UNIT I

- Problem Solving by Search-I: Introduction to AI, Intelligent Agents
- Problem Solving by Search –II: Problem-Solving Agents, Searching for Solutions
- Uninformed Search Strategies: Breadth-first search, Uniform cost search, Depth-first search, Iterative deepening Depth-first search, Bidirectional search
- Informed (Heuristic) Search Strategies: Greedy best-first search, A* search, Heuristic Functions
- Beyond Classical Search: Hill-climbing search, Simulated annealing search
- Local Search in Continuous Spaces
- Searching with Non-Deterministic Actions,
- Searching with Partial Observations
- Online Search Agents and Unknown Environment .

UNIT I

Artificial Intelligence:

- **“Artificial Intelligence is the ability of a computer to act like a human being”.**
- Artificial intelligence systems consist of people, procedures, hardware, software, data, and knowledge needed to develop computer systems and machines that demonstrate the characteristics of intelligence

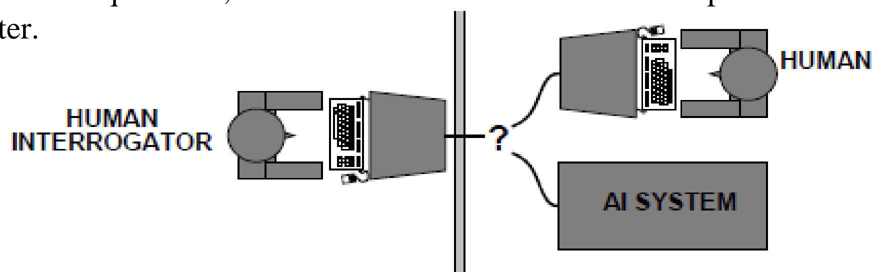
Programming Without AI	Programming With AI
A computer program without AI can answer the **specific** questions it is meant to solve.	A computer program with AI can answer the **generic** questions it is meant to solve.
Modification in the program leads to change in its structure.	AI programs can absorb new modifications by putting highly independent pieces of information together. Hence you can modify even a minute piece of information of program without affecting its structure.
Modification is not quick and easy. It may lead to affecting the program adversely.	Quick and Easy program modification.

Four Approaches of Artificial Intelligence:

- Acting humanly: The Turing test approach.
- Thinking humanly: The cognitive modelling approach.
- Thinking rationally: The laws of thought approach.
- Acting rationally: The rational agent approach.

<u>Acting humanly: The Turing Test approach</u>

The **Turing Test**, proposed by Alan Turing (1950), was designed to provide a satisfactory operational definition of intelligence. A computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or from a computer.



-
- **natural language processing** to enable it to communicate successfully in English;
- **knowledge representation** to store what it knows or hears;
- **automated reasoning** to use the stored information to answer questions and to draw new conclusions
- **machine learning** to adapt to new circumstances and to detect and extrapolate patterns.

Thinking humanly: The cognitive modelling approach

Analyse how a given program thinks like a human, we must have some way of determining how humans think. The interdisciplinary field of **cognitive science** brings together computer models from AI and experimental techniques from psychology to try to construct precise and testable theories of the workings of the human mind. Although cognitive science is a fascinating field in itself, we are not going to be discussing it all that much in this book. We will occasionally comment on similarities or differences between AI techniques and human cognition. Real cognitive science, however, is necessarily based on experimental investigation of actual humans or animals, and we assume that the reader only has access to a computer for experimentation. We will simply note that AI and cognitive science continue to fertilize each other, especially in the areas of vision, natural language, and learning.

Thinking rationally: The "laws of thought" approach

The Greek philosopher Aristotle was one of the first to attempt to codify **``right thinking,"** that is, irrefutable reasoning processes. His famous **syllogisms** provided patterns for argument structures that always gave correct conclusions given correct premises.

For example, ``Socrates is a man; all men are mortal; therefore Socrates is mortal."

These laws of thought were supposed to govern the operation of the mind, and initiated the field of **logic**.

Acting rationally: The rational agent approach

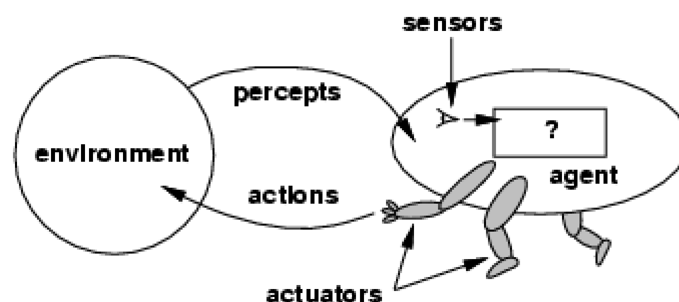
Acting rationally means acting so as to achieve one's goals, given one's beliefs. An agent is just something that perceives and acts.

The right thing: that which is expected to maximize goal achievement, given the available information Does not necessary involve thinking.

For Example - blinking reflex- but should be in the service of rational action.

Agents and environments

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.



- Human Sensors:
  Eyes, ears, and other organs for sensors.
- Human Actuators:
  Hands, legs, mouth, and other body parts.
- Robotic Sensors:
  Mic, cameras and infrared range finders for sensors
- Robotic Actuators:
  Motors, Display, speakers etc

The Al Terminology used in Agents

1) Percept

The term percept refers to the agent's perceptual inputs at any given instant. Examples -

1) A human agent percepts "Bird flying in the sky "through eyes and takes i (photograph)".

2) A robotic agent perceive "Temperature of a boiler" through cameras and takes the control action.

2) Percept Sequence

An agent's percept sequence is the complete history of everything the agent has ever perceived. Agent has choice of action at any given instant and it can depend on the entire percept sequence agent has recorded.

3) Agent Function

It is defined as mathematical function which maps each and every possible percept sequence to a possible action. This function has input as percept sequence and it gives output as action.Agent function can be represented in a tabular form.

Example -

ATM machine is agent, it display menu for withdrawing money, when ATM card is inserted. When provided with percept sequence (1) A transaction type and (2) PIN number, then only user gets cash.

4) Agent Program

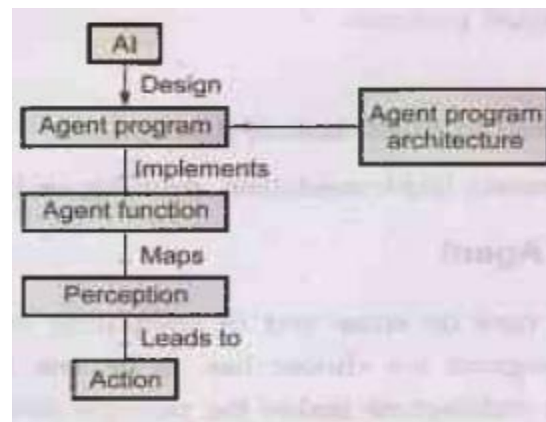
When we want to develop agent program we need to tabulate all the agent functions that describes any given agent.

1.3.1 Architecture of Agent

• The agent program runs on some sort of computing device, which is called the architecture. The program we choose has to be one that the architecture will accept and run. The architecture makes the percepts from the sensors available to the program, runs the program and feeds the program's action choices to the effectors as they are generated. The relationship among agents, architectures and programs can be summed up as follows:

$$Agent = Architecture + Program$$

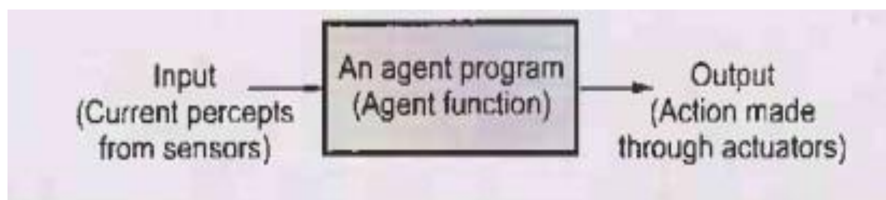
• Following diagram illustrates the agent's action process, as specified by architecture. This can be also termed as agent's structure.



Role of an Agent Program

• An agent program is internally implemented as agent function.

• An agent program takes input as the current percept from the sensor and return an action to theeffectors (Actuators).



Properties of Environment

The environment has multifold properties −

Discrete / Continuous − If there are a limited number of distinct, clearly defined, states of the environment, the environment is discrete For example, chess; otherwise it is continuous For example, driving.

Fully Observable / Partially Observable − If it is possible to determine the complete state of the environment at each time point from the precepts it is observable; otherwise it is only partially observable.

Static / Dynamic − If the environment does not change while an agent is acting, then it is static; otherwise it is dynamic.

Single agent / Multiple agents − The environment may contain other agents which may be of the same or different kind as that of the agent.

Accessible / Inaccessible − If the agent's sensory apparatus can have access to the complete state of the environment, then the environment is accessible to that agent.

Deterministic / Non-deterministic − If the next state of the environment is completely determined by the current state and the actions of the agent, then the environment is deterministic; otherwise it is non-deterministic.

Episodic / Non-episodic − In an episodic environment, each episode consists of the agent perceiving and then acting. The quality of its action depends just on the episode itself. Subsequent episodes do not depend on the actions in the previous episodes. Episodic environments are much simpler because the agent does not need to think ahead.

- A rational agent should be **autonomous**-it should learn from its own prior knowledge (experience).

Task environments, which are essentially the "problems" to which rational agents are the "solutions."

PEAS: Performance Measure, Environment, Actuators, Sensors Consider, e.g., the task of designing an automated taxi driver:

Agent Type	Performance measure	Environment	Actuators	Sensors
Taxi Driver	Safe, fast, legal , comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering wheel, accelerator, brake, signal, horn	Cameras, speedometer, GPS, engine sensors, keyboard

Types of AI Agents

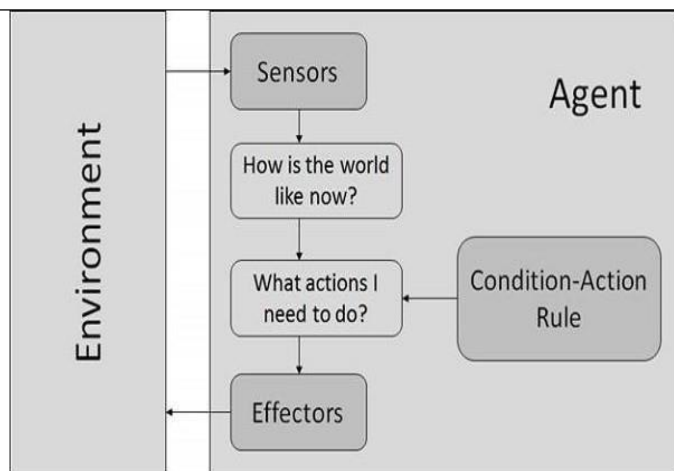
Agents can be grouped into five classes based on their degree of perceived intelligence and capability. All these agents can improve their performance and generate better action over the time.

1)The Simple reflex agents

- The Simple reflex agents are the simplest agents. These agents take decisions on the basis of the current percepts and ignore the rest of the percept history(**past State).**
- These agents only succeed in the fully observable environment.
- The Simple reflex agent does not consider any part of percepts history during their decision and action process.
- The Simple reflex agent works on Condition-action rule, which means it maps the current state to action. Such as a Room Cleaner agent, it works only if there is dirt in the room.
- Problems for the simple reflex agent design approach:
    - They have very limited intelligence
    - They do not have knowledge of non-perceptual parts of the current state
    - Mostly too big to generate and to store.
    - Not adaptive to changes in the environment.

Condition-Action Rule − It is a rule that maps a state (condition) to an action.

Ex: if car-in-front-is-braking then initiate-braking.



function SIMPLE-REFLEX-AGENT(percept)

returns an action

persistent: rules, a set of condition–action rules

state ← INTERPRET-INPUT(percept)

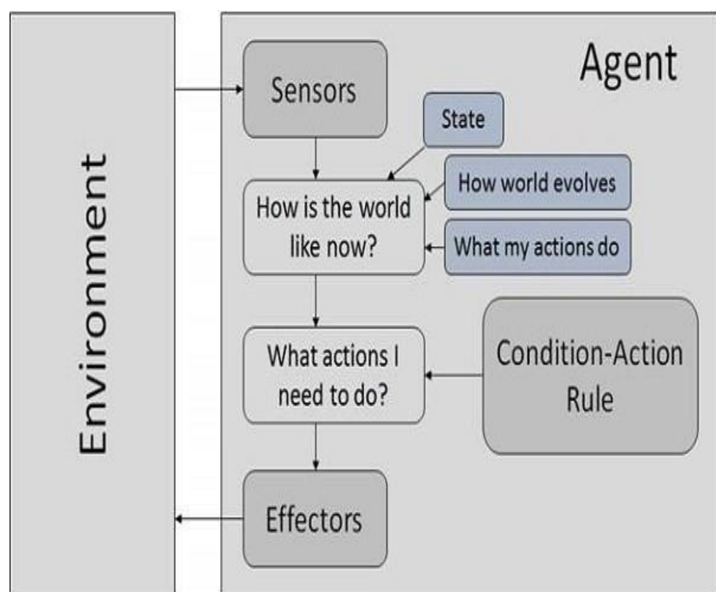
rule ← RULE-MATCH(state, rules)

action ← rule.ACTION

return action

2) 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.
- These agents have the model, "which is knowledge of the world" and based on the model they perform actions.
- Updating the agent state requires information about:
  - How the world evolves
  - How the agent's action affects the world.

