Chapter 3: Problem Solving Using Searching

- In artificial intelligence, problems can be solved by using searching algorithms, evolutionary computations, knowledge representations, etc.
- I am going to discuss the various searching techniques that are used to solve a problem.
- In general, searching is referred to as finding information one needs.

The process of problem-solving using searching consists of the following steps.

- Define the problem
- Analyze the problem
- Identification of possible solutions
- Choosing the optimal solution
- Implementation

Properties of search algorithms

- **Completeness:** A search algorithm is said to be complete when it gives a solution or returns any solution for a given random input.
- **Optimality:** If a solution found is best (lowest path cost) among all the solutions identified, then that solution is said to be an optimal one.
- **Time complexity:** The time taken by an algorithm to complete its task is called time complexity. If the algorithm completes a task in a lesser amount of time, then it is an efficient one.
- **Space complexity:** It is the maximum storage or memory taken by the algorithm at any time while searching.

These properties are also used to compare the efficiency of the different types of searching algorithms.

Problem solving agents:

- Problem-solving agents are intelligent agents that operate by searching for a sequence of actions that lead to the desired goal state. They can be applied to various problem types and domains.
- Here's an overview of problem types, formulations, and example problems:

Problem Types:

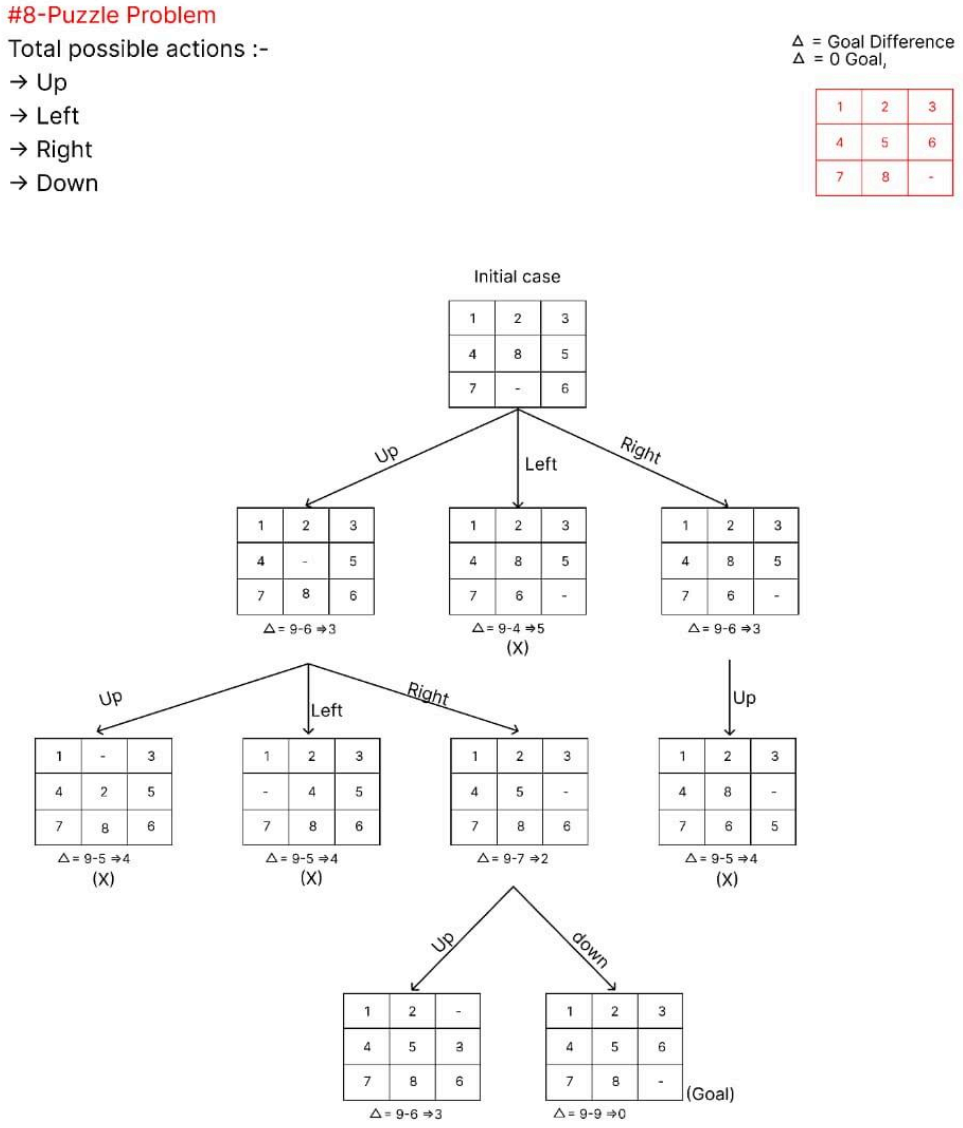
- **Deterministic problems:** In deterministic problems, the outcome of actions is entirely predictable. These are often represented as state transition diagrams or state transition tables.
- **Stochastic problems**: Stochastic problems involve uncertainty in the outcome of actions. The result of an action may vary probabilistically.
- **Single-agent problems:** In single-agent problems, there is only one agent operating in the environment. The agent's actions affect the environment, but there are no other competing agents.
- **Multi-agent problems**: Multi-agent problems involve multiple agents operating in the environment, each pursuing its own goals. The actions of one agent may affect the environment and the goals of other agents.
- **Adversarial problems:** Adversarial problems involve competing against an intelligent adversary. Examples include games like chess or poker.

Problem Formulations:

Initial State: Defines the starting configuration of the problem.
Actions: Describes the set of possible actions available to the agent.
Transition Model: Specifies the result of each action in terms of how it changes the state of the problem.
Goal Test: Determines whether a given state is a goal state.
Path Cost: Assigns a cost to each path (sequence of actions) in the problem space.

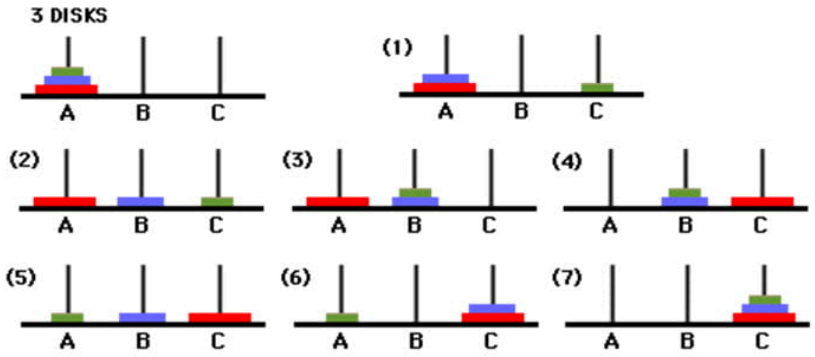
Example Problems:

1) **8-Puzzle:** A classic puzzle where you need to rearrange eight numbered tiles in a 3x3 grid by sliding them into the empty space.



2) Tower of Hanoi:

- A mathematical puzzle where you have three pegs and a number of disks of different sizes which can be slid onto any peg. The puzzle starts with the disks in a neat stack in ascending order of size on one peg, with the smallest disk on top, creating a conical shape.
- The mission is to move all the disks to some other tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −
- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.



The Tower of Hanoi puzzle with n disks can be solved in minimum $\mathbf{2^n-1}$ steps. This presentation shows that a puzzle with 3 disks has taken $\mathbf{2^3 - 1 = 7}$ steps.

A Recursive algorithm for the TOH is:

```
START
Procedure Hanoi(disk, source, dest, aux)
  IF disk == 1, THEN
    move disk from source to dest
  ELSE
    Hanoi(disk - 1, source, aux, dest)    // Step 1
    move disk from source to dest         // Step 2
    Hanoi(disk - 1, aux, dest, source)    // Step 3
  END IF
END Procedure
STOP
```

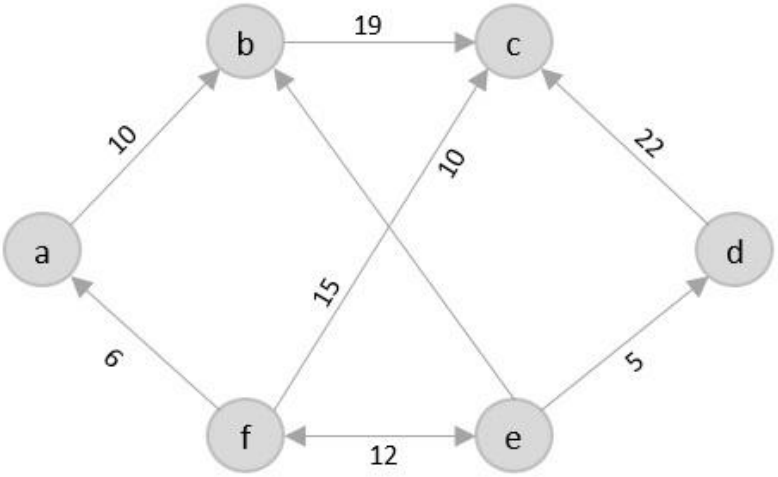
Python Implementations of TOH

```
def hanoi(n, f, to, via):
    if n == 1:
        print("Move disk 1 from",f,"to",to);
    else:
```

```
    hanoi(n-1, f, via, to)
    print("Move disk",n,"from",f,"to",to);
    hanoi(n-1, via, to, f)
n = 3
f = 'A'
to = 'B'
via = 'C'
hanoi(n, f, via, to)
```

3) Traveling Salesman Problem (TSP):

- Given a list of cities and the distances between each pair of cities, the task is to find the shortest possible route that visits each city exactly once and returns to the original city.
- The traveling salesman problem is a graph computational problem where the salesman needs to visit all cities (represented using nodes in a graph) in a list just once and the distances (represented using edges in the graph) between all these cities are known. The solution that is needed to be found for this problem is the shortest possible route in which the salesman visits all the cities and returns to the origin city.
- If you look at the graph below, considering that the salesman starts from the vertex ‘a’, they need to travel through all the remaining vertices b, c, d, e, f and get back to ‘a’ while making sure that the cost taken is minimum.



There are various approaches to find the solution to the traveling salesman problem: naive approach, greedy approach, dynamic programming approach, etc. In this tutorial we will be learning about solving traveling salesman problems using a greedy approach.

