UNIT – II

Problem Solving Agents

- It’s a goal-based agent. In Problem-solving agents **Goal formulation** is the first step. Goal formulation based on the **current situation** and the agent’s **performance measure**.
- The agent has to find out which sequence of actions will get to a goal state.
- Agent needs to decide what sorts of actions and states are to be considered
- The process of examining different possible sequences 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, then the recommended actions can be carried out , this phase is known as **execution.**
- The complete process of an agent to solve the problems can be shown as “formulate, search, execute” design for the agent.
- A simple problem-solving agent
    - Formulate a goal and a problem
    - Search for a sequence of actions to solve the problem
    - Execute the actions one at a time

A problem can be defined formally by five components

- **Initial State:** This represents the starting point for the AI agent, establishing the context in which the problem is addressed. The initial state may also involve initializing methods for problem-solving.

- **Action:** This stage involves selecting functions associated with the initial state and identifying all possible actions. Each action influences the progression toward the desired goal.

- **Transition:** This component integrates the actions from the previous stage, leading to the next state in the problem-solving process. Transition modeling helps visualize how actions affect outcomes.

- **Goal Test:** This stage verifies whether the specified goal has been achieved through the integrated transition model. If the goal is met, the action ceases, and the focus shifts to evaluating the cost of achieving that goal.

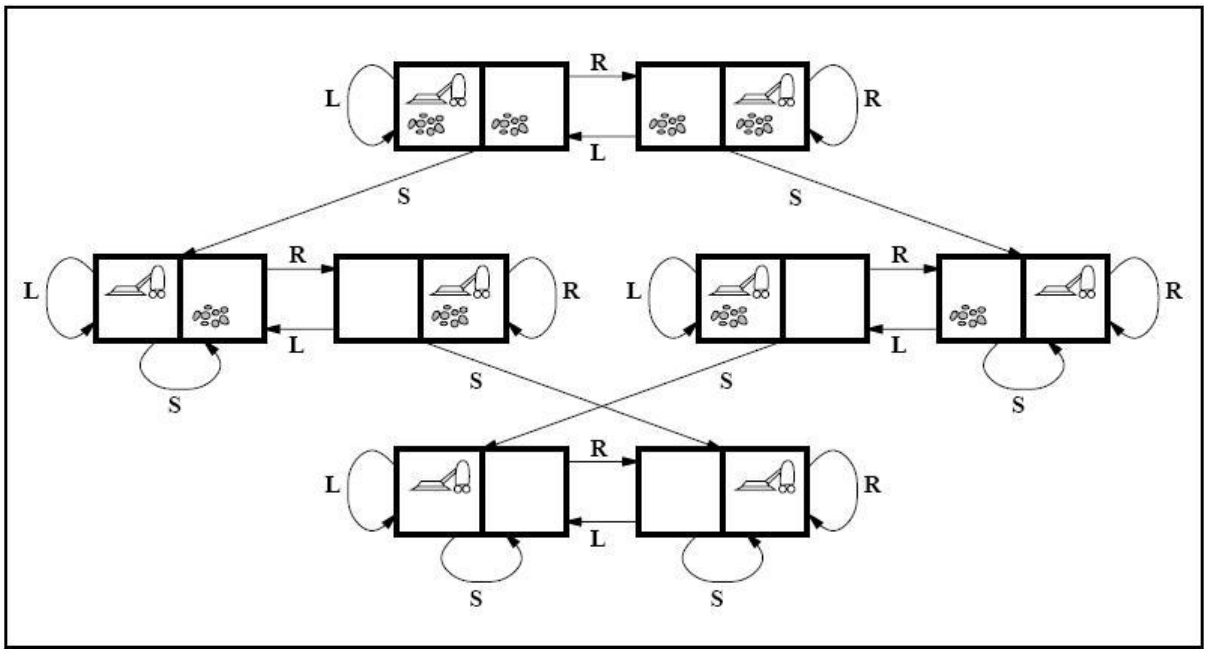
- **Path Costing:** This component assigns a numerical value representing the cost of achieving the goal.

