Unit II

INTRODUCTION TO SEARCH:- Searching for solutions, Uniformed search strategies, Informed search strategies, Local search algorithms and optimistic problems, Adversarial Search, Search for games, Alpha - Beta pruning.

PROBLEM SOLVING THROUGH AI

Problem solving = Searching for a goal state.

- Take the problem real-world problem and then give solutions to those problems with the help of searching techniques.
- State Space Search is the primary tool for successful design and implementation of search algorithms.

The method of solving problem through AI involves the process of defining the search space, deciding start and goal state and then finding the path from start stated to goal state through search space.

The movement from start state to goal state is guided by set of rules specifically designed for that particular problem called the production rules. The production rules are the valid moves described by the problems.

State space search: State space search is a process used in the field of computer science, including artificial intelligence (AI), in which successive configurations or states of an instance are considered, with the goal of finding a goal state with a desired property.

Problem: It is the question which is to be solved. For solving the problem it needs to be precisely defined. The definition means, defining the start state, goal state, other valid states and transitions.

Search space: It is the complete set of states including start and goal states, where the answer of the problem is to be searched.

Search: It is the process of finding the solution in search space. The input to search space algorithm is problem and output is solution in form of action sequence.

Well-defined problem: A problem description has three major components: initial state, final (goal) state, space including transition function or path function. A path cost function assigns some numeric value to each path that indicates the goodness of that path. Sometimes a problem may have additional component in the form of heuristic information.

Solution of the problem: A solution of the problem is a path from initial state to goal state. The movement from start states to goal state is guided by transition rules. Among all the solutions, whichever solution has least path cost is called optimal solution.

Thus, to build an AI computational system to solve a particular problem the following activities are needed to be performed:

1. Define the problem precisely: definition must include precise specifications of initial states and final states of the problem.
2. Analyze the problem: abstract the salient features of the problem that can have an immense impact on the appropriateness of various possible techniques used for solving the problem.
3. Isolate and represent the task knowledge that is necessary to solve the problem
4. Choose the best problem solving technique and apply it to the particular problem.

Example 1:

The eight tile puzzle problem formulation

The eight tile puzzle consist of a 3 by 3 (3*3) square frame board which holds 8 movable tiles numbered 1 to 8. One square is empty, allowing the adjacent tiles to be shifted. The objective of the puzzle is to find a sequence of tile movements that leads from a starting configuration to a goal configuration.

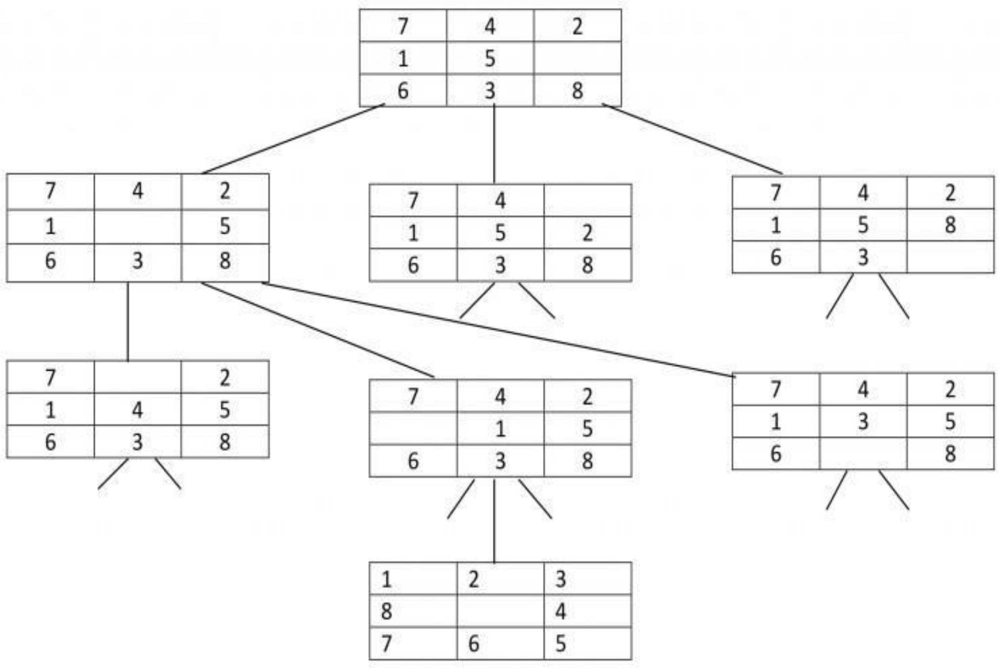
7	4	2
	5	
6	3	8

Transformed to be:

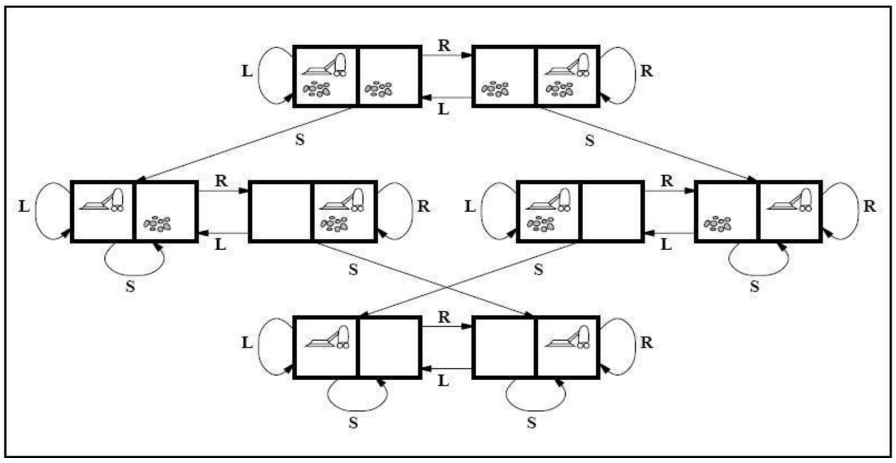
1	2	3
8		4
7	6	5

- States: It specifies the location of each of the 8 tiles and the blank in one of the nice squares.
- Initial state : Any state can be designated as the initial state.
- Goal : Many goal configurations are possible one such is shown in the figure
- Legal moves (or state) : They generate legal states that result from trying the four actions-
  - Blank moves left
  - Blank moves right
  - Blank moves up
  - Blank moves down
- Path cost: Each step costs 1, so the path cost is the number of steps in the path.

The tree diagram showing the search space is shown in figure:



Example 2: **Vacuum World**



State Space for the vacuum world. Links denote actions: L=Left, R=Right, S=Suck.

The problem is formulated as follows:

- States: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus there are $2 * 2^2 = 8$ possible world states. A larger environment with *n* locations has $n * 2^n$ states.
- Initial State: Any state can be designated as the initial state.
- Actions: In this simple environment, each state has just three actions: *Left, Right,* and *Suck*. Larger environments might also include *Up* and *Down*.
- Transition Model: The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Sucking* in a clean square have no effect.
- Goal Test: This checks whether all the squares are clean.
- Path Cost: Each steps costs 1, so the path cost is the number of steps in the path.

Compared with the real world, this toy problem has discrete locations, discrete dirt, reliable cleaning, and it never gets messed up once cleaned.

Note:

Toy vs. Real-world problem - The problem-solving approach has been applied to a vast array of task environments. We must describe the distinguishing between toy and real-world problems: A toy problem is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is usable by different researchers to compare the performance of algorithms. A real-world problem is one whose solutions people actually care about. They tend not to have a single agreed-upon description, so we will do our best describing the formulations used. Examples are as follows:

Toy problems:	**Real World problems:**
– 8-puzzle – 8-queen/n-queen – cryptarithmetic – vacuum world – missionaries and cannibals	– Touring problems – Traveling Salesperson (TSP) – VLSI layout – Robot navigation – Automatic assembly sequencing – Internet searching

Problem Representation in AI

The most common methods of problem representation in AI are:-

1. State Space Representation
2. Problem Reduction

State Space Representation: A set of all possible states for a given problem is known as the state space of the state space of the problem. Suppose you are asked to make a cup of coffee. What will you do? You will verify whether the necessary ingredients like instant coffee powder, milk powder, sugar, kettle, stove etc. are available.

If so, you will follow the following steps:

1. Boil necessary water in the kettle.
2. Take some of the boiled water in a cup add necessary amount of instant coffee powder to make decoction.
3. Add milk powder to the remaining boiling water to make milk.
4. Mix decoction and milk.
5. Add sufficient quantity of sugar to your taste and the coffee is ready.

Problem Reduction**:** In this method a complex problem is broken down or decomposed into a set of primitive sub problems. Solutions for these primitive sub-problems are easily obtained. The solutions for all the sub-problems collectively give the solution for the complex problem.

PRODUCTION SYSTEM

The production system is a model of computation that can be applied to implement search algorithms and model human problem solving. Such problem solving knowledge can be packed up in the form of little quanta called productions. A production is a rule consisting of a situation recognition part and an action part.

If one adapts the system with production rule and rule interpreter then that system is called production system.

A production system consists of following components.

1. A set of production rules, which are of the form A→B. Each rule consists of left hand side constituent that represent the current problem state and a right hand side that represent an output state. A rule is applicable if its left hand side matches with the current problem state.
2. A database, which contains all the appropriate information for the particular task. Some part of the database may be permanent while some part of this may pertain only to the solution of the current problem.
3. A control strategy that specifies order in which the rules will be compared to the database of rules and a way of resolving the conflicts that arise when several rules match simultaneously.
4. A rule applier, which checks the capability of rule by matching the content state with the left hand side of the rule and finds the appropriate rule from database of rules.

Four classes of production systems:-

1. A monotonic production system
2. A non monotonic production system
3. A partially commutative production system
4. A commutative production system

Advantages of production systems

- Production systems provide an excellent tool for structuring AI programs.

- Production Systems are highly modular because the individual rules can be added, removed or modified independently.
- The production rules are expressed in a natural form, so the statements contained in the knowledge base should the recording of an expert thinking out loud.

Disadvantages of Production Systems

- One important disadvantage is the facts that it may be very difficult analyze the flow of control within a production system because the individual rules don't call each other.

Note: A Production System is best suited AI Technique to solve the State Space Problems like Water Jug, and Missionaries and Cannibals. these problems consists of State, State Space and Goal State. State = Initial state of the Problem. State space = Intermediate States between Initial State and Goal State. Goal State = Final state of the problem. - to solve state space problems Production System is best suited AI Technique

Water-Jug Problem:

Statement: We are given 2 jugs, a 4 liter one and a 3- liter one. Neither have any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can we get exactly 2 liters of water in to the 4-liter jugs?

The state space for this problem can be defined as

{(x, y) where x = 0,1,2,3,4 & y = 0,1,2,3}

'x' represents the number of liters of water in the 4-liter jug and 'y' represents the number of liters of water in the 3-liter jug. The initial state is (0, 0) that is no water on each jug. The goal state is to get (2, n) for any value of 'n'.

To solve this we have to make some assumptions not mentioned in the problem. They are

1. **We can fill a jug from the pump.**
2. **We can pour water out of a jug to the ground.**
3. **We can pour water from one jug to another.**
4. **There is no measuring device available.**

The various operators (Production Rules) that are available to solve this problem may be stated as given in the following figure:

The production rules are formulated as follows:-

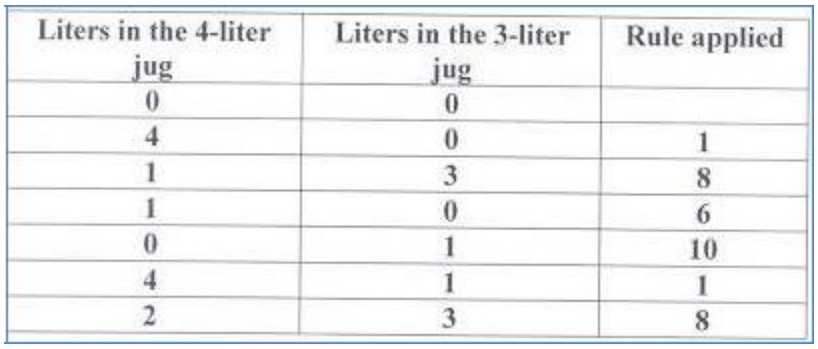
- Rule 1: (x, y) -> (4, y) (Fill the 4 liter jug)
- Rule 2: (x, y) -> (x, 3) (Fill the 3 liter jug)
- Rule 3: (x, y) -> (x-x1, y) (Pour some water out from 4 liter jug)
- Rule 4: (x, y) -> (x, y-x1) (Pour some water out from 3 liter jug)
- Rule 5: (x, y) -> (0, y) (Empty the 4 liter jug)
- Rule 6: (x, y) -> (x, 0) (Empty the 3 liter jug)
- Rule 7: (x, y) -> (4, y-(4-x)) (Fill the 4 liter jug by pouring some water from 3 liter jug)
- Rule 8: (x, y) -> (x-(3-y), 3) (Fill the 3 liter jug by pouring some water from 4 liter jug)
- Rule 9: (x, y) -> (x +y, 0) (Empty 3 liter jug by pouring all its water in to 4 liter jug)
- Rule 10: (x, y) -> (0, x +y) (Empty 4 liter jug by pouring all its water in to 3 liter jug)
- ~~Rule 11: (0, 2) -> (2, 0) (Pour the 2 liters from 3 liter jug into 4 liter jug)~~
- ~~Rule 12: (2, y) -> (0, y) (Empty the 2 liters in the 4 liter jug on the ground)~~

These are set of rules, which can be applied to solve water-jug problem. Solution of the problem will include applying appropriate rules in the specific sequence to transform the start state to goal state.

