**III. PROBLEM SPACE**

1. A **‘problem space’** is an abstract space. The problem space theory uses the approach of *defining the problem to find the solution*. A problem space will ***not offer details*** of a solution but instead focus on ***steps and goals*** involved in working through a problem. Basically:

* A problem space encompasses all *valid states* that can be generated by the application of any combination of *operators* on any combination of *objects*.
* The problem space may contain one or more *solutions.*

**IV. SEARCH TECHNIQUES**

A ‘**search**’ refers to ***search for a solution*** in a problem space. A Hierarchical representation of most search algorithms is illustrated below. The representation begins with two types of search:

• **Uninformed Search:** Also called blind, exhaustive or brute-force search, it uses no information about the problem to guide the search and therefore may not be very efficient.

• **Informed Search:** Also called **heuristic** or **intelligent search**, this uses information about the problem to guide the search—usually guesses the distance to a goal state and is therefore efficient, but the search may not be always possible.



**Uninformed Search**



**V. HEURISTIC SEARCH TECHNIQUES**

Blind search is not always possible, because it requires too much time or Space (memory). To find a solution in proper time rather than a complete solution in unlimited time we use heuristics. ‘***A heuristic function is a function that maps from problem state descriptions to measures of desirability, usually represented as numbers***’. The technique that makes us of ***heuristic functions***are called ***heuristic algorithms***

Heuristic search methods use knowledge about the ***problem domain*** and choose promising operators first. These heuristic search methods use ***heuristic functions*** to evaluate the next state towards the goal state. For finding a solution, by using the heuristic technique, one should carry out the following steps:

* ***Add domain-specific information*** to select what is the best path to continue searching along.
* ***Define a heuristic function h(n)*** that estimates the ‘goodness’ of a node n.

Specifically, **h(n) = estimated cost (or distance) of minimal cost path from *n* to a goal state.**

* The term, heuristic means ‘serving to aid discovery’ and is an estimate, based on domain specific information that is computable from the current state description of how close we are to a goal.

Finding **a route from one city to another city** is an example of a search problem in which different search orders and the use of heuristic knowledge are easily understood.

* **State:** The current city in which the traveler is located.
* **Operators:** Roads linking the current city to other cities.
* **Cost Metric:** The cost of taking a given road between cities.
* **Heuristic information:** The search could be guided by the direction of the goal city from the current city.

Some prominent Intelligent search/ Heuristic search algorithms are stated below:

1. **Generate and Test Search**
2. **Hill Climbing**
3. **Best-first Search**
4. **A\* Search**
5. **Problem Reduction(AO\*)**
6. **Constraint Search**
7. **Means-ends analysis**

**1) Generate-And-Test Algorithm**

Generate-and-test search algorithm is a very simple algorithm that guarantees to find a solution if done systematically and there exists a solution.

**Algorithm: Generate-And-Test**

1. Generate a possible solution.

2. Test to see if this is the expected solution.

3. If the solution has been found quit else go to step 1.



Generate-and-test, like depth-first search, requires that complete solutions be generated for testing. In its most systematic form, it is only an exhaustive search of the problem space. Solutions can also be generated randomly but solution is not guaranteed.

This approach is what is known as ***British Museum algorithm:*** finding an object in the British Museum by wandering randomly.

The main drawback is that it takes long time and large space.

**2) Hill Climbing**

It is a variant of generate and test algorithm. Generation of next state depends on feedback from test procedure. Then this feedback is utilized by the generator in deciding the next move in search space.

**Generate and test function + heuristic function 🡪 Tell the closeness of goal state**

In hill climbing the most promising child is selected for expansion. When the children have been generated, alternative choices are evaluated using some type of heuristic function.

**Types of Hill Climbing**

1. Simple Hill climbing
2. Steepest Ascent Hill Climbing
3. **Simple Hill Climbing**

It uses heuristic to move only to states that are better than the current state. It aims to move always to better state when possible.

**Algorithm for Simple Hill Climbing:**

*Step 1: Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make initial state as current state.*

*Step 2: Loop until the solution state is found or there are no new operators present which can be applied to current state.*

*a) Select a state that has not been yet applied to the current state and apply it to produce a new state.*

*b) Perform these to evaluate new state*

*i. If the current state is a goal state, then stop and return success.*

*ii. If it is better than the current state, then make it current state and proceed further.*

*iii. If it is not better than the current state, then continue in the loop until a solution is found.*

*Step 3 : Exit.*

1. **Steepest-Ascent Hill Climbing**

It is a variation of simple hill climbing. Instead of moving to the first state that is better, move to best possible state that is on move away. Here the order of operators does not matter. It is not just climb to a better state but climb up the steepest slope.

**Algorithm**

*Step 1: Evaluate the initial state. If it is goal state then exit else make the initial state as current state.*

*Step 2: Repeat these steps until a solution is found or current state does not change*

1. *Let ‘target’ be a state such that any successor of the current state will be better than it;*
2. *if target is better than current state set current state to Target*
3. *for each operator that applies to the current state*

*a. Create a new state*

*b. evaluate the new state*

* *if this state is goal state then quit else compare with ‘target’*
* *if this state is better than ‘target’, set this state as ‘target’*

*Step 3: Exit*

**2.3 Best First Search**

All search methods can be broadly classified into two categories:

1. Uninformed (or Exhaustive or Blind) methods, where the search is carried out without any additional information that is already provided in the problem statement. Some examples include Breadth-First Search, Depth First Search etc.
2. Informed (or Heuristic) methods, where the search is carried out by using additional information to determine the next step towards finding the solution. BFS is an example of such algorithms

If we consider searching as a form of traversal in a graph, an uninformed search algorithm would blindly traverse to the next node in a given manner without considering the cost associated with that step. An informed search, like BFS, on the other hand, would use an evaluation function to decide which among the various available nodes is the most promising (or ‘BEST’) before traversing to that node.

