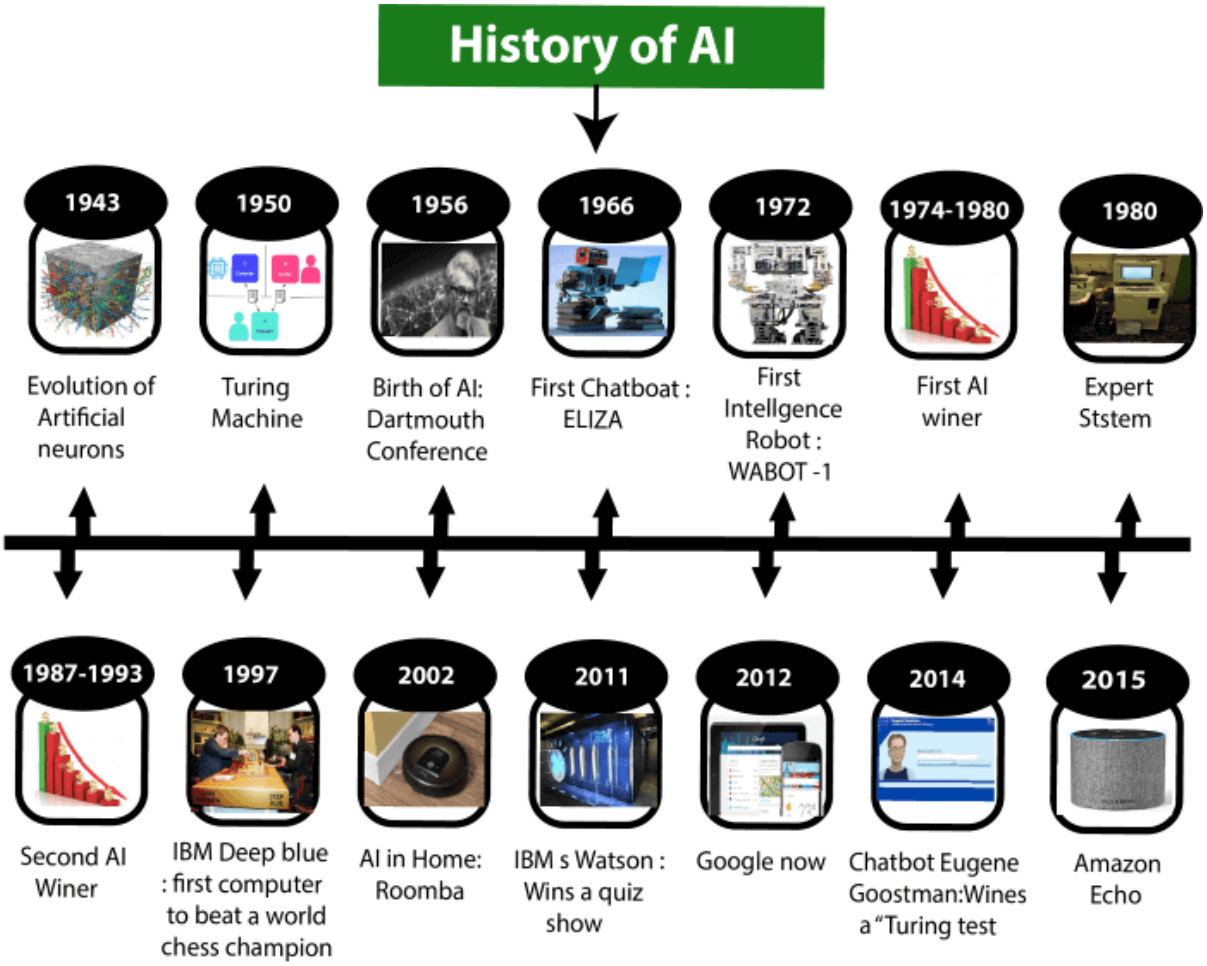
UNIT - I

History of Artificial Intelligence

Artificial Intelligence is not a new word and not a new technology for researchers. This technology is much older than you would imagine. Even there are the myths of Mechanical men in Ancient Greek and Egyptian Myths. Following are some milestones in the history of AI which defines the journey from the AI generation to till date development.



Maturation of Artificial Intelligence (1943-1952)

Year 1943: The first work which is now recognized as AI was done by Warren McCulloch and Walter pits in 1943. They proposed a model of **artificial neurons**.

Year 1949: Donald Hebb demonstrated an updating rule for modifying the connection strength between neurons. His rule is now called **Hebbian learning**.

Year 1950: The Alan Turing who was an English mathematician and pioneered Machine learning in 1950. Alan Turing publishes **"Computing Machinery and Intelligence"** in which he proposed a test. The test can check the machine's ability to exhibit intelligent behaviour equivalent to human intelligence, called a **Turing test**.

The birth of Artificial Intelligence (1952-1956)

Year 1955: An Allen Newell and Herbert A. Simon created the "first artificial intelligence program"Which was named as **"Logic Theorist"**. This program had

proved 38 of 52 Mathematics theorems, and find new and more elegant proofs for some theorems.

Year 1956: The word "Artificial Intelligence" first adopted by American Computer scientist John McCarthy at the Dartmouth Conference. For the first time, AI coined as an academic field.

At that time high-level computer languages such as FORTRAN, LISP, or COBOL were invented. And the enthusiasm for AI was very high at that time.

The golden years-Early enthusiasm (1956-1974)

Year 1966: The researchers emphasized developing algorithms which can solve mathematical problems. Joseph Weizenbaum created the first chatbot in 1966, which was named as ELIZA.

Year 1972: The first intelligent humanoid robot was built in Japan which was named as WABOT-1.

The first AI winter (1974-1980)

The duration between years 1974 to 1980 was the first AI winter duration. AI winter refers to the time period where computer scientist dealt with a severe shortage of funding from government for AI researches.

During AI winters, an interest of publicity on artificial intelligence was decreased.

A boom of AI (1980-1987)

Year 1980: After AI winter duration, AI came back with "Expert System". Expert systems were programmed that emulate the decision-making ability of a human expert.

In the Year 1980, the first national conference of the American Association of Artificial Intelligence **was held at Stanford University**.

The second AI winter (1987-1993)

The duration between the years 1987 to 1993 was the second AI Winter duration.

Again Investors and government stopped in funding for AI research as due to high cost but not efficient result. The expert system such as XCON was very cost effective.

The emergence of intelligent agents (1993-2011)

Year 1997: In the year 1997, IBM Deep Blue beats world chess champion, Gary Kasparov, and became the first computer to beat a world chess champion.

Year 2002: for the first time, AI entered the home in the form of Roomba, a vacuum cleaner.

Year 2006: AI came in the Business world till the year 2006. Companies like Facebook, Twitter, and Netflix also started using AI.

Deep learning, big data and artificial general intelligence (2011-present)

Year 2011: In the year 2011, IBM's Watson won jeopardy, a quiz show, where it had to solve the complex questions as well as riddles. Watson had proved that it could understand natural language and can solve tricky questions quickly.

Year 2012: Google has launched an Android app feature "Google now", which was able to provide information to the user as a prediction.

Year 2014: In the year 2014, Chatbot "Eugene Goostman" won a competition in the infamous "Turing test."

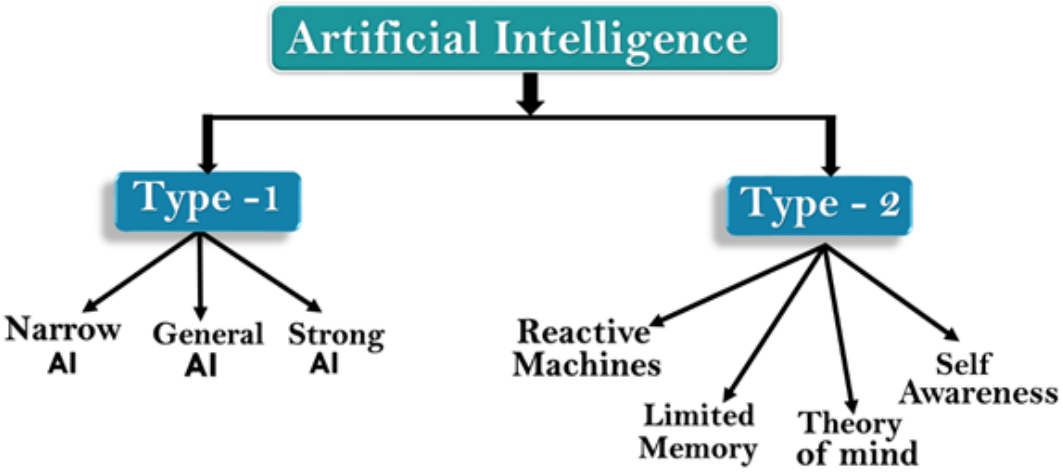
Year 2018: The "Project Debater" from IBM debated on complex topics with two master debaters and also performed extremely well.

Google has demonstrated an AI program "Duplex" which was a virtual assistant and which had taken hairdresser appointment on call, and lady on other side didn't notice that she was talking with the machine.

Now AI has developed to a remarkable level. The concept of Deep learning, big data, and data science are now trending like a boom. Nowadays companies like Google, Facebook, IBM, and Amazon are working with AI and creating amazing devices. The future of Artificial Intelligence is inspiring and will come with high intelligence.

Types of Artificial Intelligence:

Artificial Intelligence can be divided in various types, there are mainly two types of main categorization which are based on capabilities and based on functionally of AI. Following is flow diagram which explain the types of AI.



AI type-1: Based on Capabilities

1. Weak AI or Narrow AI:

Narrow AI is a type of AI which is able to perform a dedicated task with intelligence. The most common and currently available AI is Narrow AI in the world of Artificial Intelligence.

Narrow AI cannot perform beyond its field or limitations, as it is only trained for one specific task. Hence it is also termed as weak AI. Narrow AI can fail in unpredictable ways if it goes beyond its limits.

Apple Siriis a good example of Narrow AI, but it operates with a limited pre-defined range of functions.

IBM's Watson supercomputer also comes under Narrow AI, as it uses an Expert system approach combined with Machine learning and natural language processing.

Some Examples of Narrow AI are playing chess, purchasing suggestions on e-commerce site, self-driving cars, speech recognition, and image recognition.

2. General AI:

General AI is a type of intelligence which could perform any intellectual task with efficiency like a human.

The idea behind the general AI to make such a system which could be smarter and think like a human by its own.

Currently, there is no such system exist which could come under general AI and can perform any task as perfect as a human.

The worldwide researchers are now focused on developing machines with General AI.

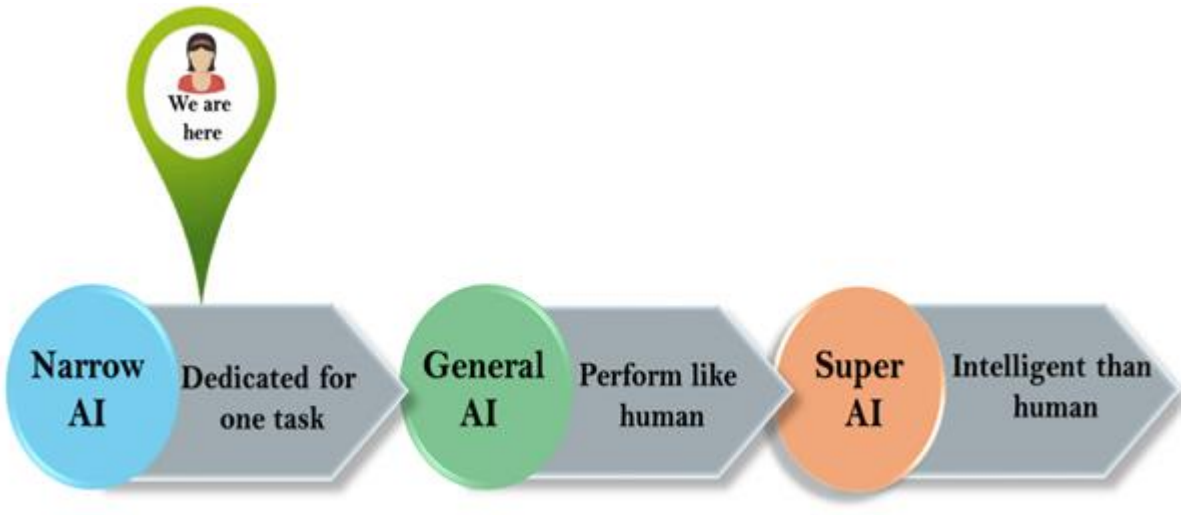
As systems with general AI are still under research, and it will take lots of efforts and time to develop such systems.

3. Super AI:

Super AI is a level of Intelligence of Systems at which machines could surpass human intelligence, and can perform any task better than human with cognitive properties. It is an outcome of general AI.

Some key characteristics of strong AI include capability include the ability to think, to reason, solve the puzzle, make judgments, plan, learn, and communicate by its own.

Super AI is still a hypothetical concept of Artificial Intelligence. Development of such systems in real is still world changing task.



Artificial Intelligence type-2: Based on functionality

1. Reactive Machines

Purely reactive machines are the most basic types of Artificial Intelligence.

Such AI systems do not store memories or past experiences for future actions.

These machines only focus on current scenarios and react on it as per possible best action.

IBM's Deep Blue system is an example of reactive machines.

Google's AlphaGo is also an example of reactive machines.

2. Limited Memory

Limited memory machines can store past experiences or some data for a short period of time.

These machines can use stored data for a limited time period only.

Self-driving cars are one of the best examples of Limited Memory systems. These cars can store recent speed of nearby cars, the distance of other cars, speed limit, and other information to navigate the road.

3. Theory of Mind

Theory of Mind AI should understand the human emotions, people, beliefs, and be able to interact socially like humans.

This type of AI machines are still not developed, but researchers are making lots of efforts and improvement for developing such AI machines.

4. Self-Awareness

Self-awareness AI is the future of Artificial Intelligence. These machines will be super intelligent, and will have their own consciousness, sentiments, and self-awareness.

