PROBLEM-SOLVING AGENT

PROBLEM-SOLVING AGENT

The problem-solving agent performs **precisely** by **defining problems** and its **several solutions**.

- According to **psychology**, "**a problem-solving refers to a state where we wish to reach to a definite goal from a present state or condition.**"
- According to computer science, a problem-solving is a part of artificial intelligence which encompasses a number of techniques such as algorithms, heuristics to solve a problem.

STEPS PERFORMED BY PROBLEM-SOLVING AGENT

Goal Formulation: It is the first and simplest step in problem-solving.

- It organizes the steps/sequence required to formulate one goal out of multiple goals as well as

actions to achieve that goal.

- Goal formulation is based on the current situation and the agent's performance measure (discussed below).

Problem Formulation: It is the most **important** step **of problem-solving** which decides what actions should be taken to achieve the formulated goal.

There are following five components involved in problem formulation:

- **Initial State:** It is the starting state or initial step of the agent towards its goal.
- **Actions:** It is the description of the possible actions available to the agent.
- **Transition Model:** It describes what each action does.
- **Goal Test:** It determines if the given state is a goal state.
- **Path cost:** It assigns a numeric cost to each path that follows the goal. The problem-solving

agent selects a cost function, which reflects its performance measure. Remember, **an optimal solution has the lowest path cost among all the solutions.**

- **Search:** It identifies all the best possible sequence of actions to reach the goal state from the current state. It takes a problem as an input and returns solution as its output.
- **Solution:** It finds the best algorithm out of various algorithms, which may be proven as the best optimal solution.
- **Execution:** It executes the best optimal solution from the searching algorithms to reach the goal state from the current state.

NOTE: **Initial state, actions**, and **transition model** together define the **state-space** of the problem implicitly. State-space of a problem is a set of all states which can be reached from the initial state followed by any sequence of actions. The state-space forms a directed map or graph where nodes are

the states, links between the nodes are actions, and the path is a sequence of states connected by the sequence of actions.

```
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
```

A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions

EXAMPLE PROBLEMS

Basically, there are two types of problem approaches:

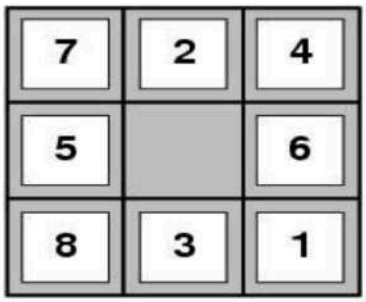
- **Toy Problem:** It is a concise and exact description of the problem which is used by the researchers to compare the performance of algorithms.
- **Real-world Problem:** It is real-world based problems which require solutions. Unlike a toy problem, it does not depend on descriptions, but we can have a general formulation of the problem.

SOME TOY PROBLEMS

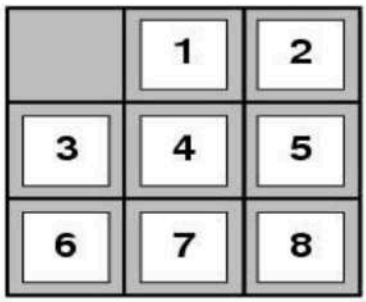
8 Puzzle Problem: Here, we have a 3×3 matrix with movable tiles numbered from 1 to 8 with a blank space. The tile adjacent to the blank space can slide into that space. The objective is to

reach a specified goal state similar to the goal state, as shown in the below figure.

SOME TOY PROBLEMS



Start State



Goal State

THE PROBLEM FORMULATION IS AS FOLLOWS:

- **States:** It describes the location of each numbered tiles and the blank tile.
- **Initial State:** We can start from any state as the initial state.
- **Actions:** Here, actions of the blank space is defined, i.e., either **left, right, up or down**
- **Transition Model:** It returns the resulting state as per the given state and actions.

- **Goal test:** It identifies whether we have reached the correct goal-state.
- **Path cost:** The path cost is the number of steps in the path where the cost of each step is 1.

EIGHT PUZZLE PROBLEM

- We also know the **eight-puzzle problem** by the name of **N puzzle problem** or **sliding puzzle problem.**
- **N-puzzle** that consists of N tiles (N+1 titles with an empty tile) where N can be 8, 15, 24 and so on.
- In our example **N = 8.** (that is **square root of (8+1) = 3 rows and 3 columns**).
- In the same way, if we have N = 15, 24 in this way, then they have Row and columns as follow **(square root of (N+1) rows and square root of (N+1) columns).**
- That is if **N=15** than number of rows and columns= 4, and if **N= 24** number of rows and columns= 5.

- So, basically in these types of problems we have given a **initial state or initial configuration (Start state) and a Goal state or Goal Configuration.**

Here We are solving a problem of 8 puzzle that is a 3x3 matrix.

Initial state

1	2	3
	4	6
7	5	8

Goal state

1	2	3
4	5	6
7	8	

The puzzle can be solved by moving the tiles one by one in the single empty space and thus achieving the Goal state.