BFS uses the concept of a Priority queue and heuristic search. To search the graph space, the BFS method uses two lists for tracking the traversal. An ‘Open’ list that keeps track of the current ‘immediate’ nodes available for traversal and a ‘CLOSED’ list that keeps track of the nodes already traversed.

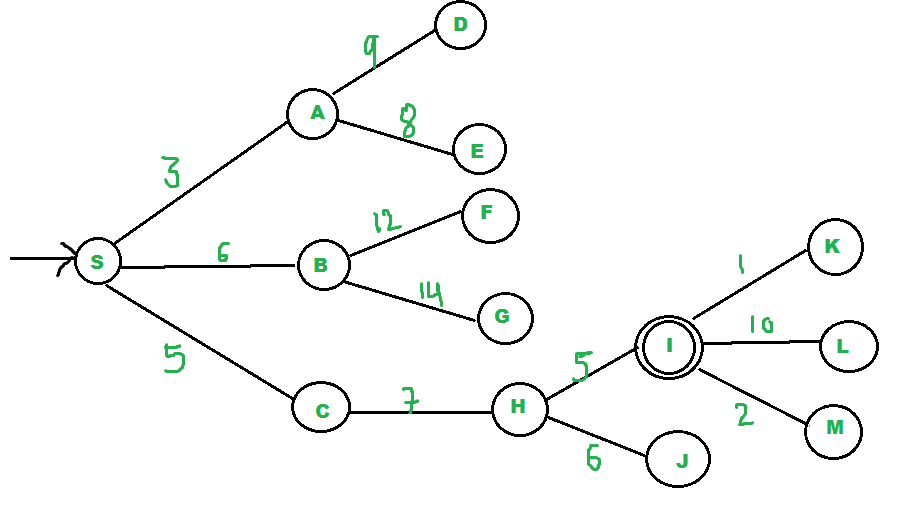
**Best First Search Algorithm**

1. Create 2 empty lists: OPEN and CLOSED
2. Start from the initial node (say N) and put it in the ‘ordered’ OPEN list
3. Repeat the next steps until the GOAL node is reached
   1. If the OPEN list is empty, then EXIT the loop returning ‘False’
   2. Select the first/top node (say N) in the OPEN list and move it to the CLOSED list. Also, capture the information of the parent node
   3. If N is a GOAL node, then move the node to the Closed list and exit the loop returning ‘True’. The solution can be found by backtracking the path
   4. If N is not the GOAL node, expand node N to generate the ‘immediate’ next nodes linked to node N and add all those to the OPEN list
   5. Reorder the nodes in the OPEN list in ascending order according to an evaluation function f(n)

This algorithm will traverse the shortest path first in the queue. The time complexity of the algorithm is given by O(n\*logn).

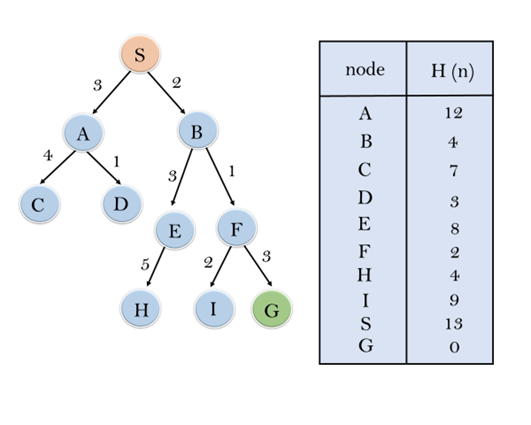
**Advantages**:  
1. Can switch between BFS and DFS, thus gaining the advantages of both.  
2. More efficient when compared to DFS.

**Disadvantages:**  
1. Chances of getting stuck in a loop are higher.

****

Starting node is S and goal node is L

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.



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

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

**Iteration2:** Open[E,F,A],Closed[S,B]  
                  : Open [E, A], Closed [S, B, F]

**Iteration3:** Open[I,G,E,A],Closed[S,B,F]  
                  : Open [I, E, A], Closed [S, B, F, G]

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

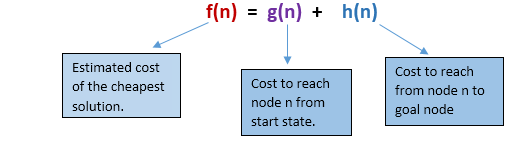
**2.4 A\*Algorithm**

**A \* algorithm** is a searching algorithm that searches for the shortest path between the initial and the final state. It is used in various applications, such as maps. In maps the A\* algorithm is used to calculate the shortest distance between the source (initial state) and the destination (final state).

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 and uses g(n)+h(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.

****

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.

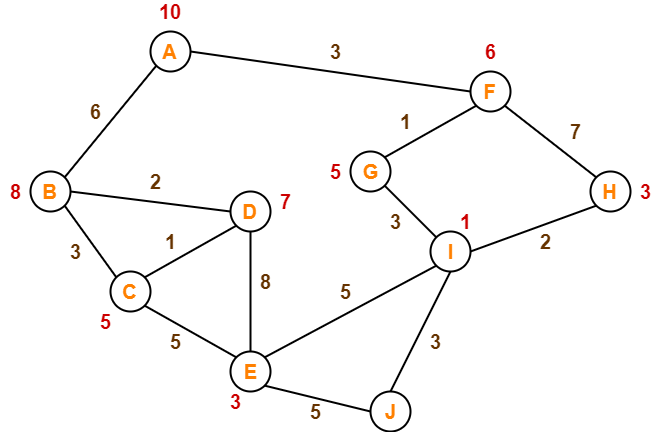
Disadvantages:

* 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.

