Agents in Artificial Intelligence

In artificial intelligence, an agent is a computer program or system that is designed to perceive its environment, make decisions and take actions to achieve a specific goal or set of goals. The agent operates autonomously, meaning it is not directly controlled by a human operator.

Agents can be classified into different types based on their characteristics, such as whether they are reactive or proactive, whether they have a fixed or dynamic environment, and whether they are single or multi-agent systems.

Reactive agents are those that respond to immediate stimuli from their environment and take actions based on those stimuli. Proactive agents, on the other hand, take initiative and plan ahead to achieve their goals.

The environment in which an agent operates can also be fixed or dynamic. Fixed environments have a static set of rules that do not change, while dynamic environments are constantly changing and require agents to adapt to new situations.

Multi-agent systems involve multiple agents working together to achieve a common goal. These agents may have to coordinate their actions and communicate with each other to achieve their objectives.

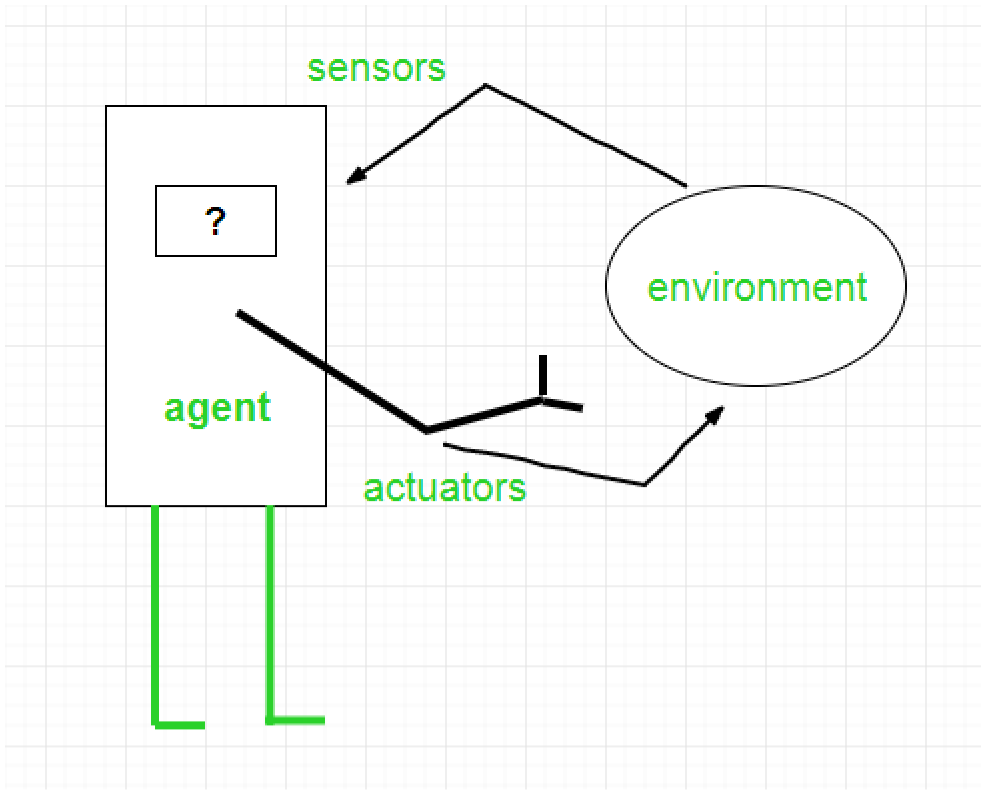
Agents are used in a variety of applications, including robotics, gaming, and intelligent systems. They can be implemented using different programming languages and techniques, including machine learning and natural language processing.

Artificial intelligence is defined as the study of rational agents. A rational agent could be anything that makes decisions, as a person, firm, machine, or software. It carries out an action with the best outcome after considering past and current percepts(agent's perceptual inputs at a given instance). An AI system is composed of an **agent and its environment**. The agents act in their environment. The environment may contain other agents.

An agent is anything that can be viewed as :

- perceiving its environment through **sensors** and
- acting upon that environment through **actuators**

Note: Every agent can perceive its own actions (but not always the effects)



To understand the structure of Intelligent Agents, we should be familiar with *Architecture* and *Agent* programs. **Architecture** is the machinery that the agent executes on. It is a device with sensors and actuators, for example, a robotic car, a camera, and a PC. **Agent program** is an implementation of an agent function. An **agent function** is a map from the percept sequence(history of all that an agent has perceived to date) to an action.

Agent = Architecture + Agent Program

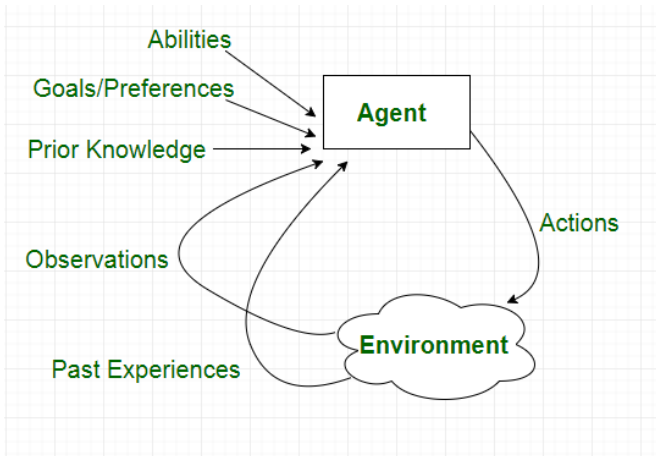
Examples of Agent:

There are many examples of agents in artificial intelligence. Here are a few:

- **Intelligent personal assistants:** These are agents that are designed to help users with various tasks, such as scheduling appointments, sending messages, and setting reminders. Examples of intelligent personal assistants include Siri, Alexa, and Google Assistant.
- **Autonomous robots:** These are agents that are designed to operate autonomously in the physical world. They can perform tasks such as cleaning, sorting, and delivering goods. Examples of autonomous robots include the Roomba vacuum cleaner and the Amazon delivery robot.
- **Gaming agents:** These are agents that are designed to play games, either against human opponents or other agents. Examples of gaming agents include chess-playing agents and poker-playing agents.
- **Fraud detection agents:** These are agents that are designed to detect fraudulent behavior in financial transactions. They can analyze patterns of behavior to identify suspicious activity and

alert authorities. Examples of fraud detection agents include those used by banks and credit card companies.

- **Traffic management agents:** These are agents that are designed to manage traffic flow in cities. They can monitor traffic patterns, adjust traffic lights, and reroute vehicles to minimize congestion. Examples of traffic management agents include those used in smart cities around the world.
- A **software agent** has Keystrokes, file contents, received network packages which act as sensors and displays on the screen, files, sent network packets acting as actuators.
- A Human-agent has eyes, ears, and other organs which act as sensors, and hands, legs, mouth, and other body parts acting as actuators.
- A **Robotic agent** has Cameras and infrared range finders which act as sensors and various motors acting as actuators.