One solution is applying the rules in the sequence 2, 9, 2, 7, 5, 9. The solution is presented in the following table:-

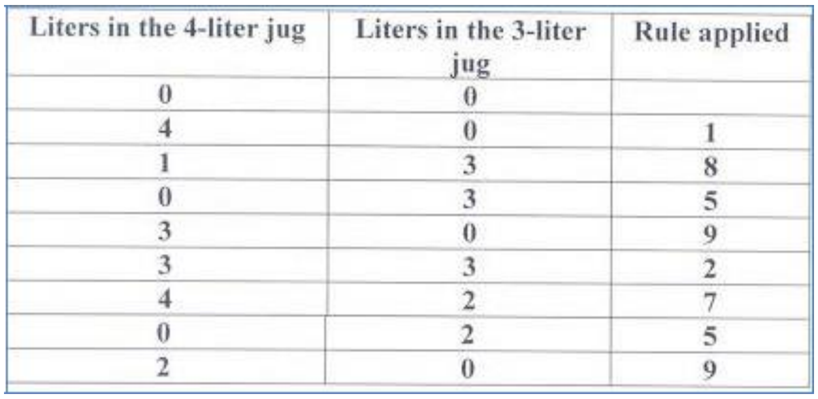
Rule Applied	Water in 4 liter jug	Water in 3 liter jug
Start State	0	0
2	0	3
9	3	0
2	3	3
7	4	2
5	0	2
9	2	0

Solution 2:



Liters in the 4-liter jug	Liters in the 3-liter jug	Rule applied
0	0	
4	0	1
1	3	8
1	0	6
0	1	10
4	1	1
2	3	8

Solution 3:



Liters in the 4-liter jug	Liters in the 3-liter jug	Rule applied
0	0	
4	0	1
1	3	8
0	3	5
3	0	9
3	3	2
4	2	7
0	2	5
2	0	9

Measuring Problem-Solving Performance

- Completeness: Is the algorithm guaranteed to find a solution when there is one.
- Optimality: Does the strategy find the optimal solution.
- Time Complexity: How long does it take to find solution?
- Space Complexity: How much memory is needed to perform the search?

AI & Search Process:

- Every AI program has to do the process of searching for the solution steps are not explicit in nature.
- This searching is needed for solution steps that are not known beforehand and have to be found out.

Basically, to do a search process, the following are needed.

- The Initial State description of the problem. For example, the initial positions of all pieces in the chess board.
- A set of logical operators that change the state. In chess game, it is the rules of the game.
- The final or the goal state. The final position that a problem solver has to reach in order to succeed.

Searching can be defined as a sequence of steps that transforms the initial state to the goal state. The searching process in AI can be broadly classified into two major types:

1. Brute Force Search or uninformed or blind search.
2. Heuristic Search or informed search.

Brute Force Search:

While searching you have no clue whether one non-goal state is better than any other. Your search is blind. You don't know if your current exploration is likely to be fruitful. Such kind of search are referred as Brute Force Search.

Brute force search is also known as uninformed search or Blind search. These are commonly used search procedures which explore all the alternatives during the search process. They do not have any domain-specific knowledge. All they need are the initial state, the final state and a set of legal operators. It is a search methodology having no additional information about states beyond that provided in the problem definitions. In this search total search space is looked for the solution.

Various Brute Force search are:

- Breadth-first search
- Uniform-cost search
- Depth-first search and Depth-limited search
- Iterative deepening depth-first search
- Bidirectional search

Heuristic Search**:** Heuristic search is also known as guided search or informed search. These are search techniques where additional information about the problem is provided in order to guide the search in a specific direction.

Heuristics are approximations used to minimize the searching process. Generally, two categories of problems use heuristics.

- Problems for which no exact algorithms are known and one needs to find an approximate and satisfying solution. e.g. speech recognition.
- Problems for which exact solutions are known, but computationally infeasible e.g. Rubik's Cube, Chess etc.

The heuristics which are needed for solving problems are generally represented as a heuristic function which maps the problem states into numbers. These numbers are then approximately used to guide search. The following algorithms make use of heuristic evaluation functions:

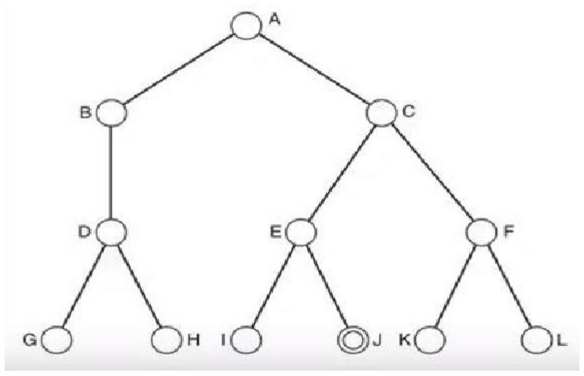
(a) Hill Climbing
(b) Best-First Search
(c) A* Algorithm
(d) AO* Algorithm
(e) Beam Search
(f) Constraint Satisfaction
(g) Min-Max Search
(h) Alpha - Beta pruning

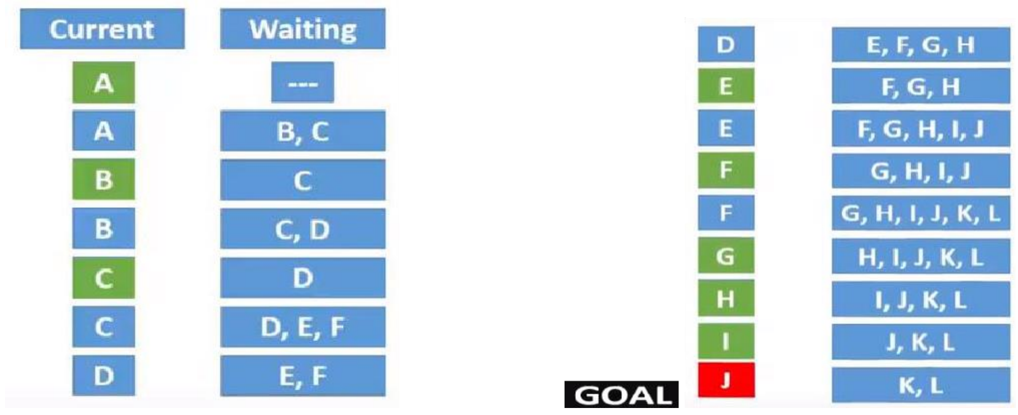
<u>Breadth-First Search (BFS):</u>

- Breadth First Search (BFS) searches breadth-wise in the problem space.
- BFS was invented in late 1950s by E. F. Moore, who used it find shortest path out of a maze.
- Breadth-First search is like traversing a tree where each node is a state which may be a potential candidate for solution.
- It expands nodes from the root of the tree and then generates one level of the tree at a time until a solution is found.
- It is very easily implemented by maintaining a queue (FIFO) of nodes. Initially the queue contains just the root. In each iteration, node at the head of the queue is removed and then expanded. The generated child nodes are then added to the tail of the queue.

<u>Example 1:</u> Consider the following State Space Graph:

Implementation: fringe is a first-in-first-out (FIFO) queue, i.e., new successors goes at end of the queue.



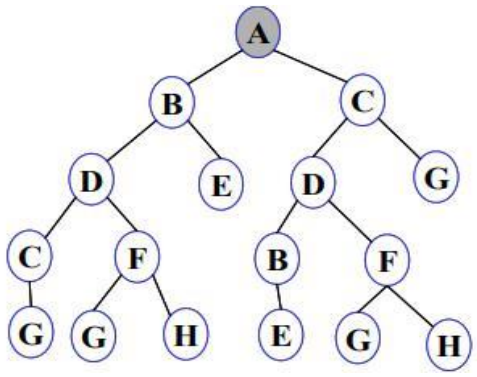


Current	Waiting
A	---
A	B, C
B	C
B	C, D
C	D
C	D, E, F
D	E, F
D	E, F, G, H
E	F, G, H
E	F, G, H, I, J
F	G, H, I, J
F	G, H, I, J, K, L
G	H, I, J, K, L
H	I, J, K, L
I	J, K, L
J	K, L

GOAL

Example 2: Consider the following State Space Graph:

- Fringe (NODE-LIST) contains only one node i.e. Root Node **A**



Is **A** a GOAL state? **NO**

Expand:
fringe = [B,C]
Is B a GOAL state?

FRINGE
~~A~~
B
C

Output:

A						

FRINGE
~~A~~
~~B~~
C

Is **B** a GOAL state? **NO**
Expand:
Fringe = [C,D,E]
Is **C** a GOAL state? **NO**

Output:

A	B					

In the same way, finally we do have:

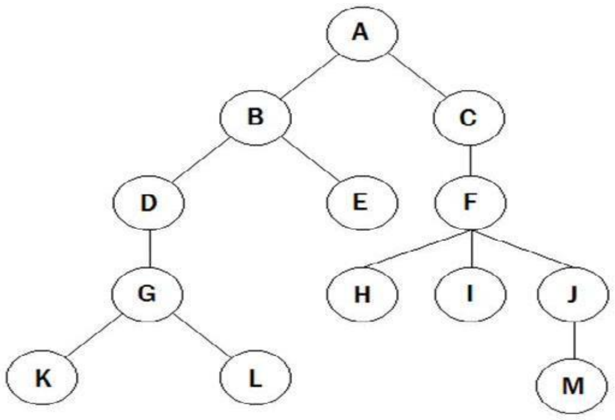
FRINGE
~~A~~
~~B~~
~~C~~
~~D~~
~~E~~
~~D~~
~~G~~

A	B	C	D	E	D	G

Goal State

BFS traverse is ABCDEDG (Explore nodes). As **G** is the GOAL node.

Exercise:



Algorithm of Breadth First Search (BFS)

1. Create a variable called NODE-LIST and set it to the initial state.
2. Loop until the goal state is found or NODE-LIST is empty.
   (a) Remove the first element, say E, from the NODE-LIST. If NODE-LIST was empty then quit.
   (b) For each way that each rule can match the state described in E do:
      i. Apply the rule to generate a new state.
      ii. If the new state is the goal state, quit and return this state.
      iii. Otherwise add this state to the end of NODE-LIST

Advantages of breadth-first search

1. Finds the path of minimal length to the goal - If there is more than one solution then BFS can find the minimal one that requires less number of steps.
2. Breadth first search will never get trapped exploring the useless path forever.
3. If there is a solution, BFS will definitely find it out.

Disadvantages of breadth-first search

1. The main drawback of Breadth first search is its memory requirement. Since each level of the tree must be saved in order to generate the next level, and the amount of memory is proportional to the number of nodes stored, the space complexity of BFS is $O(b^d)$. As a result, BFS is severely space-bound in practice so will exhaust the memory available on typical computers in a matter of minutes.
2. If the solution is farther away from the root, breath first search will consume lot of time.

