INTRODUCTION TO ARTIFICIAL INTELLIGENCE

UNIT - I

INTRODUCTION TO AI:
Agents and Environments
Intelligent Agents
Problem-Solving Agents

SEARCHING FOR SOLUTIONS:
Breadth-first search
Depth-first search
Hill-climbing search
Simulated annealing search
Local Search in Continuous Spaces

INTRODUCTION TO ARTIFICIAL INTELLIGENCE

WHAT IS ARTIFICIAL INTELLIGENCE?

It is a branch of Computer Science that pursues creating the computers or machines as intelligent as human beings. It is the science and engineering of making intelligent machines, especially intelligent computer programs.

It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable.

Artificial Intelligence is the study of how to make computers do things, which, at the moment, people do better.

Artificial Intelligence is concerned with the design of intelligence in an artificial device or "man-made" device. The term was coined by John McCarthy in 1956. Intelligence is the ability to acquire, understand and apply the knowledge to achieve goals in the world.

According to the father of Artificial Intelligence, John McCarthy, it is "The science and engineering of making intelligent machines, especially intelligent computer programs".

Artificial Intelligence is a way of making a computer, a computer-controlled robot, or a software think intelligently, in the similar manner the intelligent humans think.

AI is accomplished by studying how human brain thinks and how humans learn, decide, and work while trying to solve a problem, and then using the outcomes of this study as a basis of developing intelligent software and systems.

It has gained prominence recently due, in part, to big data, or the increase in speed, size and variety of data businesses are now collecting. AI can perform tasks such as identifying patterns in the data more efficiently than humans, enabling businesses to gain more insight out of their data.

From a business perspective AI is a set of very powerful tools, and methodologies for using those tools to solve business problems.

From a programming perspective, AI includes the study of symbolic programming, problem solving, and search.

Definitions of AI:

a) "The exciting new effort to make computers think . . . machines with minds, in the full and literal sense" (Haugeland, 1985)
"The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning..."(Bellman, 1978)

b) "The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985)
"The study of the computations that make it possible to perceive, reason, and act" (Winston, 1992)

c) "The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990)
"The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1991)

d) "A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1 990)
"The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)

The definitions on the top, (a) and (b) are concerned with reasoning, whereas those on the bottom, (c) and (d) address behavior. The definitions on the left, (a) and (c) measure success in terms of human performance, and those on the right, (b) and (d) measure the ideal concept of intelligence called rationality.

Goals of Artificial Intelligence

Following are the main goals of Artificial Intelligence:

- ✓ Create an Intelligent Agent
- ✓ Replicate human intelligence
- ✓ Solve Knowledge-intensive tasks
- ✓ An intelligent connection of perception and action
- ✓ Building a machine which can perform tasks that requires human intelligence such as:
  - Proving a theorem
  - Playing chess
  - Plan some surgical operation

- Driving a car in traffic

✓ Creating some system which can exhibit intelligent behavior, learn new things by itself, demonstrate, explain, and can advise to its user.

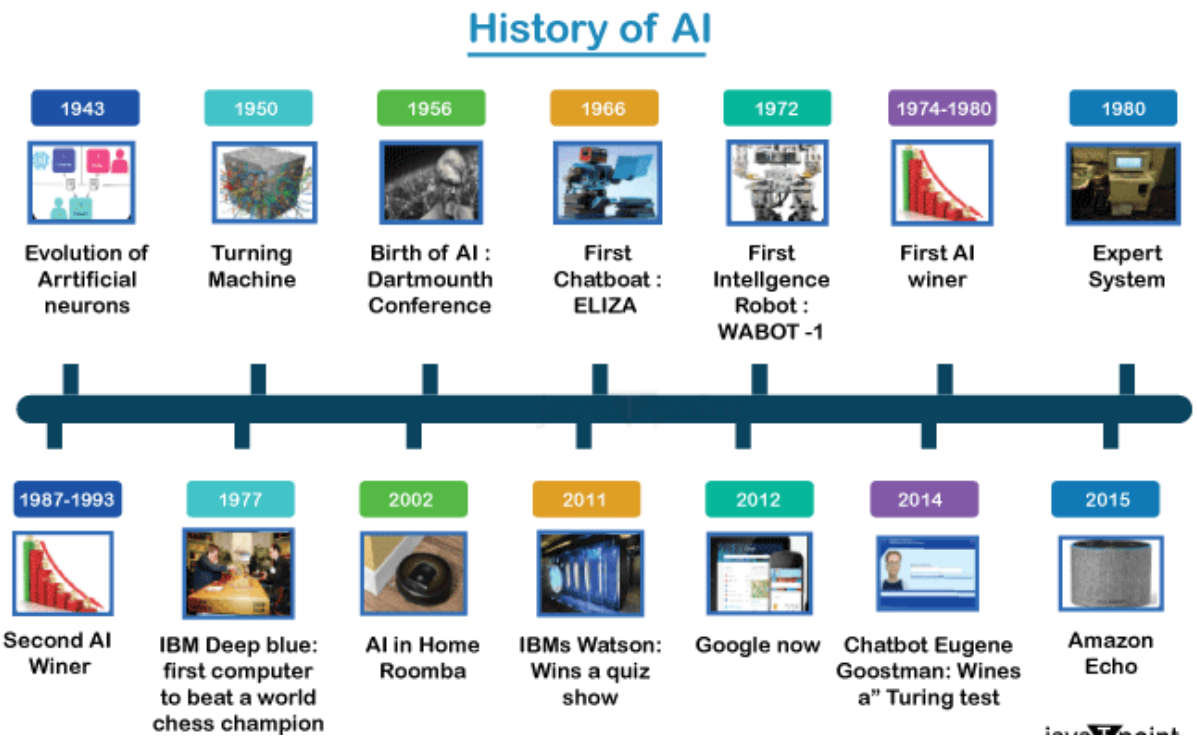
History of AI

Throughout history, people have been intrigued by the idea of making non-living things smart. In ancient times, Greek stories mentioned gods creating clever machines, and in Egypt, engineers made statues move. Thinkers like Aristotle and Ramon Llull laid the groundwork for AI by describing how human thinking works using symbols.

In the late 1800s and early 1900s, modern computing started to take shape. Charles Babbage and Ada Lovelace designed machines that could be programmed in the 1830s. In the 1940s, John Von Neumann came up with the idea of storing computer programs. At the same time, Warren McCulloch and Walter Pitts started building the basics of neural networks.

The 1950s brought us modern computers, letting scientists dig into machine intelligence. Alan Turing's Turing test became a big deal in computer smarts. The term "artificial intelligence" was first used in a 1956 Dartmouth College meeting, where they introduced the first AI program, the Logic Theorist.

The following years had good times and bad times for AI, called "AI Winters." In the 1970s and 1980s, we hit limits with computer power and complexity. But in the late 1990s, things got exciting again. Computers were faster, and there was more data. IBM's Deep Blue beating chess champion Garry Kasparov in 1997 was a big moment.

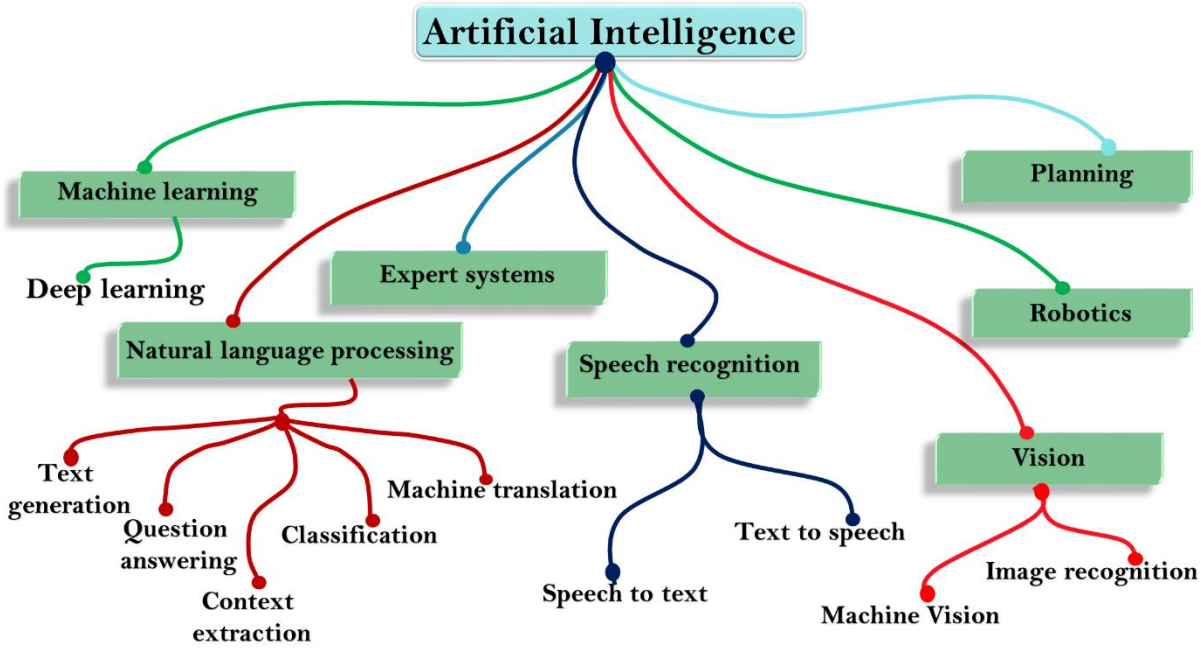


The 2000s started a new era with machine learning, language processing, and computer vision. This led to cool new products and services. The 2010s saw AI take off with things like voice assistants and self-driving cars. Generative AI, which makes creative stuff, also started getting big.
In the 2020s, generative AI like ChatGPT-3 and Google's Bard grabbed everyone's attention. These models can create all sorts of new things when you give them a prompt, like essays or art. But remember, this tech is still new, and there are things to fix, like making sure it doesn't make things up.