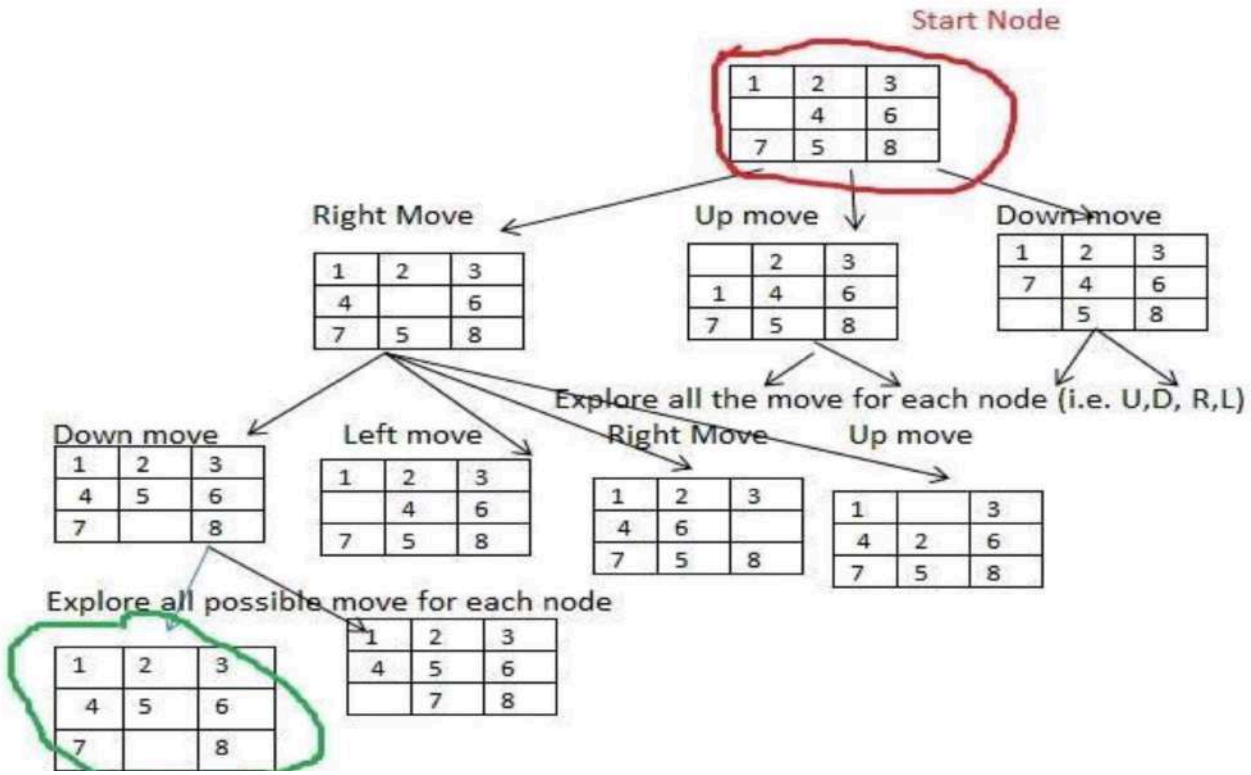
Rules of solving puzzle

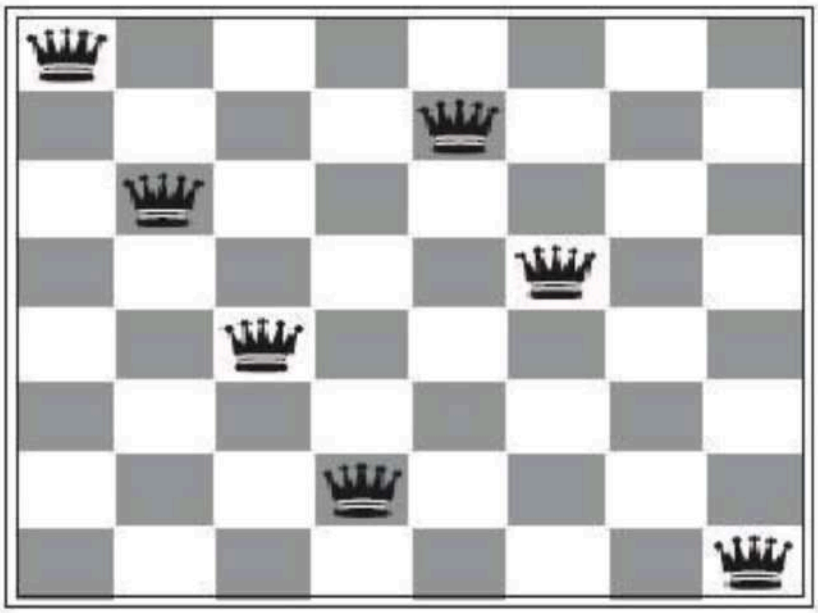
Instead of moving the tiles in the empty space we can visualize moving the empty space in place of the tile.

The empty space can only **move in four directions** (Movement of empty space)

Up Down Right or Left

The empty space **cannot move diagonally** and can take **only one step at a time**.


Start Node
Right Move
Up move
Down move
Explore all the move for each node (i.e. U,D, R,L)
Down move
Left move
Right Move
Up move
Explore all possible move for each node



Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

For this problem, there are two main kinds of formulation

- **Incremental formulation:** It starts from an empty state where the operator augments a queen at each step.

Following steps are involved in this formulation:

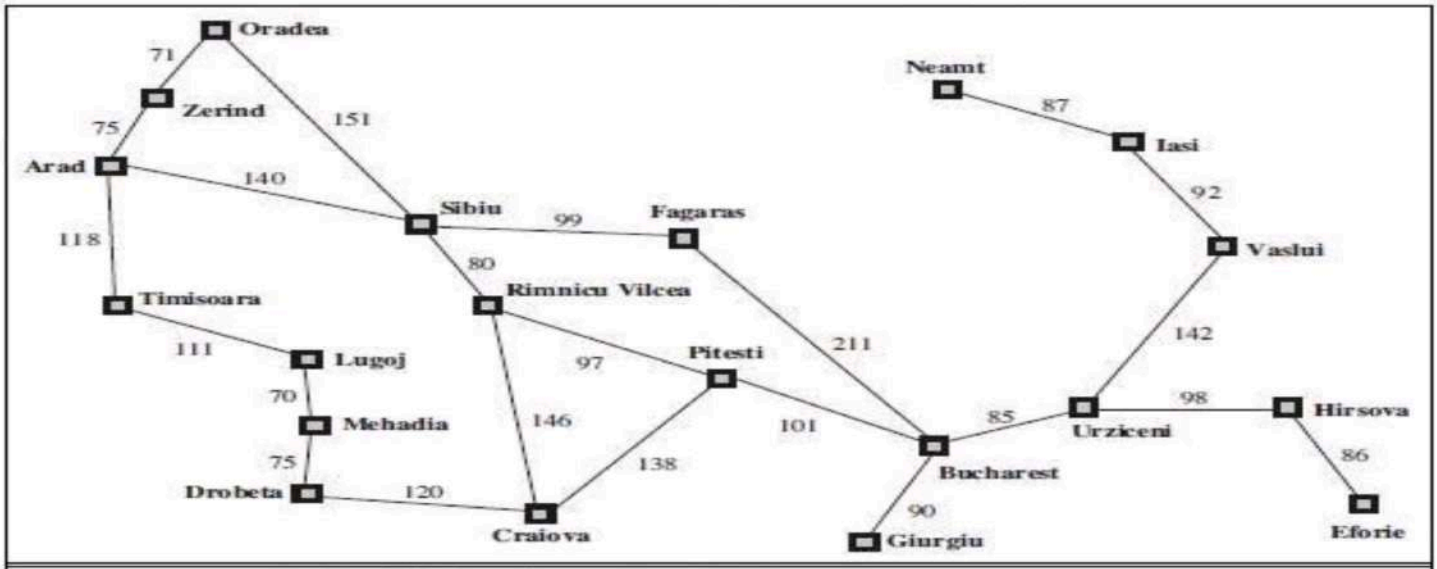
- **States:** Arrangement of any 0 to 8 queens on the chessboard.
- **Initial State:** An empty chessboard
- **Actions:** Add a queen to any empty box.
- **Transition model:** Returns the chessboard with the queen added in a box.
- **Goal test:** Checks whether 8-queens are placed on the chessboard without any attack.
- **Path cost:** There is no need for path cost because only final states are counted.

- In this formulation, there is approximately **1.8×10^{14}** possible sequence to investigate.

- **Complete-state formulation:** It starts with all the 8-queens on the chessboard and moves them around, saving from the attacks.

SOME REAL-WORLD PROBLEMS

- **Traveling salesperson problem (TSP):** It is a **touring problem** where the salesman can visit each city only once. The objective is to find the shortest tour and sell-out the stuff in each city.