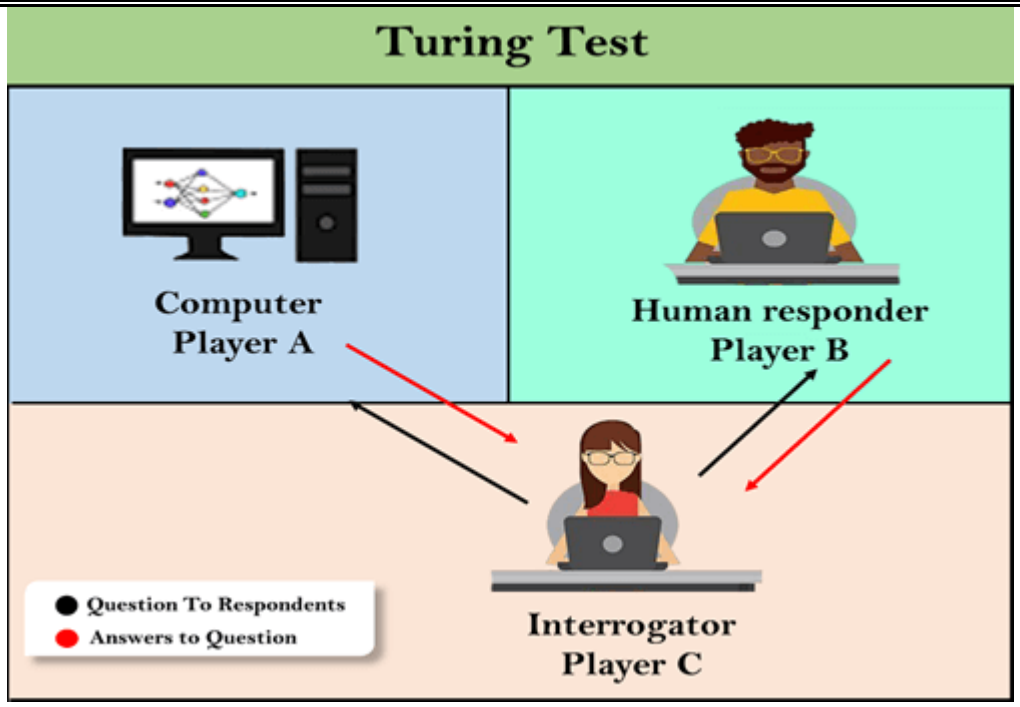
These machines will be smarter than human mind.

Self-Awareness AI does not exist in reality still and it is a hypothetical concept.

Turing Test in AI

In 1950, Alan Turing introduced a test to check whether a machine can think like a human or not, this test is known as the Turing Test. In this test, Turing proposed that the computer can be said to be an intelligent if it can mimic human response under specific conditions.

Turing Test was introduced by Turing in his 1950 paper, "Computing Machinery and Intelligence," which considered the question, "Can Machine think?"



The Turing test is based on a party game "Imitation game," with some modifications. This game involves three players in which one player is Computer, another player is human responder, and the third player is a human Interrogator, who is isolated from other two players and his job is to find that which player is machine among two of them.

Consider, Player A is a computer, Player B is human, and Player C is an interrogator. Interrogator is aware that one of them is machine, but he needs to identify this on the basis of questions and their responses.

The conversation between all players is via keyboard and screen so the result would not depend on the machine's ability to convert words as speech.

The test result does not depend on each correct answer, but only how closely its responses like a human answer. The computer is permitted to do everything possible to force a wrong identification by the interrogator.

The questions and answers can be like:

Interrogator: Are you a computer?

PlayerA (Computer): No

Interrogator: Multiply two large numbers such as (256896489*456725896)

Player A: Long pause and give the wrong answer.

In this game, if an interrogator would not be able to identify which is a machine and which is human, then the computer passes the test successfully, and the machine is said to be intelligent and can think like a human.

"In 1991, the New York businessman Hugh Loebner announces the prize competition, offering a $100,000 prize for the first computer to pass the Turing test. However, no AI program to till date, come close to passing an undiluted Turing test".

Chatbots to attempt the Turing test:

ELIZA: ELIZA was a Natural language processing computer program created by Joseph Weizenbaum. It was created to demonstrate the ability of communication between machine and humans. It was one of the first chatterbots, which has attempted the Turing Test.

Parry: Parry was a chatterbot created by Kenneth Colby in 1972. Parry was designed to simulate a person with **Paranoid schizophrenia**(most common chronic mental disorder). Parry was described as "ELIZA with attitude." Parry was tested using a variation of the Turing Test in the early 1970s.

Eugene Goostman: Eugene Goostman was a chatbot developed in Saint Petersburg in 2001. This bot has competed in the various number of Turing Test. In June 2012, at an event, Goostman won the competition promoted as largest-ever Turing test content, in which it has convinced 29% of judges that it was a human.Goostman resembled as a 13-year old virtual boy.

The Chinese Room Argument:

There were many philosophers who really disagreed with the complete concept of Artificial Intelligence. The most famous argument in this list was "**Chinese Room**."

In the year **1980, John Searle** presented "**Chinese Room**" thought experiment, in his paper "**Mind, Brains, and Program**," which was against the validity of Turing's Test. According to his argument, "**Programming a computer may make it to understand a language, but it will not produce a real understanding of language or consciousness in a computer**."

He argued that Machine such as ELIZA and Parry could easily pass the Turing test by manipulating keywords and symbol, but they had no real understanding of language. So it cannot be described as "thinking" capability of a machine such as a human.

Features required for a machine to pass the Turing test:

Natural language processing: NLP is required to communicate with Interrogator in general human language like English.

Knowledge representation: To store and retrieve information during the test.

Automated reasoning: To use the previously stored information for answering the questions.

Machine learning: To adapt new changes and can detect generalized patterns.

Vision (For total Turing test): To recognize the interrogator actions and other objects during a test.

Motor Control (For total Turing test): To act upon objects if requested.

Uninformed Search Algorithms

Uninformed search is a class of general-purpose search algorithms which operates in brute force-way. Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search.

Following are the various types of uninformed search algorithms:

Breadth-first Search

Depth-first Search

Depth-limited Search

Iterative deepening depth-first search

Uniform cost search

Bidirectional Search

1. Breadth-first Search:

Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.

BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.

The breadth-first search algorithm is an example of a general-graph search algorithm.

Breadth-first search implemented using FIFO queue data structure.

Advantages:

BFS will provide a solution if any solution exists.

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

Disadvantages:

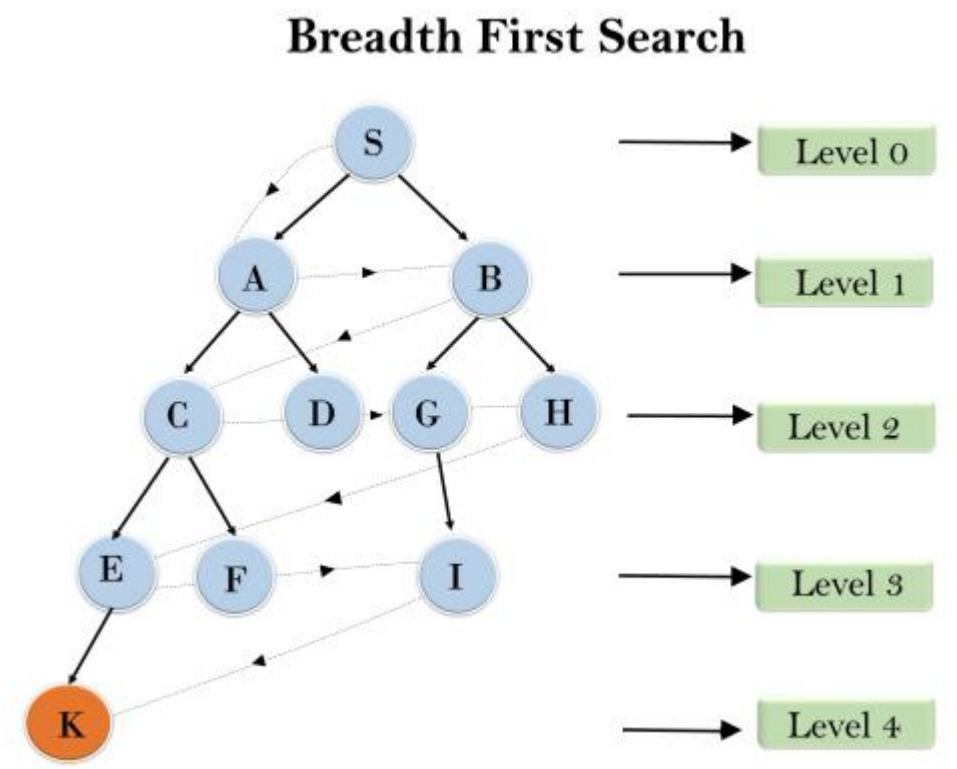
It requires lots of memory since each level of the tree must be saved into memory to expand the next level.

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:

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 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.

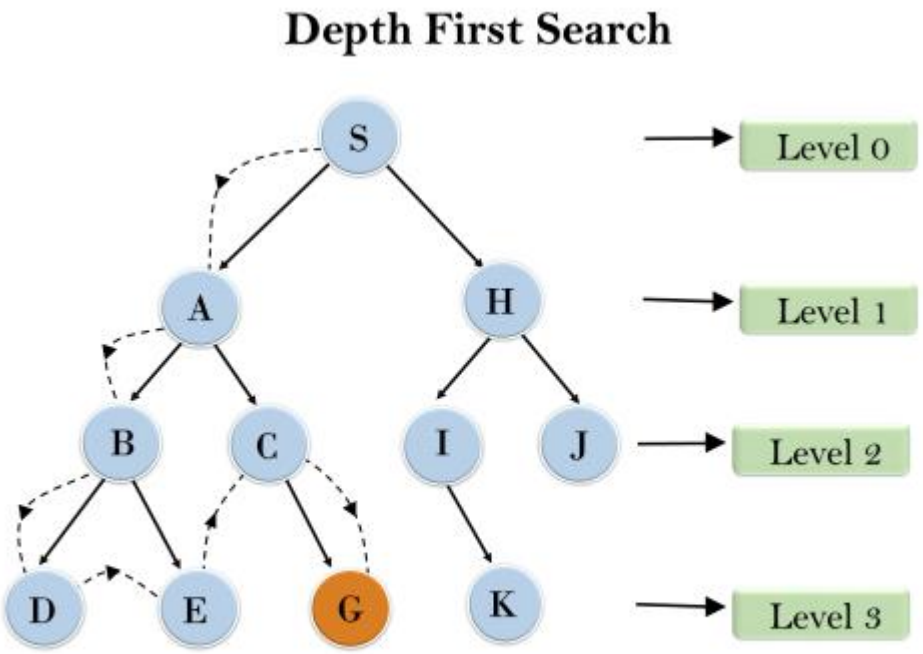
DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

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.

3. Depth-Limited Search Algorithm:

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

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

Standard failure value: It indicates that problem does not have any solution.

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

Advantages:

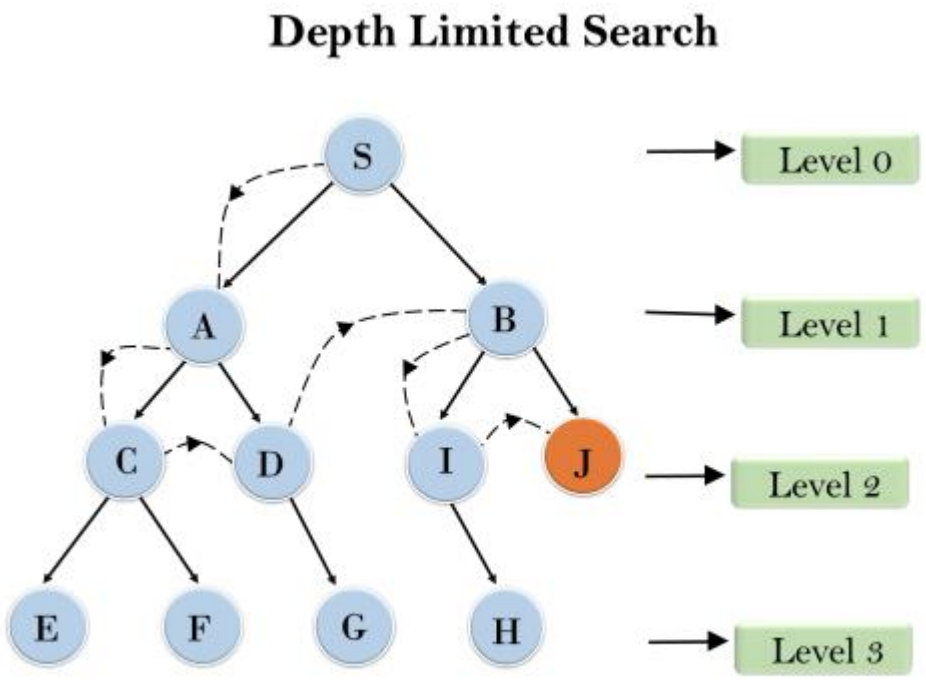
Depth-limited search is Memory efficient.

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**($b \times \ell$)**.

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

4. 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:

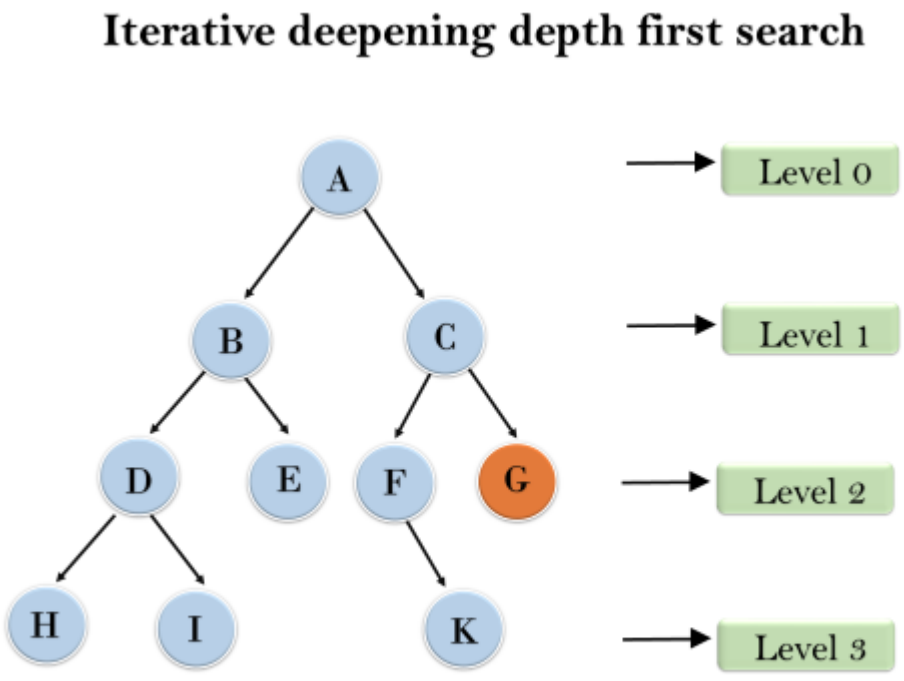
Itcombines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

Disadvantages:

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

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

Completeness:

This algorithm is complete is ifthe branching factor is finite.

Time Complexity:

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

Space Complexity:

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

Optimal:

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

6. Bidirectional Search Algorithm:

Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node. Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.

Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Advantages:

Bidirectional search is fast.

Bidirectional search requires less memory

Disadvantages:

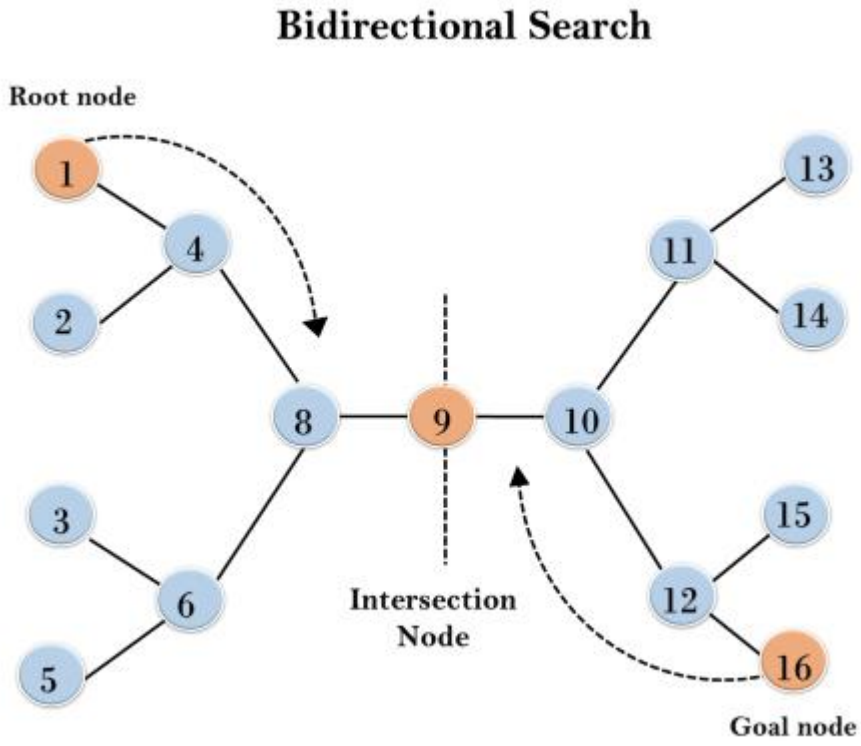
Implementation of the bidirectional search tree is difficult.

In bidirectional search, one should know the goal state in advance.

Example:

In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.

The algorithm terminates at node 9 where two searches meet.



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

Time Complexity: Time complexity of bidirectional search using BFS is **$O(b^d)$**.

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

Optimal: Bidirectional search is Optimal.

Searching is the universal technique of problem solving in AI. There are some single-player games such as tile games, Sudoku, crossword, etc. The search algorithms help you to search for a particular position in such games.

Single Agent Pathfinding Problems

The games such as 3X3 eight-tile, 4X4 fifteen-tile, and 5X5 twenty four tile puzzles are single-agent-path-finding challenges. They consist of a matrix of tiles with a blank tile. The player is required to arrange the tiles by sliding a tile either vertically or horizontally into a blank space with the aim of accomplishing some objective.

The other examples of single agent pathfinding problems are Travelling Salesman Problem, Rubik's Cube, and Theorem Proving.

Search Terminology

Problem Space – It is the environment in which the search takes place. (A set of states and set of operators to change those states)

Problem Instance – It is Initial state + Goal state.

Problem Space Graph – It represents problem state. States are shown by nodes and operators are shown by edges.

Depth of a problem – Length of a shortest path or shortest sequence of operators from Initial State to goal state.

Space Complexity – The maximum number of nodes that are stored in memory.

Time Complexity – The maximum number of nodes that are created.

Admissibility – A property of an algorithm to always find an optimal solution.

Branching Factor – The average number of child nodes in the problem space graph.

Depth – Length of the shortest path from initial state to goal state.

Brute-Force Search Strategies

They are most simple, as they do not need any domain-specific knowledge. They work fine with small number of possible states.

Requirements –

State description

A set of valid operators

Initial state

Goal state description

Breadth-First Search

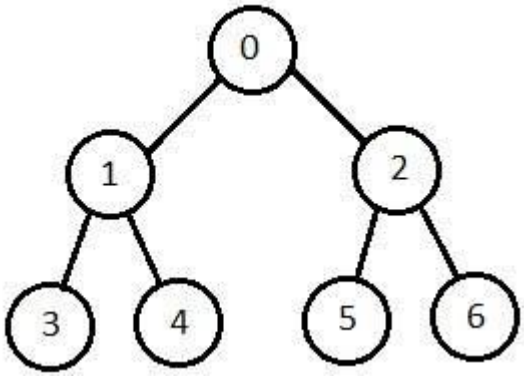
It starts from the root node, explores the neighboring nodes first and moves towards the next level neighbors. It generates one tree at a time until the solution is found. It can be implemented using FIFO queue data structure. This method provides shortest path to the solution.

If **branching factor** (average number of child nodes for a given node) = b and depth = d, then number of nodes at level d = b^d.

The total no of nodes created in worst case is $b + b^2 + b^3 + \ldots + b^d$.

Disadvantage – Since each level of nodes is saved for creating next one, it consumes a lot of memory space. Space requirement to store nodes is exponential.

Its complexity depends on the number of nodes. It can check duplicate nodes.



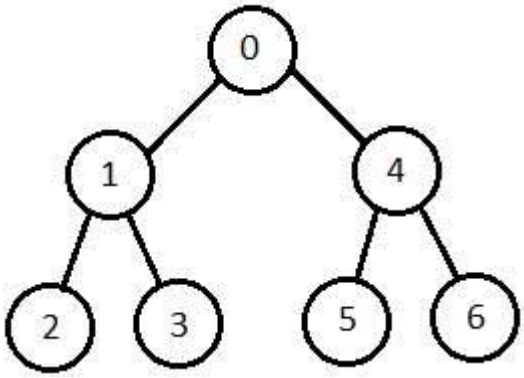
Depth-First Search

It is implemented in recursion with LIFO stack data structure. It creates the same set of nodes as Breadth-First method, only in the different order.

As the nodes on the single path are stored in each iteration from root to leaf node, the space requirement to store nodes is linear. With branching factor *b* and depth as *m*, the storage space is *bm*.

Disadvantage – This algorithm may not terminate and go on infinitely on one path. The solution to this issue is to choose a cut-off depth. If the ideal cut-off is *d*, and if chosen cut-off is lesser than *d*, then this algorithm may fail. If chosen cut-off is more than *d*, then execution time increases.

Its complexity depends on the number of paths. It cannot check duplicate nodes.



Bidirectional Search

It searches forward from initial state and backward from goal state till both meet to identify a common state.

The path from initial state is concatenated with the inverse path from the goal state. Each search is done only up to half of the total path.

Uniform Cost Search

Sorting is done in increasing cost of the path to a node. It always expands the least cost node. It is identical to Breadth First search if each transition has the same cost.

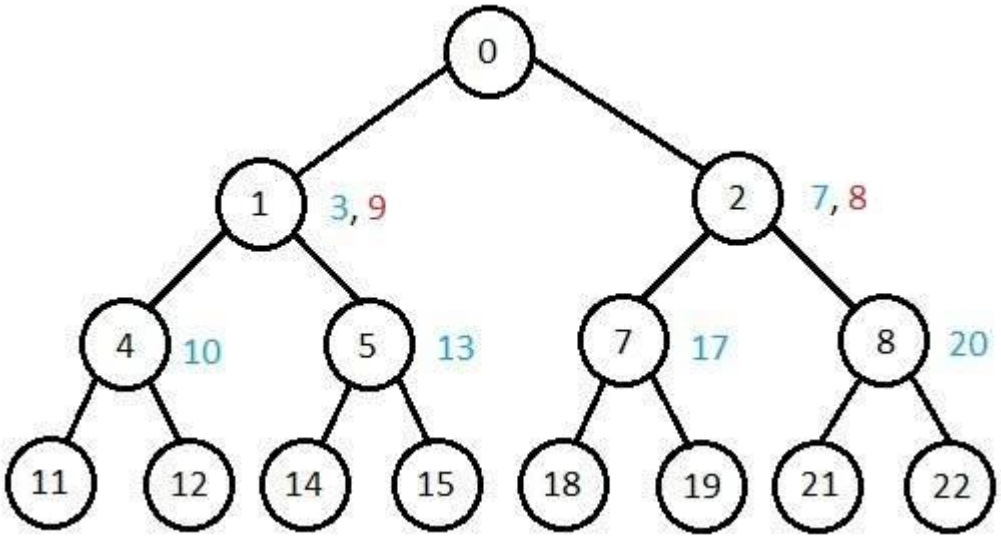
It explores paths in the increasing order of cost.

Disadvantage – There can be multiple long paths with the cost ≤ C*. Uniform Cost search must explore them all.

Iterative Deepening Depth-First Search

It performs depth-first search to level 1, starts over, executes a complete depth-first search to level 2, and continues in such way till the solution is found.

It never creates a node until all lower nodes are generated. It only saves a stack of nodes. The algorithm ends when it finds a solution at depth *d*. The number of nodes created at depth *d* is b^d and at depth *d-1* is $b^{d-1.}$



Comparison of Various Algorithms Complexities

Let us see the performance of algorithms based on various criteria –

Criterion	Breadth First	Depth First	Bidirectional	Uniform Cost	Interactive Deepening
Time	b^d	b^m	$b^{d/2}$	b^d	b^d
Space	b^d	b^m	$b^{d/2}$	b^d	b^d
Optimality	Yes	No	Yes	Yes	Yes
Completeness	Yes	No	Yes	Yes	Yes

Informed (Heuristic) Search Strategies

To solve large problems with large number of possible states, problem-specific knowledge needs to be added to increase the efficiency of search algorithms.

Heuristic Evaluation Functions

They calculate the cost of optimal path between two states. A heuristic function for sliding-tiles games is computed by counting number of moves that each tile makes from its goal state and adding these number of moves for all tiles.

Pure Heuristic Search

It expands nodes in the order of their heuristic values. It creates two lists, a closed list for the already expanded nodes and an open list for the created but unexpanded nodes.

In each iteration, a node with a minimum heuristic value is expanded, all its child nodes are created and placed in the closed list. Then, the heuristic function is applied to the child nodes and they are placed in the open list according to their heuristic value. The shorter paths are saved and the longer ones are disposed.

A * Search

It is best-known form of Best First search. It avoids expanding paths that are already expensive, but expands most promising paths first.

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

g(n) the cost (so far) to reach the node

h(n) estimated cost to get from the node to the goal

f(n) estimated total cost of path through n to goal. It is implemented using priority queue by increasing f(n).

Greedy Best First Search

It expands the node that is estimated to be closest to goal. It expands nodes based on f(n) = h(n). It is implemented using priority queue.

Disadvantage – It can get stuck in loops. It is not optimal.

Local Search Algorithms

They start from a prospective solution and then move to a neighboring solution. They can return a valid solution even if it is interrupted at any time before they end.

Hill-Climbing Search

It is an iterative algorithm that starts with an arbitrary solution to a problem and attempts to find a better solution by changing a single element of the solution incrementally. If the change produces a better solution, an incremental change is taken as a new solution. This process is repeated until there are no further improvements.

function Hill-Climbing (problem), returns a state that is a local maximum.

inputs: problem, a problem

local variables: current, a node

neighbor, a node

current <-Make_Node(Initial-State[problem])

loop

do neighbor <- a highest_valued successor of *current*

if Value[neighbor] ≤ Value[current] then

return State[current]

current <- neighbor

end

Disadvantage – This algorithm is neither complete, nor optimal.

Local Beam Search

In this algorithm, it holds k number of states at any given time. At the start, these states are generated randomly. The successors of these k states are computed with the help of objective function. If any of these successors is the maximum value of the objective function, then the algorithm stops.

Otherwise the (initial k states and k number of successors of the states = 2k) states are placed in a pool. The pool is then sorted numerically. The highest k states are selected as new initial states. This process continues until a maximum value is reached.

function BeamSearch(*problem, k*), returns a solution state.

start with k randomly generated states

loop

  generate all successors of all k states

  if any of the states = solution, then return the state

  else select the k best successors

end

Simulated Annealing

Annealing is the process of heating and cooling a metal to change its internal structure for modifying its physical properties. When the metal cools, its new structure is seized, and the metal retains its newly obtained properties. In simulated annealing process, the temperature is kept variable.

We initially set the temperature high and then allow it to 'cool' slowly as the algorithm proceeds. When the temperature is high, the algorithm is allowed to accept worse solutions with high frequency.

Start

Initialize k = 0; L = integer number of variables;

From i → j, search the performance difference Δ.

If Δ <= 0 then accept else if exp(-Δ/T(k)) > random(0,1) then accept;

Repeat steps 1 and 2 for L(k) steps.

k = k + 1;

Repeat steps 1 through 4 till the criteria is met.

End

Travelling Salesman Problem

In this algorithm, the objective is to find a low-cost tour that starts from a city, visits all cities en-route exactly once and ends at the same starting city.

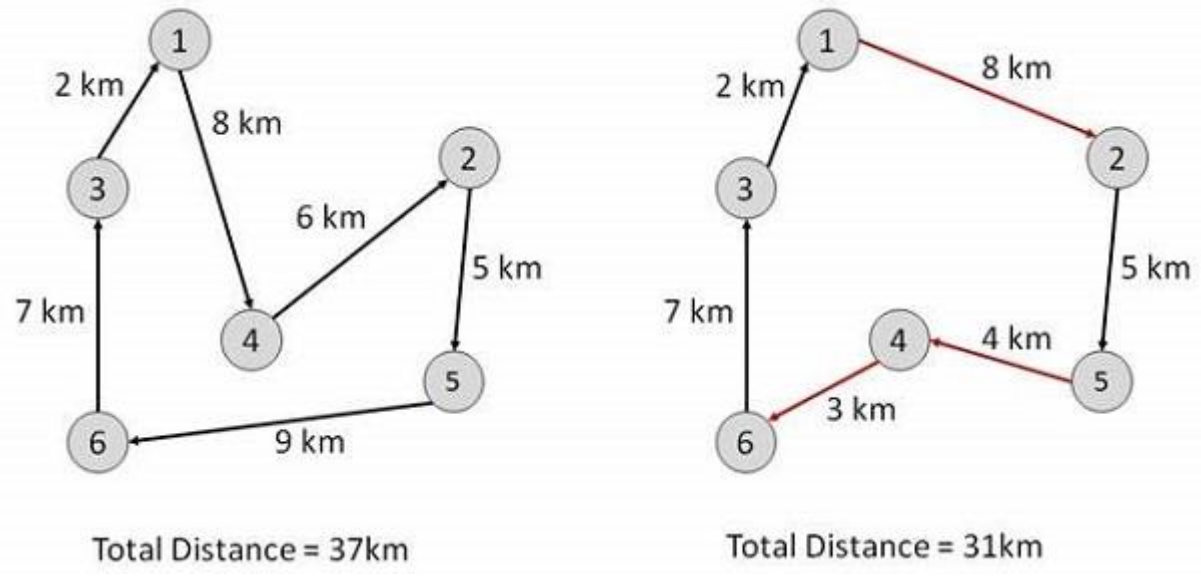
Start

  Find out all (n -1)! Possible solutions, where n is the total number of cities.