Subsets of Artificial Intelligence

Till now, we have learned about what is AI, and now we will learn in this topic about various subsets of AI. Following are the most common subsets of AI:

- Machine Learning
- Deep Learning
- Natural Language processing
- Expert System
- Robotics
- Machine Vision
- Speech Recognition



Applications of AI:

AI algorithms have attracted close attention of researchers and have also been applied successfully to solve problems in engineering. Nevertheless,

for large and complex problems, AI algorithms consume considerable computation time due to stochastic feature of the search approaches
1. Business; financial strategies
2. Engineering: check design, offer suggestions to create new product, expert systems for all engineering problems
3. Manufacturing: assembly, inspection and maintenance
4. Medicine: monitoring, diagnosing
5. Education: in teaching
6. Fraud detection
7. Object identification
8. Information retrieval
9. Space shuttle scheduling

Intelligent Systems:

In order to design intelligent systems, it is important to categorize them into four categories (Luger and Stubberfield 1993), (Russell and Norvig, 2003)
1. Systems that think like humans
2. Systems that think rationally
3. Systems that behave like humans
4. Systems that behave rationally

- **Cognitive Science: Think Human-Like**

a. Requires a model for human cognition. Precise enough models allow simulation by computers.
b. Focus is not just on behavior and I/O, but looks like reasoning process.
c. Goal is not just to produce human-like behavior but to produce a sequence of steps of the reasoning process, similar to the steps followed by a human in solving the same task.

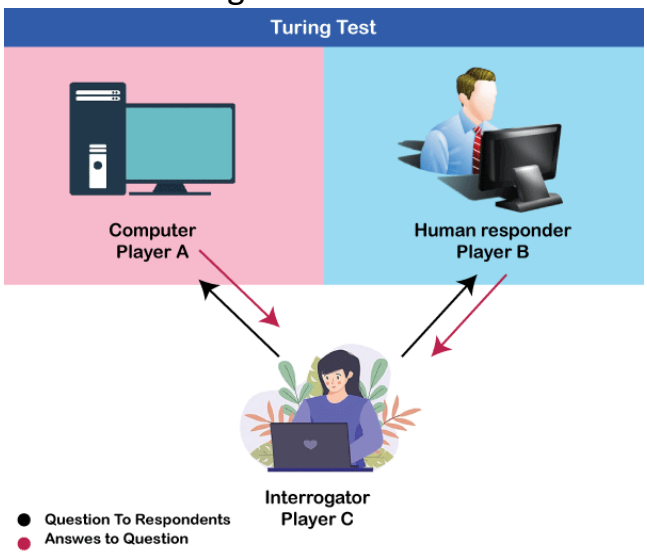
- **Laws of thought: Think Rationally**

a. The study of mental faculties through the use of computational models; that it is, the study of computations that make it possible to perceive reason and act.
Focus is on inference mechanisms that are probably correct and guarantee an optimal solution.
b. Goal is to formalize the reasoning process as a system of logical rules and procedures of inference.
c. Develop systems of representation to allow inferences to be like
—Socrates is a man. All men are mortal. Therefore Socrates is mortal"

- **Turing Test: Act Human-Like**

a. The art of creating machines that perform functions requiring intelligence when performed by people; that it is the study of, how to make computers do things which, at the moment, people do better.
b. Focus is on action, and not intelligent behavior centered around the representation of the world
c. Example: Turing Test
3 rooms contain: a person, a computer and an interrogator.

- The interrogator can communicate with the other 2 by teletype (to avoid the machine imitate the appearance of voice of the person)
- The interrogator tries to determine which the person is and which the machine is.
- The machine tries to fool the interrogator to believe that it is the human, and the person also tries to convince the interrogator that it is the human.
- If the machine succeeds in fooling the interrogator, then conclude that the machine is intelligent.

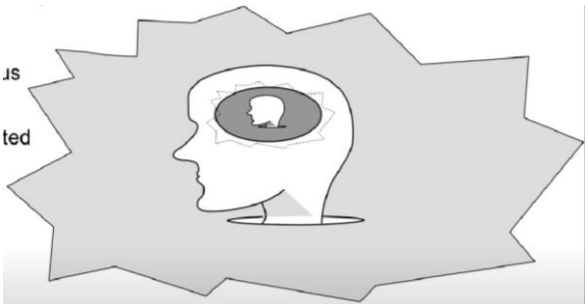


- **Rational agent: Act Rationally**

a. Tries to explain and emulate intelligent behavior in terms of computational process; that it is concerned with the automation of the intelligence.
b. Focus is on systems that act sufficiently if not optimally in all situations.
c. Goal is to develop systems that are rational and sufficient

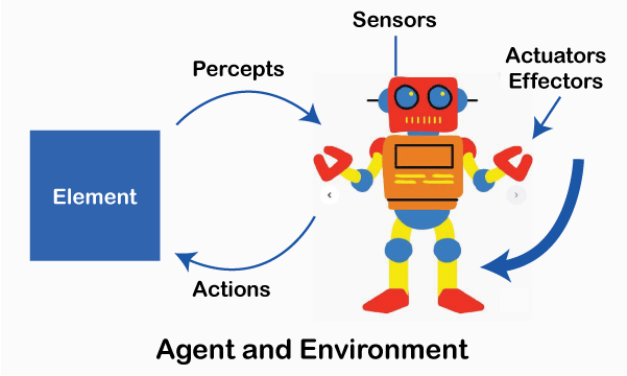
What is an Agent?

An agent can be anything that perceive its environment through sensors and act upon that environment through actuators. An Agent runs in the cycle of perceiving, thinking, and acting. An agent can be:



Human-Agent: A human agent has eyes, ears, and other organs which work for sensors and hand, legs, vocal tract work for actuators.
Robotic Agent: A robotic agent can have cameras, infrared range finder, NLP for sensors and various motors for actuators.
Software Agent: Software agent can have keystrokes, file contents as sensory input and act on those inputs and display output on the screen.
Hence the world around us is full of agents such as thermostat, cellphone, camera, and even we are also agents.

Before moving forward, we should first know about sensors, effectors, and actuators.
Sensor: Sensor is a device which detects the change in the environment and sends the information to other electronic devices. An agent observes its environment through sensors.
Actuators: Actuators are the component of machines that converts energy into motion. The actuators are only responsible for moving and controlling a system. An actuator can be an electric motor, gears, rails, etc.
Effectors: Effectors are the devices which affect the environment. Effectors can be legs, wheels, arms, fingers, wings, fins, and display screen.



Agent and Environment

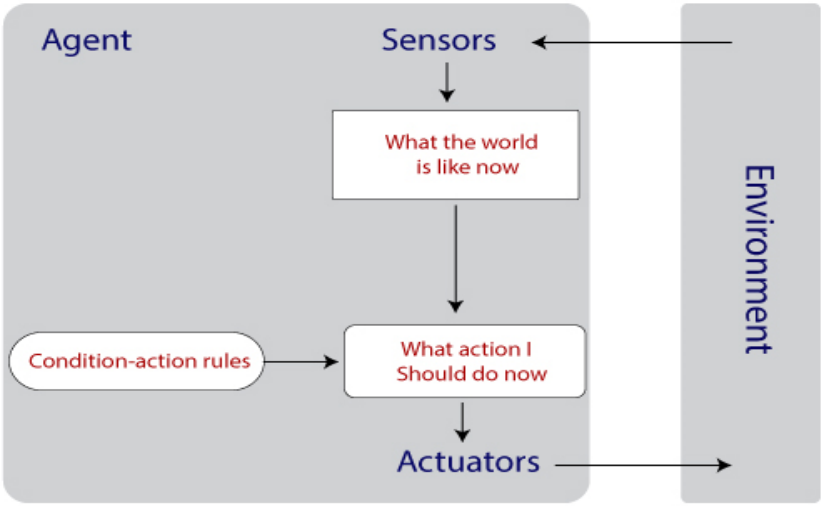
Types of agents:

Agents can be grouped into four classes based on their degree of perceived intelligence and capability :

- ✓ Simple Reflex Agents
- ✓ Model-Based Reflex Agents
- ✓ Goal-Based Agents
- ✓ Utility-Based Agents

Simple reflex agents:

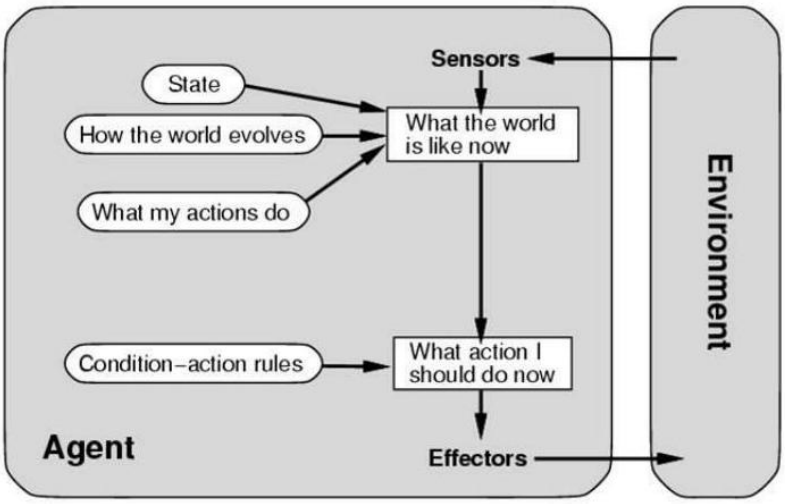
- Simple reflex agents ignore the rest of the percept history and act only on the basis of the current percept.
- The agent function is based on the condition-action rule.
- If the condition is true, then the action is taken, else not. This agent function only succeeds when the environment is fully observable.



Schematic diagram of a simple-reflex agent

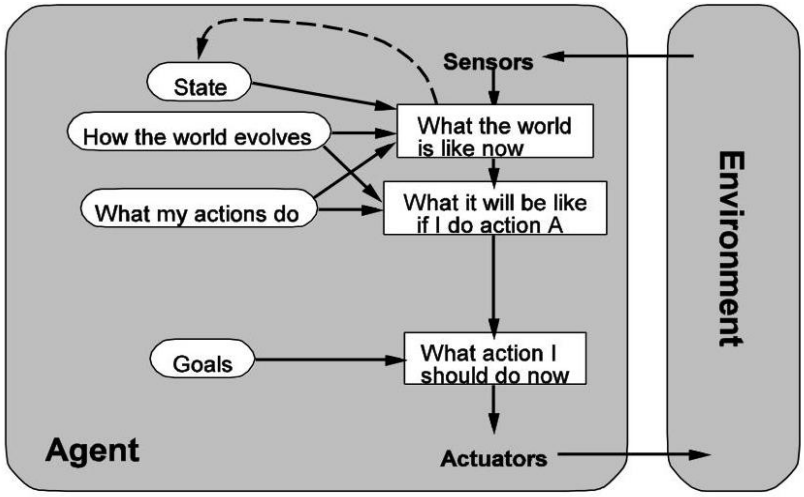
Model-based reflex agents:

- The Model-based agent can work in a partially observable environment, and track the situation.
- A model-based agent has two important factors:
- Model: It is knowledge about "how things happen in the world," so it is called a Model-based agent.
- Internal State: It is a representation of the current state based on percept history.



