
ARTIFICIAL INTELLIGENCE

V SEMESTER

Dr.Rajeshwari.J
Associate Professor
ISE dept
Dayananda Sagar College of Engineering
Bangalore

1.1 Introduction to AI

1.1.1 What is artificial intelligence?

Artificial Intelligence is the branch of computer science concerned with making computers behave like humans.

Major AI textbooks define artificial intelligence as "the study and design of intelligent agents," where an **intelligent agent** is a system that **perceives** its **environment** and **takes actions** which maximize its chances of success. **John McCarthy**, who coined the term in 1956, defines it as "the science and engineering of making intelligent machines, especially intelligent computer programs."

The definitions of AI according to some text books are categorized into four approaches and are summarized in the table below :

Systems that think like humans	**Systems that think rationally**
Systems that act like humans	**Systems that act rationally**

	Humanly	Rationally
Thinking	Thinking humanly-cognitive modeling systems should solve problems the same way humans do.	Thinking rationally-the use of logic. Need to worry about modeling uncertainity and dealing with complexity.
Acting	Acting humanly-the turing test approach	Acting rationally-the study of rational agents: the agents that maximize the expected value of their performance measure given what they currently know.

The four approaches in more detail are as follows :

(a) Acting humanly : The Turing Test approach

- Test proposed by Alan Turing in 1950
- The computer is asked questions by a human interrogator.

The computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or not. Programming a computer to pass ,the computer need to possess the following capabilities :

- **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

To pass the complete Turing Test,the computer will need

- **Computer vision** to perceive the objects,and
- **Robotics** to manipulate objects and move about.

(b)Thinking humanly : The cognitive modeling approach

We need to get inside actual working of the human mind :

(a) through introspection – trying to capture our own thoughts as they go by;
(b) through psychological experiments

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. **When we come to know how human thinks then it is possible to express the theory as computer programs**

(c) Thinking rationally : The "laws of thought approach"

- AI means thinking rationally, i.e., modeling thinking as a logical process, where conclusions are drawn based on some type of symbolic logic.
- Rational thinking is the ability to consider the relevant variables of a situation and to access, organize, and analyze relevant information (e.g., facts, opinions, judgments, and data) to arrive at a sound conclusion. 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; their study initiated a field called **logic.**

(d)Acting rationally : The rational agent approach

Rational behavior: doing the right thing. The right thing: that which is expected to maximize goal achievement, given the available information.

-How can a machine act correctly.

If we design intelligent behaviour in agent, then we can make a machine act reasonbly and correctly .

A **rational agent** is one that acts so as to achieve the best outcome.

1.2 INTELLIGENT AGENTS

1.2.1 Agents and environments

An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and SENSOR acting upon that environment through **actuators.** This simple idea is illustrated in Figure 1.2.

- A human agent has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for actuators.
- A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators.
- A software agent receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

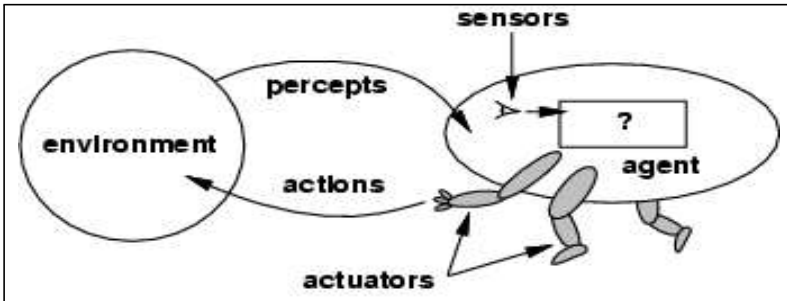


Figure 1.2 Agents interact with environments through sensors and actuators.

Percept

We use the term **percept** to refer to the agent's perceptual inputs at any given instant.

Percept Sequence

An agent's **percept sequence** is the complete history of everything the agent has ever perceived.

Agent function

Mathematically speaking, we say that an agent's behavior is described by the **agent function** that maps any given percept sequence to an action.

$$f : \mathcal{P}^* \rightarrow \mathcal{A}$$

Agent program

Internally, The agent function for an artificial agent will be implemented by an **agent program.** It is important to keep these two ideas distinct. The agent function is an abstract mathematical description; the agent program is a concrete implementation, running on the agent architecture.

To illustrate these ideas, we will use a very simple example-the vacuum-cleaner world shown in Figure 1.3. This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if

the current square is dirty, then suck, otherwise move to the other square. A partial tabulation of this agent function is shown in Figure 1.4.