Example Problems

vacuum world

This can be formulated as a problem as follows:

- **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.
- **Actions:** In this simple environment, each state has just three actions: **Left, Right, and Suck.**
- Larger environments might also include Up and Down
- **Transition model:** The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect.
- **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



Arcs denote actions { L, R, S}

states? dirt and agent location

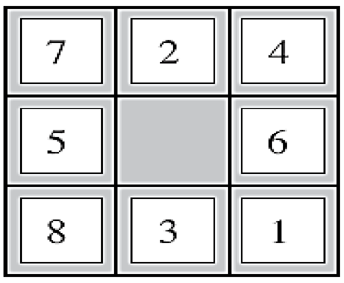
actions? Left, Right, Suck

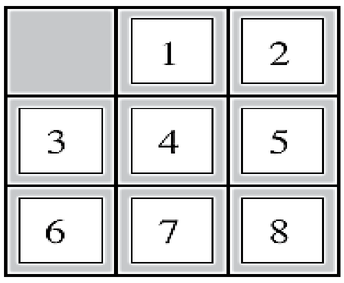
goal test? no dirt at all locations

path cost? 1 per action

8-puzzle

- consists of a 3×3 board with eight numbered tiles and a blank space.
- A tile adjacent to the blank space can slide into the space.

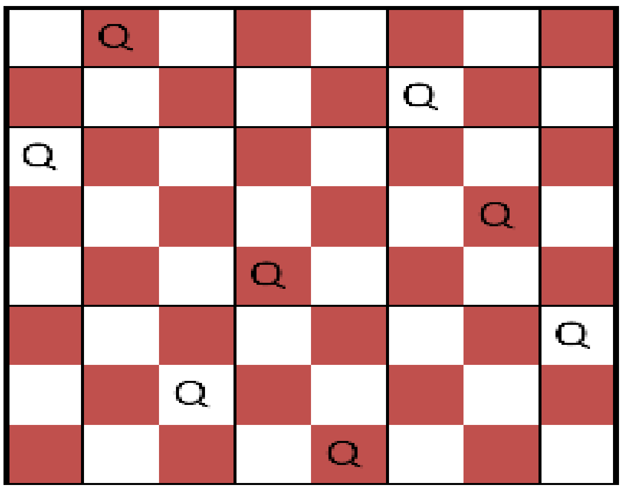

Start State


Goal State

- **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.
- **Transition model:** Given a state and action, this returns the resulting state; for example, if we apply Left to the start state in 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
- **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

8-queens problem

- The goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other.
- A queen attacks any piece in the same row, col or diagonal
- Efficient algorithms exist for whole n-queens family
- Still, an interesting test problem for search algorithms



- There are two main kinds of formulation

Incremental formulation

- Starting with an empty state
- For 8-queen problem, each action adds a queen to the state

Complete state formulation

- Starts with all 8 queens on the board
- Moves them around
- In both cases, no path cost because only the final state counts!

The first incremental formulation one might try is the following:

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.

We have 1.8*1014 possible sequences to investigate.

A better formulation would prohibit placing a queen in any square that is already attacked:

States:

Arrangements of n queens (n= 1 to 8), one per column in the leftmost n columns, with no queen attacking another are states

Actions:

Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen. 2057 sequences to investigate

Searching for Solutions

Search: Searching is a step-by-step procedure to solve a search-problem in a given search space. A search problem can have three main factors:

- **Search Space:** Search space represents a set of possible solutions, which a system may have.
- **Start State:** It is a state from where agent begins **the search**.
- **Goal test:** It is a function which observe the current state and returns whether the goal state is achieved or not.