Types of Agents

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

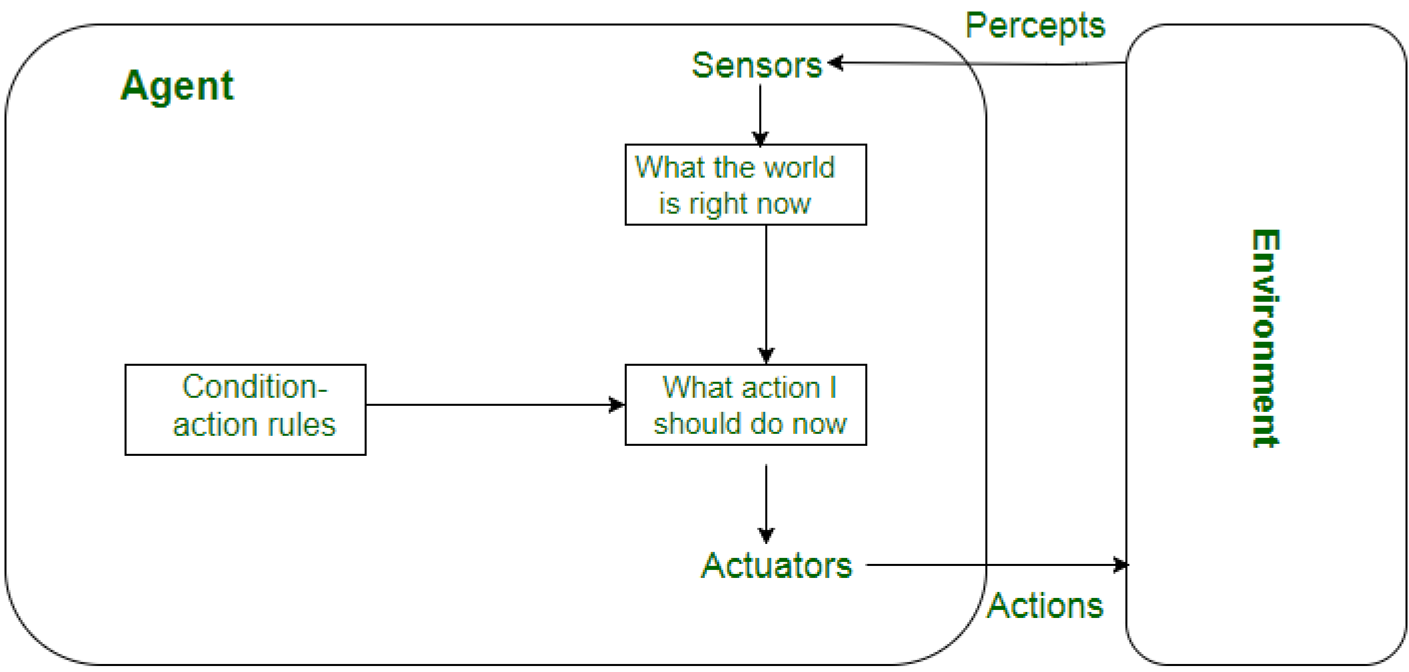
- Simple Reflex Agents
- Model-Based Reflex Agents
- Goal-Based Agents
- Utility-Based Agents
- Learning Agent
- Multi-agent systems
- Hierarchical agents

Simple reflex agents

Simple reflex agents ignore the rest of the percept history and act only on the basis of the **current percept**. Percept history is the history of all that an agent has perceived to date. The agent function is based on the **condition-action rule**. A condition-action rule is a rule that maps a state i.e, condition to an action. If the condition is true, then the action is taken, else not. This agent function only succeeds when the environment is fully observable. For simple reflex agents operating in partially observable environments, infinite loops are often unavoidable. It may be possible to escape from infinite loops if the agent can randomize its actions.

Problems with Simple reflex agents are :

- Very limited intelligence.
- No knowledge of non-perceptual parts of the state.
- Usually too big to generate and store.
- If there occurs any change in the environment, then the collection of rules need to be updated.

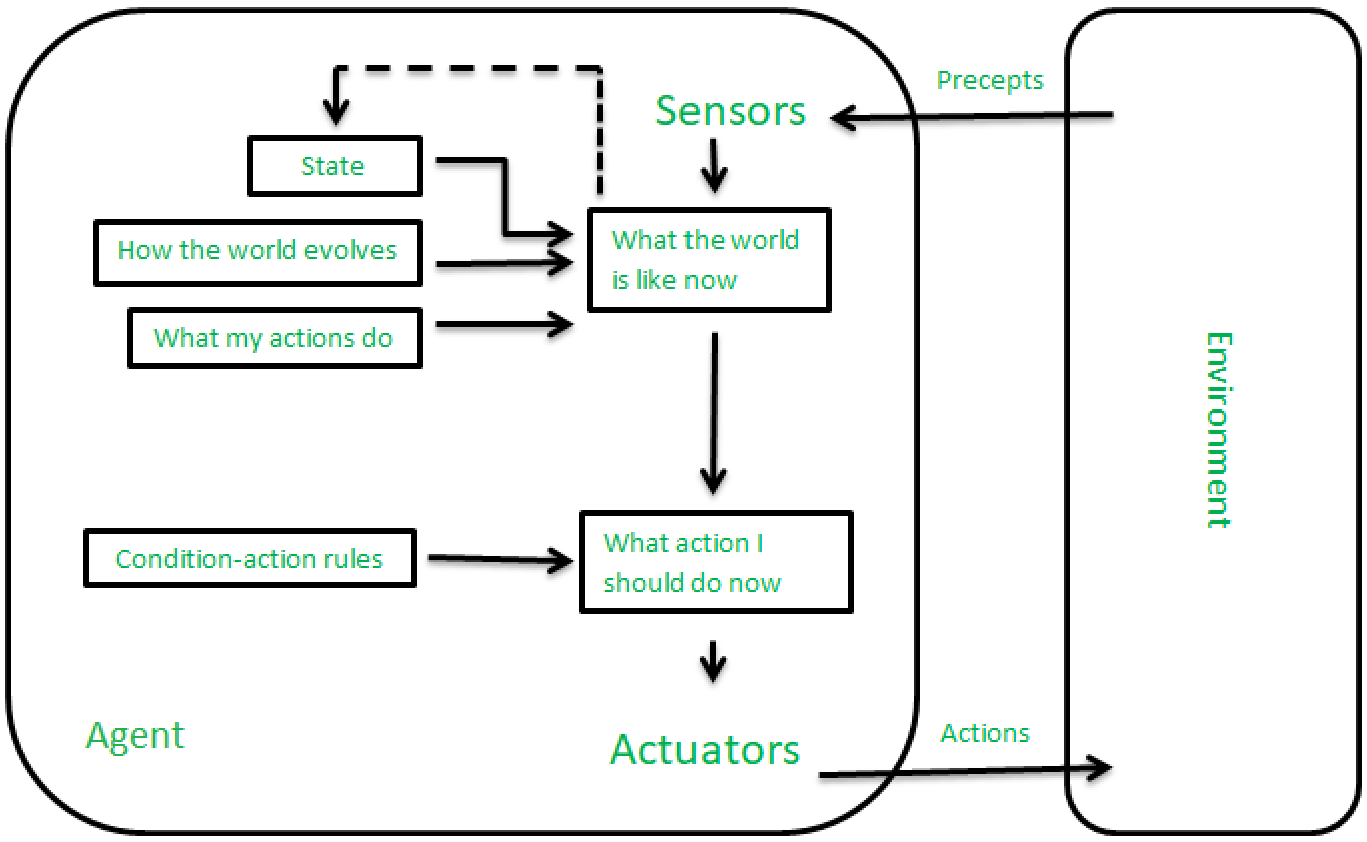


Model-based reflex agents

It works by finding a rule whose condition matches the current situation. A model-based agent can handle **partially observable environments** by the use of a model about the world. The agent has to keep track of the **internal state** which is adjusted by each percept and that depends on the percept history. The current state is stored inside the agent which maintains some kind of structure describing the part of the world which cannot be seen.

