# ARTIFICIAL INTELLIGENCE



## INTRODUCTION:-

Artificial Intelligence (AI) has grown from a small scale laboratory science into a technological and industrial success. Now we present several techniques for creating computer programs that control manufacturing processes, diagnose computer faults and human diseases, design computers, do insurance underwriting, play grand-master level chess and so on.

Definition:- There are many ways to define the field of A.I.

"AI is the study of computations that make it possible to perceive, reason, and act".

or

"The study of how to make computers do things which, at the moment, people do better".

## Goals of AI:-

From the perspective of goals, AI can be viewed as part engineering, part science.

→ The Engineering goal of AI is to solve real-world problems using AI as an armamentarium of ideas about representing knowledge, using knowledge, and assembling systems.

→ The Scientific goal of AI is to determine which ideas about representing knowledge, using knowledge, and assembling systems explain various sorts of intelligence.

## AI Problems:-

As AI research progressed and techniques for handling larger amounts of world knowledge were developed, some progress was made on the tasks. These include perception, natural language understanding and problem solving in specialized domains such as medical diagnosis and chemical analysis. In addition to these mundane tasks, many people can also perform one or may be more specialized tasks in which carefully acquired expertise is necessary. Examples of such tasks include engineering design, scientific discovery, medical diagnosis and financial planning. Some of the task domains of AI is listed below.

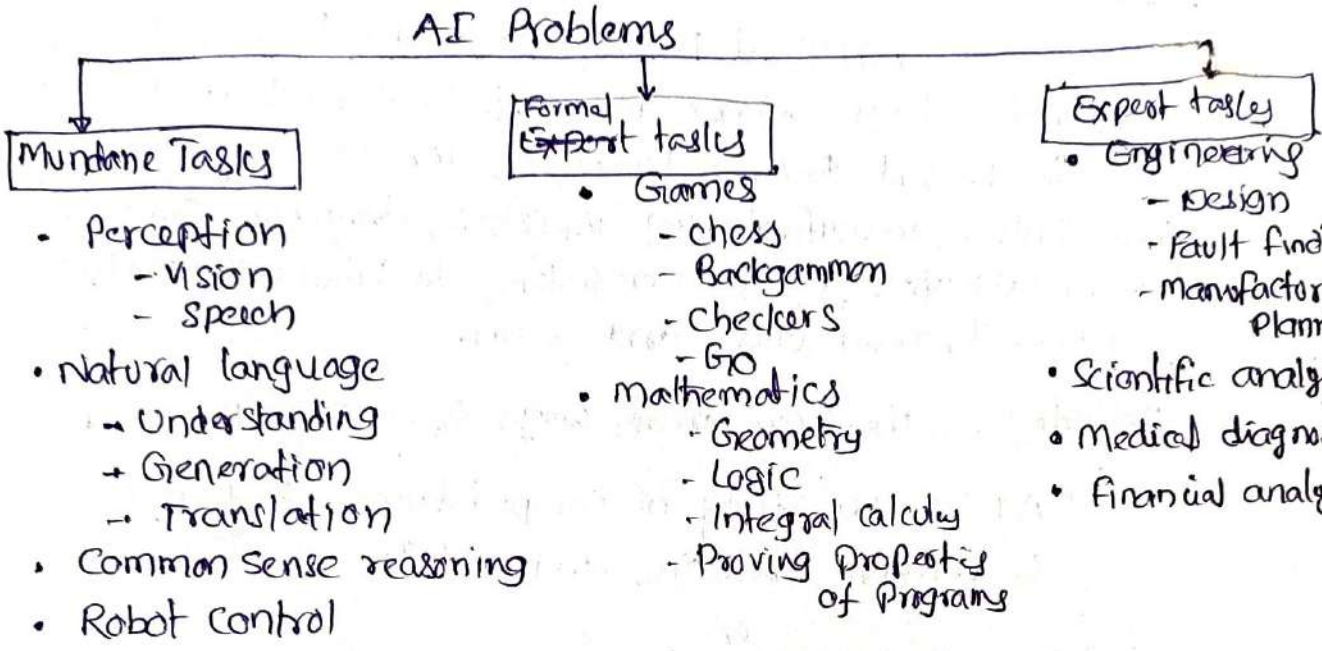



fig: Task Domains of AI.

Applications :- As the world grows more complex, we must use our material and human resources more efficiently, we need high-quality help from computers.

1. In farming, computer-controlled robots should control pests, prune trees, and selectively harvest mixed crops.
2. In manufacturing, computer-controlled robots should do the dangerous and boring assembly, inspection and maintenance jobs.
3. In medical care, computers should help practitioners with diagnosis, monitor patient's conditions, manage treatment and make beds.
4. In household work, computers should give advice on cooking and shopping, clean the floors, mow the lawn, do the laundry, and perform maintenance chores.
5. In business, computers can help us to locate pertinent information, to schedule work, to allocate resources, and to discover salient regularities in databases.
6. In engineering, computers can help us to develop more effective control strategies, to create better designs, to explain past decisions, and to identify future risks.
7. AI complements the traditional perspectives of psychology, linguistics and philosophy.
8. AI helps us to become more intelligent.
9. In schools, computers should understand why their students make mistake not just react to errors.

## THE FOUNDATIONS OF ARTIFICIAL INTELLIGENCE:

We Provide a brief History of the disciplines that contributed ideas, viewpoints and techniques to AI.

- Philosophers (going back to 400 B.C) made AI conceivable by considering the ideas that the mind is in some ways like a machine, that it operates on knowledge encoded in some internal language and that thought can be used to choose what actions to take.
- Mathematicians provided the tools to manipulate statements of logical certainty as well as uncertain, Probabilistic statements. They also set the groundwork for understanding computation and reasoning about algorithms.
- Economics formalized the Problem of making decisions that maximize the expected outcome to the decision-maker.
- Psychologists adopted the idea that humans and animals can be considered information Processing machines.
- Linguists showed that the language use fits into this model.
- Computer engineers Provided the artifacts that make AI applications possible. AI Programs tend to be large and they could not work without the great advances in speed and memory that the Computer industry has Provided
- Control theory deals with designing devices that act optimally on the basis of feedback from the environment. Initially the mathematical tools of control theory were quite different from AI, but the fields are coming closer together.

## THE HISTORY OF ARTIFICIAL INTELLIGENCE:

The History of AI has had cycles of success, misplaced optimism and resulting cutbacks in enthusiasm and funding. There have also been cycles of introducing new creative approaches and systematically refining the best ones. AI has advanced more rapidly in the Past decade because of greater use of the scientific method in experimenting with and Comparing approaches.

• The first work that is recognised as AI was done by Warren Mcculloch and Walter Pitts in 1943.

• Donald Hebb demonstrated a simple updating rule for modifying the connection strengths between neurons [Hebbian Learning] in 1949.

• Minsky and Edmonds built the first neural network in 1951

• Alan Turing introduced the Turing test, machine Learning, genetic Algorithms and reinforcement Learning.

• ~~The official birth of~~ The artificial intelligence was motivated and introduced by John McCarthy in 1956.

• AI embraced the idea of duplicating human faculties like creativity, self-improvement and language. And AI is the only field to attempt to build machines that will function autonomously in complex, changing environment.

• Newell and Simon's early success was followed up with the General Problem Solver (GPS). It was probably the first program to embody the "thinking humanly" approach.

• Herbert Gelernter constructed the Geometry Theorem Prover, which was able to prove theorems that many students of mathematics would find quite tricky in 1959

• McCarthy defined the high-level language LISP, which was to become the dominant AI programming language in 1958

• The blocks world was home to the vision Project of Huffman (1971), the vision and constraint-propagation work of David Waltz (1975), the learning theory of Patrick Winston (1970), the natural language understanding program of Terry Winograd (1972) and the planner of Scott Fahlman (1974).

• Hebb's learning methods were enhanced by Widrow (Widrow and Hoff, 1960; Widrow, 1962).

• The no of scients have been developed the various knowledge based systems like weak methods, expert systems, Constanty factors, frames during 1969-79

• In 1981, the Japanese announced the fifth generation Project, 10-year plan to build intelligent computers running PROLOG.

• John Hopfield used techniques from statistical mechanics to analyse the storage and optimization properties of networks, treating collection of nodes like collection of atoms. in 1982. It was modified by Rumelhart and mcclelland in 1986

• Recent Progress in understanding the theoretical basis for intelligence has gone hand in hand with improvements in the capabilities of real systems.

## INTELLIGENT AGENTS:- Agents and Environments

- Agent:- An agent is something that perceives and acts in an environment
- Rational Agent:- A rational agent is one that acts so as to achieve the best outcome.

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators. This idea can be shown in below.

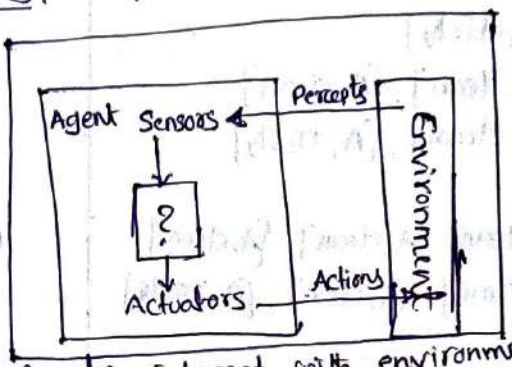



fig: Agents interact with environments through Sensors and Actuators

For example, A human Agent has eyes, ears, and other organs for Sensors and hands, legs, mouth and other body parts for actuators. A Software agent receives keystrokes, file contents and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

Agent function:- An agent's behavior is described by the agent function that maps any given percept sequence to an action.

An agent function for an artificial agent will be implemented by an agent Program. An agent function is an abstract mathematical description, the agent program is a concrete implementation, running on the agent architecture.

Consider, the vacuum-cleaner world has just two locations: Squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the sequence Square. It can choose to move left, move right, suck up the dirt or do nothing. One very simple agent function is the following: if the current square is dirty then suck, otherwise move to the other square.

Note: An intelligent agent takes the best possible action in a situation.

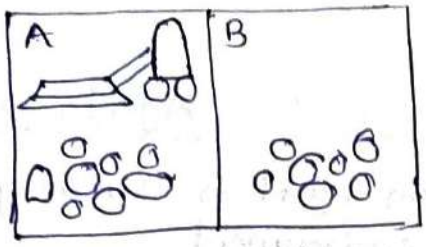



Fig: A vacuum-cleaner world with just two locations

| Percept Sequence | Action |
|---|---|
| [A, clean] | Right |
| [A, Dirty] | Suck |
| [B, clean] | Left |
| [B, dirty] | Suck |
| [A, clean], [A, clean] | Right |
| [A, clean], [A, Dirty] | Suck |
| ⋮ | ⋮ |
| [A, clean], A[clean], [A, clean] | Right |
| [A, clean], [A, clean], [A, Dirty] | Suck |
| ⋮ | ⋮ |

Fig: Partial tabulation of a simple agent function for the vacuum-cleaner world.

## Good Behavior: The concept of Rationality:

A rational agent is one that does the right thing. Conceptually speaking, every entry in the table for the agent function is filled out correctly. The right action is the one that will cause the agent to be most successful. Therefore, we will need some way to measure success.

### Performance measures:-

The Performance measure evaluates the behavior of the agent in an environment. A rational agent acts so as to maximize the expected value of the Performance measure, given the Percept-Sequence it has seen so far.

When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives. This sequence of actions causes the environment to go through a sequence of states. If the sequence is desirable, then the agent has performed well.

As a general rule, it is better to design Performance measures according to what one actually wants in the environment, rather than according to how one thinks the agent should behave.

Rationality:-

What is rational at any given time depends on four things:

- The Performance measure that defines the criterion of success.
- The agent's prior knowledge of the environment.
- The actions that the agent can perform.
- The agent's percept sequence to date.

Omniscience (infinite knowledge):- An omniscient agent knows the actual outcome of its actions and can act accordingly; but omniscience is impossible in reality. The rationality is not the same as perfection. Rationality maximizes expected performance, while perfection maximizes actual performance

Learning:- A rational agent not only to gather information, but also to learn as much as possible from what it perceives. The agent's initial configuration could reflect prior knowledge of the environment but as the agent gains experience this may be modified and augmented. An agents can improve their performance through learning.

Autonomy:- A rational agent should be autonomous, it should learn what it can to compensate for partial or incorrect prior knowledge.

## The Nature of Environments:-

A task environment specification includes the performance measure, the external environment, the actuators and the sensors.

### Specifying the task environment:-

The rationality of any agent, we have to specify the performance measure, the environment, and the agent's actuators and sensors. For the acronymically minded, we call this the PEAS (Performance, Environment, Actuators, Sensors) description. In designing the agent, the first step must always be to specify the task environment as fully as possible.

Ex: The vacuum world was a simple example: Let us consider a more complex problem: an automated taxi driver. The table below summarizes the PEAS description for the taxi's task environment.

| Agent Type | Performance measure | Environment | Actuators | Sensors |
|---|---|---|---|---|
| Taxi driver | Safe, fast, legal, comfortable trip, maximize Profits | Roads, other traffic, Pedestrains, customers | Steering, accelerator, brake, signal, horn, display | camera, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, key board |

Performance measure: Desirable qualities include getting to the correct destination; minimizing fuel consumption, minimizing the trip time and cost; minimizing violations of traffic laws and disturbances to other drivers; maximizing safety and passenger comfort; maximizing profits.

Environment: Any taxi driver must deal with a variety of roads, ranging from rural lanes and urban alleys to 12-lane freeways. The roads contain other traffic, Pedestrains, stray animals, road works, Police cars, Puddles and Potholes. The taxi must also interact with Potential and actual Passengers.

Actuators: Control over the engine through the accelerator and control over steering and braking. In addition, it will need output to a display screen or voice synthesizer to talk back to the Passengers.

Sensors: Sensors include one or more controllable TV cameras, the speedometer and the odometer. To control the vehicle properly, it should have an accelerometer, it will also need to know the mechanical state of the vehicle. A satellite global Positing System (GPS) to give it accurate Position information with respect to an electronic map, and infrared or sonar sensors to detect distances to other cars and obstacles. Finally, it will need a keyboard for the Passenger to request a destination.

In below, the basic PEAS elements for a number of additional agent types are given.

| Agent Type | Performance measure | Environment | Actuators | Sensors |
|---|---|---|---|---|
| Medical diagnosis system | Healthy Patient, minimize costs, lawsuits | Patients, hospital, staff | Display, questions, tests, diagnoses, treatments, referrals | keyboard entry of symptoms, findings, patient's answers |
| Satellite image analysis system | Correct image categorization | Downlink from orbiting satellite | Display categorization of scene | Color Pixel Arrays |
| Part-Picking robot | Percentage of parts in correct bins | conveyor belt with parts; bins | Jointed arm and Hand | Camera, joint angle sensors |
| Refinery controller | maximize purity, yield, safety | Refinery, operators | Valves, pumps, heaters, displays | Temperature, Pressure Chemical Sensors |
| Interactive English tutor | Maximize student's score on test | Set of students testing agency | Display, exercises, suggestions, corrections | keyboard entry |

fig: Examples of agent types and their PEAS Descriptions

## Properties of task environments:-

Task environments vary along several significant dimensions. They are:

1. Fully observable (accessible) vs Partially observable (inaccessible)
2. Deterministic vs Stochastic
3. Episodic vs Sequential
4. Static vs Dynamic
5. Discrete vs Continuous
6. Single agent vs multiagent

## Fully observable vs Partially observable:-

If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is fully observable. An environment might be partially observable because of noisy and inaccurate sensors. For example, an automated taxi cannot see what other drivers are thinking..

Deterministic vs Stochastic :-

If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic; otherwise, it is stochastic. In Principle, an agent need not worry about uncertainty in a fully observable, deterministic environment. If the environment is partially observable, then it could be appear to be stochastic.

Eposodic vs Sequential :-

In an eposodic task environment, the agent's experience is divided into atomic ~~action~~ episodes. Each episode consists of the agent Perceiving and then Performing a single action. Crucially, the next episode does not depend on the actions taken in previous episodes. In sequential environments, on other hand, the current decision could affect all future decisions.

~~Static~~ vs ~~Dynamic~~ :- Descrete vs Continuous:

The discrete/continuous distinction can be applied to the state of the environment, to the way time is handled, and to the percepts and actions of the agent.

For ex, chess has a discrete set of percepts and actions. Taxi driving is a continuous state and continuous-time problem.

Single agent vs multiagent :- for Static vs Dynamic property, you will see after this topic

The distinction between single-agent and multiagent environments may seem simple enough. For example, an agent solving a crossword puzzle by itself is clearly in a single-agent environment, whereas an agent playing chess is in a two-agent environment.

Chess is a competitive multi-agent environment, whereas taxi-driving agent in cooperative multi-agent environment.

The table below lists the properties of a number of familiar environments. Note that the answers are not always cut and dried. Some other answers in the table depend on how the task environment is defined. We have listed the medical-diagnosis task as single-agent because the disease process in a patient is not profitably

modeled as an agent. but a medical-diagnosis system might also have to deal with recalcitrant Patients and skeptical Staff.

| Task Environment | observable | Deterministic | Episodic | static | Discrete | Agents |
|---|---|---|---|---|---|---|
| Crossword Puzzle | Fully | Deterministic | Sequential | Static | Discrete | Single |
| Chess with a clock | Fully | Strategic | Sequential | Semi dynamic | Discrete | multi |
| Poker | Partially | Strategic | Sequential | Static | Discrete | multi |
| Backgammon | Fully | Stochastic | Sequential | Static | Discrete | multi |
| Taxi driving | Partially | Stochastic | sequential | Dynamic | continuous | multi- |
| medical diagnosis | Partaly | Stochastic | Sequential | Dynamic | continuous | Single |
| Image-Analysis | Fully | Deterministic | Episodic | Semi | Continuous | single |
| Part-Picking Robot | Partially | Stochastic | Episodic | Dynamic | continuous | single |
| Refinery Controller | Partaly | Stochastic | Sequential | Dynamic | continuous | Single |
| Interactive English tutor | Partially | Stochastic | Sequential | Dynamic | Discrete | multi |

fig: Examples of task environments and their Characteristics

Note: 1. If the environment is deterministic except for the actions of other agents, we say that the environment is strategic.

2. If the environment itself does not change with the passage of time but the agent's performance score does, then we say that environment is Semidynamic.

## THE STRUCTURE OF AGENTS :-

The Job of AI is to design the agent Program that implements the agent function mapping Percepts to actions. We assume this Program will run on some sort of computing device with physical sensors and actuators - we call this the architecture.

agent = architecture + Program

In general, the architecture makes the Percepts from the sensors available to the Program, runs the Program, and feeds the Program's action choices to actuators as they are generated.

## agent Programs :-

The agent Program implements the agent function. There exists a variety of basic agent-Program designs, reflecting the kind of information made explicit and used in the decision process. The designs vary in efficiency, compactness and flexibility. The appropriate design of the agent Program depends on the nature of environment. Notice the difference between the agent Program, which takes the current Percept as input, and the agent function which takes the entire Percept history.

The figure below shows a rather trivial agent Program that keeps track of the Percept sequence and then uses it to index into a table of actions to decide what to do. Let 'P' be the set of Possible Percepts and let 'T' be the lifetime of the agent. The lookup table will contain $\sum_{t=1}^{T} |P|^t$ entries.

```
function TABLE-DRIVEN-AGENT returns an action
    static: Percepts, a sequence, initially empty
            table, a table of actions, indexed by Percept
              sequences, initially fully specified
    append Percept to the end of Percepts
    action ← LOOKUP(Percepts, table)
    return action.
```

## BASIC KINDS OF AGENT PROGRAM :-

There are four basic kinds of agent Program that embody the principles underlying almost all intelligent systems.

1. Simple reflex agents
2. Model-based reflex agents
3. Goal-based agents
4. Utility-based agents

### 1. Simple reflex agents :-

These agents select actions on the basis of the current Percept, ignoring the rest of the Percept history. For example, the vacuum agent whose agent function is tabulated is a simple reflex agent, because its decision is based only on the current location and on whether that contains dirt. An agent Program for this agent shown below.

```
-VACUUM-
Function REFLEX-AGENT([Location, status]) returns an action
    if status = Dirty then return Suck.
    else if location = A then return Right.
    else if location = B then return Left.
```

Fig: Agent Program for a simple-reflex agent.