Goal-based agents:

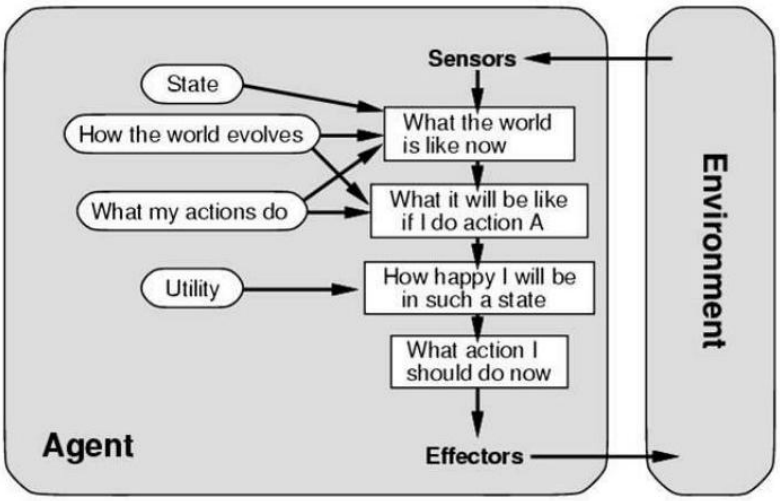
- A goal-based agent has an agenda.
- It operates based on a goal in front of it and makes decisions based on how best to reach that goal.
- A goal-based agent operates as a search and planning function, meaning it targets the goal ahead and finds the right action in order to reach it.
- Expansion of model-based agent.



Utility-based agents:

- utility-based agent is an agent that acts based not only on what the goal is, but the best way to reach that goal.
- The Utility-based agent is useful when there are multiple possible alternatives, and an agent has to choose in order to perform the best action.

- The term utility can be used to describe how "happy" the agent is.



- **Intelligent Agents:**
  Intelligent agents represent a subset of AI systems demonstrating intelligent behaviour, including adaptive learning, planning, and problem-solving. It operates in dynamic environments, where it makes decisions based on the information available to them. These agents dynamically adjust their behaviour, learning from past experiences to improve their approach and aiming for accurate solutions.

  The design of an intelligent agent typically involves four key components:
  1. **Perception:** Agents have sensors or mechanisms to observe and perceive aspects of their environment. This may involve collecting data from the physical world, accessing databases, or receiving input from other software components.
  2. **Reasoning:** Agents possess computational or cognitive capabilities to process the information they perceive. They use algorithms, logic, or machine learning techniques to analyze data, make inferences, and derive insights from the available information.
  3. **Decision-Making:** Based on their perception and reasoning, agents make decisions about the actions they should take to achieve their goals. These decisions are guided by predefined objectives, which may include optimizing certain criteria or satisfying specific constraints.
  4. **Action:** Agents execute actions in their environment to affect change and progress towards their goals. These actions can range from simple operations, such as sending a message or adjusting parameters, to more

complex tasks, such as navigating a virtual world or controlling physical devices.

Examples of Intelligent Agents include self-driving cars, recommendation systems, virtual assistants,

Following are the main four rules for an AI agent:

Rule 1: An AI agent must have the ability to perceive the environment.
Rule 2: The observation must be used to make decisions.
Rule 3: Decision should result in an action.
Rule 4: The action taken by an AI agent must be a rational action.

Applications of Intelligent Agents

Intelligent agents find applications across a wide range of domains, revolutionizing industries and enhancing human capabilities. Some notable applications include:

1. **Autonomous Systems:** Intelligent agents power autonomous vehicles, drones, and robots, enabling them to perceive their surroundings, navigate complex environments, and make decisions in real-time.
2. **Personal Assistants:** Virtual personal assistants like Siri, Alexa, and Google Assistant employ intelligent agents to understand user queries, retrieve relevant information, and perform tasks such as scheduling appointments, setting reminders, and controlling smart home devices.
3. **Recommendation Systems:** E-commerce platforms, streaming services, and social media platforms utilize intelligent agents to analyze user preferences and behavior, providing personalized recommendations for products, movies, music, and content.
4. **Financial Trading:** Intelligent agents are employed in algorithmic trading systems to analyze market data, identify trading opportunities, and execute trades autonomously, maximizing returns and minimizing risks.

Rational Agent:

A Rational agent is one that does the right thing. we say that the right action is the one that will cause the agent to be most successful. That leaves us with the problem of deciding how and when to evaluate the agent's success.

We use the term performance measure for the how—the criteria that determine how successful an agent is.

- Ex-Agent cleaning the dirty floor
- Performance Measure-Amount of dirt collected
- When to measure-Weekly for better results

What is rational at any given time depends on four things:
• The performance measure defining the criterion of success
• The agent‘s prior knowledge of the environment
• The actions that the agent can perform
• The agent‘s percept sequence up to now

Structure of an AI Agent
The task of AI is to design an agent program which implements the agent function. The structure of an intelligent agent is a combination of architecture and agent program. It can be viewed as:
Agent = Architecture + Agent program

Following are the main three terms involved in the structure of an AI agent:
Architecture: Architecture is machinery that an AI agent executes on.
Agent Function: Agent function is used to map a percept to an action.

$$f:P^* \rightarrow A$$

Agent program: Agent program is an implementation of agent function. An agent program executes on the physical architecture to produce function f.

PEAS Representation
PEAS is a type of model on which an AI agent works upon. When we define an AI agent or rational agent, then we can group its properties under PEAS representation model. It is made up of four words:
P: Performance measure
E: Environment
A: Actuators
S: Sensors
Here performance measure is the objective for the success of an agent's behaviour.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments	Touchscreen/voice entry of symptoms and findings
Satellite image analysis system	Correct categorization of objects, terrain	Orbiting satellite, downlink, weather	Display of scene categorization	High-resolution digital camera
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, tactile and joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, raw materials, operators	Valves, pumps, heaters, stirrers, displays	Temperature, pressure, flow, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, feedback, speech	Keyboard entry, voice

Figure 2.5 Examples of agent types and their PEAS descriptions.

ENVIRONMENTS:

The environment is where agent lives, operate and provide the agent with something to sense and act upon it. An environment is mostly said to be non-feministic.

Environment-Types:

1. Accessible vs. inaccessible or Fully observable vs Partially Observable:

If an agent sensor can sense or access the complete state of an environment at each point of time then it is a fully observable environment, else it is partially observable.

2. Deterministic vs. Stochastic:

If the next state of the environment is completely determined by the current state and the actions selected by the agents, then we say the environment is deterministic

3. Episodic vs. nonepisodic:

The agent's experience is divided into "episodes." Each episode consists of the agent perceiving and then acting. The quality of its action depends just on the episode itself, because subsequent episodes do not depend on what actions occur in previous episodes.

Episodic environments are much simpler because the agent does not need to think ahead.

4. 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.
5. Discrete vs. continuous:
If there are a limited number of distinct, clearly defined percepts and actions we say that the environment is discrete. Otherwise, it is continuous.

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

Figure 2.6 Examples of task environments and their characteristics.

Problem Solving Agents:

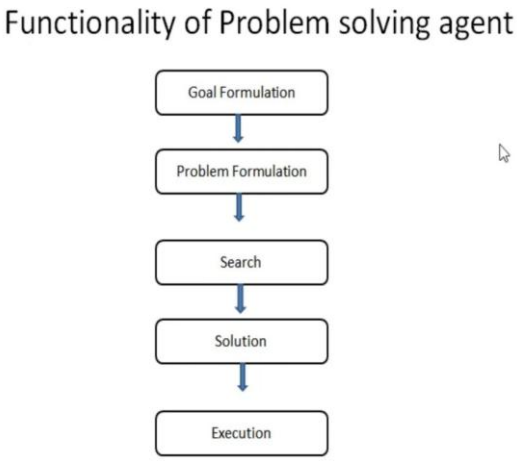
Problem Solving Agents decide what to do by finding a sequence of actions that leads to a desirable state or solution.
An agent may need to plan when the best course of action is not immediately visible. They may need to think through a series of moves that will lead them to their goal state. Such an agent is known as a **problem solving agent**, and the computation it does is known as a **search**.

The problem solving agent follows this four phase problem solving process:

1. **Goal Formulation:** This is the first and most basic phase in problem solving. It arranges specific steps to establish a target/goal that demands some activity to reach it. AI agents are now used to formulate goals.
2. **Problem Formulation:** It is one of the fundamental steps in problem-solving that determines what action should be taken to reach the goal.
3. **Search:** After the Goal and Problem Formulation, the agent simulates sequences of actions and has to look for a sequence of actions that reaches the goal. This process is called **search,** and the sequence is called

a **solution**. The agent might have to simulate multiple sequences that do not reach the goal, but eventually, it will find a solution, or it will find that no solution is possible. A search algorithm takes a problem as input and outputs a sequence of actions.

4. **Execution:** After the search phase, the agent can now execute the actions that are recommended by the search algorithm, one at a time. This final stage is known as the execution phase.



Problem Formulation: A formal definition of a problem consists of five components:

1. Initial State
2. Actions
3. Transition Model
4. Goal Test
5. Path Cost

Initial State

It is the agent's starting state or initial step towards its goal. For example, if a taxi agent needs to travel to a location(B), but the taxi is already at location(A), the problem's initial state would be the location (A).

Actions

It is a description of the possible actions that the agent can take. Given a state s, **Actions**(**s**) returns the actions that can be executed in s. Each of these actions is said to be appropriate in s.