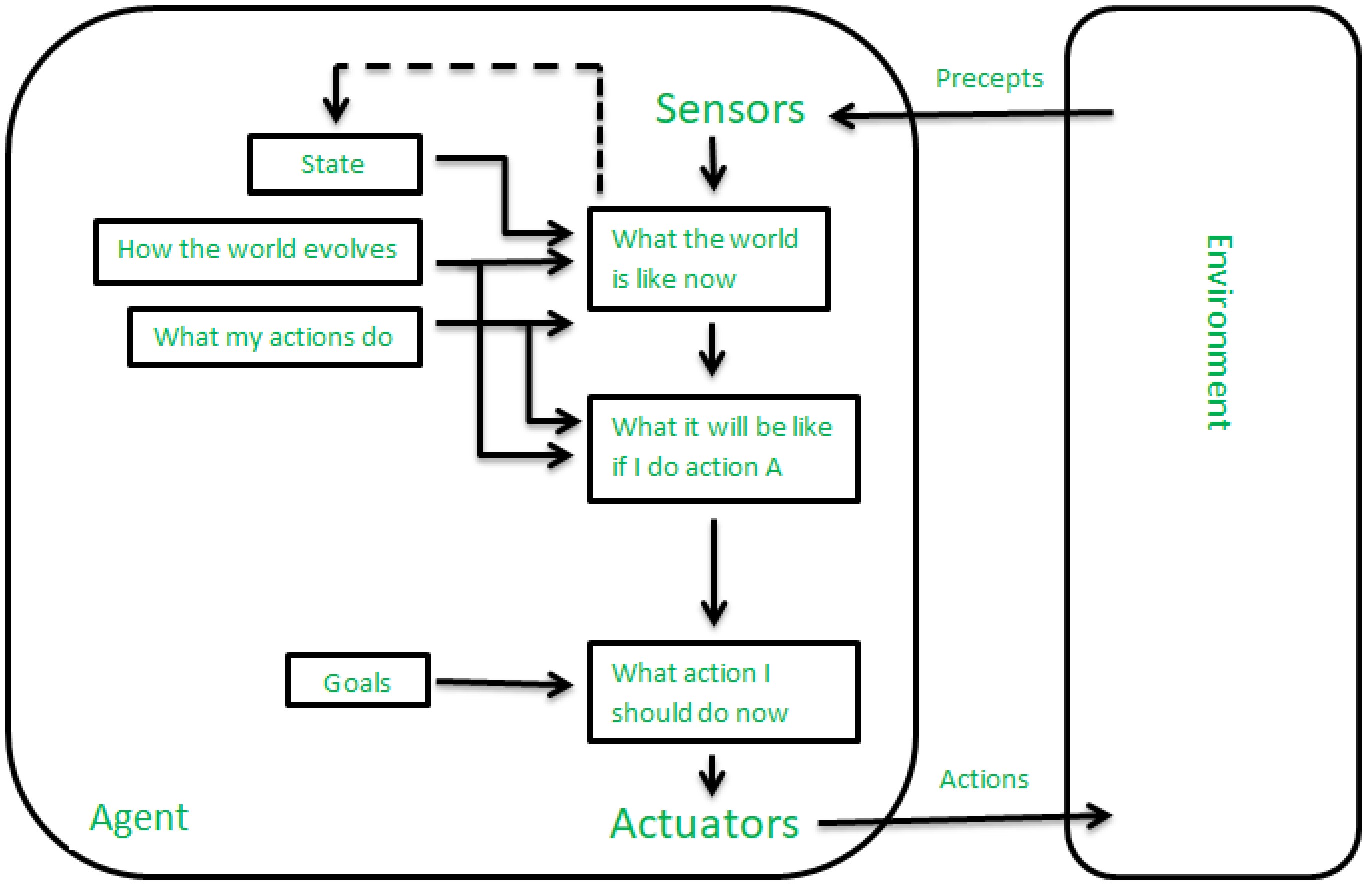
Updating the state requires information about :

- how the world evolves independently from the agent, and
- how the agent's actions affect the world.



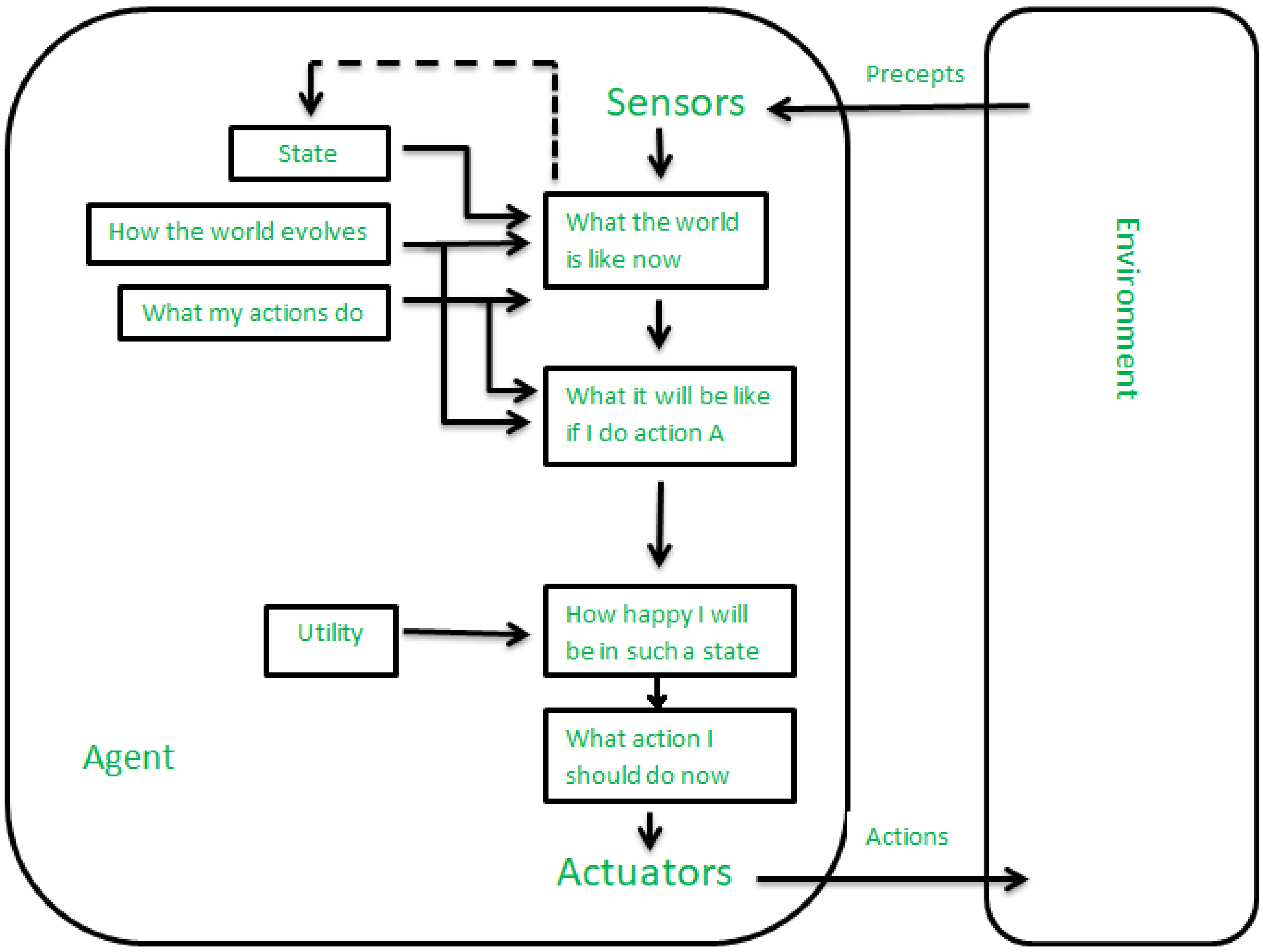
Goal-based agents

These kinds of agents take decisions based on how far they are currently from their **goal**(description of desirable situations). Their every action is intended to reduce its distance from the goal. This allows the agent a way to choose among multiple possibilities, selecting the one which reaches a goal state. The knowledge that supports its decisions is represented explicitly and can be modified, which makes these agents more flexible. They usually require search and planning. The goal-based agent's behavior can easily be changed.



Utility-based agents

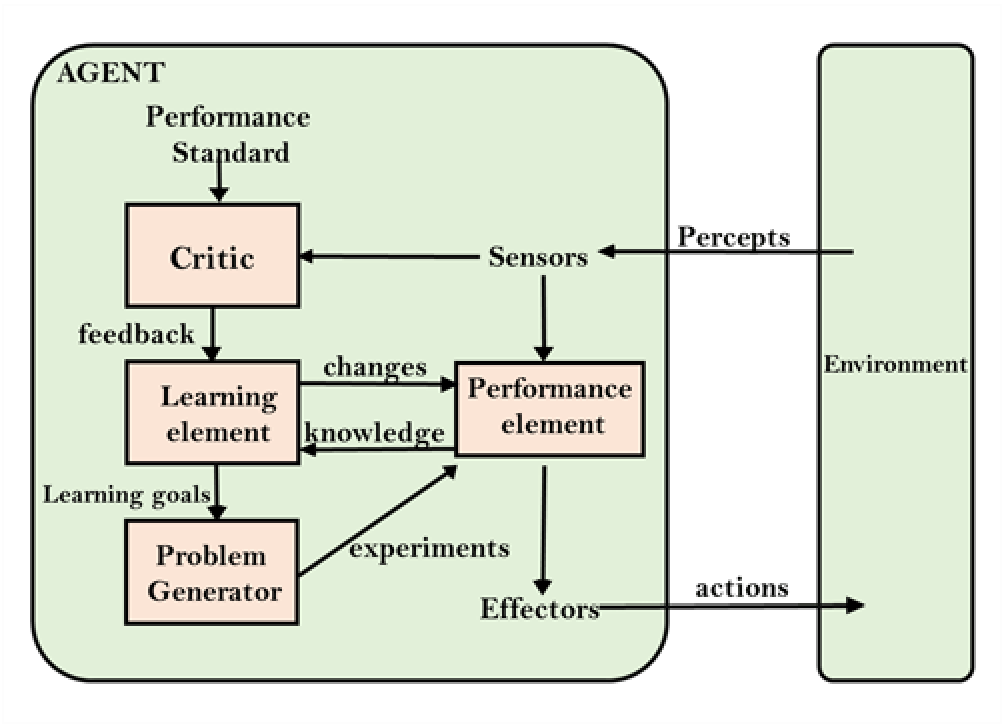
The agents which are developed having their end uses as building blocks are called utility-based agents. When there are multiple possible alternatives, then to decide which one is best, utility-based agents are used. They choose actions based on a **preference (utility)** for each state. Sometimes achieving the desired goal is not enough. We may look for a quicker, safer, cheaper trip to reach a destination. Agent happiness should be taken into consideration. Utility describes how **"happy"** the agent is. Because of the uncertainty in the world, a utility agent chooses the action that maximizes the expected utility. A utility function maps a state onto a real number which describes the associated degree of happiness.