A simplified road map of part of Romania

SOME REAL-WORLD PROBLEMS

- **VLSI Layout problem:** In this problem, millions of components and connections are positioned on a chip in order to minimize the area, circuit-delays, stray-capacitances, and maximizing the manufacturing yield.

THE LAYOUT PROBLEM IS SPLIT INTO TWO PARTS:

- **Cell layout:** Here, the primitive components of the circuit are grouped into cells, each performing its specific function. Each cell has a fixed shape and size. The task is to place the cells on the chip without overlapping each other.
- **Channel routing:** It finds a specific route for each wire through the gaps between the cells.
- **Protein Design:** The objective is to find a sequence of amino acids which will fold into 3D

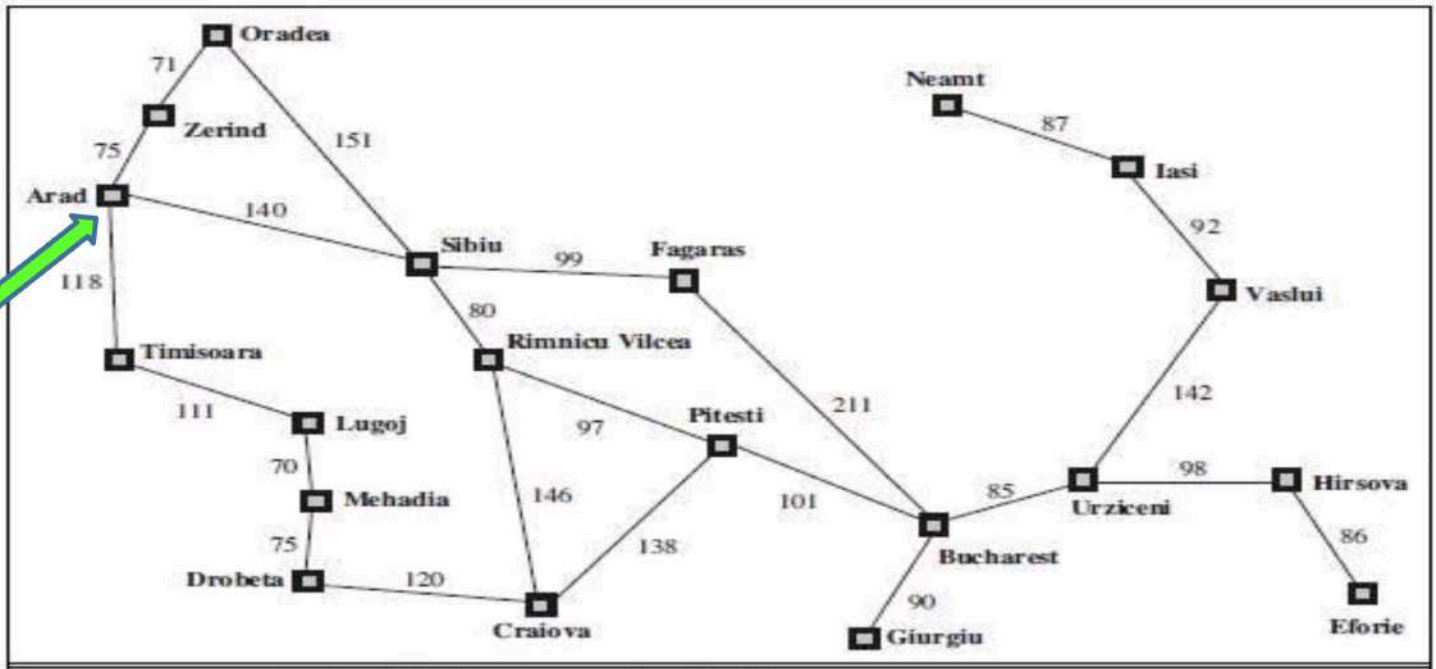
protein having a property to cure some disease.

ROBOT NAVIGATION is a generalization of the route-finding problem described earlier. Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states. For a circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs or wheels that must also be controlled, the search space becomes many-dimensional. Advanced techniques are required just to make the search space finite. We examine some of these methods in Chapter 25. In addition to the complexity of the problem, real robots must also deal with errors in their sensor readings and motor controls.

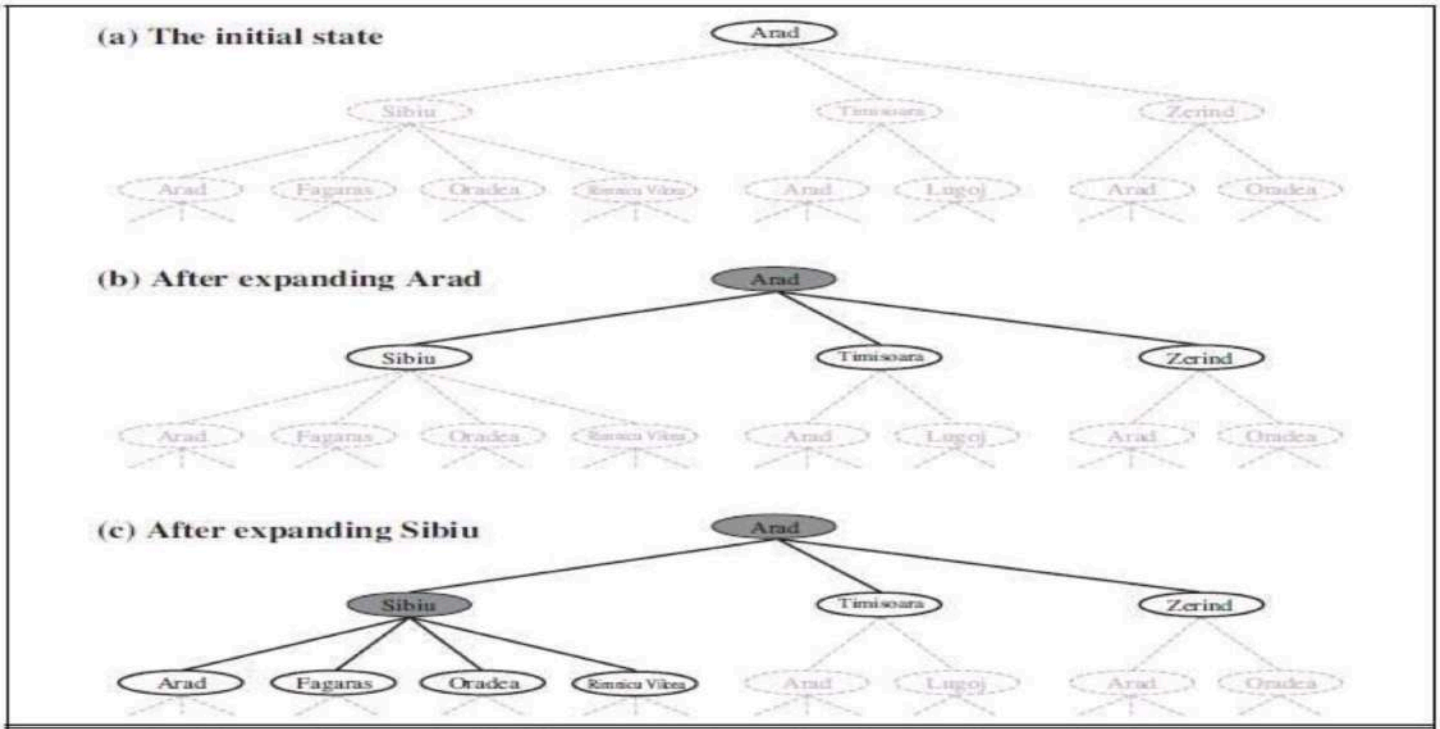
AUTOMATIC ASSEMBLY SEQUENCING of complex objects by a robot was first demonstrated by FREDDY (Michie, 1972). Progress since then has been slow but sure, to the point where the assembly of intricate objects such as electric motors is economically feasible. In assembly problems, the aim is to find an order in which to assemble the parts of some object. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done. Checking a step in the sequence for feasibility is a difficult