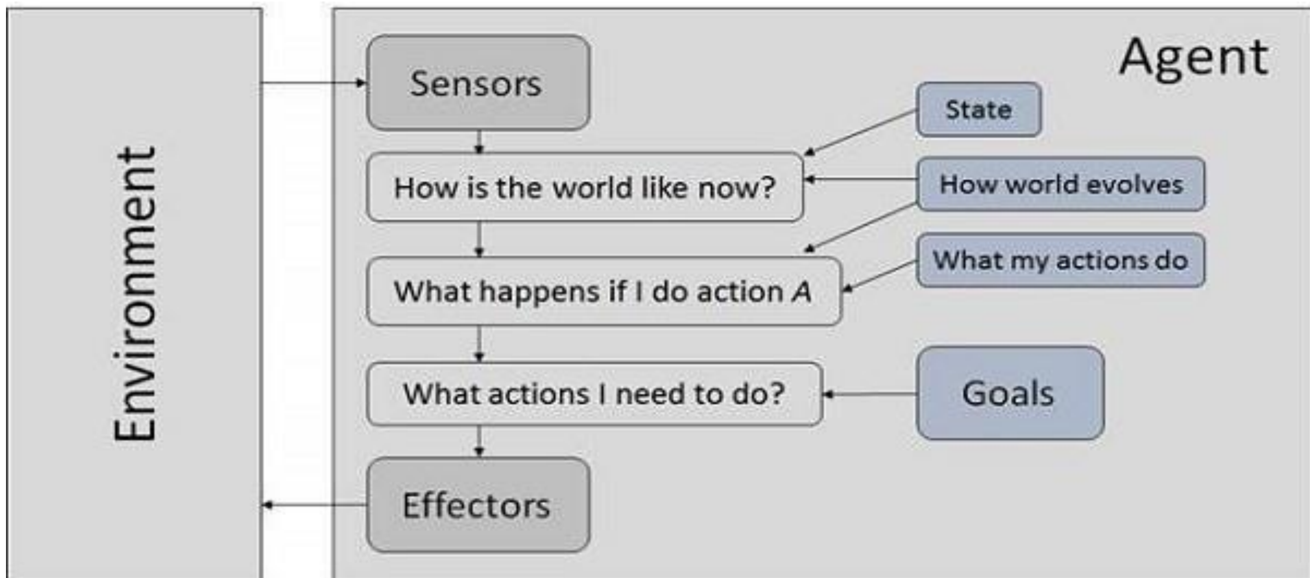


function MODEL-BASED-REFLEX-AGENT (percept) **returns** an action
persistent: state, the agent's current conception of the world state
model , a description of how the next state depends on current state and action *rules*, a set of condition–action rules *action*, the most recent action, initially
none
state ← UPDATE-STATE (state, action, percept, model)
rule←RULE-MATCH (state, rules)
action ←rule.ACTION
return action

3) Goal Based Agents:

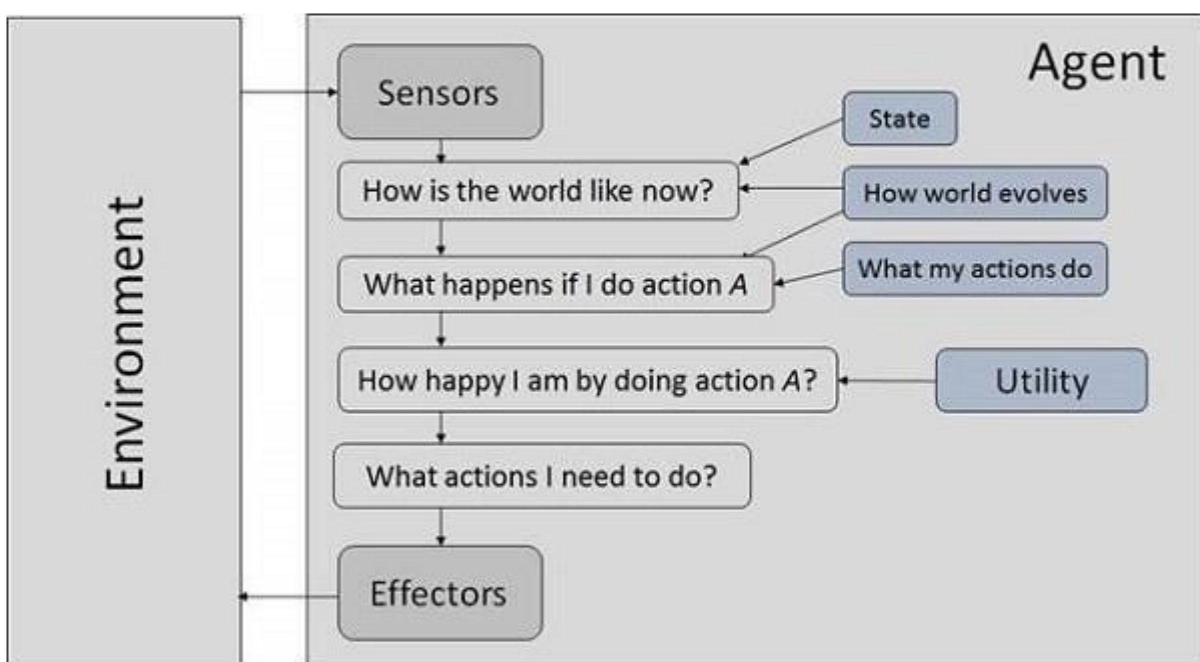
- The knowledge of the current state environment is not always sufficient to decide for an agent to what to do.
- The agent needs to know its goal which describes desirable situations.
- Goal-based agents expand the capabilities of the model-based agent by having the "goal" information.

- They choose an action, so that they can achieve the goal.
- These agents may have to consider a long sequence of possible actions before deciding whether the goal is achieved or not. Such considerations of different scenario are called searching and planning, which makes an agent proactive.



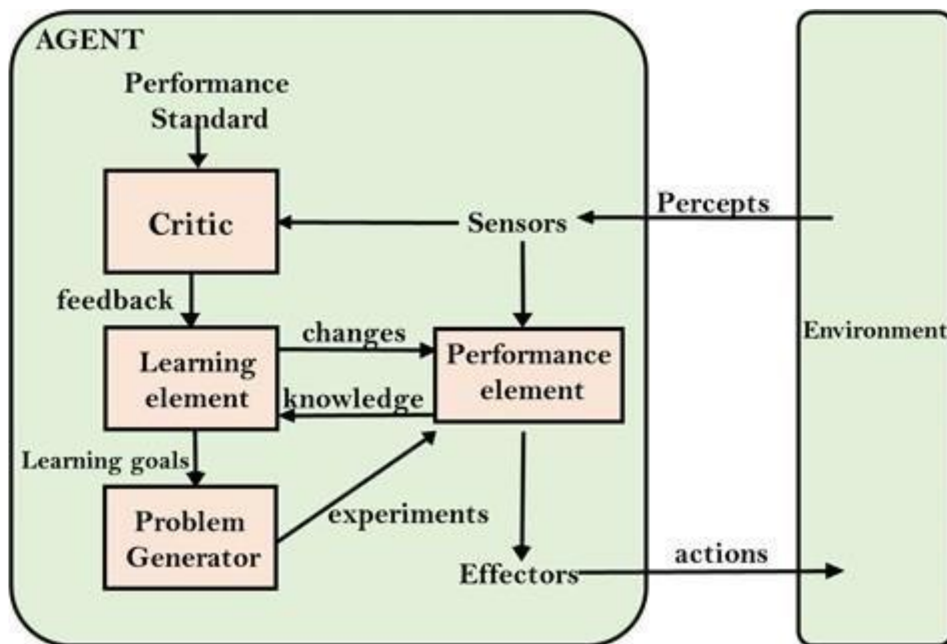
4) Utility Based Agents

- These agents are similar to the goal-based agent but provide an extra component of utility measurement**("Level of Happiness")** which makes them different by providing a measure of success at a given state.
- Utility-based agent act based not only goals but also the best way to achieve the 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 utility function maps each state to a real number to check how efficiently each action achieves the goals.



5. Learning Agents

- A learning agent in AI is the type of agent which can learn from its past experiences, or it has learning capabilities.
- It starts to act with basic knowledge and then able to act and adapt automatically through learning.
- A learning agent has mainly four conceptual components, which are:
    a. **Learning element:** It is responsible for making improvements by learning from environment
    b. **Critic:** Learning element takes feedback from critic which describes that how well the agent is doing with respect to a fixed performance standard.
    c. **Performance element:** It is responsible for selecting external action
    d. **Problem generator:** This component is responsible for suggesting actions that will lead to new and informative experiences.
- Hence, learning agents are able to learn, analyze performance, and look for new ways to improve the performance.



Problem Solving Agents

This adopts a goal and aims to satisfy it. Example : Driving from one major town to another. Steps in

Problem Solving are :

- **Goal Formulation -** based on the current situation and the agent's performance measure is the first step in problem solving.
- Decide on factors that affect desirability to achieve goal
- Decide the various sequences of actions and states to consider. Choose best one.
- Find out which actions will lead to Goal state.

The process of looking for a sequence of actions that reaches the goal is called **Search**. A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called an **execution Phase.**

Thus we have a simple **"formulate , search, execute"** design for the agent as shown in the Figure.

```
function SIMPLE_PROBLEM_SOLVING_AGENT( percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ←UPDATE-STATE(state, percept)
    if seq is empty then
        goal ←FORMULATE-GOAL(state)
            problem ← FORMULATE-PROBLEM(state, goal)
            seq ← SEARCH(problem)
            IF seq= failure then return a null action
            action ← FIRST(seq)
            seq ← REST(seq)
            return action
```

Well- defined Problems and Solutions

A **Problem** can be defined formally by 5 components

- The **initial state** that the agent starts in.

  Example : The initial state for our agent in Romania might be described as In(Arad)

- A description of the possible **actions** available to the agent. Given a particular state s, ACTION(s) returns the set of actions that can be executed in s. For example, from the state In (Arad) , the applicable actions are {Go(Sibiu), Go(Timisoara), Go(Zerind)}
- A description of what each action does: the formal name for this is the **transition model**, specified by a function RESULT(s,a) that returns the state that results from doing action a in state s.

  RESULT(In(Arad),Go (Zerind)) = In (Zerind)

- The **goal state** , which determines whether a given state is a goal state. The agent's goal in Romania is the singleton set {In(Bucharest)}
- A **path cost** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure.