  Determine the minimum cost by finding out the cost of each of these (n -1)! solutions.

Finally, keep the one with the minimum cost.

end



Iterative Deepening Search (IDS) or Iterative Deepening Depth First Search (IDDFS)

An integral component of computer science and artificial intelligence are search algorithms. They are used to solve a variety of issues, from playing games like chess and checkers to locating the shortest route on a map. The Depth First Search (DFS) method, one of the most popular search algorithms, searches a network or tree by travelling as far as possible along each branch before turning around. However, DFS has a critical drawback: if the graph contains cycles, it could become trapped in an endless loop. Utilizing Iterative Deepening Search (IDS) or Iterative Deepening Depth First Search is one technique to solve this issue (IDDFS).

What is IDS?

A search algorithm known as IDS combines the benefits of DFS with Breadth First Search (BFS). The graph is explored using DFS, but the depth limit steadily increased until the target is located. In other words, IDS continually runs DFS, raising the depth limit each time, until the desired result is obtained. Iterative deepening is a method that makes sure the search is thorough (i.e., it discovers a solution if one exists) and efficient (i.e., it finds the shortest path to the goal).

The pseudocode for IDS is straightforward:

Code

```
function iterativeDeepeningSearch(root, goal):
    depth = 0
    while True:
        result = depthLimitedSearch(root, goal, depth)
        if result == FOUND:
            return goal
        if result == NOT_FOUND:
```

```
            return None
        depthdepth = depth + 1

function depthLimitedSearch(node, goal, depth):
    if node == goal:
        return FOUND
    if depth == 0:
        return NOT_FOUND
    for child in node.children:
        result = depthLimitedSearch(child, goal, depth - 1)
        if result == FOUND:
            return FOUND
    return NOT_FOUND
```

How does IDS work?

The iterativeDeepeningSearch function performs iterative deepening search on the graph using a root node and a goal node as inputs until the goal is attained or the search space is used up. This is accomplished by regularly using the depthLimitedSearch function, which applies a depth restriction to DFS. The search ends and returns the goal node if the goal is located at any depth. The search yields None if the search space is used up (all nodes up to the depth limit have been investigated).

The depthLimitedSearch function conducts DFS on the graph with the specified depth limit by taking as inputs a node, a destination node, and a depth limit. The search returns FOUND if the desired node is located at the current depth. The search returns NOT FOUND if the depth limit is reached but the goal node cannot be located. If neither criterion is true, the search iteratively moves on to the node's offspring.

Program:

Code

```
from collections import defaultdict


class Graph:
    def __init__(self):
        self.graph = defaultdict(list)


    def add_edge(self, u, v):
        self.graph[u].append(v)
```

```
    def iddfs(self, start, goal, max_depth):
        for depth in range(max_depth+1):
            visited = set()
            if self.dls(start, goal, depth, visited):
                return True
        return False

    def dls(self, node, goal, depth, visited):
        if node == goal:
            return True
        if depth == 0:
            return False
        visited.add(node)
        for neighbor in self.graph[node]:
            if neighbor not in visited:
                if self.dls(neighbor, goal, depth-1, visited):
                    return True
        return False

# Example usage
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.add_edge(2, 3)
g.add_edge(3, 3)

start = 0
goal = 3
max_depth = 3
if g.iddfs(start, goal, max_depth):
    print("Path found")
```

```
else:
    print("Path not found")
```

Output

```
Path found
```

Advantages

IDS is superior to other search algorithms in a number of ways. The first benefit is that it is comprehensive, which ensures that a solution will be found if one is there in the search space. This is so that all nodes under a specific depth limit are investigated before the depth limit is raised by IDS, which does a depth-limited DFS.

IDS is memory-efficient, which is its second benefit. This is because IDS decreases the algorithm's memory needs by not storing every node in the search area in memory. IDS minimises the algorithm's memory footprint by only storing the nodes up to the current depth limit.

IDS's ability to be utilised for both tree and graph search is its third benefit. This is due to the fact that IDS is a generic search algorithm that works on any search space, including a tree or a graph.

Disadvantages

IDS has the disadvantage of potentially visiting certain nodes more than once, which might slow down the search. The benefits of completeness and optimality frequently exceed this disadvantage. In addition, by employing strategies like memory or caching, the repeated trips can be minimised.

UNIT – II

Informed Search Algorithms

So far we have talked about the uninformed search algorithms which looked through search space for all possible solutions of the problem without having any additional knowledge about search space. But 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.

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

Admissibility of the heuristic function is given as:

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

Here h(n) is heuristic cost, and h*(n) is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

Pure Heuristic Search:

Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.

On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues unit a goal state is found.

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

Best First Search Algorithm(Greedy search)

A* Search Algorithm

1.) Best-first Search Algorithm (Greedy Search):

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

f(n)= g(n).

Were, h(n)= estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the priority queue.

Best first search algorithm:

Step 1: Place the starting node into the OPEN list.

Step 2: If the OPEN list is empty, Stop and return failure.

Step 3: Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.

Step 4: Expand the node n, and generate the successors of node n.

Step 5: Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.

Step 6: For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.

Step 7: Return to Step 2.

Advantages:

Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.

This algorithm is more efficient than BFS and DFS algorithms.

Disadvantages:

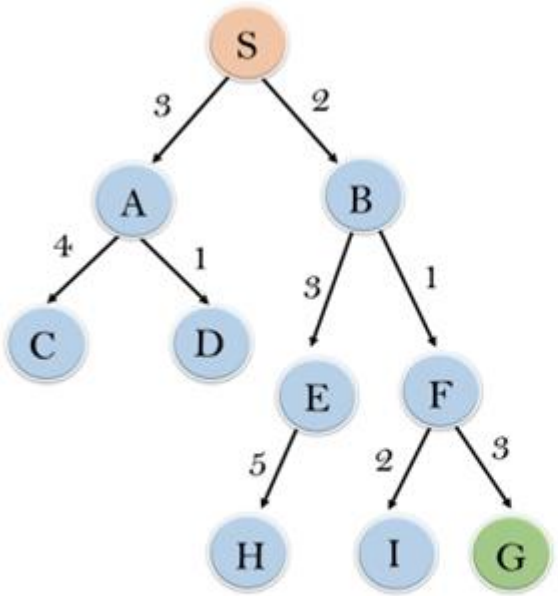
It can behave as an unguided depth-first search in the worst case scenario.

It can get stuck in a loop as DFS.

This algorithm is not optimal.

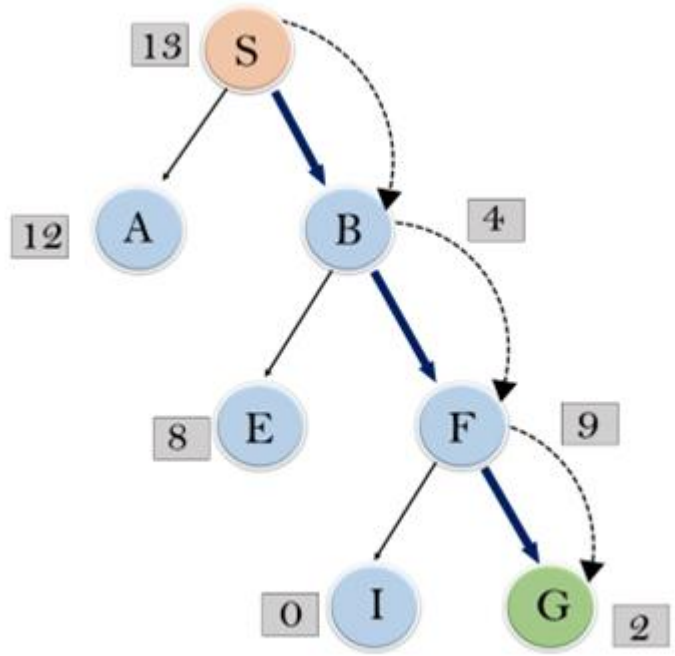
Example:

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



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

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



Expand the nodes of S and put in the CLOSED list

Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B]
: Open [E, A], Closed [S, B, F]

Iteration 3: Open [I, G, E, A], Closed [S, B, F]
: Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: **S----> B----->F----> G**

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

Space Complexity: The worst case space complexity of Greedy best first search is $O(b^m)$. Where, m is the maximum depth of the search space.

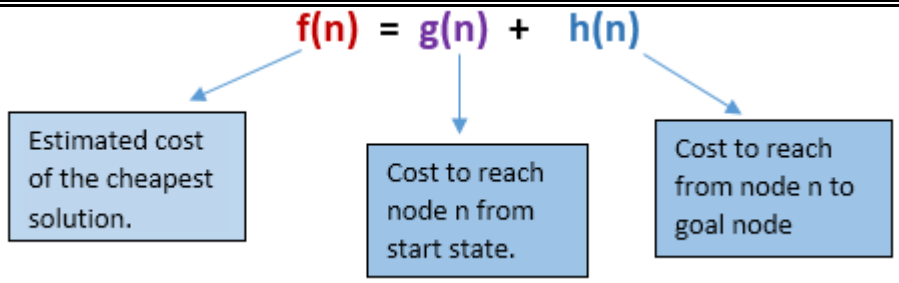
Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

Optimal: Greedy best first search algorithm is not optimal.

2.) A* Search Algorithm:

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). 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**.



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

Algorithm of A* search:

Step1: Place the starting node in the OPEN list.

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

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

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

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

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

Advantages:

A* search algorithm is the best algorithm than other search algorithms.

A* search algorithm is optimal and complete.

This algorithm can solve very complex problems.

Disadvantages:

It does not always produce the shortest path as it mostly based on heuristics and approximation.

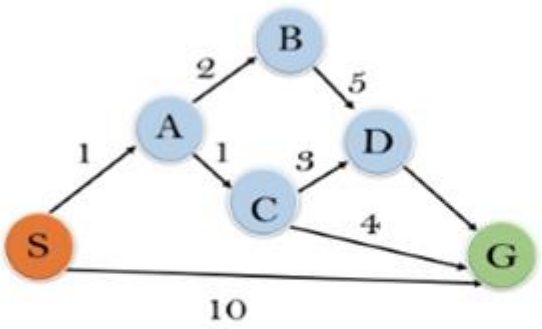
A* search algorithm has some complexity issues.

The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

Example:

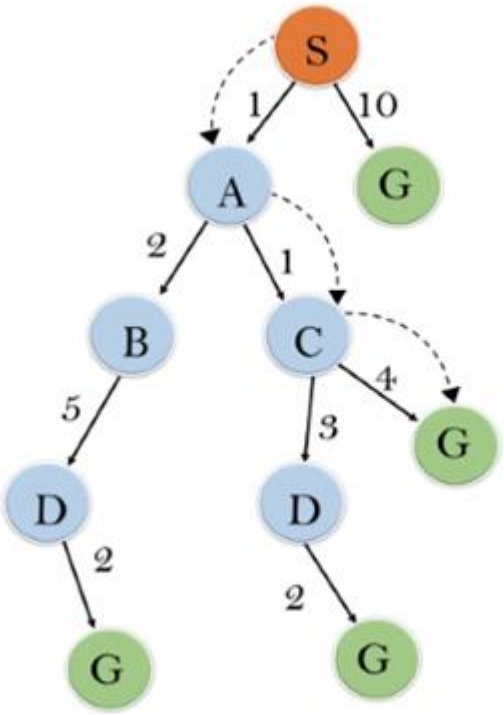
In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.

Here we will use OPEN and CLOSED list.



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

Solution:



Initialization: {(S, 5)}

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

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

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

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

Points to remember:

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

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

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

Complete: A* algorithm is complete as long as:

Branching factor is finite.

Cost at every action is fixed.

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

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

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

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)**

Iterative Deepening A* algorithm (IDA*) – Artificial intelligence

Iterative deepening A* **(IDA*)** is a graph traversal and path-finding method that can determine the shortest route in a weighted graph between a defined start node and any one of a group of goal nodes. It is a kind of **iterative deepening depth-first search** that adopts the A* search algorithm's idea of using a heuristic function to assess the remaining cost to reach the goal.
A memory-limited version of A* is called **IDA***. It performs all operations that A* does and has optimal features for locating the shortest path, but it occupies less memory.
Iterative Deepening A Star uses a **heuristic** to choose which nodes to explore and at which depth to stop, as opposed to Iterative Deepening DFS, which utilizes simple depth to determine when to end the current iteration and continue with a higher depth.

Key points

The graph traversal algorithm.

Find the shortest path in a weighted graph between the start and goal nodes using the path search algorithm.

An **alternative** to the iterative depth-first search algorithm.

Uses a heuristic function, which is a technique taken from A* algorithm.

It's a Depth-First Search algorithm, So it used less memory than the A* algorithm.

IDA* is an admissible heuristic because it never **overestimates** the cost of reaching the goal.

It's focused on searching the most promising nodes, so, it doesn't go to the same depth everywhere.

The evaluation function in **IDA*** looks like this:

Where **h** is admissible.
here,

f(n) = Total cost evaluation function.
g(n) = The actual cost from the initial node to the current node.
h(n) = Heuristic estimated cost from the current node to the goal state. it is based on the approximation according to the problem characteristics.

What is the f score?

In the IDA* algorithm, F-score is a **heuristic function** that is used to estimate the cost of reaching the goal state from a given state. It is a combination of two other heuristic functions, **g(n)** and **h(n).**
It is used to determine the order in which the algorithm expands nodes in the search tree and thus, it plays an important role in how quickly the algorithm finds a solution. A lower F-score indicates that a node is closer to the goal state and will be expanded before a node with a higher F-score.
Simply it is nothing but **g(n) + h(n).**

How IDA* algorithm work?

Step 1: Initialization

Set the root node as the current node, and find the f-score.

Sep 2: Set threshold

Set the cost limit as a **threshold** for a **node** i.e the **maximum f-score** allowed for that node for further explorations.

Step 3: Node Expansion

Expand the current node to its children and find f-scores.

Step 4: Pruning

If for any **node** the **f-score > threshold**, prune that node because it's considered too expensive for that node. and store it in the **visited node list.**

Step 5: Return Path

If the **Goal node** is found then return the **path** from the start node Goal node.