geometrical search problem closely related to robot navigation. Thus, the generation of legal actions is the expensive part of assembly sequencing. Any practical algorithm must avoid exploring all but a tiny fraction of the state space. Another important assembly problem is **protein design.**

SEARCHING FOR SOLUTIONS



A simplified road map of part of Romania.

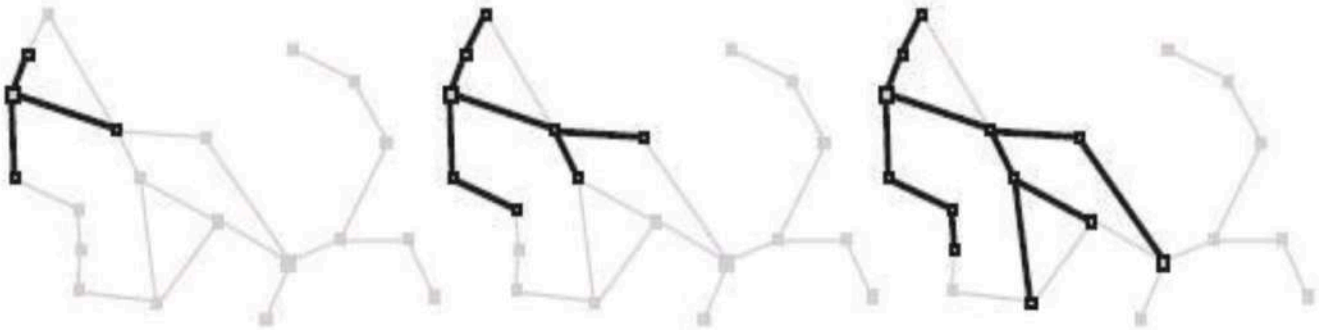


Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are

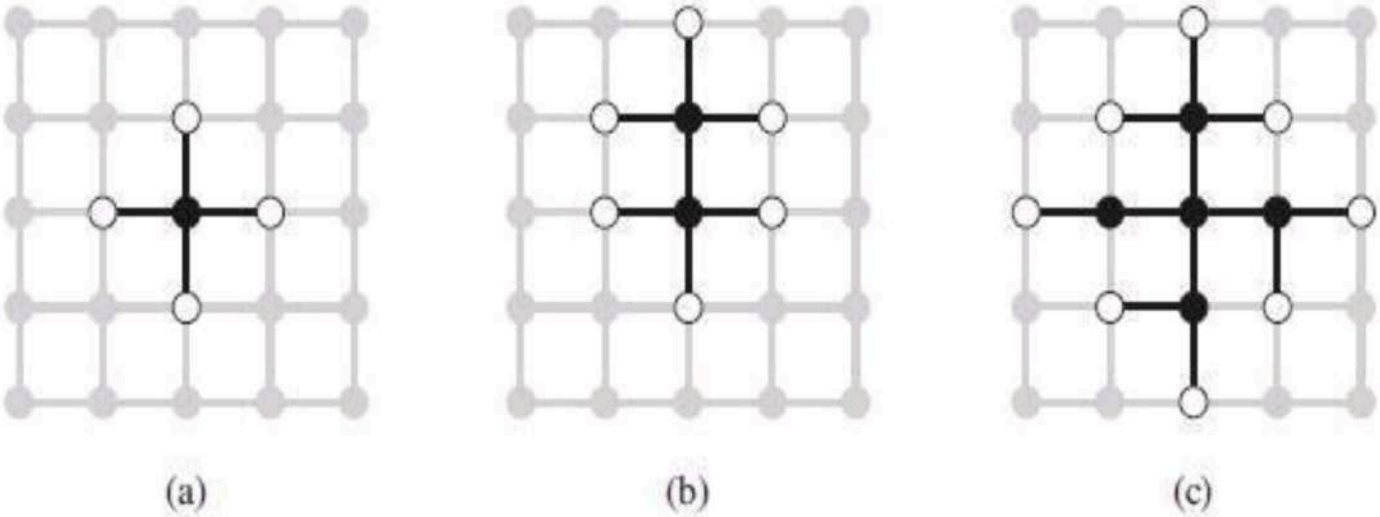
```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding 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
```

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



A sequence of search trees generated by a graph search on the Romania problem of Figure At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.



The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

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.

MEASURING PROBLEM-SOLVING PERFORMANCE

Before discussing different search strategies, the **performance measure** of an algorithm should be measured. Consequently, there are four ways to measure the performance of an algorithm:

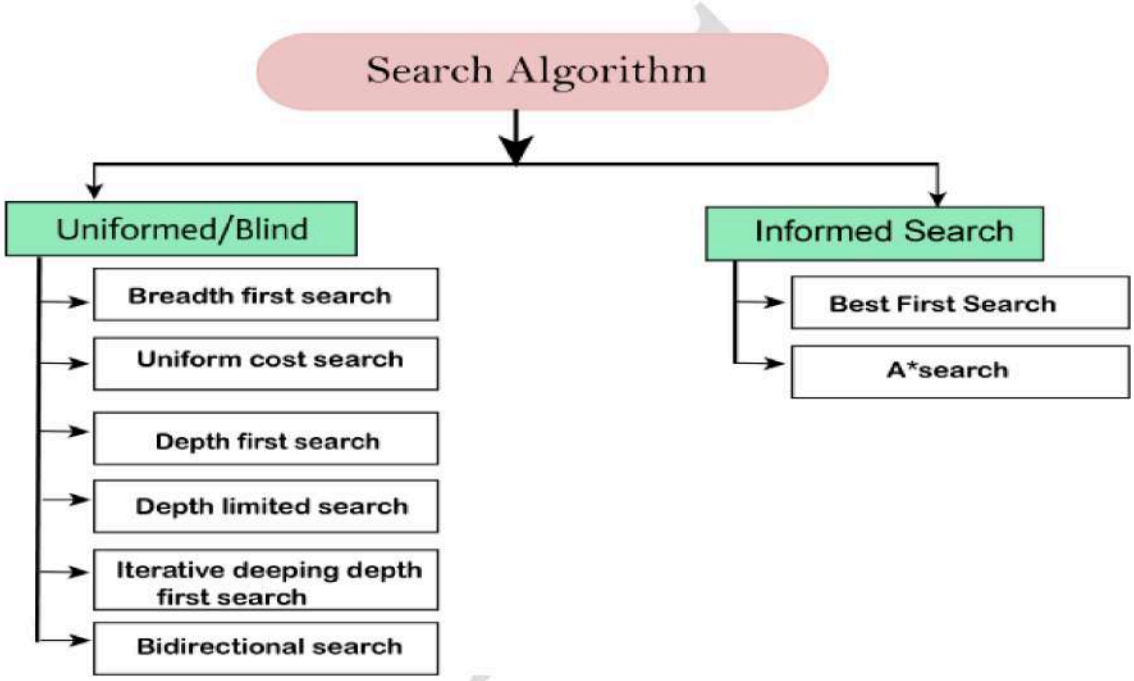
- **Completeness:** It measures if the algorithm guarantees to find a solution (if any solution exists).
- **Optimality:** It measures if the strategy searches for an optimal solution.
- **Time Complexity:** The time taken by the algorithm to find a solution.
- **Space Complexity:** Amount of memory required to perform a search.

The complexity of an algorithm depends on **branching factor** or **maximum number of**

successors, **depth of the shallowest goal node** (i.e., number of steps from root to the path) and **the maximum length of any path in a state space.**

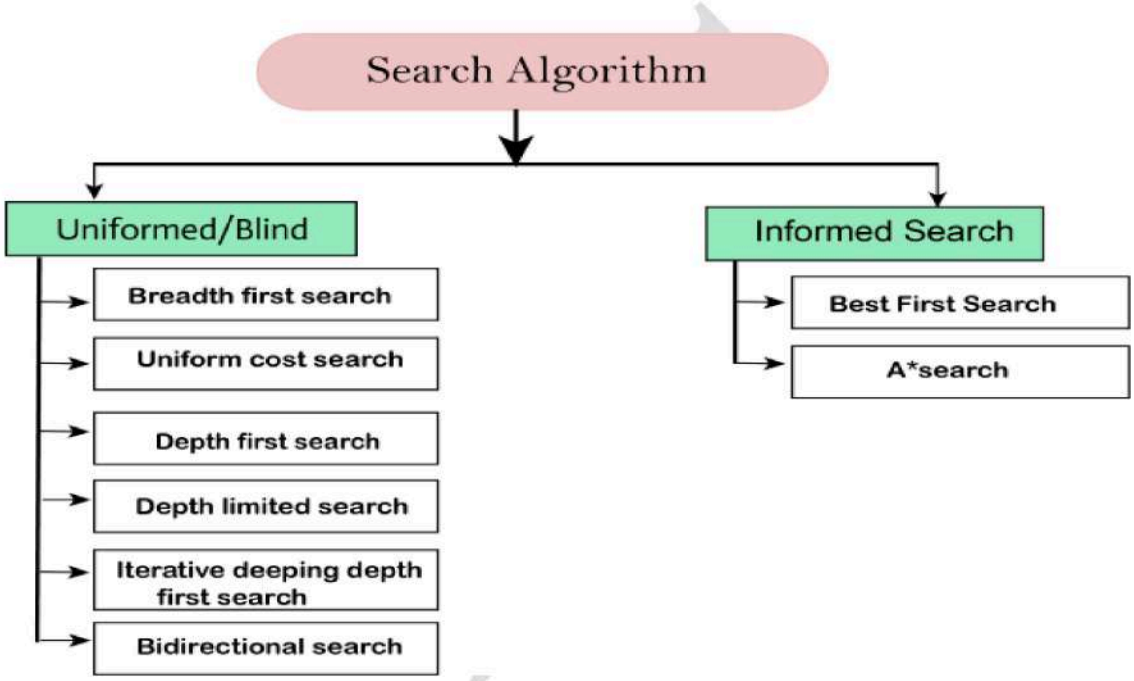
TYPES OF SEARCH ALGORITHMS

Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.


Search Algorithm
Uniformed/Blind
Breadth first search
Uniform cost search
Depth first search
Depth limited search
Iterative deeping depth first search
Bidirectional search
Informed Search
Best First Search
A*search

TYPES OF SEARCH ALGORITHMS

Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.


Search Algorithm
Uniformed/Blind
Breadth first search
Uniform cost search
Depth first search
Depth limited search
Iterative deeping depth first search
Bidirectional search
Informed Search
Best First Search
A*search

UNINFORMED/BLIND SEARCH:

The uninformed search 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 and how to identify leaf and goal nodes.

Uninformed search applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goal, so it is also called blind search. It examines each node of the tree until it achieves the goal node.

It can be divided into five main types:

- Breadth-first search
- Uniform cost search
- Depth-first search

- Iterative deepening depth-first search
- Bidirectional Search

1. BREADTH-FIRST SEARCH:-

- Breadth-first search is the most common search strategy for traversing a tree or graph.
- This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

ADVANTAGES:

- BFS will provide a solution if any solution exists.

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

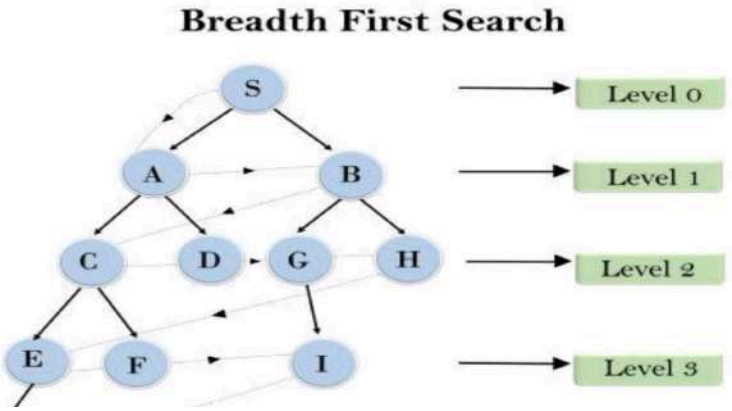
DISADVANTAGES:

- 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.

EXAMPLE:

S---> A--->B---->C--->D---->G--->H--->E---->F---->I---- >K

- The traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which



is shown by the dotted arrow,

and the traversed path will be:

- **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.