**Problem 1:**

* The numbers written on edges represent the distance between the nodes.
* The numbers written on nodes represent the heuristic value.

Find the most cost-effective path to reach from start state A to final state J using A\* Algorithm.

****

**Solution:**

**Step-01:**

* We start with node A.
* Node B and Node F can be reached from node A.

 A\* Algorithm calculates f(B) and f(F).

* f(B) = 6 + 8 = 14
* f(F) = 3 + 6 = 9

Since f(F) < f(B), so it decides to go to node F.

**Path- A → F**

**Step-02:**

 Node G and Node H can be reached from node F.

A\* Algorithm calculates f(G) and f(H).

* f(G) = (3+1) + 5 = 9
* f(H) = (3+7) + 3 = 13

 Since f(G) < f(H), so it decides to go to node G.

**Path- A → F → G**

**Step-03:**

 Node I can be reached from node G.

 A\* Algorithm calculates f(I).

f(I) = (3+1+3) + 1 = 8 It decides to go to node I.

**Path- A → F → G → I**

**Step-04:**

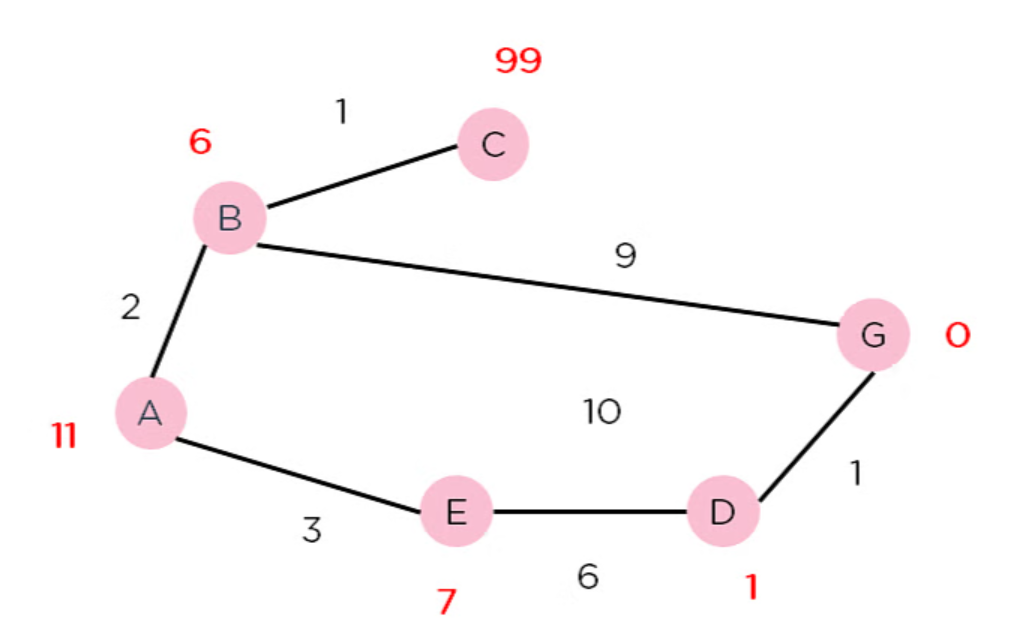
 Node E, Node H and Node J can be reached from node I.

 A\* Algorithm calculates f(E), f(H) and f(J).

* f(E) = (3+1+3+5) + 3 = 15
* f(H) = (3+1+3+2) + 3 = 12
* f(J) = (3+1+3+3) + 0 = 10

 Since f(J) is least, so it decides to go to node J.