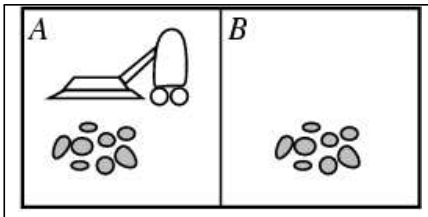


Figure 1.3 A vacuum-cleaner world with just two locations.

Agent function

Percept Sequence	**Action**
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
…	

Figure 1.4 Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 1.3.

agent program

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

Agent Function for an artificial agent will be implemented by an agent program

The Concept of Rationality

Rational Agent

A **rational agent** is one that does the right thing-conceptually speaking, every entry in the table for the agent function is filled out correctly. Obviously, doing the right thing is better than doing the wrong thing. The right action is the one that will cause the agent to be most successful.

Performance measures

A performance measure embodies the **criterion for success** of an agent's behavior. When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives. This sequence of actions causes the environment to go through a sequence of states. If the sequence is desirable, then the agent has performed well.

Rationality

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

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

This leads to a **definition of a rational agent:**
For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

Omniscience, learning, and autonomy

An **omniscient(perfect) agent** knows the *actual* outcome of its actions and can act accordingly; but omniscience is impossible in reality.

Doing actions in order to modify future percepts-sometimes called **information gathering-**is an important part of rationality.

Our definition requires a rational agent not only to gather information, but also to **learn** as much as possible from what it perceives.

To the extent that an agent relies on the prior knowledge of its designer rather than on its own percepts, we say that the agent lacks autonomy. A rational agent should be **autonomous-**it should learn what it can to compensate for partial or incorrect prior knowledge.

Task environments

We must think about **task environments,** which are essentially the "**problems**" to which rational agents are the "**solutions**."

Specifying the task environment

The rationality of the simple vacuum-cleaner agent, needs specification of

- the performance measure
- the environment
- the agent's actuators and sensors.

PEAS

All these are grouped together under the heading of the **task environment.**
We call this the **PEAS** (Performance, Environment, Actuators, Sensors) description.
In designing an agent, the first step must always be to specify the task environment as fully as possible.

Agent Type	Performance Measure	Environments	Actuators	Sensors
Taxi driver	Safe: fast, legal, comfortable trip, maximize profits	Roads,other traffic,pedestrians, customers	Steering,accelerator, brake, Signal,horn,display	Cameras,sonar, Speedometer,GPS, Odometer,engine sensors,keyboards,

				accelerometer

Figure 1.5 PEAS description of the task environment for an automated taxi.

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

Figure 1.6 Examples of agent types and their PEAS descriptions.

Properties of task environments

- Fully observable vs. partially observable
- Deterministic vs. stochastic
- Episodic vs. sequential
- Static vs. dynamic
- Discrete vs. continuous
- Single agent vs. multiagent

Fully observable vs. **partially observable.**

When an agent's sensors allow access to complete state of the environment at each point of time, then the task environment is fully observable, whereas, if the agent does not have complete and relevant information of the environment, then the task environment is partially observable.

Example: In the Checker Game, the agent observes the environment completely while in Poker Game, the agent partially observes the environment because it cannot see the cards of the other agent.

Deterministic vs. **stochastic.**

If the agent's current state and action completely determine the next state of the environment, then the environment is deterministic whereas if the next state cannot be determined from the current state and action, then the environment is Stochastic.

Example: Image analysis – Deterministic
Taxi driving – Stochastic (cannot determine the traffic behavior)

Episodic vs. **sequential**

If the agent's episodes are divided into atomic episodes and the next episode does not depend on the previous state actions, then the environment is episodic, whereas, if current actions may affect the future decision, such environment is sequential.

In **sequential environments**, on the other hand, the current decision could affect all future decisions. Chess and taxi driving are sequential:
Example: Part-picking robot – Episodic
Chess playing – Sequential

Static vs. Dynamic

If the environment changes with time, such an environment is dynamic; otherwise, the environment is static.

Example: Crosswords Puzzles have a static environment while the Physical world has a dynamic environment

Discrete vs. **continuous.**

Static vs. Dynamic

If the environment changes with time, such an environment is dynamic; otherwise, the environment is static.

Example: Crosswords Puzzles have a static environment while the Physical world has a dynamic environment

Single-agent vs. Multiagent

When a single agent works to achieve a goal, it is known as Single-agent, whereas when two or more agents work together to achieve a goal, they are known as Multiagents.

Example: Playing a crossword puzzle – single agent
Playing chess –multiagent (requires two agents)

Figure 1.7 lists the properties of a number of familiar environments.

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

Figure 1.7 Examples of task environments and their characteristics.