2. DEPTH-FIRST SEARCH

- Depth-first search is a recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

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:

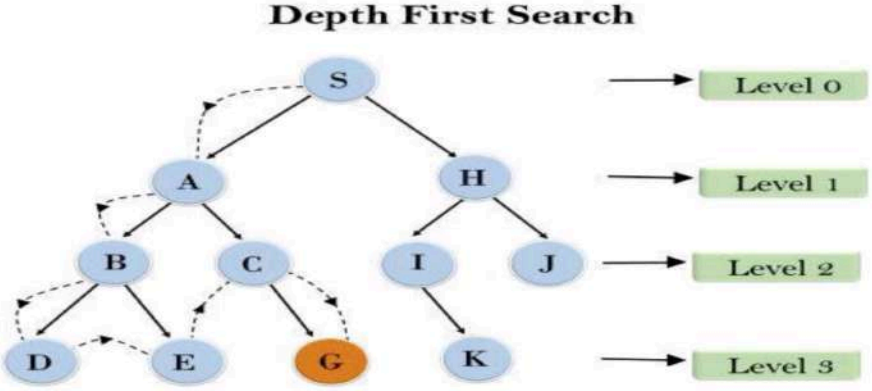
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

EXAMPLE:

Depth-first search, and it will follow the order as:

Root node--->Left node ----> right node.

It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.



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.

3. DEPTH-LIMITED SEARCH ALGORITHM:

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.
- **Cut off failure value:** It defines no solution for the problem within a given depth limit.

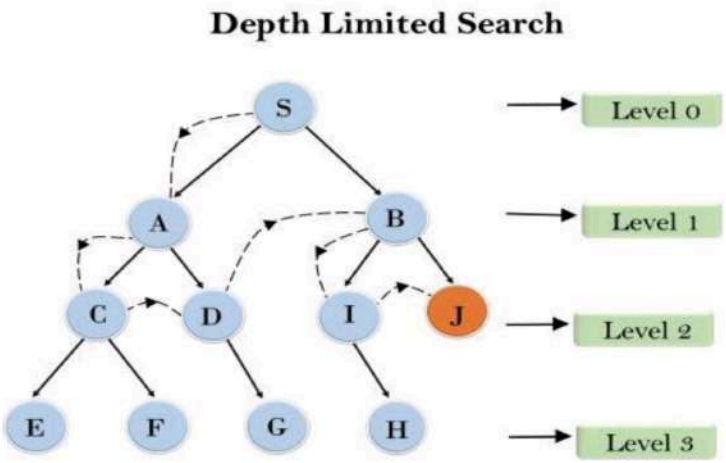
Advantages:

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.

EXAMPLE



Completeness: DLS search algorithm is complete if the solution is above the depth-limit.

Time Complexity: Time complexity of DLS algorithm is $\mathbf{O(b^{\ell})}$.

Space Complexity: Space complexity of DLS algorithm is $\mathbf{O(b \times \ell)}$.

Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $\ell > d$.

4. UNIFORM-COST SEARCH ALGORITHM

- 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.
- It can be used to solve any graph/tree where the optimal cost is in demand.
- 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.

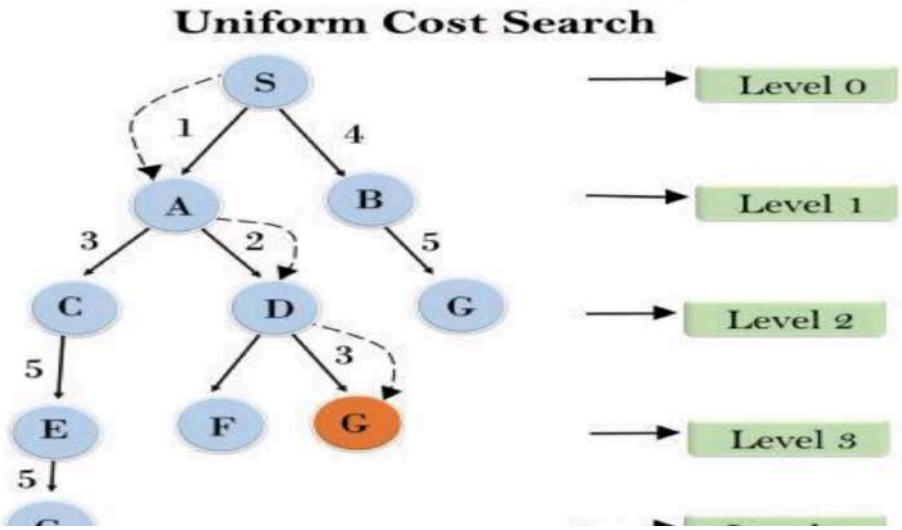
Advantages:

- 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$\mathbf{O(b^{1 + \lceil C^*/\varepsilon \rceil})}$/.

Space Complexity: The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $\mathbf{O(b^{1 + \lceil C^*/\varepsilon \rceil})}$.

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

5. ITERATIVE DEEPENING DEPTH-FIRST SEARCH

- 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.
- The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Advantages:

- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

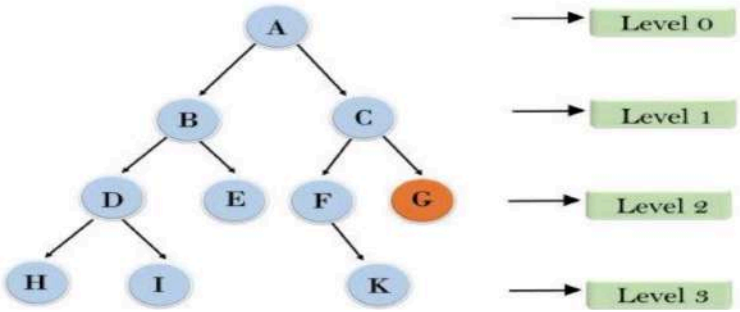
Disadvantages:

- The main drawback of IDDFS is that it repeats all the work of the previous phase.

EXAMPLE

- Following tree structure is showing the iterative deepening depth-first search.
- IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:

Iterative deepening depth first search



1'st Iteration-----> A

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

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

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

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

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 $\mathbf{O(b^d)}$.

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.

6. BIDIRECTIONAL SEARCH ALGORITHM

- 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, DLS, etc.

Advantages:

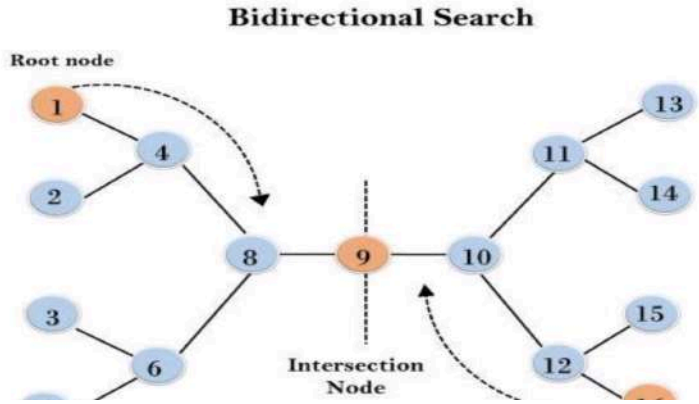
- Bidirectional search is fast.
- Bidirectional search requires less memory

Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- **In bidirectional search, one should know the goal state in advance.**

EXAMPLE

- 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 $\mathbf{O(b^d)}$.