Traveling Salesperson Algorithm

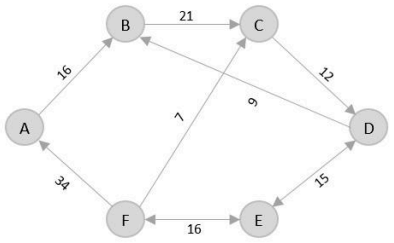
As the definition for greedy approach states, we need to find the best optimal solution locally to figure out the global optimal solution. The inputs taken by the algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges. The shortest path of graph G starting from one vertex returning to the same vertex is obtained as the output.

Algorithm

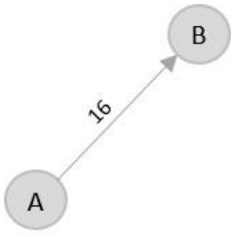
Traveling salesman problem takes a graph G {V, E} as an input and declares another graph as the output (say G') which will record the path the salesman is going to take from one node to another.
The algorithm begins by sorting all the edges in the input graph G from the least distance to the largest distance.
The first edge selected is the edge with least distance, and one of the two vertices (say A and B) being the origin node (say A).
Then among the adjacent edges of the node other than the origin node (B), find the least cost edge and add it onto the output graph.
Continue the process with further nodes making sure there are no cycles in the output graph and the path reaches back to the origin node A.
However, if the origin is mentioned in the given problem, then the solution must always start from that node only. Let us look at some example problems to understand this better.

Examples

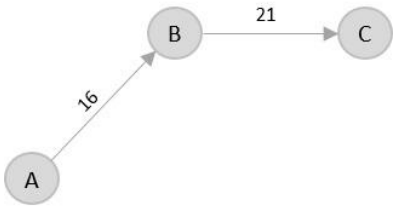
Consider the following graph with six cities and the distances between them −



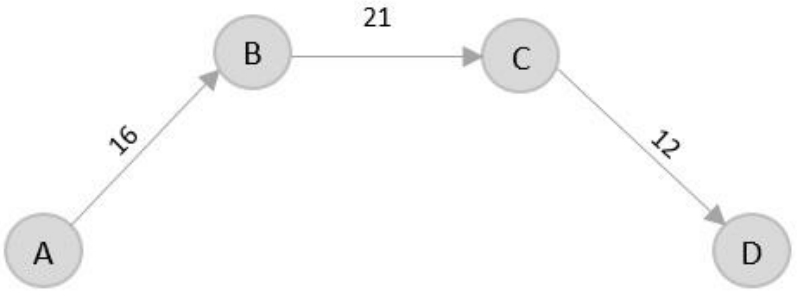
From the given graph, since the origin is already mentioned, the solution must always start from that node. Among the edges leading from A, A → B has the shortest distance.



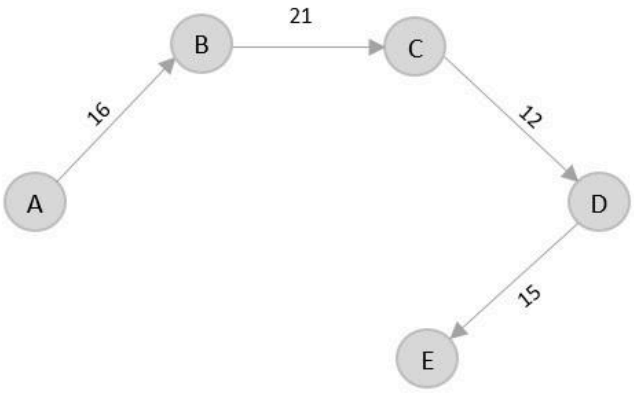
Then, B → C has the shortest and only edge between, therefore it is included in the output graph.



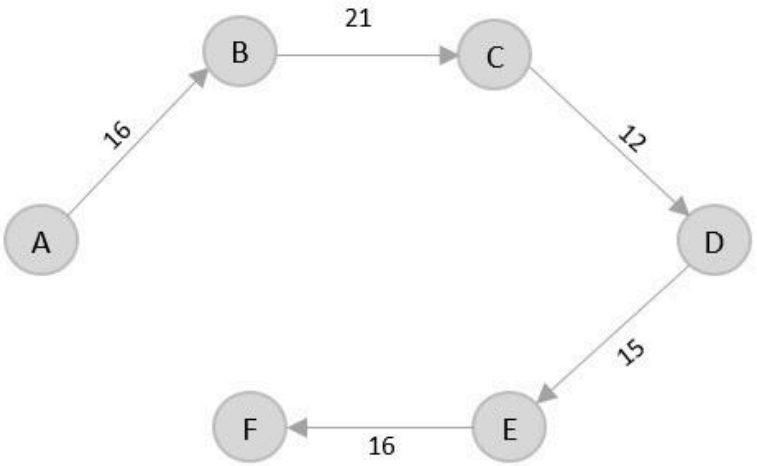
There's only one edge between C → D, therefore it is added to the output graph.



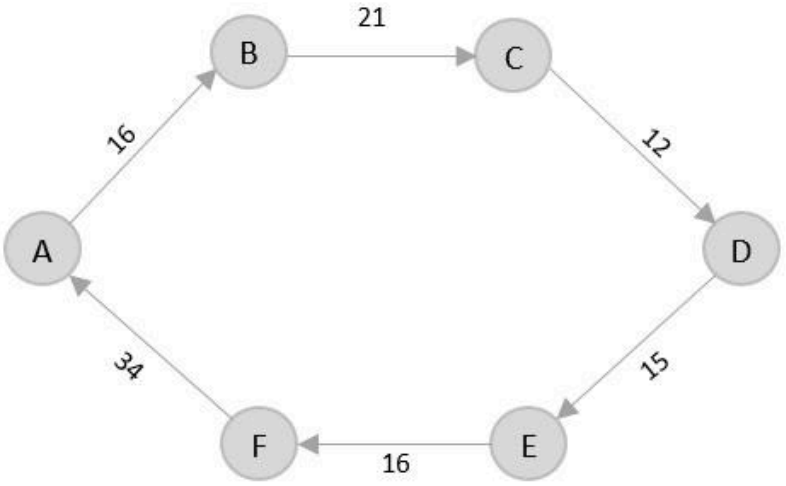
There's two outward edges from D. Even though D → B has a lower distance than D → E, B is already visited once and it would form a cycle if added to the output graph. Therefore, D → E is added into the output graph.



There's only one edge from e, that is E → F. Therefore, it is added into the output graph.



Again, even though F → C has a lower distance than F → A, F → A is added into the output graph in order to avoid the cycle that would form and C is already visited once.



The shortest path that originates and ends at A is A → B → C → D → E → F → A

The cost of the path is: 16 + 21 + 12 + 15 + 16 + 34 = 114.

Even though the cost of a path could be decreased if it originates from other nodes, the question is not raised with respect to that.

4) Water-jug Problems:

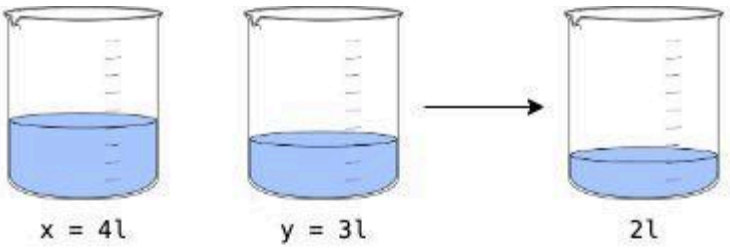
- We can fill a jug from the pump.
- We can pour water out of a jug to the ground.
- We can pour water from one jug to another.
- There is no measuring device available.

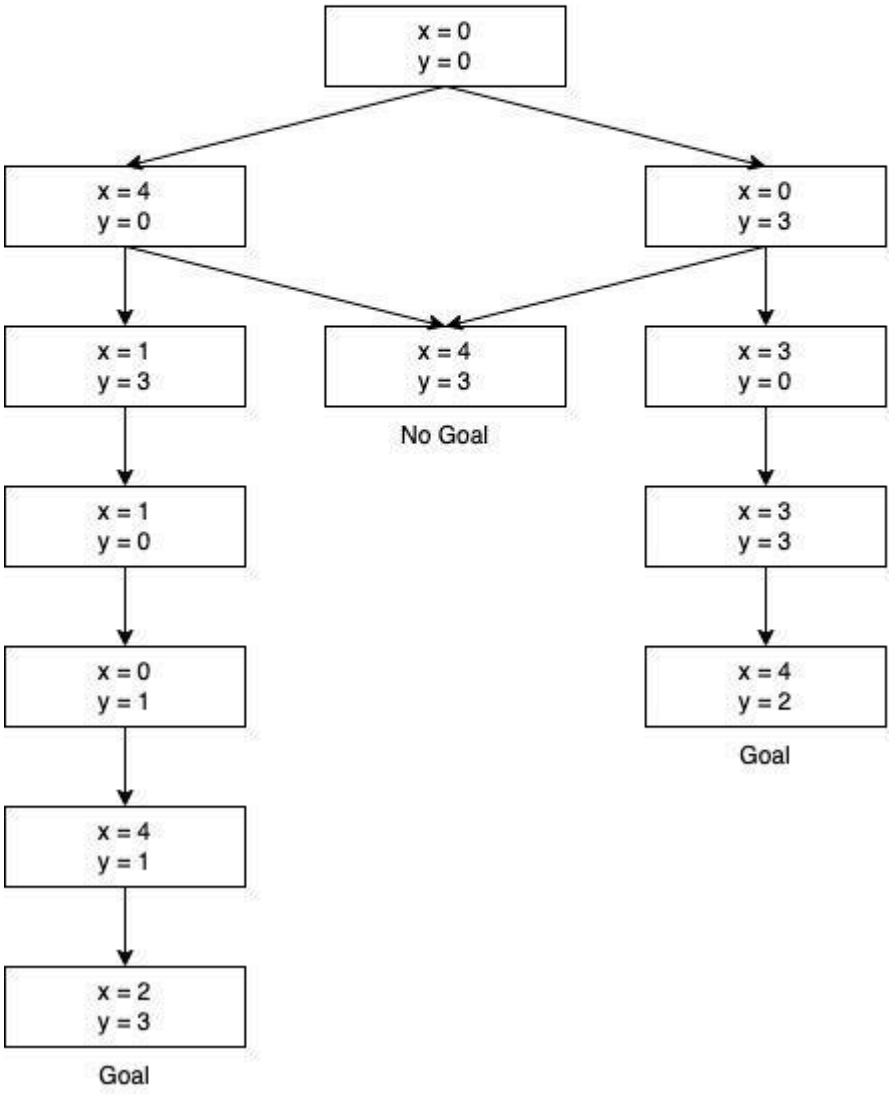
Operators — we must define a set of operators that will take us from one state to another.