The above Program is specific to one particular Vacuum environment. A more general and flexible approach is first to build a general-Purpose interpreter for condition-action rules and then to create rule sets for specific task environments.

Structure of agent Program:- It gives the structure of the general Program in schematic form, showing how the condition-action rules allow the agent to make the connection from Percept to action. We use rectangles to denote the current internal state of the agent's decision Process and ovals to represent the background information used in the Process.

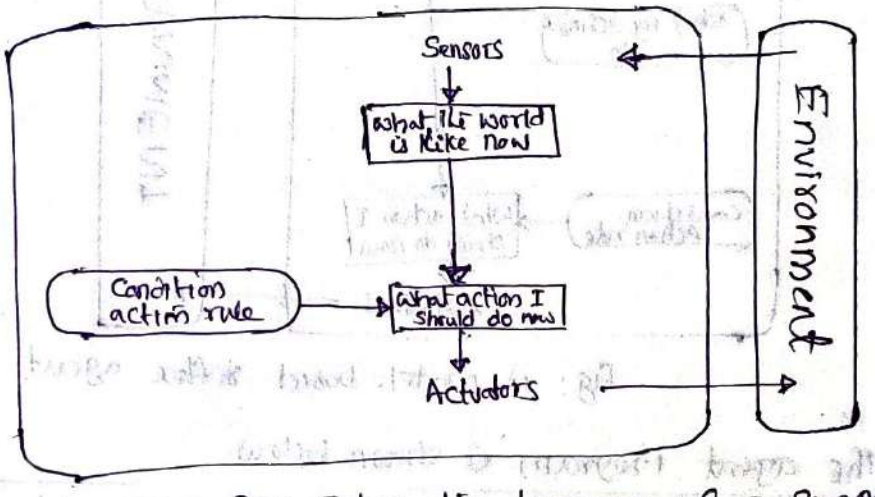



Fig: Schematic diagram of a simple reflex agent

The agent Program is very simple. The Interpret-input function generates an abstracted description of the current state from the Percept, and the RULE-MATCH function returns the first rule in the set of rules that matches the given state description.

```
function SIMPLE-REFLEX-AGENT(Percept) returns an action
    static: rules, a set of condition-action rules
        state ← INTERPRET-INPUT(Percept)
        rule ← RULE-MATCH(state, rules)
        action ← RULE-MATCH ACTION[rule]
    return action
```

Note: The many possibilities reduces from $4^T$ to just 4 in simple reflex agents

## Model-based reflex agents:-

The agent should maintain some sort of internal state that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state. Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program First, we need some information about how the world evolves independently of the agent. Second, we need some information about how the agent's own actions affect the world

The figure below gives the structure of the reflex agent with internal state, showing how the current percept is combined with the old internal state to generate the updated description of the current state.

Know about how the world works → is model of the world.

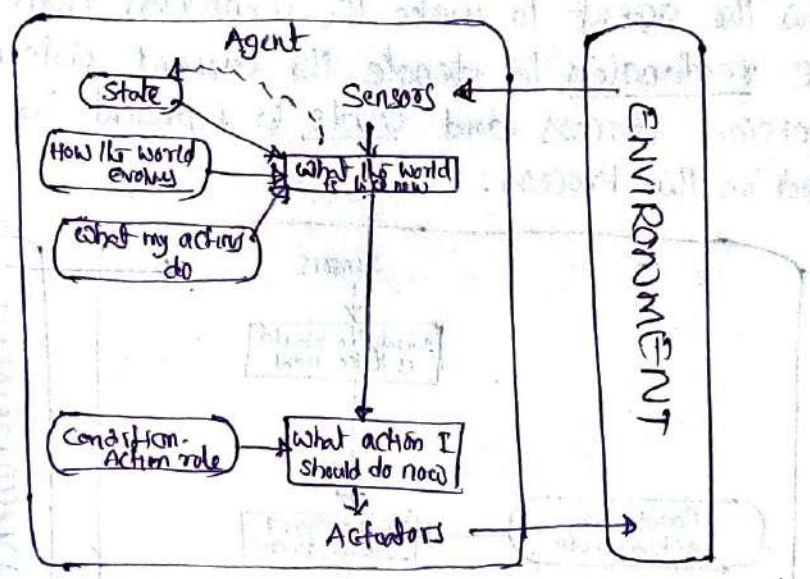



fig: A model-based reflex agent

The agent program is shown below.

```
function REFLEX-AGENT-WITH-STATE(Percept) returns an action
    static: state, a description of the current world state
            rules, a set of condition-action rules
            action, the most recent action, initially none
    state ← UPDATE-STATE(state, action, Percept)
    rule ← RULE-MATCH(state, rules)
    action ← RULE-ACTION[rule]
    return action
```

## Goal-based Agents:-

Knowing about the current state of the environment is not always enough to decide what to do, The agent needs some sort of goal information that describes situations that are desirable. The agent program can combine this with information about the results of possible actions in order to choose actions that achieve the goal. The figure below shows the goal-based agents structure

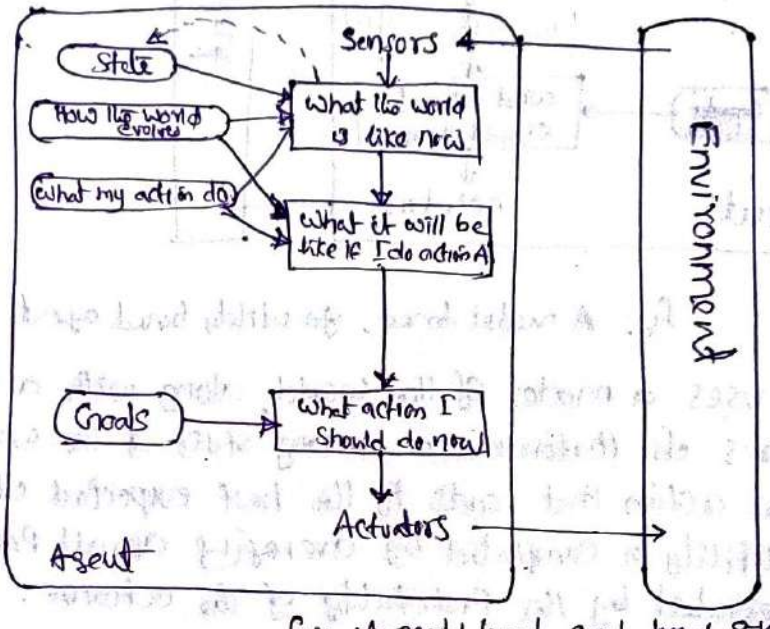



fig: A model based, goal based agent

The goal based agents appears less efficient, it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified.

## Utility-based agents:-

Goals alone are not really enough to generate high-quality behavior in most environments. Goals just provide a crude binary distinction between "happy" and "unhappy" states, whereas a more general performance measure should allow a comparision of different world states according to exactly how happy they would make the agent if they could be achieved.

A utility function maps a state onto a real number, which describes the associated degree of happiness. A complete specification of the utility function allows rational decisions in two kinds of cases where goals are inadequate. First, when there are conflicting goals, only some of which can be achieved, the utility function specifies the appropriate tradeoff. Second, when there are several goals that the agent can aim for, none of which can be achieved with certainty utility provides a way in which the likelihood of success can be weighed

up against the importance of the goals.

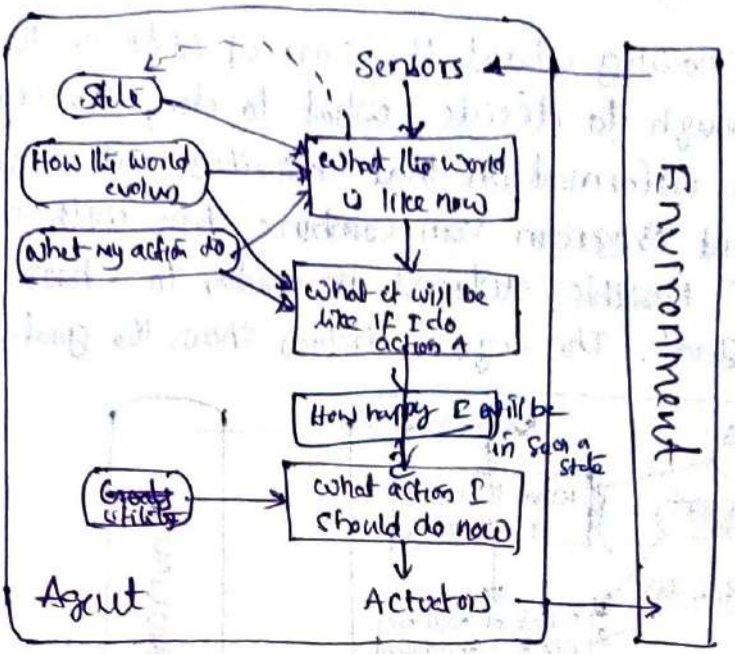



fig: A model based, go utility based agent

It uses a model of the world, along with a utility function that measures its Performance among states of the world. Then it chooses the action that leads to the best expected utility. Where expected utility is computed by averaging overall possible outcome states, weighted by the probability of the outcome.

## Learning Agents:

A learning agent can be divided into four conceptual components, as shown in figure below.

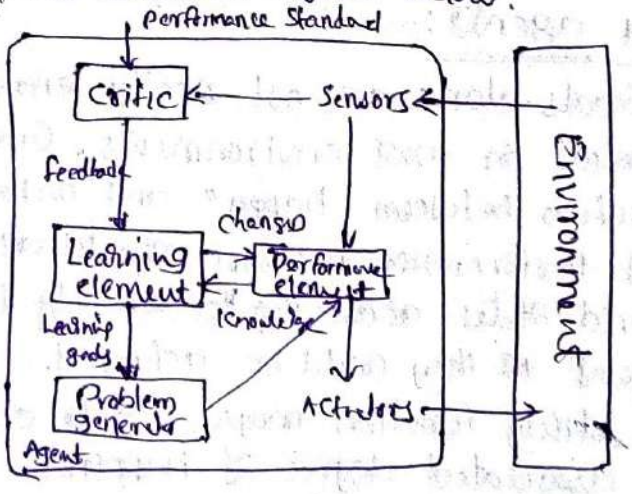



fig: A general model of learning agents

Learning: Learning in intelligent agents can be summarized as a process of modification of each component of the agent to bring the components into closer agreement with the available feedback information, thereby improving the overall performance of the agent.

The most important distinction between the learning element, which is responsible for making improvements and the Performance element, which is responsible for selecting external actions. The learning element uses feedback from the critic on how the agent is doing and determines how the Performance element should be modified to do better in the future. The critic tells the learning element how well the agent is doing w.r.to a fixed Performance standard.

The last component of the learning agent is the Problem generator. It is responsible for suggesting actions that will lead to new and informative experiences.

Static vs Dynamic :-

If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent otherwise, it is static. Static environments are easy to deal with because the agent need not keep looking at the world, while it is deciding on an action, nor need it worry about the Passage of time. Dynamic environments are continuously asking the agent what it wants to do. Taxi driving is clearly dynamic. Crossword Puzzles are static

* This Property is not included in the task environment properties given

## Problem-Solving Agents :-

Problem Solving agents decide what to do by finding Sequences of actions that lead to desirable states. Problem Solving agent is one kind of goal-based agent.

If first formulates a goal and a Problem, Searches for a sequence of actions that would Solve the Problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over. Note that when it is executing the sequence it ignores its Percepts; it assumes that the solution it has found will always work.

Problem formulation: It is the Process of deciding what actions and States to consider, given a goal.

Search: The Process of looking for a sequence is called Search.