Performance Measure:

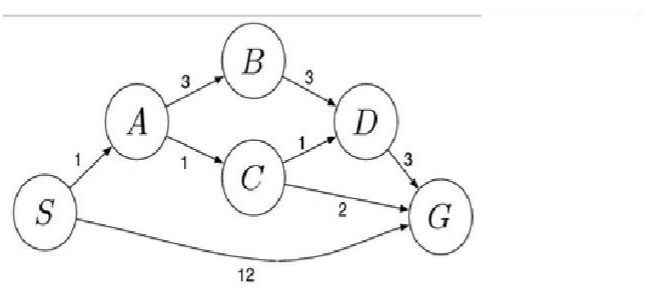
- Completeness:
  - it is easy to see that breadth-first search is complete that it visit all levels given that d factor is finite, so in some d it will find a solution.
- Optimality:
  - breadth-first search is not optimal until all actions have the same cost (because it always finds the shallowest node first)
- Space complexity and Time complexity:
  - Consider a state space where each node as a branching factor b, the root of the tree generates b nodes, each of which generates b nodes yielding b^2 each of these generates b^3 and so on.
  - In the worst case, suppose that our solution is at depth d, and we expand all nodes but the last node at level d, then the total number of generated nodes is: $b + b^2 + b^3 + b^4 + b^{d+1} - b = O(b^{d+1})$, which is the time complexity of BFS.
  - As all the nodes must retain in memory while we expand our search, then the space complexity is like the time complexity plus the root node = $O(b^{d+1})$.

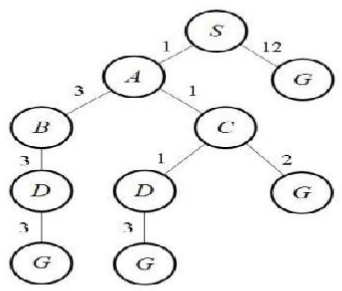
Link: https://sandipanweb.wordpress.com/2017/03/16/using-uninformed-informed-search-algorithms-to-solve-8-puzzle-n-puzzle/

<u>Uniform cost search (UCS):</u>

– Uniform-cost search expands nodes in order of their cost from the root. Uniform-cost is guided by path cost rather than path length like in BFS

- The algorithms starts by expanding the root, then expanding the node with the lowest cost from the root, the search continues in this manner for all nodes. The nodes are stored in a priority queue.
- This algorithm is also known as Dijkstra's single-source shortest algorithm.
- Uniform Cost Search is the best algorithm for a search problem, which does not involve the use of heuristics.
- It can solve any general graph for optimal cost.
- Uniform Cost Search as it sounds searches in branches which are more or less the same in cost.
- Uniform Cost Search can also be used as Breadth First Search if all the edges are given a cost of 1.
- The worst case time complexity of uniform-cost search is $O(b^{c/m})$, where c is the cost of an optimal solution and m is the minimum edge cost.





- Initialization: { [S , 0] }
- Iteration 1: { [S->A , 1] , [S->G , 12] }
- Iteration 2: { [S->A->C , 2] , [S->A->B , 4] , [S->G , 12] }
- Iteration 3: { [S->A->C->D , 3] , [S->A->B , 4] , [S->A->C->G , 4] , [S->G , 12] }
- Iteration 4: { [S->A->B , 4] , [S->A->C->G , 4] , [S->A->C->D->G , 6] , [S->G , 12] }
- Iteration 5: { [S->A->C->G , 4] , [S->A->C->D->G , 6] , [S->A->B->D , 7] , [S->G , 12] }
- Iteration 6: Gives the final output as S->A->C->G.

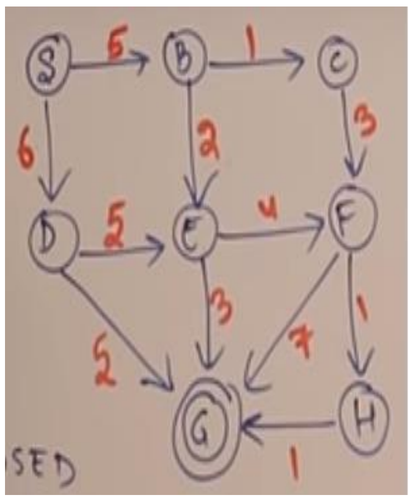
Final Path is **SACG**

Example 2:

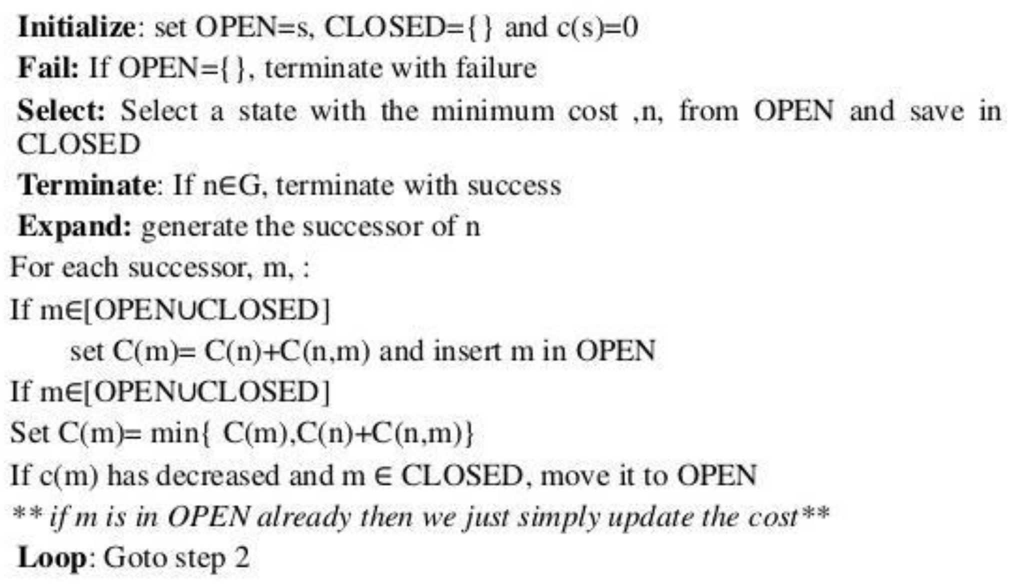
CLOSE	OPEN
	S(0)
S(0)	D(6) B(5)
S(0) B(5)	D(6) C(6) E(7)
S(0) B(5) D(6)	C(6) E(7) G(11)
S(0) B(5) D(6) C(6)	E(7) G(11) F(9)
S(0) B(5) D(6) C(6) E(7)	~~G(11)~~ F(9) G(10)
S(0) B(5) D(6) C(6) E(7) F(9)	G(10) H(10)
S(0) B(5) D(6) C(6) E(7) F(9) G(10)	

← Goal State is present in OPEN, but not having max. cost, so continue

← min cost of G should be selected here



Algorithm for UCS:



```
Initialize: set OPEN=s, CLOSED={} and c(s)=0
Fail: If OPEN={}, terminate with failure
Select: Select a state with the minimum cost ,n, from OPEN and save in
CLOSED
Terminate: If n∈G, terminate with success
Expand: generate the successor of n
For each successor, m, :
If m∈[OPEN∪CLOSED]
      set C(m)= C(n)+C(n,m) and insert m in OPEN
If m∈[OPEN∪CLOSED]
Set C(m)= min{ C(m),C(n)+C(n,m)}
If c(m) has decreased and m ∈ CLOSED, move it to OPEN
** if m is in OPEN already then we just simply update the cost**
Loop: Goto step 2
```

Advantage:

1. Guaranteed to find the least-cost solution.

Disadvantages:

1. Exponential storage required.
2. open list must be kept sorted (as a priority queue).
3. Must change cost of a node on open if a lower-cost path to it is found.

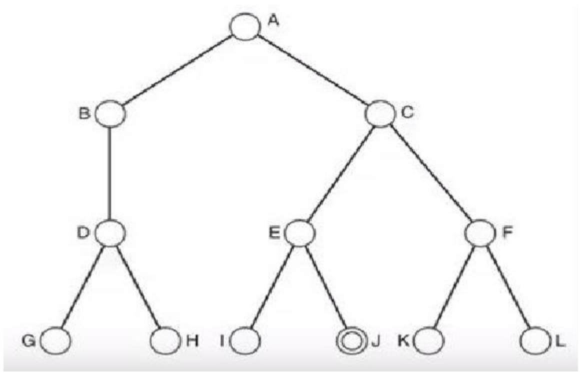
Note: Uniform-cost search is the same as Heuristic Search when no heuristic information is available (heuristic function h is always 0).

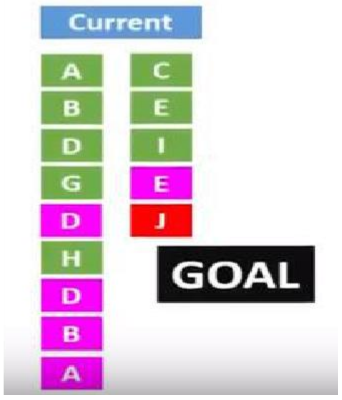
Performance Measure:

- Completeness:
  - It is obvious that UCS is complete if the cost of each step exceeds some small positive integer, this to prevent infinite loops.
- Optimality:
  - UCS is always optimal in the sense that the node that it always expands is the node with the least path cost.
- Time Complexity:
  - UCS is guided by path cost rather than path length so it is hard to determine its complexity in terms of b and d, so if we consider C to be the cost of the optimal solution, and every action costs at least e, then the algorithm worst case is $O(b^{C/e})$.
- Space Complexity:
  - The space complexity is $O(b^{C/e})$ as the time complexity of UCS.

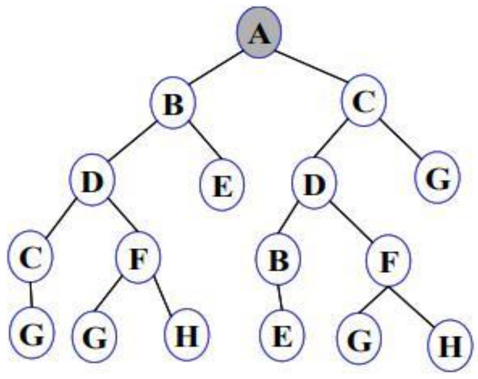
Depth First Search

- DFS progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hasn't finished exploring. DFS investigated in 19th century by French Mathemetician Charles Pierre as a strategy for solving mazes.
- DFS is implemented using STACK (Last In First Out)
- For example: Consider the state graph, start node is **A** and goal node is **J**





Exercise: Use a depth-first search to find **G** in the following search tree:

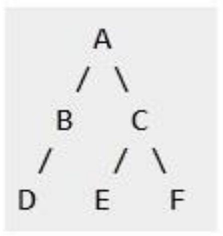
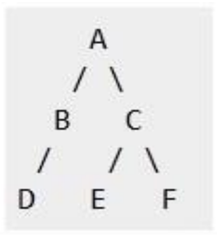


Performance Measure:

- Completeness:
    - DFS is not complete, to convince yourself consider that our search start expanding the left sub tree of the root for so long path (may be infinite) when different choice near the root could lead to a solution, now suppose that the left sub tree of the root has no solution, and it is unbounded, then the search will continue going deep infinitely, in this case we say that DFS is not complete.
- Optimality:
    - Consider the scenario that there is more than one goal node, and our search decided to first expand the left sub tree of the root where there is a solution at a very deep level of this left sub tree, in the same time the right sub tree of the root has a solution near the root, here comes the non-optimality of DFS that it is not guaranteed that the first goal to find is the optimal one, so we conclude that DFS is not optimal.
- Time Complexity:
    - Consider a state space that is identical to that of BFS, with branching factor b, and we start the search from the root.

- In the worst case that goal will be in the shallowest level in the search tree resulting in generating all tree nodes which are *O*(b^m).

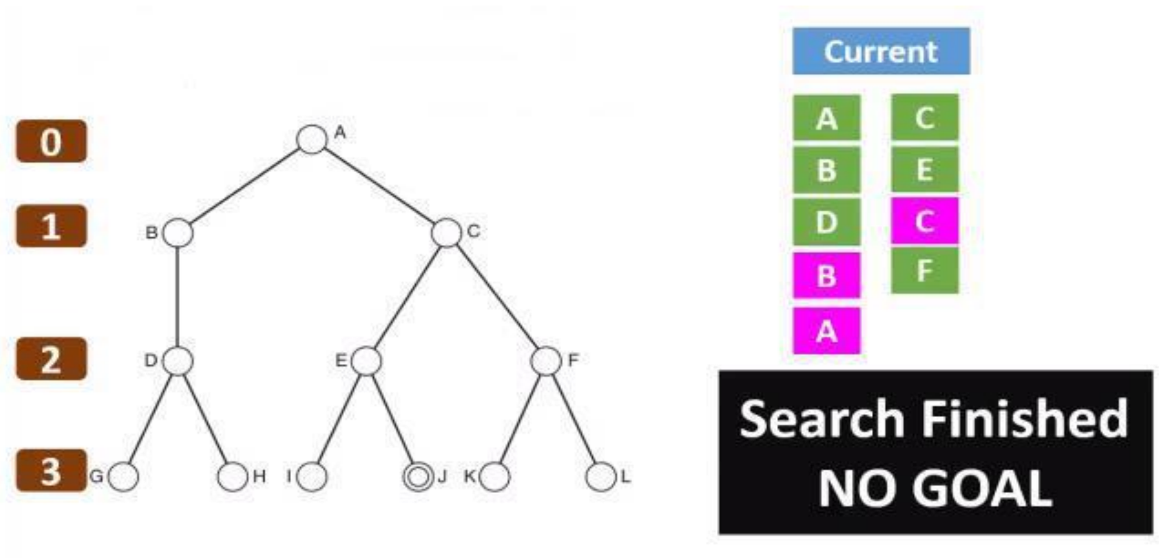
- Space Complexity:
  - Unlike BFS, our DFS has a very modest memory requirements, it needs to story only the path from the root to the leaf node, beside the siblings of each node on the path, remember that BFS needs to store all the explored nodes in memory.
  - DFS removes a node from memory once all of its descendants have been expanded.
  - With branching factor *b* and maximum depth *m,* DFS requires storage of only bm + 1 nodes which are *O*(bm) compared to the *O*(b^{d+1}) of the BFS.