The Structure of Agents:

Agent programs

agent = architecture + program

The agent programs all have the same skeleton: they take the current percept as input from the sensors and return an action to the actuators. Notice the difference between the **agent program**, which takes the current percept as input, and the **agent function**, which takes the entire percept history. The agent program takes just the current percept as input because nothing more is available from the environment; if the agent's actions depend on the entire percept sequence, the agent will have to remember the percepts.

Function TABLE-DRIVEN_AGENT(*percept*) **returns** an action

static: *percepts*, a sequence initially empty
table, a table of actions, indexed by percept sequence

append *percept* to the end of *percepts*
action LOOKUP(*percepts*, *table*)
return *action*

Figure 1.8 The TABLE-DRIVEN-AGENT program is invoked for each new percept and returns **an** action each time.

Drawbacks:

- **Table lookup** of percept-action pairs defining all possible condition-action rules necessary to interact in an environment
- **Problems**
  - Too big to generate and to store (Chess has about 10^120 states, for example)
  - No knowledge of non-perceptual parts of the current state
  - Not adaptive to changes in the environment; requires entire table to be updated if changes occur
  - Looping: Can't make actions conditional

- Take a long time to build the table
- No autonomy
- Even with learning, need a long time to learn the table entries

Some Agent Types

- **Table-driven agents**
  - use a percept sequence/action table in memory to find the next action. They are implemented by a (large) **lookup table**.
- **Simple reflex agents**
  - are based on **condition-action rules**, implemented with an appropriate production system. They are stateless devices which do not have memory of past world states.
- **Agents with memory**
  - have **internal state**, which is used to keep track of past states of the world.
- **Agents with goals**
  - are agents that, in addition to state information, have **goal information** that describes desirable situations. Agents of this kind take future events into consideration.
- **Utility-based agents**
  - base their decisions on **classic axiomatic utility theory** in order to act rationally.

Simple Reflex Agent

The simplest kind of agent is the **simple reflex agent.** These agents select actions on the basis of the *current* percept, ignoring the rest of the percept history. For example, the vacuum agent whose agent function is tabulated in Figure 1.10 is a simple reflex agent, because its decision is based only on the current location and on whether that contains dirt.

- Select action on the basis of *only the current* percept.
  E.g. the vacuum-agent
- Large reduction in possible percept/action situations(next page).
- Implemented through *condition-action rules*
  If dirty then suck

A Simple Reflex Agent: Schema

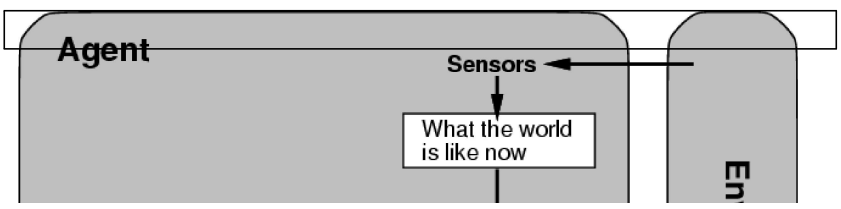


Figure 1.9 Schematic diagram of a simple reflex agent.

function SIMPLE-REFLEX-AGENT(*percept*) **returns** an action

static: *rules*, a set of condition-action rules
state INTERPRET-INPUT(*percept*)
rule RULE-MATCH(*state*, *rule*)
action RULE-ACTION[*rule*]
return *action*

Figure 1.10 A simple reflex agent. It acts according to a rule whose condition matches the current state, as defined by the percept.

function REFLEX-VACUUM-AGENT ([*location, status*]) return an action
if *status* == *Dirty* then return *Suck*
else if *location* == *A* then return *Right*
else if *location* == *B* then return *Left*

Figure 1.11 The agent program for a simple reflex agent in the two-state vacuum environment. This program implements the agent function tabulated in the figure 1.4.

- ❖ **Characteristics**
- ○ Only works if the environment is fully observable.
- ○ Lacking history, easily get stuck in infinite loops
- ○ One solution is to randomize actions

Model-based reflex agents

The most effective way to handle **partial observability** is for the agent to *keep track of the part of the world it can't see now.* That is, the agent should maintain some sort of **internal state** that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state.