Step 6: Update the Threshold

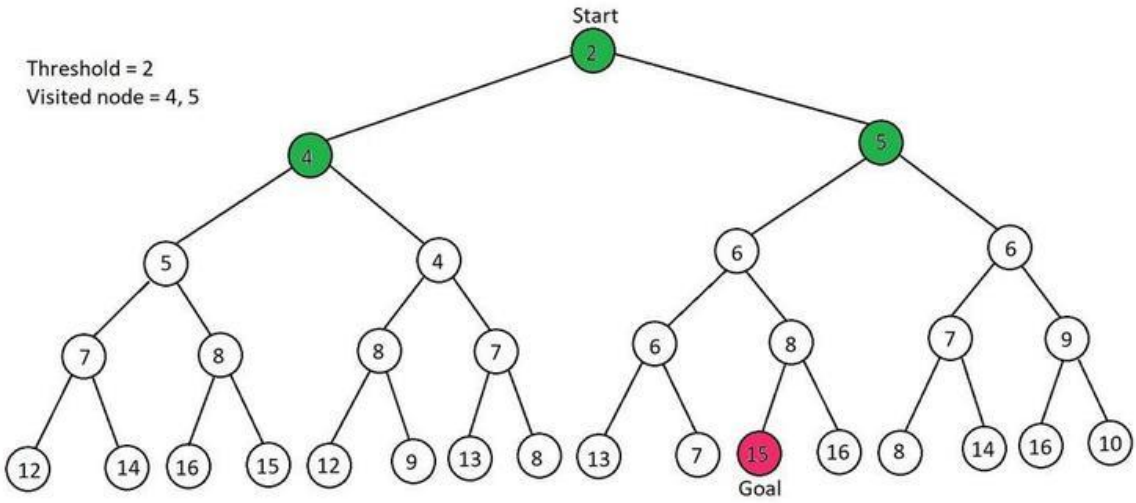
If the Goal node is not found then **repeat from step 2** by changing the threshold with the minimum pruned value from the **visited node list**. And Continue it until you reach the goal node.

Example

In the below tree, the **f score** is written inside the nodes means the f score is already computed and the start node is **2** whereas the goal node is **15**. the explored node is colored green color.

so now we have to go to a given goal by using IDA* algorithm.

Iteration 1



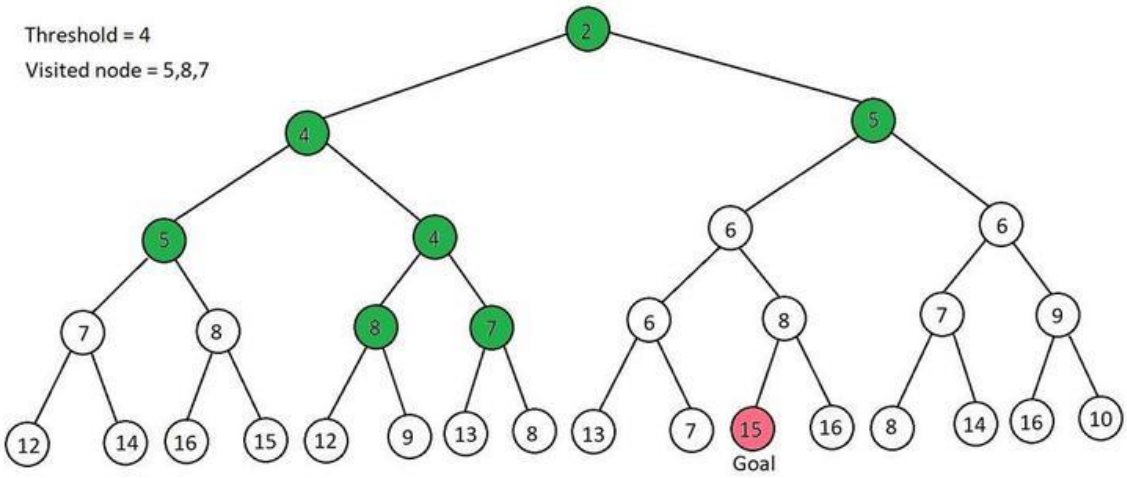
Iteration 1

Root node as current node i.e 2

Threshold = current node value (2=2). So explore its children.

4 > Threshold & 5>Threshold. So, this iteration is over and the pruned values are 4, and 5.

Iteration 2



Iteration 2

In pruned values, the least is 4, So threshold = 4

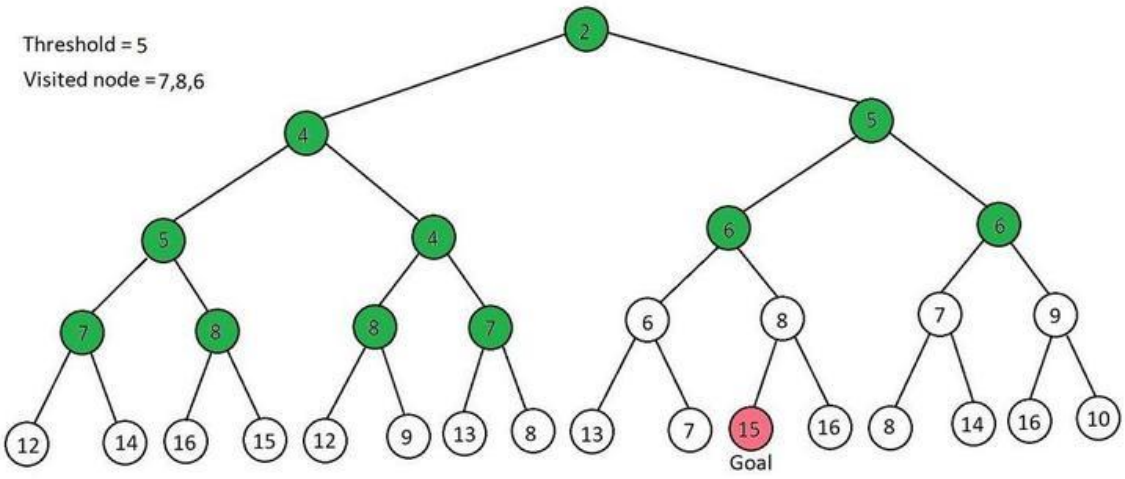
current node = 2 and 2< threshold, So explore its children. i.e two children explore one by one

So, first children 4, So, set current node = 4 i.e equal to the threshold, so, explored its children also i.e 5, 4 having 5> threshold so, pruned it and explore second child of node 4 i.e 4, so set current node = 4 = threshold, and explore its children i.e 8 & 7 having both 8 & 7 > threshold so, pruned it. At the end of this, our pruned value is 5,8,7

Similarly, Explore the second child of root node 2 i.e 5 as the current node, i.e 5>threshold, So pruned it.

So, our pruned value is 5,8,7.

Iteration 3



Iteration 3

In pruned values, the least is 5, So threshold = 5

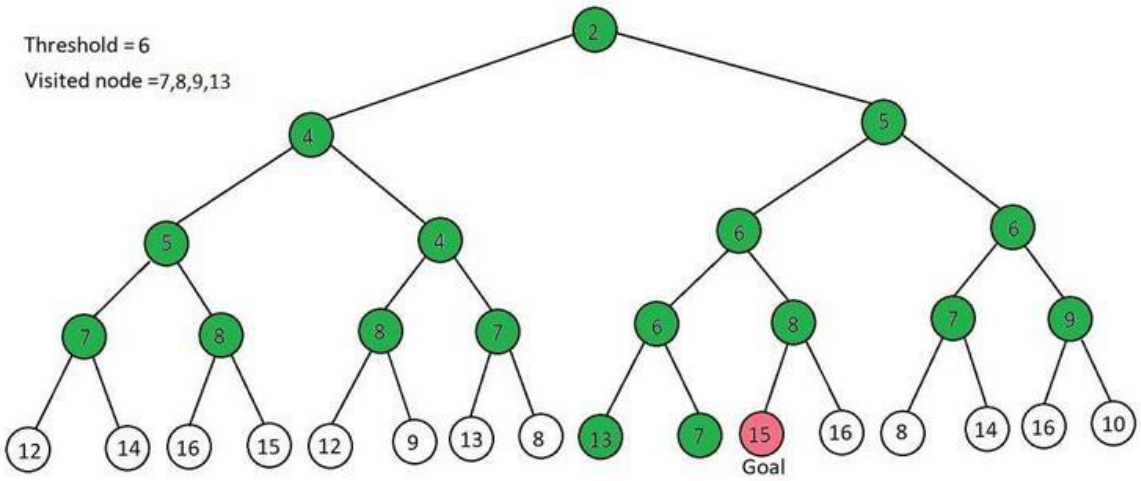
current node = root node = 2 and 2< threshold, So explore its children. i.e two children explore one by one

So, first children 4, So, set current node = 4 < threshold, so, explored its children also i.e 5, 4 having 5= threshold so explore its child also 7&8 > threshold. So, pruned it and explore the second child of node 4 i.e 4, so set current node = 4 < threshold, and explore its children i.e 8 & 7 here, both 8 & 7 > threshold so, pruned it. At the end of this, our pruned value is 7 & 8

Similarly, Explore the second child of root node 2 i.e 5 as the current node, i.e 5 = threshold, so, explored its children also i.e 6 & 6, i.e both 6 & 6 > threshold. So pruned it

So, our pruned value is 7,8 & 6

Iteration 4



In pruned values, the least value is 6, So threshold = 6

current node = root node = 2 and 2< threshold, So explore its children. i.e two children explore one by one

So, the first child is 4, So, set current node = 4 < threshold, so, explored its children also i.e 5, 4 having 5< threshold so explore its child also 7&8 > threshold. So, pruned it and explore the second child of node 4 i.e 4, so set current node = 4 < threshold, and explore its children i.e 8 & 7 here, both 8 & 7 > threshold so, pruned it. At the end of this, our pruned value is 7 & 8

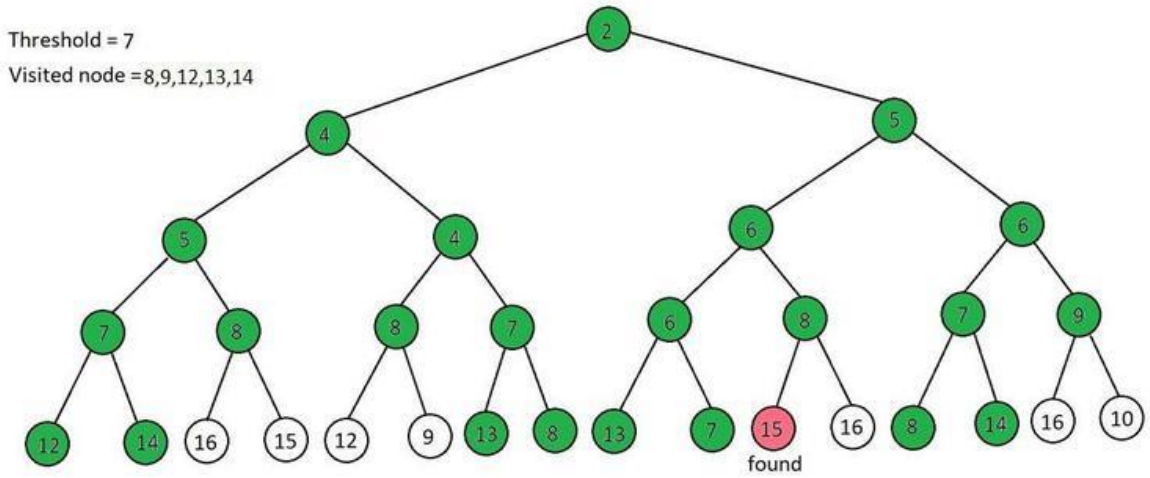
Similarly, Explore the second child of root node 2 i.e 5 as the current node, i.e 5 = threshold, so, explored its children also i.e 6 & 6, i.e both 6 & 6 = threshold, So, explore one by one,

The first 6 has two children i.e 6 & 8, having 6 = threshold. So, explore its child also i.e 13 & 7. here both 13 & 7 > Threshold. So, pruned it. next is 8 > Threshold. pruned it, So, pruned value at this stage is 13,7 & 8.

Explore the second child of 5 i.e 6 = Threshold. So, explore its child i.e 7 & 9. Both are greater than Threshold. So, pruned it

So, our pruned values are 13,7,8 & 9.

Iteration 5



In pruned values, the least value is 7, So threshold = 7

current node = root node = 2 and 2< threshold, So explore its children. i.e two children explore one by one

So, the first child is 4, So, set current node = 4 < threshold, so, explored its children also i.e 5, 4

The first child of 4 is 5 i.e 5< threshold so explore its child also 7&8, Here 7 = threshold. So, explore its children i.e 12 & 14, both > Threshold. So, pruned it. And the second child of 5 is 8 > Threshold, So, pruned it. At this stage, our pruned value is 12, 14 & 7.

Now explore the second child of node 4 i.e 4, so set current node = 4 < threshold, and explore its children i.e 8 & 7 here, 8 > threshold so, pruned it. then go to the second child i.e 7 = Threshold, So explore its children i.e 13 & 8. having both > Threshold. So pruned it. At the end of this, our pruned value is 12,14,8 & 13

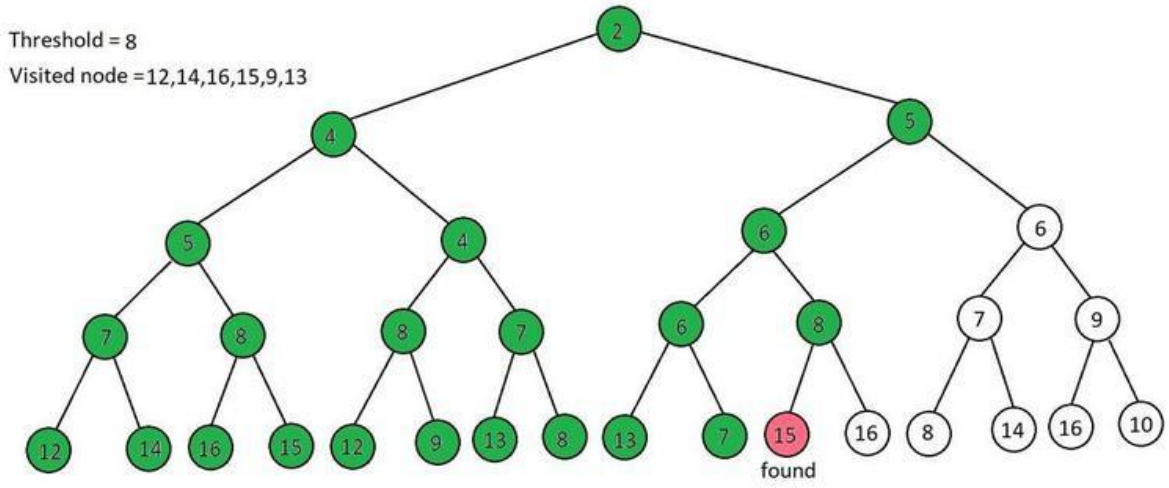
Similarly, Explore the second child of root node 2 i.e 5 as the current node, i.e 5 < threshold, so, explored its children also i.e 6 & 6, i.e both 6 & 6 < threshold, So, explore one by one,

The first 6 has two children i.e 6 & 8, having 6 < threshold. So, explore its child also i.e 13 & 7. here 13 > Threshold. So, pruned it. And 7= Threshold. And it hasn't any child. So, the shift to the next sub-child of 6 i.e 8 > threshold, So, pruned it. The pruned value at this stage is 12,14,8 &13

Explore the second child of 5 i.e 6 < Threshold. So, explore its child i.e 7 & 9. Here 7 = Threshold, So, explore its children i.e 8 & 14, Both are greater than Threshold. So, pruned it, Now the sub child of 6 is 9 > Threshold, So, pruned it.