Sr.	Current State	Next State	Descriptions
1	(x, y) if x < 4	(4, y)	Fill the 4 gallon jug
2	(x, y) if y < 3	(x, 3)	Fill the 3 gallon jug
3	(x, y) if x > 0	(x - d, y)	Pour some water out of the 4 gallon jug
4	(x, y) if y > 0	(x, y - d)	Pour some water out of the 3 gallon jug
5	(x, y) if y > 0	(0, y)	Empty the 4 gallon jug
6	(x, y) if y > 0	(x, 0)	Empty the 3 gallon jug on the ground
7	(x, y) if x + y >= 4 and y > 0	(4, y - (4 - x))	Pour water from the 3 gallon jug into the 4 gallon jug until the 4 gallon jug is full
8	(x, y) if x + y >= 3 and x > 0	(x - (3 - y), 3)	Pour water from the 4 gallon jug into the 3-gallon jug until the 3 gallon jug is full
9	(x, y) if x + y <= 4 and y > 0	(x + y, 0)	Pour all the water from the 3 gallon jug into the 4 gallon jug
10	(x, y) if x + y <=3 and x > 0	(0, x + y)	Pour all the water from the 4 gallon jug into the 3 gallon jug
11	(0, 2)	(2, 0)	Pour the 2 gallons from 3 gallon jug into the 4 gallon jug
12	(2, y)	(0, y)	Empty the 2 gallons in the 4 gallon jug on the ground

Initial state=(0,0)

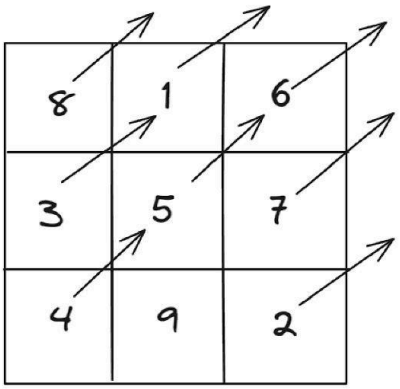
Goal state=(2,n) or (n,2)


x = 4l
y = 3l
2l


x = 0
y = 0
x = 4
y = 0
x = 0
y = 3
x = 1
y = 3
x = 4
y = 3
No Goal
x = 3
y = 0
x = 1
y = 0
x = 3
y = 3
x = 0
y = 1
x = 4
y = 2
Goal
x = 4
y = 1
x = 2
y = 3
Goal

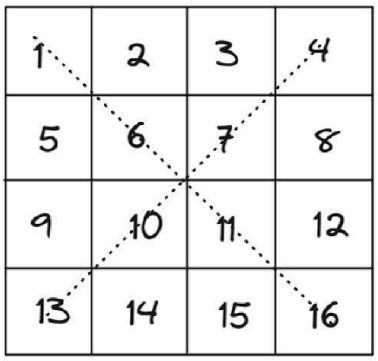
Magic Squares

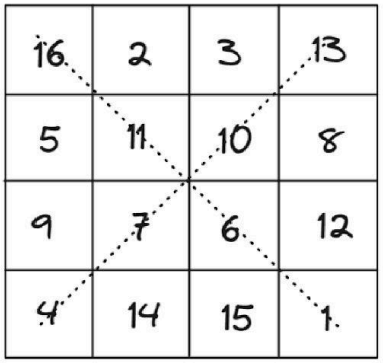
I. 3 x 3



Step 1: Place the first number i.e., 1 in the middle of the box (if the box is made up of odd number).
Step2: Then point the arrow of the number (i.e., 1) to the right side.
Step3: Place the second number at the bottom of the column where the arrow is pointed.
Step4: Place all the numbers till all the boxes are filled.
Step5: Stop

II. 4 x 4





Step1: Place all the values from number 1 to 16 serially from the top left corner of the box.

Step2: Now make an imaginary or dotted line in the diagonal section from top left to bottom right and top right to bottom left.

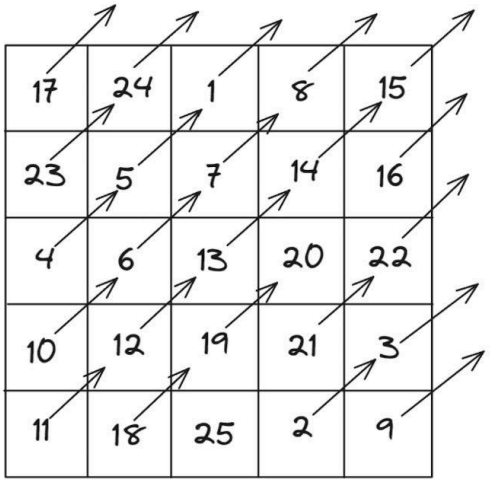
Step3: Now flip the numbers placed in within the diagonal line in opposite direction as shown in figure.

PS: if number 1 was in top left then it goes to bottom right and vice versa.

Step4: After the new numbers in diagonal section is placed then place the other numbers serially.

Step5: Stop

III. 5 x 5



Step 1: Place the first number i.e., 1 in the middle of the box (if the box is made up of odd number).
Step2: Then point the arrow of the number (i.e., 1) to the right side.
Step3: Place the second number at the bottom of the column where the arrow is pointed.
Step4: Place all the numbers till all the boxes are filled.
Step5: Stop

IV. 6 x 6

1	35	3	4	32	6
30	8	28	27	11	25
13	23	15	16	20	18
19	17	21	22	14	24
12	26	10	9	29	7
31	5	33	34	2	36

Step1: Make the box according to the given criteria.

Step2: Divide the box from the middle with having 3x3 boxes inside the 6x6 box.

Step3: Then create the imaginary or dotted line for the whole box in diagonal section.

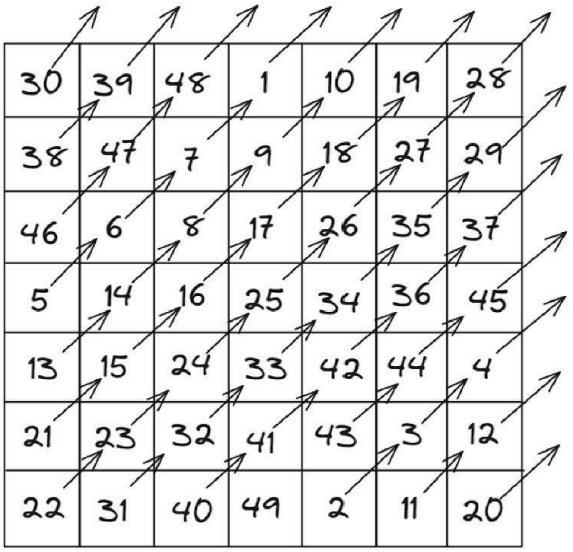
Step4: Place the numbers serially from top left corner to the bottom right corner, but the number should be placed on those blocks where dotted lines are diagonally placed. (Eg: 1 and 3 are placed on those blocks where the diagonal dotted line passes)

Step5: Assume that the bottom right corner (where number 36 is placed) is 1 (imagine it), then serially fill the other numbers which are left to be filled out, from left to right side.

PS: The numbers already present in the diagonal section should not be repeated.

Step6: Stop

V. 7 x 7



Step 1: Place the first number i.e., 1 in the middle of the box (if the box is made up of odd number).
Step2: Then point the arrow of the number (i.e., 1) to the right side.
Step3: Place the second number at the bottom of the column where the arrow is pointed.
Step4: Place all the numbers till all the boxes are filled.
Step5: Stop

VI. 8 x 8

1	63	62	4	5	59	58	8
56	10	11	53	52	14	15	49
48	18	19	45	44	22	23	41
25	39	38	28	29	35	34	32
33	31	30	36	37	27	26	40
24	42	43	21	20	46	47	17
16	50	51	13	12	54	55	9
57	7	6	60	61	3	2	64

Step1: Make the box according to the given criteria.

Step2: Divide the box from the middle with having 4x4 boxes inside the 8x8 box.

Step3: Then create the imaginary or dotted line for the whole box in diagonal section.

Step4: Place the numbers serially from top left corner to the bottom right corner, but the number should be placed on those blocks where dotted lines are diagonally placed. (Eg: 1 and 3 are placed on those blocks where the diagonal dotted line passes)

Step5: Assume that the bottom right corner (where number 36 is placed) is 1 (imagine it), then serially fill the other numbers which are left to be filled out , from left to right side.

PS: The numbers already present in the diagonal section should not be repeated.

Step6: Stop

Types of search algorithms

Now let's see the types of the search algorithm.

Based on the search problems, we can classify the search algorithm as

1. Uninformed search
2. Informed search

Uninformed search algorithms

- The uninformed search algorithm does not have any domain knowledge such as closeness, location of the goal state, etc. it behaves in a brute-force way.
- It only knows the information about how to traverse the given tree and how to
- find the goal state.
- This algorithm is also known as the Blind search algorithm
- or Brute -Force algorithm.

The uninformed search strategies are of six types.

They are-

- Breadth-first search
- Depth-first search
- Depth-limited search
- Iterative deepening depth-first search
- Bidirectional search
- Uniform cost 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.
- The BFS algorithm starts searching from the root node of the tree and expands all successor nodes at the current level before moving to nodes of the 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.

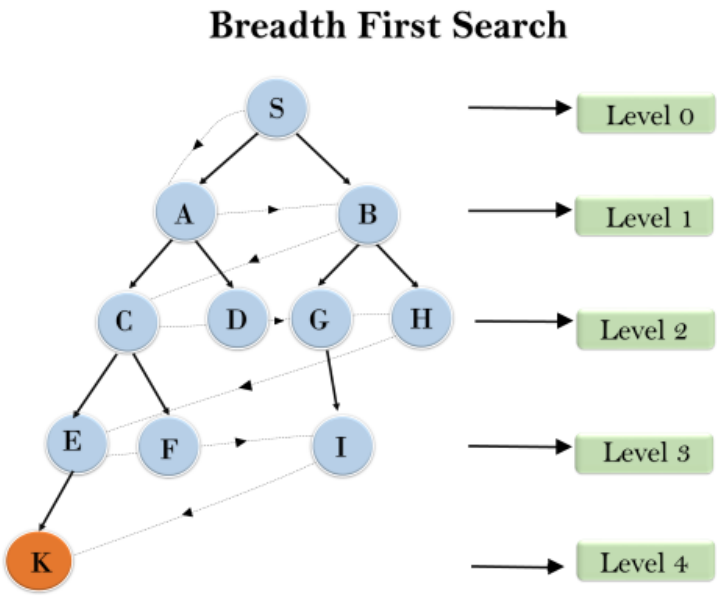
Disadvantages:

- 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.