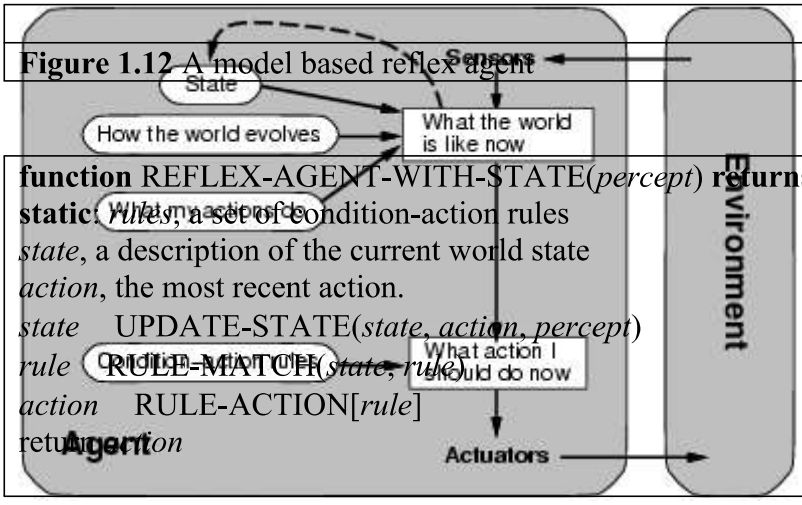


Figure 1.12 A model based reflex agent

function REFLEX-AGENT-WITH-STATE(*percept*) **returns** an action
static: *rules*, a set of condition-action rules
state, a description of the current world state
action, the most recent action.
state UPDATE-STATE(*state*, *action*, *percept*)
rule RULE-MATCH(*state*, *rule*)
action RULE-ACTION[*rule*]
return *action*

Figure 1.13 Model based reflex agent. It keeps track of the current state of the world using an internal model. It then chooses an action in the same way as the reflex agent.

Goal-based agents

It is not sufficient to have the current state information unless the goal is not decided. Therefore, a goal-based agent selects a way among multiple possibilities that helps it to reach its goal. Searching and planning should be done to reach the goal.

Knowing about the current state of the environment is not always enough to decide what to do. For example, at a road junction, the taxi can turn left, turn right, or go straight on. The correct decision depends on where the taxi is trying to get to.

Figure 1.13 shows the goal-based agent's structure.

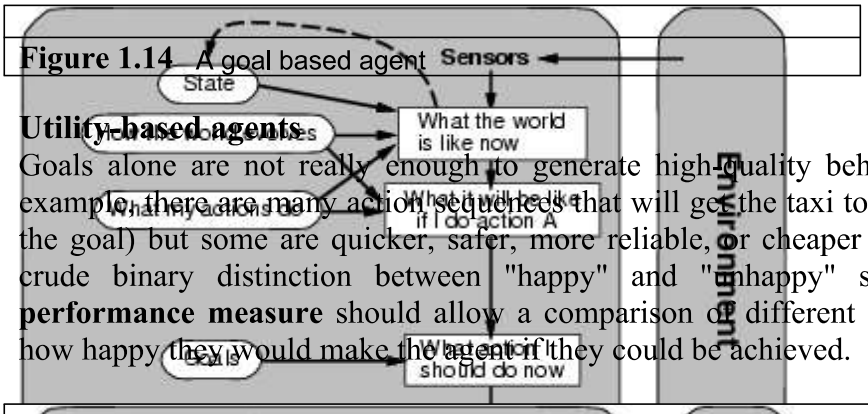


Figure 1.14 A goal based agent

Utility-based agents

Goals alone are not really enough to generate high-quality behavior in most environments. For example, there are many action sequences that will get the taxi to its destination (thereby achieving the goal) but some are quicker, safer, more reliable, or cheaper than others. Goals just provide a crude binary distinction between "happy" and "unhappy" states, whereas a more general **performance measure** should allow a comparison of different world states according to exactly how happy they would make the agent if they could be achieved.