BFS	**DFS**
BFS Stands for "**Breadth First Search**".	**DFS** stands for "**Depth First Search**".
BFS starts traversal from the root node and then explore the search in the level by level manner i.e. as close as possible from the root node.	DFS starts the traversal from the root node and explore the search as far as possible from the root node i.e. depth wise.
Breadth First Search can be done with the help of **queue** i.e. **FIFO** implementation.	Depth First Search can be done with the help of **Stack** i.e. **LIFO** implementations.
This algorithm works in single stage. The visited vertices are removed from the queue and then displayed at once.	This algorithm works in two stages – in the first stage the visited vertices are pushed onto the stack and later on when there is no vertex further to visit those are popped-off.
BFS is **slower** than DFS.	DFS is faster than BFS.
BFS requires **more** memory compare to DFS.	DFS require **less** memory compare to BFS.
Applications of BFS > To find Shortest path > Single Source & All pairs shortest paths > In Spanning tree > In Connectivity	**Applications of DFS** > Useful in Cycle detection > In Connectivity testing > Finding a path between V and W in the graph. > useful in finding spanning trees & forest.
BFS is useful in finding shortest path.BFS can be used to find the shortest distance between some starting node and the remaining nodes of the graph.	DFS in not so useful in finding shortest path. It is used to perform a traversal of a general graph and the idea of DFS is to make a path as long as possible, and then go back (**backtrack**) to add branches also as long as possible.
Example :  A, B, C, D, E, F	**Example :**  A, B, D, C, E, F

Depth-Limited Search:

The unbounded tree problem appeared in DFS can be fixed by imposing a limit on the depth that DFS can reach, this limit we will call depth limit l, this solves the infinite path problem.

Example: Depth = 2, Goal Node = J



Performance Measure:

- Completeness:
  - The limited path introduces another problem which is the case when we choose l < d, in which is our DLS will never reach a goal, in this case we can say that DLS is not complete.
- Optimality:
  - One can view DFS as a special case of the depth DLS, that DFS is DLS with l = infinity.
  - DLS is not optimal even if l > d.
- Time Complexity: $O(b^l)$
- Space Complexity: $O(b^l)$

Depth-First Iterative Deepening Search:

It is a search strategy resulting when you combine BFS and DFS, thus combining the advantages of each strategy, taking the completeness and optimality of BFS and the modest memory requirements of DFS.

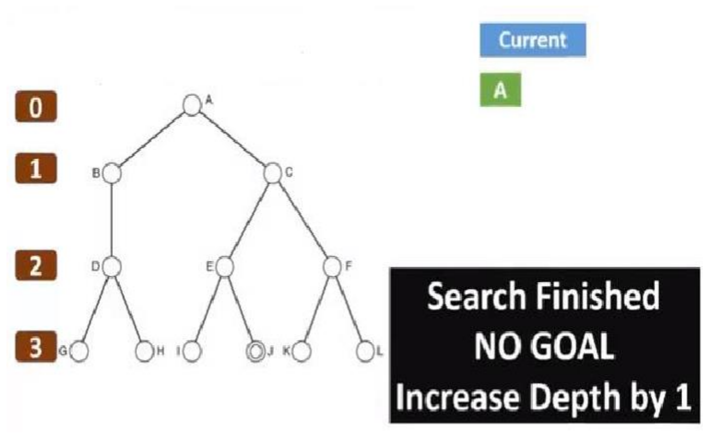
IDS works by looking for the best search depth d, thus starting with depth limit 0 and make a BFS and if the search failed it increase the depth limit by 1 and try a BFS again with depth 1 and so on – first d = 0, then 1 then 2 and so on – until a depth d is reached where a goal is found.

- is complete for finite b
- is optimal in terms of the solution depth (and optimal in general if path cost is non-decreasing function of depth)
- has asymptotic time complexity $O(b^d)$ in the worst case, ironically better than that of plain depth-first search!
- has very modest memory requirements $O(b^d)$ if expansion-based or O(d) if backtracking-based.
- generates fewer nodes than the version of BFS that checks for goals at expansion time
- is **asymptotically optimal in terms of time and space among brute-force tree searches that find optimal solutions!**

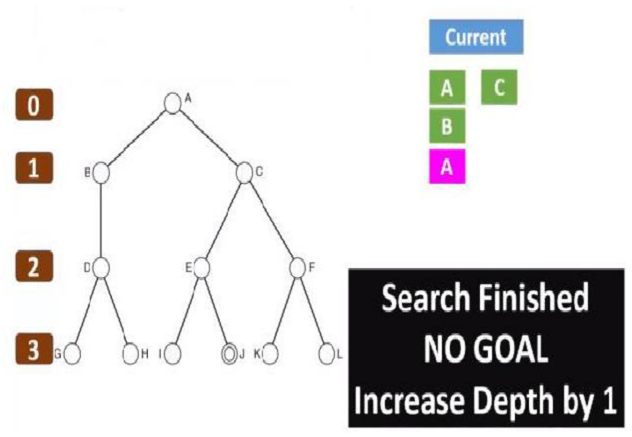
***Exercise**: Show that the number of nodes generated by DFID is less than* $b^d(\frac{1}{(1-\frac{1}{b})^2})$.

Example:

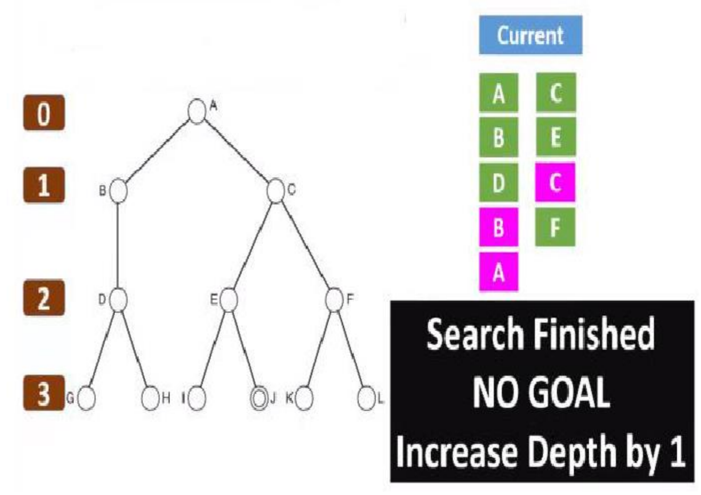
Iteration 1: Depth = 0



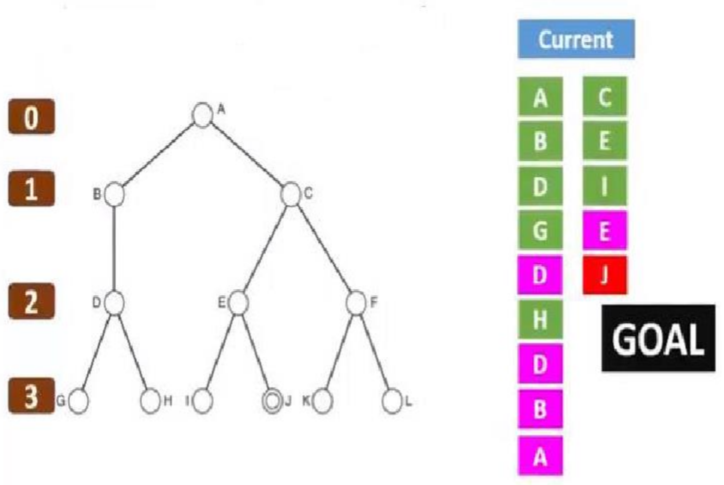
Iteration 2: Depth = 1



Iteration 3: Depth = 2



Iteration 4: Depth = 3



Performance Measure:

- Completeness:
  - IDS is like BFS, is complete when the branching factor b is finite.
- Optimality:
  - IDS is also like BFS optimal when the steps are of the same cost.
- Time Complexity:
  - One may find that it is wasteful to generate nodes multiple times, but actually it is not that costly compared to BFS, that is because most of the generated nodes are always in the deepest level reached, consider that we are searching a binary tree and our depth limit reached 4, the nodes generated in last level = 2^4 = 16, the nodes generated in all nodes before last level = $2^0 + 2^1 + 2^2 + 2^3 = 15$
  - Imagine this scenario, we are performing IDS and the depth limit reached depth d, now if you remember the way IDS expands nodes, you can see that nodes at depth d are generated once, nodes at depth d-1 are generated 2 times, nodes at depth d-2 are generated 3 times and so on, until you reach depth 1 which is generated d times, we can view the total number of generated nodes in the worst case as:
    - $N(IDS) = (b)d + (d-1)b^2 + (d-2)b^3 + ... + (2)b^{d-1} + (1)b^d = O(b^d)$

- If this search were to be done with BFS, the total number of generated nodes in the worst case will be like:
  - $N(BFS) = b + b^2 + b^3 + b^4 + \ldots.\ b^d + (b^{d+1} - b) = O(b^{d+1})$
- If we consider a realistic numbers, and use b = 10 and d = 5, then number of generated nodes in BFS and IDS will be like
  - N(IDS) = 50 + 400 + 3000 + 20000 + 100000 = 123450
  - N(BFS) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100
  - BFS generates like 9 time nodes to those generated with IDS.

- Space Complexity:
  - IDS is like DFS in its space complexity, taking *O*(bd) of memory.

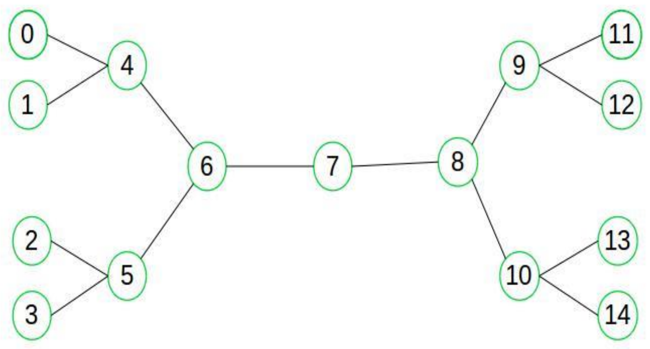
Bi-directional Search

– Bidirectional Search, as the name implies, searches in two directions at the same time: one forward from the initial state and the other backward from the goal

– Suppose if branching factor of tree is b and distance of goal vertex from source is d, then the normal BFS/DFS searching complexity would be O(b^d). On the other hand, if we execute two search operation then the complexity would be O(b^{d/2}) for each search and total complexity would be O(b^{d/2}+b^{d/2}) which is far less than O(b^d).

– We can consider bidirectional approach when-
  1. Both initial and goal states are unique and completely defined.
  2. The branching factor is exactly the same in both directions.

Consider the following simple example:



Suppose we want to find if there exists a path from vertex 0 to vertex 14. Here we can execute two searches, one from vertex 0 and other from vertex 14. When both forward and backward search meet at vertex 7, we know that we have found a path from node 0 to 14 and search can be terminated now. We can clearly see that we have successfully avoided unnecessary exploration.

ADVANTAGES

1. The merit of bidirectional search is its speed. Sum of the time taken by two searches (forward and backward) is much less than the $O(b^d)$ complexity.
2. It requires less memory.