Precepts
Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Utility
How happy I will be in such a state
What action I should do now
Agent
Actuators
Actions
Environment

Learning Agent :

A learning agent in AI is the type of agent that can learn from its past experiences or it has learning capabilities. It starts to act with basic knowledge and then is able to act and adapt automatically through learning.
A learning agent has mainly four conceptual components, which are:

1. **Learning element:** It is responsible for making improvements by learning from the environment
2. **Critic:** The learning element takes feedback from critics which describes how well the agent is doing with respect to a fixed performance standard.
3. **Performance element:** It is responsible for selecting external action
4. **Problem Generator:** This component is responsible for suggesting actions that will lead to new and informative experiences.



Multi-agent systems:

These agents interact with other agents to achieve a common goal. They may have to coordinate their actions and communicate with each other to achieve their objective.

A multi-agent system (MAS) is a system composed of multiple interacting agents that are designed to work together to achieve a common goal. These agents may be autonomous or semi-autonomous and are capable of perceiving their environment, making decisions, and taking actions to achieve the common objective.

MAS can be used in a variety of applications, including transportation systems, robotics, and social networks. They can help improve efficiency, reduce costs, and increase flexibility in complex systems.

MAS can be classified into different types based on their characteristics, such as whether the agents have the same or different goals, whether the agents are cooperative or competitive, and whether the agents are homogeneous or heterogeneous.

In a homogeneous MAS, all the agents have the same capabilities, goals, and behaviors. In contrast, in a heterogeneous MAS, the agents have different capabilities, goals, and behaviors. This can make coordination more challenging but can also lead to more flexible and robust systems.

Cooperative MAS involves agents working together to achieve a common goal, while competitive MAS involves agents working against each other to achieve their own goals. In some cases, MAS can also involve both cooperative and competitive behavior, where agents must balance their own interests with the interests of the group.

MAS can be implemented using different techniques, such as game theory, machine learning, and agent-based modeling. Game theory is used to analyze strategic interactions between agents and predict their behavior. Machine learning is used to train agents to improve their decision-making capabilities over time. Agent-based modeling is used to simulate complex systems and study the interactions between agents.

Overall, multi-agent systems are a powerful tool in artificial intelligence that can help solve complex problems and improve efficiency in a variety of applications.

Hierarchical agents:

These agents are organized into a hierarchy, with high-level agents overseeing the behavior of lower-level agents. The high-level agents provide goals and constraints, while the low-level agents carry out specific tasks. Hierarchical agents are useful in complex environments with many tasks and sub-tasks.

Hierarchical agents are agents that are organized into a hierarchy, with high-level agents overseeing the behavior of lower-level agents. The high-level agents provide goals and constraints, while the low-level agents carry out specific tasks. This structure allows for more efficient and organized decision-making in complex environments.

Hierarchical agents can be implemented in a variety of applications, including robotics, manufacturing, and transportation systems. They are particularly useful in environments where there are many tasks and sub-tasks that need to be coordinated and prioritized.

In a hierarchical agent system, the high-level agents are responsible for setting goals and constraints for the lower-level agents. These goals and constraints are typically based on the overall objective of the system. For example, in a manufacturing system, the high-level agents might set production targets for the lower-level agents based on customer demand.

The low-level agents are responsible for carrying out specific tasks to achieve the goals set by the high-level agents. These tasks may be relatively simple or more complex, depending on the specific application. For example, in a transportation system, the low-level agents might be responsible for managing traffic flow at specific intersections.

Hierarchical agents can be organized into different levels, depending on the complexity of the system. In a simple system, there may be only two levels: high-level agents and low-level agents. In a more complex system, there may be multiple levels, with intermediate-level agents responsible for coordinating the activities of lower-level agents.

One advantage of hierarchical agents is that they allow for a more efficient use of resources. By organizing agents into a hierarchy, it is possible to allocate tasks to the agents that are best suited to carry them out, while avoiding duplication of effort. This can lead to faster, more efficient decision-making and better overall performance of the system.

Overall, hierarchical agents are a powerful tool in artificial intelligence that can help solve complex problems and improve efficiency in a variety of applications.

Uses of Agents :

Agents are used in a wide range of applications in artificial intelligence, including:

Robotics: Agents can be used to control robots and automate tasks in manufacturing, transportation, and other industries.

Smart homes and buildings: Agents can be used to control heating, lighting, and other systems in smart homes and buildings, optimizing energy use and improving comfort.

Transportation systems: Agents can be used to manage traffic flow, optimize routes for autonomous vehicles, and improve logistics and supply chain management.

Healthcare: Agents can be used to monitor patients, provide personalized treatment plans, and optimize healthcare resource allocation.

Finance: Agents can be used for automated trading, fraud detection, and risk management in the financial industry.

Games: Agents can be used to create intelligent opponents in games and simulations, providing a more challenging and realistic experience for players.

Natural language processing: Agents can be used for language translation, question answering, and chatbots that can communicate with users in natural language.

Cybersecurity: Agents can be used for intrusion detection, malware analysis, and network security.

Environmental monitoring: Agents can be used to monitor and manage natural resources, track climate change, and improve environmental sustainability.

Social media: Agents can be used to analyze social media data, identify trends and patterns, and provide personalized recommendations to users.

Overall, agents are a versatile and powerful tool in artificial intelligence that can help solve a wide range of problems in different fields.

Problem-solving agents:

Rational agents or Problem-solving agents in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result. Problem-solving agents are the **goal-based agents and use atomic representation**.

The reflex agent of AI directly maps states into action. Whenever these agents fail to operate in an environment where the state of mapping is too large and not easily performed by the agent, then the stated problem dissolves and sent to a problem-solving domain which breaks the large stored problem into the smaller storage area and resolves one by one. The final integrated action will be the desired outcomes.

On the basis of the problem and their working domain, different types of problem-solving agent defined and use at an atomic level without any internal state visible with a problem-solving algorithm. The problem-solving agent performs precisely by defining problems and several solutions. So we can say that problem solving is a part of artificial intelligence that encompasses a number of techniques such as a tree, B-tree, heuristic algorithms to solve a problem.

We can also say that a problem-solving agent is a result-driven agent and always focuses on satisfying the goals.

There are basically three types of problem in artificial intelligence:

1. Ignorable: In which solution steps can be ignored.

2. Recoverable: In which solution steps can be undone.

3. Irrecoverable: Solution steps cannot be undo.

Steps problem-solving in AI: The problem of AI is directly associated with the nature of humans and their activities. So we need a number of finite steps to solve a problem which makes human easy works.

These are the following steps which require to solve a problem :

- **Problem definition:** Detailed specification of inputs and acceptable system solutions.
- **Problem analysis:** Analyse the problem thoroughly.
- **Knowledge Representation:** collect detailed information about the problem and define all possible techniques.
- **Problem-solving:** Selection of best techniques.

Components to formulate the associated problem:

- **Initial State:** This state requires an initial state for the problem which starts the AI agent towards a specified goal. In this state new methods also initialize problem domain solving by a specific class.
- **Action:** This stage of problem formulation works with function with a specific class taken from the initial state and all possible actions done in this stage.
- **Transition:** This stage of problem formulation integrates the actual action done by the previous action stage and collects the final stage to forward it to their next stage.
- **Goal test:** This stage determines that the specified goal achieved by the integrated transition model or not, whenever the goal achieves stop the action and forward into the next stage to determines the cost to achieve the goal.
- **Path costing:** This component of problem-solving numerical assigned what will be the cost to achieve the goal. It requires all hardware software and human working cost.