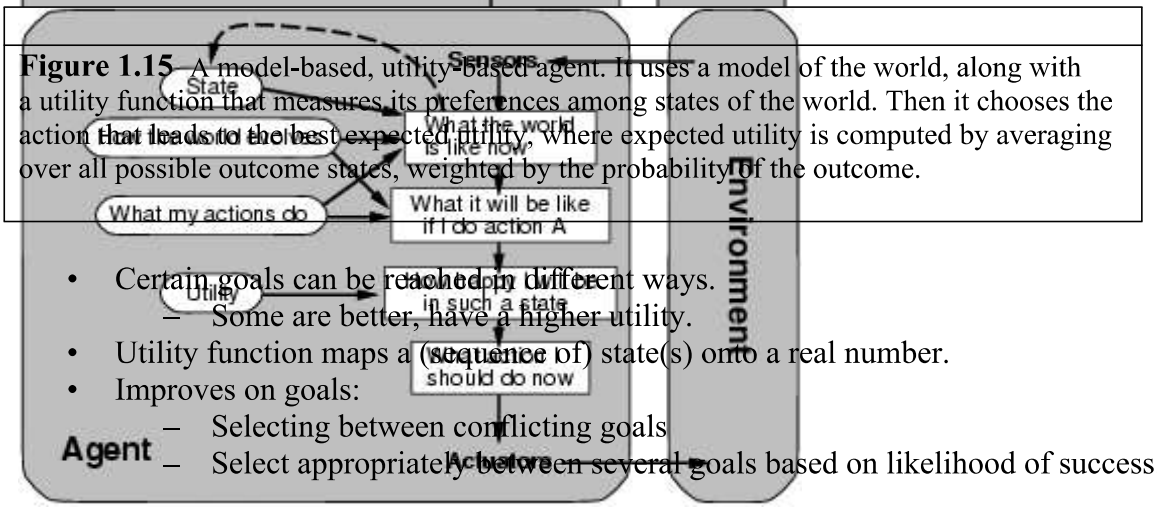


Figure 1.15 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

- Certain goals can be reached in different ways.
  - Some are better, have a higher utility.
- Utility function maps a (sequence of) state(s) onto a real number.
- Improves on goals:
  - Selecting between conflicting goals
  - Select appropriately between several goals based on likelihood of success.

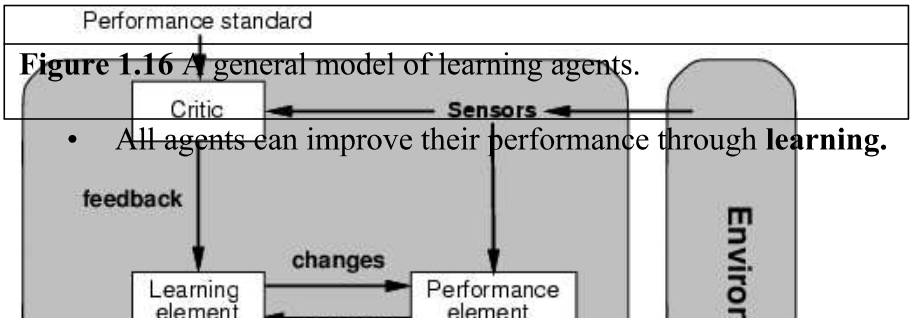


Figure 1.16 A general model of learning agents.

- All agents can improve their performance through **learning.**

A learning agent can be divided into four conceptual components, as shown in Figure 1.15 The most important distinction is between the **learning element**, which is responsible for making improvements, and the **performance element**, which is responsible for selecting external actions. The performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on actions. The learning element uses feedback from the **critic** on how the agent is doing and determines how the performance element should be modified to do better in the future.

The last component of the learning agent is the **problem generator**. It is responsible for suggesting actions that will lead to new and **informative experiences**. But if the agent is willing to explore a little, it might discover much better actions for the long run. The problem generator's job is to suggest these **exploratory actions**. This is what scientists do when they carry out experiments.

Summary: Intelligent Agents

- An **agent** perceives and acts in an environment, has an architecture, and is implemented by an agent program.
- Task environment – **PEAS (P**erformance**, E**nvironment, **A**ctuators, **S**ensors**)**
- The most challenging environments are inaccessible, nondeterministic, dynamic, and continuous.
- An **ideal agent** always chooses the action which maximizes its expected performance, given its percept sequence so far.
- An **agent program** maps from percept to action and updates internal state.
  - **Reflex agents** respond immediately to percepts.
    - simple reflex agents
    - model-based reflex agents
  - **Goal-based agents** act in order to achieve their goal(s).
  - **Utility-based agents** maximize their own utility function.
- All agents can improve their performance through **learning.**

1.3.1 Problem Solving by Search

An important aspect of intelligence is ***goal-based*** problem solving.
The **solution** of many **problems** can be described by finding a **sequence of actions** that lead to a desirable **goal.** Each action changes the ***state*** and the aim is to find the sequence of actions and states that lead from the initial (start) state to a final (goal) state.

A well-defined problem can be described by:

- **Initial state**
- **Operator or successor function** - for any state x returns s(x), the set of states reachable from x with one action
- **State space** - all states reachable from initial by any sequence of actions
- **Path** - sequence through state space
- **Path cost** - function that assigns a cost to a path. Cost of a path is the sum of costs of individual actions along the path
- **Goal test** - test to determine if at goal state

What is Search?

Search is the systematic examination of **states** to find path from the **start/root state** to the **goal state.**

The set of possible states, together with *operators* defining their connectivity constitute the *search space*.
The output of a search algorithm is a solution, that is, a path from the initial state to a state that satisfies the goal test.