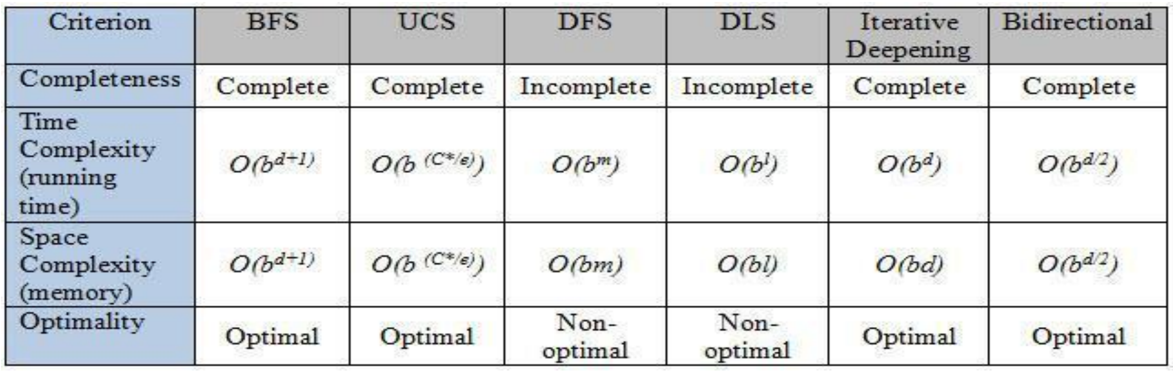
DISADVANTAGES

1. Implementation of bidirectional search algorithm is difficult because additional logic must be included to decide which search tree to extend at each step.
2. One should have known the goal state in advance.
3. The algorithm must be too efficient to find the intersection of the two search trees.
4. It is not always possible to search backward through possible states.

Performance Measure:

- Completeness:
    - Bidirectional search is complete when we use BFS in both searches, the search that starts from the initial state and the other from the goal state.
- Optimality:
    - Like the completeness, bidirectional search is optimal when BFS is used and paths are of a uniform cost – all steps of the same cost.
    - Other search strategies can be used like DFS, but this will sacrifice the optimality and completeness, any other combination than BFS may lead to a sacrifice in optimality or completeness or may be both of them.
- Time and Space Complexity:
    - May be the most attractive thing in bidirectional search is its performance, because both searches will run the same amount of time meeting in the middle of the graph, thus each search expands $O(b^{d/2})$ node, in total both searches expand $O(b^{d/2} + b^{d/2})$ node which is too far better than the $O(b^{d+1})$ of BFS.
    - If a problem with b = 10, has a solution at depth d = 6, and each direction runs with BFS, then at the worst case they meet at depth d = 3, yielding 22200 nodes compared with 11111100 for a standard BFS.
    - We can say that the time and space complexity of bidirectional search is $O(b^{d/2})$.

Comparision of Uninformed Search



Criterion	BFS	UCS	DFS	DLS	Iterative Deepening	Bidirectional
Completeness	Complete	Complete	Incomplete	Incomplete	Complete	Complete
Time Complexity (running time)	$O(b^{d+1})$	$O(b^{(C^*/\epsilon)})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space Complexity (memory)	$O(b^{d+1})$	$O(b^{(C^*/\epsilon)})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimality	Optimal	Optimal	Non-optimal	Non-optimal	Optimal	Optimal

- b = Branching factor
- d = Depth of the shallowest solution
- m = Maximum depth of the search tree
- l = Depth Limit

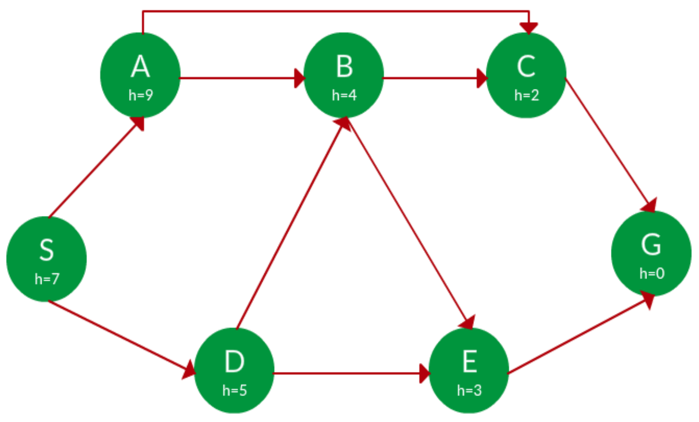
Heuristic/ Informed/ Guided Search

Informed search algorithms use domain knowledge. In an informed search, problem information is available which can guide the search. Informed search strategies can find a solution more efficiently than an uninformed search strategy. Informed search is also called a Heuristic search. A heuristic is a way which might not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.

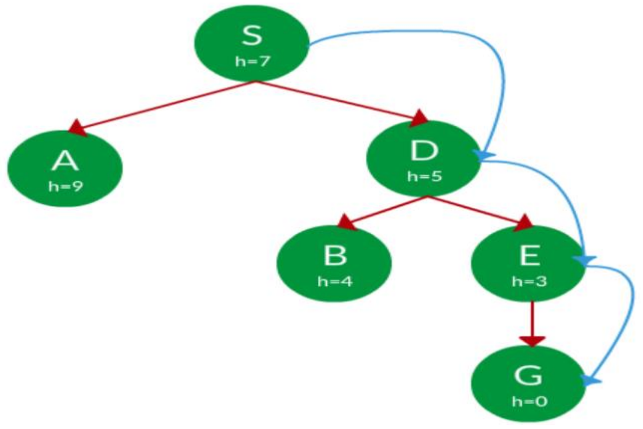
To solve large problems with large number of possible states, problem-specific knowledge needs to be added to increase the efficiency of search algorithms. In an informed search, a heuristic is a function that estimates how close a state is to the goal state. For examples – Manhattan distance, Euclidean distance, etc. (Lesser the distance, closer the goal). In greedy search, we expand the node closest to the goal node. The "closeness" is estimated by a heuristic h(x).

Heuristic: A heuristic h is defined as-
h(x) = Estimate of distance of node x from the goal node.
Lower the value of h(x), closer is the node from the goal.

Strategy: Expand the node closest to the goal state, *i.e.* expand the node with lower h value.
Example: Find the path from S to G using greedy search. The heuristic values h of each node below the name of the node.

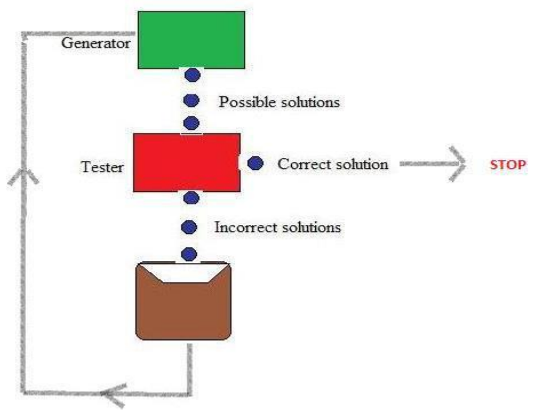


Solution. Starting from S, we can traverse to A(h=9) or D(h=5). We choose D, as it has the lower heuristic cost. Now from D, we can move to B(h=4) or E(h=3). We choose E with lower heuristic cost. Finally, from E, we go to G(h=0). This entire traversal is shown in the search tree below, in blue



Generate and Test Search:

- Generate and Test search guarantee to find a solution if done systematically and there exist a solution. It is the simplest heuristic search technique which used DFS with backtracking.



Algorithm:

- Step 1: Generate a possible solution
- Step 2: Test & see if this is the expected solution
- Step 3: If the solution has been found QUIT else GOTO step 1.

Properties:

- Completeness
- Non-redundant
- Informed search

Hill Climbing Search:

- The name hill climbing is derived from simulating the situation of a person climbing the hill. The person will try to move forward in the direction of at the top of the hill. His movement stops when it reaches at the peak of hill and no peak has higher value of heuristic function than this.
- 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 neighbour has a higher value. This category of application includes job-shop scheduling, vehicle routing etc.
- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance travelled by the salesman.
- It is also called greedy local search as it only looks to its good immediate neighbour 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.

Algorithm

- Define the current state as an initial state
- Apply any possible operation on the current state and generate a possible solution
- Compare the newly generated solution with the goal state
- If the goal is achieved or no new states can be created, quit. Otherwise, return to step 2

Main features of Hill Climbing Algorithm:

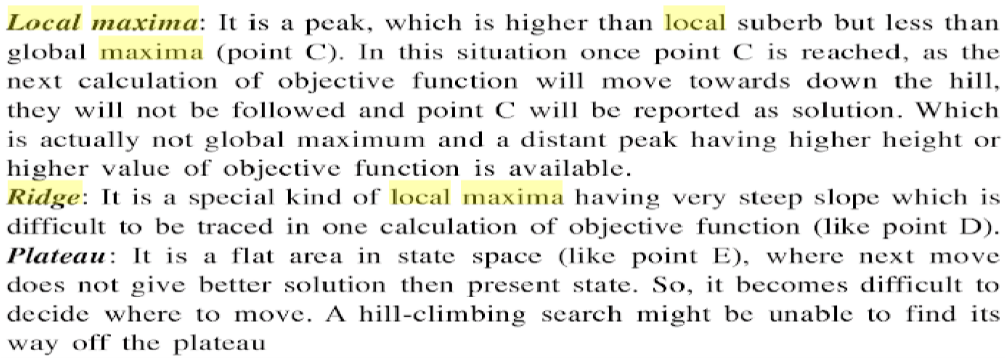
- Generate and Test variant: Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- Greedy approach: Hill-climbing algorithm search moves in the direction which optimizes the cost.
-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

Advantages:

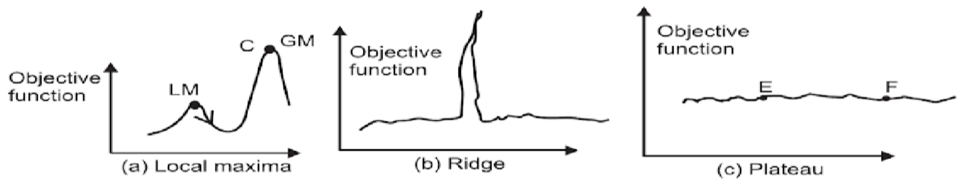
- Useful for AI problems where knowledge of the path is not important, so in obtaining the problem solution, it is not recorded.
- It is also helpful to solve pure optimization problems where the objective is to find the best state according to the objective function.

Disadvantages:

This technique works but as it uses local information that's why it can be fooled. The algorithm doesn't maintain a search tree, so the current node data structure need only record the state and its objective function value. It assumes that local improvement will lead to global improvement.



Local maxima: It is a peak, which is higher than local suberb but less than global maxima (point C). In this situation once point C is reached, as the next calculation of objective function will move towards down the hill, they will not be followed and point C will be reported as solution. Which is actually not global maximum and a distant peak having higher height or higher value of objective function is available.
Ridge: It is a special kind of local maxima having very steep slope which is difficult to be traced in one calculation of objective function (like point D).
Plateau: It is a flat area in state space (like point E), where next move does not give better solution then present state. So, it becomes difficult to decide where to move. A hill-climbing search might be unable to find its way off the plateau



Limitations of hill climbing

1. Local maximum : At a local maximum all neighboring states have a values which is worse than the current state. Since hill-climbing uses a greedy approach, it will not move to the worse state and terminate itself. The process will end even though a better solution may exist.
   To overcome local maximum problem : Utilize backtracking technique. Maintain a list of visited states. If the search reaches an undesirable state, it can backtrack to the previous configuration and explore a new path.
2. Plateau : On plateau all neighbors have same value . Hence, it is not possible to select the best direction.
   To overcome plateaus : Make a big jump. Randomly select a state far away from the current state. Chances are that we will land at a non-plateau region.

3. Ridge : Any point on a ridge can look like peak because movement in all possible directions is downward. Hence the algorithm stops when it reaches this state.
   To overcome Ridge : In this kind of obstacle, use two or more rules before testing. It implies moving in several directions at once.

There are few solutions to overcome these situations:

1. We can **backtrack** to one of the previous states and explore other directions
2. We can skip few states and make a **jump** in new directions
3. We can **explore several directions** to figure out the correct path

Types of Hill Climbing Algorithm:

- Simple hill Climbing:
- Steepest-Ascent hill-climbing
- Simulated Annealing hill-climbing

Simple Hill climbing algorithm:

- Simple hill climbing is the simplest way to implement a hill climbing algorithm.
- It only evaluates the neighbour node state at a time and selects the first one which optimizes current cost and set it as a current state.
- It only checks it's 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 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:
    - If it is goal state, then return success and quit.
- Else if it is better than the current state then assigns new state as a current state.
- Else if not better than the current state, then return to step2.
- Step 5: Exit.

Steepest-Ascent Hill-Climbing:

Steepest-Ascent Hill-Climbing algorithm (gradient search) is a variant of Hill Climbing algorithm. A useful variation on simple hill climbing considers all the moves from the current state and selects the best one as the next state. The method is called steepest-ascent hill climbing or gradient search.