Properties of Search Algorithms:

Following are the four essential properties of search algorithms to compare the efficiency of these algorithms:

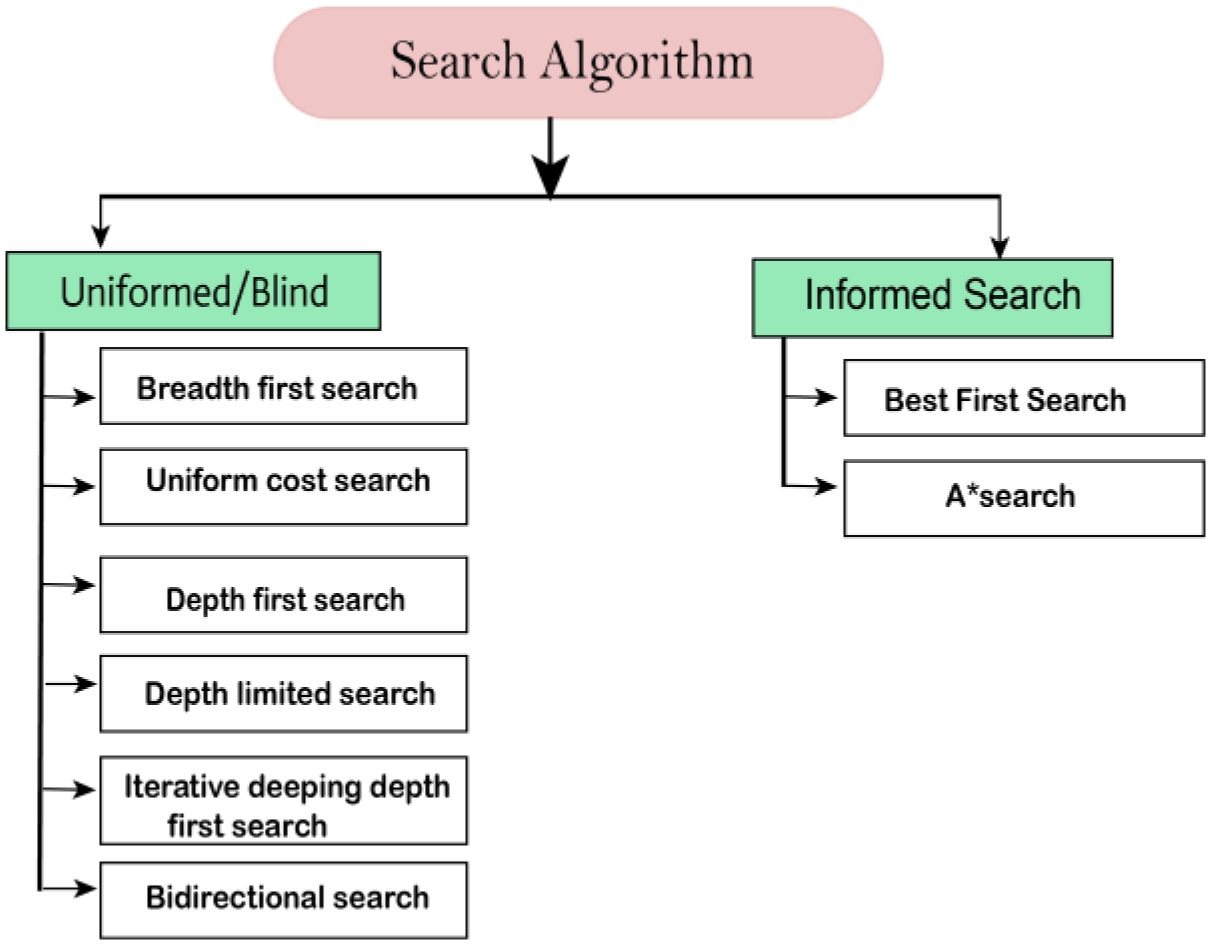
- **Completeness:** A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.

- **Optimality:** If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.
- **Time Complexity:** Time complexity is a measure of time for an algorithm to complete its task.
- **Space Complexity:** It is the maximum storage space required at any point during the search, as the complexity of the problem.