Problem-solving agents

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*

A Problem solving agent is a **goal-based** agent. It decide what to do by finding sequence of actions that lead to desirable states. The agent can adopt a goal and aim at satisfying it.
To illustrate the agent's behavior, let us take an example where our agent is in the city of Arad,which is in Romania. The agent has to adopt a **goal** of getting to Bucharest.

Steps performed by Problem-solving agent

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

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.

The **search algorithm** takes a **problem** as **input** and returns a **solution** in the form of **action sequence.** Once a solution is found, the **execution phase** consists of carrying out the recommended action..
Figure 1.18 shows a simple "formulate, search, execute" design for the agent. Once solution has been executed,the agent will formulate a new goal.

```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
inputs : percept, a percept
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 do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH( problem)
action ← FIRST(seq);
 seq ←REST(seq)
return action
```

Figure 1.18 A Simple problem solving agent. It first formulates a **goal** and a **problem**, searches for a sequence of actions that would solve a problem, and executes the actions one at a time.

- The agent design assumes the Environment is
    - **Static**: The entire process carried out without paying attention to changes that might be occurring in the environment.
    - **Observable :** The initial state is known and the agent's sensor detects all aspects that are relevant to the choice of action
    - **Discrete :** With respect to the state of the environment and percepts and actions so that alternate courses of action can be takenjhngfbuuuu 69*
    - **Deterministic:** The next state of the environment is completely determined by the current state and the actions executed by the agent. Solutions to the problem are single sequence of actions

An agent carries out its plan with eye closed. This is called an open loop system because ignoring the percepts breaks the loop between the agent and the environment.

1.3.1.1 Well-defined problems and solutions

A **problem** can be formally defined by **four components**:

- The **initial state** that the agent starts in. The initial state for our agent of example problem is described by *In(Arad)*
- A **Successor Function** returns the possible **actions** available to the agent. Given a state x, SUCCESSOR-FN(x) returns a set of {action, successor} ordered pairs where each action is one of the legal actions in state x, and each successor is a state that can be reached from x by applying the action.

For example,from the state In(Arad),the successor function for the Romania problem would return

{ [Go(Sibiu),In(Sibiu)],[Go(Timisoara),In(Timisoara)],[Go(Zerind),In(Zerind)] }

- **State Space** : The set of all states reachable from the initial state. The state space forms a graph in which the nodes are states and the arcs between nodes are actions.
- A **path** in the state space is a sequence of states connected by a sequence of actions.
- The **goal test** determines whether the given state is a goal state.

- A **path cost** function assigns numeric cost to each action. For the Romania problem the cost of path might be its length in kilometers.
- The **step cost** of taking action a to go from state x to state y is denoted by c(x,a,y). The step cost for Romania are shown in figure 1.19. It is assumed that the step costs are non negative.
- A **solution** to the problem is a path from the initial state to a goal state.
- An **optimal solution** has the lowest path cost among all solutions.