In other words, in the case of hill climbing technique we picked any state as a successor which was closer to the goal than the current state whereas, in Steepest-Ascent Hill Climbing algorithm, we choose the best successor among all possible successors and then update the current state.

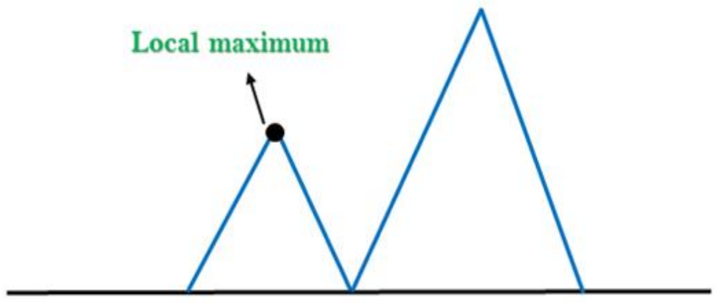
Algorithm for Steepest-Ascent hill climbing:

- Step 1: Evaluate the initial state, if it is goal state then return success and stop, else make current state as 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 goal state, then return it and quit, else compare it to the SUCC.
        d. If it is better than SUCC, then set new state as SUCC.
        e. If the SUCC is better than the current state, then set current state to SUCC.
- Step 5: Exit.

Problems in Hill Climbing Algorithm:

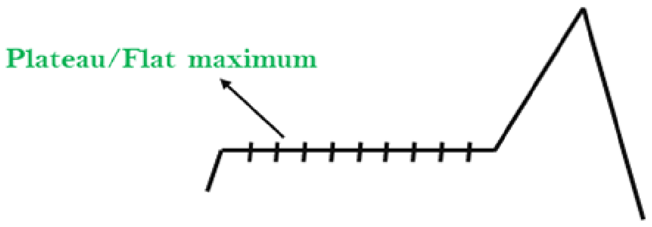
Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighbouring states, but there is another state also present which is higher than the local maximum.

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.



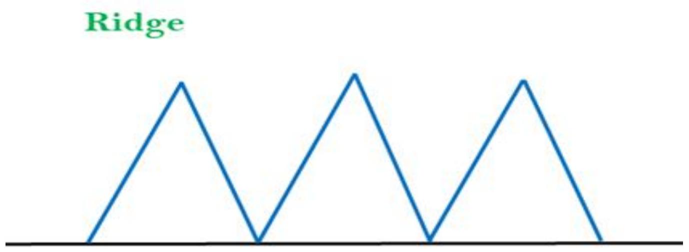
Plateau: A plateau is the flat area of the search space in which all the neighbour states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

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 non-plateau region.



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 Hill Climbing:

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

In mechanical term, **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.

Note: SA is a combination of simple hill climbing and random walk.

<u>Advantages of Simulated Annealing Hill Climbing</u>:

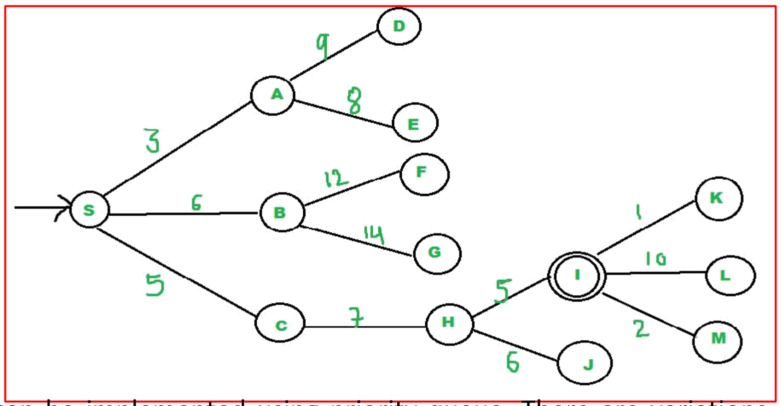
- Easy to code for complex problems
- Gives "good" solution
- Guarantee to find optimal solution

<u>Dis-advantages of Simulated Annealing Hill Climbing</u>:

- Takes a long time to find near optimal solution, but still FAST compared to algorithms
- SA doesn't have any memory to keep records of previously visited solution, hence there will always be a possibility for the search to return to such a solution again.

Best-First Search

- In the Best-First Search, "best-first" refers to the method of exploring the node with the best "score" first.
- Search start from root node, node to be expanded next is selected on the basis of evaluation function f(n).
- An evaluation function is used to assign a score to each candidate node. Node having lowest value of f(n) is selected first as it indicates that the goal is nearest from this node.
- It is implemented using priority queue, highest priority is given to the node have least f(n) value.
- In the graph of problem representation, one evaluation function (which corresponds to heuristic function) is attached with every node. The value of evaluation function may depend upon cost or distance of current node from goal node. The decision of which node to be expanded depends on the value of this evaluation function. The best first can be explained from the following tree: Example 1:

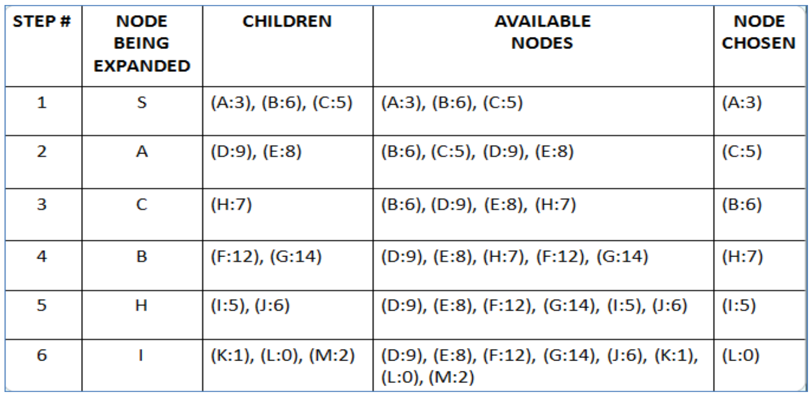


The best-first search can be implemented using priority queue. There are variations of best-first search. Example: Greedy best search, A* and recursive best-first search.

Two lists of nodes are used to implement a graph search procedure:

- OPEN: these are the nodes that have been generated and have had the heuristic function applied to them but have not been examined yet.
- CLOSED: these are the nodes that have already been examined. These nodes are kept in the memory if we want to search a graph rather than a tree because a new node will be generated, we will have to check whether it has been generated earlier.

From the above, we start from source "S" and search for goal "I" using given costs.



STEP #	NODE BEING EXPANDED	CHILDREN	AVAILABLE NODES	NODE CHOSEN
1	S	(A:3), (B:6), (C:5)	(A:3), (B:6), (C:5)	(A:3)
2	A	(D:9), (E:8)	(B:6), (C:5), (D:9), (E:8)	(C:5)
3	C	(H:7)	(B:6), (D:9), (E:8), (H:7)	(B:6)
4	B	(F:12), (G:14)	(D:9), (E:8), (H:7), (F:12), (G:14)	(H:7)
5	H	(I:5), (J:6)	(D:9), (E:8), (F:12), (G:14), (I:5), (J:6)	(I:5)
6	I	(K:1), (L:0), (M:2)	(D:9), (E:8), (F:12), (G:14), (J:6), (K:1), (L:0), (M:2)	(L:0)

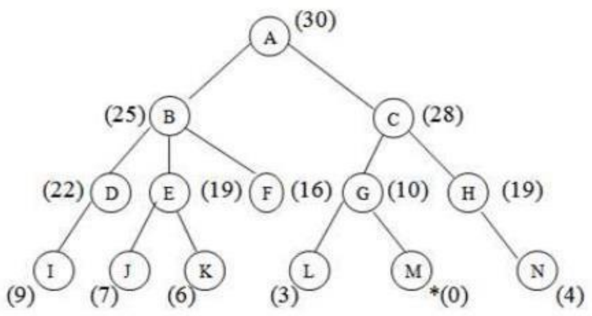
Algorithm:

1. Put the initial node on a list, say OPEN.
2. If (OPEN=EMPTY or OPEN=GOAL) terminate search, else
3. Remove the first node from OPEN (say node a)
4. If (a=GOAL) terminate search with success, else
5. Generate all the successors of node ‘a’. Send node ‘a’ to a list called ‘CLOSED’. Find out the value of heuristic function of all nodes. Sort all the children generated so far on the basis of their utility value. Select the node of minimum heuristic value for further expansion.
6. GO back to step 2.

Note: The difference is in heuristic function. Uniform Cost Search is uninformed search. It doesn't use any domain knowledge. It expands the least cost node and it does so in every direction because no information about the goal is provided.

Best first search is informed search. It uses the heuristic function to estimate how close to goal the current state is (are we getting close to the goal?). Yes, both methods have a list of expanded nodes but best-first search will try to minimize that number of expanded nodes (Path Cost + Heuristic Function). An example of best-first search is e.g. A* algorithm.

Example 2:



Consider the node A as our root node. So the first element of the queue is A whish is not our goal node, so remove it from the queue and find its neighbor that are to inserted in ascending order.

PRIORITY QUEUE (PQ)	A

REMOVE A FROM PQ

PRIORITY QUEUE (PQ)	B(25)	C(28)

REMOVE B FROM PQ

PRIORITY QUEUE (PQ)	F(16)	E(19)	D(22)	C(28)

REMOVE F FROM PQ, NO FURTHER CHILD NODES

REMOVE E FROM PQ

PRIORITY QUEUE (PQ)	K(6)	J(7)	D(22)	C(28)

REMOVE K FROM PQ, NO FURTHER CHILD NODES

REMOVE J FROM PQ, NO FURTHER CHILD NODES

REMOVE D FROM PQ

PRIORITY QUEUE (PQ)	I(9)	C(28)

REMOVE I FROM PQ, NO FURTHER CHILD NODES

REMOVE C FROM PQ

PRIORITY QUEUE (PQ)	G(10)	H(19)

REMOVE G FROM PQ

PRIORITY QUEUE (PQ)	M(0)	L(3)	H(19)

Now, M is in FRONT node of queue and it is our goal state, so return SUCCESS.

Advantage:

- It is more efficient than that of BFS and DFS.
- Time complexity of Best first search is much less than Breadth first search.
- The Best first search allows us to switch between paths by gaining the benefits of both breadth first and depth first search. Because, depth first is good because a solution can be found without computing all nodes and Breadth first search is good because it does not get trapped in dead ends.

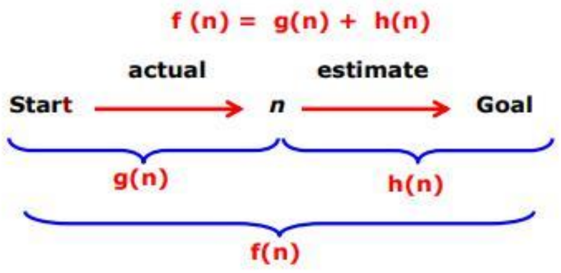
Disadvantages:

Sometimes, it covers more distance than our consideration.

A* Algorithm

- A* Algorithm is the specialization of Best First Search in which the cost associated with a node is f(n) = g(n) + h(n), where g(n) is the cost of the path from the initial state to node n and h(n) is the heuristic estimate or the cost or a path from node n to a goal. Thus, f(n) estimates the lowest total cost of any solution path going through node n. At each point, a node with lowest f value is chosen for expansion.

- A* algorithm guides an optimal path to a goal if the heuristic function h(n) is admissible, meaning it never overestimates actual cost
- The * represents that the algorithm is admissible as it guarantees to give optimal solution.