We have a simple "formulate, search, execute" design for the agent, as shown in below.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(Percept)
                                returns an action
  inputs : Percept, a Percept
  static : seq, an action sequence, initially empty
           state, some description of current world state
           goal, a goal, initially null
           problem, a problem formulation
  state ← UPDATE-STATE(state, Percept)
  if seq is empty then do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← FIRST(seq)
      seq ← REST(seq)
  return action
```

## Well-defined Problems and Solutions:

A Problem can be defined formally by FOUR Components

1. The initial state that the agent starts in.
2. The description of possible actions available to the agent. The most common formulation uses a successor function.

Together, the initial state and Successor function implicitly define the statespace of the Problem, the set of all states reachable from the initial state. The state space forms a graph in which the nodes are states and the arcs between nodes are actions. A path in the state space is a sequence of states connected by a sequence of actions

3. The goal test, which determines whether a given state is a goal state.
4. A path cost function that assigns a numeric cost to each path. The Problem-solving agent chooses a cost function that reflects its own performance measure.

NOTE: A solution to a Problem is a Path from initial state to a goal state.

## Example Problems:-

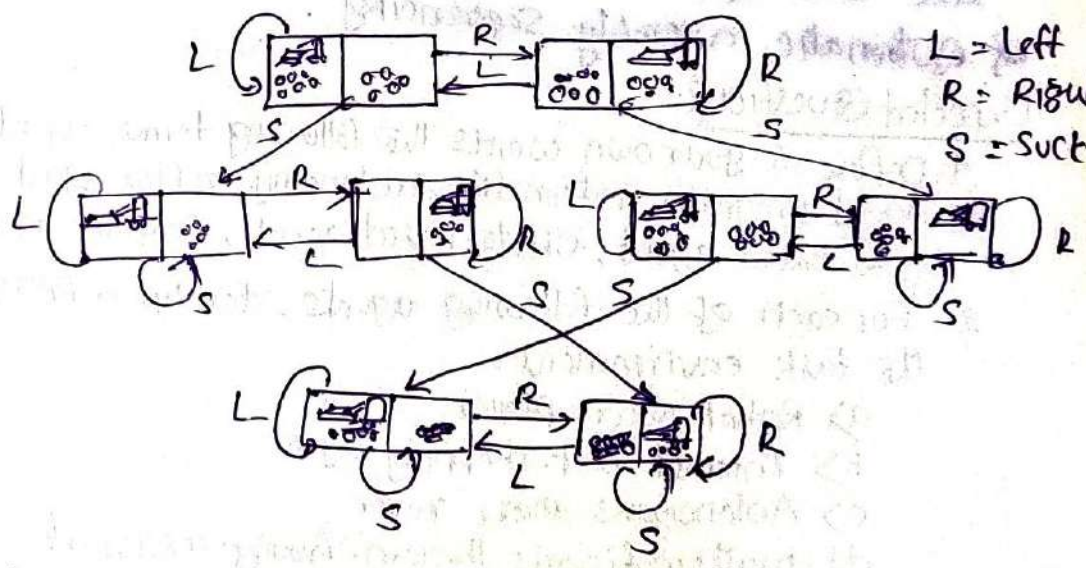
The Problem-Solving approach has been applied to a vast array of task environments. We list some of the best known Problems - toy and real-world Problems.

### Toy Problems:-

The Vacuum World can be formulated as a Problem as follows:

1. States:- The agent is in one of two locations, each of which might or might not contain dirt. Thus there are $2 \times 2^2 = 8$ Possible world states
2. initial state:- Any state can be designated as the initial state
3. Successor function: This generates the legal states that result from trying the three actions (Left, right and suck).
4. Goal test: This checks whether all the squares are clean
5. Path cost: Each step costs 1, so the Path cost is the no. of steps in the Path.

The state space for the Vacuum world:




Note: A large environment with 'n' locations has $n \times 2^n$ states.

other Problems are the 8-puzzle, 8-queens Problem, 15 puzzle Problem etc.

## Real-world Problems:

Route-finding Algorithms are used in a variety of applications, such as routing in computer networks, military operations planning and airline travel planning systems. Consider a simplified example of an airline travel problem specified as follows

- States - Each is represented by a location and current time.
- Initial state - This is specified by the problem.
- Successor functions - This returns the states resulting from taking any scheduled flight, leaving later than the current time plus the within-airport transit time, from the current airport to another.
- Goal test - Are we at the destination by some prespecified time?
- Path cost - This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

The other problems from this category are Touring problems like traveling salesperson problem, VLSI layout, Robot navigation & Automatic assembly sequencing.

## Expected Questions:-

1. Define in your own words the following terms: agent, agent function, agent program, rationality, autonomy, reflex agent, model-based agent, goal-based agent, utility based agent, learning agent.
2. For each of the following agents, develop a PEAS description of the task environment: 4x4 = 16
   a) Robot soccer player
   b) Internet book-shopping agent
   c) Autonomous mars rover
   d) mathematician's theorem-proving assistant.
3. For each of the agent types listed above, characterize the environment according to the properties and select a suitable agent design.
4. List and explain the basic kinds of agent programs.

4x4 = 16

# Problems, Problem Spaces, and Search

To build a system to solve a particular problem, we need to do four things:

1. Define the problem precisely.
2. Analyze the problem.
3. Isolate and represent the task knowledge that is necessary to solve the problem
4. Choose the best problem-solving techniques and apply them to the particular problem.

## Defining the Problem as a State Space Search

The state space representation forms the basis of most of the AI methods. Search is a very important process in the solution of hard problems for which no more direct techniques are available.

In order to show the generality of the state space representation, we use it to describe a problem very different from 'chess'

Example: A Water jug Problem:

Statement: There are two jugs with a 4-gall capacity 4-gallon and 3-gallon. Neither has any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug?

Description:

The state space for this problem can be described as the set of ordered pairs of integers $(x, y)$, such that $x = 0, 1, 2, 3$ or 4 and $y = 0, 1, 2, 3$; $x$ represents the no. of gallons of water in the 4-gallon jug, and 'y' represents the quantity of water in the 3-gallon jug.

∴ The start state is (0,0)

∴ The goal state is (2,n)

After specifying the initial states as well as goal states, it is better to specify a set of rules that describe the actions available. The Problem can be solved by using rules, in combination with an appropriate control strategy, to move through the Problem space until a path from an initial state to a goal state is found.

Production Rules:

To solve the water jug problem, in addition to the Problem description, is a control structure that loops through a simple cycle in which some rule whose left side matches the current state is chosen, the appropriate change to the state is made as described in the corresponding right side, and the resulting state is checked to see if it corresponds to a goal state.

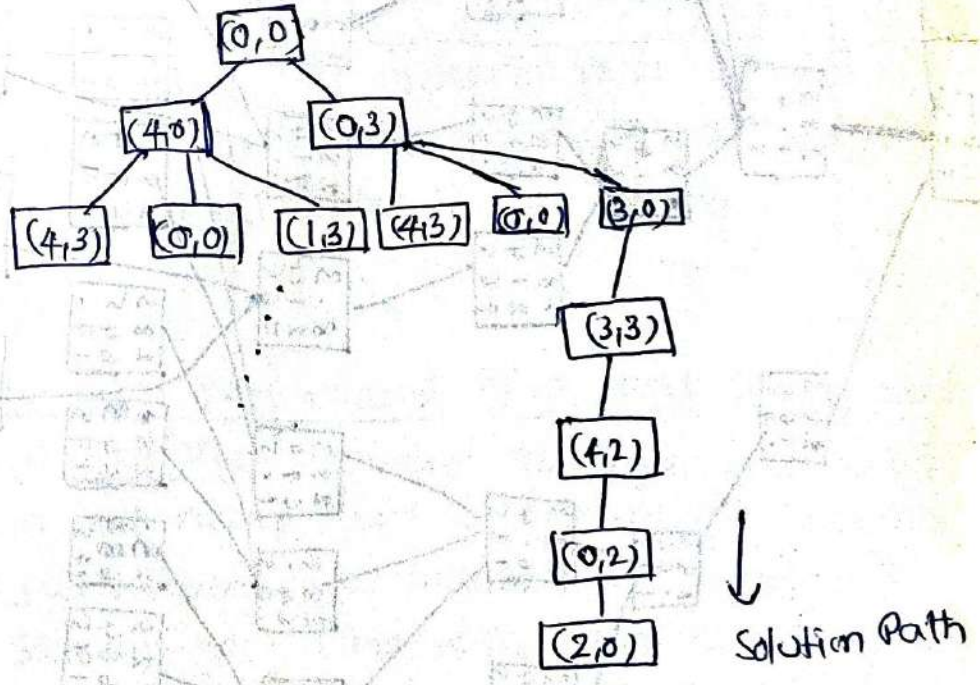
1. $(x, y) \longrightarrow (4, y)$ if $x < 4$ — Fill the 4-gallon jug
2. $(x, y) \longrightarrow (x, 3)$ if $y < 3$ — Fill the 3-gallon jug
3. $(x, y) \longrightarrow (x-d, y)$ if $x > 0$ — Pour some water out of the 4-gallon jug
4. $(x, y) \longrightarrow (x, y-d)$ if $y > 0$ — Pour some water out of the 3-gallon jug
5. $(x, y) \longrightarrow (0, y)$ if $x > 0$ — Empty the 4-gallon jug on the ground
6. $(x, y) \longrightarrow (x, 0)$ if $y > 0$ — Empty the 3-gallon jug on ground
7. $(x, y) \longrightarrow (4, y-(4-x))$ if $x+y \geq 4$ and $y > 0$ — Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full.
8. $(x, y) \longrightarrow (x-(3-y), 3)$ if $x+y \geq 3$ and $x > 0$ — Pour water from the 4-gallon jug into the 3-gallon until the 3-gallon jug is full
9. $(x, y) \longrightarrow (x+y, 0)$ if $x+y \leq 4$ and $y > 0$ — Pour all the water from the 3-gallon jug into 4-gallon jug
10. $(x, y) \longrightarrow (0, x+y)$ if $x+y \leq 3$ and $x > 0$ — Pour all the water from the 4-gallon jug into 3-gallon jug

## Solution:

For the water jug problem, there are several sequences of operators that solve the problem.

| Gallons in the 4-gallon jug | Gallons in the 3-gallon jug | Rule applied |
|---|---|---|
| 0 | 0 | |
| | | 2 |
| 0 | 3 | |
| | | 9 |
| 3 | 0 | |
| | | 2 |
| 3 | 3 | |
| | | 7 |
| 4 | 2 | |
| | | 5 |
| 0 | 2 | |
| | | 9 |
| 2 | 0 | |

Search tree




## * State space representation for 8-puzzle Problem

- Start state

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| 7 | · | 5 |

- goal state

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | · | 4 |
| 7 | 6 | 5 |

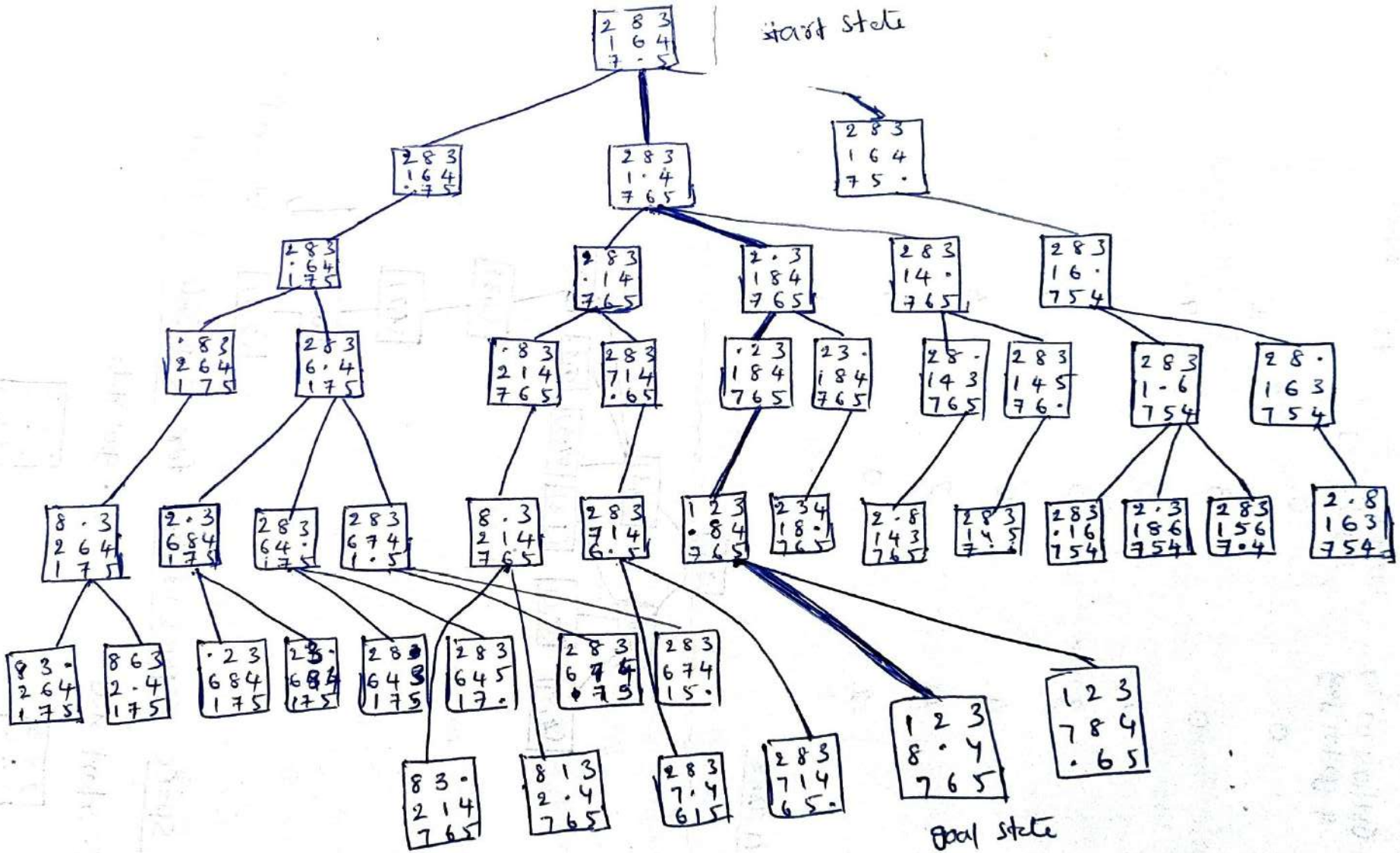



fig: A search tree for the 8-puzzle.

# Production Systems

Search is useful to structure AI programs in a way that facilitates describing and performing the search process. Production systems provide such structures.

A Production System consists of:

i) A set of rules, each consisting of a leftside that determines the applicability of the rule and a rightside that describes the operation to be performed, if the rule is applied.

ii) One or more knowledge/databases that contain whatever information is appropriate for the particular task.

iii) A control strategy that specifies the order in which the rules will be compared to the database and a way of resolving the conflicts that arise when several rules match at once.

iv) A rule applier.

## Control Strategies

It specifies the order in which the rule will be compared to the database and a way of resolving the conflicts when several rules match at once.

### requirements:

- The first requirement of a good control strategy is that it cause motion:- control strategies that do not cause motion will never lead to a solution. Consider in the water jug problem, we implement the simple control strategy of starting each time at the top of the list of rules and choosing the first applicable one. If we did that, we would never solve the problem.

- The second requirement of a good control strategy is that it be systematic: The requirement that a control strategy corresponds to the need for global motion as well as for local motion. If the control strategy is not systematic,

We may explore a particular useless sequence of operators several times before we finally find a solution.

- For the water jug problem, most control strategies that cause motion and are systematic will lead to an answer. For some problems like 'The traveling Salesman Problem' would take more time. So, we need a new control strategy, called branch-and-bound. Using this technique, we are still guaranteed to find the shortest path but it still requires exponential time and the exact amount of time it saves for a particular problem depends on the order in which the paths are explored. But it is still inadequate for solving large problems.

In order to solve many hard problems efficiently, we introduce the heuristic search.

NOTE:

1. Basic Production system languages are OPS5 and ACT
2. 'SOAR' is a general Problem-solving architecture.

# PROBLEM Characteristics

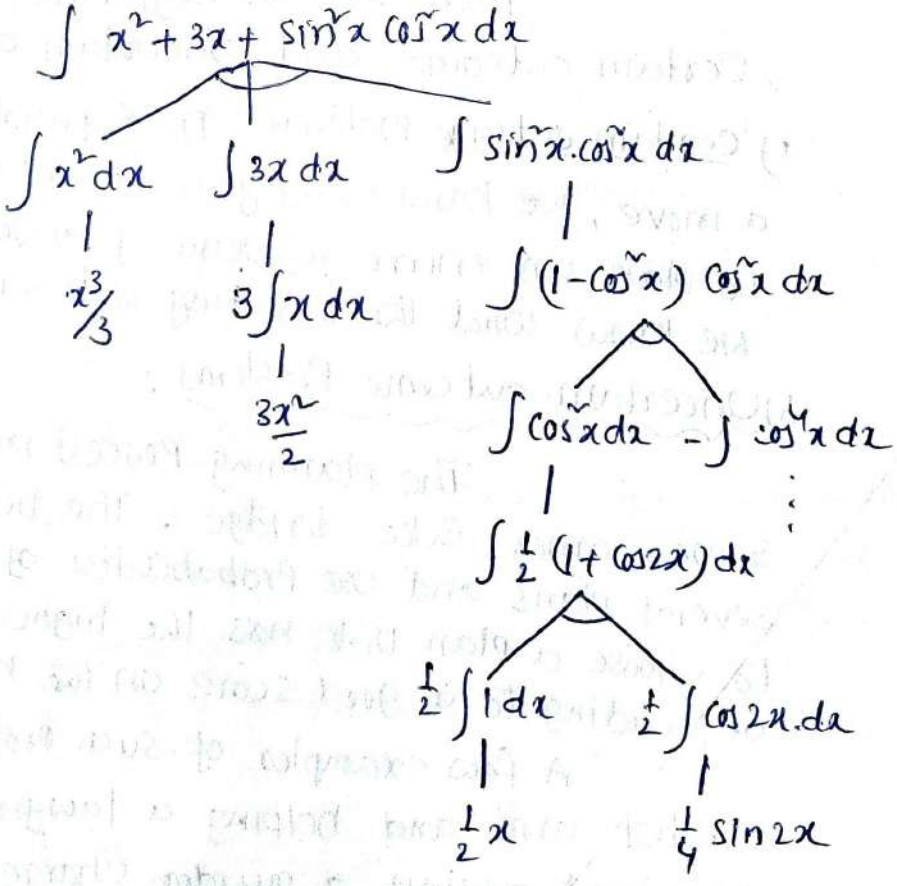
In order to choose the most appropriate method for a particular problem, it is necessary to analyze the problem along several key dimensions.

1. Is the Problem Decomposable?
2. Can Solution steps be ignored or undone?
3. Is the universe Predictable?
4. Is a good Solution absolute or Relative?
5. Is the Solution a state or a Path?
6. What is the role of Knowledge?
7. Does the task require interaction with a Person?

## 1) Is the Problem Decomposable?

We can solve the large Problems easily by breaking it down into some sub problems, each of which we can then solve by using a small collection of specific rules.

ex:- Symbolic Enlegration



2) Can Solution Steps be ignored or Undone

There are three important classes of Problems

a) Ignorable Problems, in which solution steps can be ignored
ex: Theorem Proving

Ignorable Problems can be solved using a simple control structure that never backtracks. Such a control structure is easy to implement.

b) Recoverable Problems, in which solution steps can be undone
ex: 8-puzzle

Recoverable Problems can be solved by a slightly more complicated control strategy that does make mistakes. Backtracking will be necessary to recover from such mistakes.

c) Irrecoverable Problems, in which solution steps cannot be undone. ex: Chess

Some irrecoverable Problems can be solved by recoverable style methods used in a Planning Process.

3) Is the Universe Predictable?

There are two different type Problems, namely Certain-outcome and uncertain outcome Problems.

a) Certain-outcome Problems: In 8-puzzle, Every time we make a move, we know exactly what will happen. ie It is possible to plan an entire sequence of moves and be confident that we know what the resulting state will be.

b) Uncertain-outcome Problems:

The planning process may not be possible in some games, like bridge. The best way is investigate several plans and use probabilities of the various outcomes to choose a plan that has the highest estimated probability of leading to a good score on the hand.

A few examples of such Problems are controlling a robot arm and helping a lawyer decide how to defend his client against a murder charge.

## 4) Is a good solution absolute or Relative

Depending upon the solution, the Problems can be classified into two, namely any-Path and best-Path Problems

### i) Any-Path Problems:

In this type of problems, we may get more than one solution. Any Path (solution) will lead to the answer.

so, consider the database

1. Marcus was a man
2. Marcus was a Pompeian
3. Marcus was born in 40 A.D
4. All men are mortal
5. All pompeians died when the volcano erupted in 79 A.D
6. No mortal lives longer than 150 years
7. It is now 1991 A.D

Suppose we ask the question "Is marcus alive?", we can easily derive an answer to the question.

Solution 1

| | |
|---|---|
| axiom 1 | 1. Marcus was a man |
| axiom 4 | 4. All men are mortal |
| 1, 4 | 8. Marcus is mortal |
| axiom 3 | 3. Marcus was born in 40 A.D |
| axiom 7 | 7. It is now 1991 A.D |
| 3, 7 | 9. Marcus age is 1951 years |
| axiom 6 | 6. No mortal lives longer than 150 years |
| 8, 6, 9 | 10. Marcus is dead |

Solution 2

| | |
|---|---|
| axiom 7 | 7. It is now 1991 A.D |
| axiom 5 | 5. All pompeians died in 79 A.D |
| 7, 5 | 11. All pompeians are dead now |
| axiom 2 | 2. Marcus was a Pompeian |
| 11, 2 | 12. Marcus is dead. |

### ii) Best-Path Problems:

In this type of problems, our goal is to find the best solution (or) best path among all the remaining solutions or paths.

The best example is 'traveling Salesman Problem'.

2) Can Solution Steps be ignored or Undone

5) Is the Solution a State or a Path?

Consider the ambiguity sentence,
"I will meat you at diamond" ~~mtg~~. In this sentence the word "diamond" might have the following set of meanings:

i) A geometrical shape with four equal sides
ii) A baseball field
iii) An extremely hard and valuable gemstone.

Because of the interaction among the interpretations of the constituents of this sentence, some search may be required to find a complete interpretation for the sentence.

The natural language understanding problem solution is a state of the world.

For water jug problems, whose solution is a Path to a state.

6) What is the Role of Knowledge?

The Problems, chess and News paper story understanding illustrate the difference between problems for which a lot of knowledge is important only to constrain the search for a solution and those for which a lot of knowledge is required even to be able to recognize a solution.

7) Does the task require interaction with a Person?

Sometimes it is useful to program computers to solve problems in ways that the majority of people would not be able to understand. This is fine, if the level of interaction between the computer and its human users is Problem-in Solution-out. But increasingly we are building programs that require intermediate interaction with people, both to provide additional input to the program and to provide additional reassurance to the user. There are two types of problems

i) Solitary, in which the computer is given a problem description and produces an answer with no intermediate communication and with no demand for an explanation of reasoning process.

ii) Conversational, in which there is intermediate communication between a person and the computer.

# UNIT-II

## SEARCHING:-

Having formulated some Problems, we now need to Solve them. This is done by a search through the state space. In general Every search process can be viewed as a traversal of a tree structure in which each node represents a Problem state and each arc represents a relationship between the states represented by the nodes it connects. In general, we may have a search ~~tree~~ graph rather than a search tree, when the same state can be reached from multiple paths.

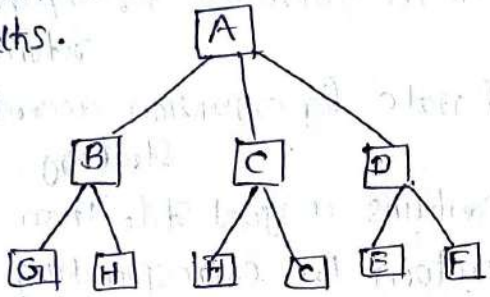



fig: Search tree

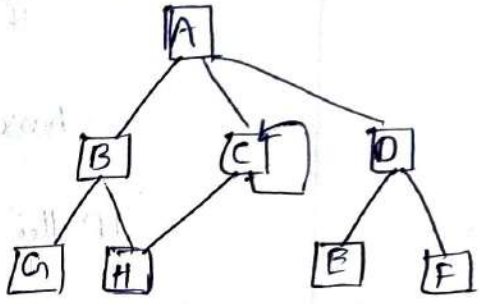
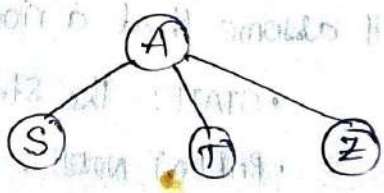



fig: Search graph.

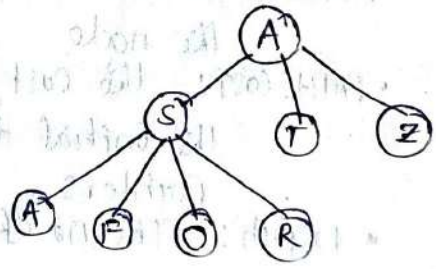
Figure below shows some of expansions in the search tree for finding a route from A to B. The root of the search tree is a search node corresponding to the initial state, In(A).

(a) Initial state


(b) After expanding A




(c) After expanding S




The first step is to test whether this is a goal state. Clearly it is not, but it is important to check so that we can solve trick problems. Because this is not a goal state, we need to consider some other states. This is done by expanding the current state, ie applying the successor function to the current state.

thereby generating a new set of states. We continue choosing, testing and expanding until either a solution is found or there are no states to be expanded. The choice of which state to expand is determined by the search strategy. The general tree-search algorithm is described in below.

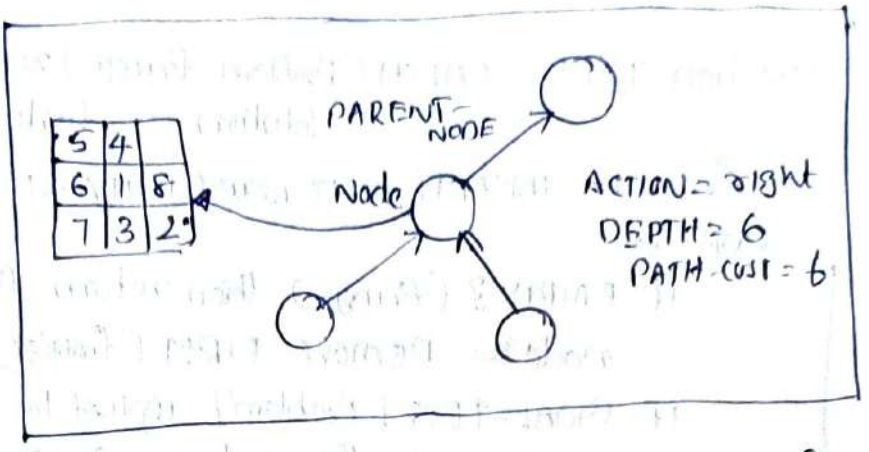
```
function TREE-SEARCH (Problem, Strategy) returns a
                                    solution, or failure
    initialize the search tree using the initial state of
                                              Problem
    Loop do
        if there are no candidates for expansion then
                                        return failure
        choose a leaf node for expansion according to
                                        strategy.
        If the node contains a goal state then
                        return the corresponding solution
        else
            expand the node and add the resulting nodes
                        to the search tree.
```

fig: general tree-search Algorithm

There are many ways to represent nodes, but we will assume that a node is a data structure with five components.

- STATE:- the state in the state space to which the node corresponds.
- PARENT-NODE:- the node in the search tree that generated this node
- Action:- the action that was applied to the parent to generate the node
- PATH-COST:- the cost, denoted by $g(n)$, of the path from the initial state to the node, as indicated by parent pointers
- Depth:- The no. of steps along the path from the initial state

It is important to remember the distinction between nodes and states. A node is a bookkeeping data structure used to represent the search tree. A state corresponds to a configuration of the world. Thus, nodes are on particular paths, as defined by PARENT-NODE pointers, whereas states are not. The node data structure is shown below.




We also need to represent the collection of nodes that have been generated but not yet expanded, this collection is called the fringe. Each element of the fringe is a leaf node, that is, a node with no successors in the tree. The simplest representation of the fringe would be a set of nodes. The search strategy then would be a function that selects the next node to be expanded from this set. Therefore, we will assume that the collection of nodes is implemented as a queue. The operations on a queue are as follows:

- MAKE-QUEUE (element, ...) creates a queue with the given element(s).
- EMPTY? (Queue) returns true only if there are no more elements in the queue
- FIRST (queue) returns the first element of the queue
- REMOVE-FIRST (Queue) returns FIRST (Queue) and removes it from the queue.
- INSERT (element, queue) inserts an element into the queue and returns the resulting queue.
- INSERT-ALL (elements, queue) inserts a set of elements into the queue and returns the resulting queue.

With this information, we can write more formal version of the general tree-search Algorithm shown in below. Note that a single, general TREE-SEARCH algorithm can be used to solve any problems; specific variants of the algorithm embody different strategies.