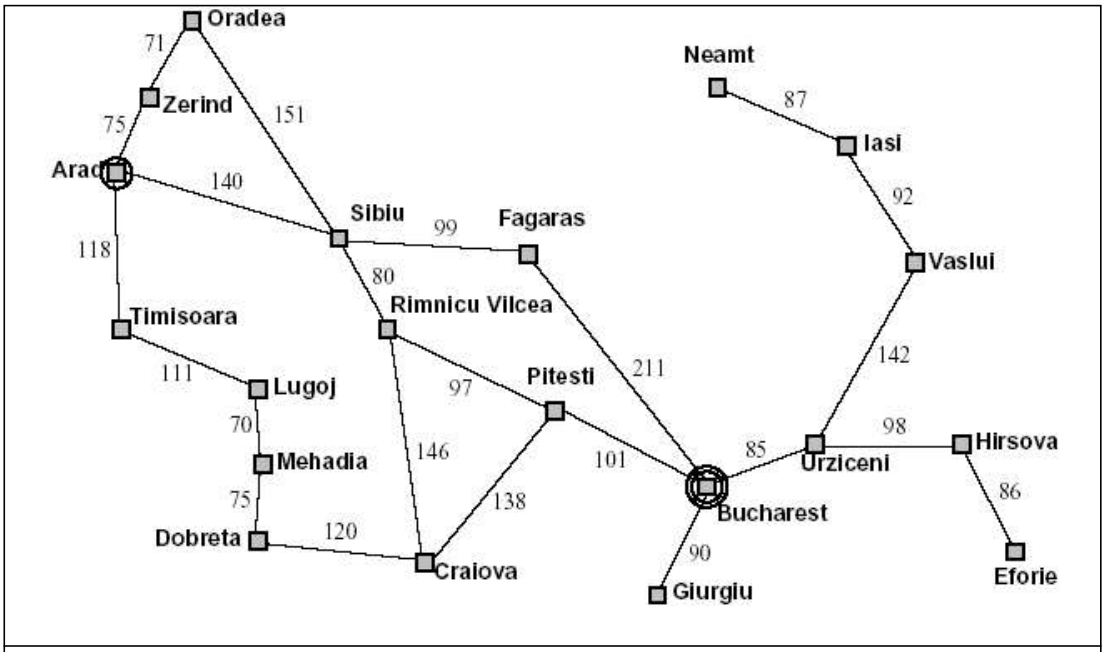


Figure 1.19 A simplified Road Map of part of Romania

1.3.2 EXAMPLE PROBLEMS

The problem solving approach has been applied to a vast array of task environments. Some best known problems are summarized below. They are distinguished as toy or real-world problems

A **toy problem** is intended to illustrate various problem solving methods. It can be easily used by different researchers to compare the performance of algorithms.

A **real world problem** is one whose solutions people actually care about.

1.3.2.1 TOY PROBLEMS

Vacuum World Example

- **States**: The agent is in one of two locations.,each of which might or might not contain dirt. Thus there are 2 x 2^2 = 8 possible world states.
- **Initial state**: Any state can be designated as initial state.
- **Successor function** : This generates the legal states that results from trying the three actions (left, right, suck). The complete state space is shown in figure 1.20
- **Goal Test** : This tests whether all the squares are clean.

- **Path test** : Each step costs one ,so that the the path cost is the number of steps in the path.

Vacuum World State Space

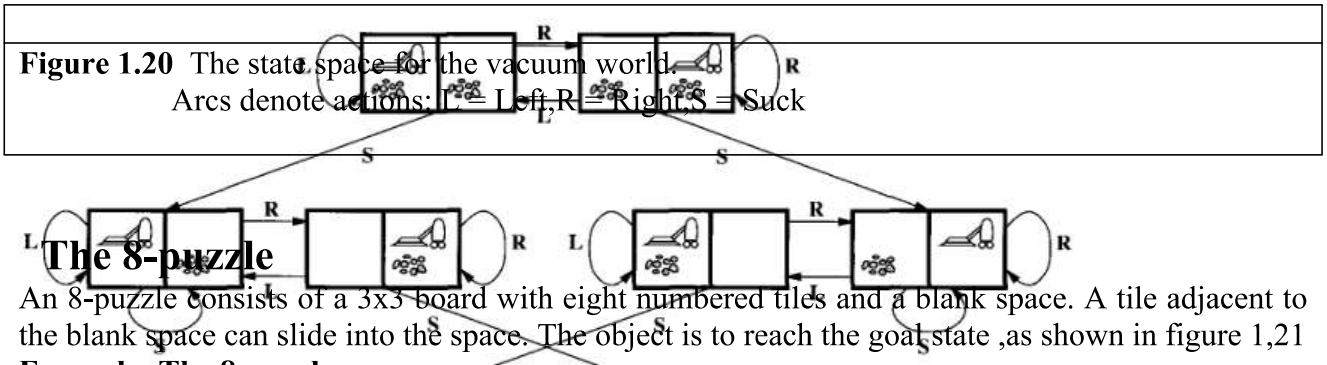


Figure 1.20 The state space for the vacuum world. Arcs denote actions: L = Left,R = Right,S = Suck

The 8-puzzle

An 8-puzzle consists of a 3x3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach the goal state ,as shown in figure 1,21

Example: The 8-puzzle

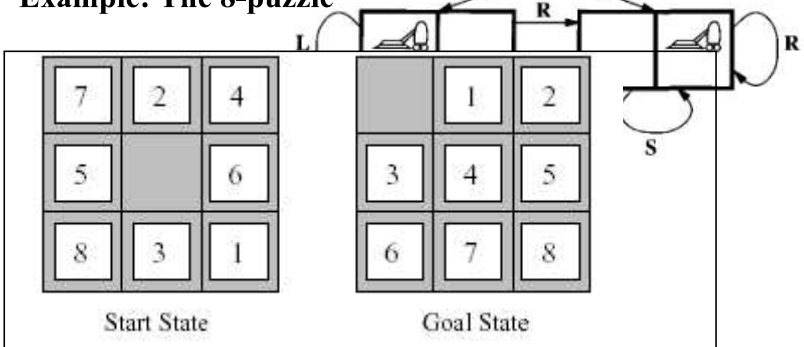


Figure 1.21 A typical instance of 8-puzzle.