Where, f(n) = evaluation function

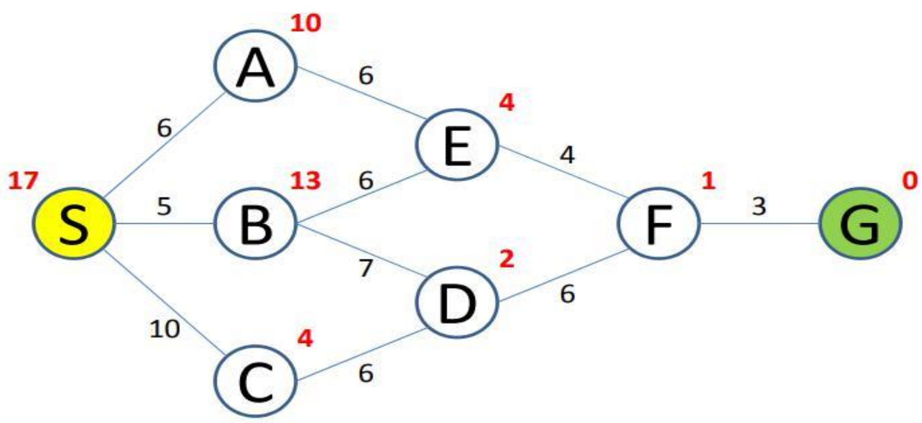
g(n) = cost (or distance) of current node from start node

h(n) = cost of current node from goal nodeA heuristics h is an admissible heuristics if the following condition holds for every state a:

$$h(a) \leq h^*(a).$$

A* algorithm:

1. Place the starting node's' on 'OPEN' list.
2. If 'OPEN' is empty, stop and return failure.
3. Remove from OPEN the node 'n' that has the smallest value of f* (n). If node 'n' is goal node, return success and stop otherwise.
4. Expand 'n' generating all of its successors 'n' and place 'n' on CLOSED. For every successor 'n' if 'n' is not already OPEN, attach a back pointer to 'n'. Compute f*(n) and place it on CLOSED.
5. Each 'n' that is already on OPEN or CLOSED should be attached to back pointers which reflect the lowest f*(n) path. If 'n' was on CLOSED and its pointer was changed, remove it and place it on OPEN.
6. Return to step 2.



Example:

To describe this accurately, we have to introduce the following notations for every vertex n of the search tree:

- $h^*(n)$: the *optimal* cost of getting to a goal vertex from n .
- $g^*(n)$: the *optimal* cost of getting to n from the initial vertex.
- $f^*(n)=g^*(n)+h^*(n)$: Consequently, the optimal cost of getting to a goal vertex from the initial vertex via n .

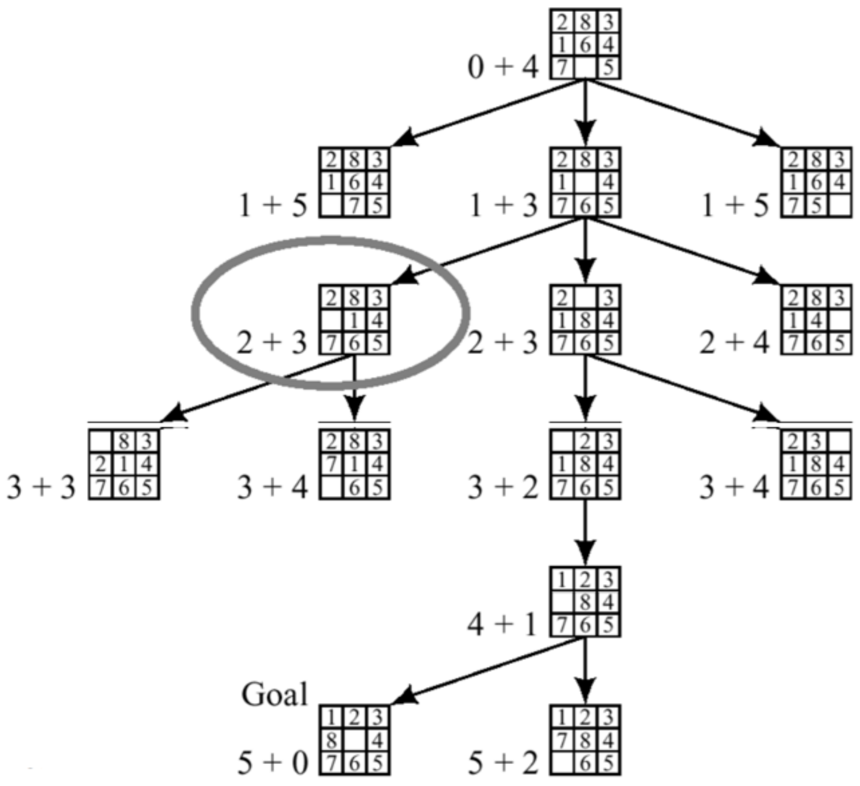
Performance Measure:

- Completeness:
  - If there is a solution, the method finds it in any state-space graph.
  - If there is no solution, the method recognizes this fact in the case of a finite state-space graph.
- Optimality: generating the optimal solution is guaranteed.

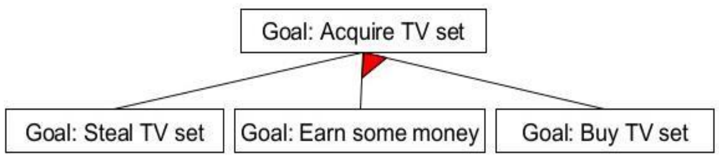
Optimality of A* Algorithm

Provided that h(n) never overestimates the cost to reach the goal, then in tree search A* gives the optimal solution

- Suppose G2 is a suboptimal goal node generated to the tree
- Let C* be the cost of the optimal solution
- Because G2 is a goal node, it holds that h(G2) = 0, and we know that f(G2) = g(G2) > C*
- On the other hand, if a solution exists, there must exist a node n that is on the optimal solution path in the tree
- Because h(n) does not overestimate the cost of completing the solution path, f(n) = g(n) + h(n) ≤ C*
- We have shown that f(n) ≤ C* < f(G2), so G2 will not be expanded and A* must return an optimal solution



Problem Reduction Search



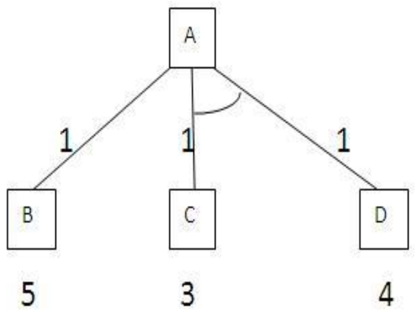
AND-OR Graphs

If we are looking for a sequence of actions to achieve some goal, then one way to do it is to use state-space search, where each node in your search space is a state of the world, and you are searching for a sequence of actions that get you from an initial state to a final state.

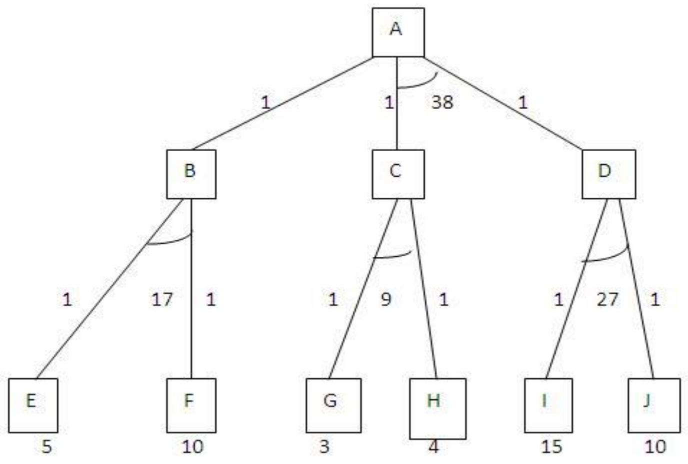
Another way is to consider the different ways that the goal state can be decomposed into simpler sub-goals. To represent problem reduction techniques we need to use an AND-OR graph/tree.

- This decomposition or reduction generates arcs that we call AND arcs. One AND arc may point to a number of successor nodes, all of which must be solved in order for the arc to point to a solution.

- As in OR graph, several arcs may emerge from a single node, indicating the variety of ways in which the original problem might be solved. That is why is called AND-OR graph
- AO* algorithm is problem reduction technique for solving problems presented in cyclic graph.



If we look just at the nodes and choose for expansion the one with the lowest 'f' value, we must select C. But here in this case it is better to explore the path going through B. Since to use C we must also use D for a total cost of 9(C+D+2) compared to the cost of 6 that we get by going through B.



The most promising single node is G with an 'f' value of 3. It is even part of the most promising arc G-H, with a total cost of

9. But that arc is not part of the current best path. Since to use it we must also use the arc I-J with a cost of 27. The path from A through B to E & F is better, with a total cost of 18. So we should not expand G next; rather we should examine either E or F.

<u>*AO** Algorithm</u>

1. Initialize the graph to start node
2. Traverse the graph following the current path accumulating nodes that have not yet been expanded or solved
3. Pick any of these nodes and expand it and if it has no successors call this value *FUTILITY* otherwise calculate only *f*' for each of the successors.
4. If *f*' is 0 then mark the node as *SOLVED*
5. Change the value of *f*' for the newly created node to reflect its successors by back propagation.
6. Wherever possible use the most promising routes and if a node is marked as *SOLVED* then mark the parent node as *SOLVED*.
7. If starting node is *SOLVED* or value greater than *FUTILITY*, stop, else repeat from 2.

<u>A* vs. AO* Algorithm:</u> The main difference between the A* and AO* algorithms are:

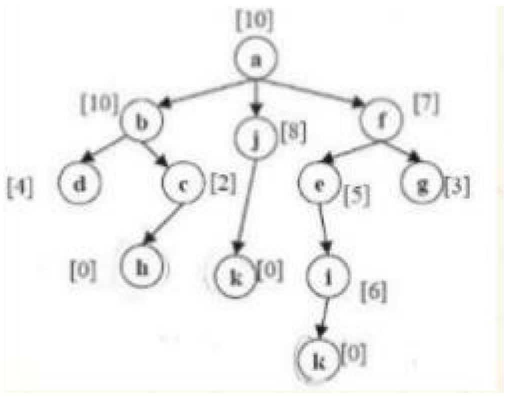
- A* algorithm is an OR Graph Algorithm while AO* is an AND-OR Graph Algorithm.

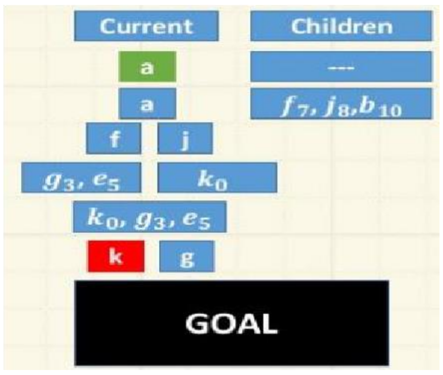
- A* algorithm cost function include f’ = g’ + h’ while AO* algorithm cost function is simply f’ = h’.
- AO* algorithm does not always give minimum cost and the problems can be divided into simpler tasks because of being an AND-OR graph.
- OR graph algorithm it just find only one solution (i.e., either OR solution means this OR this OR this). But in the AND-OR graph algorithm find more than one solution by ANDing two or more branches.

Beam Search

- Beam search explores more than one path together. A factor k is used to determine the number of branches explored at a time.
- If k=2, then 2 branches are explored at a time. For k=4, 4 branches are explored simultaneously.
- The branches selected are the best branches based on the heuristic evaluation function.

Example:





Algorithm:

1. Assume width of beam ω
2. Assign the initial node on a list START.
3. If START has no node or START=GOAL, stop search.
4. Otherwise take the first node from START. Call this node a.
5. If (a=GOAL) terminate search with success.
6. Else if node has children, expand all of them. Add them at the last of START.
7. Use a heuristic function to sort the nodes of START.
8. Identify the nodes to be expanded. The number of nodes not more than ω.
9. Name the new list as START1
10. Replace START with START1.
11. GOTO step 2.

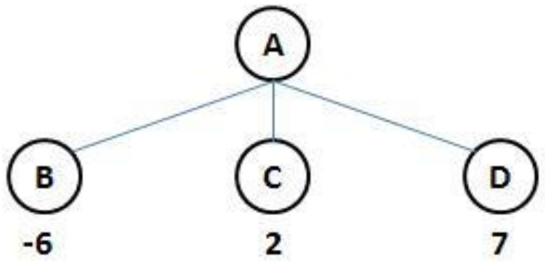
Adversarial Search

- It relates to competitive environment in which the agent goals are conflict giving rise to adversarial search.
- There are two methods for game playing:
  1. Min-Max Procedure
  2. Alpha-Beta Pruning (Cut-offs)

Min-Max Search

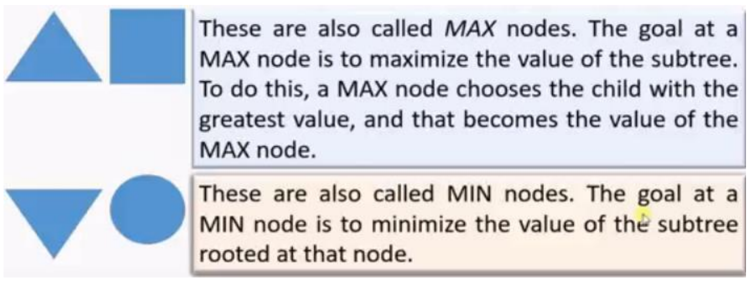
- Mini-max strategy is a simple strategy for two player game. Here, one player is called “maximizer” and the other called “minimizer. Maximizer tries to maximize its score while minimizer tries to minimize maximizer’s score.
- A game can be defined a search problem with the following components:
  - **Initial state**: It comprises the position of the board and showing whose move it is.
  - **Successor function**: It defines what the legal moves a player can make are.
  - **Terminal state**: It is the position of the board when the game gets over.