2. 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.

Note: Backtracking is an algorithm technique for finding all possible solutions using recursion.

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 the goal node than the BFS algorithm (if it traverses in the right path).

Disadvantage:

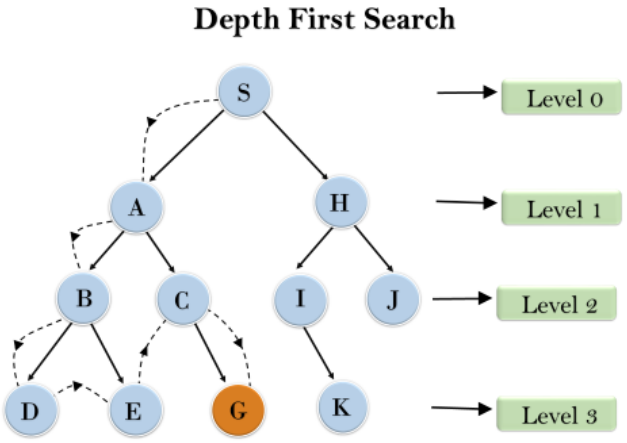
- There is the possibility that many states keep reoccurring, and there is no guarantee of finding the solution.
- The DFS algorithm goes for deep down searching and sometimes it may go to the infinite loop.

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 the goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found the 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 a 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 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 be treated as it has no successor nodes further.

Depth-limited search can be terminated with two Conditions of failure:

- Standard failure value: It indicates that the problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.

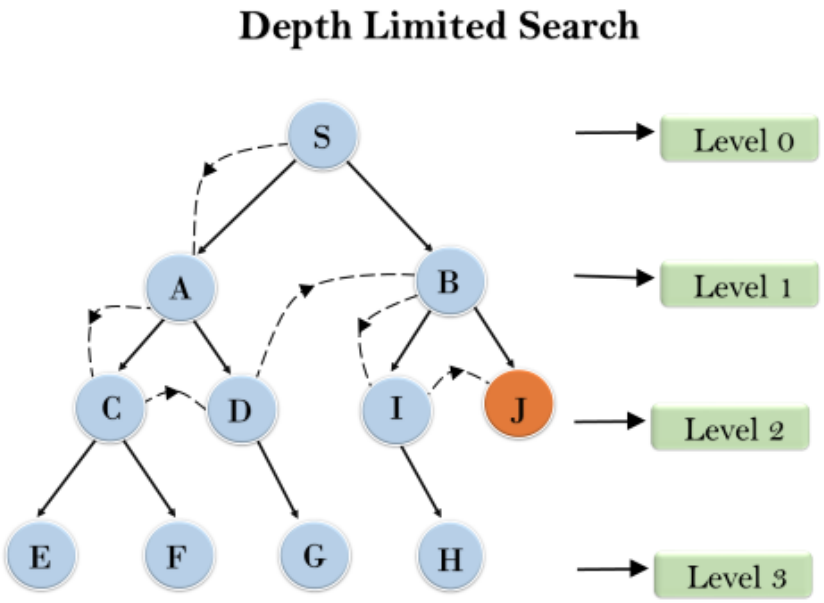
Advantages:

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^{ℓ})**.

Space Complexity: Space complexity of DLS algorithm is O**(b×ℓ)**.

Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if ℓ>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 from 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 the BFS algorithm if the path cost of all edges is the same.

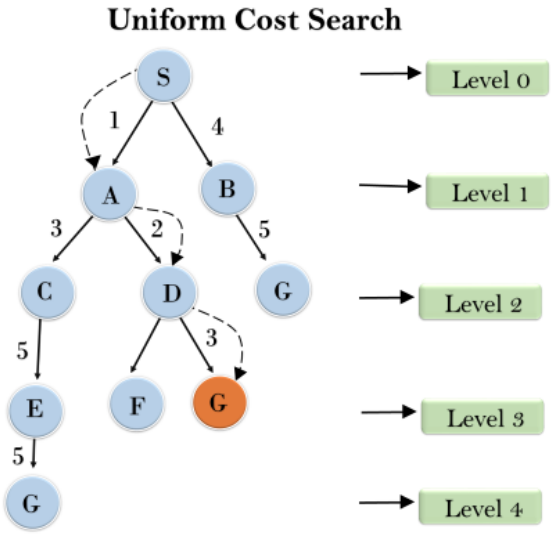
Advantages:

- 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 involved in searching and is 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$\mathbf{O(b^{1 + \lfloor C^*/\varepsilon \rfloor})}$/.

Space Complexity:

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

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 for uninformed search when the search space is large, and depth of the goal node is unknown.

Advantages:

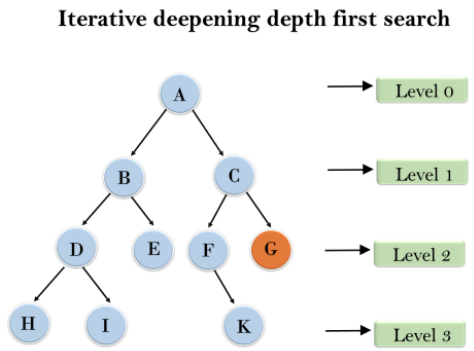
- It Combines the benefits of BFS and DFS search algorithms in terms of fast search and memory efficiency.

Disadvantages:

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

Example:

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



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 if the 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:

The 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

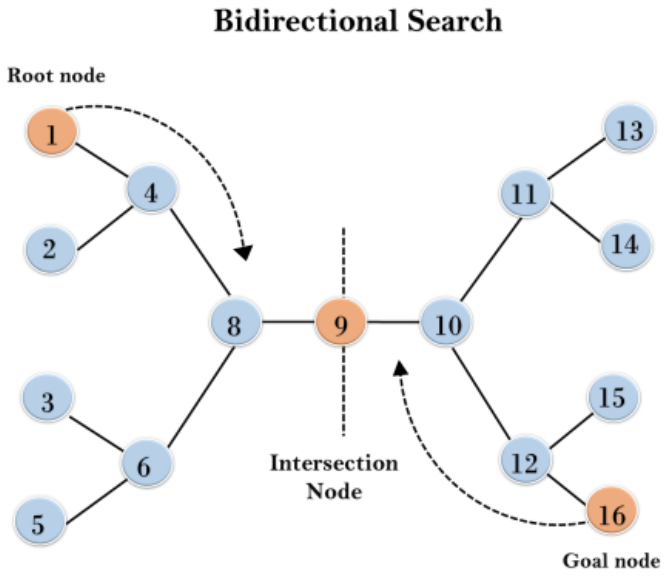
Disadvantages:

- 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, a 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.



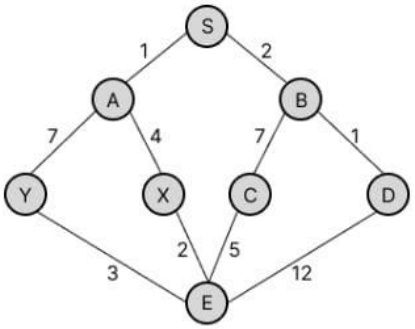
Completeness: Bidirectional Search is complete if we use BFS in both searches.

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



n	h(n)
A	5
B	6
C	4
D	15
X	5
Y	8

Greedy:

→ This algorithm visits the next state based on heuristics function f(n) = h(n) with the high lowest heuristic value often called Greedy.

→ It doesn't consider the cost of the path to the particular state. All it cares about is that which next state from the current state has the lowest heuristics.

Start node : S
S → A : f(A) = h(A) = 5
S → B : f(B) = h(B) = 6 (x)

Expand S → A then,
S → A → Y : f(Y) =8 = h(Y) (x)
S → A → X : f(X) =5 = h(X)

again,
Expanding S → A → X
S → A → A → X → E : f(E) = h(E) = 0 (Goal)
Same

So, Shortest Best Path is S → A → X → E , and (Path cost = 7) is the best cost path.

A* Search:

→ This algorithm visits the next state based on heuristics function f(n) = h(n) + g(n), where h(n) component is the same heuristics applied as in the Greedy but g(n) component is the path from the initial state to the particular state.

→ It doesn't choose next state only with the lowest heuristics value but it selects the one that gives the lowest value when considering its heuristics and cost of getting to that state.

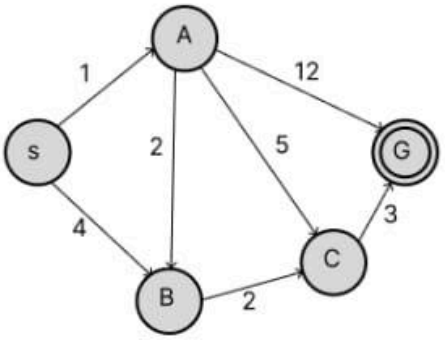
Start node : S
S → A : f(A) = g(A) + h(A) = 1 + 5 = 6
S → B : f(B) = g(B) + h(B) = 2 + 6 = 8

Expand S → A then,
S → A → X : f(X) = g(X) + h(X) = (1 + 4) + 5 = 10
S → A → Y : f(Y) = g(Y) + h(Y) = (1 + 7) + 8 = 16

Expanding S → B
S → B → C : f(C) = 13
S → B → D : f(D) = 18

again,
Expanding S → A → X
S → A → X → E : f(E) = 7 + 0 = 7

So, Shortest Best Path is S → A → X → E , and (Path cost = 7) is the best cost path.



Node	h(n)
S	7
A	6
B	2
C	1
G	0

Greedy:

Start node : S
S → A : f(A) = 6
S → B : f(B) = 2

Expand S → B then,
S → B → C : f(C) = 1

again,
Expanding S → B → C
S → B → C → G : f(G) = h(G) = 0

So, Shortest Best Path is S → B → C → G , and (Path cost = 9) is the best cost path.

A* Search:

Start node : S
S → A : f(A) = g(A) + h(A) = 1 + 6 = 7
S → B : f(B) = g(B) + h(B) = 4 + 2 = 6

Expanding S → B,
S → B → C : f(C) = 7

Expand S → A then,
S → A → B : f(B) = 5
S → A → C : f(C) = 7
S → A → G : f(G) = 13

Expanding S → B → C
S → B → C → G : f(G) = 9

again,
Expanding S → A → B
S → A → B → C : f(C) = 6

again,
Expanding S → A → B → C
S → A → B → C → G : f(G) = 8

again,
Expanding S → A → C
S → A → C → G → : f(G) = 9

So, Shortest Best Path is S → A → B → C → G , and (Path cost = 8) is the best cost path.