```
function TREE-SEARCH(Problem, fringe) return a a
                                    Solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[Problem]), fringe)
loop do
    if EMPTY?(fringe) then return failure
      node ← REMOVE-FIRST(fringe)
    if GOAL-TEST[Problem] applied to STATE[node]
          succeeds then return SOLUTION(node)
  fringe ← INSERT-ALL(EXPAND(node, Problem), fringe)
```

```
function EXPAND(node, Problem) returns a set of nodes
        successors ← the empty set
  for each <action, result> in Successor-FN[Problem]
                          (State[node]) do
        S ← a new NODE
        STATE[S] ← result
        Parent-node[S] ← node
          ACTION[S] ← action
        PATH-cost[S] ← PATH-cost[node] + STEP-cost(node, action, S)
          DEPTH[S] ← DEPTH[node] + 1
        add S to successors
  return successors
```

Measuring Problem-Solving Performance :-

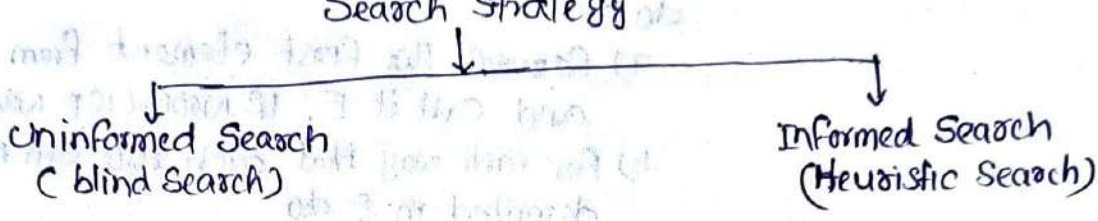
We will evaluate an algorithms Performance in four ways.

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

Complexity depends on b, the branching factor in the State Space, and d, the depth of the shallowest solution.

## Search Strategies:-

The Search Strategies are distinguished by the order in which nodes are expanded. Basically, there are two kinds of Search Strategies.




Uninformed Search:- The term means that they have no additional information about States beyond that provided in the Problem definition All they can do is generate successors and distinguish a goal state from a non goal state.

Examples: Breadth-first Search, uniform-Cost Search, Depth-first Search, Depth-limited search, Iterative deepening depth first search and Bidirectional Search.

Informed Search:- Strategies that know whether one non goal state is "more promising" than another are called informed Search or heuristic Search Strategies

Examples:- Best-first Search, Greedy best-first Search, A* Search memory bounded heuristic Search, memory-bounded A*, Simplified Memory bounded A* (SMA*) and Local Search Algorithms – Hill climbing Search, Local beam Search Genetic Algorithms etc.

### Breadth-first Search:-

Breadth-first Search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors and so on. In general all the nodes are expanded at a given depth in the Search tree before any nodes at the next level are expanded.

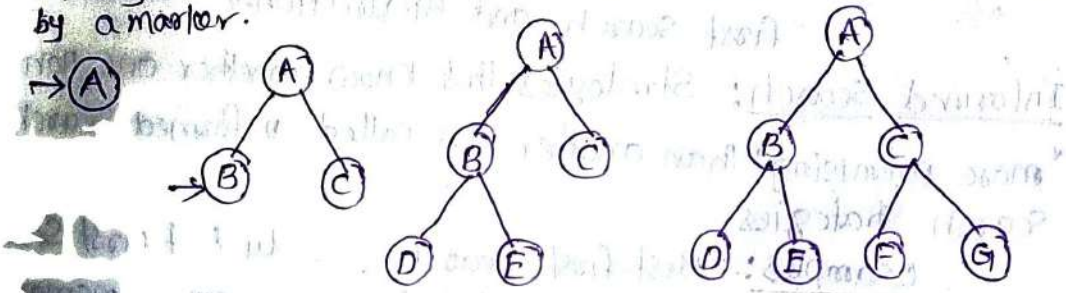
Breadth-first Search can be implemented by calling TREE SEARCH with an empty fringe that is a first-in-first-out (FIFO) Queue, assuring that the nodes that are visited first will be expanded first. The FIFO Queue puts all newly generated successors

at the end of the queue, which means the shallow nodes are expanded before deeper nodes.

Algorithm: Breadth-First Search

1. Create a variable called NODE-LIST and set it to the initial state
2. Until a goal state is found or NODE-LIST is empty do
   a) Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty, 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 a goal state, quit and return this state
      iii) Otherwise, add the new state to the end of NODE-LIST.

The figure below shows the progress of the search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.




Advantages:- 1. Breadth-first search will not get trapped exploring a blind alley.

2. If there is a solution, then Breadth-first search is guaranteed to find it. If there are multiple solutions, then a minimal solution will be found.

Disadvantages

1. Breadth-first search requires more memory since all the nodes generated so far must be stored
2. Breadth-first search in which all parts of the tree must be examined to level 'n' before any nodes on level n+1 can be examined.

NOTE: Breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node.

## Time and space Complexity :-

We consider a hypothetical state space where every state has 'b' successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of $b^2$ at the second level. Each of these generates b more nodes, yielding $b^3$ nodes at the third level, and so on.

Now suppose that the solution is at depth 'd'. In the worst case, we would expand all but the last node at level 'd' generating $b^{d+1} - b$ nodes at level d+1. Then the total no. of nodes generated is

$$b + b^2 + b^3 + \cdots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

Every node that is generated must remain in memory, because it is either part of the fringe or is an ancestor of the fringe node. The space complexity is the same as the time complexity.

## Uniform-Cost Search:-

Breadth-first search is optimal when all step costs are equal, because it always expands the shallowest unexpanded node. Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost. Note that if all step costs are equal, this is identical to breadth-first search.

Uniform-cost search does not care about the no. of steps a path has, but only about their total cost. Therefore, it will get stuck in an infinite loop if it ever expands a node that has a zero-cost action leading back to the same state. It is complete and optimal if the cost of each step exceeds some positive bound $\epsilon$.

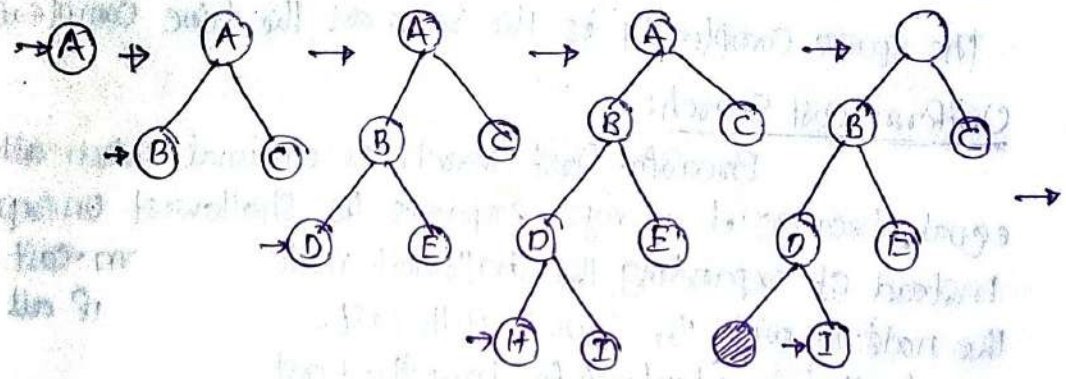
## Time and Space Complexity :-

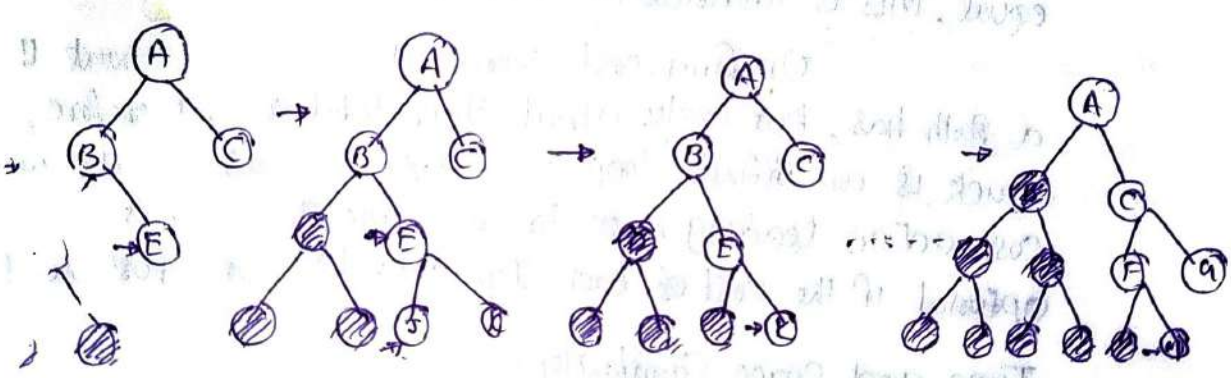
Uniform-cost search is guided by path costs rather than depths, so its complexity cannot easily be characterized in terms of b and d. Let $C^*$ be the cost of the optimal solution and assume that every action costs at least $\epsilon$. Then the algorithm's worst-case time and space complexity is $O(b^{\lceil C^*/\epsilon \rceil})$, which can be much greater than $b^d$. When all steps costs are equal. $b^{\lceil C^*/\epsilon \rceil}$ is just $b^d$.

## Depth-First Search:-

Depth-first search always expands the deepest node in the current fringe of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors.

This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue, also known as a stack. DFS has very modest memory requirements. It needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored. The progress of search is shown below.




Algorithm: Depth-First Search

1. If the initial state is goal state, quit and return success
2. Otherwise, do the following until success or failure is signaled
   a) Generate a successor, E, of the initial state. If there are no more successors, signal failure
   b) Call Depth-first search with E as initial state
   c) If success is returned, signal success. Otherwise continue in this loop.

## Local beam Search:

... track of K states rather than just



### Complexity:

For a state space with branching factor 'b' and maximum depth 'm', depth-first search requires storage of only 'bm+1' nodes. Time complexity is $O(b^m)$ and space complexity is $O(bm)$

A variant of depth-first search called backtracking search uses still less memory. In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next. In this way, only $O(m)$ memory is needed rather than $O(bm)$.

### Advantages:

1. Depth-first search requires less memory since only the nodes on the current path are stored.
2. By chance, DFS may find a solution without examining much of the search space at all.

### Disadvantages:-

1. DFS will get trapped exploring a blind alley.
2. DFS may find a long path to a solution in one part of the tree, when a shorter path exists in some other, unexplored part of the tree.

### Depth-Limited Search:-

Depth-limited search imposes a fixed depth limit on a depth-first search. The depth limit solves the infinite-path problem.

Pseudo code for recursive depth-limited search:-

```
function DEPTH-LIMITED-SEARCH (Problem, limit) returns a solution, or failure/cutoff
    return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[Problem]), Problem, limit)
function RECURSIVE-DLS(node, Problem, limit) returns a solution, or failure/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[Problem](State[node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, Problem) do
        result ← RECURSIVE-DLS(Successor, Problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cut-off-occurred? then return cutoff else return failure
```

Notice that depth-limited search can terminate with two kinds of failure: the standard failure value indicates no action; the cutoff value indicates no solution within the depth limit.

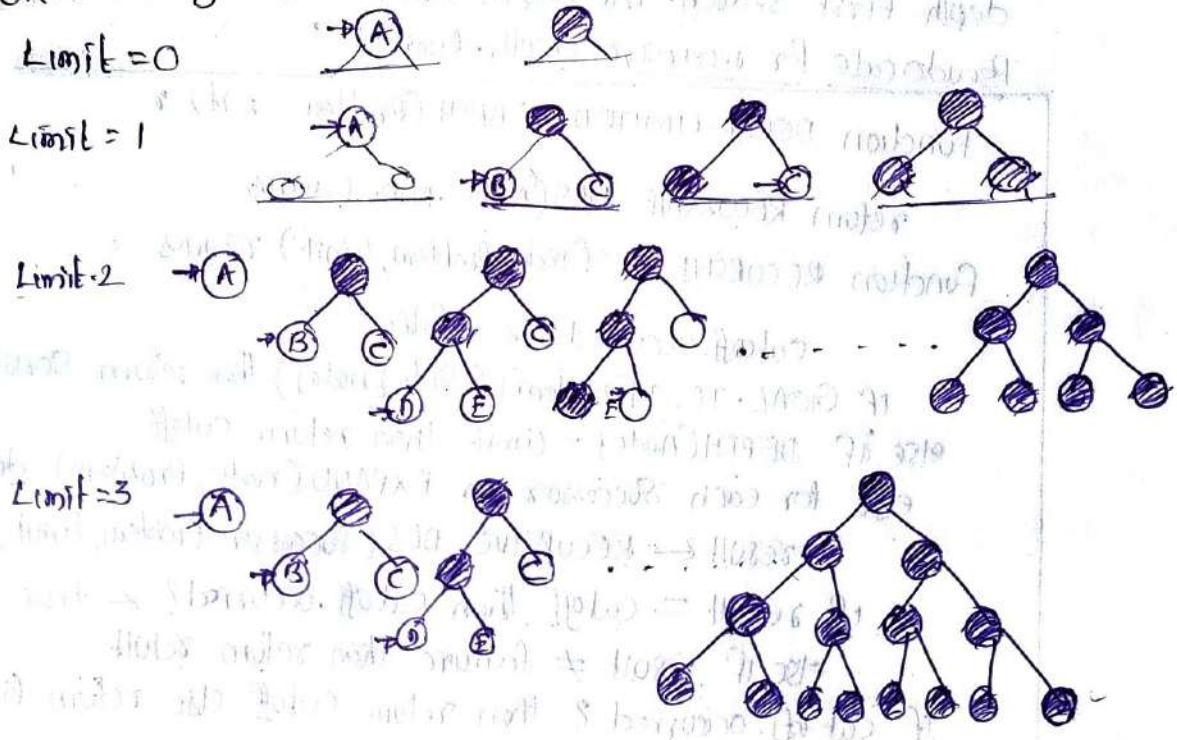
- Time Complexity is $O(b^l)$
- Space complexity is $O(bl)$

## Iterative deepening depth-first Search:-

It's a general strategy, often used in combination with depth-first search, that finds the best depth limit. It does this by gradually increasing the limit, first 0, then 1, then 2 and so on, until a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node. The algorithm is shown below

```
function ITERATIVE-DEEPENING-SEARCH(Problem) returns a solution
                                                or failure
    inputs: Problem, a Problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(Problem, depth)
        if result ≠ cutoff then return result
```

Iterative deepening combines the benefits of depth first and breadth-first search. The figure below shows four iterations of IDS on a binary tree.



## Local beam Search

... keeps track of K states rather than just

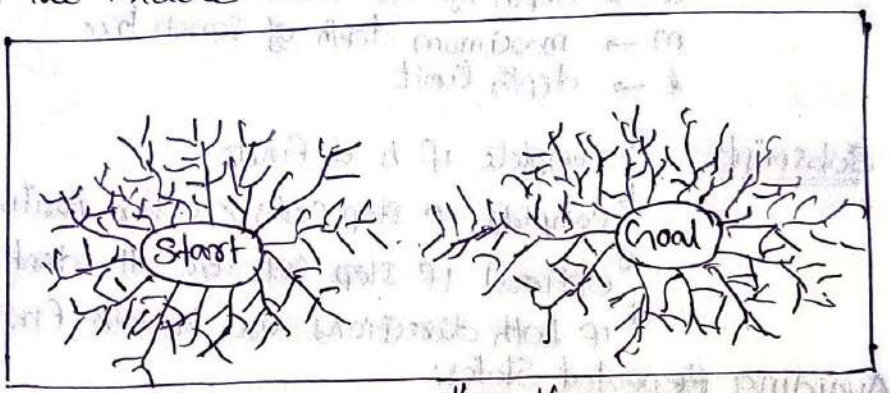


Iterative deepening search may seem wasteful, because states are generated multiple times. The total In an iterative deepening Search, the nodes on the bottom level are generated once, those on the next to bottom level are generated twice, and so on, up to the children of the root, which are generated $d$ times. So the total no. of nodes generated is

$$N(IDS) = (d)b + (d-1)b^2 + \dots + (1)b^d$$ which gives time complexity of $O(b^d)$. Space Complexity is $O(bd)$.

## Bidirectional Search :-

The idea behind bidirectional Search is to run two simultaneous searches - one forward from the initial state and the other backward from the goal, stopping when the two searches meet in the middle.




The motivation is that $b^{d/2} + b^{d/2}$ is much less than $b^d$, or in the figure, the area of the two small circles is less than the area of one big circle centered on the start and reaching of the goal.

Bidirectional Search is implemented by having one or both of the searches check each node before it is expanded to see if it is in the fringe of the other search tree; if so, a solution has been found. Bidirectional search can enormously reduce time complexity, but it is not always applicable and may require too much space.

The time complexity of bidirectional search is $O(b^{d/2})$. At least one of the search trees must be kept in memory so that the membership check can be done, hence the space complexity is also $O(b^{d/2})$.

The algorithm is complete and optimal, if both searches are breadth first, other combinations may sacrifice completeness, optimality or both

## Comparing Uninformed Search Strategies:

The figure below compares search strategies in terms of the four evaluation criteria set.

| Criterion | Breadth-First | Uniform Cost | Depth-First | Depth-Limited | Iterative Deepening | Bidirectional (if applies) |
|---|---|---|---|---|---|---|
| Complete? | yes[a] | yes[a,b] | NO | NO | yes[a] | yes[a,d] |
| Time | $O(b^{d+1})$ | $O(b^{\lceil C^*/\epsilon \rceil})$ | $O(b^m)$ | $O(b^\ell)$ | $O(b^d)$ | $O(b^{d/2})$ |
| Space | $O(b^{d+1})$ | $O(b^{\lceil C^*/\epsilon \rceil})$ | $O(bm)$ | $O(b\ell)$ | $O(bd)$ | $O(b^{d/2})$ |
| Optimal? | yes[c] | yes | NO | NO | yes[c] | yes[c,d] |

$\therefore$ $b \rightarrow$ branching factor
$d \rightarrow$ depth of the shallowest solution
$m \rightarrow$ maximum depth of search tree
$\ell \rightarrow$ depth limit

Subscripts: [a] complete if $b$ is finite
[b] complete if step costs $\geq \epsilon$ for positive $\epsilon$
[c] optimal if step costs are all identical
[d] if both directions use breadth-first search

## Avoiding Repeated States:

Up to this point, we have all but ignored one of the most important complications to the search process: the possibility of wasting time by expanding states that have already been encountered and expanded before. We can modify the general-search algorithm to include a data structure called the closed list, which stores every expanded node. If the current node matches a node on the closed list, it is discarded instead of being expanded.