So, our pruned values are 12,14,8,13 & 9.

Iteration 6



Iteration 6

In pruned values, the least value is 8, So threshold = 8

current node = root node = 2 and 2< threshold, So explore its children. i.e two children explore one by one

So, the first child is 4, So, set current node = 4 < threshold, so, explored its children also i.e 5, 4

The first child of 4 is 5 i.e 5< threshold so explore its child also 7&8, Here 7 < threshold. So, explore its children i.e 12 & 14, both > Threshold. So, pruned it. And the second child of 5 is 8 = Threshold, So, So, explore its children i.e 16 & 15, both > Threshold. So, pruned it. At this stage, our pruned value is 12, 14, 16 & 15.

Now explore the second child of node 4 i.e 4, so set current node = 4 < threshold, and explore its children i.e 8 & 7 here, 8 = Threshold, So, So, explore its children i.e 12 & 9, both > Threshold. So, pruned it. then go to the second child i.e 7 < Threshold, So explore its children i.e 13 & 8. having

13 > Threshold. So pruned it. and 8 = Threshold and it hasn't any child. At the end of this, our pruned values are 12, 14, 16, 15, and 13.

Similarly, Explore the second child of root node 2 i.e 5 as the current node, i.e 5 < threshold, so, explored its children also i.e 6 & 6, i.e both 6 & 6 < threshold, So, explore one by one,

The first 6 has two children i.e 6 & 8, having 6 < threshold. So, explore its child also i.e 13 & 7. here 13 > Threshold. So, pruned it. And 7<Threshold. And it hasn't any child. So, the shift to the next sub-child of 6 i.e 8 = threshold. So, explored its children also i.e 15 & 16, Here **15 = Goal Node**. So, stop this iteration. Now no need to explore more.

The goal path is **2–>5–>6–>8–>15**

IDA* can be used to solve various real-life problems that are:

The 15-Puzzle Problem:

The 15-puzzle problem is a classic example of a **sliding puzzle game**. It consists of a 4×4 grid of numbered tiles with one tile missing. The aim is to rearrange the tiles to form a specific goal configuration. The state space of the puzzle can be represented as a tree where each node represents a configuration of the puzzle and each edge represents a legal move. IDA* can be used to find the **shortest sequence of moves** to reach the goal state from the initial state.

The 8-Queens Problem:

The 8-Queens problem is a classic example of the **n-Queens problem** where n queens have been placed on an n x n chessboard such that no two queens attack each other. The state space of the problem can be represented as a tree where each node represents a configuration of the chessboard and each edge represents the placement of a queen. IDA* can be used to find the **minimum number of queens** that need to be moved to reach the goal state.

In both of these examples, IDA* is used to find the optimal solution by using a combination of depth-first search and a heuristic function to limit the search space. The algorithm incrementally increases the depth bound, allowing it to find the solution without exploring the entire state space, which would be infeasible for larger problems.

Advantages and disadvantages

Advantages

IDA* is **guaranteed** to find the optimal solution if one exists.

IDA* avoids the exponential **time complexity** of traditional Depth First Search. by using an “iterative deepening” approach, where the search depth is gradually increased.

IDA* uses a **limited amount of memory** as compared to the A* algorithm because it uses Depth First Search.

IDA* is an admissible heuristic, it **never overestimates** the cost of reaching the goal.

It’s efficient in handling large numbers of states and large branch factors.

Disadvantages

Explore the visited node again and again. it doesn’t keep track of the visited nodes.

IDA* may be **slower** to get a solution than other search algorithms like A* or Breadth-First Search because it explores and repeats the explore node again and again.

It takes more time and power than the A* algorithm.

It’s important to note that IDA* is not suitable for all types of problems, and the choice of algorithm will depend on the specific characteristics of the problem you’re trying to solve.

Branch and bound

What is Branch and bound?

Branch and bound is one of the techniques used for problem solving. It is similar to the backtracking since it also uses the state space tree. It is used for solving the optimization problems and minimization problems. If we have given a maximization problem then we can convert it using the Branch and bound technique by simply converting the problem into a maximization problem.

Let's understand through an example.

Jobs = {j1, j2, j3, j4}

P = {10, 5, 8, 3}

d = {1, 2, 1, 2}

The above are jobs, problems and problems given. We can write the solutions in two ways which are given below:

Suppose we want to perform the jobs j1 and j2 then the solution can be represented in two ways:

The first way of representing the solutions is the subsets of jobs.

S1 = {j1, j4}

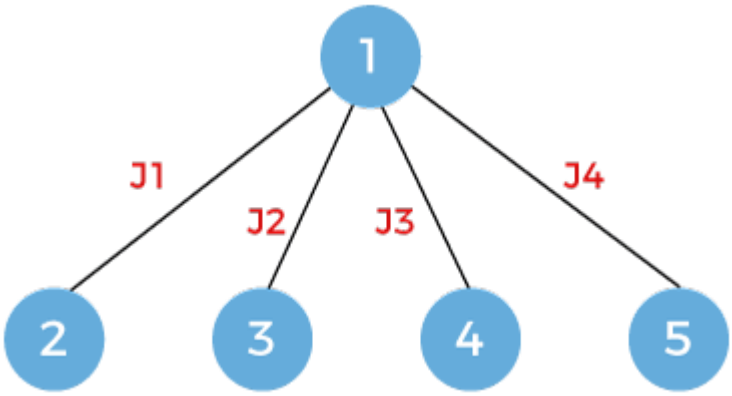
The second way of representing the solution is that first job is done, second and third jobs are not done, and fourth job is done.

S2 = {1, 0, 0, 1}

The solution s1 is the variable-size solution while the solution s2 is the fixed-size solution.

First, we will see the subset method where we will see the variable size.

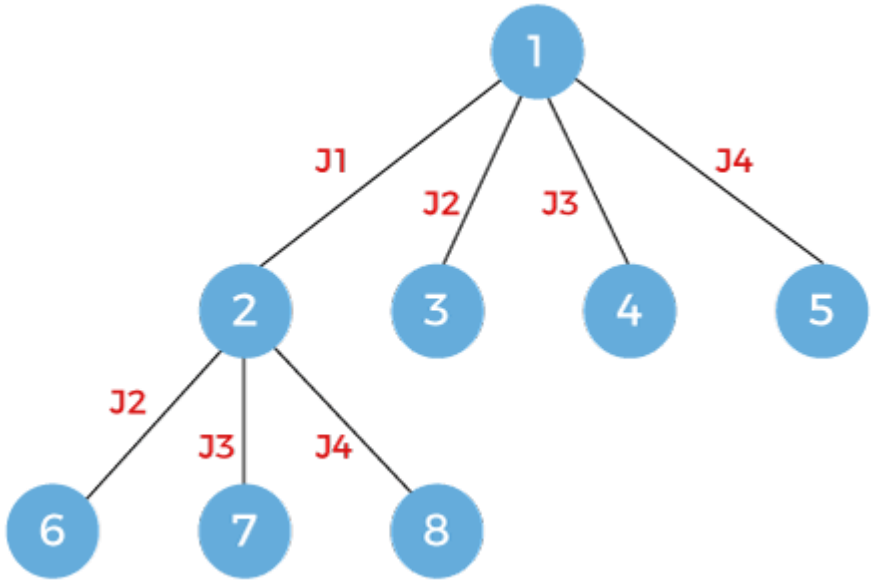
First method:



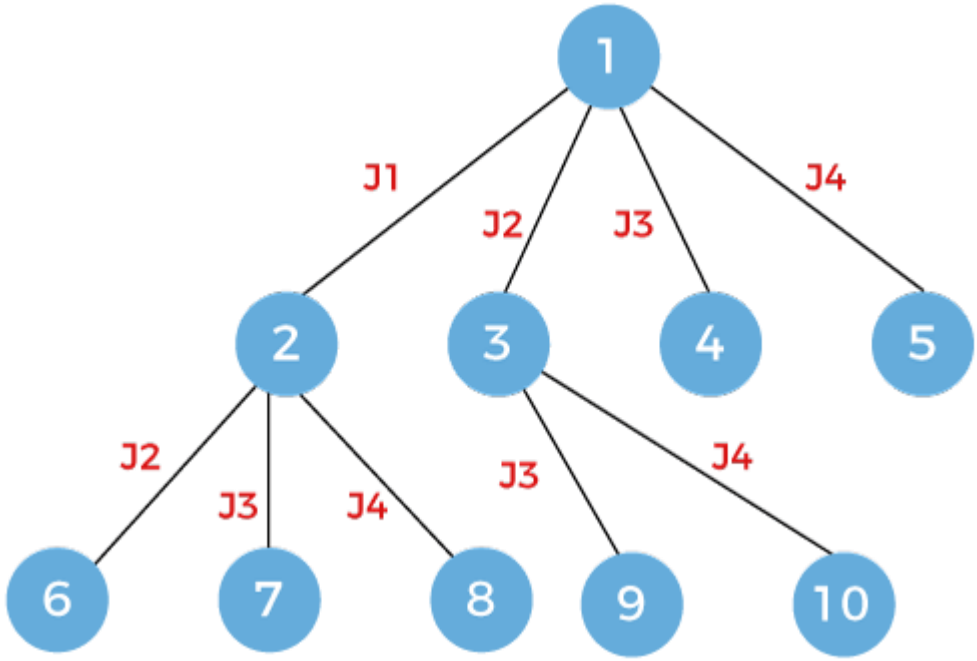
In this case, we first consider the first job, then second job, then third job and finally we consider the last job.

As we can observe in the above figure that the breadth first search is performed but not the depth first search. Here we move breadth wise for exploring the solutions. In backtracking, we go depth-wise whereas in branch and bound, we go breadth wise.

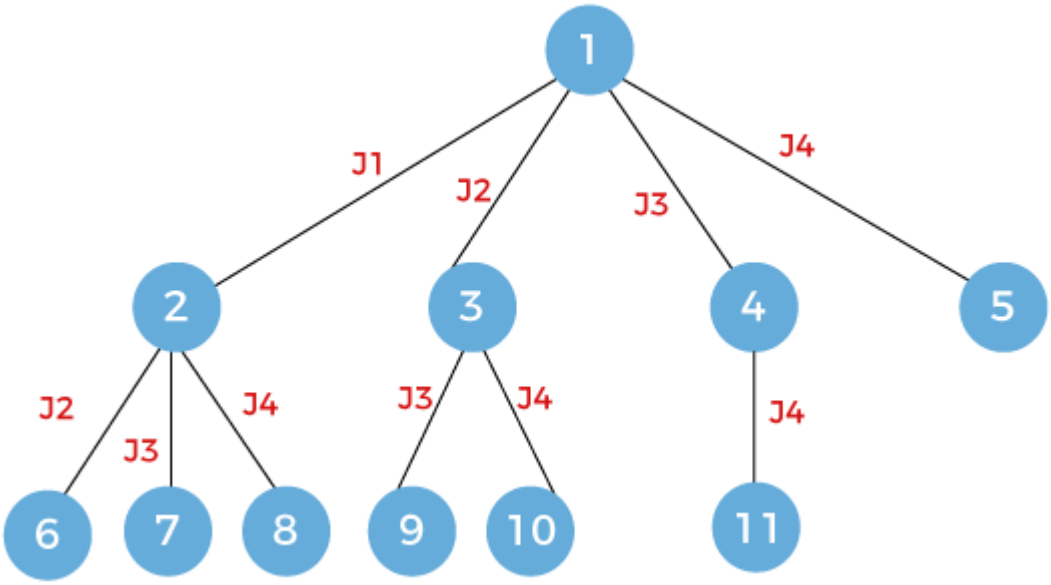
Now one level is completed. Once I take first job, then we can consider either j2, j3 or j4. If we follow the route then it says that we are doing jobs j1 and j4 so we will not consider jobs j2 and j3.



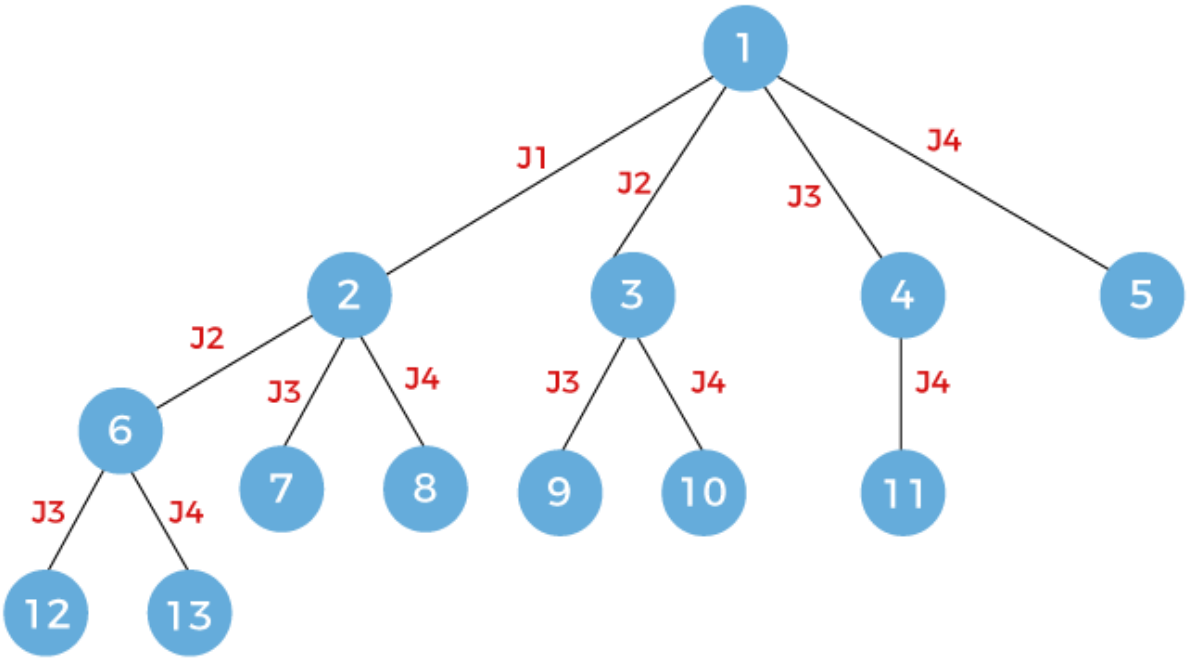
Now we will consider the node 3. In this case, we are doing job j2 so we can consider either job j3 or j4. Here, we have discarded the job j1.