#Greedy Vs A* Search

Similarities:-
→Both selects/expands node with least evaluation function.
→Both are Informed Search.
→Best First Search Algorithm.

Greedy Search	A* Search
→ It considers heuristic value only.	→ It considers heuristic value only.
→Evaluation function f(n) = h(n)	→Evaluation function f(n) = g(n) + h(n) where, g(n) = path cost from start node to node 'n'. h(n) = heuristic value of node 'n'.
→Comparatively simple & faster.	→Comparatively slower & complex
→No Backtracking of path.	→Considers Backtracking.
→Comparatively low time-complexity & space-complexity	→Comparatively more time-complexity & space-complexity

Greedy Search Comparative study Analysis

- **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 Comparative study Analysis

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)

What is the Heuristic Function?

- If there are no specific answers to a problem or the time required to find one is too great, a heuristic function is used to solve the problem.
- A heuristic Function can be represented mathematically such as: f(n)= g(n) + h(n), where
- f(n)= estimated cost of the cheapest solution
- g(n)= cost to reach node n from the start state
- h(n)= cost to reach from node n to goal node
- The aim is to find a quicker or more approximate answer, even if it is not ideal. Put another way, utilizing a heuristic means trading accuracy for speed.
- A heuristic is a function that determines how near a state is to the desired state. Heuristics functions vary depending on the problem and must be tailored to match that particular challenge.
- The majority of AI problems revolve around a large amount of information, data, and constraints, and the task is to find a way to reach the goal state.
- The heuristics function in this situation informs us of the proximity to the desired condition.
- The distance formula is an excellent option if one needed a heuristic function to assess how close a location in a two-dimensional space was to the objective point.

Heuristics is a method of problem-solving where the goal is to come up with a workable solution in a feasible amount of time. Heuristic techniques strive for a rapid solution that stays within an appropriate accuracy range rather than a perfect solution.When it seems impossible to tackle a specific problem with a step-by-step approach, heuristics are utilized in AI (artificial intelligence) and ML (machine learning). Heuristic functions in AI prioritize speed above accuracy; hence they are frequently paired with optimization techniques to provide better outcomes.

Properties of a Heuristic Search Algorithm

Heuristic search algorithms have the following properties:

- **Admissible Condition:** If an algorithm produces an optimal result, it is considered admissible.
- **Completeness:** If an algorithm ends with a solution, it is considered complete.
- **Dominance Property:** If A1 and A2 are two heuristic algorithms and have h1 and h2 heuristic functions, respectively, then A1 Will dominate A2 if h1 is superior to h2 for all possible values of node n.
- **Optimality Property:** If an algorithm is thorough, allowable, and dominates the other algorithms, that'll be the optimal one and will unquestionably produce an optimal result.

Different Categories of Heuristic Search Techniques in AI

We can categorize the Heuristic Search techniques into two types:

Direct Heuristic Search Techniques

- Direct heuristic search techniques may also be called blind control strategy, blind search, and uninformed search.
- They utilize an arbitrary sequencing of operations and look for a solution throughout the entire state space. These include Depth First Search (DFS) and Breadth First Search (BFS).
- BFS is a heuristic search method to diagram data or quickly scan intersection or tree structures. DFS is predicated on the likelihood of last in, first out. Similarly, the LIFO stack data structure is used to complete the process in recursion.

Weak Heuristic Techniques

- Weak heuristic techniques are known as a Heuristic control strategy, informed search, and Heuristic search. These are successful when used effectively on the appropriate tasks and typically require domain-specific knowledge.
- To explore and expand, users require additional information to compute preferences across child nodes. A heuristic function is connected to each node.

Let's first look at some of the strategies we frequently see before detailing specific ones. Here are a few examples.

- **A* Search**
- **Best-first search**
- **Tabu search**
- **Bidirectional search**
- **Constant satisfaction problems**
- **Hill climbing**

An Example of Heuristic function in AI

A variety of issues can be solved using a heuristic function in AI.

Let's talk about some of the more popular ones.

Traveling Salesman Problem

What is the quickest path between each city and its starting point, given a list of cities and the distances between each pair of them?

This problem could be brute-forced for a small number of cities. But as the number of cities grows, finding a solution becomes more challenging.

This issue is well-solved by the nearest-neighbor heuristic, which directs the computer to always choose the closest unexplored city as the next stop on the path. While NN only sometimes offers the optimum solution, it is frequently near enough that the variation is insignificant to respond to the salesman's problem. This approach decreases TSP's complexity from O(n!) to O (n^2).

Search Engine

People have been interested in SEO as long as there have been search engines. Users want to quickly discover the information they need when utilizing a search engine. Search engines use heuristics to speed up the search process because such a staggering amount of data is available. A heuristic could initially attempt each alternative at each stage. Still, as the search progresses, it can quit at any point if the present possibility is inferior to the best solution already found. The search engine's accuracy and speed can be improved in this way.

Hill Climbing Algorithm in Artificial Intelligence

- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.
- Hill climbing algorithm is a technique which is used for optimizing mathematical problems. One of the widely discussed examples of Hill climbing algorithms is the Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.
- It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.
- A node of hill climbing algorithm has two components which are state and value.
- Hill Climbing is mostly used when a good heuristic is available.
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

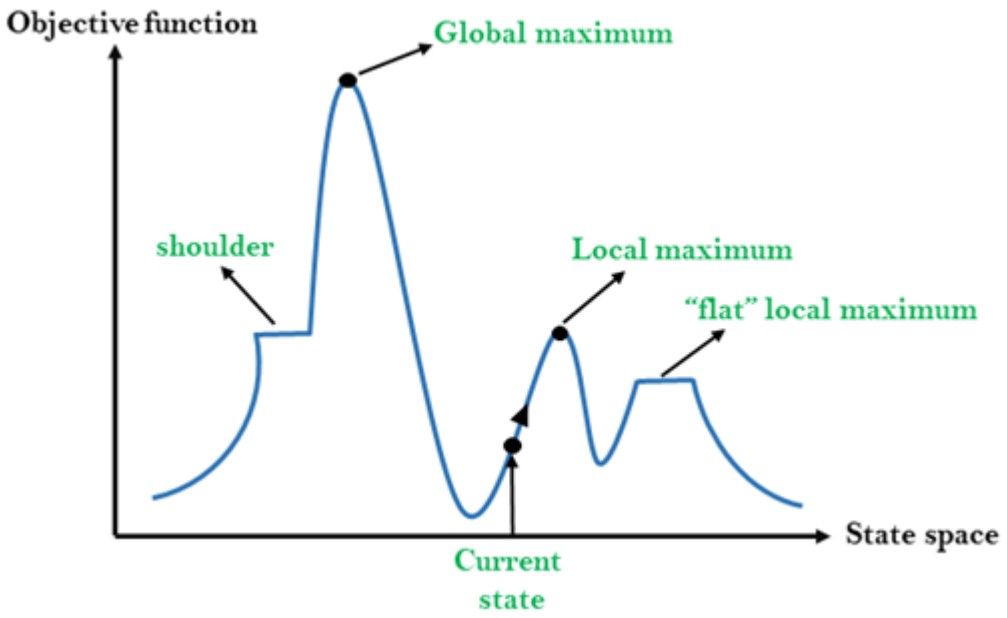
Features of Hill Climbing:

Following are some main features of Hill Climbing Algorithm:

- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

State-space Diagram for Hill Climbing:

- The state-space landscape is a graphical representation of the hill-climbing algorithm which is showing a graph between various states of algorithm and Objective function/Cost.
- On the Y-axis we have taken the function which can be an objective function or cost function, and state-space on the x-axis. If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum. If the function of the Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.



Different regions in the state space landscape:

- **Local Maximum:** Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.
- **Global Maximum:** Global maximum is the best possible state of the state space landscape. It has the highest value of objective function.
- **Current state:** It is a state in a landscape diagram where an agent is currently present.
- **Flat local maximum:** It is a flat space in the landscape where all the neighbor states of current states have the same value.
- **Shoulder:** It is a plateau region which has an uphill edge.

Types of Hill Climbing Algorithm:

- **Simple hill Climbing:**
- **Steepest-Ascent hill-climbing:**
- **Stochastic hill Climbing:**

1. Simple Hill Climbing:

Simple hill climbing is the simplest way to implement a hill climbing algorithm. It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and sets it as a current state. It only checks its one successor state, and if it finds better than the current state, then move else be in the same state. This algorithm has the following features:

- **Less time consuming**
- **Less optimal solution and the solution is not guaranteed**

Algorithm for Simple Hill Climbing:

- **Step 1: Evaluate the initial state, if it is a goal state then return success and Stop.**
- **Step 2: Loop Until a solution is found or there is no new operator left to apply.**
- **Step 3: Select and apply an operator to the current state.**
- **Step 4: Check new state:**
    a. **If it is a goal state, then return to success and quit.**
    b. **Else if it is better than the current state then assign a new state as a current state.**
    c. **Else if not better than the current state, then return to step2.**
- **Step 5: Exit.**

2. Steepest-Ascent hill climbing:

- The steepest-Ascent algorithm is a variation of a simple hill climbing algorithm. This algorithm examines all the neighboring nodes of the current state and selects one

neighbor node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbors

Algorithm for Steepest-Ascent hill climbing:

- **Step 1: Evaluate the initial state, if it is the goal state then return success and stop, else make the current state as the initial state.**
- **Step 2: Loop until a solution is found or the current state does not change.**
    a. **Let SUCC be a state such that any successor of the current state will be better than it.**
    b. **For each operator that applies to the current state:**
        a. **Apply the new operator and generate a new state.**
        b. **Evaluate the new state.**
        c. **If it is a goal state, then return it and quit, else compare it to the SUCC.**
        d. **If it is better than SUCC, then set a new state as SUCC.**
        e. **If the SUCC is better than the current state, then set the current state to SUCC.**
- **Step 5: Exit.**

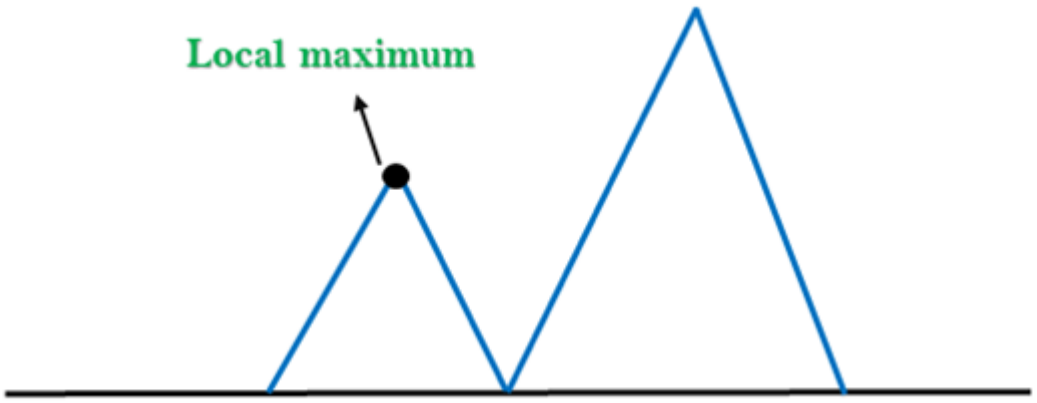
3. Stochastic hill climbing:

- Stochastic hill climbing does not examine all its neighbors before moving. Rather, this search algorithm selects one neighbor node at random and decides whether to choose it as a current state or examine another state.