```
function GRAPH-SEARCH (Problem, fringe) returns a solution or failure
        closed ← an empty set
    fringe ← INSERT (MAKE-NODE(INITIAL-STATE[Problem], fringe)
loop do
    if EMPTY? (fringe) then return failure
    node ← REMOVE-FIRST (fringe)
  if GOAL-TEST [Problem] (state[node]) then return SOLUTION(node)
  if STATE[node] is not in closed then
        add STATE[node] to closed
  fringe ← INSERT-ALL (EXPAND (node, Problem), fringe)
```

## Local beam Search:

... which keeps track of K states rather than just ...



## Searching with Partial Information :-

If the environment is fully observable and deterministic, the agent can calculate exactly which state results from any sequence of actions and always knows which state it is in. What happens when the knowledge of the states or actions is incomplete? We find that different types of incompleteness lead to three distinct problem types.

### 1. Sensorless Problems (Conformant Problems):-

If the agent has no sensors at all, then it could be in one of several possible initial states, and each action might therefore lead to one of several possible successor states. We call each such set of states a belief state.

To solve sensorless problems, we search in the space of belief states rather than physical states. The initial state is a belief state, and each action maps from a belief state to another belief state. An action is applied to a belief state by unioning the results of applying the action to each physical state in the belief state. A path now connects several belief states, and a solution is now a path that leads to a belief state, all of whose members are goal states. In general, if the physical state space has $S$ states, the belief state space has $2^S$ belief states.

### 2. Contingency Problems:-

If the environment is partially observable or if actions are uncertain, then the agent's percepts provide new information after each action. Each possible percept defines a contingency that must be planned for.

The algorithms for contingency problems are more complex than the standard search algorithms.

### 3. Exploration Problems:-

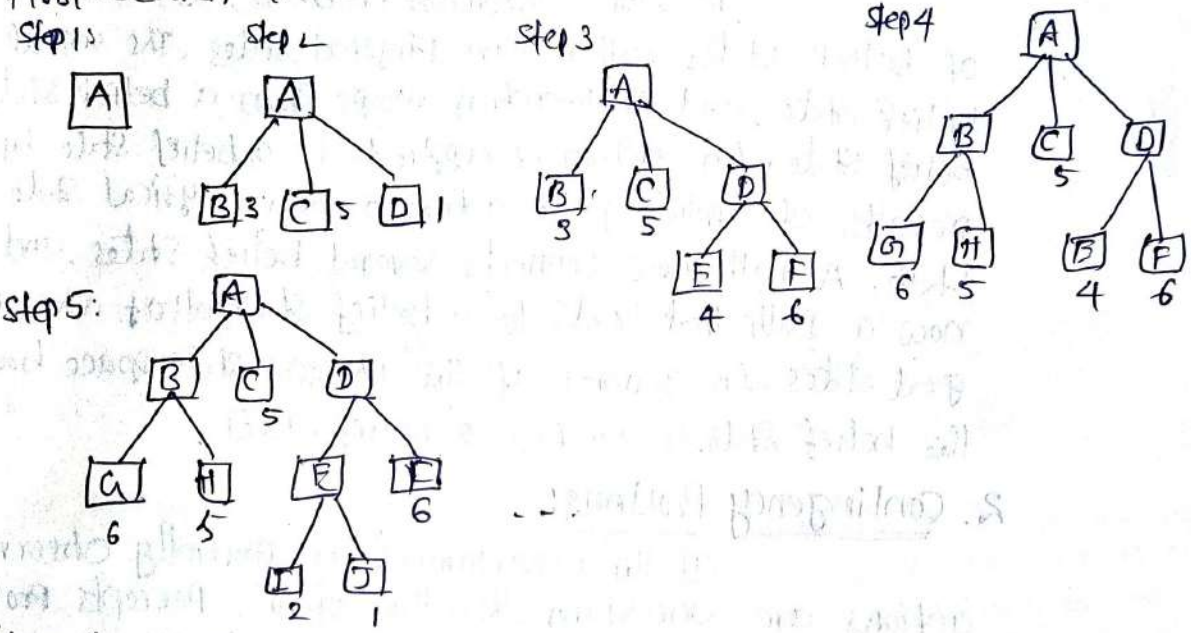
When the states and actions of the environment are unknown, the agent must ~~select~~ act to discover them. Exploration problems can be viewed as an extreme case of contingency problems.

NOTE :- A problem is called adversarial if the uncertainty is caused by the actions of another agent.

# INFORMED (HEURISTIC) SEARCH STRATEGIES:

## Best-First Search:-

Best-first Search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an evaluation function, $f(n)$. Traditionally, the node with the lowest evaluation is selected for expansion, because the evaluation measures distance to the goal. In best-first search, one move is selected, but the others are kept around so that they can be revisited later if the selected path becomes less promising. This is done by applying an appropriate heuristic function of each of them, $h(n)$. The figure below shows the best-first-search procedure.




**Algorithm:-** Best-First Search

1. Start with OPEN containing just the initial state
2. Until a goal is found or there are no nodes left on OPEN do
   a) Pick the best node on OPEN
   b) Generate its successors
   c) For each successor do:
      i> If it has not been generated before, evaluate it, add it to OPEN, and record its Parent.
      ii> If it has been generated before, change the Parent if this new Path is better than the Previous One. In that case update the cost of getting to this node and to any successors that this node may already have.

OPEN — nodes that have been generated but not yet examined
CLOSED — nodes that have already been examined.

Local beam Search

... of k states rather than just

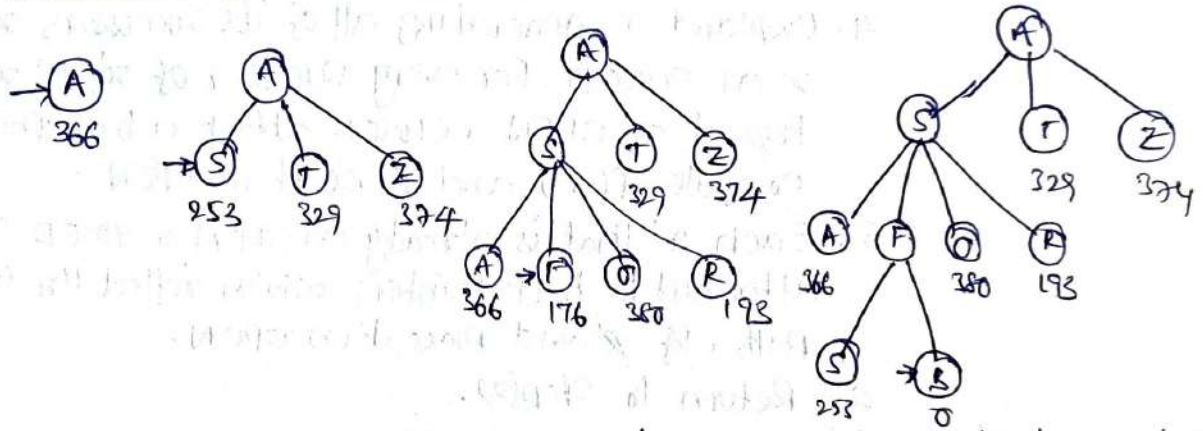
## Greedy best-first Search :-

Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using heuristic function

$f(n) = h(n)$.

Greedy best-first search resembles depth-first search in the way it prefers to follow a single path all the way to the goal, but will backup when it hints a dead end. It is not optimal, but is often efficient.

### Stages in a greedy best-first search :-




∴ The worst-case time complexity and space complexity is $O(b^m)$, where $m$ is the maximum depth of the search tree.

## A* Search :-

The most widely-known form of best-first search is called. It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal.

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

Since $g(n)$ gives the path cost from the start node to node $n$, and $h(n)$ is the estimated cost of the cheapest path from $n$ to the goal, we have $f(n)$ = estimated cost of the cheapest solution through $n$. Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of $g(n) + h(n)$. So A* search is both complete and optimal.

The optimality of A* is straightforward to analyse if it is used with TREE-SEARCH. In this case, A* is optimal, if $h(n)$ is an admissible heuristic ie provided that $h(n)$ never overestimates the cost

to reach the goal. Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is. Since $g(n)$ is the exact cost to reach $n$, we have as immediate consequence that $f(n)$ never overestimates the true cost of a solution through $n$.

## A* Search Algorithm:-

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

## Observations:-

1. A* using TREE-SEARCH is optimal, if $h(n)$ is admissible. If $h(n)$ does not overestimate the cost of completing the solution path, then we know that $f(n) = g(n) + h(n) \leq C^*$, $C^*$ is optimal solution.

2. If we use the GRAPH-SEARCH algorithm, then this condition breaks down. If we impose an extra requirement on $h(n)$, consistency this condition holds. A heuristic $h(n)$ is consistent if, for every node $n$ and every successor $n'$ of $n$ generated by any action $a$, the estimated cost of reaching the goal from $n$ is no longer than the step cost of getting to $n'$ plus the estimated cost of reaching the goal from $n'$. $h(n) \leq c(n, a, n') + h(n')$.

So, A* using GRAPH-SEARCH is optimal. If $h(n)$ is consistent.

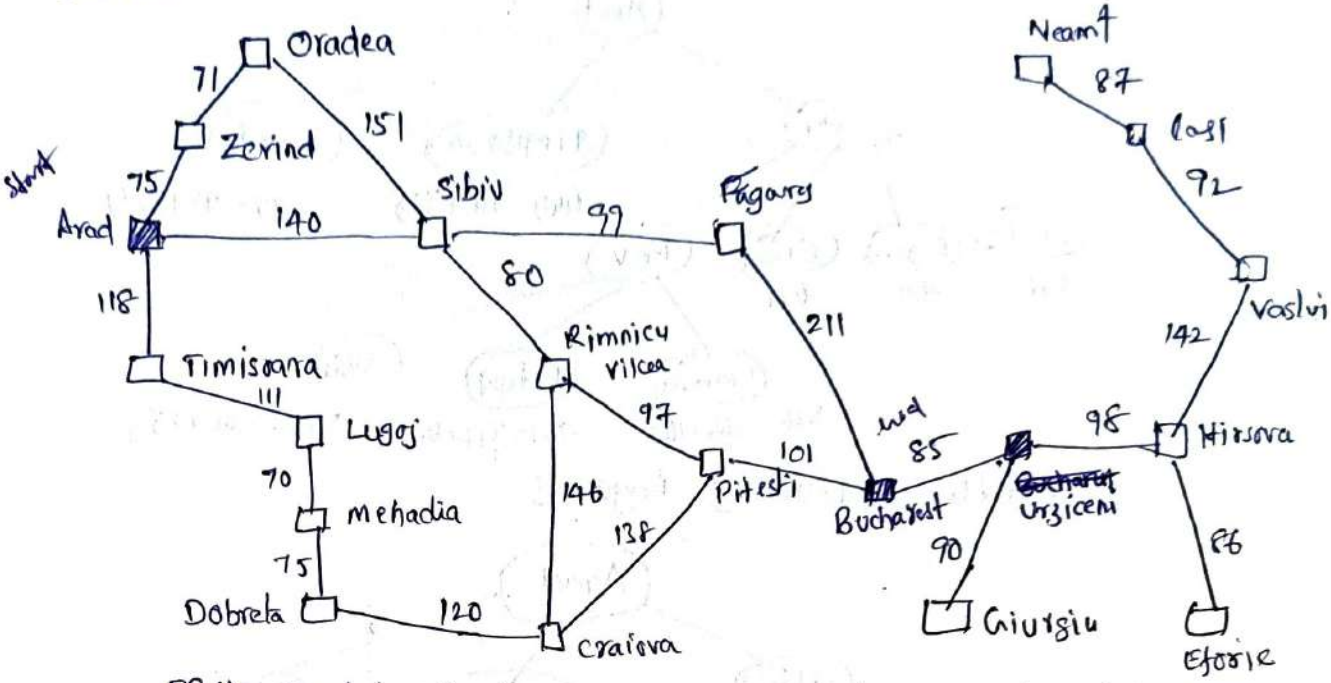
3. If $C^*$ is the cost of the optimal solution path, then we can say the following:

- A* expands all nodes with $f(n) < C^*$
- A* might then expand some of the nodes right on the "goal contour" (where $f(n) = C^*$) before selecting a goal node.

Drawback:- The space of complexity of A* is still prohibitive.

## A* Search:

**example:-** The values of 'g' are computed from the step costs in figure below.



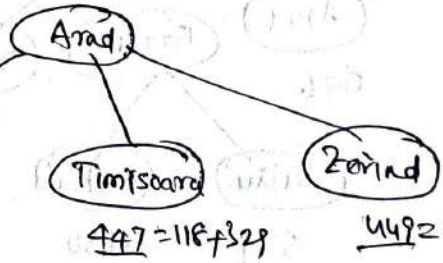

If the goal is Bucharest, we will need to know the straight-line distances to Bucharest, which are shown below.

| | | |
|---|---|---|
| Arad – 366 | Giurgiu – 77 | Oradea – 380 |
| Bucharest – 0 | Hirsova – 151 | Pitesti – 100 |
| Craiova – 160 | Iasi – 226 | Rimnicu Vilcea – 193 |
| Dobreta – 242 | Lugoj – 244 | Sibiu – 253 |
| Eforie – 161 | Mehadia – 241 | Timisoara – 329 |
| Fagaras – 176 | Neamt – 234 | Urziceni – 80 |
| | | Vaslui – 199 |
| | | Zerind – 374 |

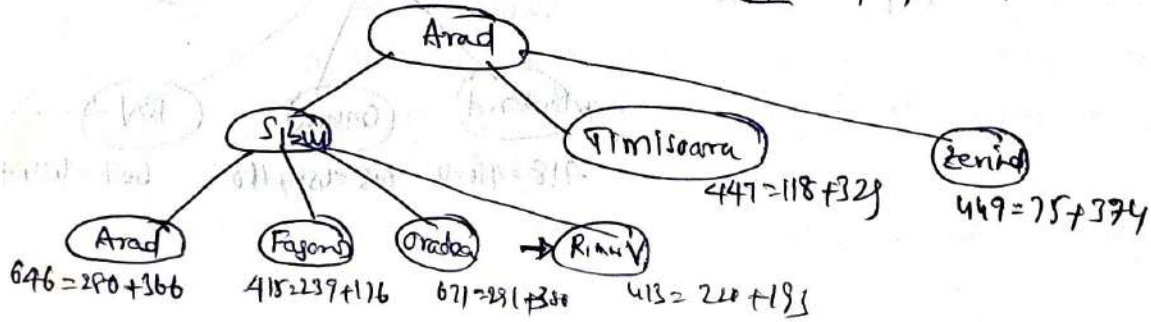
fig: Values of $h_{SLD}$ – straight-line distances to Bucharest

**Stages in A* Search for Bucharest:**

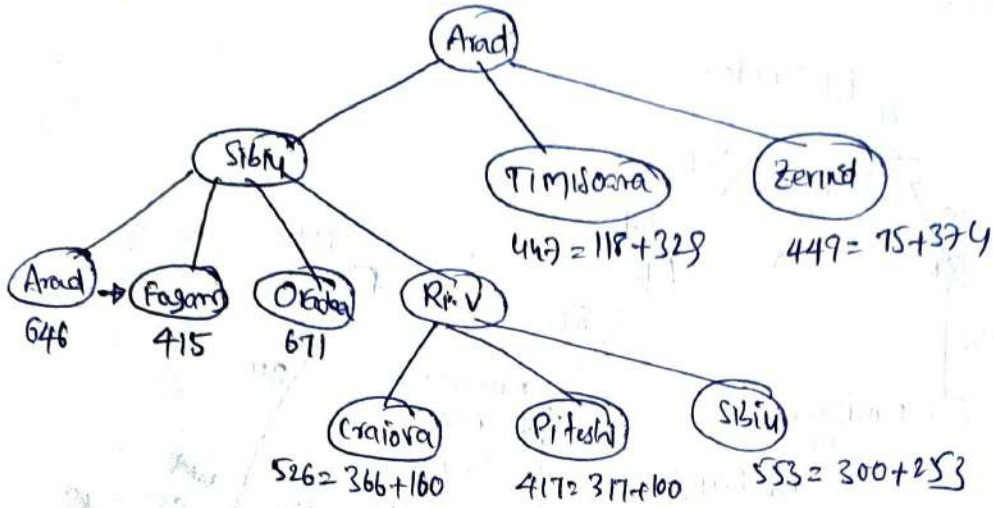
- Initial State → Arad
  366 = 0 + 366
- After expanding Arad



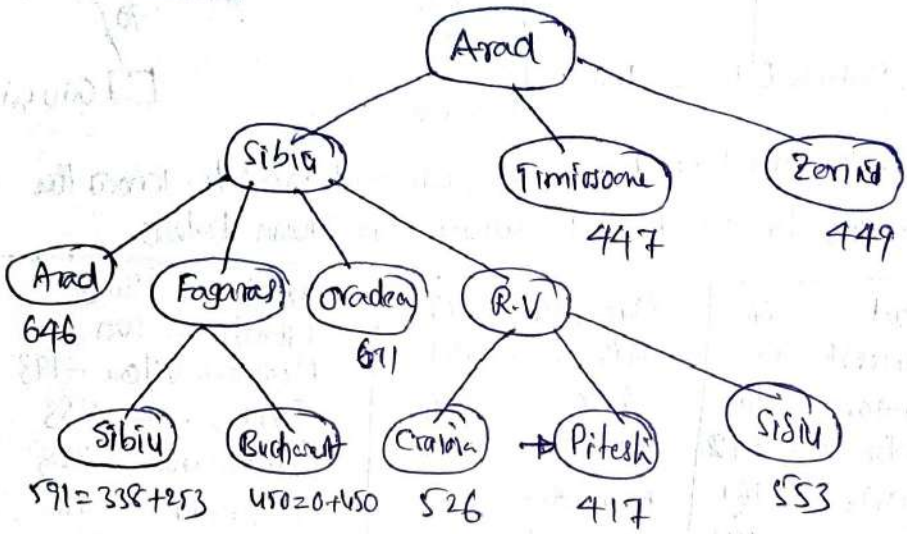

- After expanding Sibiu



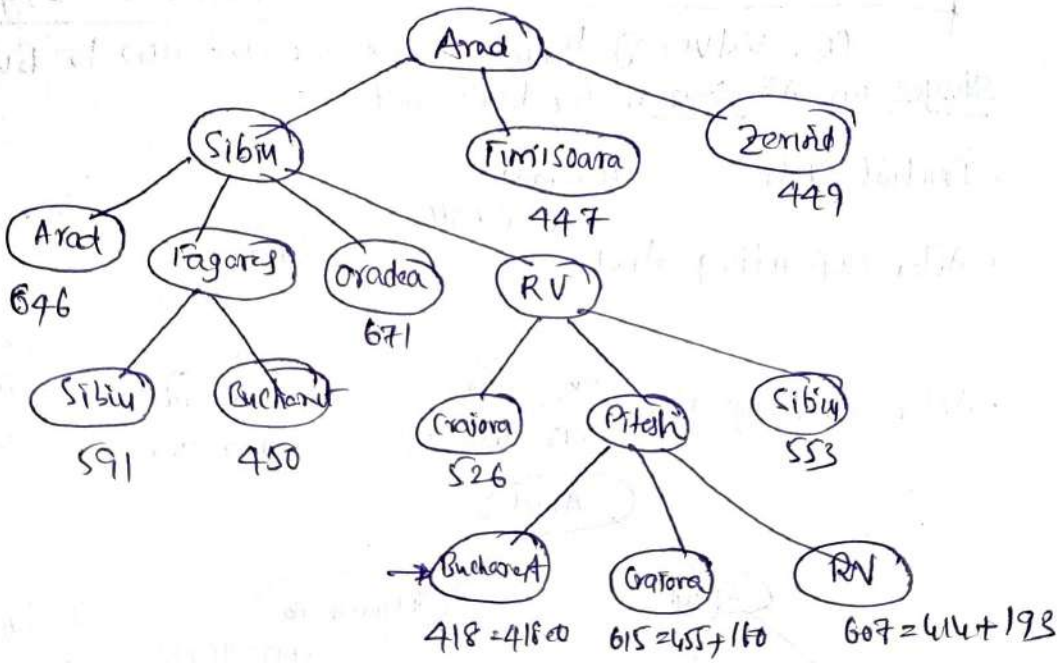
- After expanding Rimnicu Vilcea




- After expanding Fagaras




- After expanding Pitesti



## Memory-bounded Heuristic Search:-

Iterative-Deepening A*:- The main difference between IDA* and standard iterative deepening is that the cutoff used is the f-cost (g+h) rather than the depth; at each iteration, the cutoff value is smallest f-cost of any node that exceeded the cutoff on the previous iteration.

### Recursive best-First Search (RBFS):-

RBFS is a simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only one linear space. The Algorithm is shown below.

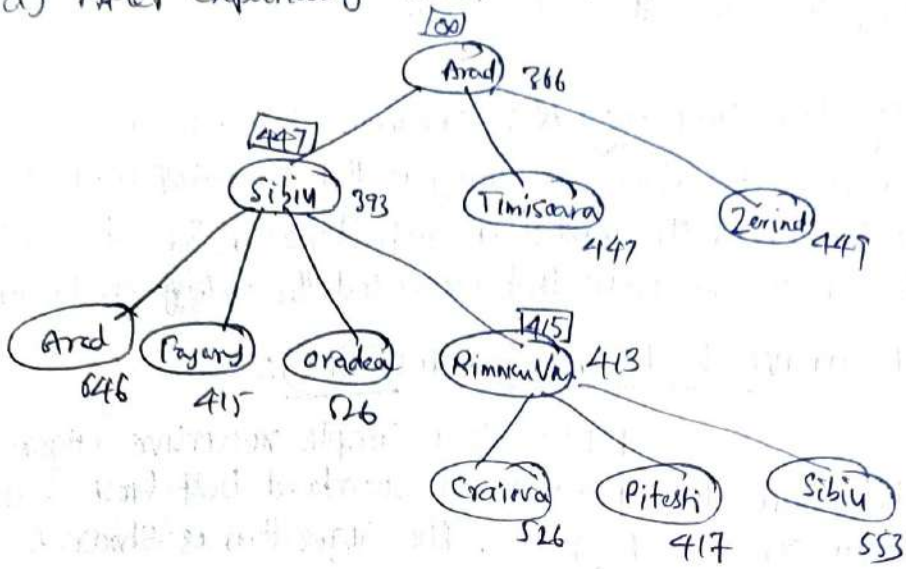
```
function RECURSIVE-BEST-FIRST SEARCH (Problem) returns a
                                    solution, or failure
    RBFS (Problem, MAKE-NODE (INITIAL-STATE [Problem]), ∞)