- **Utility function**: It is a function which assigns a numeric value for the outcome of a game. For instance, in chess or tic-tac-toe, the outcome is either a win, a loss, or a draw, and these can be represented by the values +1, -1, or 0, respectively



- It is also assumed that the maximizer makes the first move (not essential, as minimizer can also make first move). The maximizer, always tries to go a position where the static evaluation function value is the maximum positive value.
- The maximizer being the player to make the first move, and will move to node D because the static evaluation function value for that is maximum. . The same above figure shows that if the minimizer has to make the first move, it will go to node B because the static evaluation function value at that node will be advantageous to it.

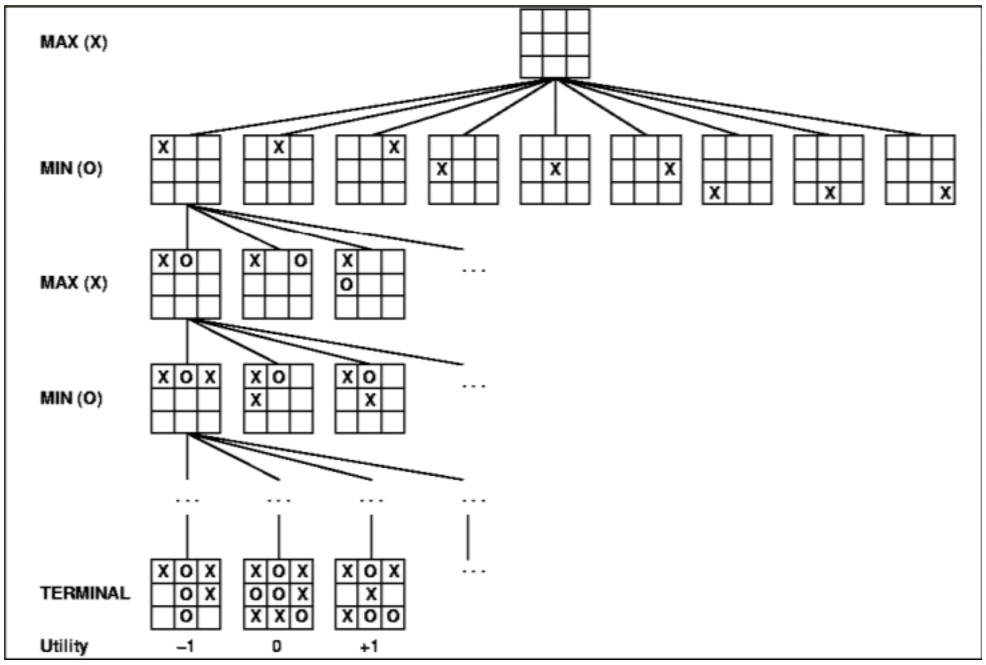
- Once the static evaluation function is applied at the leaf nodes, backing up values can begin. First we compute the backed-up values at the parents of the leaves.
- Also, game playing strategy never stops with one level but looks ahead, i.e., moves a couple of levels downwards to choose the optimal path.



▲ ■	These are also called *MAX* nodes. The goal at a MAX node is to maximize the value of the subtree. To do this, a MAX node chooses the child with the greatest value, and that becomes the value of the MAX node.
▼ ●	These are also called MIN nodes. The goal at a MIN node is to minimize the value of the subtree rooted at that node.

<u>Example:</u>

In a Tic-Tac-Toe game:



Outcome as loss with -1, win with +1 and draw with 0

Consider the following example:

```
MAX ......  S
          / | \
        /   |   \
MIN .... A    B    C
        /|\   |\   |\
       / | \  | \  | \
      D  E  F G  H I  J
    100  3 -1 6  5 2  9
```

- A's backed-up value is -1 (= min (100, 3, -1), meaning that if the opponent ever reaches the board associated with this node, then it will pick the move associated with the arc from A to F. Similarly, B's backed-up value is 5 (corresponding to child H) and C's backed-up value is 2 (corresponding to child I).
- Next, we backup values to the next higher level, in this case to the MAX node S. Since it is our turn to move at this node, we select the move that looks best based on the backed-up values at each of S's children. In this case the best child is B since B's backed-up value is 5 (= max (-1, 5, 2)). So the minimax value for the root node S is 5, and the move selected based on this 2-ply search is the move associated with the arc from S to B.
- It is important to notice that the backed-up values are used at nodes A, B, and C to evaluate which is best for S; we do *not* apply the static evaluation function at any non-leaf node. Why? Because it is assumed that the values computed at nodes farther ahead in the game (and therefore lower in the tree) are more accurate evaluations of quality and therefore are preferred over the evaluation function values if applied at the higher levels of the tree.

Alpha-Beta Pruning

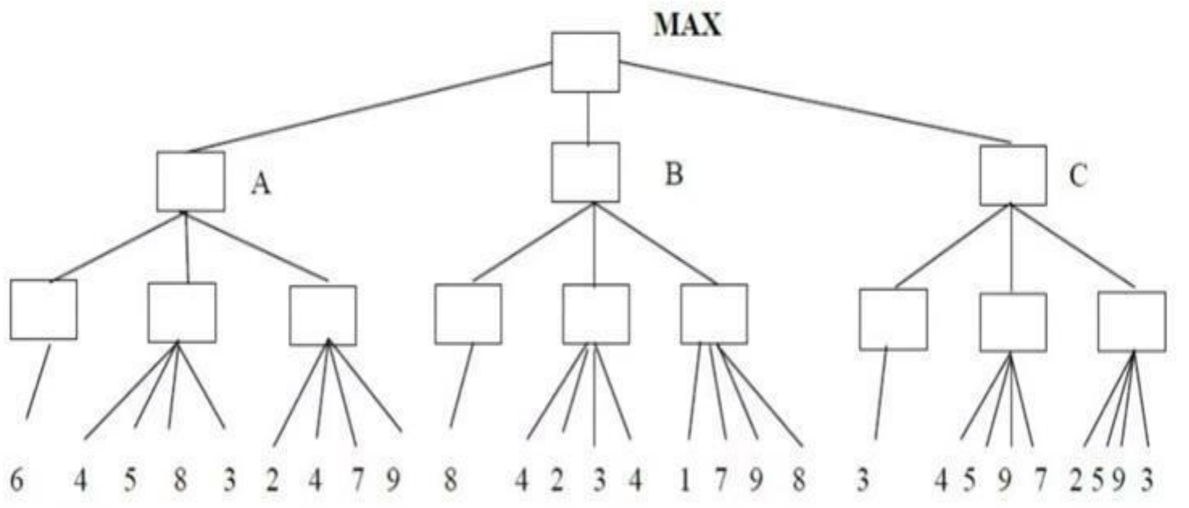
The method that we are going to look in this article is called alpha-beta pruning. If we apply alpha-beta pruning to a standard minimax algorithm, it returns the same move as the standard one, but it removes (prunes) all the nodes that are possibly not affecting the final decision.

Alpha-Beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by mini-max algorithm in its search tree.

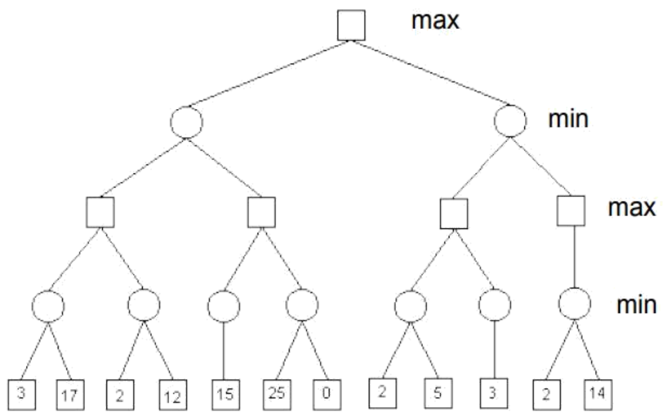
- α is a value which is best for Max player (highest value)
- β is a value which is best for Min player (lowest value)

Each node will keep its α-β values

For Min node, if β<= α of max ancestor, PRUNE.
For Max node, if β>= α of min ancestor, PRUNE.

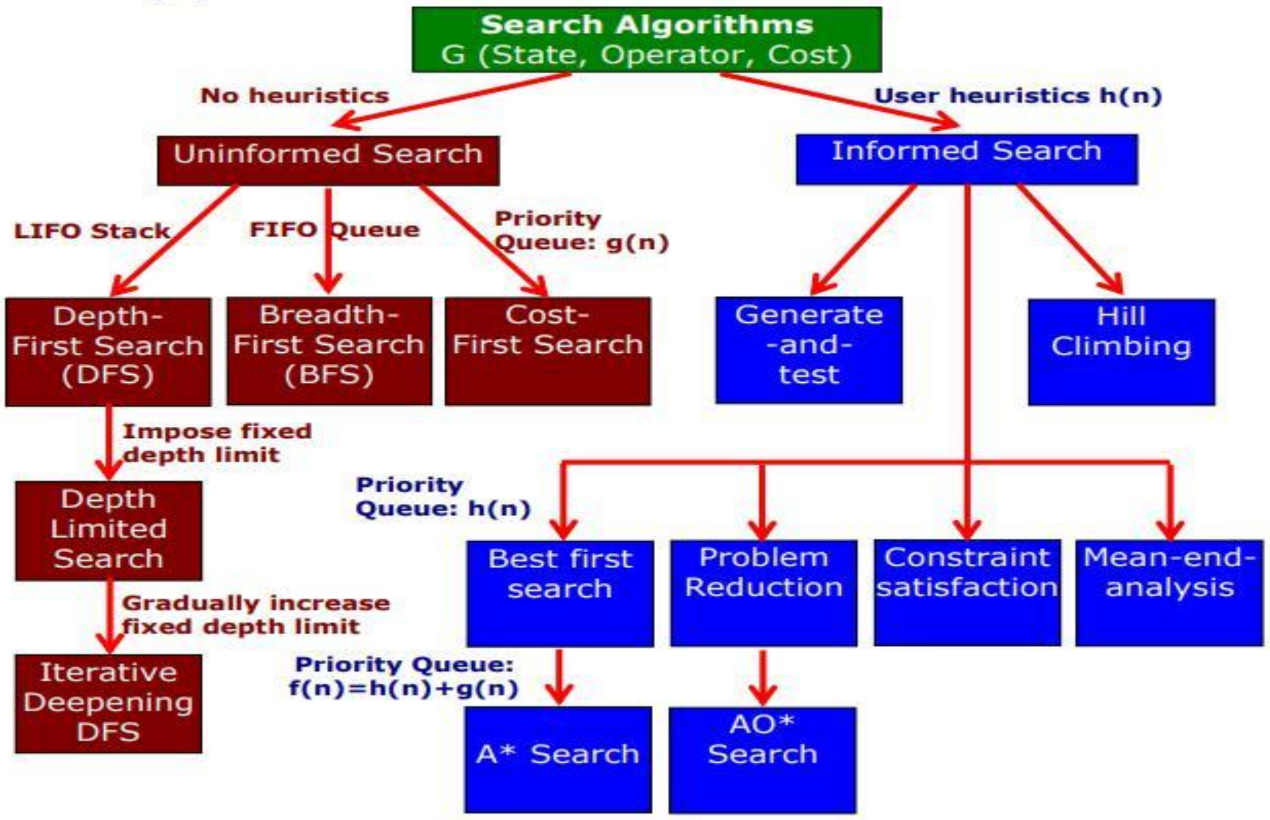


Exercise: Consider the following state space graph



a) What is value at the root, using minimax alone?
b) What nodes could have been pruned from the search using alpha-beta pruning? Show values of alpha and beta

Difference between Uninformed search and informed search:


Search Algorithms
G (State, Operator, Cost)
No heuristics
User heuristics h(n)
Uninformed Search
Informed Search
LIFO Stack
FIFO Queue
Priority Queue: g(n)
Depth-First Search (DFS)
Breadth-First Search (BFS)
Cost-First Search
Generate-and-test
Hill Climbing
Impose fixed depth limit
Depth Limited Search
Priority Queue: h(n)
Best first search
Problem Reduction
Constraint satisfaction
Mean-end-analysis
Gradually increase fixed depth limit
Iterative Deepening DFS
Priority Queue: f(n)=h(n)+g(n)
A* Search
AO* Search