Problems in Hill Climbing Algorithm:

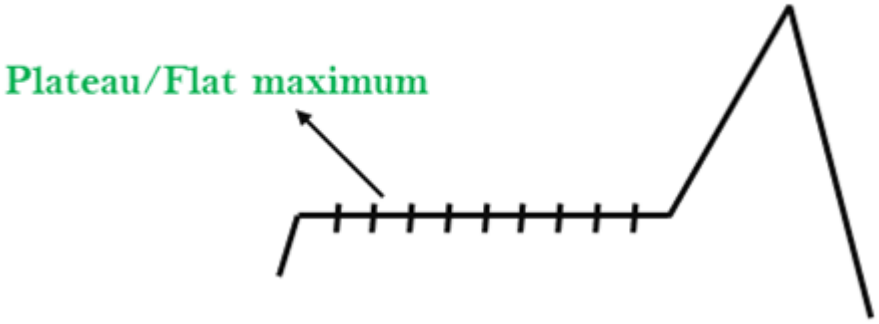
1. Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

Solution: Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.



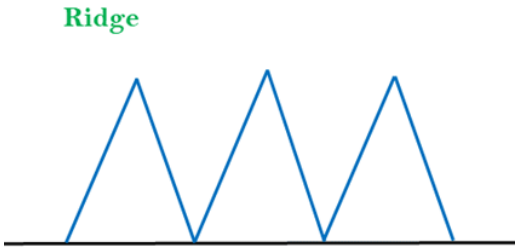
2. Plateau: A plateau is the flat area of the search space in which all the neighbor states of the current state contain the same value, because this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

Solution: The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find a non-plateau region.



3. Ridges: A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.

Solution: With the use of bidirectional search, or by moving in different directions, we can improve this problem.



Simulated Annealing:

- A hill-climbing algorithm which never makes a move towards a lower value is guaranteed to be incomplete because it can get stuck on a local maximum. And if the algorithm applies a random walk, by moving a successor, then it may be complete but not efficient. Simulated Annealing is an algorithm which yields both efficiency and completeness.
- In mechanical terms 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.

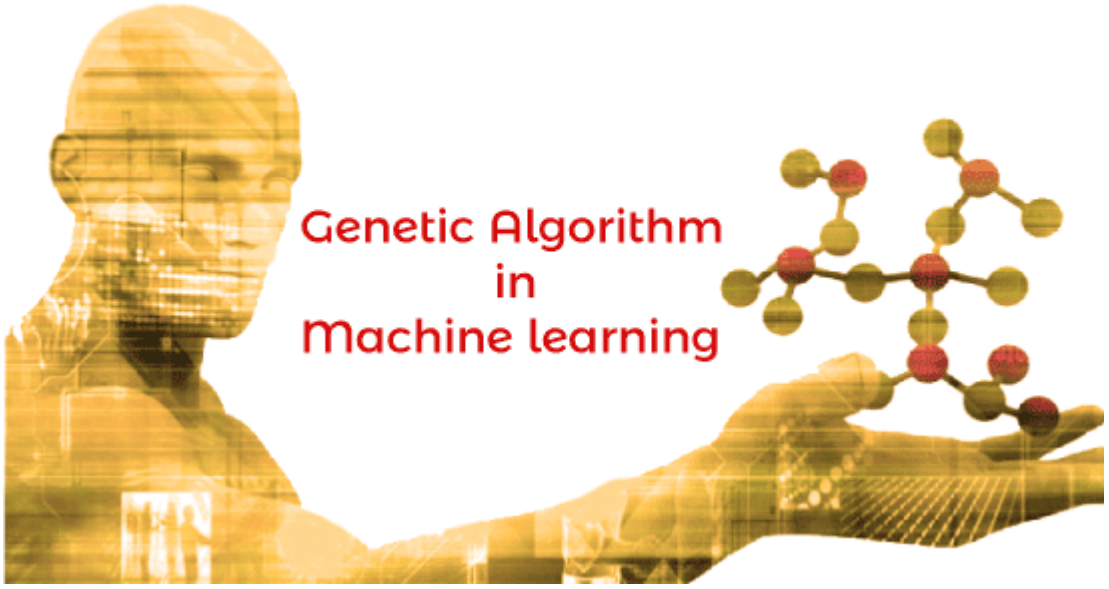
Comparative study of Informed Search

Algorithm	Description	Space Complexity	Time Complexity	Completeness	Optimality
Greedy Search	Chooses the node that appears to be the closest to the goal according to the heuristic function.	O(b^m)	O(b^m)	No	No
A* Search	A popular informed search algorithm that uses both the cost to reach a node and an estimate of the cost to the goal.	O(b^m)	O(b^m)	Yes	Yes
Hill Climbing Search	Iteratively improves a candidate solution by making small incremental changes until no further improvements can be made.	O(1)	High	No	No

- 'b' represents the branching factor (average number of successors per state).
- 'm' represents the maximum depth of the search tree.
- Hill Climbing Search's time complexity is highly dependent on the specific problem instance and the landscape of the search space, hence it's represented as "High". It can vary from linear to exponential depending on the problem. However, its space complexity is typically constant as it only maintains the current state.

Genetic Algorithm in Machine Learning

- *A genetic algorithm is an adaptive heuristic search algorithm inspired by "Darwin's theory of evolution in Nature*." It is used to solve optimization problems in machine learning. It is one of the important algorithms as it helps solve complex problems that would take a long time to solve.



- Genetic Algorithms are being widely used in different real-world applications, for example, Designing electronic circuits, code-breaking, image processing, and artificial creativity.
- In this topic, we will explain Genetic algorithm in detail, including basic terminologies used in Genetic algorithm, how it works, advantages and limitations of genetic algorithm, etc.

What is a Genetic Algorithm?

Before understanding the Genetic algorithm, let's first understand basic terminologies to better understand this algorithm:

- **Population:** Population is the subset of all possible or probable solutions, which can solve the given problem.
- **Chromosomes:** A chromosome is one of the solutions in the population for the given problem, and the collection of genes generates a chromosome.

- **Gene:** A chromosome is divided into a different gene, or it is an element of the chromosome.
- **Allele:** Allele is the value provided to the gene within a particular chromosome.
- **Fitness Function:** The fitness function is used to determine the individual's fitness level in the population. It means the ability of an individual to compete with other individuals. In every iteration, individuals are evaluated based on their fitness function.
- **Genetic Operators:** In a genetic algorithm, the best individual mate to regenerate offspring better than parents. Here genetic operators play a role in changing the genetic composition of the next generation.
- **Selection:** After calculating the fitness of every existent in the population, a selection process is used to determine which of the individualities in the population will get to reproduce and produce the seed that will form the coming generation.

Types of selection styles available

- Roulette wheel selection
- Event selection
- Rank- grounded selection

So, now we can define a genetic algorithm as a heuristic search algorithm to solve optimization problems. It is a subset of evolutionary algorithms, which is used in computing. A genetic algorithm uses genetic and natural selection concepts to solve optimization problems.

How Does Genetic Algorithm Work?

The genetic algorithm works on the evolutionary generational cycle to generate high-quality solutions. These algorithms use different operations that either enhance or replace the population to give an improved fit solution.

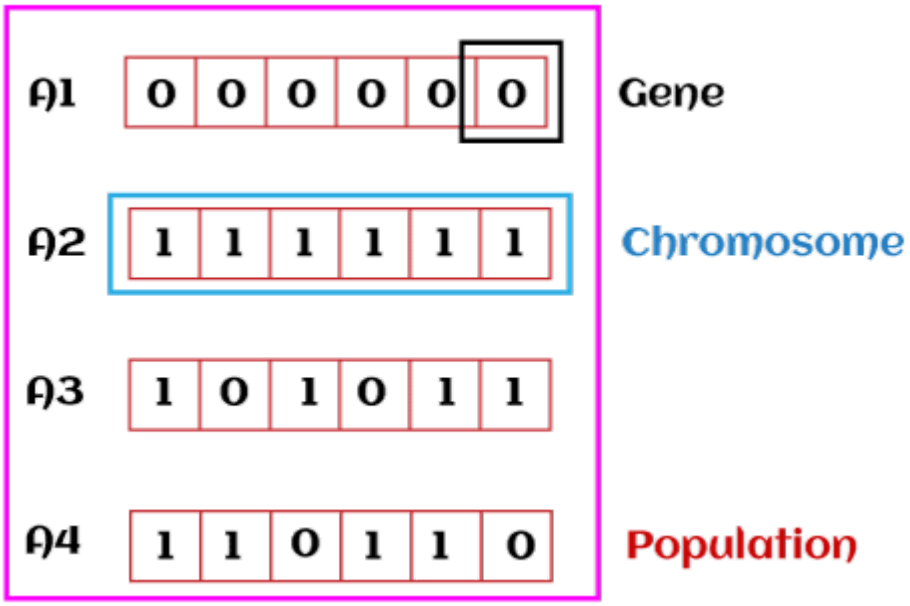
It basically involves five phases to solve the complex optimization problems, which are given as below:

- **Initialization**

- **Fitness Assignment**
- **Selection**
- **Reproduction**
- **Termination**

1. Initialization

The process of a genetic algorithm starts by generating the set of individuals, which is called population. Here each individual is the solution for the given problem. An individual contains or is characterized by a set of parameters called Genes. Genes are combined into a string and generate chromosomes, which is the solution to the problem. One of the most popular techniques for initialization is the use of random binary strings.