Types of search algorithms

There are two types of search strategies:

- Uninformed Search Strategy (Blind search)
- Informed Search Strategy (Heuristic search)



Uninformed Search Strategy (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.
- Various uniformed search strategies are
    - Breadth-first search
    - Uniform cost search
    - Depth-first search
    - Depth Limited Search
    - Iterative deepening depth-first search
    - Bidirectional Search

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 solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

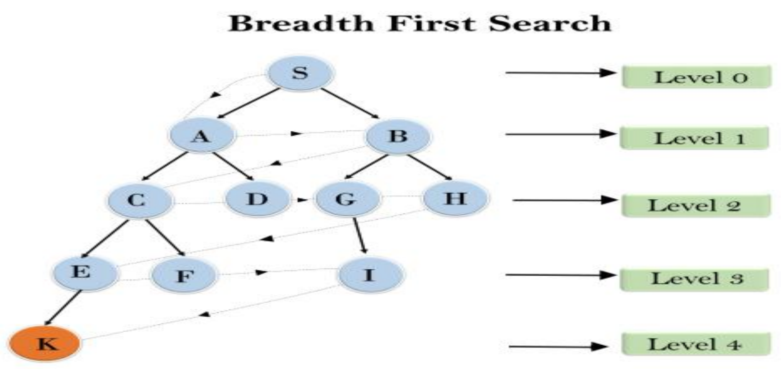
Disadvantages:

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

Example:

In the below tree structure, we have shown 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:

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



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.

Depth-first Search:

- Depth-first search isa 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:

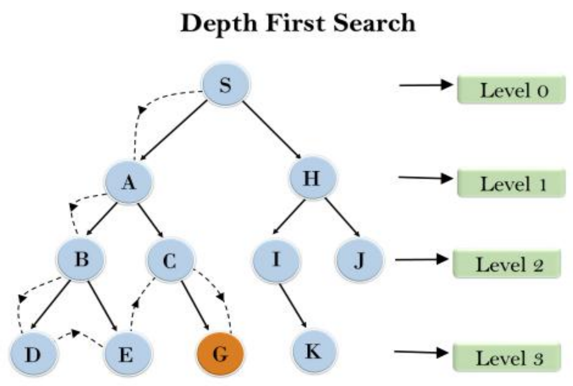
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Example:

In the below search tree, we have shown the flow of 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.

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

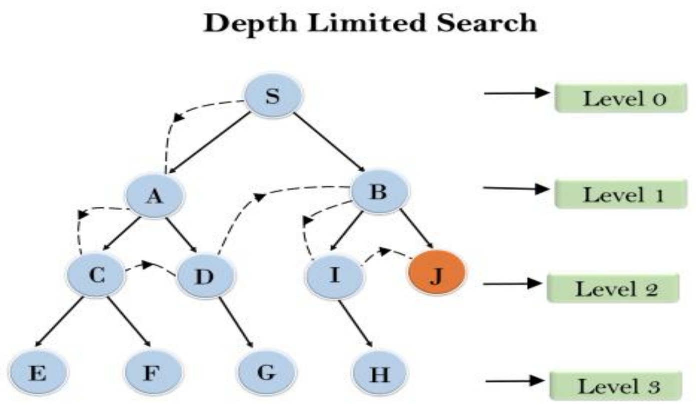
Advantages:

Depth-limited search is Memory efficient.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

Example:



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

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

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

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

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.

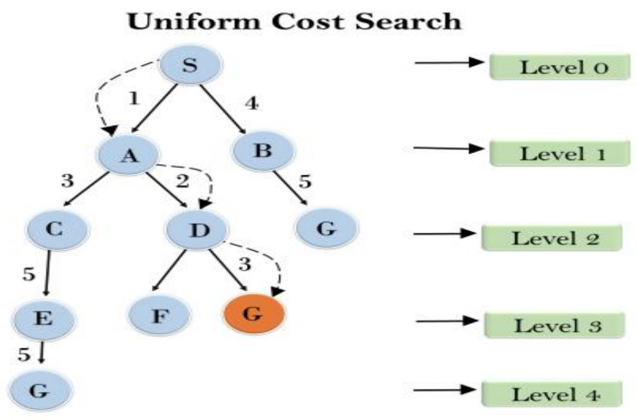
Advantages:

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

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

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

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:

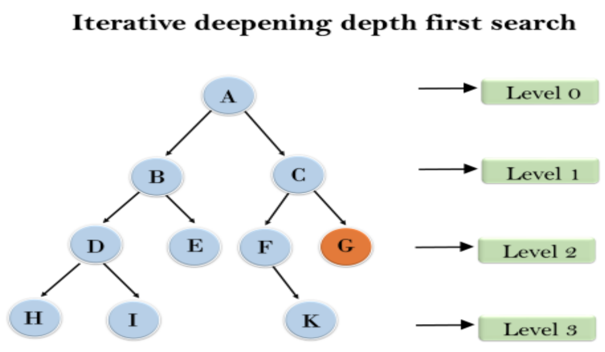
- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

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:



1'stIteration----->A
2'ndIteration---->A,B,C
3'rdIteration------>A,B,D,E,C,F,G
4'thIteration------>A,B,D,H,I,E,C,F,K,G
In the fourth iteration, the algorithm will find the goal node.

Completeness: This algorithm is complete is ifthe branching factor is finite.

Time Complexity: Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **$O(b^d)$**.

Space Complexity: The space complexity of IDDFS will be **O(bd)**.

Optimal: IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

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.

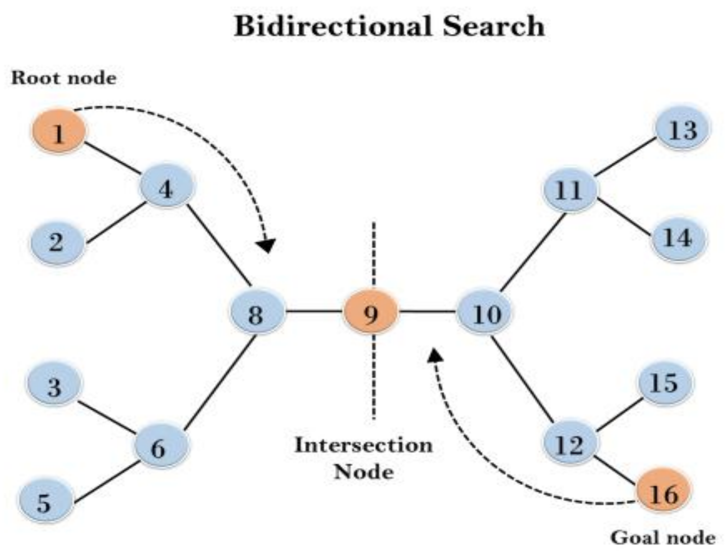
- Bidirectional search requires less memory

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 **O(b^{d})**.

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

Optimal: Bidirectional search is Optimal.

Informed Search Algorithms

- Informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc. This knowledge help agents to explore less to the search space and find more efficiently the goal node.
- 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**.
- 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:

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,

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:

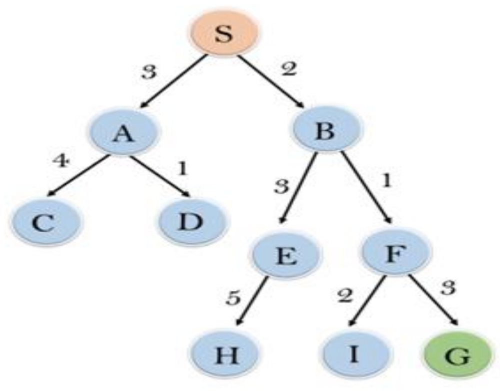
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