**Path- A → F → G → I → J**

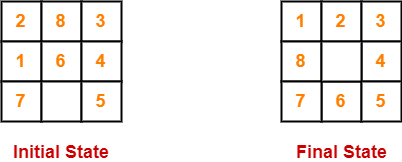
****

**start node A**

**Goal state 5**

**Problem:-2**

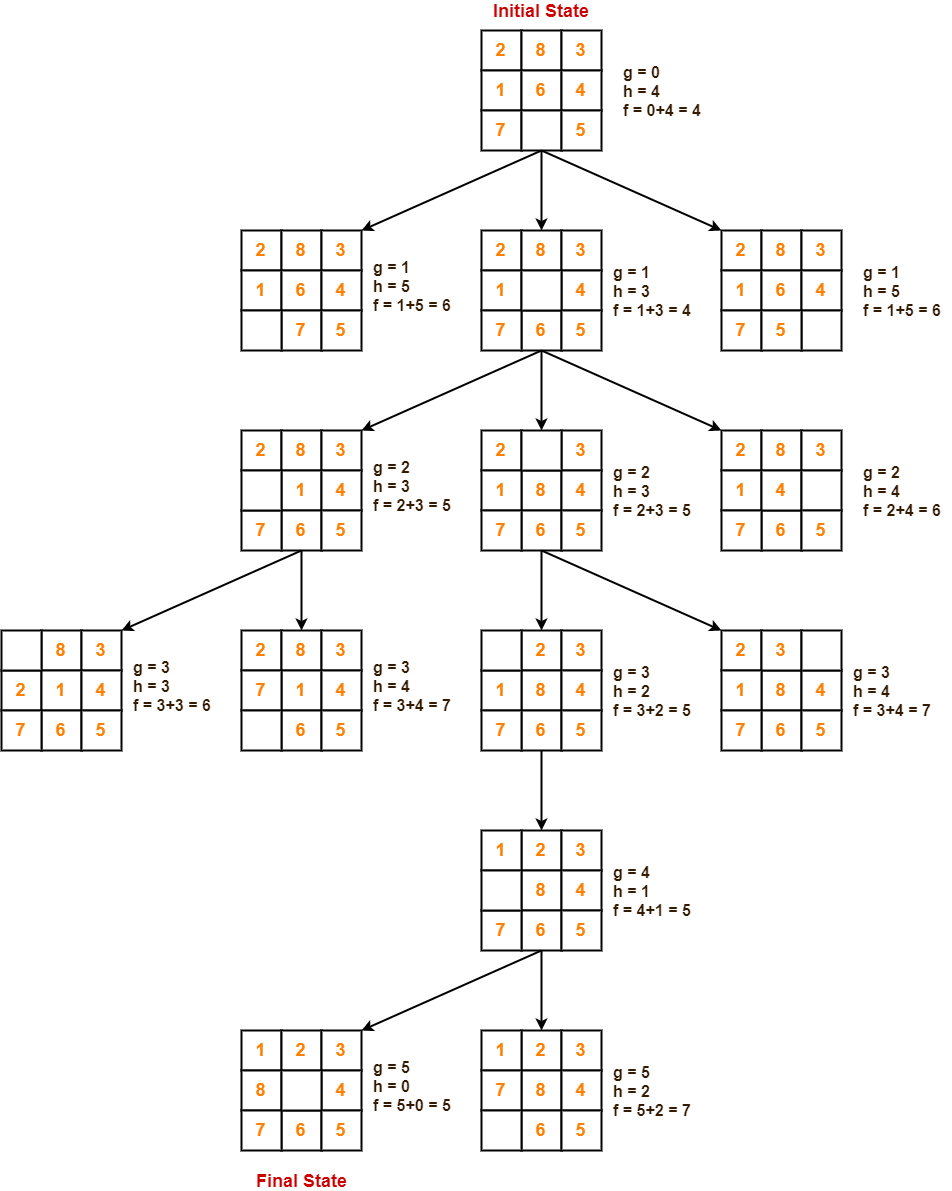
Given an initial state of a 8-puzzle problem and final state to be reached-



Find the most cost-effective path to reach the final state from initial state using A\* Algorithm.

Consider g(n) = Depth of node and h(n) = Number of misplaced tiles.

**Solution:**

****

**What is AO\* Algorithm?**

The AO\* algorithm, short for "Anytime Optimistic" algorithm, is a search algorithm used in artificial intelligence and computer science to find the optimal path or solution in a graph or state space. It is particularly useful in applications like robotics, path finding, and planning, where finding the best possible solution is essential.

The AO\* algorithm belongs to the family of informed search algorithms, meaning it utilizes heuristics or estimated cost functions to guide the search process. It efficiently balances the trade-off between computation time and the quality of the solution. Unlike some other algorithms that focus solely on finding the optimal solution, AO\* provides a series of progressively improving solutions, making it adaptable to various scenarios.

**Working of AO\* Algorithm**

The **AO\* algorithm** is designed to efficiently search through a state space or graph to find the optimal solution while providing the flexibility to return intermediate solutions at any time. Its operation can be broken down into several key steps:

**1. Initialization**

The algorithm begins with the initialization of critical components:

**Start State:**

It starts from the initial state, which represents the current state of the problem or the starting point in a graph.

**Cost Estimates:**

For each state, AO\* maintains an "optimistic" cost estimate, denoted as g\*(s), which serves as a lower bound on the true cost from the start state to that state. Initially, these cost estimates are set to infinity for all states except the start state, which has a cost estimate of zero.

**Priority Queue:**

AO\* uses a priority queue (often implemented as a binary heap) to keep track of states that need to be expanded. States are prioritized in the queue based on their g\*(s) values, with states having lower cost estimates being higher in priority.

**2. Iterative Expansion**

The core of AO\* is an iterative process that repeatedly selects and expands the most promising state from the priority queue. This process continues until certain termination conditions are met. Here's how it works:

* **Selecting a State:**  
  The algorithm selects the state with the lowest g\*(s) value from the priority queue. This state represents the most promising path discovered so far.
* **Expanding a State:**  
  Once a state is selected, AO\* generates its successor states, which are the states reachable from the current state by taking valid actions or moving along edges in the graph. These successor states are generated and evaluated.
* **Updating Cost Estimates:**  
  For each successor state, the algorithm updates its g\*(s) value. The updated value depends on the cost of reaching that successor state from the current state and the g\*(s) value of the current state.
* **Adding to Priority Queue:**  
  The newly generated states, along with their updated g\*(s) values, are added to the priority queue.

**3. Termination**

The search process continues until certain termination conditions are met. These conditions can include:

* **A predefined time limit:**  
  The algorithm stops after a specified amount of time has elapsed.
* **A user request:**  
  The user can request the algorithm to stop and return the best solution found so far.
* **The discovery of an optimal solution:**  
  If AO\* finds a solution that satisfies the problem constraints, it can terminate.

**4. Solution Retrieval**

One of the unique features of AO\* is its ability to return solutions incrementally. At any point during the search, the user can decide to stop the algorithm and retrieve the best solution found so far. This flexibility is particularly valuable in real-time systems where immediate responses are required, and waiting for the optimal solution may not be feasible.

**5. Adaptation**

Another essential aspect of AO\* is its adaptability. It can adjust its search strategy based on available computational resources and user requirements. If more time or computational power is available, AO\* can perform a more exhaustive search to improve the solution quality. Conversely, if resources are limited, it can return a solution quickly without completing the entire search.