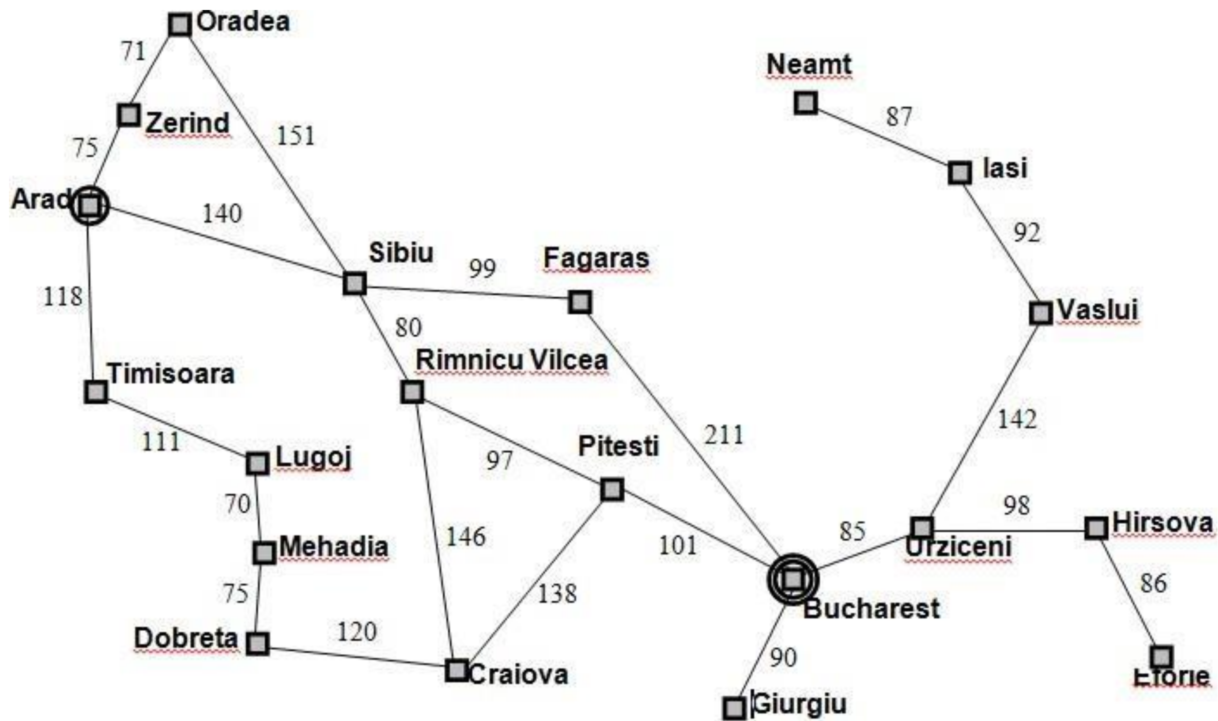


Fig : A simplified road map of part of Romania

On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest Formulate goal: be in Bucharest
Formulate problem:
states: various cities
actions: drive between cities
Find solution: sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Problem Formulation is the process of deciding what actions and states to consider, given a goal. Knowledge available to the agent is considered

- Current state
- Outcome of actions

<u>Example Problems</u>

A **Toy Problem** is intended to illustrate or exercise various problem-solving methods. A **real-world problem** is one whose solutions people actually care about.

Toy Problems:

<u>Vaccum World</u>

States : The state is determined by both the agent location and the dirt locations. The agent is in one of the 2 locations, each of which might or might not contain dirt. Thus there are $2*2^2 = 8$

possible world states.

Initial state: Any state can be designated as the initial state.

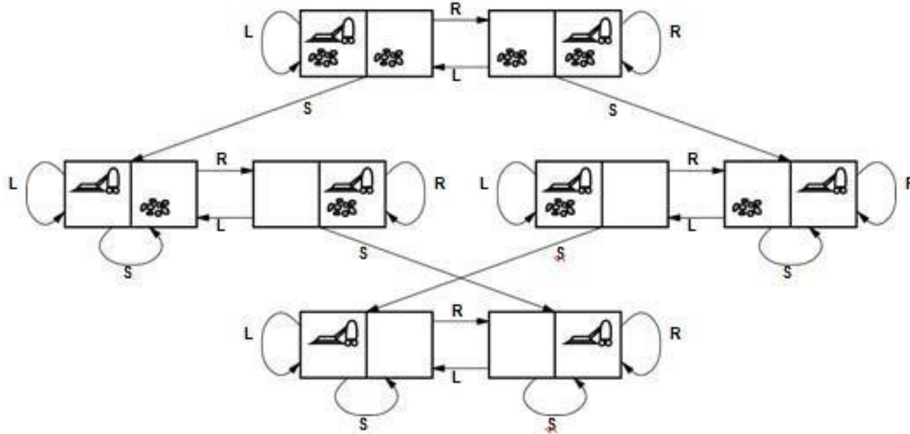
Actions: In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.

Transition model: The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Suck*ing in a clean square have no effect. The complete state space is shown in Figure.

Goal test: This checks whether all the squares are clean.

Path cost: Each step costs 1, so the path cost is the number of steps in the path.

Example: vacuum world state space graph



8- Puzzle Problem

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

States: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
Initial state: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states.
Actions: The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*. Different subsets of these are possible depending on where the blank is.
Transition model: Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state in Figure 3.4, the resulting state has the 5 and the blank switched. **Goal test**: This checks whether the state matches the goal configuration shown in Figure. **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

Search Algorithms

Figure shows the first few steps in growing the search tree for finding a route from Arad to Bucharest. The root node of the tree corresponds to the initial state, *In(Arad)*. The first step is to test whether this is a goal state. We do this by **expanding** the current state; that is, GENERATING applying each legal action to the current state, thereby **generating** a new set of states. In PARENT NODE this case, we add three branches from the **parent node** *In(Arad)* leading to three new **child** CHILD NODE **nodes**: *In(Sibiu), In(Timisoara),* and *In(Zerind)*. Now we must choose which of these three possibilities to consider further.

Suppose we choose Sibiu first. We check to see whether it is a goal state (it is not) and then expand it to get *In(Arad)*, *In(Fagaras)*, *In(Oradea)*, and *In(RimnicuVilcea)*. We can then choose any of these four or go

LEAF NODE back and choose Timisoara or Zerind. Each of these six nodes is a **leaf node**, that is, a

node with no children in the tree. The set of all leaf nodes available for expansion at any given FRONTIER point is called the **frontier**.

The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand. The general TREE-SEARCH algorithm is shown as follows:

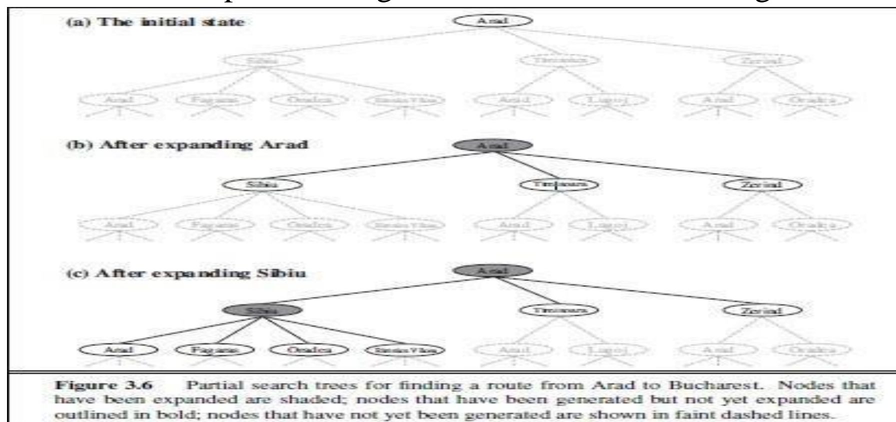


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     add the node to the explored set
     expand the chosen node, adding the resulting nodes to the frontier
        only if not in the frontier or explored set
```

PROBLEM-SOLVING APPROACH IN ARTIFICIAL INTELLIGENCE PROBLEMS

The **reflex agents** are known as the simplest agents because they directly map states into actions. Unfortunately, these agents fail to operate in an environment where the mapping is too large to store and learn. **Goal-based agent,** on the other hand, considers future actions and the desired outcomes.

Here, we will discuss one type of goal-based agent known as a **problem-solving agent**, which uses atomic representation with no internal states visible to the *problem-solving algorithms*.

Problem-solving agent

The problem-solving agent perfoms precisely by defining problems and its several solutions.

- According to psychology, "*a problem-solving refers to a state where we wish to reach to a definite goal from a present state or condition.*"
- According to computer science, *a problem-solving is a part of artificial intelligence which encompasses a number of techniques such as algorithms, heuristics to solve a problem.*

Therefore, a problem-solving agent is a **goal-driven agent** and focuses on satisfying the goal.

PROBLEM DEFINITION

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

(i) **Define** the problem precisely. This definition must include specification of the initial situations and also final situations which constitute (i.e) acceptable solution to the problem.

(ii) **Analyze** the problem (i.e) important features have an immense (i.e) huge impact on the appropriateness of various techniques for solving the problems.

(iii) **Isolate and represent** the knowledge to solve the problem.

(iv) **Choose the best** problem – solving techniques and apply it to the particular problem.

<u>Steps performed by Problem-solving agent</u>

- **Goal Formulation:** It is the first and simplest step in problem-solving. It organizes the steps/sequence required to formulate one goal out of multiple goals as well as actions to

achieve that goal. Goal formulation is based on the current situation and the agent's performance measure (discussed below).

- **Problem Formulation:** It is the most important step of problem-solving which decides what actions should be taken to achieve the formulated goal. There are following five components involved in problem formulation:
- **Initial State:** It is the starting state or initial step of the agent towards its goal.
- **Actions:** It is the description of the possible actions available to the agent.
- **Transition Model:** It describes what each action does.
- **Goal Test:** It determines if the given state is a goal state.
- **Path cost:** It assigns a numeric cost to each path that follows the goal. The problem-solving agent selects a cost function, which reflects its performance measure. Remember, **an optimal solution has the lowest path cost among all the solutions.**

Note: Initial state, actions, and **transition model** together define the **state-space** of the problem implicitly. State-space of a problem is a set of all states which can be reached from the initial state followed by any sequence of actions. The state-space forms a directed map or graph where nodes are the states, links between the nodes are actions, and the path is a sequence of states connected by the sequence of actions.

- **Search:** It identifies all the best possible sequence of actions to reach the goal state from the current state. It takes a problem as an input and returns solution as its output.
- **Solution:** It finds the best algorithm out of various algorithms, which may be proven as the best optimal solution.
- **Execution:** It executes the best optimal solution from the searching algorithms to reach the goal state from the current state.

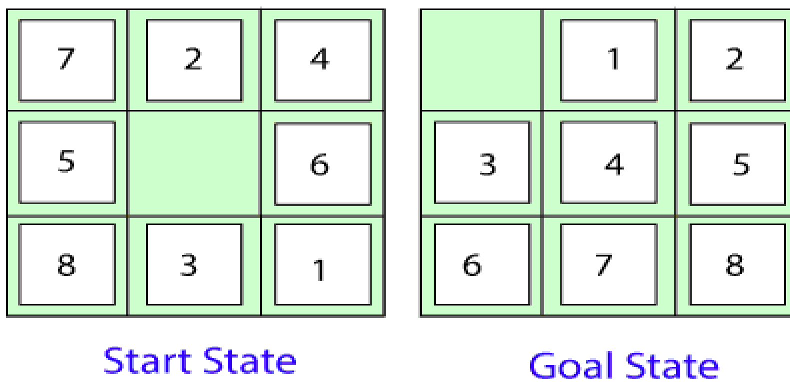
Example Problems

Basically, there are two types of problem approaches:

- **Toy Problem:** It is a concise and exact description of the problem which is used by the researchers to compare the performance of algorithms.
- **Real-world Problem:** It is real-world based problems which require solutions. Unlike a toy problem, it does not depend on descriptions, but we can have a general formulation of the problem.

Some Toy Problems

- **8 Puzzle Problem:** Here, we have a 3×3 matrix with movable tiles numbered from 1 to 8 with a blank space. The tile adjacent to the blank space can slide into that space. The objective is to reach a specified goal state similar to the goal state, as shown in the below figure.
- In the figure, our task is to convert the current state into goal state by sliding digits into the blank space.

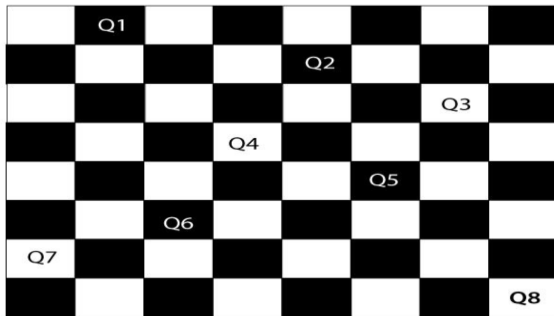


In the above figure, our task is to convert the current(Start) state into goal state by sliding digits into the blank space.

The problem formulation is as follows:

- **States:** It describes the location of each numbered tiles and the blank tile.
- **Initial State:** We can start from any state as the initial state.
- **Actions:** Here, actions of the blank space is defined, i.e., either **left, right, up or down**
- **Transition Model:** It returns the resulting state as per the given state and actions.
- **Goal test:** It identifies whether we have reached the correct goal-state.
- **Path cost:** The path cost is the number of steps in the path where the cost of each step is 1.
  Note: The 8-puzzle problem is a type of **sliding-block problem** which is used for testing new search algorithms in artificial intelligence.

- **8-queens problem:** The aim of this problem is to place eight queens on a chessboard in an order where no queen may attack another. A queen can attack other queens either **diagonally or in same row and column.**
  From the following figure, we can understand the problem as well as its correct solution.



It is noticed from the above figure that each queen is set into the chessboard in a position where no other queen is placed diagonally, in same row or column. Therefore, it is one right approach to the 8-queens problem.

For this problem, there are two main kinds of formulation:

1. **Incremental formulation:** It starts from an empty state where the operator augments a queen at each step.

Following steps are involved in this formulation:

- **States:** Arrangement of any 0 to 8 queens on the chessboard
- **Initial State:** An empty chessboard
- **Actions:** Add a queen to any empty box.
- **Transition model:** Returns the chessboard with the queen added in a box.
- **Goal test:** Checks whether 8-queens are placed on the chessboard without any attack.
- **Path cost:** There is no need for path cost because only final states are counted.

In this formulation, there is approximately **1.8×10^{14}** possible sequence to investigate.

2. **Complete-state formulation:** It starts with all the 8-queens on the chessboard and moves them around, saving from the attacks.

Following steps are involved in this formulation

- **States:** Arrangement of all the 8 queens one per column with no queen attacking the other queen.
- **Actions:** Move the queen at the location where it is safe from the attacks.

This formulation is better than the incremental formulation as it reduces the state space from **1.8×10^{14}** to **2057**, and it is easy to find the solutions.

Some Real-world problems

- **Traveling salesperson problem(TSP):** It is a **touring problem** where the salesman can visit each city only once. The objective is to find the shortest tour and sell-out the stuff in each city.
- **VLSI Layout problem:** In this problem, millions of components and connections are positioned on a chip in order to minimize the area, circuit-delays, stray-capacitances, and maximizing the manufacturing yield.

The layout problem is split into two parts:

- **Cell layout:** Here, the primitive components of the circuit are grouped into cells, each performing its specific function. Each cell has a fixed shape and size. The task is to place the cells on the chip without overlapping each other.
- **Channel routing:** It finds a specific route for each wire through the gaps between the cells.
- **Protein Design:** The objective is to find a sequence of amino acids which will fold into 3D protein having a property to cure some disease.

Searching for solutions

We have seen many problems. Now, there is a need to search for solutions to solve them. In this section, we will understand how searching can be used by the agent to solve a problem.

For solving different kinds of problem, an agent makes use of different strategies to reach the goal by searching the best possible algorithms. This process of searching is known as **search strategy.**

SEARCH FOR SOLUTIONS/ SEARCH STRATEGIES

Search Algorithm Terminologies:

- **Search:** Searchingis a step by step procedure to solve a search-problem in a given searchspace. A search problem can have three main factors:
  a. **Search Space:** Search space represents a set of possible solutions, which a systemmay have.
  b. **Start State:** It is a state from where agent begins **the search**.
  c. **Goal test:** It is a function which observe the current state and returns whether thegoal state is achieved or not.

- **Search tree:** A tree representation of search problem is called Search tree. The root of thesearch 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 transitionmodel.
- **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.

Properties of Search Algorithms:

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

Completeness: A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.

Optimality: If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.

Time Complexity: Time complexity is a measure of time for an algorithm to complete its task.

Space Complexity: It is the maximum storage space required at any point during the search, asthe complexity of the problem.

Types of Search Algorithms

There are two types of strategies that describe a solution for a given problem:

1. **Uninformed Search (Blind Search)**

This type of search strategy does not have any additional information about the states except theinformation provided in the problem definition. They can only generate the successors and distinguish a goal state from a non-goal state. These type of search does not maintain any internal state, that's why it is also known as **Blind search.**

There are following types of uninformed searches:

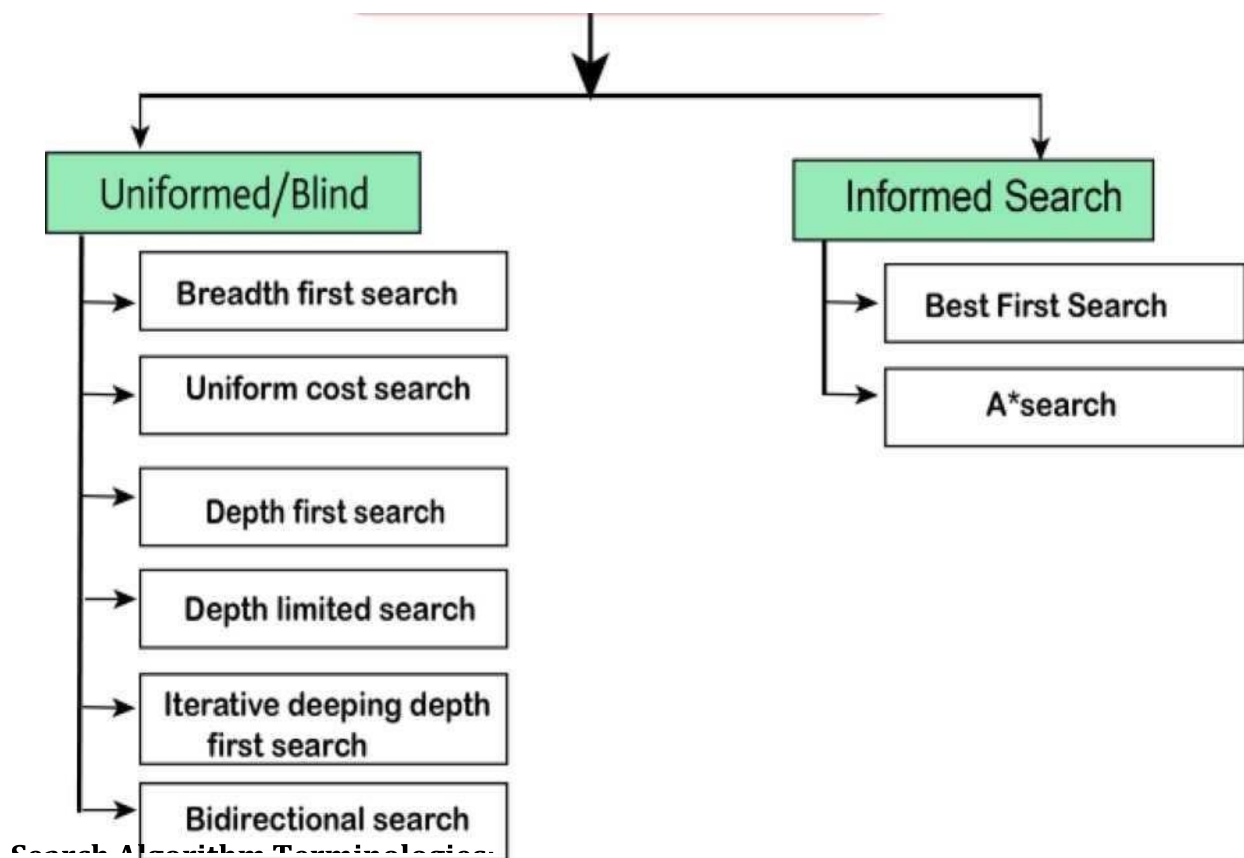
- Breadth-first search
- Uniform cost search
- Depth-first search
- Depth-limited search

- Iterative deepening search
- Bidirectional search

2. **Informed Search (Heuristic Search)**

This type of search strategy contains some additional information about the states beyond the problem definition. This search uses problem-specific knowledge to find more efficient solutions. This search maintains some sort of internal states via heuristic functions (which provides hints), soit is also called **heuristic search.**

- Best first search (Greedy search)
- A* search



UNINFORMED SEARCH ALGORITHMS

Uninformed search is a class of general-purpose search algorithms which operates in brute force-way. Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search.

Following are the various types of uninformed search algorithms:

1. BREADTH-FIRST SEARCH:

- 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 successornode at the current level before moving to nodes of next level.
- 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.

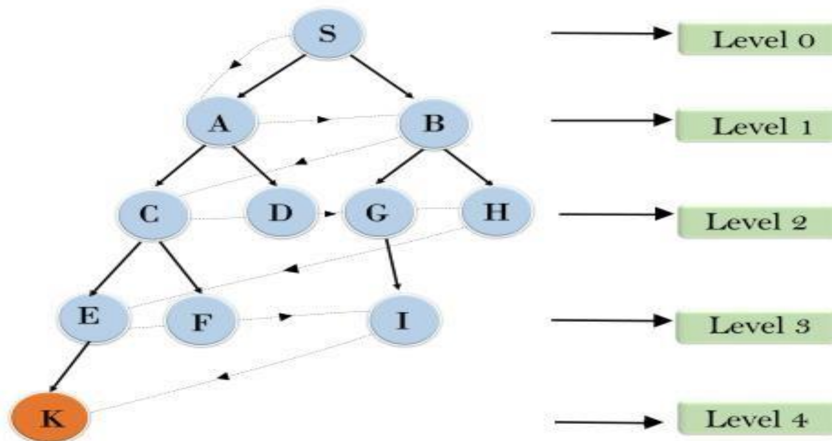
Disadvantages:

- 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.

Example:

In the below tree structure, we have shown the traversing of the tree using BFS algorithm fromthe 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.

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

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.

2. DEPTH-FIRST SEARCH

- Depth-first search isa recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows eachpath to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.

Note: Backtracking is an algorithm technique for finding all possible solutions usingrecursion.

Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the pathfrom root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in theright path).

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.

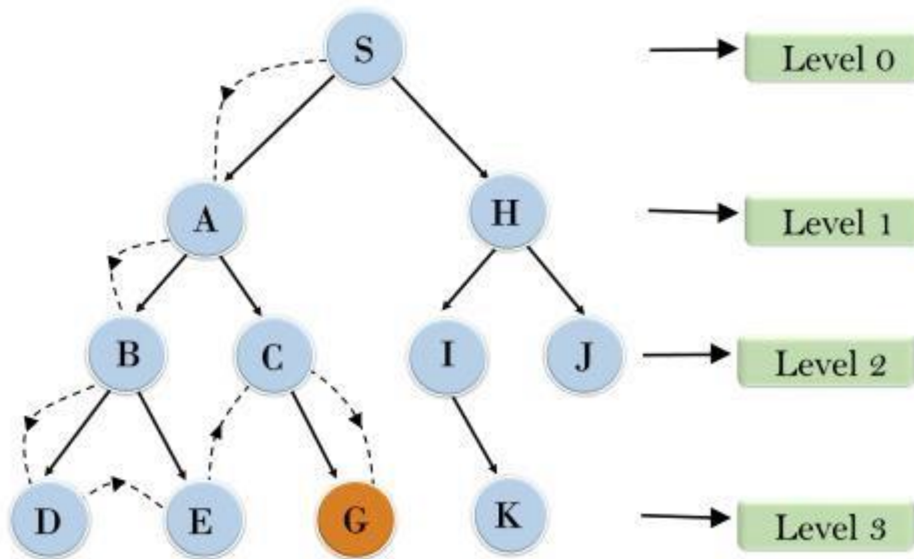
Example:

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. Afterbacktracking it will traverse node C and then G, and here it will terminate as it found