Transition Model

It describes what each action does. It is specified by a function **Result**(**s, a**) that returns the state that results from doing action an in state s.

The initial state, actions, and transition model together define the **state space** of a problem, a set of all states reachable from the initial state by any

sequence of actions. The state space forms a graph in which the nodes are states, and the links between the nodes are actions.

Goal Test

It determines if the given state is a goal state. Sometimes there is an explicit list of potential goal states, and the test merely verifies whether the provided state is one of them. The goal is sometimes expressed via an abstract attribute rather than an explicitly enumerated set of conditions.

Path Cost

It assigns a numerical cost to each path that leads to the goal. The problem solving agents choose a cost function that matches its performance measure. Remember that the optimal solution has the lowest path cost of all the solutions.

EXAMPLES:

Well Defined problems and solutions:

A problem can be defined formally by 4 components:

- ✓ The initial state of the agent is the state where the agent starts in. In this case, the initial state can be described as In: Arad
- ✓ The possible actions available to the agent, corresponding to each of the state the agent resides in.
  For example, ACTIONS(In: Arad) = {Go: Sibiu, Go: Timisoara, Go: Zerind}. Actions are also known as operations.
- ✓ A description of what each action does. the formal name for this is Transition model, Specified by the function Result(s,a) that returns the state that results from the action a in state s.
  We also use the term Successor to refer to any state reachable from a given state by a single action.
  For example: Result(In(Arad),GO(Zerind))=In(Zerind)

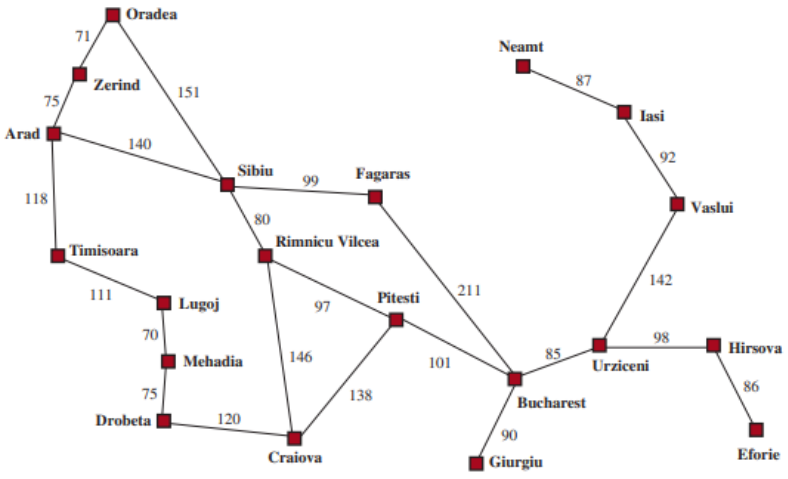


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

Together the initial state, actions and transition model implicitly defines the state space of the problem State space: set of all states reachable from the initial state by any sequence of actions

- ✓ The goal test, determining whether the current state is a goal state. Here, the goal state is {In: Bucharest}
- ✓ The path cost function, which determine the cost of each path, which is reflecting in the performance measure.
- ✓ we define the cost function as c(s, a, s‘), where s is the current state and a is the action performed by the agent to reach state s‘.

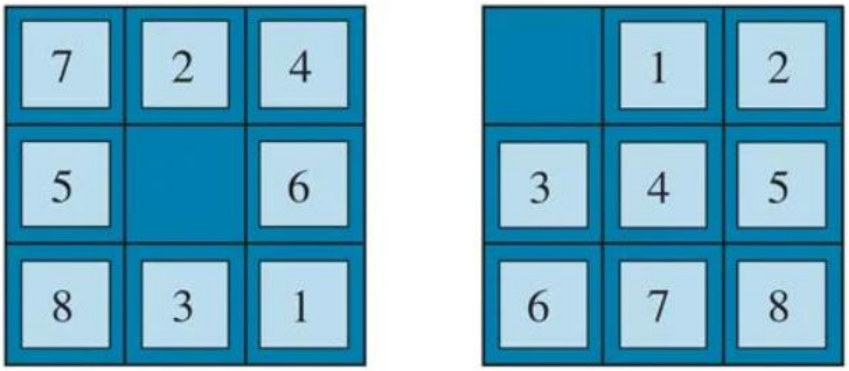
8 Puzzle Problem

In a **sliding-tile puzzle**, a number of tiles (sometimes called blocks or pieces) are arranged in a grid with one or more blank spaces so that some of the tiles can slide into the blank space. One variant is the Rush Hour puzzle, in which cars and trucks slide around a 6 x 6 grid in an attempt to free a car from the traffic jam. Perhaps the best-known variant is the **8- puzzle** (see Figure below), which consists of a 3 x 3 grid with eight numbered tiles and one blank space, and the **15-puzzle** on a 4 x 4 grid. The object is to reach a specified goal state, such as the one shown on the right of the figure. The standard formulation of the 8 puzzles is as follows:

STATES: A state description specifies the location of each of the tiles.
INITIAL STATE: Any state can be designated as the initial state. (Note that a parity property partitions the state space—any given goal can be reached from exactly half of the possible initial states.)

ACTIONS: While in the physical world it is a tile that slides, the simplest way of describing action is to think of the blank space moving **Left**, **Right**, **Up**, or **Down**. If the blank is at an edge or corner then not all actions will be applicable.

TRANSITION MODEL: Maps a state and action to a resulting state; for example, if we apply Left to the start state in the Figure below, the resulting state has the 5 and the blank switched.



A typical instance of the 8-puzzle

GOAL STATE: It identifies whether we have reached the correct goal state. Although any state could be the goal, we typically specify a state with the numbers in order, as in the Figure above.

ACTION COST: Each action costs 1

STATE SPACE SEARCH/PROBLEM SPACE SEARCH:

The state space representation forms the basis of most of the AI methods. State-space consists of all the possible states together with actions to solve a specific problem. In the graphical representation of state space (such as trees), the states are represented by nodes while the actions are represented by arcs. Any state space has an initial 'Start' node and an ending 'Goal' node or multiple Goal States. The path from the initial start node to the final goal node is known as the 'Solution' for the particular problem.

Formal Description of the problem:

1. Define a state space that contains all the possible configurations of the relevant objects.
2. Specify one or more states within that space that describe possible situations from which the problem solving process may start (initial state)
3. Specify one or more states that would be acceptable as solutions to the problem. (goal states) Specify a set of rules that describe the actions (operations) available.

For a clear understanding of state space, consider the following problem [Figure 2].

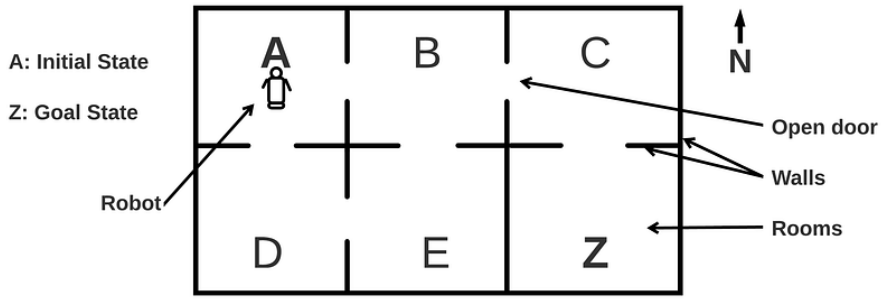


Figure 2: Example Problem

Imagine there is a robot in room 'A' (initial state), and it needs to go to room 'Z' (goal state). We can draw a state space in terms of a tree if we consider all the possible movements of the robot in each room (node). For example, when the robot is at initial state A, he can either go to B or D. When the robot moved to the next state B, he can move to C, E, or back to A [Figure 3].

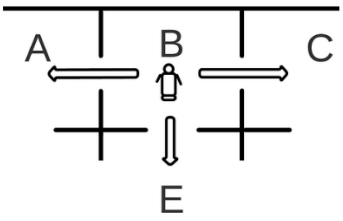


Figure 3: Possible paths for robot at state B

Based on all the possible movements of the robot at each state we can draw the state space for the above scenario as follows:

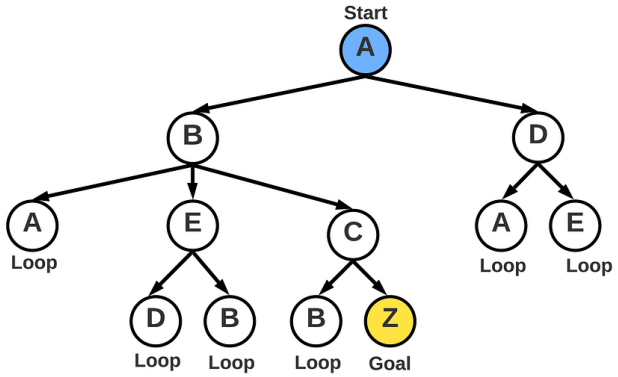


Figure 4: State-space diagram

We can identify many paths beside the direct path A, B, C, Z.
Ex: A B C Z
A B A B C Z
A D E B C Z
A D E B A B C Z
....
It can be observed that some paths are shorter while others are longer. The real problem arises here. We can figure out the best/shortest path in cases like the above example where the state space is small. But imagine a state-space with hundreds of thousands of nodes. How can we explore such a state-space? The solution is nothing but search algorithms.
Since the state spaces are very large in general, we employ search algorithms to navigate through the state-space effectively and efficiently. A search algorithm defines how to find the path (solution) from the initial state to the goal state. Different algorithms define different methods to move from the current state (node) to the next state (node). Some algorithms provide just the systematic ways to explore the state space while others tell how to explore effectively and efficiently.

Search algorithms

Since the state spaces are very large in general, we employ search algorithms to navigate through the state-space effectively and efficiently. A search algorithm defines how to find the path (solution) from the initial state to the goal state. Different algorithms define different methods to move from the current state (node) to the next state (node). Some algorithms provide just the systematic ways to explore the state space while others tell how to explore effectively and efficiently.