**Difference between A\* Algorithm and AO\* Algorithm**

| **Aspect** | **A\* Algorithm** | **AO\* Algorithm** |
| --- | --- | --- |
| Anytime Nature | No | Yes (Provides incremental solutions) |
| Optimistic Cost Estimates | Single cost estimate based on heuristics | Optimistic cost estimates for each state |
| Adaptability | Fixed-depth search | Adapts to available time and resources |
| Prioritization | Always seeks the single optimal path | Balances quality vs. time efficiently |
| Solution Completeness | Finds the single optimal solution | Provides a series of progressively improving solutions |
| Termination Condition | Typically continues until optimal solution found | Can be terminated at any time, returning the best solution found so far |
| Use Cases | Used when finding the absolute best solution is top priority | Preferred in real-time systems and applications requiring adaptability |
| Real-time Applications | Less suitable for real-time systems | Widely used in robotics, video games, network routing, autonomous vehicles, etc. |
| Quality vs. Time Trade-off | Prioritizes solution quality over time efficiency | Balances between solution quality and computation time efficiently |
| Search Strategy | Fixed-depth search with no adaptability | Adapts search strategy based on available computational resources |

**Advantages and Disadvantages of AO\* Algorithm**

**Advantages**

1. **Adaptability:**  
   AO\* can adapt to changing requirements and computational resources, making it suitable for real-time systems.
2. **Incremental Solutions:**  
   It provides incremental solutions, allowing users to make progress while the search continues.
3. **Optimistic Estimates:**  
   The use of optimistic cost estimates can guide the search efficiently.
4. **Heuristic Guidance:**  
   Like A\*, it benefits from heuristic guidance, improving search efficiency.

**Disadvantages**

1. **Quality vs. Time Trade-off:**  
   AO\* sacrifices optimality for adaptability. It may not always find the absolute best solution but provides a good compromise between quality and time.
2. **Complexity:**  
   Implementing AO\* can be more complex than simpler algorithms due to its adaptability and nature.
3. **Heuristic Quality:**  
   The effectiveness of AO\* heavily depends on the quality of the heuristic function. Poor heuristics can lead to suboptimal solutions.

**Real-life Applications of AO\* Algorithm**

AO\* finds applications in various fields:

**Robotics:**

AO\* is widely used in robotic path planning. Robots can navigate complex environments while continuously improving their routes.

**Video Games:**

In video games, AO\* is used for character pathfinding, ensuring that game characters move efficiently and avoid obstacles.

**Network Routing:**

In computer networking, AO\* helps in finding optimal routes for data packets in dynamic networks.

**Autonomous Vehicles:**

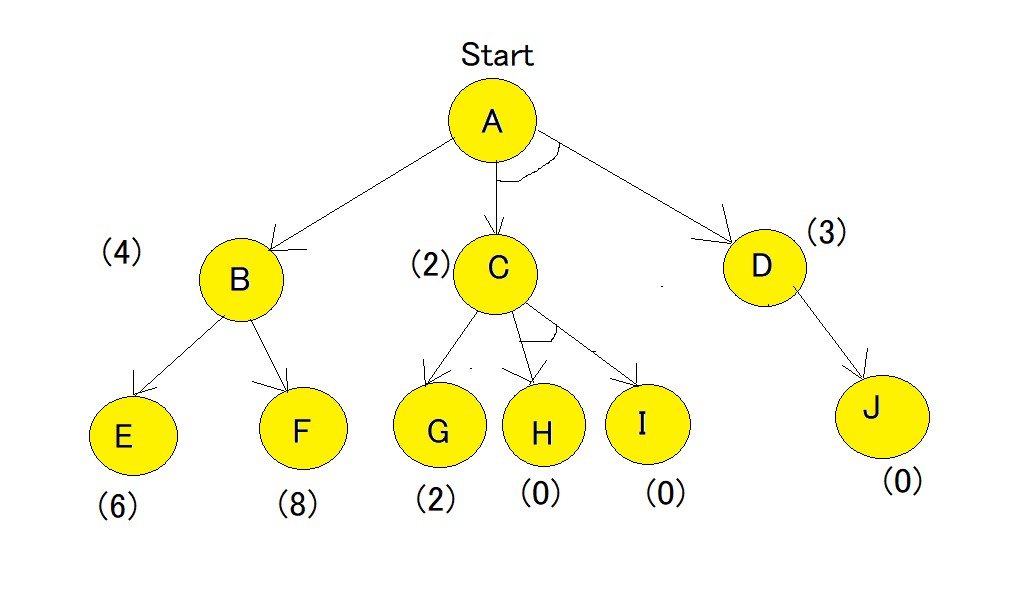
Autonomous vehicles use AO\* for real-time route planning, taking into account traffic conditions and obstacles.

**Natural Language Processing:**

AO\* is used in parsing and grammar-checking algorithms to generate syntactically correct sentences incrementally.

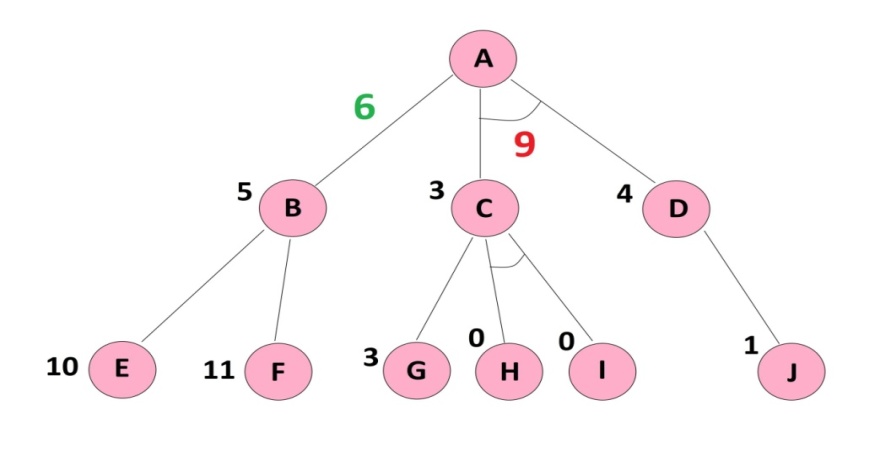
**Resource Management:**

It is used in resource allocation and scheduling, such as allocating resources in cloud computing.