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)**.

3.DEPTH-LIMITED SEARCH ALGORITHM:

Completeness: DFS search algorithm is complete within finite state space as it will expand

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

- 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.

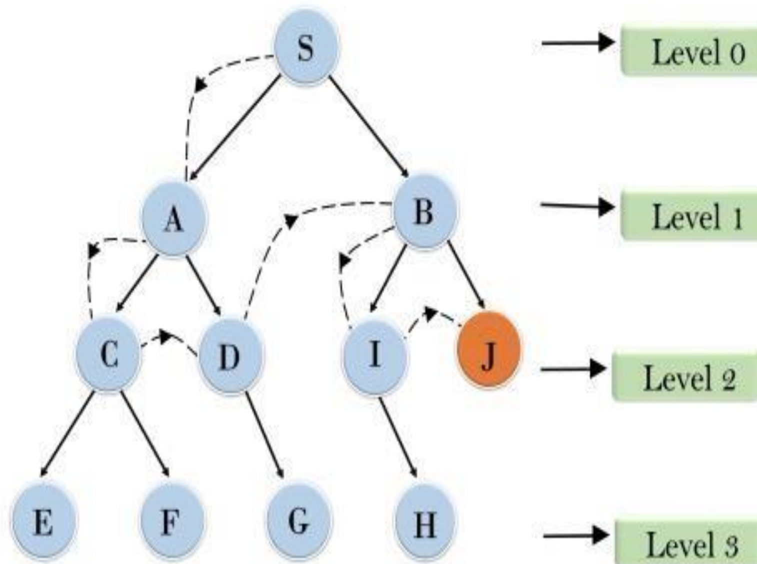
Advantages

Depth-limited search is Memory

Disadvantage

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

Example:



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

Time Complexity: Time complexity of DLS algorithm is

Space Complexity: Space complexity of DLS algorithm is

Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not

4. UNIFORM-COST SEARCH ALGORITHM:

Uniform-cost search is a searching algorithm used for traversing a weighted

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.

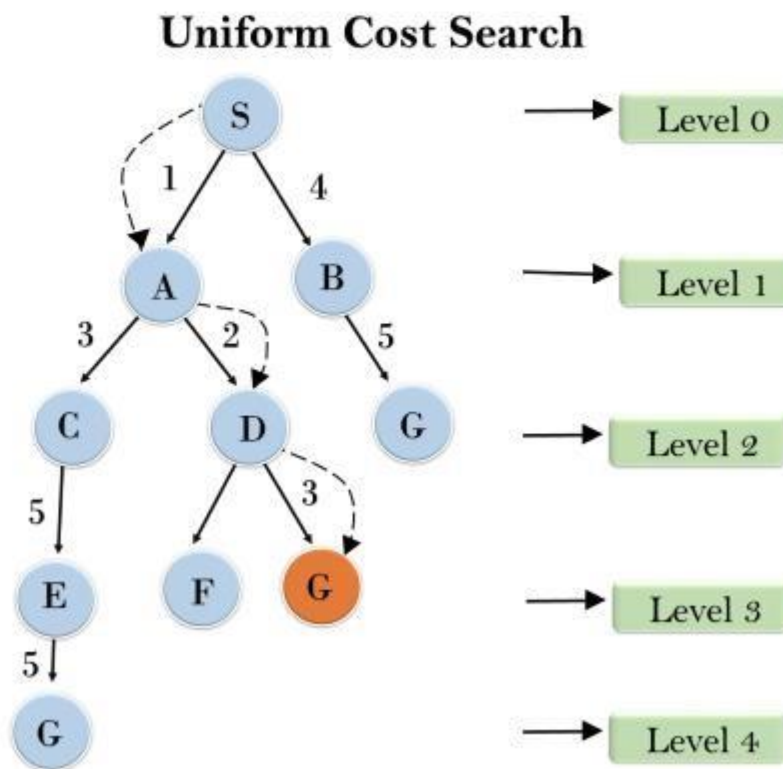
Advantages:

- Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantages:

- It does not care about the number of steps involve in searching and only concernedabout path cost. Due to which this algorithm may be stuck in an

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 $\mathbf{O(b^{1 + [C^*/\varepsilon]})}$/.

Space Complexity:

The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $\mathbf{O(b^{1 + [C^*/\varepsilon]})}$.

Optimal:

Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

5. ITERATIVE DEEPENINGDEPTH-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.

Advantages:

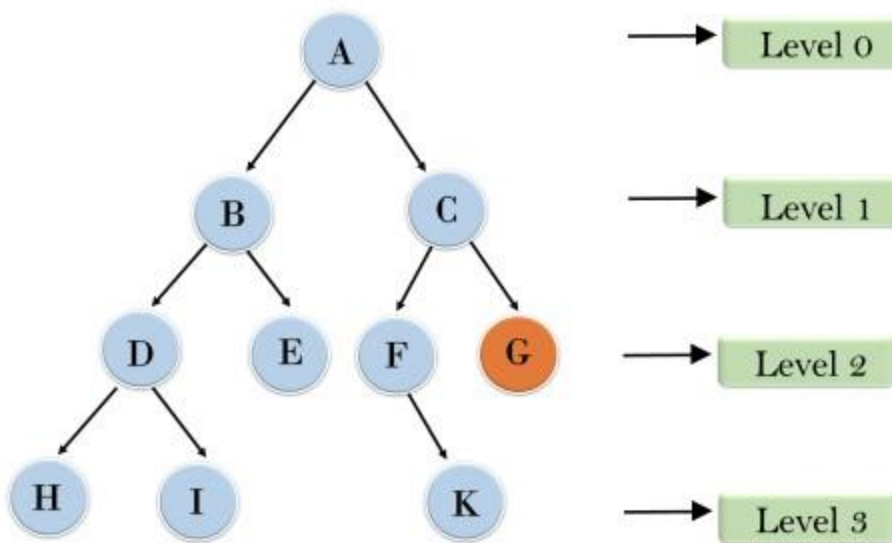
- Itcombines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

Disadvantages:

The main drawback of IDDFS is that it repeats all the work of the previousphase.

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:



1'st Iteration -- > A
2'n Iteration----> A, B, C
d Iteration------>A, B, D, E, C, F, G
3'r Iteration ---- >A, B, D, H, I, E, C, F, K, G
In the fourth iteration, the algorithm will find the goal

Completeness:

This algorithm is complete is ifthe 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.

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 goalnode. Bidirectional search replaces one single search graph with two small subgraphs in whichone starts the search from an initial vertex and other starts from goal vertex. The search stops

Bidirectional search can use search techniques such as BFS, DFS, DLS,

Advantages

- Bidirectional search is fast.
- Bidirectional search requires less memory

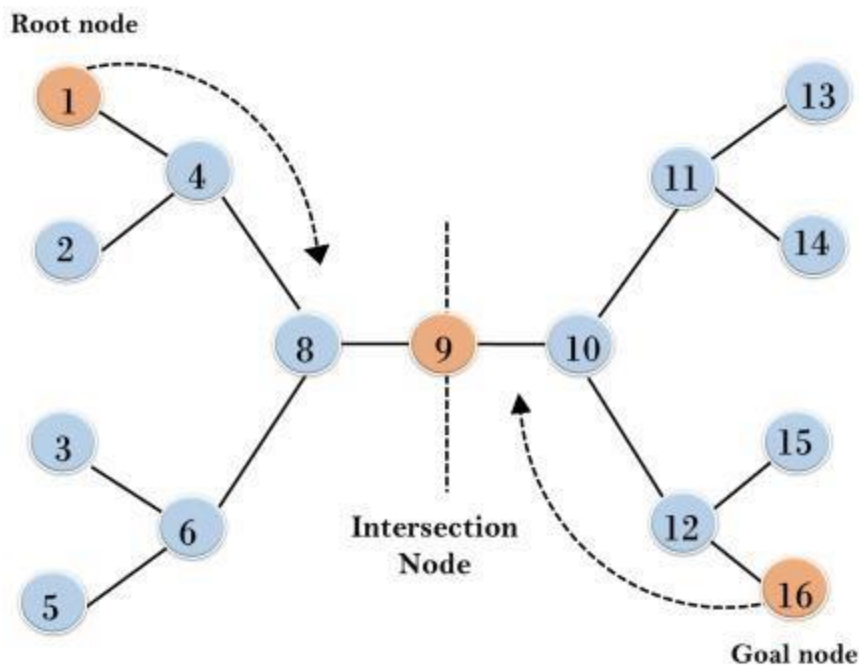
Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- **In bidirectional search, one should know the goal state in advance.**

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.



Completeness: Bidirectional Search is complete if we use BFS in both

Time Complexity: Time complexity of bidirectional search using BFS is

Space Complexity: Space complexity of bidirectional search is

Optimal: Bidirectional search is optimal

INFORMED SEARCH ALGORITHMS

So far we have talked about the uninformed search algorithms which looked through search space for all possible solutions of the problem without having any additional knowledge about search space. But informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc. This knowledge help agents to explore less to the search space and find more efficiently the goal node.

The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal.

The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Admissibility of the heuristic function is given as:

1. h(n) <= h*(n)

Here h(n) is heuristic cost, and h*(n) is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

Pure Heuristic Search:

Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.

On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues unit a goal state is found.

In the informed search we will discuss two main algorithms which are given below:

- **Best First Search Algorithm(Greedy search)**
- **A* Search Algorithm**

1.) BEST-FIRST SEARCH ALGORITHM (GREEDY SEARCH):

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

1. f(n)= g(n).

Were, h(n)= estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the priority queue.

Best first search algorithm:

- **Step 1:** Place the starting node into the OPEN list.

- **Step 2:** If the OPEN list is empty, Stop and return failure.
- **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
- **Step 4:** Expand the node n, and generate the successors of node n.
- **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- **Step 6:** For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
- **Step 7:** Return to Step 2.

Advantages:

- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

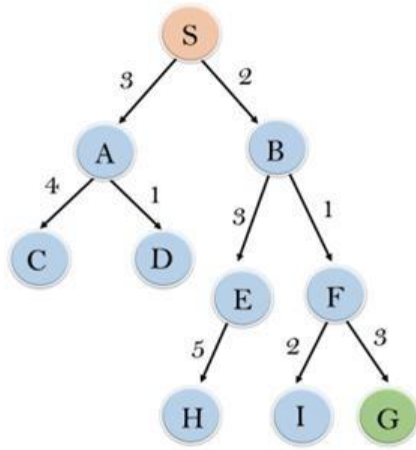
Disadvantages:

- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.

Example:

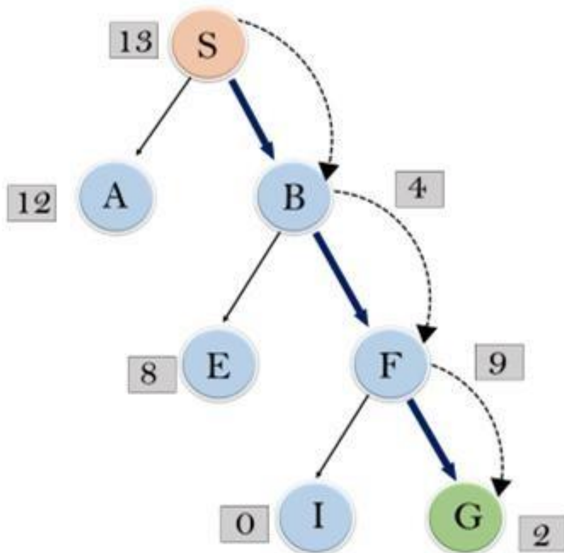
Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.

Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists. Following are the iteration for traversing the above example.



Expand the nodes of S and put in the CLOSED list

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

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

Iteration 2: Open [E, F, A], Closed [S, B]
: Open [E, A], Closed [S, B, F]

Iteration 3: Open [I, G, E, A], Closed [S, B, F]
: Open [I, E, A], Closed [S, B, F, G]

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

Time Complexity: The worst case time complexity of Greedy best first search is $O(b^m)$.

Space Complexity: The worst case space complexity of Greedy best first search is $O(b^m)$. Where, m is the maximum depth of the search space.

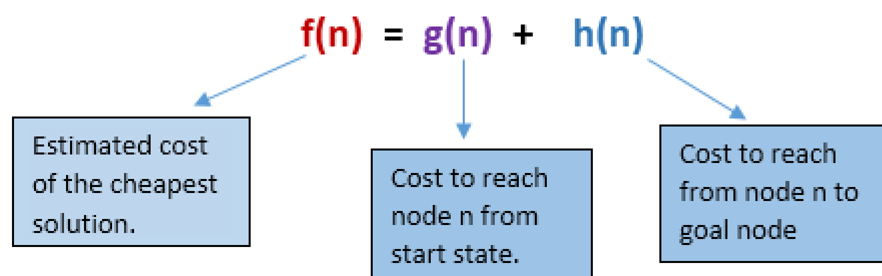
Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

Optimal: Greedy best first search algorithm is not optimal.

2.) A* SEARCH ALGORITHM:

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).

In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a **fitness number**.



Algorithm of A* search:

Step1: Place the starting node in the OPEN list.

At each point in the search space, only those node is expanded which have the lowest value of f(n), and the algorithm terminates when the goal node is found.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Return to **Step 2**.

Advantages:

- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

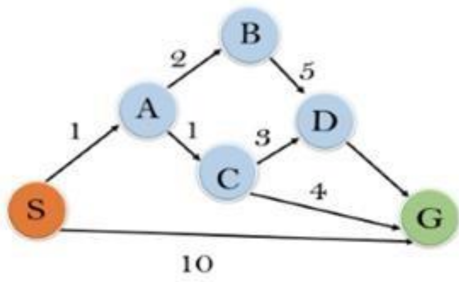
Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

Example:

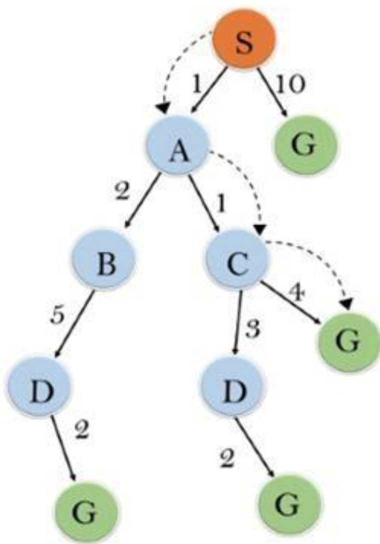
In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.

Here we will use OPEN and CLOSED list.



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Solution:



Initialization: {(S, 5)}

Iteration1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Iteration 4 will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.

Points to remember:

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n)<="" li="">

Complete: A* algorithm is complete as long as:

- Branching factor is finite.
- Cost at every action is fixed.

Optimal: A* search algorithm is optimal if it follows below two conditions:

- **Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
- **Consistency:** Second required condition is consistency for only A* graph-search.

If the heuristic function is admissible, then A* tree search will always find the least cost path.

Time Complexity: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is **O(b^d)**

BEYOND CLASSICAL SEARCH

1. **Hill Climbing search/Steepest-ascent Hill Climbing/Greedy LocalSearch**

- moves in the direction of increasing value(uphill) to reach a peak state(goal).
- it terminates when it reaches a "peak" where no neighbor has a higher value.
- Used when a good heuristic is available
- does not maintain a search tree, so the current node data structure need onlyto record the state and its objective function value(i.e., f(n)
- At each step the current node is replaced by the best neighbor

- Algorithm

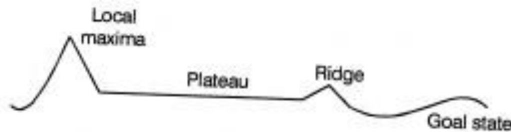
MAX Version

function HILL-CLIMBING(problem) **returns** a state that is a local maximumcurrent <-
MAKE-NODE(INITIAL-STATE[problem])
loop do
neighbor <- a highest-valued successor of current
if VALUE[neighbor] ≤ VALUE[current] **then return**
STATE[current]current <- neighbor

MIN Version

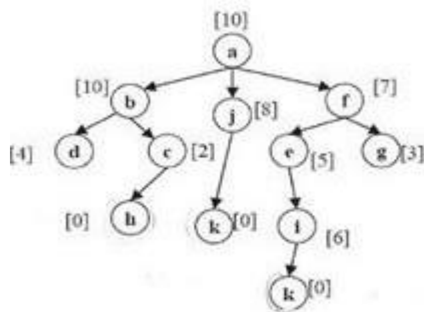
function HILL-CLIMBING(problem) **returns** a state that is a local maximumcurrent <-
MAKE-NODE(INITIAL-STATE[problem])
loop do
neighbor <- a lowest-valued successor of current
if VALUE[neighbor] ≥ VALUE[current] **then return**
STATE[current]current <- neighbor

- Example: Drawbacks of hill climbing(pictorial representation)



1. Local maxima/foot hills /no uphills step

- A state that is better(higher) than each of its neighboring states, but notbetter(lower) than some other states far away
- Example: consider the following tree. 'a' is an initial state and 'h' and 'k'are final states. The number near the states is the heuristic value



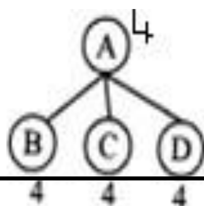
ɔy applying hill climbing algorithm, weget **a->f->g** which is **ocal maxima**.

Hill climbing cannot go back and choose a new node (i.e. 'e') since it keeps no history

- to overcome local maxima , **backtrack** to one of the previous states andexplore other directions

2. Plateau (flat-plateau whose edges go downhill. **or shoulder** - a plateau that hasan uphill edge.)

- **flat area** of the search space in which all the **neighboring(successors) stateshave same value**=> therefore **cannot determine the best direction**
- Example



The evaluation function value of B, C, D are same and at the same time Best successor has the same value as the current state

- To overcome plateau skip few states or make a **jump** in new directions

3. <u>**Ridges**</u>
   - A special kind of local maxima
   - Defn-1: An area of the search space which is higher than the surroundingarea and that itself has a slope
   - Defn-2: is a sequence of local maxima
   - To overcome ridges **explore in several directions** to find out the correctpath

- <u>**Variants of hill-climbing**</u>

1. **Stochastic hill climbing** => choose successor at random
   - **does not focus on all the nodes**
   - selects **one node at random** and decides whether it should be expanded orsearch for a better one.
   - the probability of selection can vary with the steepness of the uphill move.
   - Better than steepest ascent
2. **First-choice hill climbing/Simple hill climbing** => choose firstsuccessor that is better than current state
   - implements stochastic hill climbing by generating successors randomlyuntil one is generated that is better than the current state
3. **Random-restart hill climbing** => try with different initial states ifchosen initial state fails
   - based on **try and try strategy**
   - It conducts a **series of hill-climbing searches from randomly generated initial state**, stopping when a goal is found. (Or) generates different initial state until it finds a goal state.

2. <u>**Simulated annealing**</u>

- **Annealing:** is the process of **controlling cooling**
- Inspired from <u>the process of annealing in metal work</u>
- Physical process in which physical substances such as metals are melted and then gradually cooled until some solid state is reached=> goal is to produce a minimal energy final state.
- The rate at which the system is cooled is called **annealing schedule**
- During cooling process, sometimes a worsen state (unaccepted state) can bemade.
- The worsen state can be accepted by some probability(accept this state withprobability less than 1)
- The probability is defined as
- **Note: the chance of accepting worst solution is done only when thetemperature is high**
- **Physical annealing is simulated=> simulated annealing**

- Simulated Annealing
  - Variants of hill climbing in which at the beginning of the process Some downhillmoves can be made=> helps to avoid local maxima
  - Used when the number of moves from a given state is large
  - The probability in simulated annealing is defined as
  - Annealing schedule consists of 4 components
    i) Starting temperature(T_s)
    ii) Final temperature(T_f)
    iii) Temperature length(T_L): criteria used to decide when the temperatureof the system should be reduced)
    iv) Cooling criteria/temperature decrement(f(t)): amount by which thetemperature will be reduced each time it is changed
  - Algorithm in simple words
    - Generate a random new neighbor from current state.
    - If it's better take it.
    - If it's worse then take it with **some probability proportional to thetemperature and the delta between the new and old states**
  - Algorithm

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← next.VALUE – current.VALUE
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(ΔE/T)
```