function RBFS (Problem, node, f-limit) returns a solution or
                    failure and a new. f-cost limit
  if GOAL-TEST [Problem] (state) then return node
    successors ← EXPAND (node, Problem)
  if successors is empty then return failure, ∞
  for each 's' in successors do
      f[s] ← max[g(s) + h(s), f[node]]
  repeat
      best ← the lowest f-value node in successors
  if f[best] > f-limit then return failure, f[best]
  alternative ← the second-lowest f-value among successors
  result, f[best] ← RBFS (Problem, best, min (f-limit, alternative))
  if result ≠ failure then return result.
```

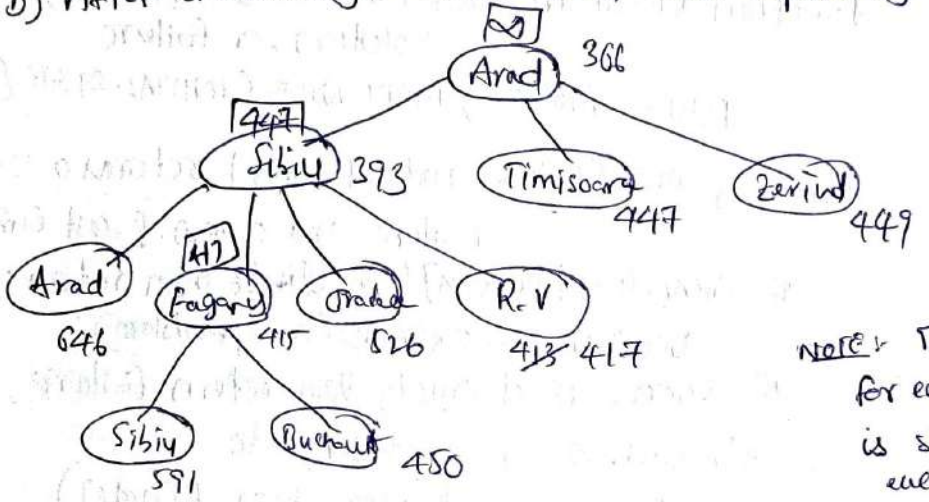
It's structure is similar to that of a recursive depth-first search, but rather than continuing indefinitely down the current path, it keeps track of the f-value of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with the best f-value of its children.

The figure below shows how RBFS reaches to Bucharest

a) After expanding Arad, Sibiu and Rimnicu Vilcea

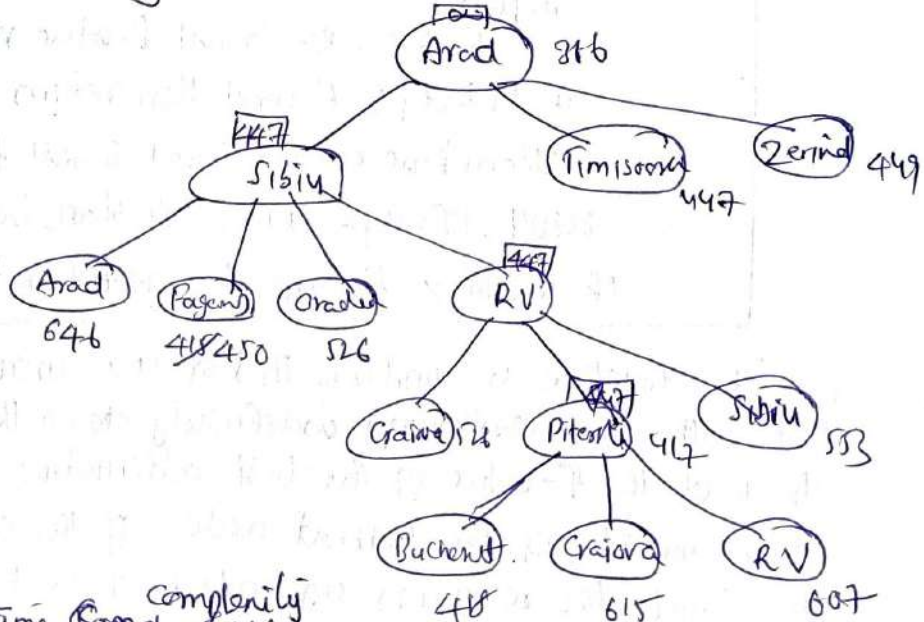



b) After unwinding back to sibiu and expanding Fagaras




NOTE: The f-cost value for each recursive call is shown on top of each current node

c) After switching back to Rimuicu Vilcea and expanding Pitesti




Space and Time ~~Complexity~~ complexity:

Its space complexity is $O(bd)$, but its time complexity is rather difficult to characterize; it depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded.

## Simplified Memory-bounded A* (SMA*):-

SMA* proceeds just like A*, expanding the best leaf until memory is full. At this point, it cannot add a new node to the search tree without dropping an old one. SMA* always drops the worst leaf node - the one with the highest f-value. Like RBFS, SMA* then backs up the value of the forgotten node to its parent. In this way, the ancestor of a forgotten subtree knows the quality of the best path in that subtree.

When all the leaf nodes have the same f-value, then expand the newest best leaf and deleting the oldest worst leaf. SMA* is complete if there is any reachable solution as it is optimal if any optimal solution is reachable.

## Heuristic Functions:-

The performance of heuristic search algorithms depends on the quality of the heuristic function.

One way to characterize the quality of a heuristic is the effective branching factor $b^*$. If the total no. of nodes generated by A* for a particular problem is N, and the solution depth is d, then $b^*$ is the branching factor that a uniform tree of depth d would have to have in order to contain 'N+1' nodes. Thus,

$$N+1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

The effective branching factor can vary across problem instances, but usually it is fairly constant for sufficiently hard problems. A well-designed heuristic would have a value of $b^*$ close to 1.

Good heuristics can sometimes be constructed by relaxing the problem. (A problem with fewer restrictions on the actions is called a relaxed problem). The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem. It can be constructed by precomputing solution costs for subproblems in a pattern database. The idea behind pattern databases is to store these exact solution costs for every possible subproblem instance.

Good heuristics can be constructed by learning from experience with the problem class.

Expected or IMPORTANT QUESTIONS:

1. Define in your own words the following terms: state, state space, search tree, search node, goal, action, successor function and branching factor.
2. Give the initial state, goal test, successor function, and cost function for each of the following. Choose a formulation that is precise enough to be implemented.
   a) A 3-foot-tall monkey is in a room where some bananas are suspended from the 8-foot ceiling. He would like to get the bananas. The room contains two stackable, movable, climbable 3-foot-high crates.
   b) You have three jugs, measuring 12 gallons, 8 gallons and 3 gallons and a water faucet. You can fill the jugs up or empty them out from one to another or onto the ground. You need to measure out exactly one gallon.
3. Formulate the missionaries and cannibals problem precisely, making only those distinctions necessary to ensure a valid solution. Draw a diagram of the complete state space.
   [Hint: 3 missionaries and 3 cannibals are on one side of the river, along with a boat that can hold one or two people. Find a way to get everyone to other side, without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place.]
4. Prove the following statement: "Breadth-first search, depth-first search and uniform cost search are special cases of best-first search"
5. Show that the Tower of Hanoi problem can be classified under the area of AI. Give a state space representation of the problem.
6. Describe with necessary diagrams, a suitable state space representation for 8 puzzle problem and explain how the problem can be solved by state space search.
7. a) Discuss the areas of application of AI
   b) Discuss the tic-tac-toe problem in detail and explain how it can be solved using AI technique.
8. Write A* Algorithm with example and Comment on its Performance.
9. Explain Breadth-First Search and Depth-First Search and develop algorithms for them. List the advantages and disadvantages of both.
10. State and explain the Water jug problem. Give the solution for it.

# HEURISTIC SEARCH TECHNIQUES

Informed Search:

## Local Search Algorithms and optimization Problems

The Search Algorithms that we have seen so far are designed to explore search spaces systematically. This systematically is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path and which have not. When a goal is found, the path to that goal also constitutes a solution to the problem.

Local Search Algorithms operate using a single current state (rather than multiple paths) and generally move only to neighbors of that state. They are not systematic and have two advantages.

1) They use very little memory
2) They can find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.

In addition to finding goals, local search algorithms are useful for solving pure optimization problems. To understand local search, we consider state space landscape. A landscape has both location and elevation. If elevation corresponds to cost, then the aim is to find the lowest valley, a global minimum; if elevation corresponds to an objective function, then the aim is to find the highest peak, a global maximum. A complete local search algorithm always finds a goal if one exists, an optimal algorithm finds a global minimum/maximum always.

### Hill climbing Search:

Hill climbing algorithm is simply a loop that continually moves in the direction of increasing values i.e uphill. It terminates when it reaches a "peak" where no neighbor has a higher value. The algorithm does not maintain a search tree so the current node data structure need only ~~data structure~~ record the state and its objective function value.

```
function HILL-CLIMBING (problem) returns a state that is a local
        maximum
inputs: problem, a problem
local variables: current, a node; neighbor, a node:
  current ← MAKE-NODE (INITIAL-STATE[problem])
  loop do
     neighbor ← a highest-valued successors of current
     IF VALUE[neighbor] ≤ VALUE[current]
          then return STATE[current]
     current ← neighbor
```

Hill climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next. Hill climbing often makes very rapid progress towards a solution, because it is usually quite easy to improve a bad state.

Drawbacks:

1. Local maxima, a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum.
2. Ridges, It results in a sequence of local maxima that is very difficult for greedy algorithms to navigate
3. plateaux, a plateau is an area of the state space landscape where the evaluation function is flat

Types of Hill climbing Search:

1. Stochastic hill climbing, chooses at random from among the uphill moves.
2. First-choice hill climbing, implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state
3. Random-restart hill climbing, adopts the well known adage, "If at first you don't succeed, try, try again". It conducts a series of hill climbing searches from randomly generated initial states, stopping when a goal is found.

Simulated Annealing Search:

Simulated annealing algorithm is quite similar to hill climbing. Instead of picking the best move, however, it picks a random move. If the move improves the situation, it is always accepted. Otherwise, the alg accepts the move with some probability less than 1.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to 'temperature'
  local variables: current, a node; next, a node;
                   T, a temperature controlling the probability of
                   downward steps
  current ← MAKE-NODE(INITIAL-STATE(problem))
  for t ← 1 to ∞ do
      T ← schedule[t]
    If T=0 then return current
      next ← a randomly selected successor of current
    ΔE ← value[next] - value[current]
    If ΔE > 0 then current ← next
    else current ← next only with probability e^(ΔE/T)
```

# HEURISTIC SEARCH TECHNIQUES

## Local beam Search:

The local beam search keeps track of K-states rather than just one. It begins with 'K' randomly generated states. At each step, all the successors of all K states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the K-best successors from the complete list and repeats. In a local beam search, useful information is passed among the K parallel search threads.

Drawback: Local beam search can suffer from a lack of diversity among the K-states - they can quickly become concentrated in a small region of the state space, making the search a little more than an expensive version of hill climbing

Stochastic beam Search: Instead of choosing the best K from the pool of candidate successors, it chooses K successors at random, with the probability of choosing a given successor being an increasing function of its value.

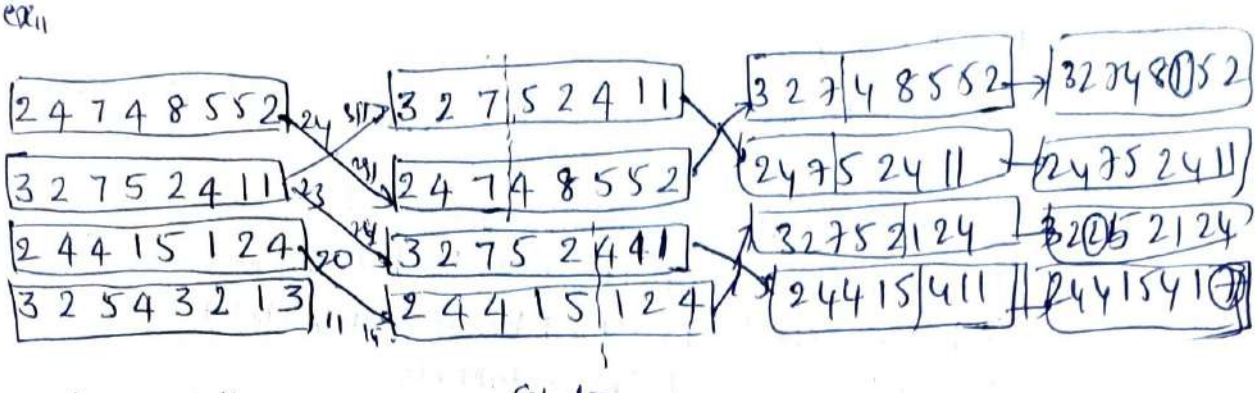
## Genetic Algorithms:

A genetic algorithm is a variant of stochastic beam search in which successor states are generated by combining two parent states, rather than by modifying a single state.

```
function GENETIC-ALGORITHM(Population, FITNESS-FN) returns an individual
    inputs: Population, a set of individuals
            FITNESS-FN, a function that measures the fitness of an individual
    repeat
        new_population ← empty set
        Loop for i from 1 to SIZE(population) do
            x ← RANDOM-SELECTION(Population, FITNESS-FN)
            y ← RANDOM-SELECTION(Population, FITNESS-FN)
            Child ← REPRODUCE(x,y)
            If(Small random Probability) then Child ← MUTATE(child)
            add Child to new-population
        Population ← new-population
    Until some individual is fit enough, or enough time has elapsed
    return the best individual in Population, according to FITNESS-FN

function REPRODUCE(x,y) returns an individual
    inputs: x,y, Parent individuals
        n ← LENGTH(x)
        c ← random number from 1 to n
    return APPEND(SUBSTRING(x,1,c), SUBSTRING(y,c+1,n))
```

ex:-




## Local Search in Continuous Spaces:

Suppose we want to place three new airports anywhere in a country such that the sum of squared distances from each city to its nearest airport is minimized. Then the state space is defined by the coordinates of the airports: $(x_1, y_1)$, $(x_2, y_2)$ and $(x_3, y_3)$. The objective function $f(x_1, y_1, x_2, y_2, x_3, y_3)$ is relatively easy to compute for any particular state once we compute the closest cities.

One way to avoid continuous problems is simply to discretize the neighborhood of each state. for ex, we can move only one airport at a time in either the x or y directions by a fixed amount $\pm\delta$. With 6 variables, this gives 12 successors for each state. We can then apply any of the local search algorithms described previously. One can also apply stochastic Hill climbing and simulated annealing directly, without discretizing the space. There are many methods that attempt to use the gradient of the landscape to find a maximum. The gradient of the objective function is a vector $\nabla f$ that gives magnitude and direction of the steepest slope. For the above problem, we have

$$\nabla f = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right)$$

In some cases, we can find a maximum by solving the equation $\nabla f = 0$. We can compute the gradient locally but not globally. Even so, we can still perform steepest-ascent hill climbing by updating the current state via the formula $x \leftarrow x + \alpha \nabla f(x)$.

the current ← next only when

# HEURISTIC SEARCH TECHNIQUES

Heuristics Search is a very general method applicable to a large class of problems. A Heuristic is a technique that improves the efficiency of a search process - using good heuristics we can hope to get good solutions to hard problems, such as TSP, in less than exponential time.

One example of good general-purpose heuristic that is useful for a variety of combinatorial problems is the nearest neighbour heuristic.

Heuristic function:

It is a function that maps from problem state descriptions to measures of desirability, usually represented as numbers. Well-designed heuristic functions can play an important part in efficiently guiding a search process toward a solution. The purpose of a heuristic function is to guide the search process in the most profitable direction by suggesting which path to follow first when more than one is available.

Figure below shows some simple heuristic functions for a few problems.

chess — the material advantage of our side over the opponent

TSP — the sum of the distances so far

Notice that sometimes a high value of the heuristic function indicates a relatively good position (ex. chess), while at other times a low value indicates an advantageous situation (ex. Traveling Salesman Problem). The program that uses the value of the function can attempt to minimize it or to maximize it as appropriate.

There are several search methods are introduced for solving complex problems. But these methods are also known as 'Weak methods'. They are listed below.

1. Depth-first Search (DFS)
2. Breadth-first Search (BFS)
3. Generate-and-test
4. Hill Climbing
5. Best-First Search
6. Problem Reduction
7. Constraint satisfaction
8. Means-ends Analysis

## 1) Depth-first Search (DFS)

Algorithm:

i) If the initial state is a goal state, quit and return success.

ii) Otherwise, do the following until success or failure is signaled:

(a) Generate a successor, E, of the initial state. If there are no more successors, signal failure.

(b) Call Depth-First Search with E as the initial state

(c) If success is returned, signal success. Otherwise continue in this loop.

Figure below shows a snapshot of a DFS for the water jug problem.