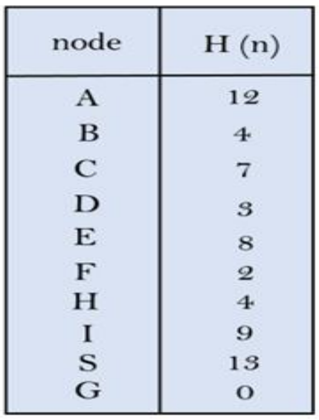
Disadvantages:

- 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:

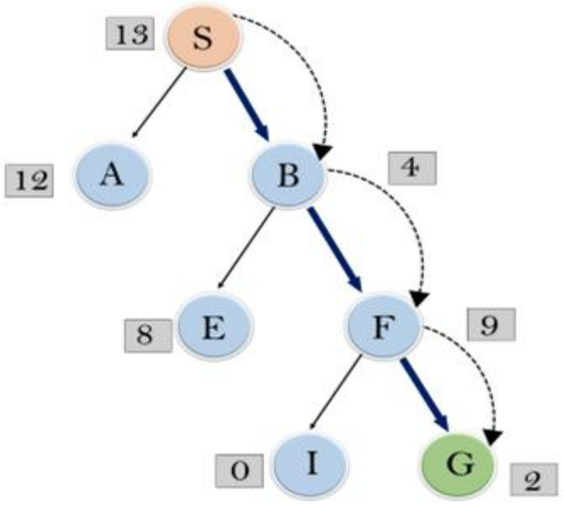
- Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.





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.



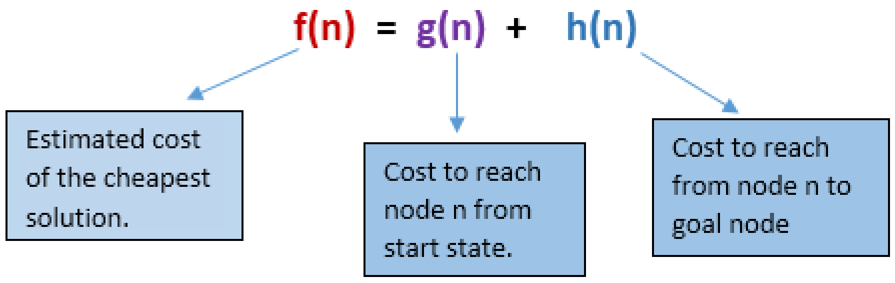
- **Expand the nodes of S and put in the CLOSED list**
- **Initialization:** Open [A, B], Closed [S]
- **Iteration 1:** Open [A], Closed [S, B]
- **Iteration2:** Open[E,F,A],Closed[S,B]
  : Open [E, A], Closed [S, B, F]

- **Iteration3:** 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.

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).
- 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**.


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

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:

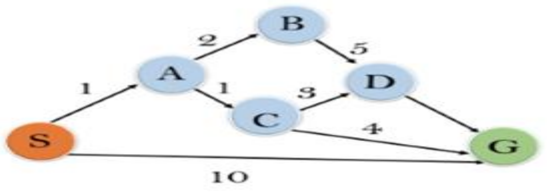
- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

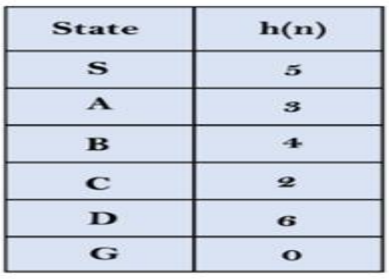
Disadvantages:

- 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.

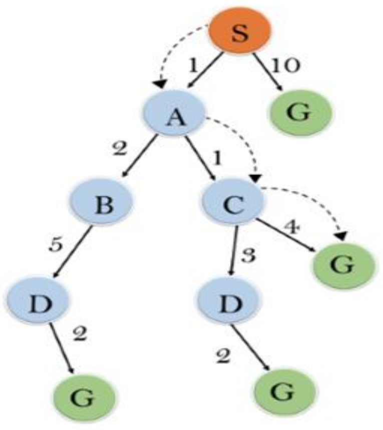
Example:

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.