The search algorithms are of two types as uninformed and informed search algorithms [Figure 5]. Informed search algorithms provide just a systematic way to explore the state space and no additional information are provided to support the search process. Breadth-first search, Uniform search, Depth-first search, Depth limited search, Iterative deepening search, and Bi-direction search are the 06 main uninformed search algorithms.

On the other hand, informed search algorithms such as Greedy search and A* search algorithms are based on additional information which makes the searching procedure both systematic and effective.

Search Algorithm Terminologies:

- **Search:** Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:
    - Search Space: Search space represents a set of possible solutions, which a system may have.
    - Start State: It is a state from where agent begins the search.
    - Goal test: It is a function which observe the current state and returns whether the goal state is achieved or not.
- **Search tree:** A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
- **Actions:** It gives the description of all the available actions to the agent.
- **Transition model:** A description of what each action do, can be represented as a transition model.
- **Path Cost:** It is a function which assigns a numeric cost to each path.
- **Solution:** It is an action sequence which leads from the start node to the goal node.
- **Optimal Solution:** If a solution has the lowest cost among all solutions.

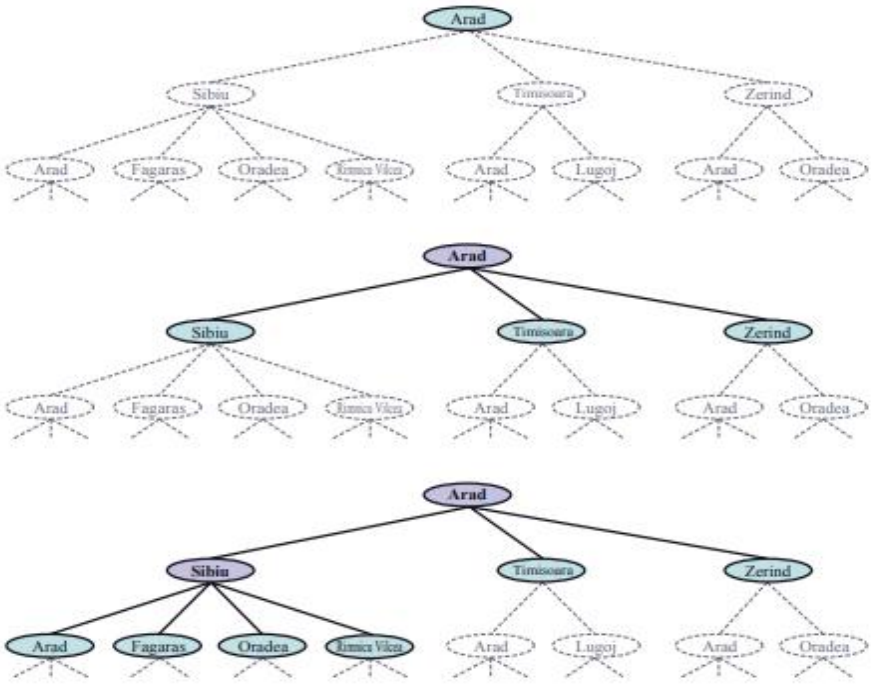


Figure 3.4 Three partial search trees for finding a route from Arad to Bucharest. Nodes that have been *expanded* are lavender with bold letters; nodes on the frontier that have been *generated* but not yet expanded are in green; the set of states corresponding to these two types of nodes are said to have been *reached*. Nodes that could be generated next are shown in faint dashed lines. Notice in the bottom tree there is a cycle from Arad to Sibiu to Arad; that can't be an optimal path, so search should not continue from there.

Properties of Search Algorithms:

Following are the four essential properties of search algorithms to compare the efficiency of these algorithms:

1. **Completeness:** Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not (The algorithm should not miss any node before reaching any goal node/end of the state space).
2. **Cost Optimality:** Ability to find the highest quality solution (Does it find a solution with the lowest path cost of all solutions?).
3. **Time complexity:** How long the algorithm takes to process the nodes. (The execution time of the algorithm/ CPU time)
4. **Space complexity:** How much memory is used by the algorithm to store nodes.

Time and space complexity are considered with respect to some measure of the problem difficulty. In theoretical computer science, the typical measure is the size of the state-space graph, |V| + |E|, where |V| is the number of vertices (state nodes) of the graph and |E| is the number of edges (distinct state/action pairs). This is appropriate when the graph is an explicit data structure, such as the map of Romania. But in many AI problems, the graph is represented only implicitly by the initial state, actions, and transition model.

For an implicit state space, complexity can be measured in terms of d, the depth or number of actions in an optimal solution; m, the maximum number of actions in any path; and b, the branching factor or number of successors of a node that need to be considered.

Types of search algorithms

Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.

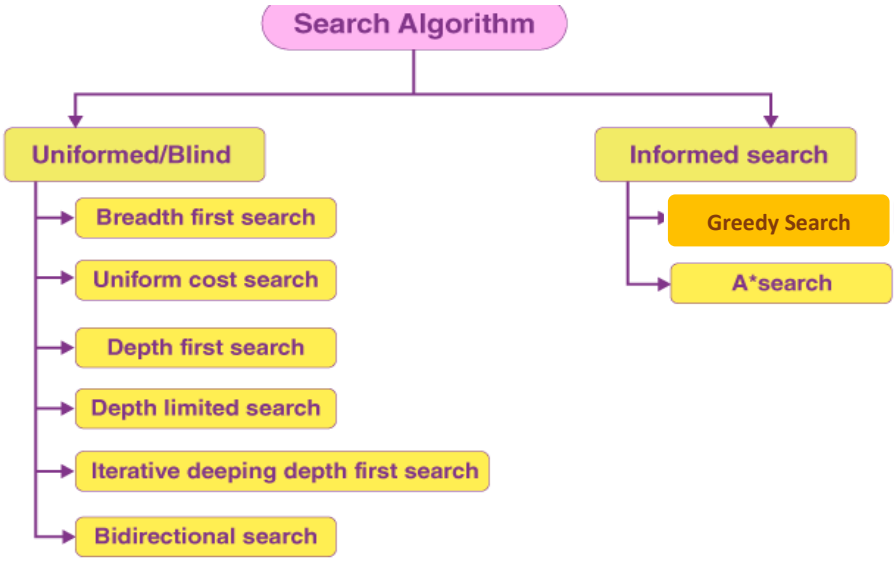


Figure 5: AI Search Algorithms Classification

1. Uninformed Search Algorithms

An uninformed search algorithm is given no clue about how close a state is to the goal(s). For example, consider our agent in Arad with the goal of reaching Bucharest. An uninformed agent with no knowledge of Romanian geography has no clue whether going to Zerind or Sibiu is a better first step. In contrast, an informed agent (Section 3.5) who knows the location of each city knows that Sibiu is much closer to Bucharest and thus more likely to be on the shortest path.

1.1 Breadth-first Search (BFS)

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.

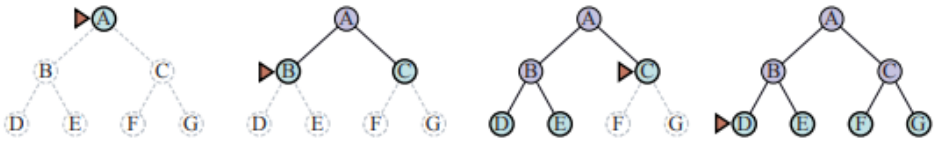


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.
- It also helps in finding the shortest path in goal state, since it needs all nodes at the same hierarchical level before making a move to nodes at lower levels.
- It is also very easy to comprehend with the help of this we can assign the higher rank among path types.

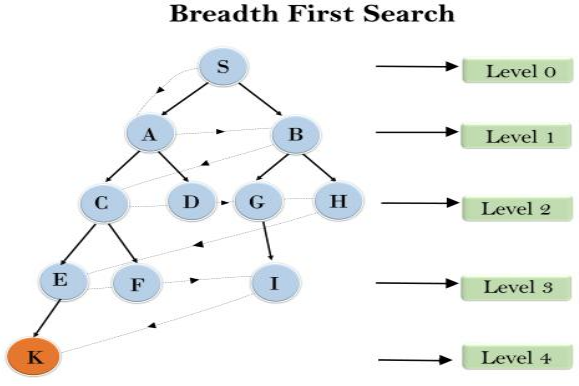
Disadvantages:

- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.
- It can be very inefficient approach for searching through deeply layered spaces, as it needs to thoroughly explore all nodes at each level before moving on to the next

Example:

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

1. S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K



Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

$T(b) = 1+b^2+b^3+.....+ b^d = O(b^d)$

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

Breadth First Search (BFS) Pseudocode

The below pseudocode outlines the Breadth-First Search algorithm, here the problem represents the pathfinding problem.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

Figure 3.9 Breadth-first search and uniform-cost search algorithms.

Applications Of Breadth-First Search Algorithm

- ✓ **GPS Navigation systems:** Breadth-First Search is one of the best algorithms used to find Neighbouring locations by using the GPS system.
- ✓ **Broadcasting:** Networking makes use of what we call as packets for communication. These packets follow a traversal method to reach various networking nodes. One of the most commonly used traversal methods is Breadth-First Search. It is being used as an algorithm that is used to communicate broadcasted packets across all the nodes in a network.

1. 2. Depth-first Search (DFS)

- Depth-first search is a recursive algorithm for traversing a tree or graph data structure. It generally starts by exploring the deepest node in the frontier. Starting at the root node, the algorithm proceeds to search to the deepest level of the search tree until nodes with no successors are reached. Suppose the node with unexpanded successors is encountered then the search backtracks to the next deepest node to explore alternative paths.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).
- With the help of this we can stores the route which is being tracked in memory to save time as it only needs to keep one at a particular time.

Disadvantage:

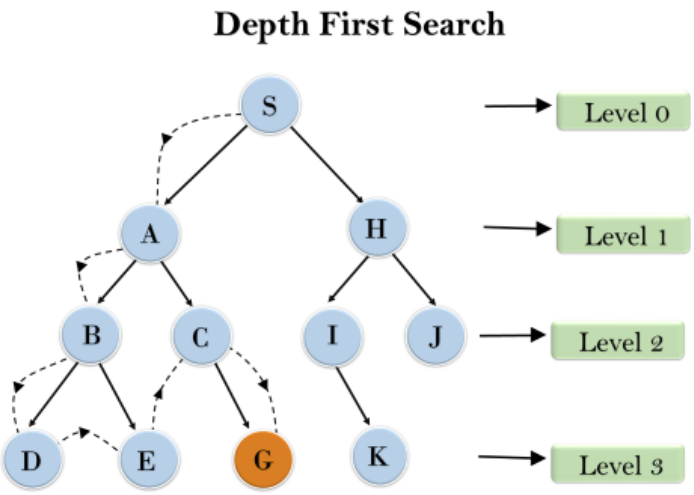
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.
- The depth-first search (DFS) algorithm does not always find the shortest path to a solution.