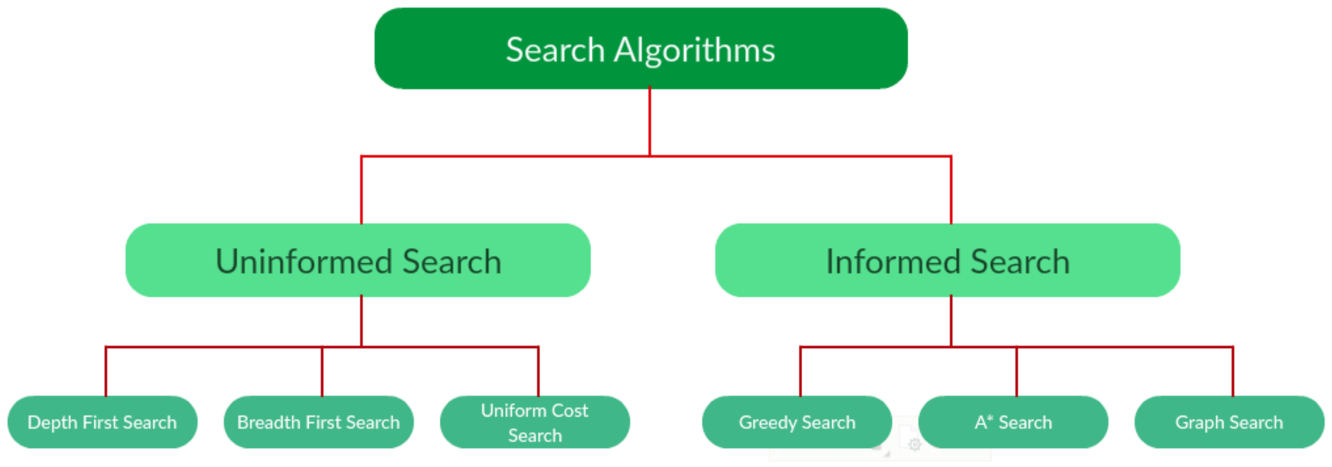
Search Algorithms in AI

Artificial Intelligence is the study of building agents that act rationally. Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.

- A search problem consists of:
    - **A State Space.** Set of all possible states where you can be.
    - **A Start State.** The state from where the search begins.
    - **A Goal State.** A function that looks at the current state returns whether or not it is the goal state.
- The **Solution** to a search problem is a sequence of actions, called the **plan** that transforms the start state to the goal state.
- This plan is achieved through search algorithms.

Types of search algorithms:

There are far too many powerful search algorithms out there to fit in a single article. Instead, this article will discuss *six* of the fundamental search algorithms, divided into *two* categories, as shown below.



Note that there is much more to search algorithms than the chart I have provided above. However, this article will mostly stick to the above chart, exploring the algorithms given there.

Uninformed Search Algorithms:

The search algorithms in this section have no additional information on the goal node other than the one provided in the problem definition. The plans to reach the goal state from the start state differ only by the order and/or length of actions. Uninformed search is also called **Blind search**. These algorithms can only generate the successors and differentiate between the goal state and non goal state.

The following uninformed search algorithms are discussed in this section.

1. Depth First Search
2. Breadth First Search
3. Uniform Cost Search

Each of these algorithms will have:

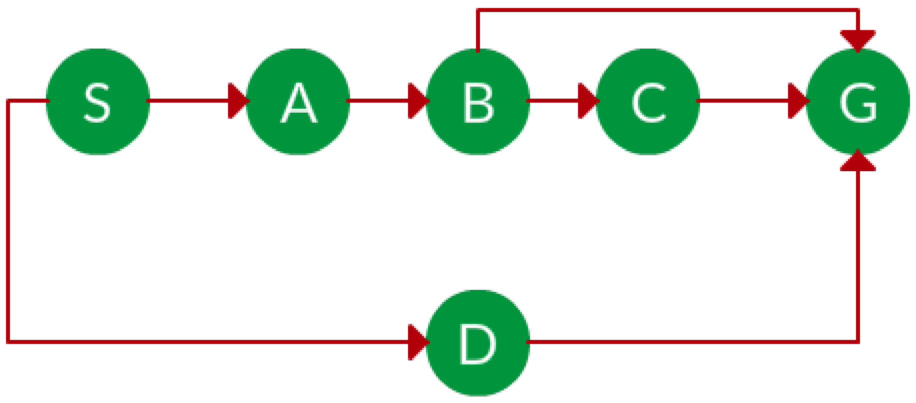
- A problem **graph,** containing the start node S and the goal node G.
- A **strategy,** describing the manner in which the graph will be traversed to get to G.
- A **fringe,** which is a data structure used to store all the possible states (nodes) that you can go from the current states.
- A **tree,** that results while traversing to the goal node.
- A solution **plan,** which the sequence of nodes from S to G.

Depth First Search:

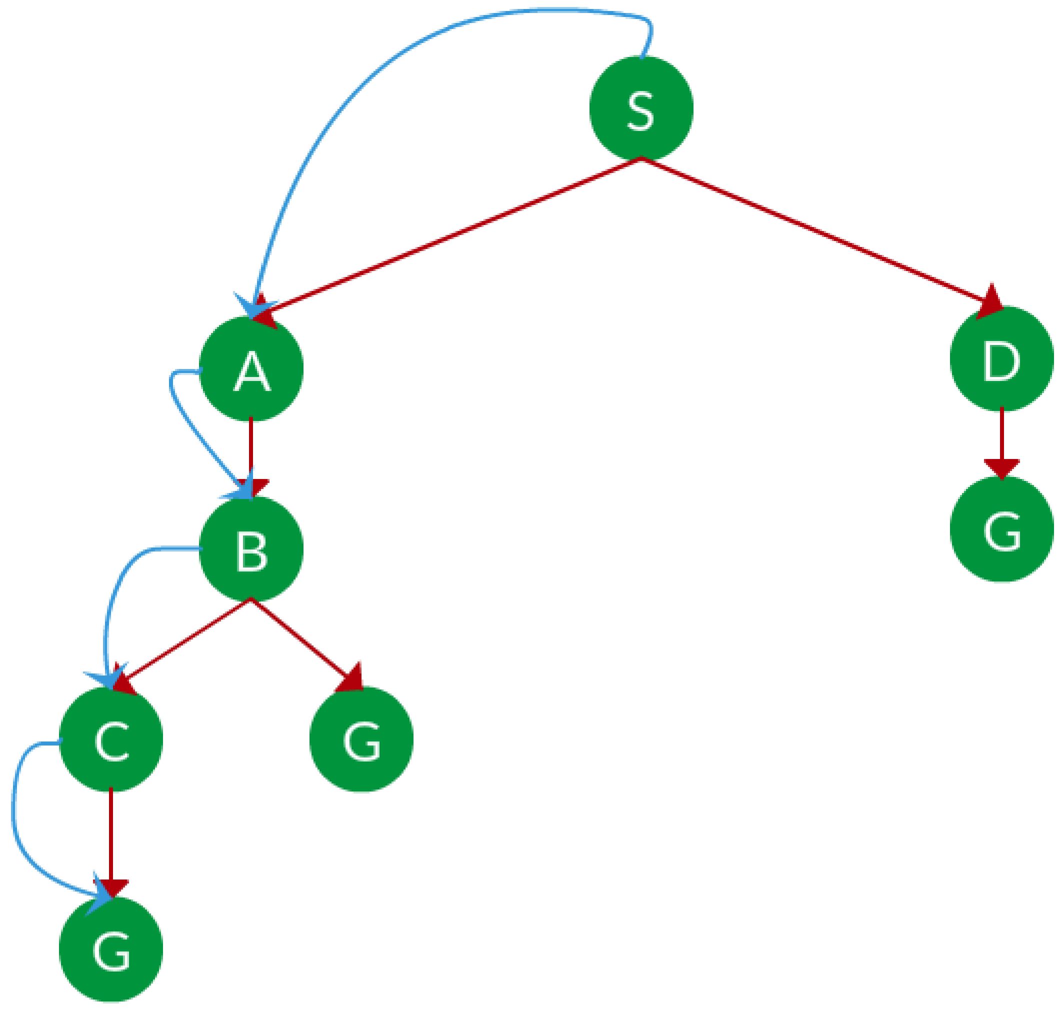
Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. It uses last in-first-out strategy and hence it is implemented using a stack.