2. Fitness Assignment

Fitness function is used to determine how fit an individual is? It means the ability of an individual to compete with other individuals. In every iteration, individuals are evaluated based on their fitness function. The fitness function provides a fitness score to each individual. This score further determines the probability of being selected for reproduction. The high the fitness score, the more chances of getting selected for reproduction.

3. Selection

The selection phase involves the selection of individuals for the reproduction of offspring. All the selected individuals are then arranged in a pair of two to increase reproduction. Then these individuals transfer their genes to the next generation.

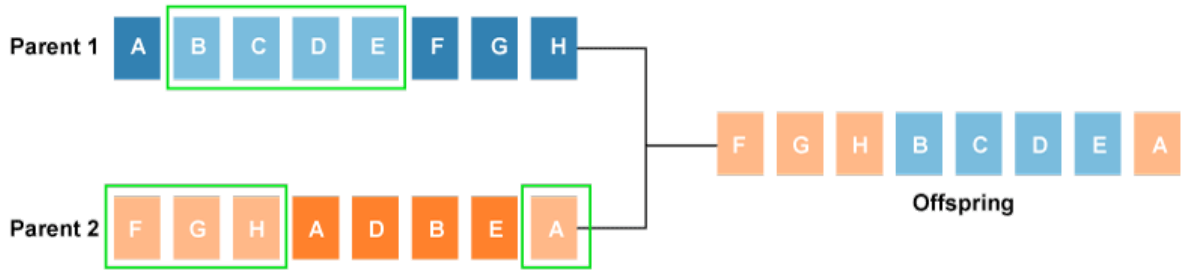
There are three types of Selection methods available, which are:

- **Roulette wheel selection**
- **Tournament selection**
- **Rank-based selection**

4. Reproduction

After the selection process, the creation of a child occurs in the reproduction step. In this step, the genetic algorithm uses two variation operators that are applied to the parent population. The two operators involved in the reproduction phase are given below:

- **Crossover**: The crossover plays a most significant role in the reproduction phase of the genetic algorithm. In this process, a crossover point is selected at random within the genes. Then the crossover operator swaps genetic information of two parents from the current generation to produce a new individual representing the offspring.



The genes of parents are exchanged among themselves until the crossover point is met. These newly generated offspring are added to the population. This process is also called crossover. Types of crossover styles available:

- **One point crossover**
- **Two-point crossover**
- **Livery crossover**

- **Inheritable Algorithms crossover**

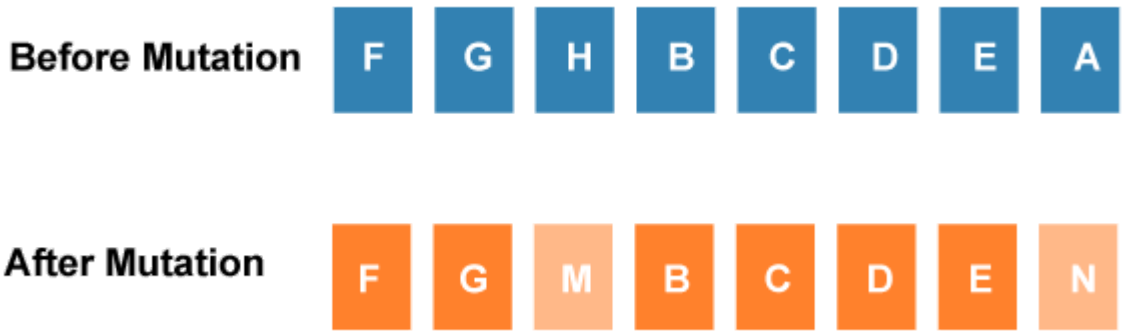
- **Mutation**

  The mutation operator inserts random genes in the offspring (new child) to maintain the diversity in the population. It can be done by flipping some bits in the chromosomes.

  Mutation helps in solving the issue of premature convergence and enhances diversification. The below image shows the mutation process:

  Types of mutation styles available,

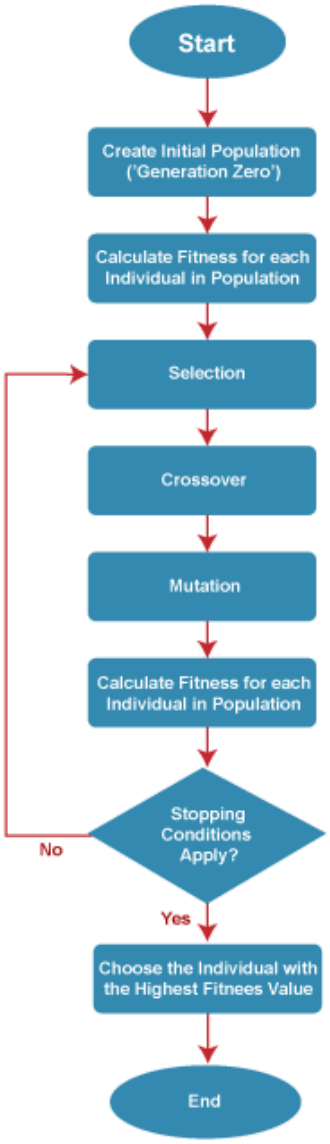
  - Flip bit mutation
  - Gaussian mutation
  - Exchange/Swap mutation



5. Termination

After the reproduction phase, a stopping criterion is applied as a base for termination. The algorithm terminates after the threshold fitness solution is reached. It will identify the final solution as the best solution in the population.

General Workflow of a Simple Genetic Algorithm



Advantages of Genetic Algorithm

- The parallel capabilities of genetic algorithms are best.
- It helps in optimizing various problems such as discrete functions, multi-objective problems, and continuous functions.
- It provides a solution for a problem that improves over time.
- A genetic algorithm does not need derivative information.

Limitations of Genetic Algorithms

- Genetic algorithms are not efficient algorithms for solving simple problems.
- It does not guarantee the quality of the final solution to a problem.
- Repetitive calculation of fitness values may generate some computational challenges.

Difference between Genetic Algorithms and Traditional Algorithms

- A search space is the set of all possible solutions to the problem. In the traditional algorithm, only one set of solutions is maintained, whereas, in a genetic algorithm, several sets of solutions in search space can be used.
- Traditional algorithms need more information in order to perform a search, whereas genetic algorithms need only one objective function to calculate the fitness of an individual.
- Traditional Algorithms cannot work parallelly, whereas genetic Algorithms can work parallelly (calculating the fitness of the individualities are independent).
- One big difference in genetic Algorithms is that rather than operating directly on seeker results, inheritable algorithms operate on their representations (or rendering), frequently pertained to as chromosomes.
- One of the big differences between traditional algorithms and genetic algorithms is that it does not directly operate on candidate solutions.
- Traditional Algorithms can only generate one result in the end, whereas Genetic Algorithms can generate multiple optimal results from different generations.
- The traditional algorithm is not more likely to generate optimal results, whereas Genetic algorithms do not guarantee to generate optimal global results, but also there is a great possibility of getting the optimal result for a problem as it uses genetic operators such as Crossover and Mutation.
- Traditional algorithms are deterministic in nature, whereas Genetic algorithms are probabilistic and stochastic in nature.

$Maximize\ f(x) = 15x - x^2\ where\ 0 \leq x \leq 16\ \ n = 4\ (population\ size)$

Step 1 → *Initilization* : *selecting* 4 *random numbers*

0, 4, 3, 14

now,

Step 2 → *Fitness calculation* $f(x) = 15x - x^2$

Where,

$x = 0, \quad f(x) = 0$ ……. ④

$x = 4, \quad f(x) = 44$ ……. ①

$x = 3, \quad f(x) = 36$ ……. ②

$x = 14, \quad f(x) = 14$ ……. ③

now,

In Binary [4-bit representant]

$A : 0 \rightarrow 0000$

$B : 4 \rightarrow 0100$

$C : 3 \rightarrow 0011$

$D : 14 \rightarrow 1110$

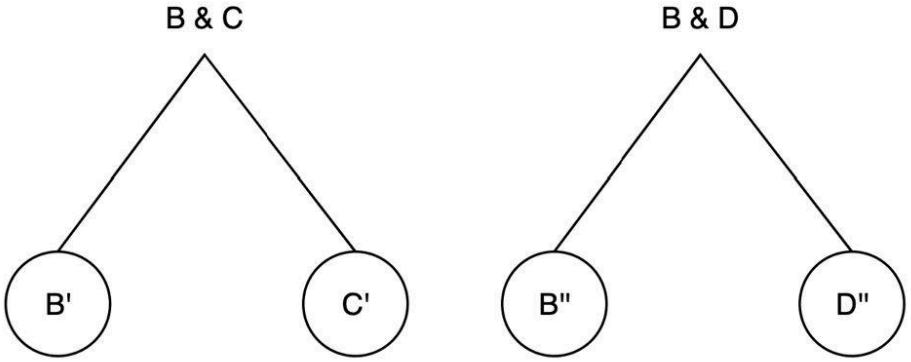
0 ……. ④

44 ……. ①

36 ……. ②

14 ……. ③

Step 3 → Selection : 2 parent pairs



Crossover

B = 0100

C = 0011

B = 0100

D = 1110

B′ = 0111 =7

B′′ = 0110 =6

C′ = 0000 = 0

D′′ = 1100= 12

Now,

$x \rightarrow f(x) = 15x - x^2$
$7 \rightarrow 56$
$6 \rightarrow 54$
$0 \rightarrow 0$
$12 \rightarrow 36$

when $x = 7$ $f(x) = 56$ *is maximum.*

$Maximize\ f(x) = x^2\ for\ 0 \leq x \leq 31 \quad n = 6$

$Step\ 1 \rightarrow Initilization : selecting\ 6\ random\ numbers,$

$0, 2, 4, 9, 24, 25$

$now,$

$Step\ 2 \rightarrow Fitness\ calculation\ f(x) = x^2$

$Where,$

$x = 0, \quad f(x) = 0 \ \ldots\ldots\ldots\ldots\ldots \textcircled{6}$

$x = 2, \quad f(x) = 4 \ \ldots\ldots\ldots\ldots\ldots \textcircled{5}$

$x = 4, \quad f(x) = 16 \ \ldots\ldots\ldots\ldots \textcircled{4}$

$x = 9, \quad f(x) = 81 \ \ldots\ldots\ldots\ldots \textcircled{3}$

$x = 24, \quad f(x) = 516 \ \ldots\ldots\ldots\ldots \textcircled{2}$

$x = 25, \quad f(x) = 625 \ \ldots\ldots\ldots\ldots \textcircled{1}$

$now,$

$In\ Binary\ [5\text{-}bit\ representant]$

$A : 0 \rightarrow 00000$ — $0 \ \ldots\ldots\ldots\ldots\ldots \textcircled{6}$

$B : 2 \rightarrow 00010$ — $4 \ \ldots\ldots\ldots\ldots\ldots \textcircled{5}$

$C : 4 \rightarrow 00100$ — $16 \ \ldots\ldots\ldots\ldots\ldots \textcircled{4}$