Example:

In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:

Root node--->Left node ----> right node.

It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.



Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.
Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:
$T(n)= 1+ n^2+ n^3 +.....+ n^m=O(n^m)$
Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)
Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.
Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

Depth First Search (DFS) Algorithm Pseudocode
Take a look at the below Pseudocode which explains the working of DFS

```
function findPath(robot, start, goal):
       stack ← empty stack
       visited ← empty set
       stack.push(start)

        while not stack.isEmpty( ):
               current ← stack.pop( )
               if current == goal:
                  return "Path found"
```

```
            visited.add(current)
            neighbors ← robot.getNeighbors(current)

            for neighbor in neighbors:
                  if neighbor not in visted:
                     stack.push(neighbor)
      return "Path not found"
```

Step wise DFS Pseudocode Explanation

1. Initialize an empty **stack** and an empty set for visited vertices.
2. Push the start vertex onto the stack.
3. While the stack is not empty:
    - Pop the current vertex.
    - If it's the goal vertex, return "Path found".
    - Add the current vertex to the visited set.
    - Get the neighbors of the current vertex.
    - For each neighbor not visited, push it onto the stack.
4. If the loop completes without finding the goal, return "Path not found".

Applications Of Depth-First Search Algorithm

- ✓ **Finding Strongly Connected Components of a graph:** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.
- ✓ **Web crawlers:** Depth-first search can be used in the implementation of web crawlers to explore the links on a website.
- ✓ **Model checking:** Depth-first search can be used in model checking, which is the process of checking that a model of a system meets a certain set of properties.

1.3 Depth-Limited Search Algorithm:

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

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

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

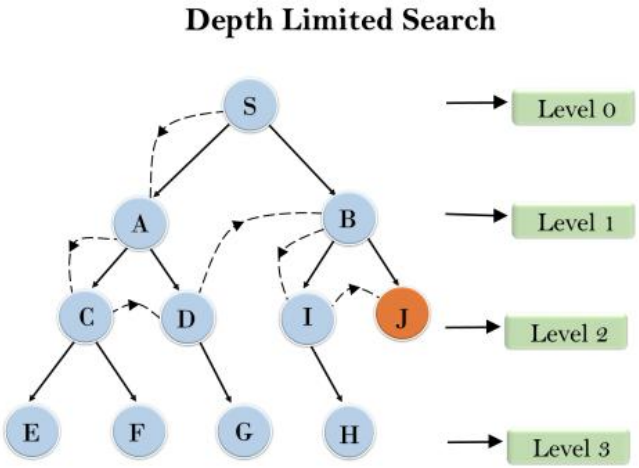
Advantages:

- Depth-Limited Search will restrict the search depth of the tree, thus, the algorithm will require fewer memory resources than the straight BFS (Breadth-First Search) and IDDFS (Iterative Deepening Depth-First Search). After all, this implies automatic selection of more segments of the search space and the consequent why consumption of the resources. Due to the depth restriction, DLS omits a predicament of holding the entire search tree within memory which contemplatively leaves room for a more memory-efficient vice for solving a particular kind of problems.
- When there is a leaf node depth which is as large as the highest level allowed, do not describe its children, and then discard it from the stack.
- Depth-Limited Search does not explain the infinite loops which can arise in classical when there are cycles in graph of cities.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.
- The effectiveness of the Depth-Limited Search (DLS) algorithm is largely dependent on the depth limit specified. If the depth limit is set too low, the algorithm may fail to find the solution altogether.

Example:



Completeness: DLS search algorithm is complete if the solution is above the depth-limit.

Time Complexity: Time complexity of DLS algorithm is $O(b^{\ell})$ where b is the branching factor of the search tree, and l is the depth limit.

Space Complexity: Space complexity of DLS algorithm is $O(b \times \ell)$ where b is the branching factor of the search tree, and l is the depth limit.

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

1.4. Iterative deepening depth-first Search:

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.

This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found. This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Here are the steps for Iterative deepening depth first search algorithm:

- Set the depth limit to 0.
- Perform DFS to the depth limit.
- If the goal state is found, return it.
- If the goal state is not found and the maximum depth has not been reached, increment the depth limit and repeat steps 2-4.
- If the goal state is not found and the maximum depth has been reached, terminate the search and return failure.

Advantages:

- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.
- It is a type of straightforward which is used to put into practice since it builds upon the conventional depth-first search algorithm.
- It is a type of search algorithm which provides guarantees to find the optimal solution, as long as the cost of each edge in the search space is the same.
- It is a type of complete algorithm, and the meaning of this is it will always find a solution if one exists.
- The Iterative Deepening Depth-First Search (IDDFS) algorithm uses less memory compared to Breadth-First Search (BFS) because it only stores the current path in memory, rather than the entire search tree.

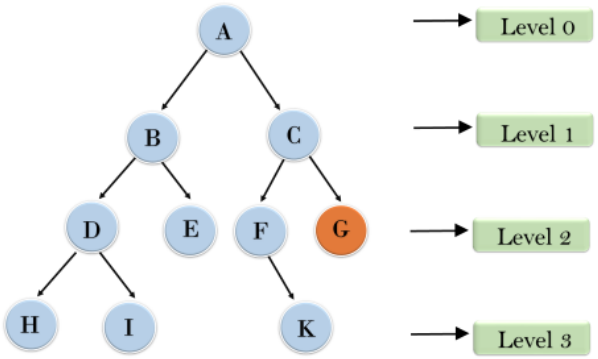
Disadvantages:

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

Example:

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

Iterative deepening depth first search



1'st Iteration-----> A
2'nd Iteration----> A, B, C
3'rd Iteration------>A, B, D, E, C, F, G
4'th Iteration------>A, B, D, H, I, E, C, F, K, G

In the fourth iteration, the algorithm will find the goal node.
Completeness:
This algorithm is complete is if the branching factor is finite.
Time Complexity:
Let's suppose b is the branching factor and depth is d then the worst-case time complexity is $O(b^d)$.
Space Complexity:
The space complexity of IDDFS will be O(bd).
Optimal:
IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

1.5. Uniform-cost Search Algorithm:
Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

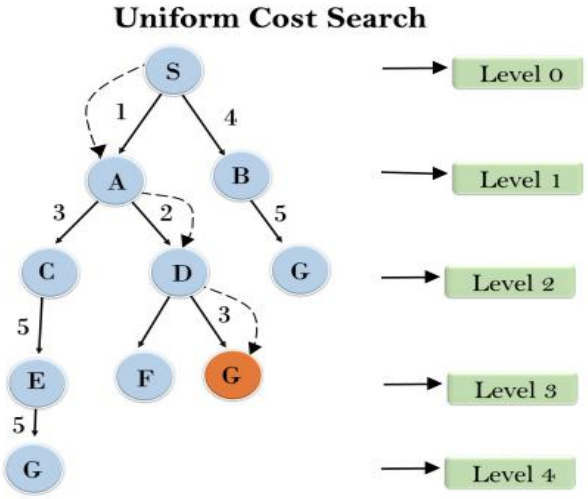
Advantages:

- Uniform cost search is optimal because at every state the path with the least cost is chosen.
- It is an efficient when the edge weights are small, as it explores the paths in an order that ensures that the shortest path is found early.
- It's a fundamental search method that is not overly complex, making it accessible for many users.
- It is a type of comprehensive algorithm that will find a solution if one exists. This means the algorithm is complete, ensuring it can locate a solution whenever a viable one is available. The algorithm covers all the necessary steps to arrive at a resolution.

Disadvantages:

- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.
- When in operation, UCS shall know all the edge weights to start off the search.
- This search holds constant the list of the nodes that it has already discovered in a priority queue. Such is a much weightier thing if you have a large graph. Algorithm allocates the memory by storing the path sequence of prioritizes, which can be memory intensive as the graph gets larger. With the help of Uniform cost search we can end up with the problem if the graph has edge's cycles with smaller cost than that of the shortest path.
- The Uniform cost search will keep deploying priority queue so that the paths explored can be stored in any case as the graph size can be even bigger that can eventually result in too much memory being used.

Example:



Completeness:
Uniform-cost search is complete, such as if there is a solution, UCS will find it.
Time Complexity:
Let C* is Cost of the optimal solution, and ε is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.
Hence, the worst-case time complexity of Uniform-cost search is $O(b^{1 + [C^*/\varepsilon]})$/.
Space Complexity:
The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $O(b^{1 + [C^*/\varepsilon]})$.
Optimal: Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

1.6. Bidirectional Search Algorithm:

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

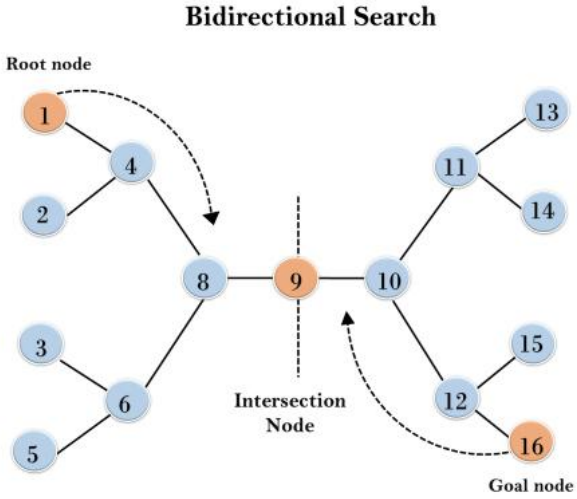
- Bidirectional search is fast.
- Bidirectional search requires less memory
- The graph can be extremely helpful when it is very large in size and there is no way to make it smaller. In such cases, using this tool becomes particularly useful.
- The cost of expanding nodes can be high in certain cases. In such scenarios, using this approach can help reduce the number of nodes that need to be expanded.

Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.
- Finding an efficient way to check if a match exists between search trees can be tricky, which can increase the time it takes to complete the task.

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

The algorithm terminates at node 9 where two searches meet.



Comparing uninformed search algorithms

This comparison is for tree-like search versions which don't check for repeated states. For graph searches which do check, the main differences are that depth-first search is complete for finite state spaces, and the space and time complexities are bounded by the size of the state space (the number of vertices and edges, |V|+|E|).

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

2. Informed Search Algorithms

In informed search algorithms additional information is used to make the search more efficient and effective. That additional information is called **heuristics**. Heuristics are not theories but some common sense experience like information.

In informed search algorithms, to find the best node to be visited next, we use an *evaluation function* **f(n)** which assists the child node to decide on which node to be visited next. Then we traverse to the next node with the **least f(n)** value. Depending on the f(n), we have two informed search algorithms as greedy search and A* search algorithms.

2.1 Greedy Search Algorithms

In greedy search, the heuristic values of child nodes are considered. The path is determined by calculating the path with the nodes with the lowest heuristic values. Another fact to be noticed is that usually the initial node has the highest heuristic value and the goal node has the lowest. But there can be exceptions like getting a mid-range value for the initial node also.

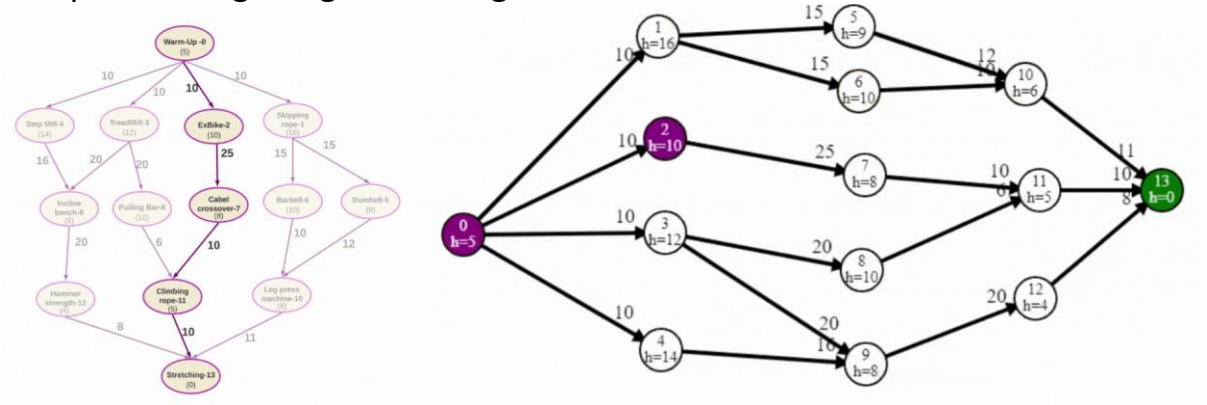


Figure 12: State-Space (left) and state-space traversal (right) in greedy search

f(n) = h(n)

= Summation (Heuristic values of nodes)

= Node0 + Node2 + Node7 + Node11 + Node13

= 5 + 10 + 8 + 5 + 0

= 28

Path: 0, 2, 7, 11, 13

In greedy search, you can see we do not visit all the child nodes of a particular node. We find heuristics of each child node only the one with the lowest value is inserted to the OPEN list to be processed. Therefore greedy search is not complete. As well as this is not the best path (not the shortest path - we found this path in uniform cost search). Therefore greedy search is not optimal, also.

As a solution, we should consider not only the heuristic value but also the path cost. Here, comes the A* algorithm.

2.2 A* Search Algorithms

In the A* algorithm, we consider both path cost and heuristics. In A* the f(n) function comprises two components: path cost [g(n)] and heuristic value ([h(n)]. The f(n) value for node 'a' can be calculated as follows:

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

$f(n)_a$ = Evaluation value at the particular node (node 'a')
$g(n)_a$ = Total path cost from start node to particular node(node 'a')
$h(n)_a$ = The heuristic value of the particular node(node 'a')
Now, we will find the best path according to the A* search algorithm for our previous problem. We have to find the f(n) value for each node and select the child node with the least f(n) value and traverse until meeting the goal node. In the example, only the f(n) values of goals in the path are mentioned.

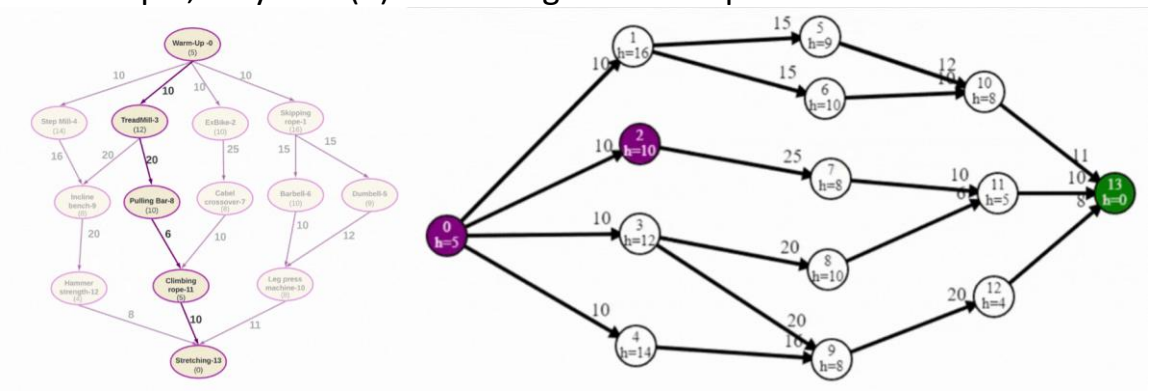


Figure 13: State-Space (left) and state-space traversal (right) in A* search

At Node-0 :
$f(n)_0 = 0 + 5 = 5$
At Node-3:
$f(n)_3 = 10 + 12 = 22$
At Node-8:
$f(n)_8 = 30 + 10 = 40$
At Node-11:
$f(n)_{11} = 36 + 5 = 41$
At Node-13:
$f(n)_{13} = 46 + 0 = 46$

$f(n)_{total} = f(n)_0 + f(n)_3 + f(n)_8 + f(n)_{11} + f(n)_{13} = 154$
Path: 0, 3, 8, 11, 13

A* algorithm is complete since it checks all the nodes before reaching to goal node/end of the state space. It is optimal as well because considering both path cost and heuristic values, therefore the lowest path with the lowest heuristics can be found with A*. One drawback of A* is it stores all the nodes it processes in the memory. Therefore, for a state space of branching factor 'b', and the depth 'd' the space and time complexities of A* are denoted by **O(b^d)**. The time complexity of A* depends on the heuristic values.

Local Search in Artificial Intelligence

Local search algorithms are essential tools in artificial intelligence and optimization, employed to find high-quality solutions in large and complex problem spaces. Key algorithms include Hill-Climbing Search, Simulated Annealing, Local Beam Search, Genetic Algorithms, and Tabu Search.
Each of these methods offers unique strategies and advantages for solving optimization problems.

1. Hill Climbing Search

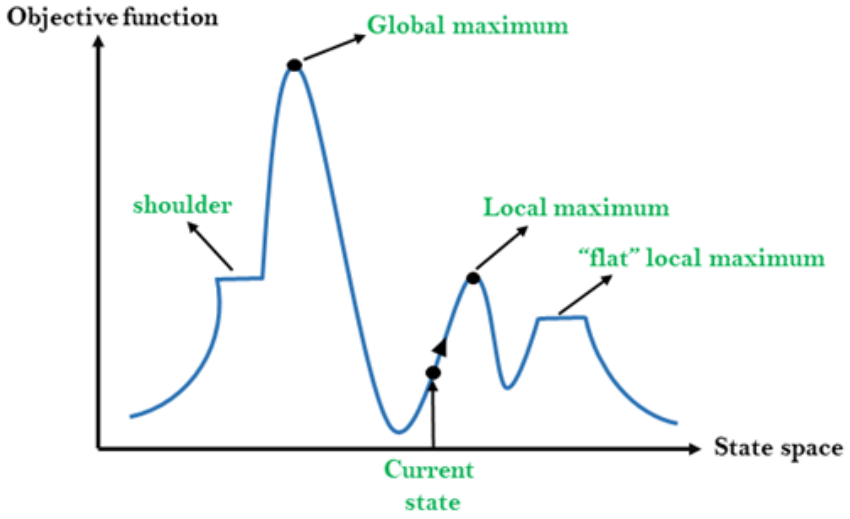
- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbour has a higher value.
- Hill climbing algorithm is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

Features of Hill Climbing:
Following are some main features of Hill Climbing Algorithm:

- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.

- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.
- **Deterministic Nature:**
  Hill Climbing is a deterministic optimization algorithm, which means that given the same initial conditions and the same problem, it will always produce the same result. There is no randomness or uncertainty in its operation.
- **Local Neighbourhood:**
  Hill Climbing is a technique that operates within a small area around the current solution. It explores solutions that are closely related to the current state by making small, gradual changes. This approach allows it to find a solution that is better than the current one although it may not be the global optimum.

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



Different regions in the state space landscape:
Local Maximum: Local maximum is a state which is better than its neighbour states, but there is also another state which is higher than it.

Global Maximum: Global maximum is the best possible state of state space landscape. It has the highest value of objective function.
Current state: It is a state in a landscape diagram where an agent is currently present.
Flat local maximum: It is a flat space in the landscape where all the neighbor states of current states have the same value.
Shoulder: It is a plateau region which has an uphill edge.

Advantages of Hill climb algorithm:
The merits of Hill Climbing algorithm are given below.

1. The first part of the paper is based on Hill's diagrams that can easily put things together, ideal for complex optimization problem. A hill climbing algorithm is first invention of hill climbers.
2. It uses much less RAM for the current problem state and the solutions located around it than comparing the algorithm to a tree search method which will require inspecting the entire tree. Consequently, reducing the total memory resources to be used. Space is what matters solutions should occupy a convenient area to consume as little of memory as possible.
3. When it comes to the acceleration of the hill up, most of the time it brings a closure in the local maximum straight away. This is the route if having quickly getting a solution, outshining acquiring a global maximum, is an incentive.