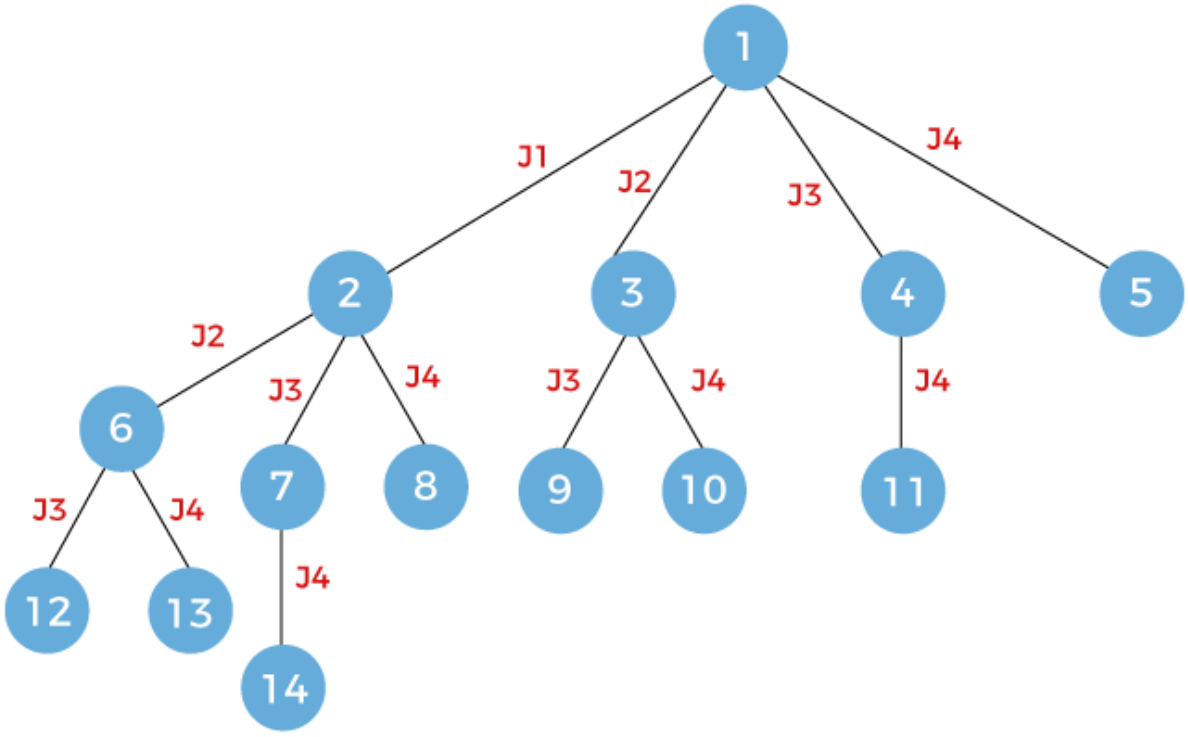
Now we will expand the node 4. Since here we are doing job j3 so we will consider only job j4.



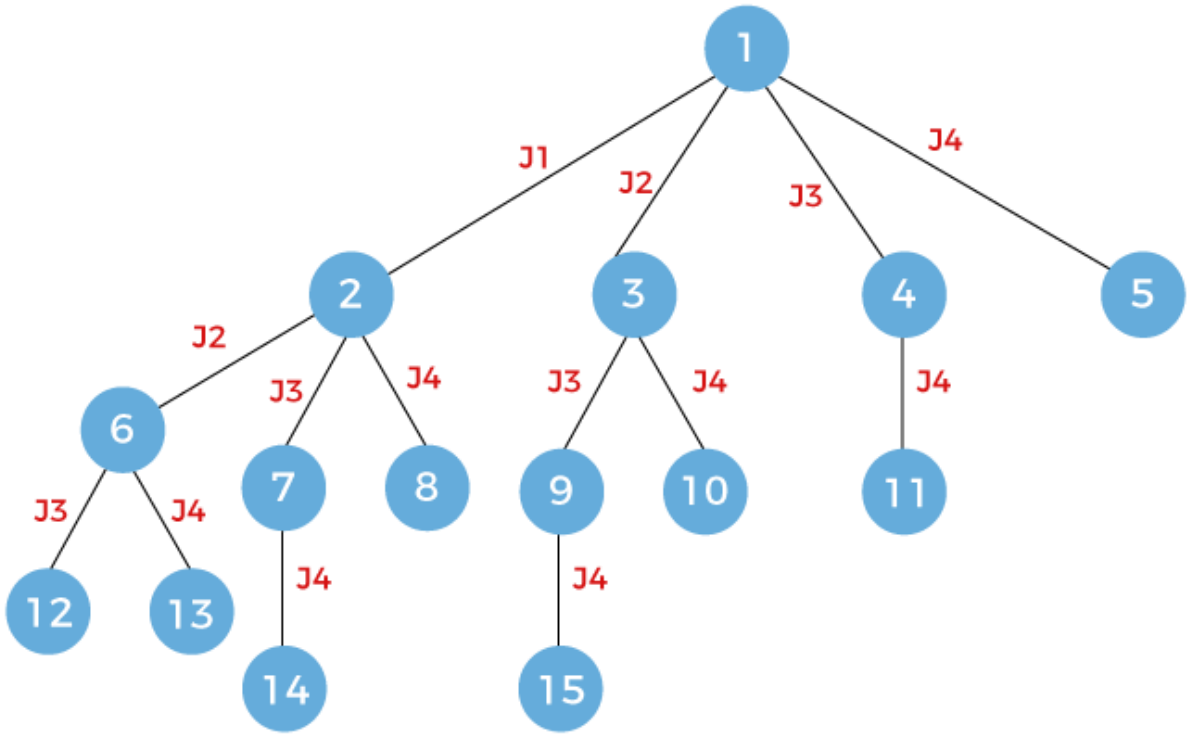
Now we will expand node 6, and here we will consider the jobs j3 and j4.



Now we will expand node 7 and here we will consider job j4.



Now we will expand node 9, and here we will consider job j4.



The last node, i.e., node 12 which is left to be expanded. Here, we consider job j4.

The above is the state space tree for the solution s1 = {j1, j4}

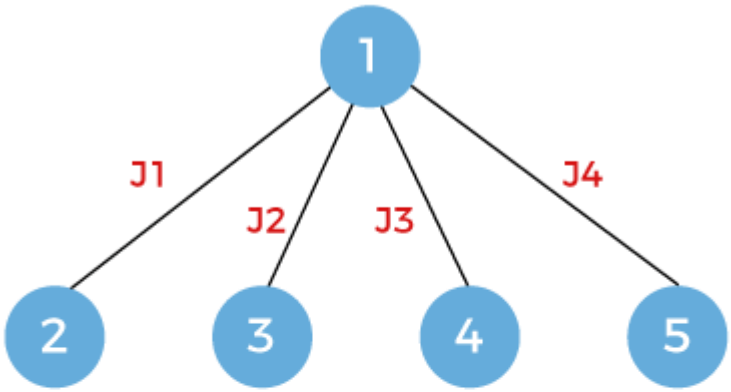
Second method:

We will see another way to solve the problem to achieve the solution s1.

First, we consider the node 1 shown as below:

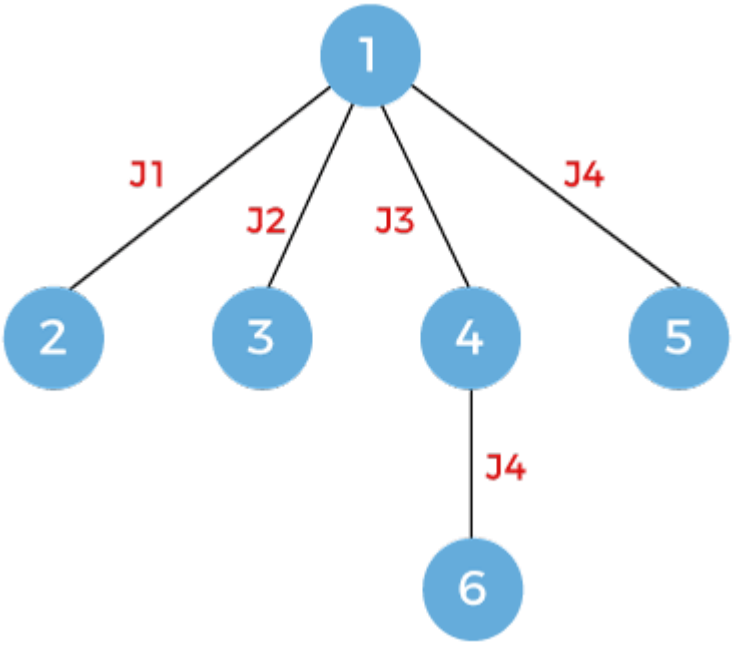
Now, we will expand the node 1. After expansion, the state space tree would be appeared as:

On each expansion, the node will be pushed into the stack shown as below:

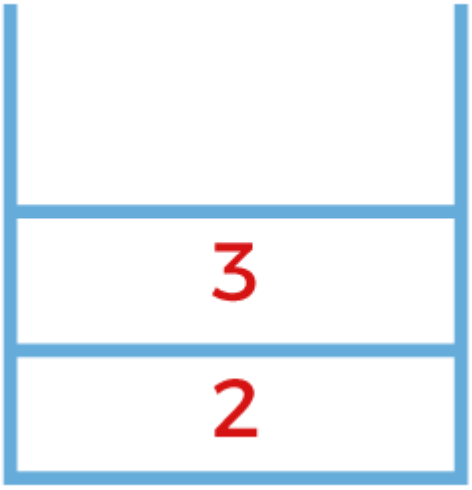


Now the expansion would be based on the node that appears on the top of the stack. Since the node 5 appears on the top of the stack, so we will expand the node 5. We will pop out the node 5 from the stack. Since the node 5 is in the last job, i.e., j4 so there is no further scope of expansion.

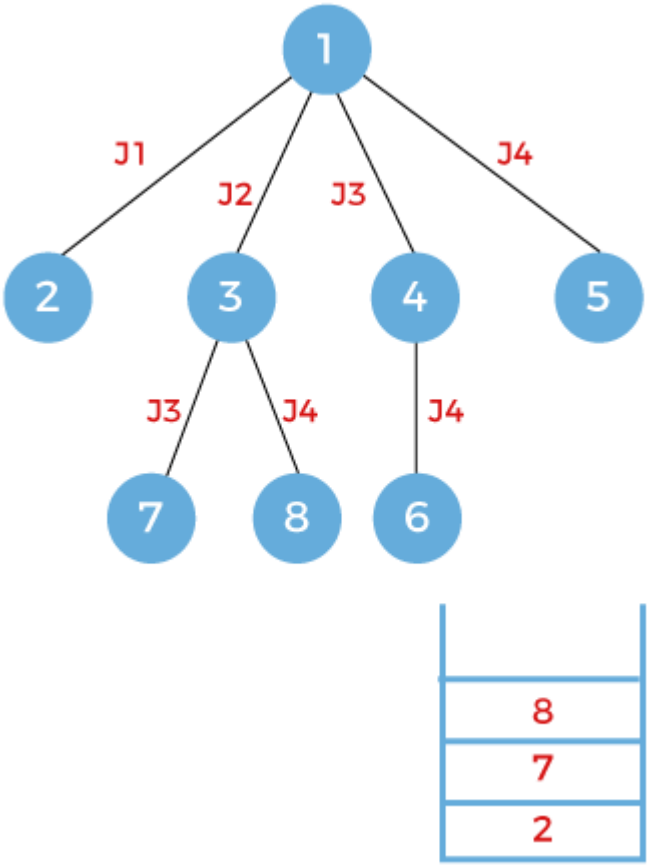
The next node that appears on the top of the stack is node 4. Pop out the node 4 and expand. On expansion, job j4 will be considered and node 6 will be added into the stack shown as below:



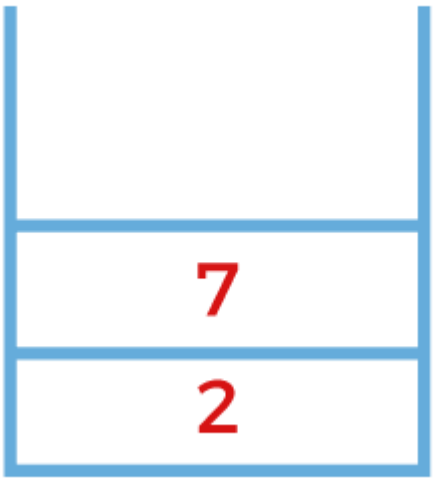
The next node is 6 which is to be expanded. Pop out the node 6 and expand. Since the node 6 is in the last job, i.e., j4 so there is no further scope of expansion.