  
Here, in the above example all numbers in brackets are the heuristic value i.e h(n). Each edge is considered to have a value of 1 by default.

**Step-1**  
Starting from node A, we first calculate the best path.  
f(A-B) = g(B) + h(B) = 1+4= 5 , where 1 is the default cost value of travelling from A to B and 4 is the estimated cost from B to Goal state.  
f(A-C-D) = g(C) + h(C) + g(D) + h(D) = 1+2+1+3 = 7 , here we are calculating the path cost as both C and D because they have the AND-Arc. The default cost value of travelling from A-C is 1, and from A-D is 1, but the heuristic value given for C and D are 2 and 3 respectively hence making the cost as 7.  
  
**The minimum cost path is chosen i.e A-B.**

**Step-2**  
Using the same formula as step-1, the path is now calculated from the B node,  
f(B-E) = 1 + 6 = 7.  
f(B-F) = 1 + 8 = 9  
Hence, the B-E path has lesser cost. Now the heuristics have to be updated since there is a difference between actual and heuristic value of B. The minimum cost path is chosen and is updated as the heuristic , in our case the value is 7. And because of change in heuristic of B there is also change in heuristic of A which is to be calculated again.  
f(A-B) = g(B) + updated((h(B)) = 1+7=8  
  
**Step-3**  
Comparing path of f(A-B) and f(A-C-D) it is seen that f(A-C-D) is smaller. Hence f(A-C-D) needs to be explored.  
Now the current node becomes C node and the cost of the path is calculated,  
f(C-G) = 1+2 = 3  
f(C-H-I) = 1+0+1+0 = 2  
f(C-H-I) is chosen as minimum cost path,also there is no change in heuristic since it matches the actual cost. Heuristic of path of H and I are 0 and hence they are solved, but Path A-D also needs to be calculated , since it has an AND-arc.  
f(D-J) = 1+0 = 1, hence heuristic of D needs to be updated to 1. And finally the f(A-C-D) needs to be updated.  
f(A-C-D) = g(C) + h(C) + g(D) + updated((h(D)) = 1+2+1+1 =5.  
**As we can see that the solved path is f(A-C-D).**

****

**KNOWLEDGE REPRESENTATION**

For the purpose of solving complex problems encountered in AI, we need both a large amount of knowledge and some mechanism for manipulating that knowledge to create solutions to new problems. A variety of ways of representing knowledge (facts) have been exploited in AI programs. In all variety of knowledge representations, we deal with two kinds of entities.

A. **Facts**: Truths in some relevant world. These are the things we want to represent.

B. **Representations of facts** in some chosen formalism. These are things we will actually be able to manipulate.

One way to think of structuring these entities is at two levels : (a**) the knowledge level**, at which facts are described, and (b) the symbol level, at which representations of objects at the knowledge level are defined in terms of symbols that can be manipulated by programs.

The facts and representations are linked with two-way mappings. This link is called representation mappings. The forward representation mapping maps from facts to representations. The backward representation mapping goes the other way, from representations to facts.

One common representation is natural language (particularly English) sentences. Regardless of the representation for facts we use in a program, we may also need to be concerned with an English representation of those facts in order to facilitate getting information into and out of the system. We need mapping functions from English sentences to the representation we actually use and from it back to sentences.

****

Fig: Mapping between Facts and Representations

* 1. **Techniques of knowledge representation**

The various knowledge representation scheme included in AI are:

1. Logic
2. Semantic Network
3. Production Rules
4. Frames
5. Associative Networks
6. Fuzzy Logic
7. Object Oriented Methods

**LOGIC**

Logical representation means drawing a conclusion based on various conditions. This representation lays down some important communication rules. It consists of precisely defined **syntax and semantics** which supports the sound inference. **Each sentence can be translated into logics using syntax and semantics.**

**Syntax (How to write/ Structure):**

* Syntaxes are the rules which decide how we can construct legal sentences in the logic.
* It determines which symbol we can use in knowledge representation.
* How to write those symbols.

**Semantics (meaning part):**

* Semantics are the rules by which we can interpret the sentence in the logic.
* Semantic also involves assigning a **meaning** to each sentence.

For instance, logics are used to design the digital circuits, knowledge representation is based on logical formalisms. Logical representation can be categorized into mainly two logics:

* **Propositional Logics**
* **Predicate logics**

**II. Propositional Logic**

Propositional logic is logic at **sentential level**. Propositional logic applies the **Boolean logic** to convert our real-world data into a format that is readable to the computer. A proposition has TRUTH values (0 and 1) which mean it can have one of the two values i.e. **True or False but not both**. It is the most basic and widely used logic. For Example,

1. The sun rises in the East and sets in the West.**T**

2. 1 + 1 = 2 **T**

3. 'b' is a vowel. **F**

All of the above sentences are propositions, where the first two are Valid(True) and the third one is Invalid(False). ***Some sentences that do not have a truth value or may have more than one truth value are not propositions.*** For Example,

1. What time is it?

2. Go out and play.

3. x + 1 = 2.

The above sentences are **not propositions** as the first two do not have a truth value, and the third one may be true or false.

**1. Features of Propositional Logic**

* Propositional logic consists of an object, relations or function, and **logical connectives**.
* These connectives are also called logical operators, which connect two sentences.
* The **propositions and connectives** are the basic elements of the propositional logic.
* A proposition formula which is always true is called **tautology**, and it is also called a valid sentence.
* A proposition formula which is always false is called **Contradiction**.
* Statements which are questions, commands, or opinions are not propositions such as "**Where is Rohini**", "**How are you**", "**What is your name**", are not propositions.