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

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

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

Heuristic Functions

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:

1. 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.

we look at heuristics for the 8-puzzle, in order to understand the nature of heuristics in general.

The 8-puzzle was one of the earliest heuristic search problems.

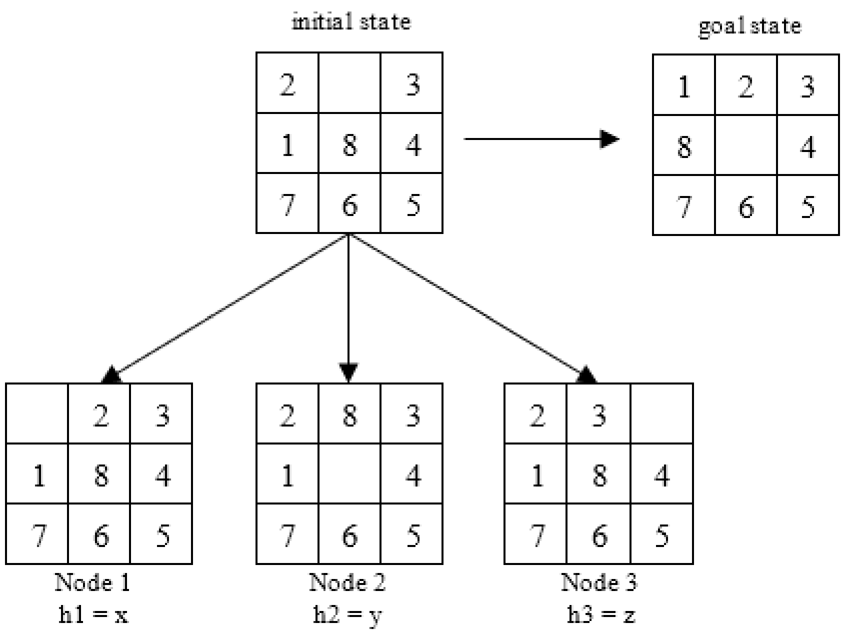
The instance of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration

The heuristic function should never over estimate the number of steps to the goal

Two commonly used candidates:

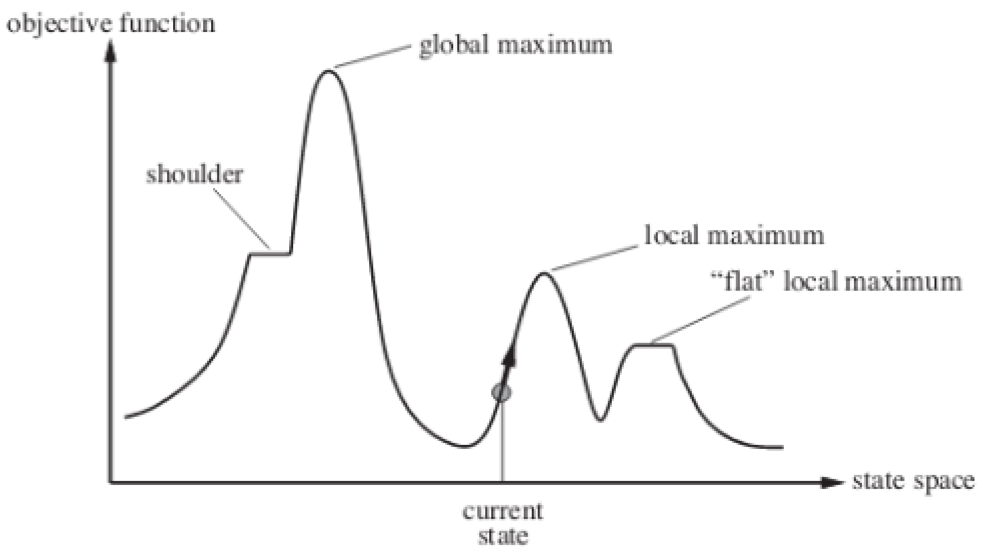
h1=the number of misplaced tiles

h2=the sum of the Manhattan distances of the tiles from their goal positions (i.e., no. of squares from desired location of each tile)