Space Complexity: Space complexity of

bidirectional search is $\mathbf{O(b^d)}$.

Optimal: Bidirectional search is Optimal.

INFORMED SEARCH ALGORITHMS

INFORMED SEARCH ALGORITHMS

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.

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

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**

1.) BEST-FIRST SEARCH ALGORITHM (GREEDY SEARCH)

- Greedy best-first search algorithm 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.

$$\mathbf{f(n)= g(n).}$$

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:

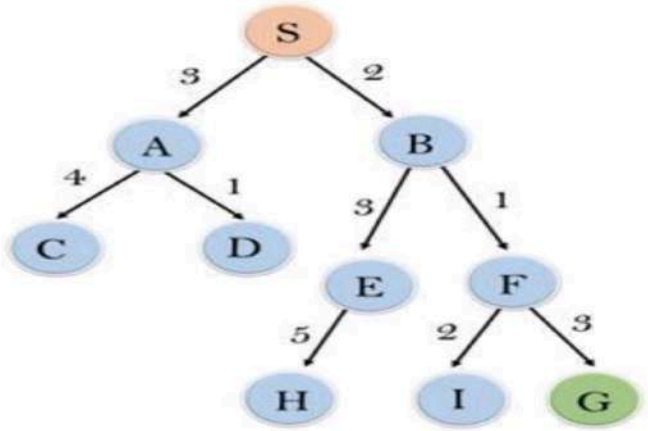
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

DISADVANTAGES:

- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.

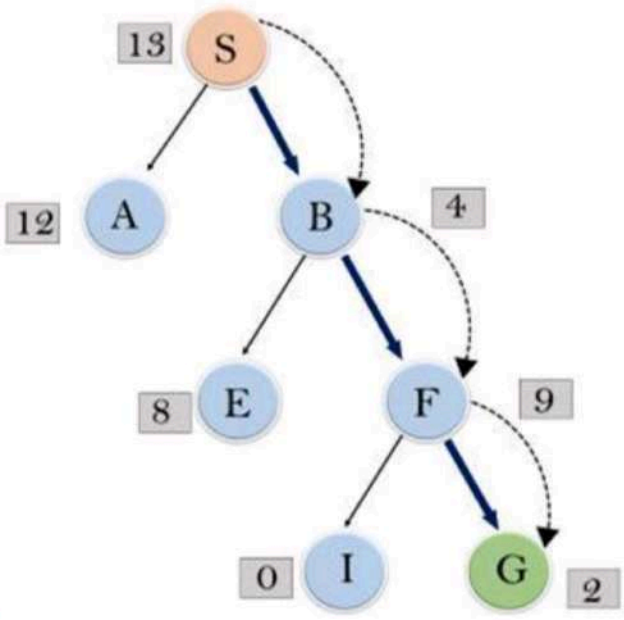
EXAMPLE

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.



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

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



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.

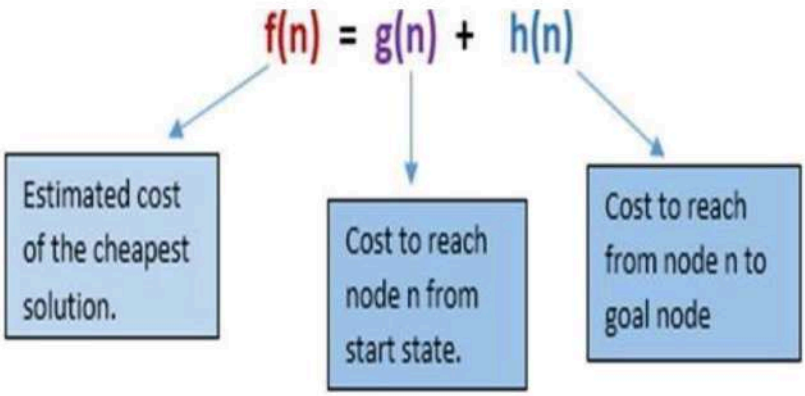
Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

Optimal: Greedy best first search algorithm is not optimal.

2.) A* SEARCH ALGORITHM:

- A* search is the most commonly known form of best-first search.
- It uses heuristic function h(n), and cost to reach the node n from the start state g(n) to find the **best shortest path.**
- It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently.
- 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.
- A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).
- 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**.



NOTE: At each point in the search space, only those nodes is expanded which have the lowest value of f(n), and the algorithm terminates when the goal node is found.

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**.

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.

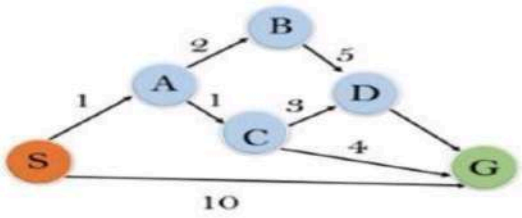
DISADVANTAGES:

- 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.

EXAMPLE

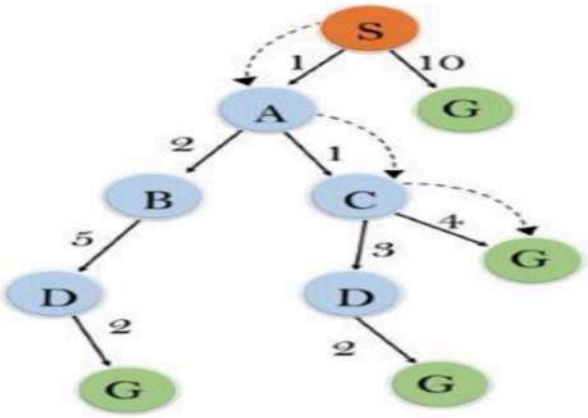
- In this example, we will traverse the given graph using the A* algorithm.
- The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.

Here we will use OPEN and CLOSED list.



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

SOLUTION:



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.

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.
- A* algorithm expands all nodes which satisfy the condition f(n)<="" li="">

COMPLETE:

A* algorithm is complete as long as:

- Branching factor is finite.
- 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.