**2. Types of Propositions**

There are two types of propositions:

* **Atomic:-** Atomic means a single preposition like ‘the sky is blue’, ‘hot days are humid’, water is liquid, etc.
* **Complex:-** Complex prepositions are those, which have been formed by connecting one, two, or more sentences. The propositions are combined together using**Logical Connectives or Logical Operators.**

**3. Logical Connectives:**

Logical connectives are used to connect two simpler propositions or representing a sentence logically. We can create compound propositions with the help of logical connectives. There are mainly five connectives, which are given as follows:



1. **Negation:** A sentence such as ¬ P is called negation of P. A literal can be either Positive literal or negative literal.
2. **Conjunction:** A sentence which has **∧**connective such as, **P ∧ Q** is called a conjunction.  
   **Example:** Rohan is intelligent and hardworking. It can be written as,  
   **P= Rohan is intelligent,**

**Q= Rohan is hardworking**.so we can write it as **P∧ Q**.

1. **Disjunction:** A sentence which has ∨ connective, such as **P ∨ Q**. is called disjunction, where P and Q are the propositions. **Example: "Ritika is a doctor or Engineer"**,  
   **P= Ritika is Doctor.**

**Q= Ritika is Doctor**, so we can write it as **P ∨ Q**.

1. **Implication:** A sentence such as P → Q, is called an implication. Implications are also known as **if-then** rules. It can be represented as

Eg: **If it is raining, then the street is wet.**

**P= It is raining,**

**Q= Street is wet**, so it is represented **as P → Q**

1. **Biconditional:** A sentence such as **P⇔ Q** is a Biconditional sentence**,**

Example **If I am breathing, then I am alive**

**P= I am breathing,**

**Q= I am alive**, it can be represented as **P ⇔ Q.**

**4. Truth Table:**

In propositional logic, we need to know the truth values of propositions in all possible scenarios. We can combine all the possible combination with logical connectives, and the representation of these combinations in a tabular format is called **Truth table**. Following are the truth table for all logical connectives:



**Example:** Mark sentence whose proposition can be determined:

1. What time is it?
2. Go out and play.
3. x + 1 = 2.
4. Aristotle is dead.
5. Hildesheim is on the Rhine.
6. Logic is fun.

**5. Properties**

1. **Satisfiable:** A atomic propositional formula is satisfiable if there is an interpretation for which it is true.
2. **Tautology:** A propositional formula is valid or a tautology it is *true for all possible interpretations.*
3. **Contradiction:** A propositional formula is contradictory (unsatisfiable) *if there is no interpretation for which it is true* (i.e. all have false values).
4. **Contingent:** A propositional logic can be contingent which means it can be neither a tautology nor a contradiction.



**III.** **Predicate Logic or First-Order Predicate Logic (FOPL)**

Predicate logic is a verb phrase template that describes a property of **objects**, or a relationship among objects represented by the variables. Its basic unit is a predicate/ argument structure called an **atomic sentence**

**Syntax of First-Order Logic**

* Constants KingJohn, 2, …
* Predicates Brother, >, …
* Functions Sqrt, LeftArmOf, …
* Variables x, y, a, b, …
* Connectives ∧ ∨ ¬ ⇒ ⇔
* Equality =
* **Quantifiers **

**Components of First-Order Logic**

* **Term (represent object)**
  + Constant, e.g. Red
  + Variables, e.g. x1,x2…..
  + Function of constant, e.g. Color(Block1)
  + Function of Variable, weight(x1)
* **Predicate relating objects (no variable)**
  + Atomic Sentence
    - Brother (John, Richard)
    - Married (Mother(John), Father(John))
* **Complex Sentences**
  + Atomic sentences + logical connectives
    - Brother (John, Richard) ∧¬Brother (John, Father(John))

**Propositional logic n fopl pdf**

**Resolution pdf**

**I. HORN CLAUSE**

A*clause* (i.e., a [*disjunction*](https://mathworld.wolfram.com/Disjunction.html)*of*[*literals*](https://mathworld.wolfram.com/Literal.html)) is called a ***Horn clause*** if it contains at most one [positive literal](https://mathworld.wolfram.com/PositiveLiteral.html). Horn clauses are usually written as

|  |
| --- |
| L_1,...,L_n=>L(=¬L_1 v ... v ¬L_n v L) |

or

|  |
| --- |
| L_1,...,L_n=>(=¬L_1 v ... v ¬L_n), |

where n>=0 and L is the only [positive literal](https://mathworld.wolfram.com/PositiveLiteral.html).

A [***definite clause***](https://mathworld.wolfram.com/DefiniteClause.html) is a Horn clause that has [exactly one](https://mathworld.wolfram.com/ExactlyOne.html) [positive literal](https://mathworld.wolfram.com/PositiveLiteral.html). A Horn clause without a [positive literal](https://mathworld.wolfram.com/PositiveLiteral.html) is called a [***goal***](https://mathworld.wolfram.com/Goal.html).