Example:

Question. Which solution would DFS find to move from node S to node G if run on the graph below?



Solution. The equivalent search tree for the above graph is as follows. As DFS traverses the tree "deepest node first", it would always pick the deeper branch until it reaches the solution (or it runs out of nodes, and goes to the next branch). The traversal is shown in blue arrows.



Path: S -> A -> B -> C -> G

= the depth of the search tree = the number of levels of the search tree.

= number of nodes in level .

Time complexity: Equivalent to the number of nodes traversed in

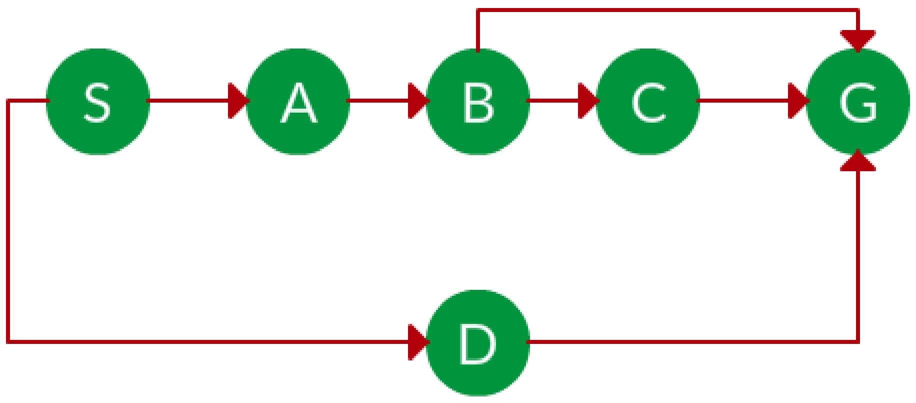
DFS.

Space complexity: Equivalent to how large can the fringe get.
Completeness: DFS is complete if the search tree is finite, meaning for a given finite search tree, DFS will come up with a solution if it exists.
Optimality: DFS is not optimal, meaning the number of steps in reaching the solution, or the cost spent in reaching it is high.

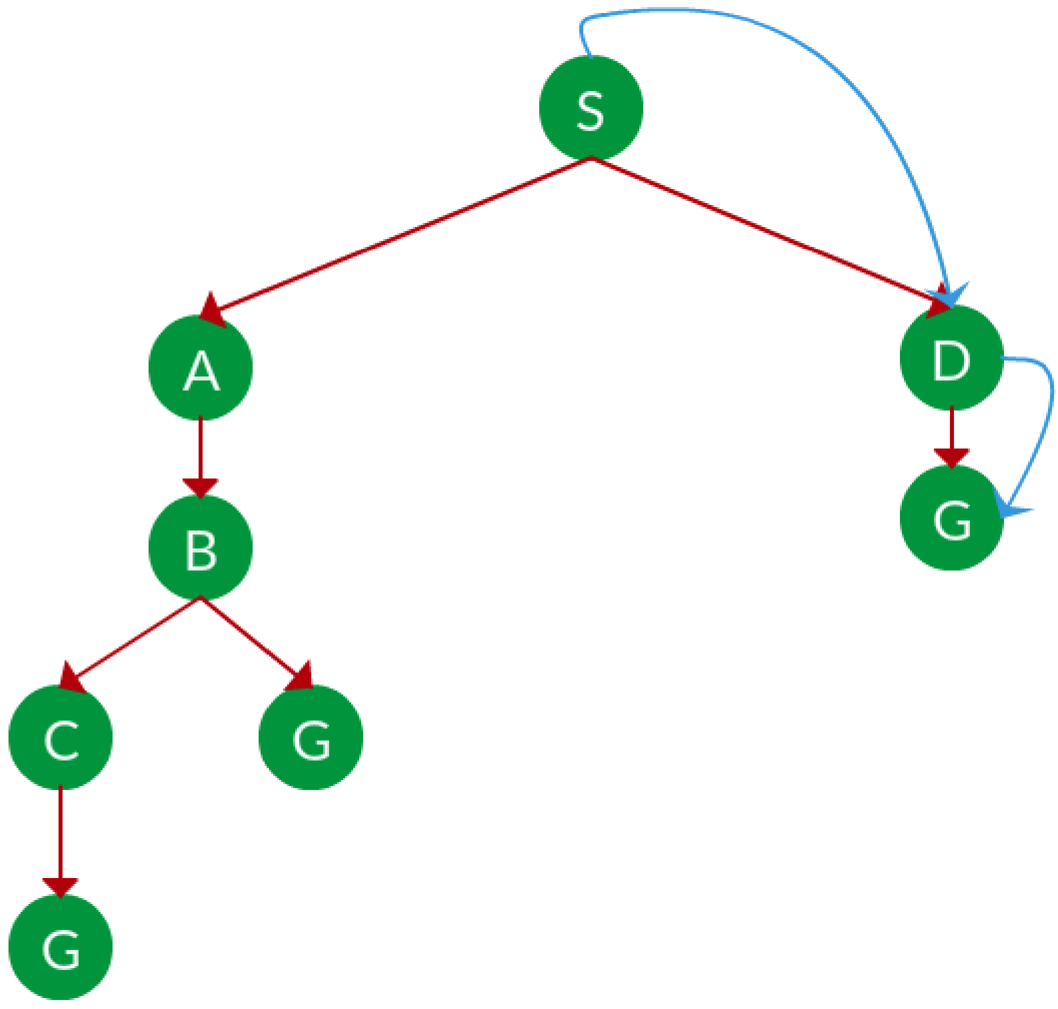
Breadth First Search:

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. It is implemented using a queue.

Example:
Question. Which solution would BFS find to move from node S to node G if run on the graph below?



Solution. The equivalent search tree for the above graph is as follows. As BFS traverses the tree "shallowest node first", it would always pick the shallower branch until it reaches the solution (or it runs out of nodes, and goes to the next branch). The traversal is shown in blue arrows.



Path: S -> D -> G

= the depth of the shallowest solution.

= number of nodes in level .

Time complexity: Equivalent to the number of nodes traversed in BFS until the shallowest solution.

Space complexity: Equivalent to how large can the fringe get.

Completeness: BFS is complete, meaning for a given search tree, BFS will come up with a solution if it exists.

Optimality: BFS is optimal as long as the costs of all edges are equal.

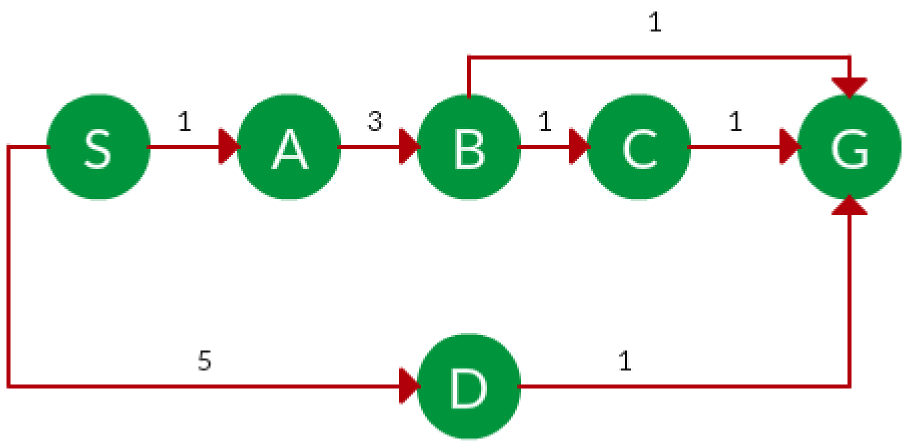
Uniform Cost Search:

UCS is different from BFS and DFS because here the costs come into play. In other words, traversing via different edges might not have the same cost. The goal is to find a path where the cumulative sum of costs is the least.

