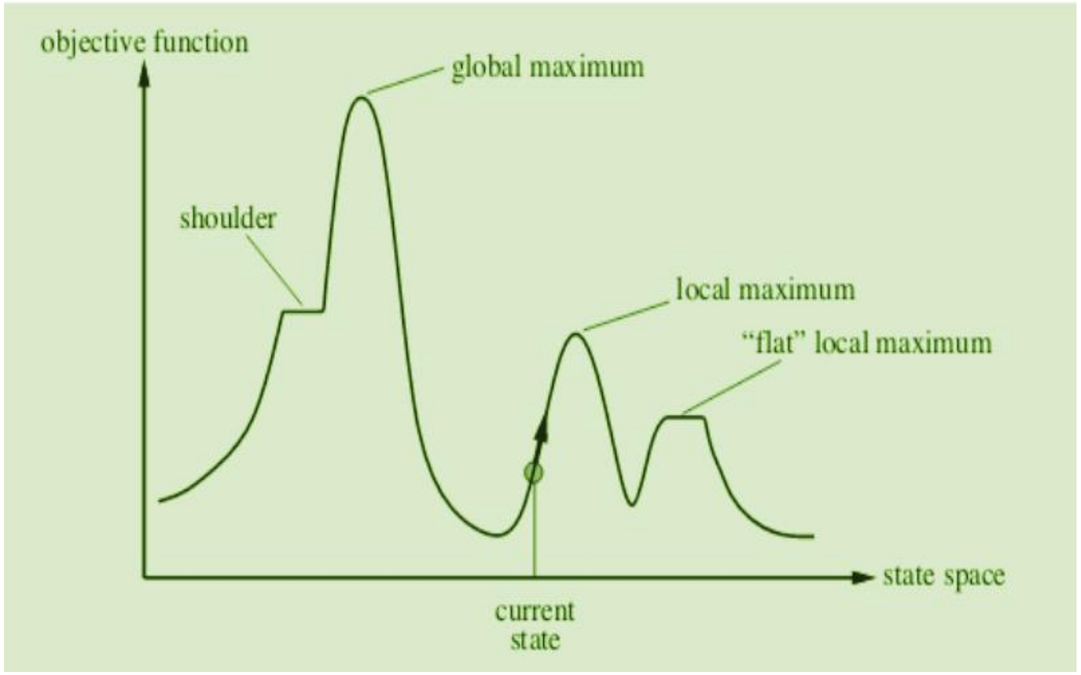
- It does not examine all the neighboring nodes before deciding which node to select. It just selects a neighboring node at random and decides (based on the amount of improvement in that neighbor) whether to move to that neighbor or to examine another.

- ➢ Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make the initial state the current state.
- ➢ Repeat these steps until a solution is found or the current state does not change.
- ➢ Select a state that has not been yet applied to the current state.
- ➢ Apply the successor function to the current state and generate all the neighbor states.
- ➢ Among the generated neighbor states which are better than the current state choose a state randomly (or based on some probability function).
- ➢ If the chosen state is the goal state, then return success, else make it the current state and repeat step 2 of the second point.
- ➢ Exit from the function.

State Space diagram for Hill Climbing

- X-axis: denotes the state space i.e. states or configuration our algorithm may reach.
- Y-axis: denotes the values of objective function corresponding to a particular state.

The best solution will be a state space where the objective function has a maximum value(global maximum).



Beyond Classical Search : Local Search Algorithms and Optimization Problems

- The informed and uninformed search expands the nodes systematically in two ways:

1. keeping different paths in the memory and
2. selecting the best suitable path,

- A local search algorithm completes its task by traversing on a single current node rather than multiple paths and following the neighbors of that node generally.