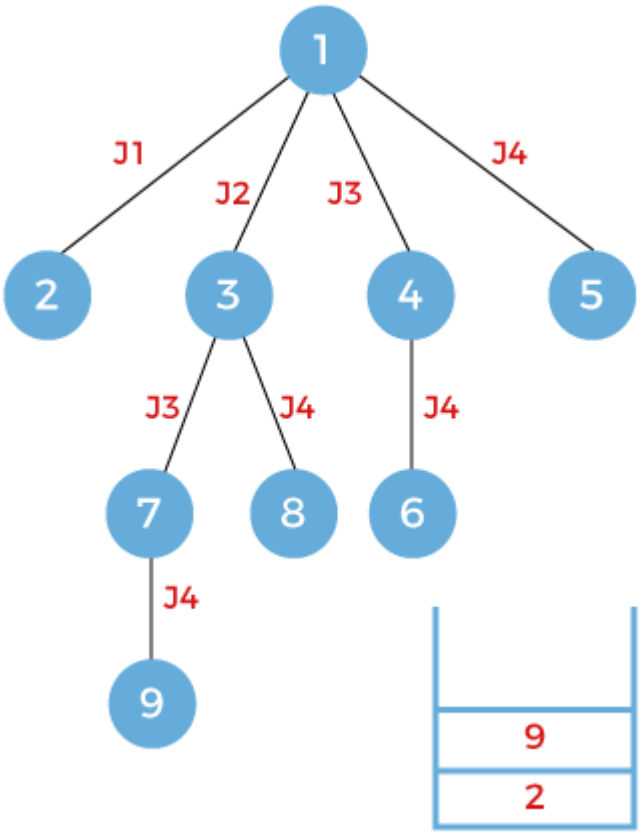
The next node to be expanded is node 3. Since the node 3 works on the job j2 so node 3 will be expanded to two nodes, i.e., 7 and 8 working on jobs 3 and 4 respectively. The nodes 7 and 8 will be pushed into the stack shown as below:



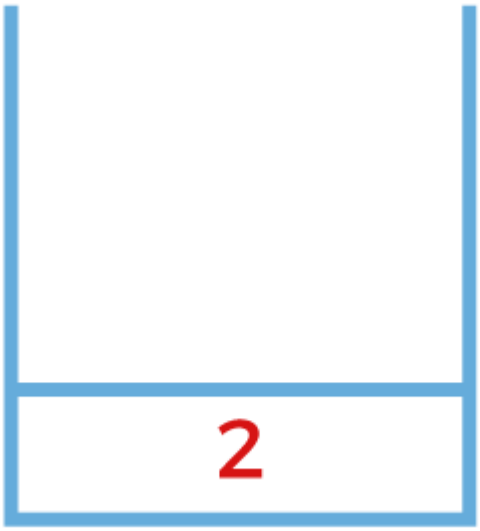
The next node that appears on the top of the stack is node 8. Pop out the node 8 and expand. Since the node 8 works on the job j4 so there is no further scope for the expansion.



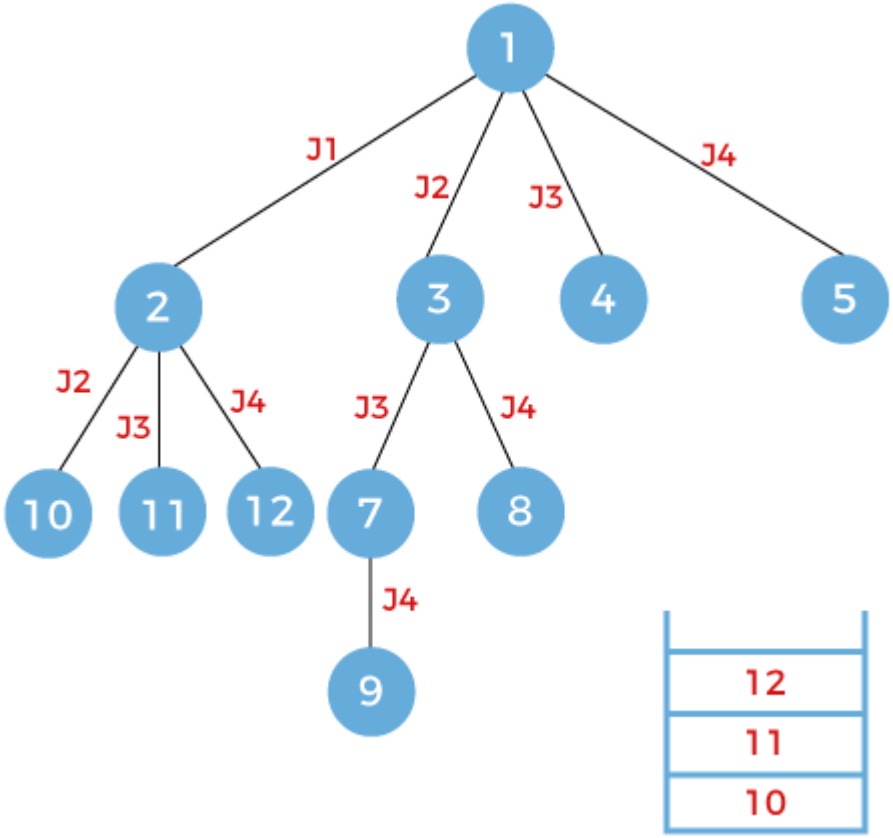
The next node that appears on the top of the stack is node 7. Pop out the node 7 and expand. Since the node 7 works on the job j3 so node 7 will be further expanded to node 9 that works on the job j4 as shown as below and the node 9 will be pushed into the stack.



The next node that appears on the top of the stack is node 9. Since the node 9 works on the job 4 so there is no further scope for the expansion.



The next node that appears on the top of the stack is node 2. Since the node 2 works on the job j1 so it means that the node 2 can be further expanded. It can be expanded upto three nodes named as 10, 11, 12 working on jobs j2, j3, and j4 respectively. There newly nodes will be pushed into the stack shown as below:



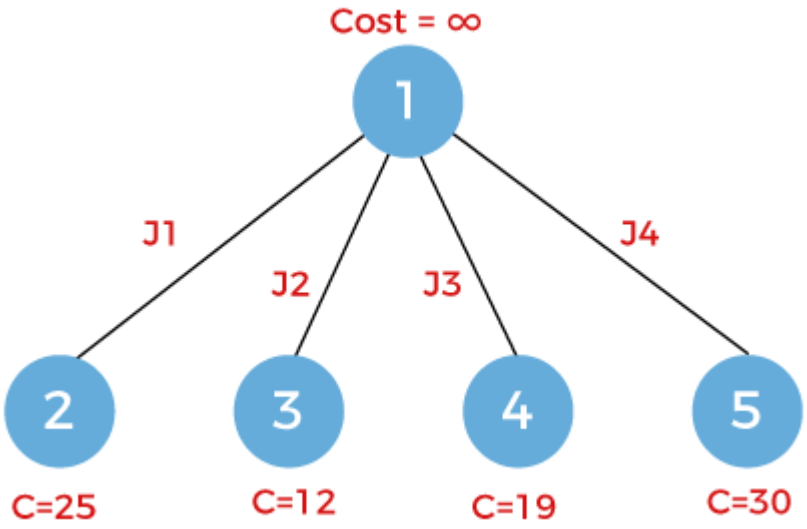
In the above method, we explored all the nodes using the stack that follows the LIFO principle.

Third method

There is one more method that can be used to find the solution and that method is Least cost branch and bound. In this technique, nodes are explored based on the cost of the node. The cost of the node can be defined using the problem and with the help of the given problem, we can define the cost function. Once the cost function is defined, we can define the cost of the node.

Let's first consider the node 1 having cost infinity shown as below:

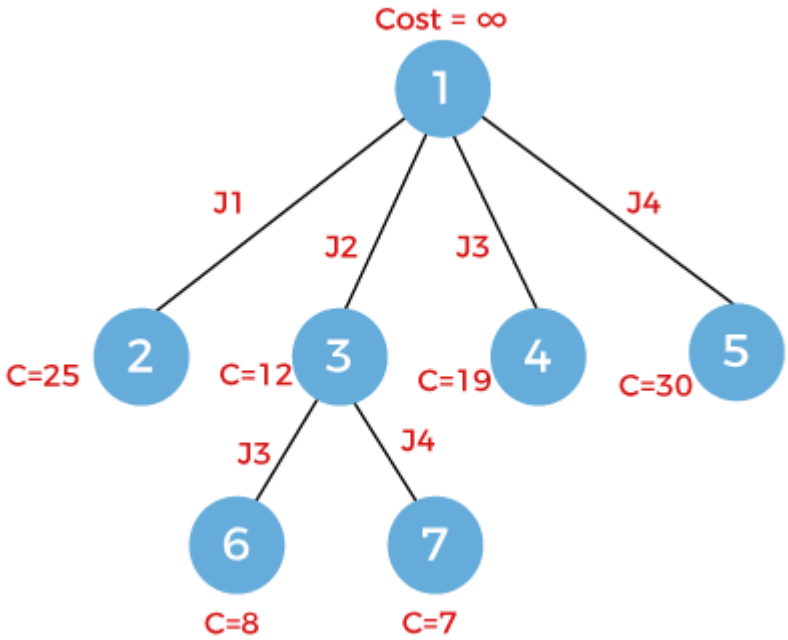
Now we will expand the node 1. The node 1 will be expanded into four nodes named as 2, 3, 4 and 5 shown as below:



Let's assume that cost of the nodes 2, 3, 4, and 5 are 25, 12, 19 and 30 respectively.

Since it is the least cost branch n bound, so we will explore the node which is having the least cost. In the above figure, we can observe that the node with a minimum cost is node 3. So, we will explore the node 3 having cost 12.

Since the node 3 works on the job j2 so it will be expanded into two nodes named as 6 and 7 shown as below:

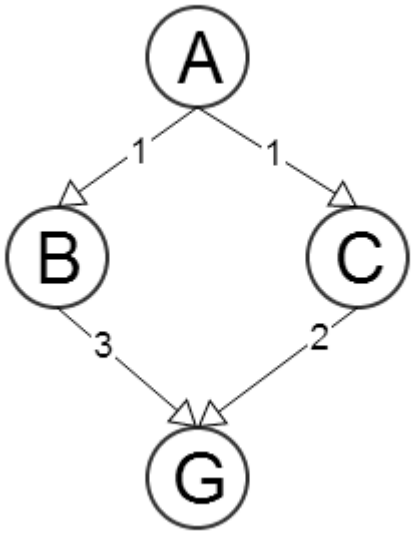


The node 6 works on job j3 while the node 7 works on job j4. The cost of the node 6 is 8 and the cost of the node 7 is 7. Now we have to select the node which is having the minimum cost. The node 7 has the minimum cost so we will explore the node 7. Since the node 7 already works on the job j4 so there is no further scope for the expansion.

What are Admissible heuristics?

Admissible heuristics are used to estimate the cost of reaching the goal state in a search algorithm. Admissible heuristics never overestimate the cost of reaching the goal state. The use of admissible heuristics also results in optimal solutions as they always find the cheapest path solution.

For a heuristic to be admissible to a search problem, needs to be lower than or equal to the actual cost of reaching the goal.



Here is an example:

In the A* search algorithm, the evaluation function (where {\displaystyle n}n is the current node) is:

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

where

f(n) = evaluation function.

g(n) = cost from start node to current node

h(n) = estimated cost from current node to goal

Here, h(n) gets calculated with the use of the heuristic function. If a non-admissible heuristic was used, it is possible that the algorithm would not reach the optimal solution because of an overestimation in the evaluation function.

What is non-admissible heuristic?

Non-admissible heuristics may overestimate the cost of reaching the goal state. There is no guarantee that they will reach an optimal solution.

What happens when a heuristic is not admissible?

When a non-admissible heuristic is used in an algorithm, it may or may not result in an optimal solution.

But, sometimes non-admissible heuristics expand a smaller amount of nodes. As a result, it is possible that the total cost (search cost + path cost) could end up being lower than an optimal solution that would be found by using an admissible heuristic.

What is the difference between admissible heuristics & consistent heuristics?

A heuristic is considered to be consistent if the estimated cost from one node to the successor node, added to the estimated cost from the successor node to the goal is less than or equal to the estimated cost from the current node to the goal state.

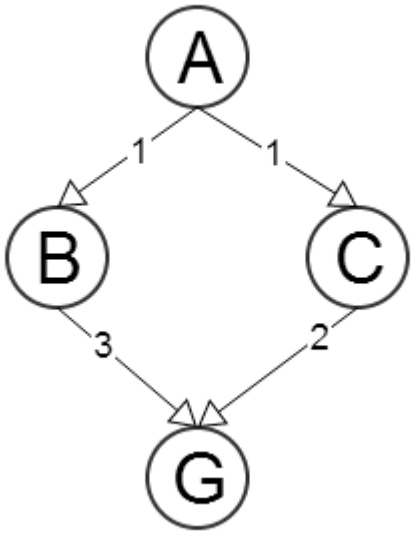
In an admissible heuristic, the estimated cost from the current node to the goal state is never greater than the actual cost.

All consistent heuristics are admissible heuristics, however, all admissible heuristics are not necessarily consistent heuristics.

How does an admissible heuristic ensure an optimal solution?

If the heuristic function isn't admissible, then it is possible to have an estimation that is larger than the actual path cost from some node to a goal node. If this higher path cost estimation is on the least cost path (that you are trying to find), the algorithm will not explore it and it may find another (not the cheapest) path to the goal.

Consider this example:



Say A and G are the starting and goal nodes respectively. Now let $h(N)$ be an estimate of the path's length from node N to G, $\forall N$ in the graph. Say $c(N,Xi)$ is the step cost function from node N to its neighbor Xi, $\forall N$ and i=1..m, where m is the number of neighbors of N (i.e., a function that returns the cost of the edge between node N and one of its neighbors).

Let the heuristics be

- $h(B)$=3
- $h(C)$=4

This heuristics function H will not be admissible, because

$h(C)$=4>$c(C,G)$=2

If the $A*$ algorithm starts from node A, it will then select the node B for the purpose of expansion and, after this, it will proceed to node G from there. And the path will be $A \rightarrow B \rightarrow G$ with cost 4, instead of $A \rightarrow C \rightarrow G$ with cost 3. If the heuristic function was admissible this would not have happened.

if the heuristic had been admissible A->B could be chosen for the next node to expand, but after that, the A* would select A->C instead of A->B->G. And in the end, it would end up with A->C->G.

This is because of the way in which A* works. It expands the node that has the least sum of the distance to that node + heuristic estimation from that node. d(A,G) + h(G) = 4 + 0 = 4 and d(A,C) + h(C) = 1 + something<=2 (because it is admissible). C has the lower sum and hence A* will pick it. In the same way, it will then expand G and identify the least path.

Admissible heuristics make sure to find the shortest path with the least cost. The solution itself will be optimal if the heuristic is consistent.

Here is an alternative explanation:

The fact that the heuristic is admissible means that it does not overestimate the effort to reach the goal. i.e., $h(n) \leq h*(n)$ for all n in the state space (in the 8-puzzle, which means is that just for any permutation of the tiles and the goal you are currently considering) where $h*(n)$ is the optimal cost to reach the target.

The most logical reason why $A*$ offers optimal solutions if $h(n)$ is admissible is due to the fact that it sorts all nodes in OPEN in ascending order of $f(n)=g(n)+h(n)$ and, also, because it does not stop when generating the goal but when expanding it.

Due to the fact that nodes are expanded in ascending order of $f(n)$ you know that no other node is more promising than the current one. $h(n)$ is admissible so that having the lowest $f(n)$ means that it has an opportunity to reach the goal via a cheaper path that the other nodes in OPEN have not. This holds true unless you can manage to prove the opposite, i.e., by expanding the current node.

Because $A*$ will only stop when it proceeds to expand the goal node (instead of stopping when generating it) you can be absolutely sure that no other node leads to it via a cheaper path.