- **Property of simulated annealing search :**T decreases slowly enough thensimulated annealing search will find a global optimum with probability one
- **Applications**
  a. VLSI layout
  b. Airline scheduling
- **Difference between simulated annealing and hill climbing**
  1. Annealing schedule must be maintained
  2. Moves to the worst state is accepted
  3. Maintain best state found so far

17. LOCAL SEARCH IN CONTINUOUS SPACE:

- The distinction between discrete and continuous environments pointing out that most real-world environments are continuous.
- A discrete variable or categorical variable is a type of statistical variable that can assume only fixed number of distinct values.
- Continuous variable, as the name suggest is a random variable that assumes all the possible values in a continuum.
  - Which leads to a solution state required to reach the goal node.
  - But beyond these **"classical search algorithms,"** we have some **"local search algorithms" where the path cost does not matters, and only focus on solution-state needed to reach the goal node.**
    - **Example: Greedy BFS* Algorithm.**
- A local search algorithm completes its task by traversing on a single current node rather than multiple paths and following the neighbors of that node generally.
    - **Example:** Hill climbing and simulated annealing can handle continuous state and action spaces, because they have infinite branching factors.

Solution for Continuous Space:

- One way to avoid continuous problems is simply to **discretize the neighborhood of each** state.
- Many methods attempt to use the **gradient of the landscape to find a maximum. The** gradient of the objective function is a vector ∇f that gives the magnitude and direction of the steepest slope.

Local search in continuous space:

- Given a continuous state space $S = \{(x_1, x_2, \ldots, x_N) \mid x_i \in \mathbb{R}\}$
- Given a continuous objective function $f(x_1, x_2, \ldots, x_N)$
- The gradient of the objective function is a vector $\nabla f = (\partial f/\partial x_1, \partial f/\partial x_2, \ldots, \partial f/\partial x_N)$
- The gradient gives the magnitude and direction of the steepest slope at a point.

18. SEARCHING WITH NON-DETERMINISTIC ACTIONS:

- In an environment, the agent can calculate exactly which state results from any sequence of actions and always knows which state it is in.

  - **Searching with non-deterministic Actions**
  - **Searching with partial observations**

- When the environment is nondeterministic, percepts tell the agent which of the possible outcomes of its actions has actually occurred.
- In a partially observable environment, every percept helps narrow down the set of possible states the agent might be in, thus making it easier for the agent to achieve its goals.

Example: Vacuum world, v2.0

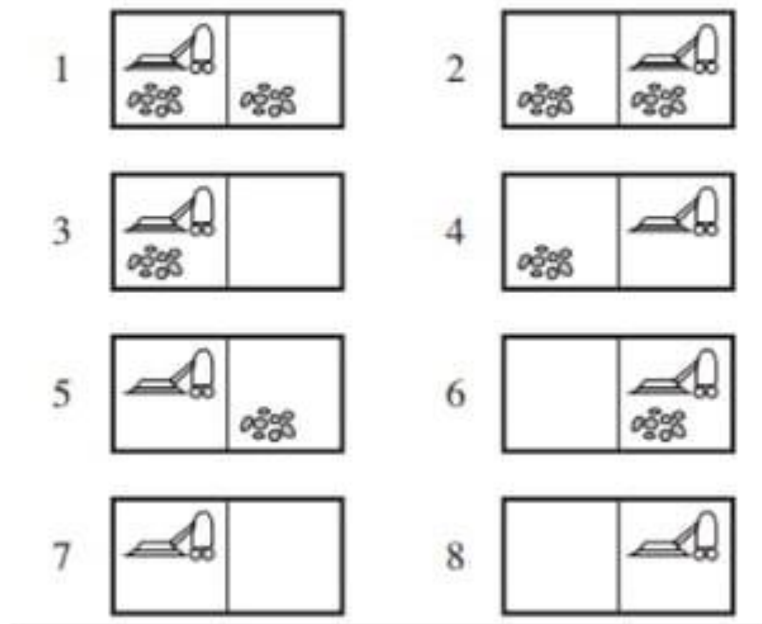
- In the erratic vacuum world, the **Suck action** works as follows:
  - When applied to a **dirty square** the action cleans the square and sometimes cleans up dirt in an adjacent square, too.
  - When applied to a **clean square** the action sometimes deposits dirt on the carpet.
- **Solutions** for nondeterministic problems can contain **nested if–then–else** statements; this means that they are **trees rather than sequences**.

The eight possible states of the vacuum world; states 7 and 8 are goal states.

- **Suck(p1, dirty)=** (p1,clean) and sometimes (p2, clean)
- **Suck(p1, clean)=** sometimes (p1,dirty)

Solution: contingency plan

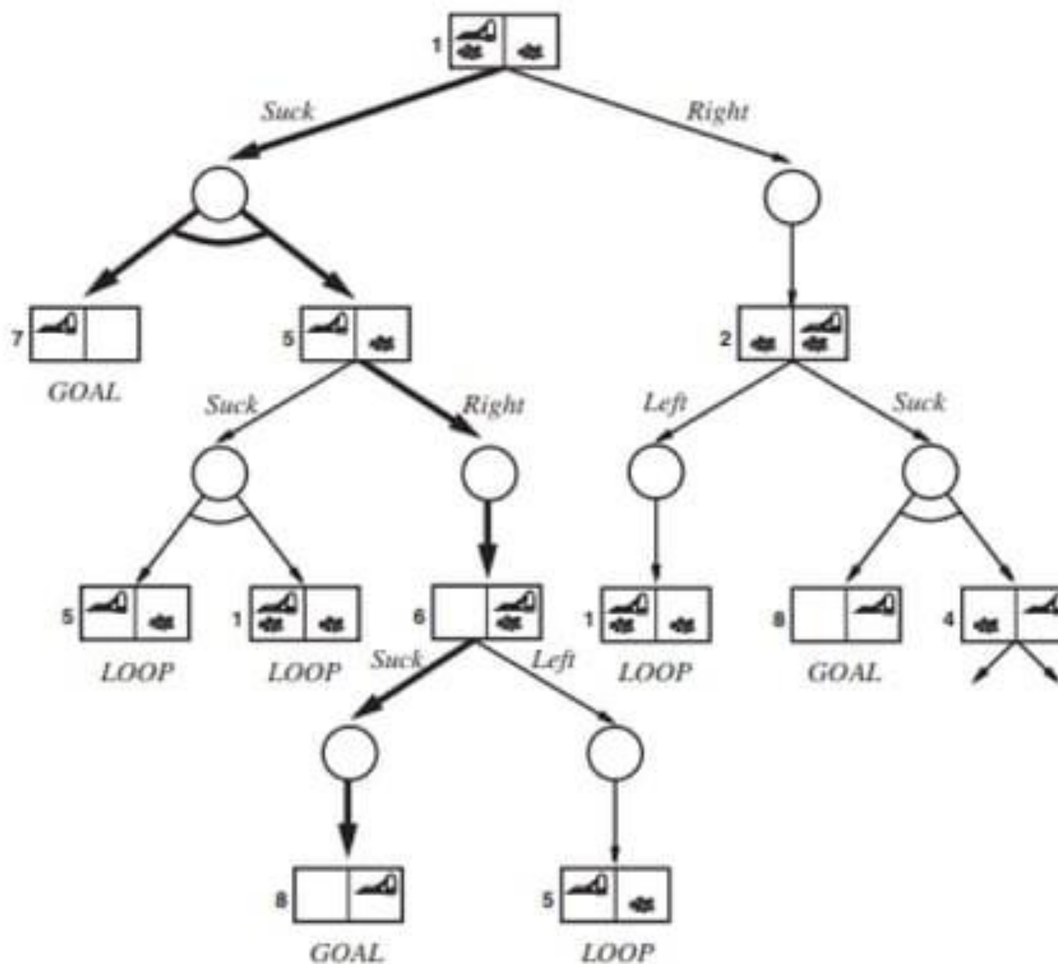
- [Suck, **if** State = 5 **then** [Right, Suck] **else** [ ]].
- **nested if–then–else statements**