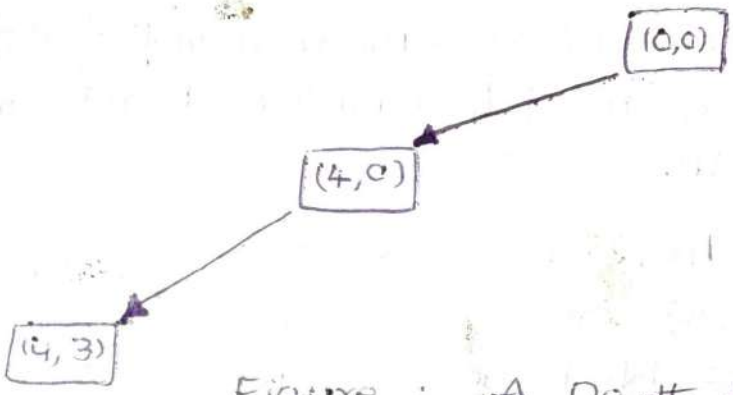



Figure : A Depth-First Search Tree

## Advantages of Depth-First Search:

- Depth-First search requires less memory since only the nodes on the current path are stored. This contrasts with breadth-First search, where all of the tree that has so far been generated must be stored
- By chance, depth-First search may find a solution without examining much of the search space at all. This contrasts with breath-First search in which all parts of the tree must be examined to level n before any nodes on level n+1 can be examined. This is particularly significant if many acceptable solutions exist. Depth-First search can stop when one of them is found

## 2) Breadth-First Search

Algorithm :

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

## Breadth First Search (BFS)

Figure below shows a snapshot of a B.F.S for the water jug problem

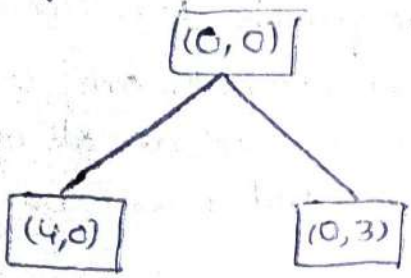



Figure: One level of a Breadth-First-Search Tree

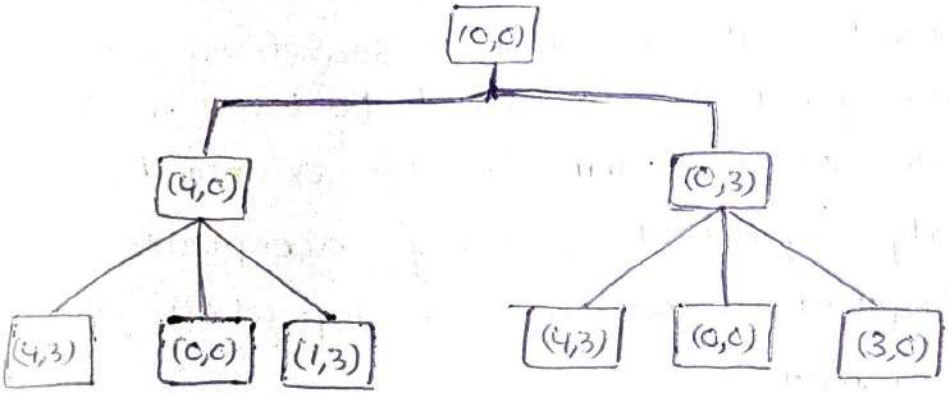



Figure: Two Levels of a Breadth-First Search Tree

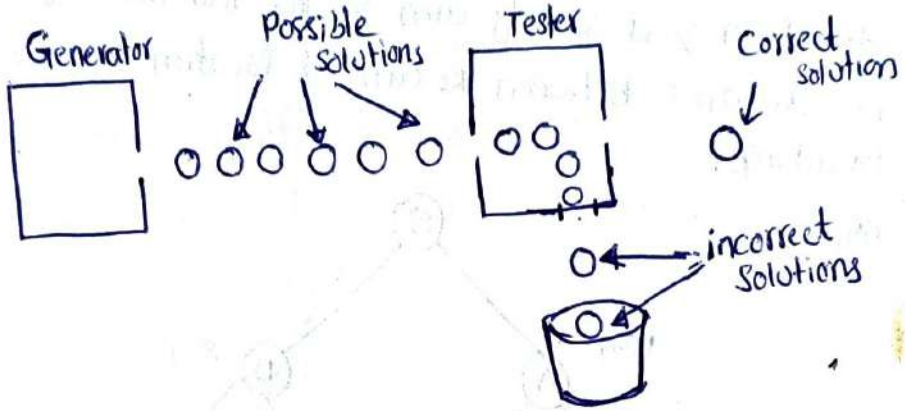
Advantages of Breadth-First Search

- Breadth-First search will not get trapped exploring a blind alley. This contrasts with depth-first searching, which may follow a single, unfruitful path for a very long time, perhaps forever, before the path actually terminates in a state that has no successors. This is a particular problem in d.f.s if there are loops unless special care is expended to test for such a situation..
- If there is a solution, then breadth-first-search is guaranteed to find it. Furthemore, if there are multiple solutions, then a minimal solution will be found. This is guaranteed by the fact that longer paths are never explored - until all shorter ones have already been examined. This contrasts with depth-first-search, which may find a long path to solution in one part of the tree, when a shorter path exists in some other, unexplored part of the tree

## 3) Generate-and-test:

• The generate-and-test strategy is the simplest technique It uses two basic modules. one module, the generator enumerates a possible solutions. The second, the tester, evaluates each proposed solution, either accepting or rejecting that solution.

The generate-and-test algorithm is a DFS procedure since complete solutions must be generated before they can be tested




Algorithm:

~~Algorithm: Means. Ends Analysis~~

1. Generate a Possible solution
2. Test to see if this is actually a solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states:
3. If a solution has been found, quit. Otherwise, return to step1

If the generation of solution is done systematically, then this Problem will find a solution if it exists. If a problem space is very large, it may take very long time to obtain the solutions.

Ex:- Consider the puzzle that consists of four six sided cubes, with each side of each cube painted one of four colors. A solution to the puzzle consists of an arrangement of the cubes in a row such that on all four sides of the row one black face of each color is showing This Problem can be solved by a person in several minutes by systematically trying all possibilities - It can be solved even more quickly using a heuristic generate-and-test procedure. A quick glance at the four blocks reveals that there are more faces than others. Thus when placing a block with several red faces, it would be good idea to use as few of them as possible as outside faces. Using this heuristic, many configurations need never be explored and a solution can be found quickly.

## 4) Hill Climbing

Hill Climbing is a variant of generate-and-test in which feedback from the test procedure is used to help the generator decide which direction to move in the search space. It is often used when a good heuristic function is available for evaluating states but when no other useful knowledge is available. For example, suppose you are in an unfamiliar city without a map and you want to get downtown. you simply aim for the tall buildings. The heuristic function is distance between the current location and the location of the tall buildings.

eg,

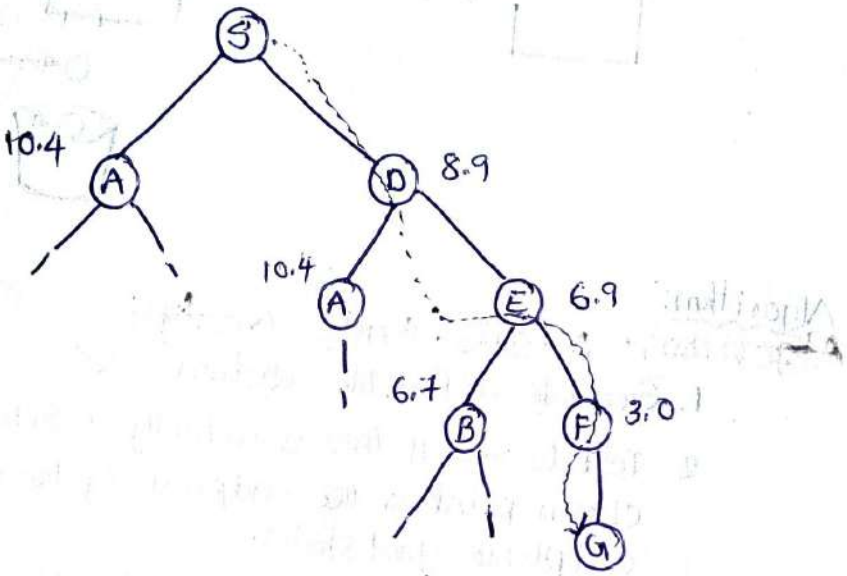



fig: an example of Hill Climbing

There are three types of Hill climbing methods.

1. Simple Hill climbing
2. Steepest-Ascent Hill climbing (gradient Search)
3. Simulated Annealing

### a) Simple Hill Climbing

The key difference between this algorithm given below and generate-and-test is the use of an evaluation function as a way to inject task-specific knowledge into the control process.

To apply simple hill climbing to the colored blocks Problem, we first define a heuristic function that describes how close a particular configuration is to being a solution.

One such function is simple the sum of the no. of different colors on each of the four sides. A solution to the puzzle will have a value of 16. Next we need to define a set of rules that describe ways of transforming one configuration into another. It says simply pick a block and rotate it 90° in any direction.

Now hill climbing ~~beg~~ can begin. We generate a new state by selecting a block and rotating it If the resulting state is better, then we keep it. If not, we return to the previous state and and try a different perturbation. This can be done with the help of an algorithm shown below.

Algorithm: (Simple Hill climbing)

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state
2. Loop until a solution is found or until there are no new operators left to be applied in the current state:
   (a) Select an operator that has not yet been applied to the current state and apply it to produce a new state
   (b) Evaluate the new state.
      (i) If it is a goalstate, then return it and quit
      (ii) If it is not a goal state but it is better than the current state, then make it the current state
      (iii) If it is not better than the current state, then continue in the loop.

b) Steepest Ascent Hill Climbing:

A useful variation on simple hill climbing considers all the moves from the current state and selects the best one as the next state. This method is called Steepest Ascent hill climbing or gradient Search.

To apply steepest-ascent hill climbing to the colored blocks Problem, we must consider all Perturbations of the initial state and choose the best. For this Problem, this is difficult since there are so many possible moves. For a particular problem, the ti required to select a move and the no-of moves required to get to a solution must be considered.

Algorithm: Steepest-Ascent Hill climbing

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state
2. Loop until a solution is found or until a complete iteration produces no change to current state
   (a) Let SUCC be a state such that any possible successor of the current state will be better than SUCC
   (b) For each operator that applies to the current state do:
      i) Apply the operator and generate a new state
      ii) Evaluate the new state. If it is a goal state, then return it and quit. If not, compare it to SUCC. If it is better, then set SUCC to this state. If it is not better, leave SUCC alone
   (c) If the SUCC is better than current state, then set current state to SUCC.

Limitations:-

1) A local maximum: It is a state that is better than all its neighbours but is not better than some other states farther away. At this point, all moves appear to make things worse. They are also called "foothills".
2) A plateau:- It is a flat area of the search space in which a whole set of neighbouring states have the same value. So it is not possible to determine the best direction in which to move
3) A ridge: It is a special kind of local maximum. It is an area of the search space that is higher than surrounding areas.

There are some ways of dealing with these Problems.

## Solutions for Local Maximum:

Backtrack to some earlier node and try going in a different direction. To implement this strategy, maintain a list of paths almost taken and goback to one of them if the path that was taken leads to a dead end.

## Solution for a Plateau:

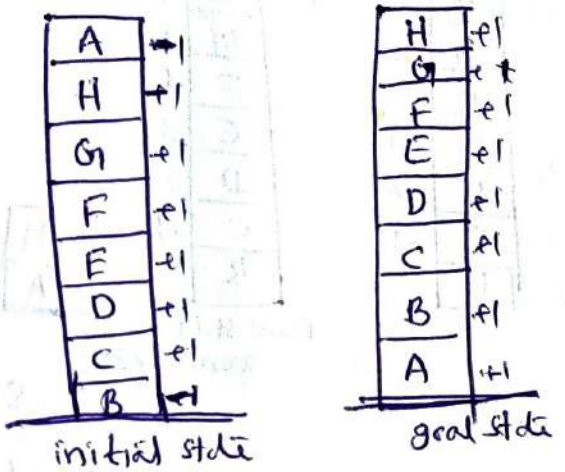
Make a big jump in some direction to try to get to a new section of the search space. If the only rules available describe single small steps, apply them several times in the same direction.

## Solution for a ridge:-

Apply two or more rules before doing the test. This corresponds to moving in several directions atonce.

## Explain Hill Climbing Problem w.r.to Blocks world Problem?

Consider the blocks world Problem as shown below

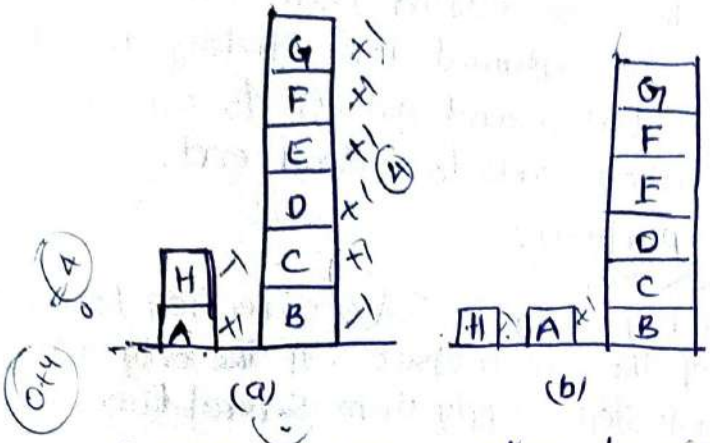



Suppose we use the heuristic function:

Local:- Add one point for every block that is resting on the thing it is supposed to be resting on. Subtract one point for every block that is sitting on the wrong thing.

The goal state has a score of 8. The initial state has a score of 4. (+6-2 = 4)

- Now let us consider two moves. In the first one let us move A to the table and place it on it. In the second one let us move A and H on to the table as shown below.

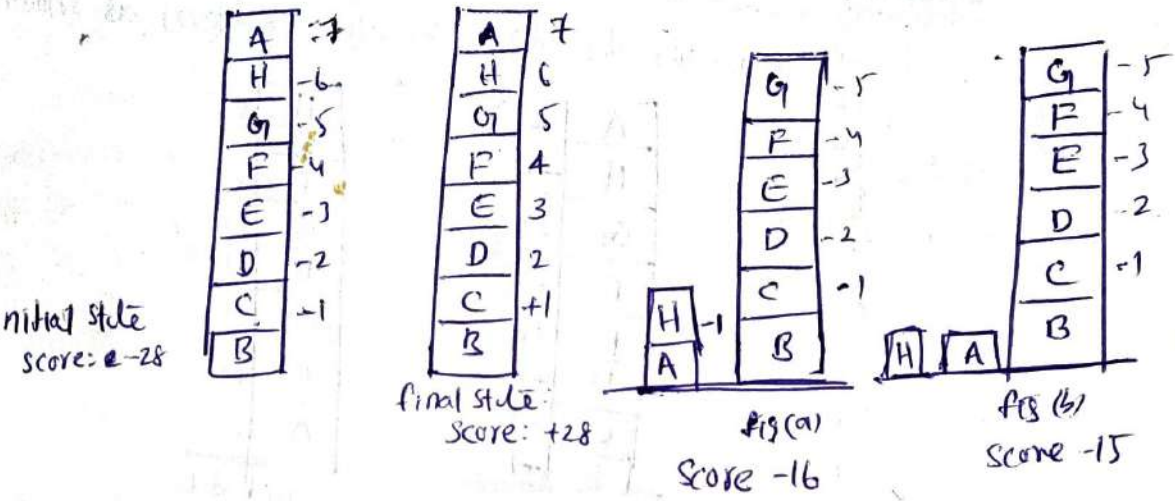



The score for the (a) is 4. and (b) is also 4. The initial state has score 4. This leads to a situation called a plateau. So, It is difficult to select which state could be better as all are leading us to same score.

Suppose we use another heuristic function

Global: For each block that has the correct support structure, add one point for every block in the support structure, subtract one point for every block that has an incorrect structure.

Using this heuristic function, the goal state has a score of 28. The initial state has a score of -28




This time the steepest ascent hill climbing will choose the move (b) which is the correct one.

## c) Simulated Annealing

Simulated Annealing is a variation of Hill climbing. Here we use the term 'objective function' which aims at minimizing value. Simulated Annealing is patterned after the physical process of annealing, in which physical substances such as metals are melted and then gradually cooled until some solid state is reached. The goal of this process is to produce a minimal-energy

the search space that is

final state. But there is some probability that a transition to a higher energy state will occur. This probability is given by the function $p = e^{-\Delta E/kT}$

$\Delta E$ = +ve change in Energy level
$T$ = Temperature
$k$ = Boltzmann's constant

The rate at which the system is cooled is called annealing schedule. The variable 'k' describes the correspondence between the units of T and E. Since in analogous process, the units for both E & T are artificial it makes no sense to include k into T. Selecting values for T that produce desirable behaviour in the Algorithm. We use the revised probability formula $p' = e^{-\Delta E/T}$

Algorithm: (Simulated Annealing)

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Initialize BEST-SO-FAR to the current state
3. Initialize T according to the annealing schedule
4. Loop until a solution is found or until there are no new operators left to be applied in the current state
   (a) Select an operator that has not yet been applied to the current state and apply it to produce a new state
   (b) Evaluate the new state. Compute
   $\Delta E$ = (Value of current) − (Value of new state)
   - If the new state is a goal state, then return it and quit
   - If it is not a goal state but is better than the current state, then make it the current state. Also set BEST-SO-FAR to this new state
   - If it is not better than the current state, then make it the current state with probability $p'$ as defined above. This step is usually implemented by invoking a random number generator to produce

a number in the range [0,1]. If that number is less than p', then the move is accepted otherwise, do nothing.

(c) Revise T as necessary according to the annealing schedule

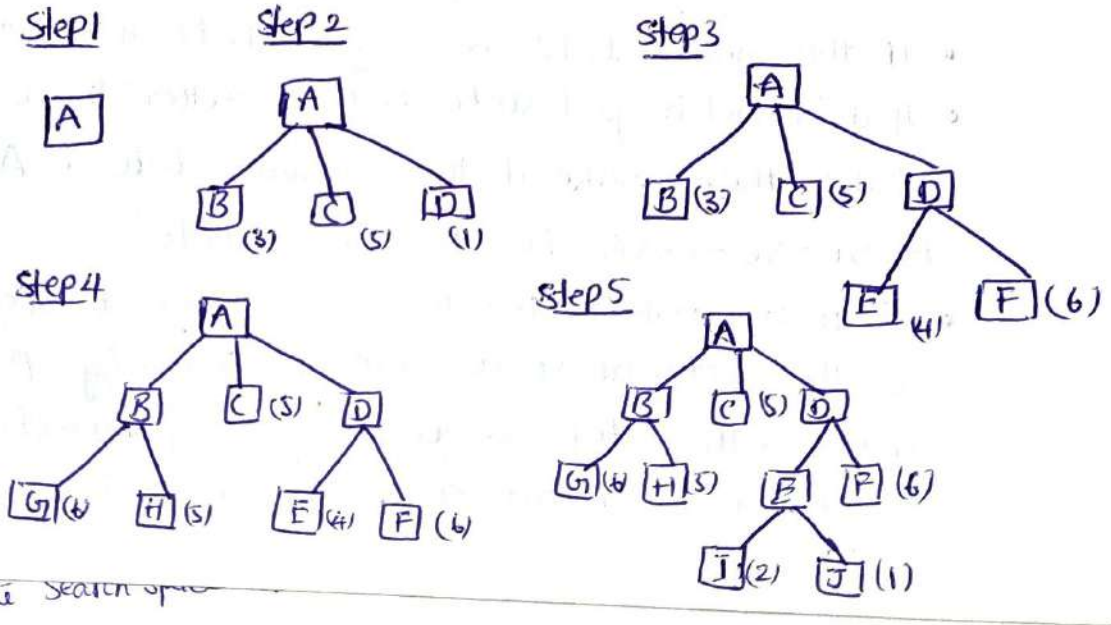
5. Return BEST-SO-FAR as the answer

* There are three Differences between Hill climbing and Annealing schedule Process (Simulated Annealing).

1. The annealing schedule must be maintained.
2. Moves to worse states may be accepted
3. It is good idea to maintain, in addition to the current state, the best state found so far. Then if the final state is worse than that earlier state, the earlier state is still available.

## 5) BEST-FIRST SEARCH.

Best-first Search is a way of combining the advantages of both DFS and BFS. At each step of the best-first search process, we select the most promising of the nodes we have generated so far. This is done by applying an appropriate heuristic functions to each of them. If one of them is a solution, we can quit. If not, all those new nodes are added to the set of nodes generated so far. Again the most promising node is selected and the process continues. This can be shown below.




the search spa

It is better to represent a graph rather than a tree as shown above. To implement such a graph-search procedure we will need to use two lists of nodes

OPEN: Nodes that have been generated and have had the heuristic function applied to them but which have not yet been examined.

CLOSED: Nodes that have been already examined.

Algorithm:

1. Start with OPEN containing just the initial state.
2. Until a goal is found or there are no nodes left on OPEN do
   a) Pick the best node on OPEN
   b) Generate its successors
   c) For each successor do:
      i) If it has not been generated before, evaluate it, add it to OPEN and record its Parent
      ii) If it has been generated before, change the Parent if this new Path is better than Previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

* The A* Algorithm uses the following functions:

- The function 'g' is a measure of the cost of getting from the initial state to the current node.
- The function h' is an estimate of the additional cost of getting from the current node to a goal state.
- The function f' is an estimate of the cost of getting from the initial state to a goal state along the Path that generated the current node.