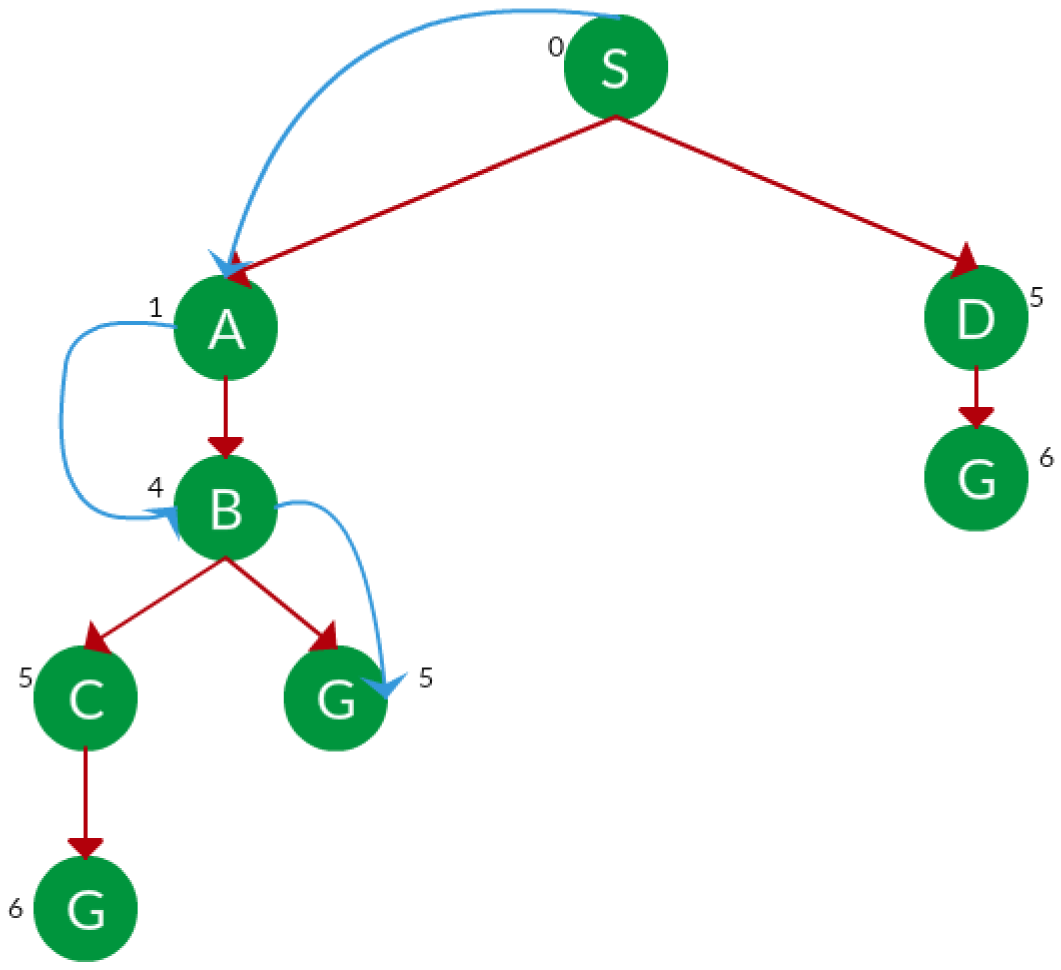
Cost of a node is defined as:

```
cost(node) = cumulative cost of all nodes from root
cost(root) = 0
```

Example:
Question. Which solution would UCS find to move from node S to node G if run on the graph below?



Solution. The equivalent search tree for the above graph is as follows. The cost of each node is the cumulative cost of reaching that node from the root. Based on the UCS strategy, the path with the least cumulative cost is chosen. Note that due to the many options in the fringe, the algorithm explores most of them so long as their cost is low, and discards them when a lower-cost path is found; these discarded traversals are not shown below. The actual traversal is shown in blue.



Path: S -> A -> B -> G
Cost: 5

Let = cost of solution.
= arcs cost.

Then effective depth

Time complexity: , **Space complexity:**

Advantages:

- UCS is complete only if states are finite and there should be no loop with zero weight.
- UCS is optimal only if there is no negative cost.

Disadvantages:

- Explores options in every "direction".
- No information on goal location.

Informed Search Algorithms:

Here, the algorithms have information on the goal state, which helps in more efficient searching. This information is obtained by something called a *heuristic*.
In this section, we will discuss the following search algorithms.

1. Greedy Search
2. A* Tree Search
3. A* Graph Search

Search Heuristics: In an informed search, a heuristic is a *function* that estimates how close a state is to the goal state. For example – Manhattan distance, Euclidean distance, etc. (Lesser the distance, closer the goal.) Different heuristics are used in different informed algorithms discussed below.

Greedy Search:

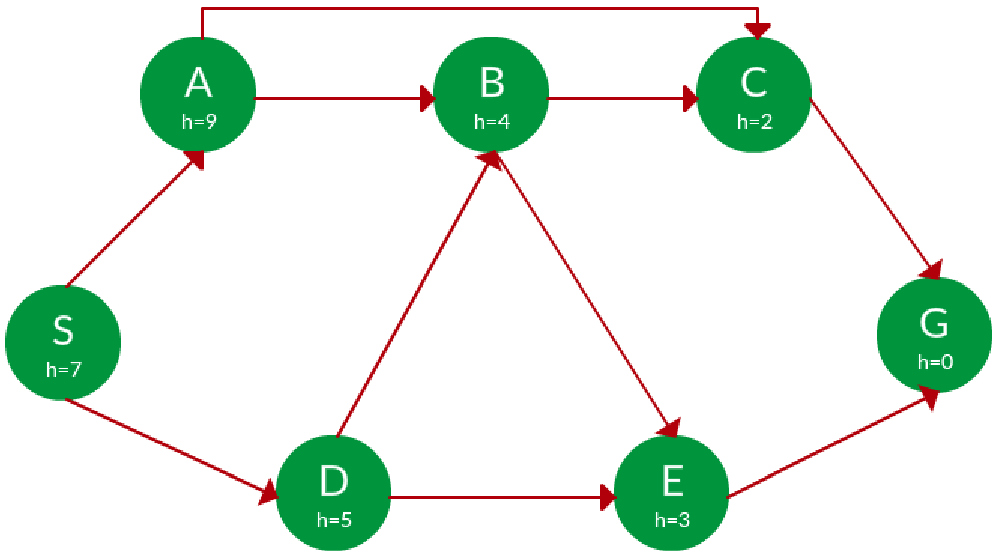
In greedy search, we expand the node closest to the goal node. The "closeness" is estimated by a heuristic h(x).

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

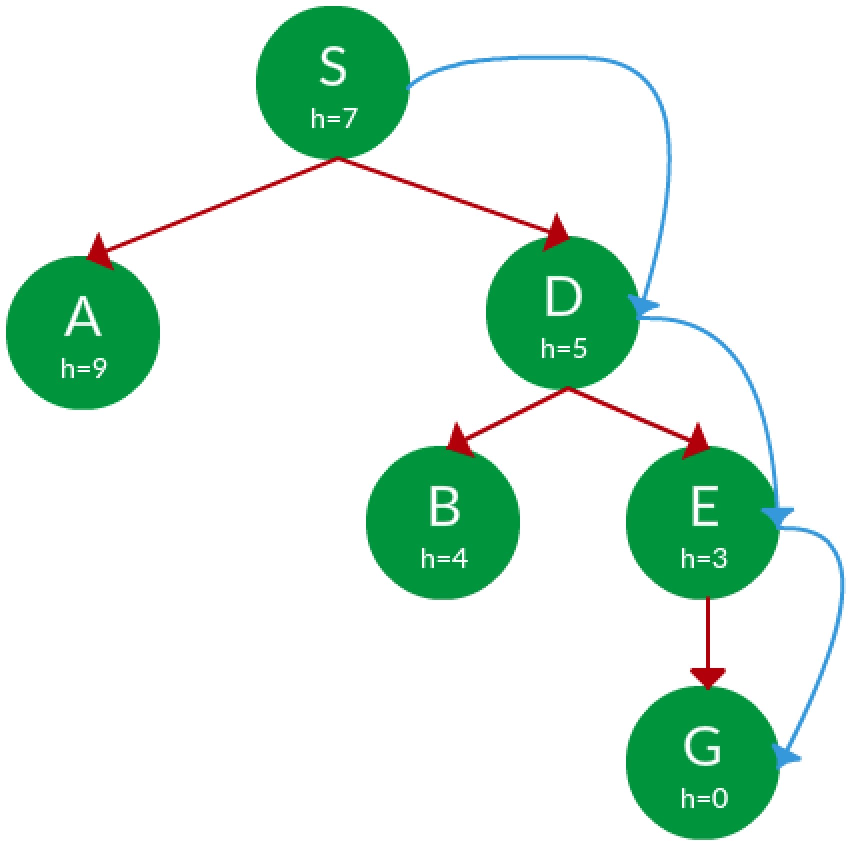
Strategy: Expand the node closest to the goal state, *i.e.* expand the node with a lower h value.