Disadvantages of Hill Climbing Algorithm

1. Concerning hill climbing, it seems that some solutions do not find the optimum point and remain stuck at a local peak, particularly where the optimization needs to be done in complex environments with many objective functions.
2. It is also superficial because it just seeks for the surrounding solution and does not get farther than that. It could be on a wrong course which is based on a locally optimal solution, and consequently Godbole needs to move far away from current position in the first place.
3. It is highly likely that end result will largely depend on¬ initial setup and state with a precedent of it being the most sensitive factor. It implies that in this case time is the perimeter of the sphere within which people develop their skills dynamically, determining the success.

Example:
To illustrate hill climbing, we will use the 8-queens problem (Figure 4.3). We will use a complete-state formulation, which means that every state has all the

components of a solution, but they might not all be in the right place. In this case every state has 8 queens on the board, one per column. The initial state is chosen at random, and the successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has 8×7=56 successors). The heuristic cost function h is the number of pairs of queens that are attacking each other; this will be zero only for solutions. (It counts as an attack if two pieces are in the same line, even if there is an intervening piece between them.) Figure 4.3(b) shows a state that has h=17. The figure also shows the h values of all its successors.

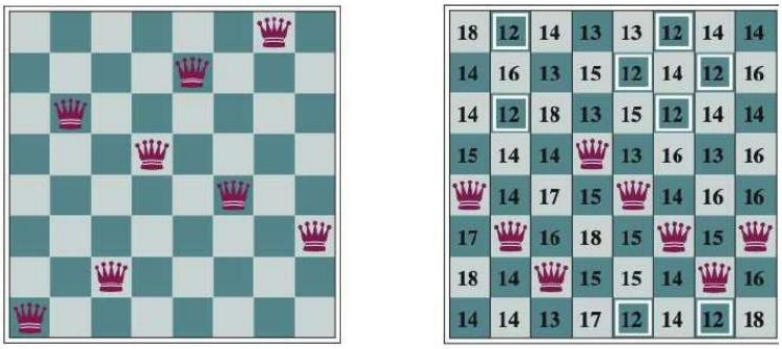
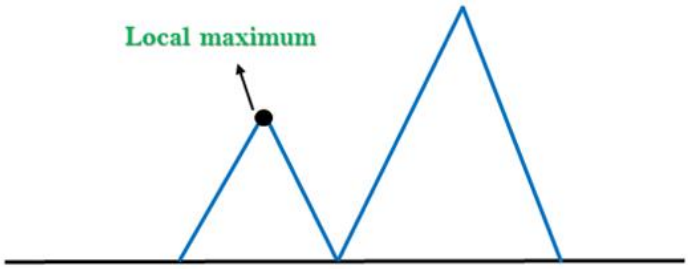


Figure 4.3 (a) The 8-queens problem: place 8 queens on a chess board so that no queen attacks another. (A queen attacks any piece in the same row, column, or diagonal.) This position is almost a solution, except for the two queens in the fourth and seventh columns that attack each other along the diagonal. (b) An 8-queens state with heuristic cost estimate $h=17$. The board shows the value of h for each possible successor obtained by moving a queen within its column. There are 8 moves that are tied for best, with $h=12$. The hill-climbing algorithm will pick one of these.

Problems in Hill Climbing Algorithm:

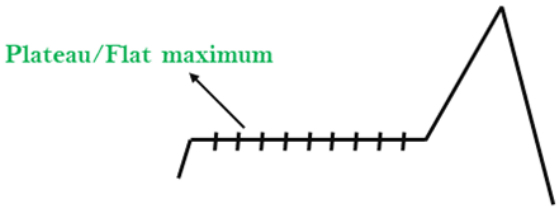
1. Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighbouring states, but there is another state also present which is higher than the local maximum.

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

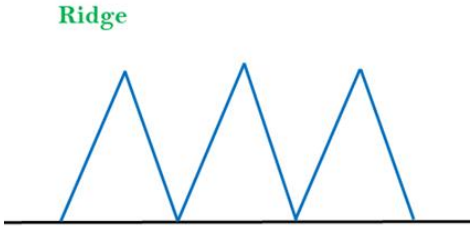


2. Plateau: A plateau is the flat area of the search space in which all the neighbour states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

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



3. Ridges: A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.
Solution: With the use of bidirectional search, or by moving in different directions, we can improve this problem.



2. **Simulated Annealing Search**

A hill-climbing algorithm that never makes —downhill moves towards states with lower value (or higher cost) is guaranteed to be incomplete, because it can stuck on a local maximum. In contrast, a purely random walk –that is, moving to a successor chosen uniformly at random from the set of successors – is complete, but extremely inefficient. Simulated annealing is an algorithm that combines hill-climbing with a random walk in some way that yields both Efficiency and completeness.
simulated annealing algorithm is quite similar to hill climbing. Instead of picking the best move, however, it picks the random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1.
The probability decreases exponentially with the —badness of the move – the amount E by which the evaluation is worsened. The probability also decreases as the "temperature" T goes down: "bad moves are more likely to be allowed at the start when temperature is high, and they become more unlikely as T

decreases. One can prove that if the schedule lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.
Simulated annealing was first used extensively to solve VLSI layout problems. It has been applied widely to factory scheduling and other large-scale optimization tasks.

SA is a metaheuristic optimization technique introduced by Kirkpatrick et al. in 1983 to solve the Travelling Salesman Problem (TSP).
The SA algorithm is based on the annealing process used in metallurgy, where a metal is heated to a high temperature quickly and then gradually cooled. At high temperatures, the atoms move fast, and when the temperature is reduced, their kinetic energy decreases as well. At the end of the annealing process, the atoms fall into a more ordered state, and the material is more ductile and easier to work with.
Similarly, in SA, a search process starts with a high-energy state (an initial solution) and gradually lowers the temperature (a control parameter) until it reaches a state of minimum energy (the optimal solution).

Advantages of Simulated Annealing

- **Ability to Escape Local Minima**: One of the most significant advantages of Simulated Annealing is its ability to escape local minima. The probabilistic acceptance of worse solutions allows the algorithm to explore a broader solution space.
- **Simple Implementation**: The algorithm is relatively easy to implement and can be adapted to a wide range of optimization problems.
- **Global Optimization**: Simulated Annealing can approach a global optimum, especially when paired with a well-designed cooling schedule.
- **Flexibility:** The algorithm is flexible and can be applied to both continuous and discrete optimization problems.

Limitations of Simulated Annealing

- **Parameter Sensitivity:** The performance of Simulated Annealing is highly dependent on the choice of parameters, particularly the initial temperature and cooling schedule.
- **Computational Time:** Since Simulated Annealing requires many iterations, it can be computationally expensive, especially for large problems.
- **Slow Convergence:** The convergence rate is generally slower than more deterministic methods like gradient-based optimization.

Applications of Simulated Annealing

- Simulated Annealing has found widespread use in various fields due to its versatility and effectiveness in solving complex optimization problems. Some notable applications include:
- Traveling Salesman Problem (TSP): In combinatorial optimization, SA is often used to find near-optimal solutions for the TSP, where a salesman must visit a set of cities and return to the origin, minimizing the total travel distance.
- VLSI Design: SA is used in the physical design of integrated circuits, optimizing the layout of components on a chip to minimize area and delay.
- Machine Learning: In machine learning, SA can be used for hyperparameter tuning, where the search space for hyperparameters is large and non-convex.
- Scheduling Problems: SA has been applied to job scheduling, minimizing delays and optimizing resource allocation.
- Protein Folding: In computational biology, SA has been used to predict protein folding by optimizing the conformation of molecules to achieve the lowest energy state.

Local Search in Continuous Spaces

As we know that environment can be discrete or continuous, but the most real-world environment are continuous.

1) It is very difficult to handle continuous state space. The successor for real-world problem are many infinite states.

2) Origin of local search in continuous spaces lies in Newton and Leibnitz in the 17th century.

3) Optimal solution for given problem in continuous spaces can be found with "Local search techniques".

Search and Evaluation Theory

1) Search is basic problem solving technique. Basically it is always related with evolution theory.

2) Charles Darwin is father of evolutionary theory. The theory was based on the origin of species by means of natural selection (1859).

3) The variations (mutations) are well known attributes of reproduction and best features are preserved in next generation in proper propagation.

4) The qualities are inherited or modified. This fact was not associated with Darwin theory.

5) Gregor Mendel (1866) theory found the fact of inheritance. He performed artificial fertilization on peas.

6) DNA module structure was identified by Watson and Crick.

A → Adenin, G- Guanine,
T→ Thymine, C - Cytosine
7) The key difference between stochastic beam search and evolution is that successors are generated from multiple organisms (states) rather than one organism (state).
8) The theory of evolution is very much rich than genetic algorithm.
9) The process of mutations involves duplication, reversals and motion of large group of DNA.
10) Most important is the fact that the genes themselves encode the mechanisms whereby the genome is reproduced and translated into an organism. In genetic algorithms, those mechanisms are a separate program that is not represented within the strings being manipulated.
11) French naturalist Jean Lamarck (1809) proposed a theory of evolution whereby traits acquired by adaptation during an organism's lifetime would be passed on to its offspring. Such a process would be effective, but does not seem to occur in nature.
12) James Baldwin (1896) proposed a superficially similar theory : - that behaviour learned during an organism's lifetime could accelerate the rate of evolution.

For example -

Suppose we want to place three new airports anywhere in India, such that the sum of squared distances from each city to its nearest airport is minimized.
i) The state space, is defined by the co-ordinates of the airports: (x_1, y_1), (x_2, y_2) and (x_3, y_3).
ii) This is a six-dimensional space: We also say that states are defined by six variables. (In general, states are defined by an n-dimensional vector of variables, x).
iii) Moving around in this space corresponds to moving one or more of the airports on the map.
iv) 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, but rather tricky to write down in general.

Problems Associated with Local Search

• Local search methods suffer from local maxima, ridges and plateaux in continuous state spaces just as much as in discrete spaces. Random restarts and simulated annealing can be used and are often helpful. High-dimensional continuous spaces are, however, big places in which it is easy to get lost.