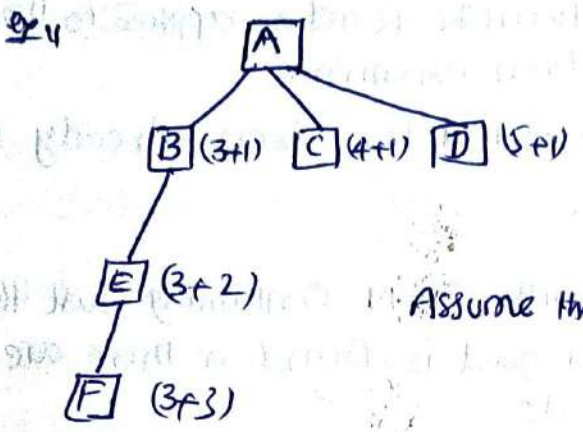
$$f' = g + h'$$

The detailed Algorithm is shown below. (refer next Page)

## Observations - about A* Algorithm

1) The Role of the 'g' function

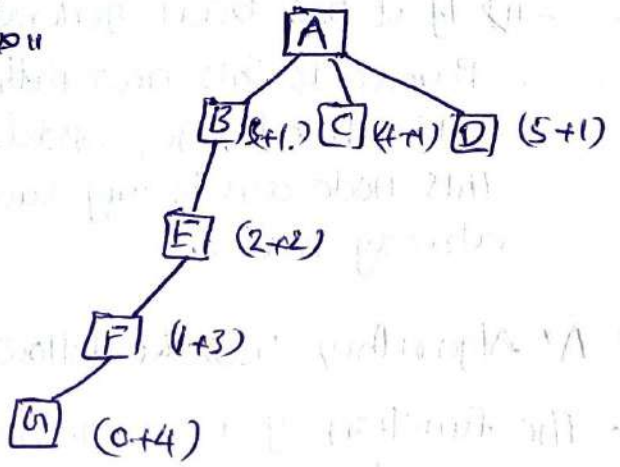
2) h' underestimates h




Assume that the cost of all arcs is 1

In this example, node B has the lowest $f'(4)$, so it is expanded first and so on. $f'(E) = 6$, which is greater than $f'(C)$. So by underestimating $h(B)$ we have wasted some effort.

3) h' overestimates h




In this example, by overestimating $h'(D)$ we make 'D' look so bad that we may find some other, worse solution without ever expanding D.

4) Relationship between trees and graphs

NOTE: A Search Algorithm that is guaranteed to find an optimal path to a goal, is called admissible.

## Algorithm: Problem Reduction

### The A* Algorithm:

The best-first search algorithm is a simplification of an algorithm called A*. This algorithm uses the same f', g and h' functions, as well as the lists OPEN and CLOSED.

Algorithm: A*

1) Start with OPEN containing only the initial node. Set that node's g value to 0, its h' value to whatever it is, and its f' value to h'+0, or h'. Set CLOSED to the empty list.
2) Until a goal node is found, repeat the following procedure: If there are no nodes on OPEN, report failure. Otherwise, Pick the node on OPEN with the lowest f' value. Call it BESTNODE. Remove it from OPEN. Place it on CLOSED. See if BESTNODE is a goal node. If so, exit and report a solution. Otherwise, generate the successors of BESTNODE but do not set BESTNODE to point to them yet. For each such SUCCESSOR, do the following:
   (a) Set SUCCESSOR to point back to BESTNODE. These backwards links will make it possible to recover the path once a solution is found.
   (b) Compute g(SUCCESSOR) = g(BESTNODE) + the cost of getting from BESTNODE to SUCCESSOR.
   (c) See if SUCCESSOR is the same as any node on OPEN. If so, call that node OLD. Since this node already exists in the graph, we can throw SUCCESSOR away and add OLD to the list of BESTNODE's successors. Now we must decide whether OLD's parent link should be reset to point to BESTNODE. It should be if the path we have just found to SUCCESSOR is cheaper than the current best path to OLD. So see whether it is cheaper to get to OLD via its current parent or to SUCCESSOR via BESTNODE by comparing their g values. If OLD is cheaper, then we

to nothing. If SUCCESSOR is cheaper, then reset OLD's parent link to point to BESTNODE, record the new cheaper path in $g(OLD)$, and update $f'(OLD)$

(d) If SUCCESSOR was not on OPEN, see if it is on CLOSED. If so, call the node on CLOSED OLD and add OLD to the list of BESTNODE's successors. Check to see if the new path or the old path is better just as in step 2(c), and set the parent link and $g$ and $f'$ values appropriately. If we have just found a better path to OLD, we must propagate the improvement to OLD's successors. This is a bit tricky. OLD points to its successors. Each successor in turn points to its successors, and so forth, until each branch terminates with a node that either is still on OPEN or has no successors. So to propagate the new cost downward, do a depth-first traversal of the tree starting at OLD, changing each node's $g$ value, terminating each branch when you reach either a node with no successors or a node to which an equivalent or better path has already been found. This condition is easy to check for. Each node's parent link points back to its best known parent. As we propagate down to a node, see if its parent points to the node we are coming from. If so, continue the propagation. If not, then its $g$ value already reflects the better path of which it is part. So the propagation may stop here. But it is possible that with the new value of $g$ being propagated downward, the path we are following may become better than the path through the current parent. So compare the two. If the path through the current parent is still better, stop the propagation. If the path we are propagating through is now better, reset the parent and continue propagation

(e) If SUCCESSOR was not already on either OPEN or CLOSED, then put it on OPEN, and add it to the list of BESTNODE's successors. Compute $f'(SUCCESSOR) = g(SUCCESSOR) + h'(SUCCESSOR$

Algorithm: Problem Reduction

## AGENDAS

An agenda is a list of tasks a system could Perform. Associated with each task there are usually two things

i) A list of reasons, called justifications
ii) A rating representing the overall weight of evidence suggesting that the task would be useful.

Algorithm: Agenda-Driven Search

1. Do until a goal state is reached & the agenda is empty
   a) Choose the most promising task from the agenda
   b) Execute the task by devoting to it the no. of resources determined by its importance. Executing this task will Probably generate additional tasks
      For each of them, do the following:
      i) See if it is already on the agenda. If so, then see if this same reason for doing it is already on its list of justifications. If so, ignore this current evidence. If this justification was not already present, add it to the list. If the task was not on the agenda, insert it.
      ii) Compute the new task's rating, combining the evidence from all its justifications.

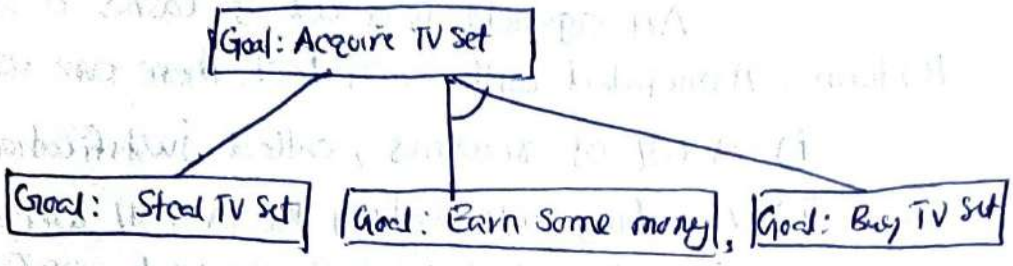
## 6) PROBLEM REDUCTION

AND-OR graphs:-

Using AND-OR graphs many complex problems can be broken down into a series of sub problems. Such that it can be solved by decomposing them into a set of smaller problems ~~All~~ All of which must be solved. These sub problems may be broken down further into sub-sub problems. This can be represented by a directed graph. This decomposition generates arcs that we call AND arcs. In an OR graph, several arcs may emerge from a single node, indicating a variety of ways in which the original

problem might be solved. This structure is called AND-OR graph.

ex:11

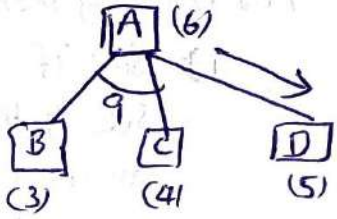



The Problem Reduction Algorithm is shown below. In order to describe this algorithm, we need to exploit a value that we call FUTILITY. If the estimated cost of a solution becomes greater than the value of FUTILITY, then we abandon the search.

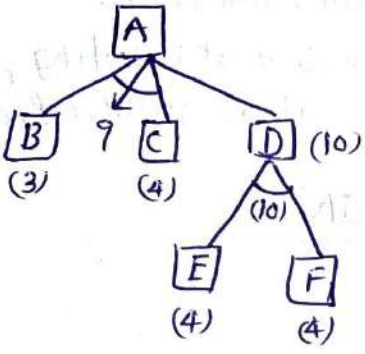
operation:

Before Step 1

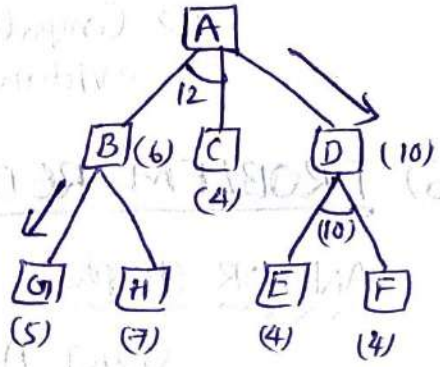

Before Step 2




Before Step 3




Before Step 4




This process continues until either a solution is found or all paths have led to dead ends, indicating that there is no solution.

## Algorithm: Problem Reduction

1. ~~Initia~~ Initialize the graph to the starting node.
2. LOOP until the starting node is Labeled SOLVED or until its cost goes above FUTILITY:

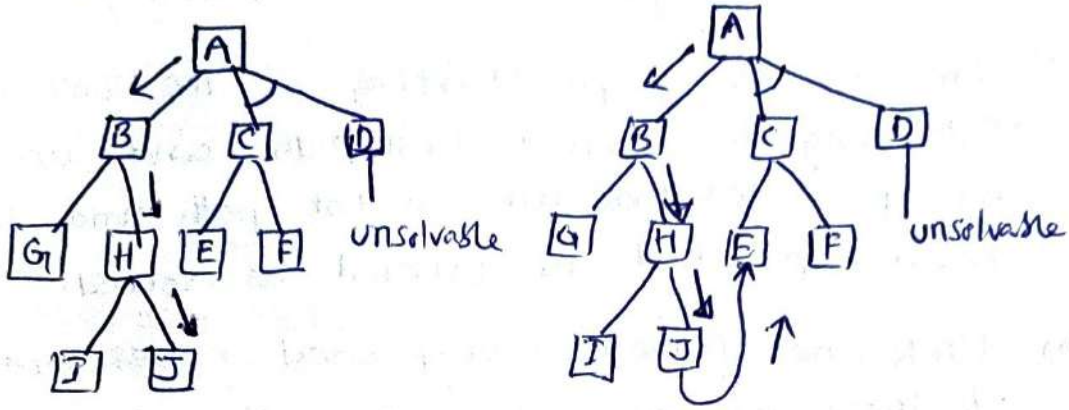
   (a) Traverse the graph, starting at the initial node and following the current best path, and accumulate the set of nodes that are on that path and have not yet been expanded or labeled as solved.

   (b) Pick one of there unexpanded nodes and expand it. If there are no successors, assign FUTILITY as the value of this node. Otherwise, add its successors to the graph and for each of them compute $f'$. If $f'$ of any node is 0, mark that node as SOLVED

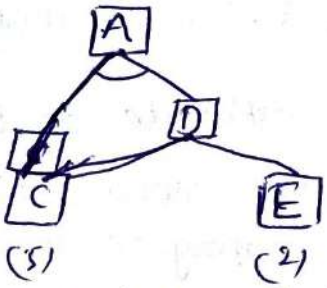
   (c) change the $f'$ estimate of the newly expanded node to reflect the new information provided by its successors. Propagate this change backward through the graph, decide If any node contains a successor arc whose descendants are all solved, label the node itself as SOLVED. At each node that is visited while going up the graph, decide which of its successor arcs is the most promising and mark it as part of the current best path. This may cause the current best path to change. This propa-gation of revised cost estimates back up the tree was not necessary in the best-first search algorithm because only unexpanded nodes were examined. But now expanded nodes must be reexamined so that the best current Path can be selected. Thus it is important that their $f'$ values be the best estimates available.

## Observations:

Sometimes, while finding the solution using AND-OR graphs, longer path may be better. This can be shown with example.




**Limitation:** It fails to take into account any interaction between subgoals.




Assuming that both node C and E ultimately lead to a solution, alg will report a complete solution that includes both of them

## The AO* Algorithm:

This is simplification of Problem Reduction Alg. Rather than the two lists OPEN and CLOSED used in the A* Alg, the AO* algorithm uses a single structure GRAPH, representing the part of the search graph that has been explicitly generated so far. Each node in the graph will also have associated with it an h' value. We will not store g value as we did in the A* algorithm. h' will serve as the estimate of goodness of a node.

Algorithm: AO*

1. Let GRAPH consist only of the node representing the initial state. (Call this node INIT). Compute h'(INIT).
2. Until INIT is labeled SOLVED or until INIT's h' value becomes greather than FUTILITY, repeat the following procedure:
   (a) Trace the labeled arcs from INIT and select for expansion one of the as yet unexpanded nodes that occurs on this path. Call the selected node NODE
   (b) Generate the successors of NODE. If there are none, then assign FUTILITY as the h' value of NODE. This is equivalent to saying that NODE is not solvable. If there are successors, then for each one (called SUCCESSOR) that is not also an ancestor of NODE do the following
      (i) Add SUCCESSOR to GRAPH
      (ii) If SUCCESSOR is a terminal node, label it SOLVED and assign it an h' value of 0.
      (iii) If SUCCESSOR is not a terminal node, computes it's h' value
   (c) Propagate the newly discovered information up the graph by doing the following: Let S be a set of nodes that have been labeled SOLVED or whose h' values have been changed and so need to have values propagated back to their parents. Initialize S to NODE. Until is empty repeat the following procedure
      i) If possible, select from S a node none of whose descendants in GRAPH occurs in S. If there is no such node, select any node from S. Call this node CURRENT, and remove it from S.
      ii) Compute the cost of each of the arcs emerging from CURRENT. The cost of each arc is equal to the sum of the h' value of each of the nodes at the end of the arc plus whatever the cost of the arc itself is

Assign as CURRENT's new h' value the minimum of the costs just computed for the arcs emerging from it.

iii) Mark the best path out of CURRENT by marking the arc that had the minimum cost as computed in the previous step.

iv) Mark CURRENT SOLVED if all of the nodes connected to it through the new labeled arcs have been labeled SOLVED.

v) If CURRENT has been labeled SOLVED or if the cost of CURRENT was just changed, then its new status must be propagated back up the graph. So add all of the ancestors of CURRENT to S.

## 7) CONSTRAINT SATISFACTION

Constraint Satisfaction is a search procedure that operates in a space of constraint sets. The initial state contains the constraints that are originally given in the problem description. A goal state is any state that has been constrained "enough", where "enough" must be defined for each problem.

Constraint Satisfaction is a two step process. First, constraints are discovered and propagated as far as possible throughout the system. Then, if there is still not a solution, search begins. A guess about something is made and added as a new constraint. Propagation can then occur with this new constraint, and so forth.

Constraint propagation terminates for one of two reasons. First, a contradiction may be detected. The and the second is that the propagation has run out of steam and there are no further changes that can be made on the basis of current knowledge. If contradiction is detected, then backtracking can be used to try a different guess and proceed with it.

Cryptarithmetic problems can be solved with help of Constraint Satisfaction only. The constraint satisfaction algorithm is show below.

Algorithm:

1. Propagate available constraints. To do this, first set OPEN to the set of all objects that must have values assigned to them in a complete solution. Then do until an inconsistency is detected or until OPEN is empty
   (a) Select an object OB from OPEN. Strengthen as much as possible the set of constraints that apply to OB.
   (b) If this set is different from the set that was assigned the last time OB was examined or if this is the first time OB has been examined, then add to OPEN all objects that share any constraints with OB.
   (c) Remove OB from OPEN.
2. If the union of the constraints discovered above define a solution, then quit and report the solution
3. If the union of the constraints discovered above define a contradiction, then return failure.
4. If neither of the above occurs, then it is necessary to make a guess at something in order to proceed. To do this loop until a solution is found or all possible solutions have been eliminated:
   (a) Select an object whose value is not yet determined and select a way of strengthening the constraints on that object. (Strengthening)
   (b) Recursively invoke constraint satisfaction with the current set of constraints, augmented by the strengthening constraint just selected.

Eg: 1) Pr Solve the Cryptarithmetic Problem

$$\begin{array}{r} \text{SEND} \\ +\ \text{MORE} \\ \hline \text{MONEY} \end{array}$$

Initial State: No two letters have the same value. The sums of the digits as shown in the Problem

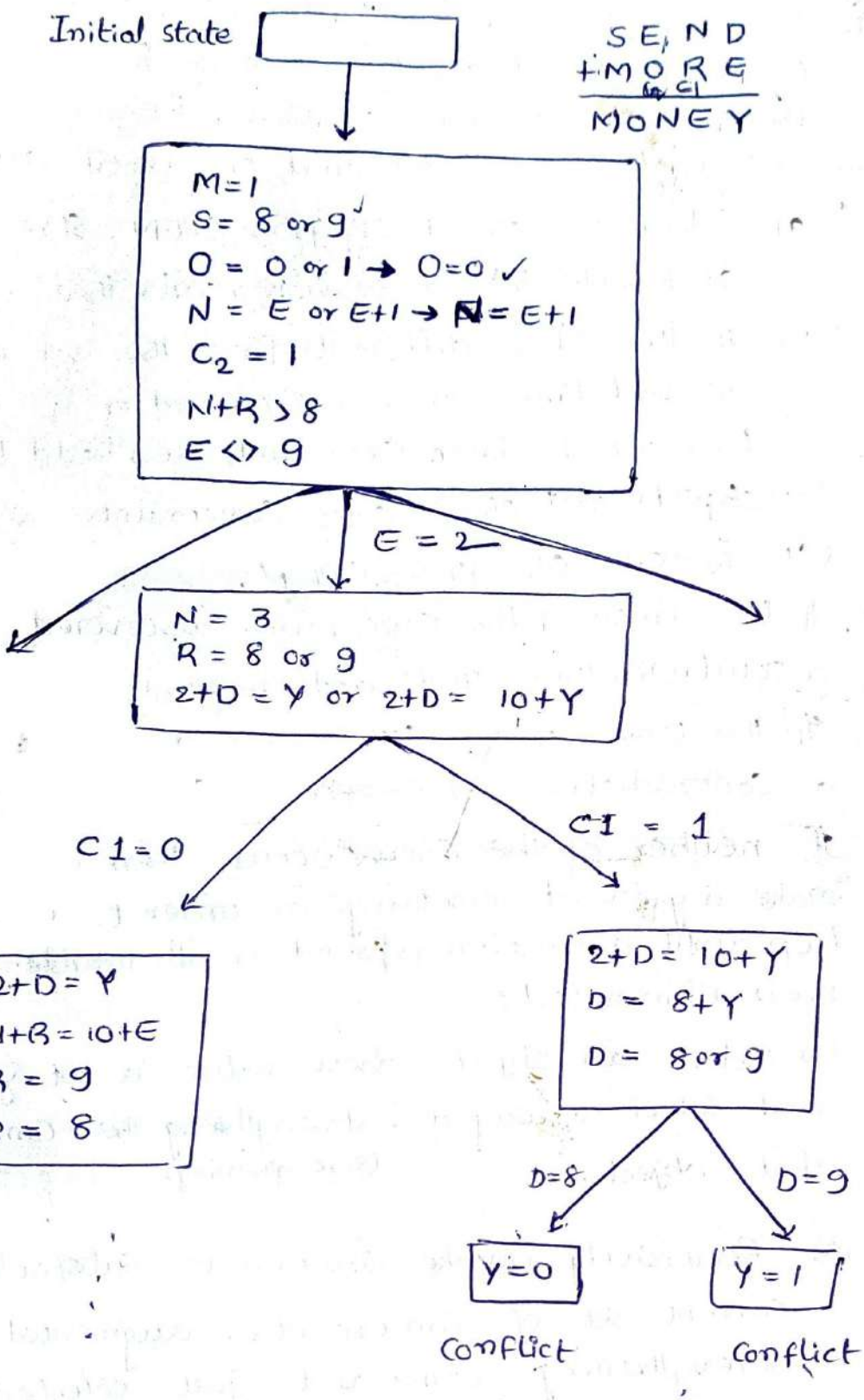



Figure : Solving a Cryptarithmetic Problem

Answer: M=1, O=0, S=9, E=5, N=6, D=7, Y=2, R=8