Local search algorithms and optimization problems

- The search algorithms that we have seen so far, explores search space systematically. i.e. when a goal state is found, the path to the goal state constitutes the solution
- In many problems, however the path to the goal state is irrelevant.
- Example :8queen problem
- If the path to goal state doesn't matter then, we can use local search algorithms
- To understand local search, we consider the state-space landscape
- A landscape has both "location" (defined by the state) and "elevation" (defined by the value of the heuristic cost function or objective function).
- If elevation corresponds to cost, then the aim is to find the lowest valley—a global minimum
- if elevation corresponds to an objective function, then the aim is to find the highest peak—a global maximum.
- Local search algorithms explore this landscape.
- A complete local search algorithm always finds a goal if one exists;
- an optimal algorithm always finds a global minimum/maximum



Local search algorithms

- Hill-climbing search
- Simulated annealing
- Local beam search
- Genetic algorithms

Hill-climbing search

- It is a local search algorithm which continuously moves in the direction of increasing value to find best solution to the problem (uphill)
- It terminates when it reaches a peak value where no neighbour has a higher value.

- The algorithm does not maintain a search tree, so the data structure for the current node need only to record the state and the value of the objective function.
- Hill climbing does not look ahead beyond the immediate neighbors of the current state
- Hill climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next
- This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia

Simulated annealing Algorithm:

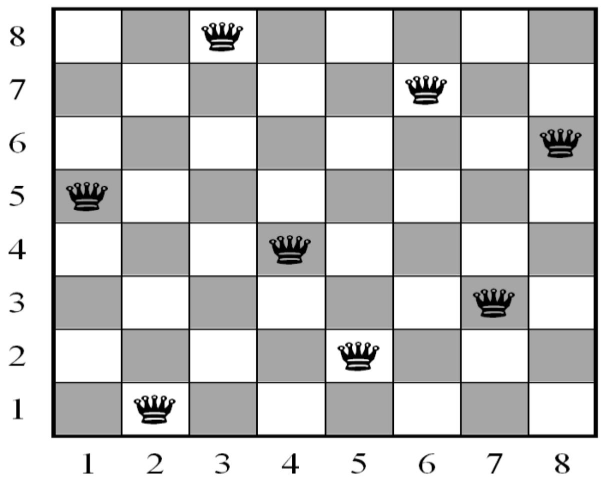
- A hill-climbing algorithm which never makes a move towards a lower value guaranteed to be incomplete because it can get stuck on a local maximum.
- If algorithm applies a random walk, moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient
- Simulated Annealing is an algorithm which yields both efficiency and completeness
- Annealing is a process of hardening a metal or glass to a high temperature then cooling gradually, so this allows the metal to reach a low-energy crystalline state
- The same process is used in simulated annealing in which the algorithm picks a random move, instead of picking the best move. If the random move improves the state, then it follows the same path. Otherwise, the algorithm follows the path which has a probability of less than 1 or it moves downhill and chooses another path

Local beam search

- In local beam search, it keeps track of k-states rather than just 1.
- It begins with k- randomly generated states. At each step all the successors of all K states are generated.
- If anyone is a goal state the algorithm terminated. Otherwise, it selects k-best successors from the list and repeats the same process.
- **Stochastic beam search:** Instead of choosing k-best successor, stochastic beam search selects k-successors randomly.
- A local beam search with k-states is Similar to running k-random restarts in parallel. This is called a local beam search.

Genetic Algorithms:

- it is a variant of stochastic beam search.
- successor are generated by combining two parent states rather than by modifying a single state.
- Like stochastic beam search genetic algorithm begins with k randomly generated states that is known as population
- Ex: consider 8-queens problem each state is represented as a 8 digit string representing position of the queen in a 8x8 board


8
7
6
5
4
3
2
1
1 2 3 4 5 6 7 8