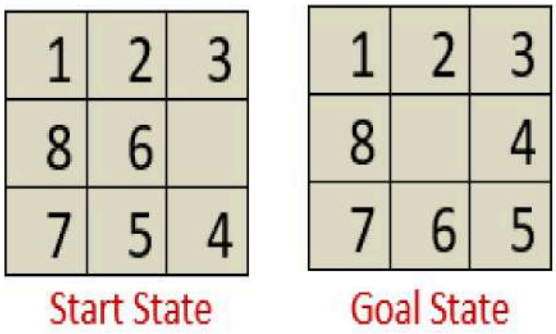
HEURISTIC FUNCTIONS

HEURISTIC FUNCTIONS

- As we have already seen that an informed search makes use of heuristic functions in order to reach the goal node in a more prominent way. Therefore, there are several pathways in a search tree to reach the goal node from the current node. The selection of a good heuristic function matters certainly. A good heuristic function is determined by its efficiency. More is the information about the problem, more is the processing time.
- Some toy problems, such as 8-puzzle, 8-queen, tic-tac-toe, etc., can be solved more efficiently with the help of a heuristic function. Let's see how:
- Consider the following 8-puzzle problem where we have a start state and a goal state. Our task is to slide the tiles of the current/start state and place it in an order followed in the goal state. There can be four moves either **left, right, up, or down**. There can be several ways to convert the

current/start state to the goal state, but, we can use a heuristic function h(n) to solve the problem more efficiently.

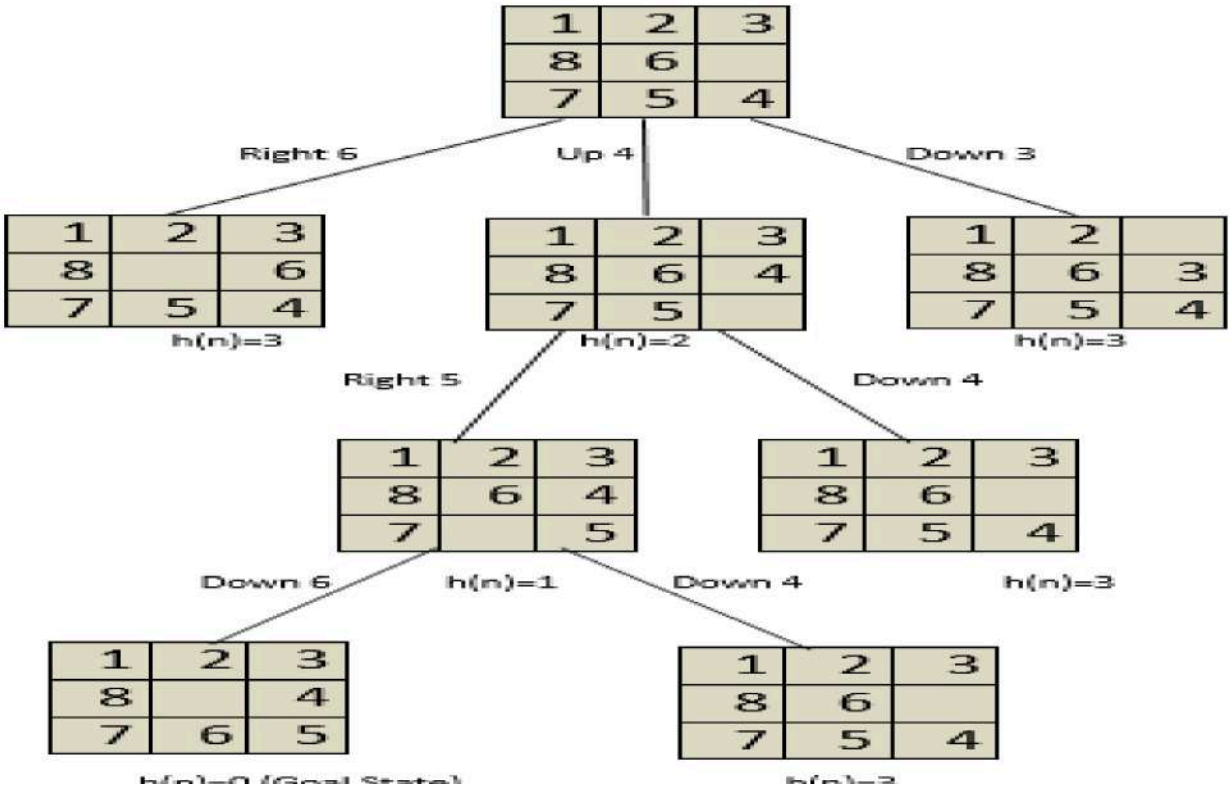
8 PUZZLE PROBLEM USING HEURISTIC FUNCTION



A heuristic function for the 8-puzzle problem is defined below:

h(n)=Number of tiles out of position.

So, there is total of three tiles out of position i.e., 6,5 and 4. Do not count the empty tile present in the goal state). i.e. h(n)=3. Now, we require to minimize the value of **h(n) =0.**


1 2 3
8 6
7 5 4
Right 6
Up 4
Down 3
1 2 3
8 6
7 5 4
h(n)=3
1 2 3
8 6 4
7 5
h(n)=2
1 2
8 6 3
7 5 4
h(n)=3
Right 5
Down 4
1 2 3
8 6 4
7 5
1 2 3
8 6
7 5 4
Down 6
h(n)=1
Down 4
h(n)=3
1 2 3
8 4
7 6 5
1 2 3
8 6
7 5 4

- It is seen from the above state space tree that the goal state is minimized from h(n)=3 to h(n)=0
- However, we can create and use several heuristic functions as per the requirement. It is also clear from the above example that a heuristic function h(n) can be defined as the information required to solve a given problem more efficiently. The information can be related to the **nature of the state, cost of transforming from one state to another, goal node characteristics,** etc., which is expressed as a heuristic function.

PROPERTIES OF A HEURISTIC SEARCH ALGORITHM

Use of heuristic function in a heuristic search algorithm leads to following properties of a heuristic search algorithm:

- **Admissible Condition:** An algorithm is said to be admissible, if it returns an optimal solution.
- **Completeness:** An algorithm is said to be complete, if it terminates with a solution (if the solution exists).
- **Dominance Property:** If there are two admissible heuristic algorithms **A1** and **A2** having **h1** and **h2** heuristic functions, then **A1** is said to dominate **A2** if **h1** is better than **h2** for all the values of node **n.**
- **Optimality Property:** If an algorithm is **complete, admissible**, and **dominating** other

algorithms, it will be the best one and will definitely give an optimal solution

LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

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

- keeping different paths in the memory and
- selecting the best suitable path,

Which leads to a solution state required to reach the goal node. But beyond these **"classical search algorithms**," we have some **"local search algorithms" where the path cost does not matters, and only focus on solution-state needed to reach the goal node.**

A local search algorithm completes its task by traversing on a single current node rather than

multiple paths and following the neighbours of that node generally.

Although local search algorithms are not systematic, still they have the following two advantages:

- 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.

Does the local search algorithm work for a pure optimized problem?

Yes, 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**. Unfortunately, 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

- **Location:** It is defined by the state.
- **Elevation:** It is defined by the value of the objective function or heuristic cost function.

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**. It is the highest point in the valley.

- **We will understand the working of these points better in Hill-climbing search.**

Below are some different types of local searches:

- Hill-climbing Search
- Simulated Annealing

- Local Beam Search