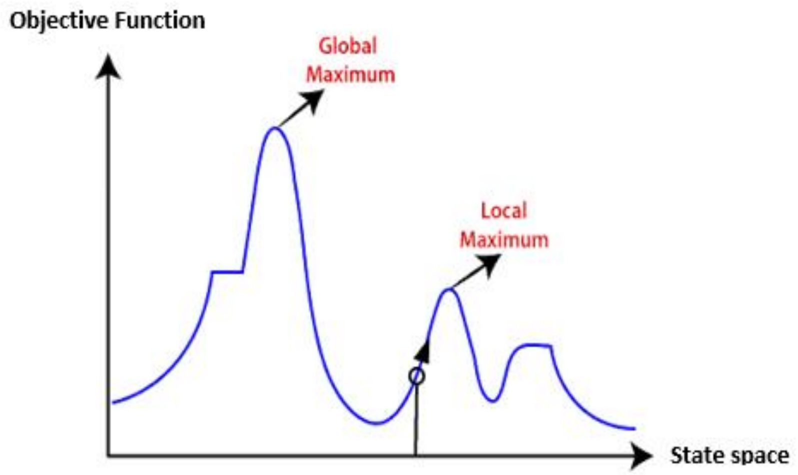
Advantages of Local Search Algorithms

- Local search algorithms use a very little or constant amount of memory as they operate only on a single path.
- Most often, they find a reasonable solution in large or infinite state spaces where the classical or systematic algorithms do not work.

- The local search algorithm works for pure optimized problems. A pure optimization problem is one where all the nodes can give a solution. But the target is to find the best state out of all according to the objective function.
- the pure optimization problem fails to find high-quality solutions to reach the goal state from the current state.

➢ Working of a Local search algorithm

example



A one-dimensional state-space landscape in which elevation corresponds to the objective function

The local search algorithm explores the above landscape by finding the following two points:

Global Minimum: If the elevation corresponds to the cost, then the task is to find the lowest valley, which is known as Global Minimum.
Global Maxima: If the elevation corresponds to an objective function, then it finds the highest peak which is called as Global Maxima.

Heuristics function

- The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.
- **Heuristics function:** Heuristic is a function which is used in Informed Search, and it finds the most promising path.
- It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal.
- The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time.
- Heuristic function estimates how close a state is to the goal.
- It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

- Admissibility of the heuristic function is given as:

  $h(n) <= h^*(n)$

Here h(n) is heuristic cost, and h*(n) is the estimated cost.

Hence heuristic cost should be less than or equal to the estimated cost.

➢ Pure Heuristic Search:

- Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.
- On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues unit a goal state is found.

In the informed search we will discuss two main algorithms which are given below:

- Best First Search Algorithm(Greedy search)
- A* Search Algorithm

Genetic Algorithms

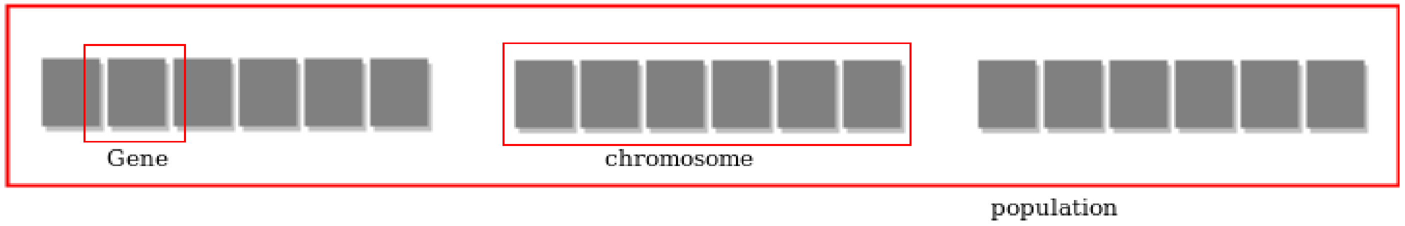
- These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space.
- **They are commonly used to generate high-quality solutions for optimization problems and search problems.**
- they simulate "survival of the fittest" among individual of consecutive generation for solving a problem.
- **Each generation consist of a population of individuals** and each individual represents a point in search space and possible solution. Each individual is represented as a string of character/integer/float/bits
- This string is analogous to the Chromosome.
- Genetic algorithms are based on an analogy with genetic structure and behaviour of chromosomes of the population. Following is the foundation of GAs based on this analogy –

1. Individual in population compete for resources and mate
2. Those individuals who are successful (fittest) then mate to create more offspring than others
3. Genes from "fittest" parent propagate throughout the generation, that is sometimes parents create offspring which is better than either parent.