AND–OR search trees:

- ***Non-deterministic action= there may be several possible outcomes***
- Search space is an **AND-OR tree**
  - Alternating **OR and AND layers**
  - ***Find solution= search this tree using same methods.***
- Solution in a non-deterministic search space
  - Not simple action sequence
  - ***Solution= subtree within search tree*** **with:**
    - Goal node at each leaf (plan covers all contingencies)
    - One action at each OR node
    - A branch at AND nodes, representing all possible outcomes
- **Execution of a solution = essentially**
- The first two levels of the search tree for the erratic vacuum world.
- State nodes are OR nodes where some action must be chosen.
- At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches.
- The solution found is shown in bold lines.

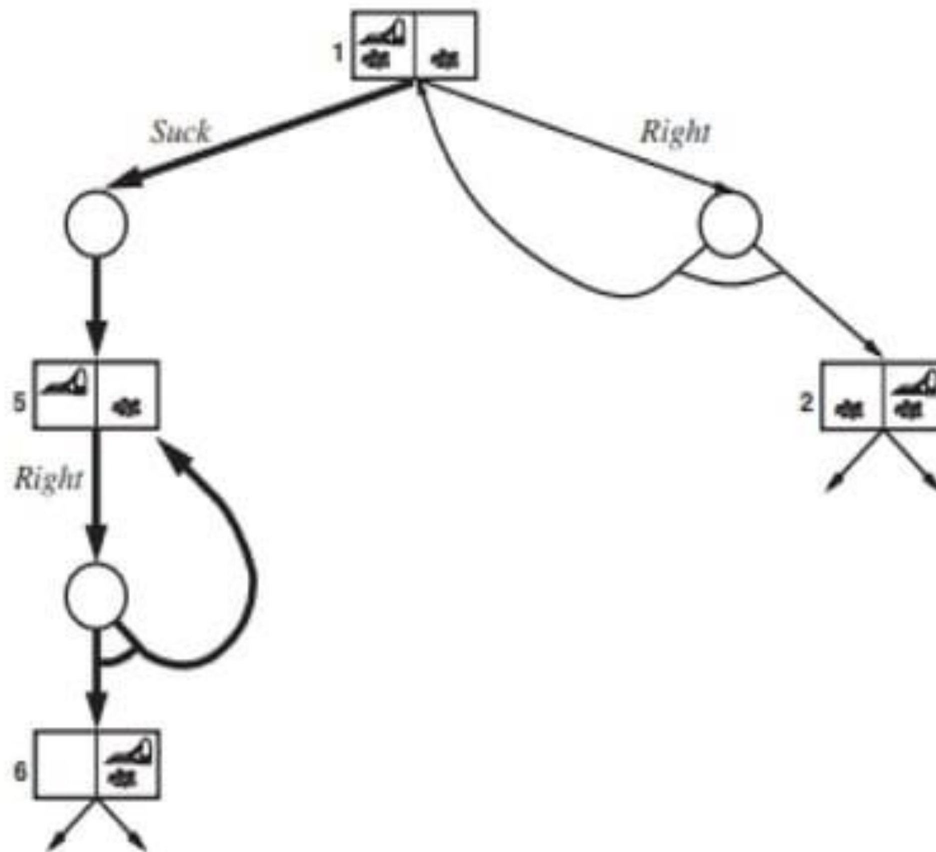
i. Non-deterministic search trees:



- **Start state = 1**
- **One solution:**
  - **Suck,**
  - **if(state=5) then [right, suck]**

ii. Non-determinism: Actions that fail (*Try, try again*):

- Action failure is often a non-deterministic outcome
  - **Creates a cycle in the search tree.**
- If no successful solution (plan) without a cycle:
  - **May return a solution that contains a cycle**
  - **Represents retrying the action**
- Infinite loop in plan execution?
  - Depends on environment
    - Action guaranteed to succeed eventually?
  - In practice: **can limit loops**
    - Plan no longer complete (could fail)



- Part of the search graph for the slippery vacuum world, where we have shown (some) cycles explicitly.
- All solutions for this problem are cyclic plans because there is no way to move reliably.

19. SEARCHING WITH PARTIAL OBSERVATIONS:

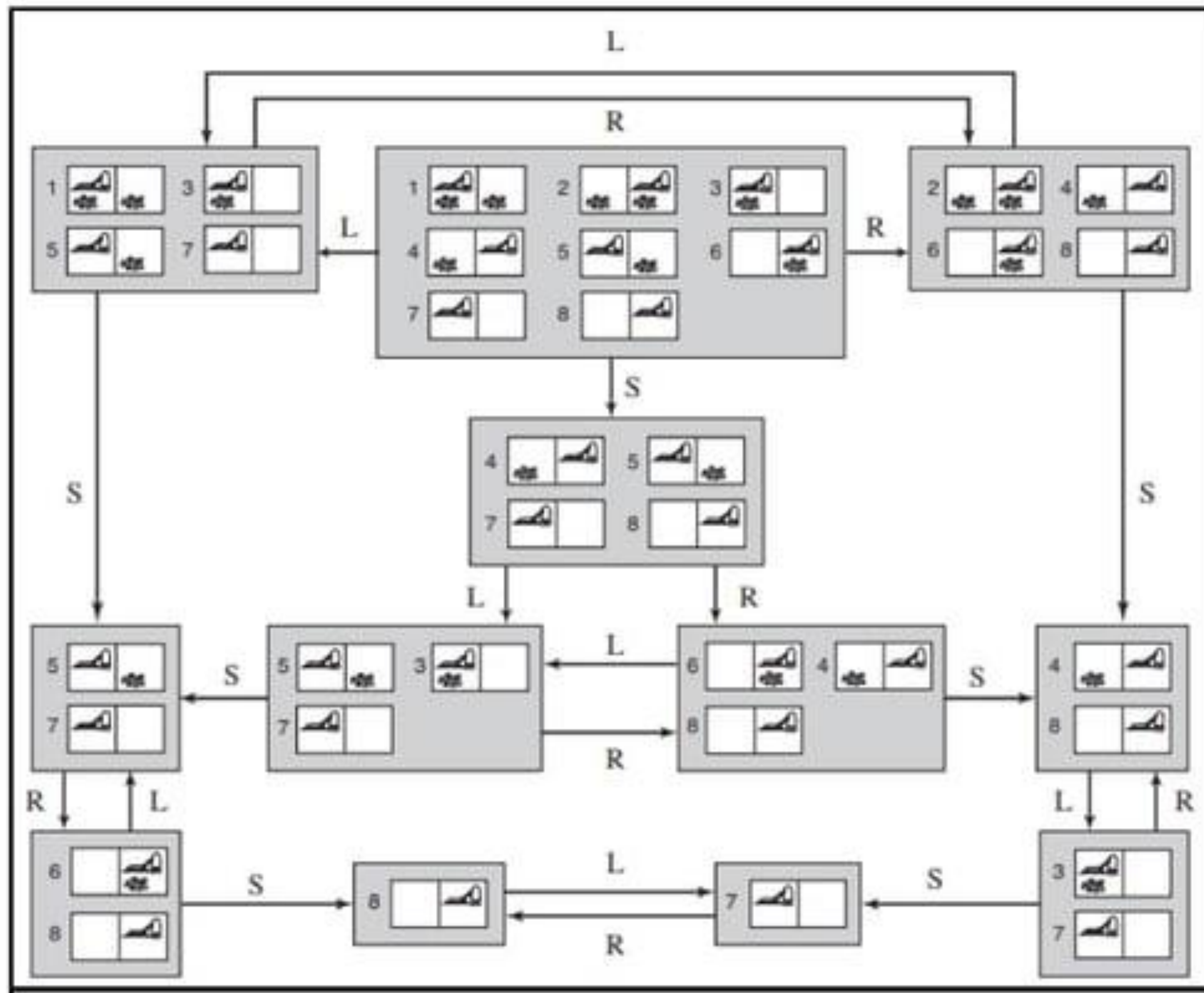
- In a partially observable environment, every percept helps narrow down the set of possible states the agent might be in, thus making it easier for the agent to achieve its goals.
- The *key* concept required for solving partially observable problems is the ***belief state***.
  - ***Belief state*** *-representing the agent's current belief about the possible physical states.*
    - **Searching with no observations**
    - **Searching with observations**

Conformant (sensorless) search: Example space:

- Belief state space for the super simple vacuum world

Observations:

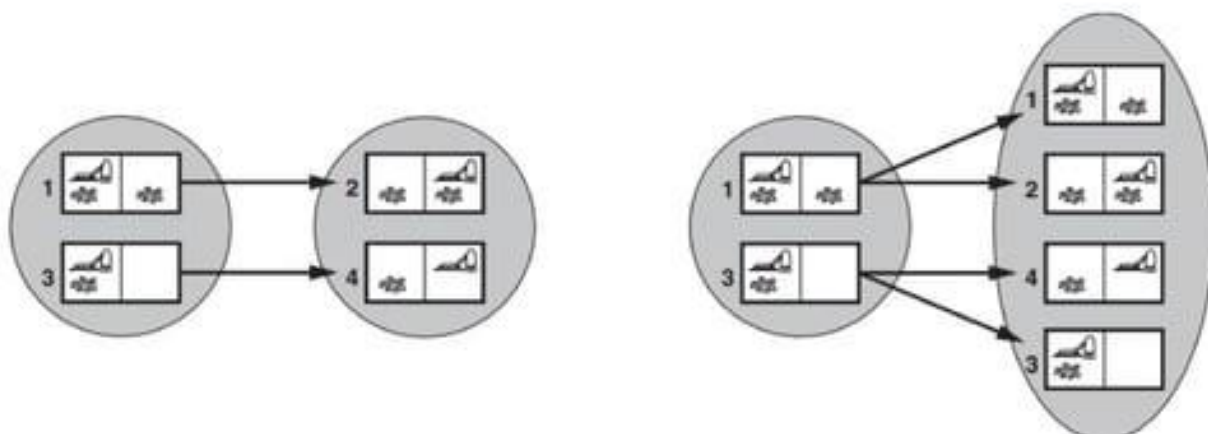
- Only **12 reachable states**. Versus **2^8= 256** possible belief states
- State space still gets huge very fast! à seldom feasible in practice
- We need sensors! à **Reduce state space** greatly!



i. **Searching with no observations:**

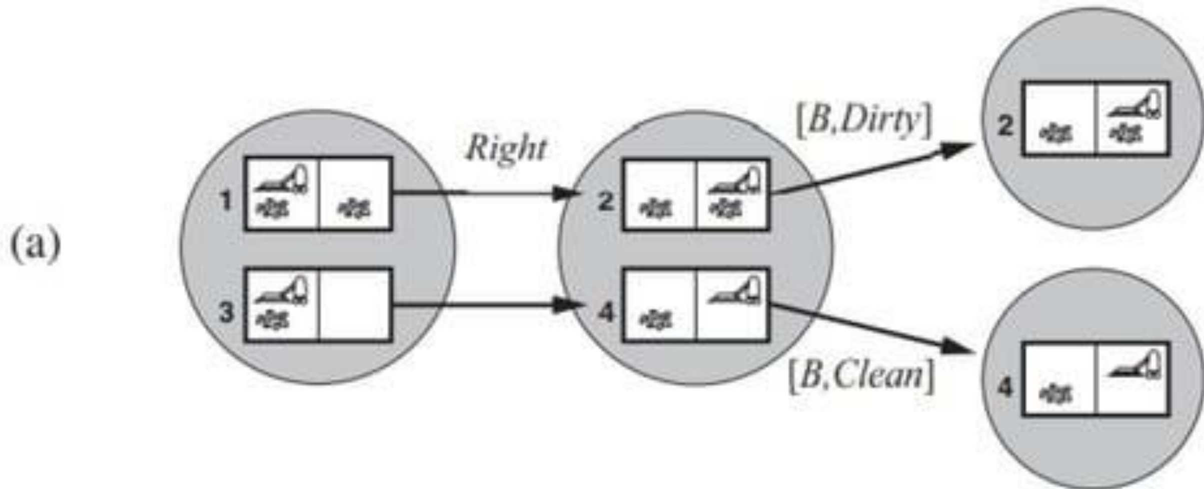
(a) Predicting the next belief state for the sensorless vacuum world with a deterministic action, Right.