Horn clauses express a subset of statements of [first-order logic](https://mathworld.wolfram.com/First-OrderLogic.html). Programming language Prolog is built on top of Horn clauses. Prolog programs are comprised of [definite clauses](https://mathworld.wolfram.com/DefiniteClause.html) and any question in Prolog is a [goal](https://mathworld.wolfram.com/Goal.html)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Disjunction form** | **Implication form** | **Read intuitively as** |
| **Definite clause** | ¬*p* ∨ ¬*q* ∨ ... ∨ ¬*t* ∨ *u* | *u* ← *p* ∧ *q* ∧ ... ∧ *t* | assume that, if *p* and *q* and ... and *t* all hold, then also *u* holds |
| **Fact** | *u* | *u* | assume that *u* holds |
| **Goal clause** | ¬*p* ∨ ¬*q* ∨ ... ∨ ¬*t* | *false* ← *p* ∧ *q* ∧ ... ∧ *t* | show that *p* and *q* and ... and *t* all hold |

In the non-propositional case, all variables in a clause are implicitly universally quantified with the scope being the entire clause. Thus, for example:

¬ *human*(*X*) ∨ *mortal*(*X*)

stands for:

∀X( ¬ *human*(*X*) ∨ *mortal*(*X*) ) as ¬a∨b= a →b

which is logically equivalent to:

∀X ( *human*(*X*) → *mortal*(*X*) )

**Expert System**

**II. Forward and Backward Chaining**

AI can help us solve numerous problems of varying complexities. One such type of problem is the case where **one has to predict outcomes using the given pool of knowledge.**  Here, the **knowledge base** is given and using **logical rules and reasoning**, one has to **predict the outcome**.

These problems are usually solved using Inference Engines, which utilize their two special modes: **Backward Chaining**and **Forward Chaining**.

**Inference Engine:**

An Inference Engine is a tool of Artificial Intelligence that is used as a component of the system to deduce new information from a **knowledge base** using **logical rules and reasoning**. The first-ever Inference Engines were a part of **expert systems** in AI. As previously stated, Inference Engines predict outcomes with the already existing pool of data, comprehensively analyzing it and using logical reasoning to predict the outcomes.

This same process would be repeated as **new facts** would be discovered and this would make the inference engine trigger additional rules for its findings. After some runs of the inference engine, it was noticed that Inference Engines works in one of the two ways, either based on goals or based on facts, which later came to be known as **forwarding chaining**and **backward chaining**.

***Forward chaining*** comes with known facts and iterates the process to find new facts while ***backward chaining starts with goals*** and works backwards to determine what conditions would be required to achieve the given goals.

## Examples regarding Inference Rules

Let’s take a look at some simple examples to help you differentiate between both sets of inference rules.

**Inference Rules**

**(i) Deductive inference rule:**

Forward Chaining: Conclude from “A” and “A implies B” to “B”.

A

A -> B

B

Example:

It is raining.

If it is raining, the street is wet.

The street is wet.

(ii) **Abductive inference rule**:

Backward Chaining: Conclude from “B” and “A implies B” to “A”.

B

A -> B

A

Example:

The street is wet.

If it is raining, the street is wet.

It is raining.

## Forward Chaining

As a data-driven as well as bottom-up logic approach, forward chaining starts from known facts and conditions, then progresses towards logical conclusion using if-then statements. Then these conditions and rules are applied to the problem until no further applicable situations are left or the limit has been reached. Forward Chaining searches for any solutions and can come up with an infinite number of possible conclusions.

The Forward-thinking approach is used in AI to help an AI agent solve logical problems by inspecting the data from the previous learning’s and then coming to a conclusion full of solutions. That’s not all, Forward Chaining might as well be used to explore the available information or answer a question or solve a problem. Forward chaining is extensively used to break down a long and complex logical approach by attaching each step once the previous one is completed. This way, ***it goes from beginning to the end with relative ease.***

### Steps for working of Forwarding Chaining

1. Step 1: We start from the already stated facts, and then, we’ll subsequently choose the ***facts that do not have any implications*** at all.
2. Step 2: Now, we will ***state those facts that*** can be inferred from available facts with satisfied premises.
3. Step 3: In step 3 we can **check** the given statement that needs to be checked and check whether ***it is satisfied with the substitution which infers all the previously stated facts. Thus we reach our goal.***

**Backward Chaining**

It is a logical process of determining unknown facts from known solutions by moving backwards from known solutions to determine the initial conditions and rules.

The Backward Chaining approach is used in AI to find the conditions and rules because of which a particular logical result or conclusion was reached. Real-life applications of Backward Chaining include use to find information regarding conclusions and solutions in reverse engineering practices as well as game theory applications.

Some other applications of Backward Chaining include automated theorem proving tools, inference engines, proof assistants and othe**r**[**artificial intelligence applications**](https://www.mygreatlearning.com/blog/business-applications-for-artificial-intelligence-and-machine-learning/).

### Steps of working for Backward Chaining

1. Step 1. In the first step, we’ll take the ***Goal Fact*** and from the goal fact, ***we’ll derive other facts*** that we’ll prove true.
2. Step 2: We’ll derive other facts from goal facts that satisfy the rules
3. Step 3: At step-3, we will extract further fact which infers from facts inferred in step 2.
4. Step 4: We’ll repeat the same until we get to a certain fact that satisfies the conditions.

## Difference between Forward Chaining and Backward Chaining

|  |  |  |
| --- | --- | --- |
| **S No** | **Forward Chaining** | **Backward Chaining** |
| 1. | It starts from known facts extract more data unit it reaches to the goal using inference rule | It starts from the goal and works backward through inference rules to find the required facts that support the goal. |
| 2. | Bottom-up Approach | Top-Down Approach |
| 3. | Known as Data-driven approach as we use given data to reach the goals | Known as goal-driven approach because we use the goal given to reach the facts that support the goals |
| 4 | Applies a breadth-first search strategy | Applies a depth-first search strategy |
| 5 | Tests for all the available rules | Only tests for certain given and selected rules |
| 6 | Suitable for planning, monitoring, control, and interpretation application. | Suitable for diagnostic, prescription, and debugging application. |
| 7. | Can generate infinite number of possible conclusions | Can generate a finite number of possible concluding facts and conditions |
| 8. | Operates in Forward Direction | Operates in Backward Direction |
| 9 | Forward Chaining is aimed for any conclusion. | Backward chaining is aimed for only the required data. |