$D : 9 \rightarrow 01001$ — $81 \ \ldots\ldots\ldots\ldots\ldots \textcircled{3}$

$E : 24 \rightarrow 11000$ — $516 \ \ldots\ldots\ldots\ldots\ldots \textcircled{2}$

$F : 25 \rightarrow 11001$ — $625 \ \ldots\ldots\ldots\ldots\ldots \textcircled{1}$

Step 3 → Selection :

$F\ \&\ E, \qquad F\&D, \qquad F\&C$

Step 4 → Crossover

F = 11001	F = 11001	F = 11001
E = 11000	D = 01001	C = 00100

F' = 11000 = 24	F'' = 11001 = 25	F''' = 11100 = 28
E' = 11001 = 25	E'' = 01001 = 9	E''' = 00001 = 1

Step 5 → Mutation [there is same value so do mutation]

$1\,2\,3\,4\,5 \rightarrow index$
$\downarrow\downarrow\downarrow\downarrow\downarrow$

E' = 11001 [∴ we can change any index to represent it's uniqueness]

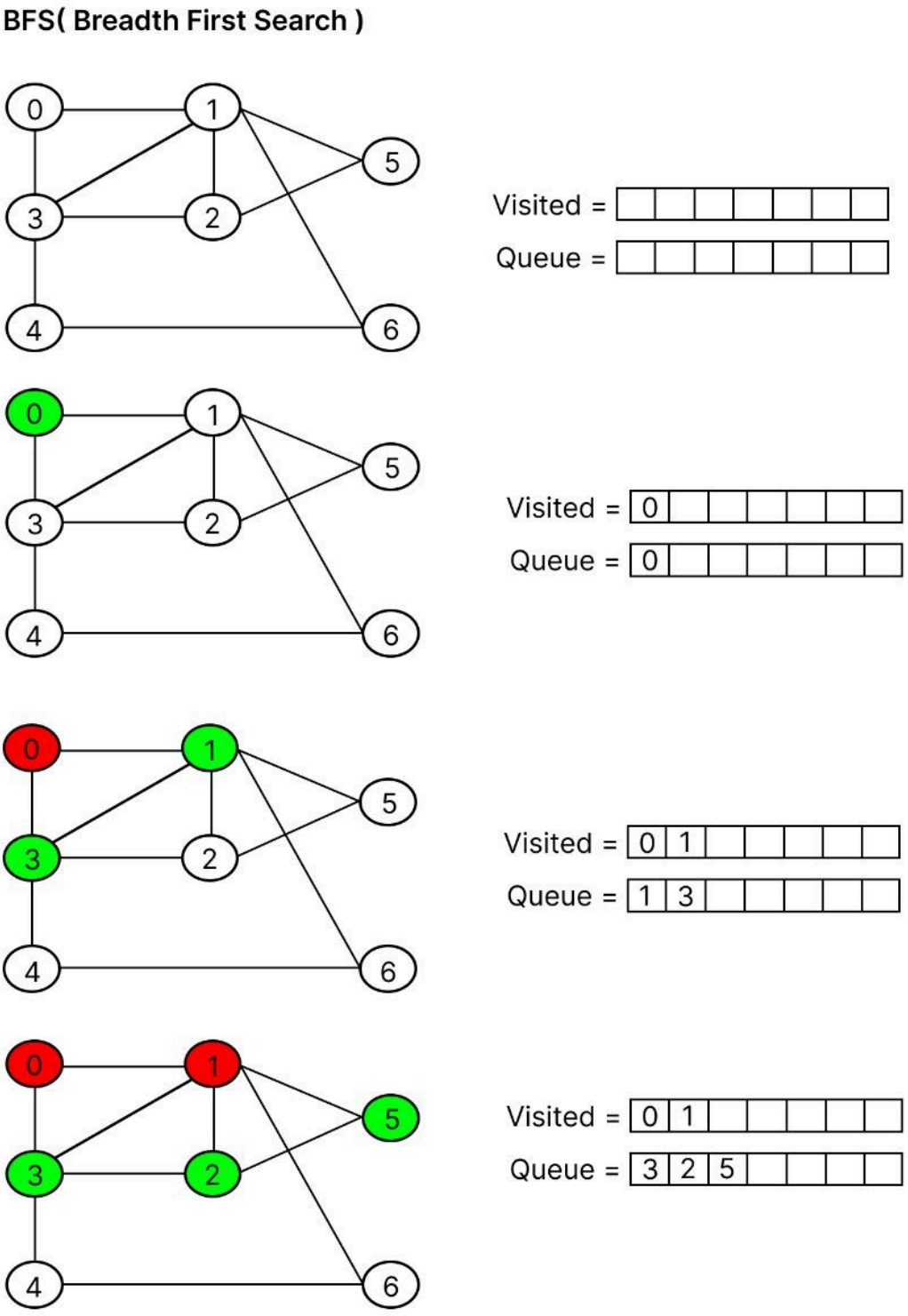
E'' = 10001 = 17

$2^{nd} generation\ f(x)$
$x \rightarrow f(x) = x^2$
$F' \rightarrow 24 \rightarrow 576$
$E'' \rightarrow 17 \rightarrow 289$
$F'' \rightarrow 25 \rightarrow 625$
$D' \rightarrow 9 \rightarrow 81$
$F''' \rightarrow 28 \rightarrow 784$
$C' \rightarrow 1 \rightarrow 1$

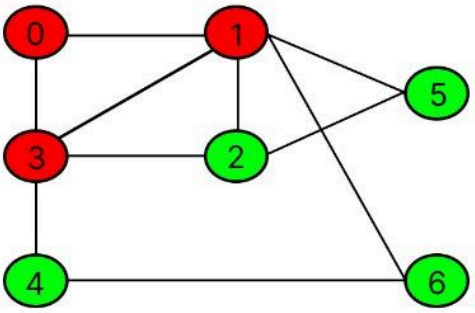
Here, x = 28, f(x) is max

BFS VS DFS Traversal

BFS Traversal

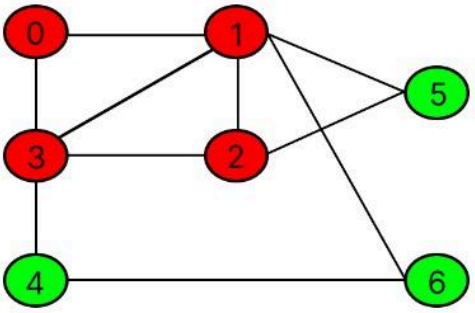


BFS(Breadth First Search)



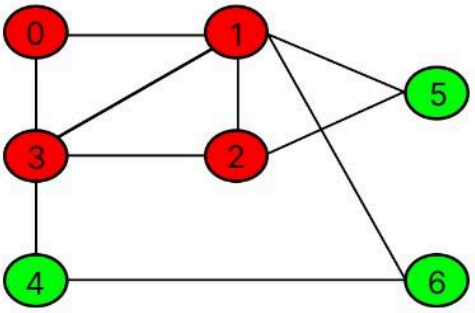
Visited = | 0 | 1 | 3 | | | | |

Queue = | 2 | 5 | 4 | | | | |



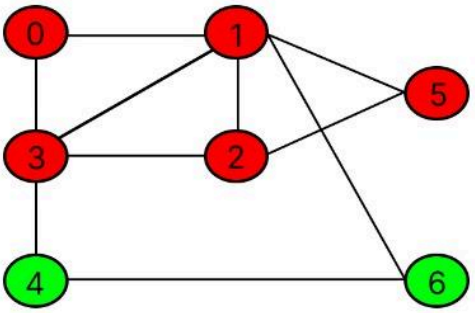
Visited = | 0 | 1 | 3 | 2 | | | |

Queue = | 5 | 4 | 6 | | | | |



Visited = | 0 | 1 | 3 | 2 | | | |

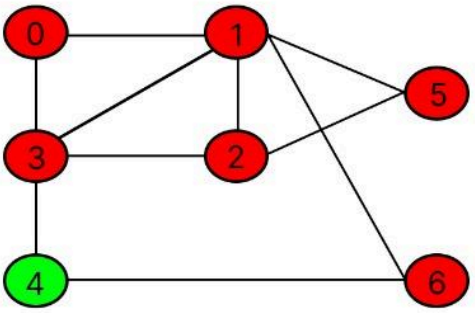
Queue = | 5 | 4 | 6 | | | | |



Visited = | 0 | 1 | 3 | 2 | 5 | | |

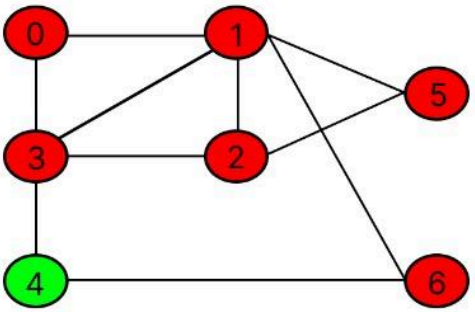
Queue = | 4 | 6 | | | | | |

BFS(Breadth First Search)



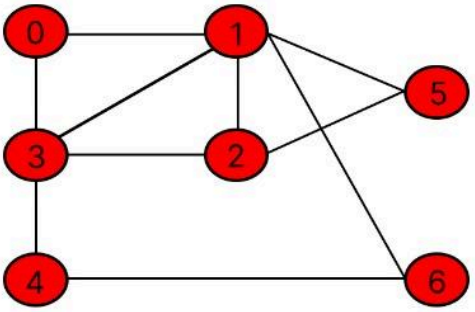
Visited = | 0 | 1 | 3 | 2 | 5 | 6 | |

Queue = | 4 | 6 | | | | | |



Visited = | 0 | 1 | 3 | 2 | 5 | 6 | |

Queue = | 4 | | | | | | |



Visited = | 0 | 1 | 3 | 2 | 5 | 6 | 4 |

Queue = | | | | | | | |

Result(print) : 0, 1, 3, 2, 5, 6, 4