(b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

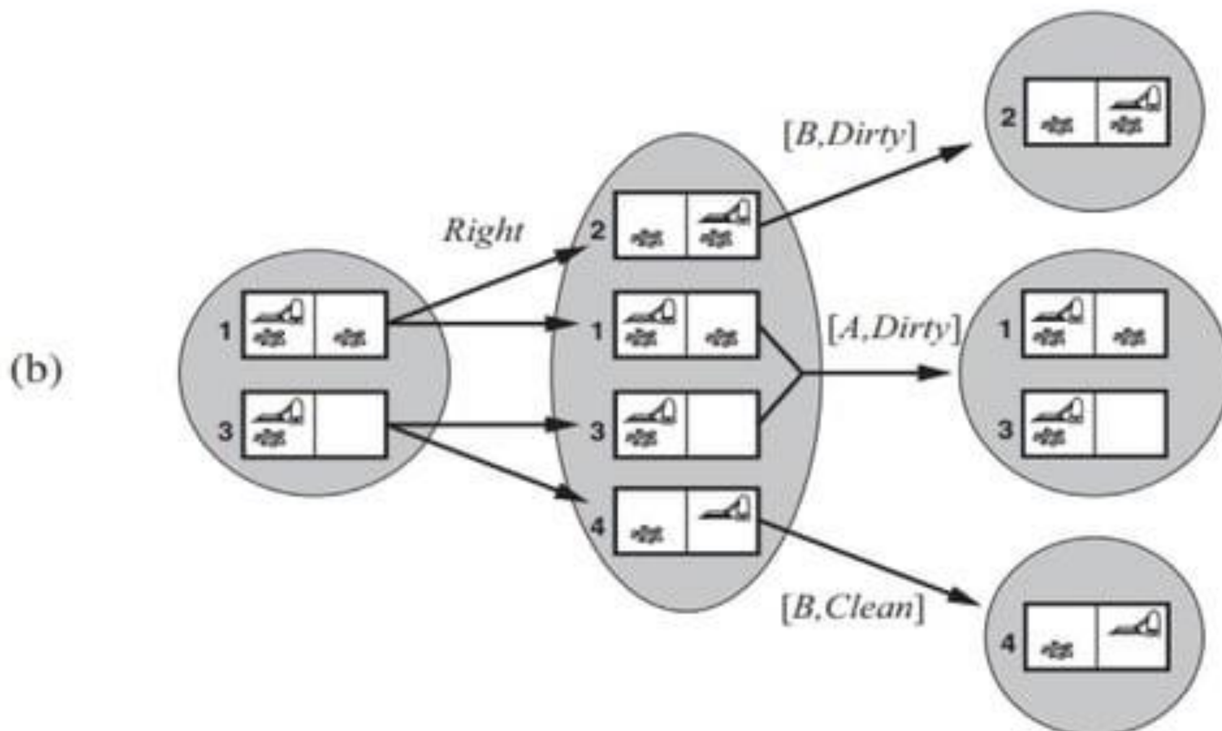


ii. Searching with observations:

(a) In the ***deterministic world***, Right is applied in the initial belief state, resulting in a new belief state with two possible physical states; **[B, Dirty] and [B, Clean].**



(b) In the slippery world, Right is applied in the initial belief state, giving a new belief state with four physical states; **[A, Dirty], [B, Dirty], and [B, Clean].**



20. ONLINE SEARCH AGENTS AND UNKNOWN ENVIRONMENTS:

- An online search problem must be solved by an agent executing actions, rather than by pure computation.
- We assume a deterministic and fully observable environment but we stipulate that the agent knows only the following:
    - **ACTIONS(s),** which returns a list of actions allowed in state's;
    - The step-cost function **c(s, a, s')**—note that this cannot be used until the agent knows that **s'** is the outcome; and
    - **GOAL-TEST(s).**

Considered "**offline**" search problem

- Works "offline" à searches to compute a whole plan...before ever acting
- Even with percepts à gets HUGE fast in real world
- Lots of possible actions, lots of possible percepts...plus non-det.

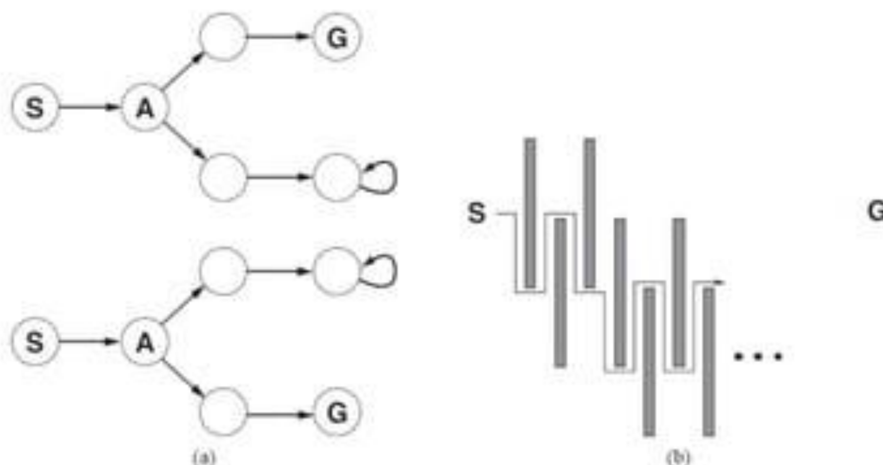
Online search

- **Idea:** Search as you go. Interleave search + action
- **Problem :** actual percepts prune huge subtree of search space @ each move
- **Condition:** plan ahead less à don't foresee problems
- **Best case** = wasted effort. Reverse actions and re-plan
- **Worst case:** not reversible actions. Stuck!

Online search only possible method in some worlds

- Agent doesn't know what states exist (exploration problem)
- Agent doesn't know what effect actions have (discovery learning)
- Possibly: do online search for a while
    - until learn enough to do more predictive search

Example:



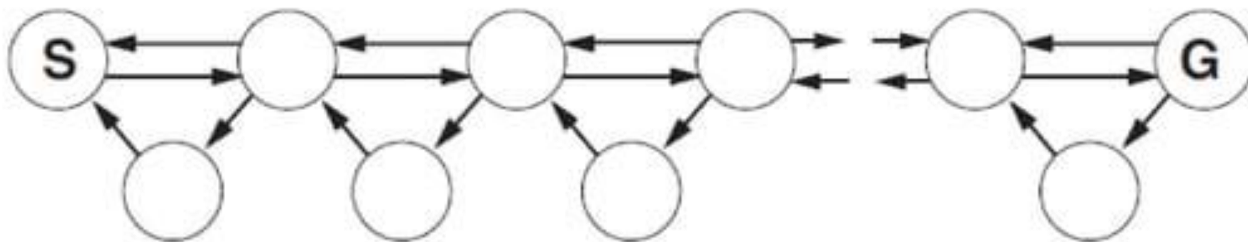
The nature of active online search:

Executing online search = algorithm for planning/acting

- Very different than offline search algorithm!
- **Offline:** search virtually for a plan in constructed search space...
    - Can use any search algorithm, e.g., A* with strong h(n)
    - A* can expand any node it wants on the frontier (jump around)
- **Online agent:** Agent literally is in some place!
    - Agent is at one node (state) on frontier of search tree
    - Can't just jump around to other states...must plan from current state.
    - (Modified) Depth first algorithms are ideal candidates!
- Heuristic functions remain critical!
    - H (n) tells depth first which of the successors to explore!
    - Admissibility remains relevant too: want to explore likely optimal paths first
    - Real agent = real results. At some point I find the goal
        - Can compare actual path cost to that predicted at each state by H(n)
        - Competitive Ratio: Actual path cost/predicted cost. Lower is better.
        - Could also be basis for developing (learning!) improved H (n) over time.

Online local search for agents:

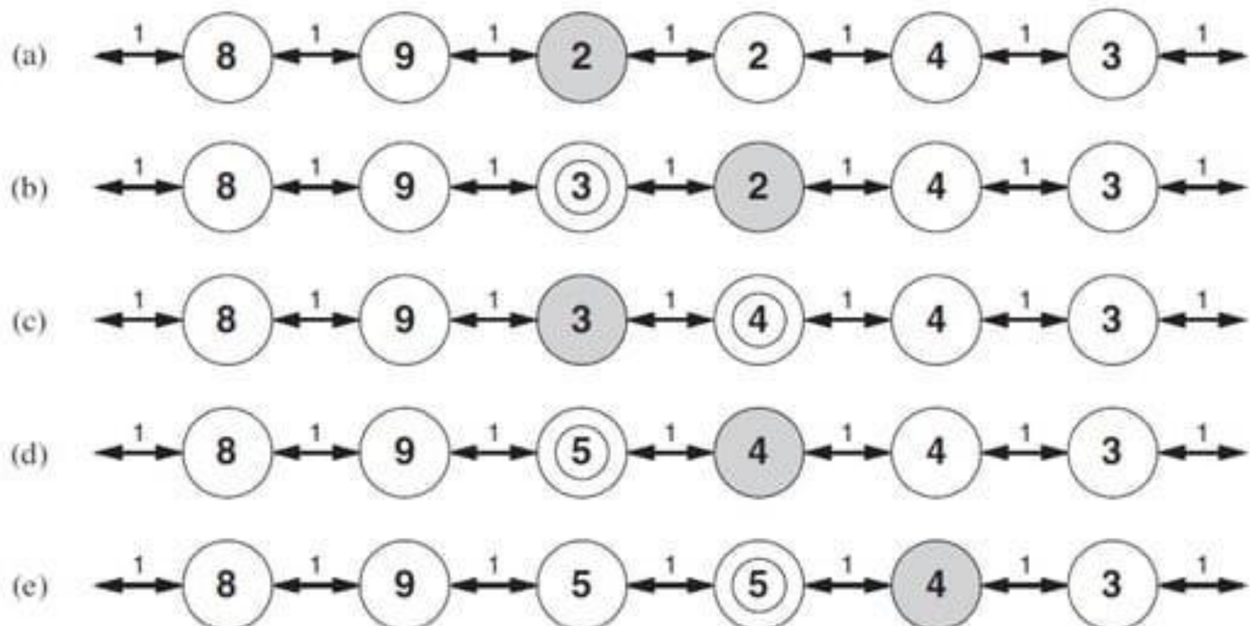
- Hill-climbing is already an online search algorithm but stops at local optimum. **How about randomization?**
    - Cannot do random restart (you can't teleport a robot)
    - **How about just a random walk instead of hill-climbing?**



- - Can be very bad (two ways back for every way forward above)
- Let's augment HC with memory
    - Learning real-time **A* (LRTA*)**
        - Updates cost estimates, g(s), for the state it leaves
        - Likes unexplored states
            - **f(s) = h(s) not g(s) + h(s) for unexplored states**

LRTA* Example:

- We are in shaded state



LRTA* algorithm:

function LRTA*-AGENT(s') **returns** an action
  inputs: s', a percept that identifies the current state
  persistent: *result*, a table, indexed by state and action, initially empty
    H, a table of cost estimates indexed by state, initially empty
    s, a, the previous state and action, initially null

  if GOAL-TEST(s') **then return** *stop*
  if s' is a new state (not in H) **then** $H[s'] \leftarrow h(s')$
  if s is not null
    $result[s, a] \leftarrow s'$
    $H[s] \leftarrow \min_{b \in \text{ACTIONS}(s)} \text{LRTA*-COST}(s, b, result[s, b], H)$
  $a \leftarrow$ an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, $result[s', b]$, H)
  $s \leftarrow s'$
  return a

function LRTA*-COST(s, a, s', H) **returns** a cost estimate
  if s' is undefined **then return** $h(s)$
  else return $c(s, a, s') + H[s']$