Example:

Question. Find the path from S to G using greedy search. The heuristic values h of each node below the name of the node.



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



Path: S -> D -> E -> G

Advantage: Works well with informed search problems, with fewer steps to reach a goal.
Disadvantage: Can turn into unguided DFS in the worst case.

A* Tree Search:

A* Tree Search, or simply known as A* Search, combines the strengths of uniform-cost search and greedy search. In this search, the heuristic is the summation of the cost in UCS, denoted by g(x), and the cost in the greedy search, denoted by h(x). The summed cost is denoted by f(x).

Heuristic: The following points should be noted wrt heuristics in A*

search.

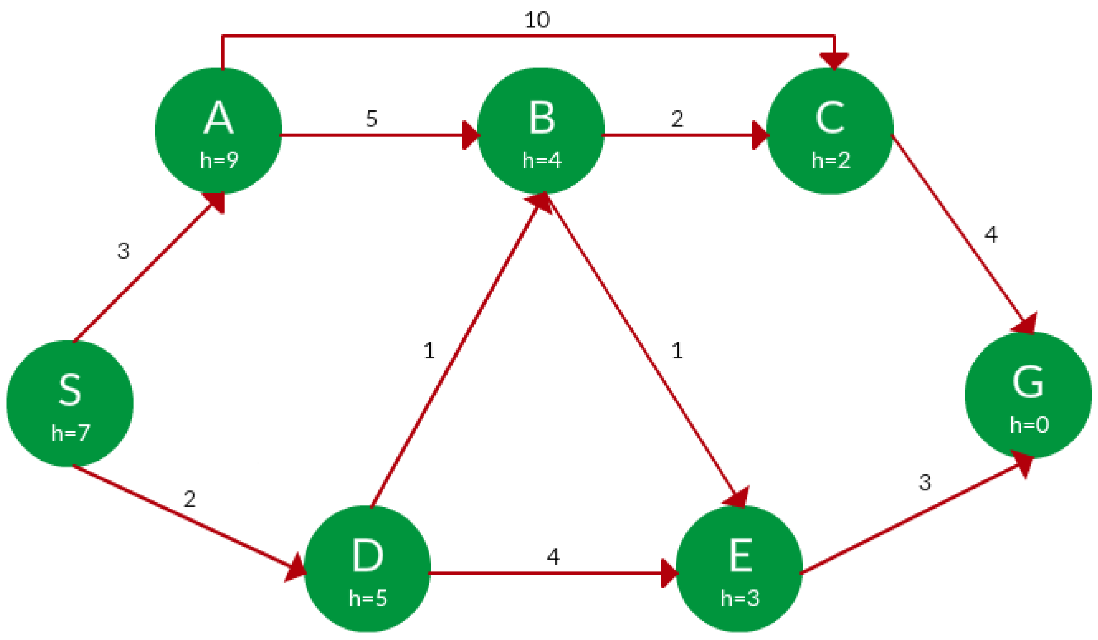
- Here, h(x) is called the **forward cost** and is an estimate of the distance of the current node from the goal node.
- And, g(x) is called the **backward cost** and is the cumulative cost of a node from the root node.
- A* search is optimal only when for all nodes, the forward cost for a node h(x) underestimates the actual cost h*(x) to reach the goal. This property of *A** heuristic is called **admissibility**.

Admissibility:

Strategy: Choose the node with the lowest f(x) value.

Example:

Question. Find the path to reach from S to G using A* search.



Solution. Starting from S, the algorithm computes g(x) + h(x) for all nodes in the fringe at each step, choosing the node with the lowest sum. The entire work is shown in the table below.

Note that in the fourth set of iterations, we get two paths with equal summed cost f(x), so we expand them both in the next set. The path with a lower cost on further expansion is the chosen path.

Path	h(x)	g(x)	f(x)
S	7	0	7
S -> A	9	3	12
S -> D	5	2	7
S -> D -> B	4	2 + 1 = 3	7
S -> D -> E	3	2 + 4 = 6	9
S -> D -> B -> C	2	3 + 2 = 5	7
S -> D -> B -> E	3	3 + 1 = 4	7
S -> D -> B -> C -> G	0	5 + 4 = 9	9
S -> D -> B -> E -> G	0	4 + 3 = 7	7

Path: S -> D -> B -> E -> G
Cost: 7

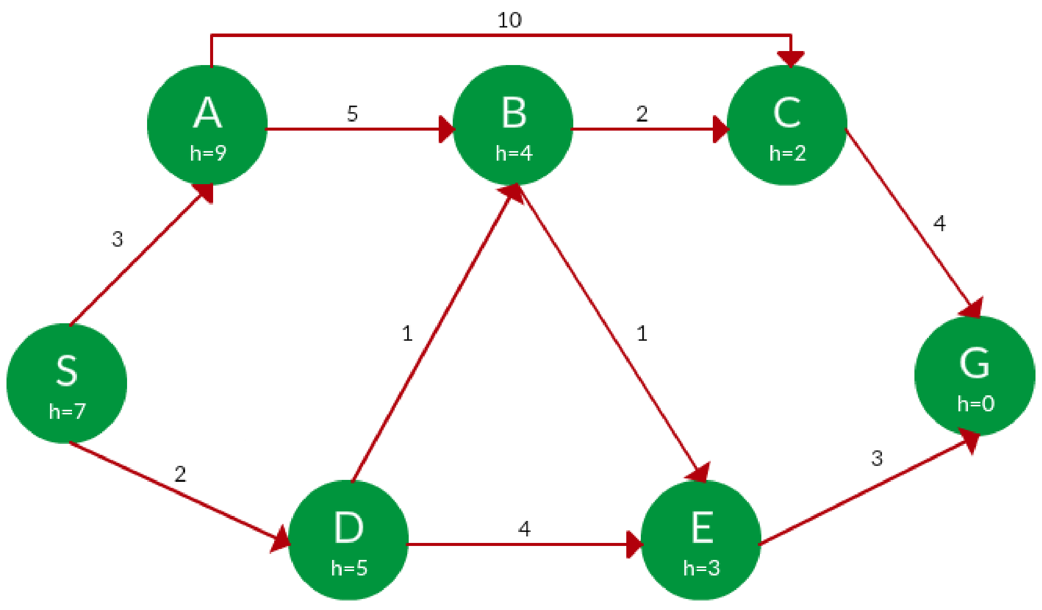
A* Graph Search:

- A* tree search works well, except that it takes time re-exploring the branches it has already explored. In other words, if the same node has expanded twice in different branches of the search tree, A* search might explore both of those branches, thus wasting time
- A* Graph Search, or simply Graph Search, removes this limitation by adding this rule: **do not expand the same node more than once.**
- **Heuristic.** Graph search is optimal only when the forward cost between two successive nodes A and B, given by h(A) – h (B), is less than or equal to the backward cost between those two nodes g(A -> B). This property of the graph search heuristic is called **consistency**.

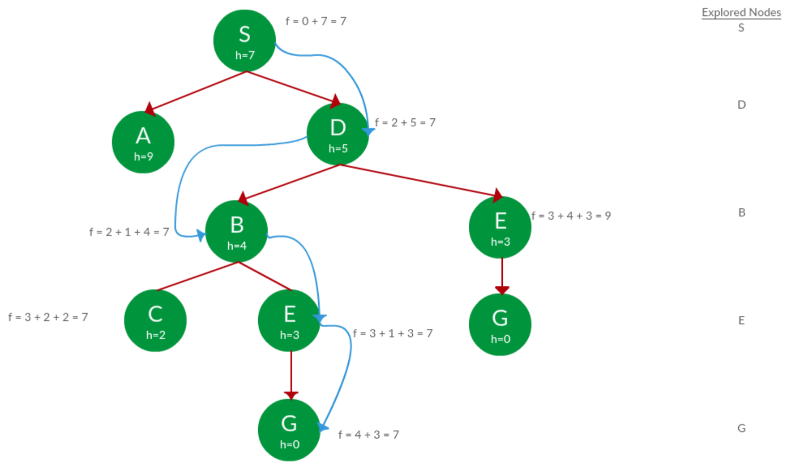
Consistency:

Example:

Question. Use graph searches to find paths from S to G in the following graph.



the **Solution.** We solve this question pretty much the same way we solved last question, but in this case, we keep a track of nodes explored so that we don’t re-explore them.



Path: S -> D -> B -> E -> G
Cost: 7