4.Thus each successive generation is more suited for their environment.

- **Search space**

- The population of individuals are maintained within search space.
- Each individual represents a solution in search space for given problem. Each individual is coded as a finite length vector (analogous to chromosome) of components.
- These variable components are analogous to Genes. Thus a chromosome (individual) is composed of several genes (variable components).



- **Fitness Score**

A Fitness Score is given to each individual which shows the ability of an individual to "compete".
The individual having optimal fitness score (or near optimal) are sought.
The GAs maintains the population of n individuals (chromosome/solutions) along with their fitness scores.
The individuals having better fitness scores are given more chance to reproduce than others. The individuals with better fitness scores are selected who mate and produce better offspring by combining chromosomes of parents.

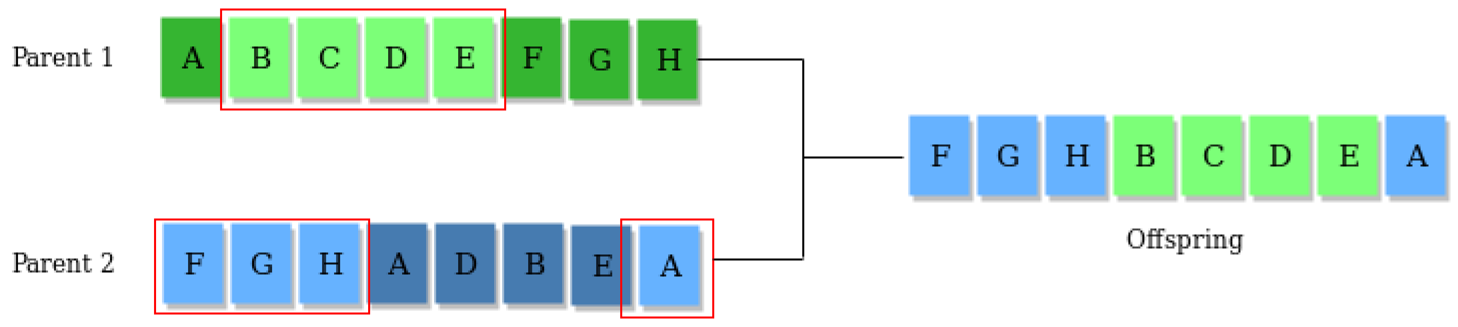
- The population size is static so the room has to be created for new arrivals.
- Each new generation has on average more "better genes" than the individual (solution) of previous generations. Thus each new generations have better **"partial solutions"** than previous generations

➢ **Operators of Genetic Algorithms**

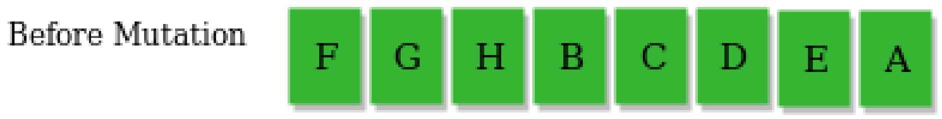
Once the initial generation is created, the algorithm evolves the generation using following operators –

1) Selection Operator: The idea is to give preference to the individuals with good fitness scores and allow them to pass their genes to successive generations.

2) Crossover Operator: This represents mating between individuals. Two individuals are selected using selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged thus creating a completely new individual (offspring). For example –



3) Mutation Operator: The key idea is to insert random genes in offspring to maintain the diversity in the population to avoid premature convergence. For example –



The whole algorithm can be summarized as –

1)Randomly initialize populations p
2) Determine fitness of population
3) Until convergence repeat:
a) Select parents from population
b) Crossover and generate new population
c) Perform mutation on new population
d) Calculate fitness for new population

Example problem and solution using Genetic Algorithms

- ➢ Why use Genetic Algorithms

- They are Robust
- Provide optimisation over large space state.
- Unlike traditional AI, they do not break on slight change in input or presence of noise
- Application of Genetic Algorithms

- ➢ Genetic algorithms have many applications, some of them are –

- Recurrent Neural Network
- Mutation testing
- Code breaking
- Filtering and signal processing
- Learning fuzzy rule base etc