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

8-queens problem

The goal of 8-queens problem is to place 8 queens on the chessboard such that no queen attacks any other.(A queen attacks any piece in the same row,column or diagonal).

Figure 1.22 shows an attempted solution that fails: the queen in the right most column is attacked by the queen at the top left.

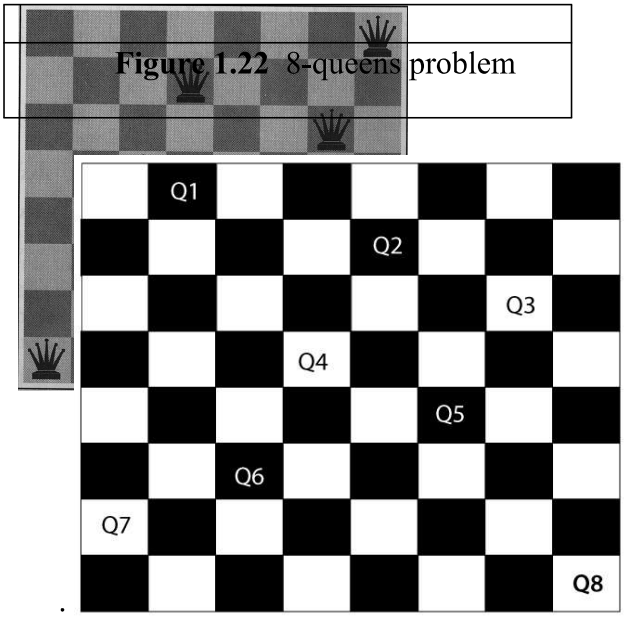


Figure 1.22 8-queens problem

. t 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:

- **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 x 10^{14}** possible sequence to investigate.

- **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.

1.3.2.2 REAL-WORLD PROBLEMS

ROUTE-FINDING PROBLEM

Route-finding problem is defined in terms of specified locations and transitions along links between them. Route-finding algorithms are used in a variety of applications,such as routing in computer networks,military operations planning,and air line travel planning systems.

AIRLINE TRAVEL PROBLEM

The **airline travel problem** is specifies as follows **:**

- **States :** Each is represented by a location(e.g.,an airport) and the current time.
- **Initial state :** This is specified by the problem.

- **Transition model :** This returns the states resulting from taking any scheduled flight(further specified by seat class and location),leaving later than the current time plus the within-airport transit time, from the current airport to another.
- **Goal Test :** Are we at the destination by some prespecified time?
- **Path cost :** This depends upon the monetary cost,waiting time,flight time,customs and immigration procedures,seat quality,time of dat,type of air plane,frequent-flyer mileage awards, and so on.

TOURING PROBLEMS

Touring problems are closely related to route-finding problems,but with an important difference.
Consider for example,the problem,"Visit every city at least once" as shown in Romania map.
As with route-finding the actions correspond to trips between adjacent cities. The state space, however,is quite different.
The initial state would be "In Bucharest; visited{Bucharest}".
A typical intermediate state would be "In Vaslui;visited {Bucharest,Urziceni,Vaslui}".
The goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

THE TRAVELLING SALESPERSON PROBLEM(TSP)

Is a touring problem in which each city must be visited exactly once. The aim is to find the shortest tour.The problem is known to be **NP-hard**. Enormous efforts have been expended to improve the capabilities of TSP algorithms. These algorithms are also used in tasks such as planning movements of **automatic circuit-board drills** and of **stocking machines** on shop floors.

VLSI layout

A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area ,minimize circuit delays,minimize stray capacitances,and maximize manufacturing yield. The layout problem is split into two parts : **cell layout** and **channel routing**.

ROBOT navigation

ROBOT navigation is a generalization of the route-finding problem. Rather than a discrete set of routes,a robot can move in a continuous space with an infinite set of possible actions and states. For a circular Robot moving on a flat surface,the space is essentially two-dimensional.
When the robot has arms and legs or wheels that also must be controlled,the search space becomes multi-dimensional. Advanced techniques are required to make the search space finite.

AUTOMATIC ASSEMBLY SEQUENCING

The example includes assembly of intricate objects such as electric motors. The aim in assembly problems is to find the order in which to assemble the parts of some objects. If the wrong order is choosen,there will be no way to add some part later without undoing somework already done.
Another important assembly problem is protein design,in which the goal is to find a sequence of Amino acids that will be fold into a three-dimensional protein with the right properties to cure some disease.

INTERNET SEARCHING

In recent years there has been increased demand for software robots that perform Internet searching.,looking for answers to questions,for related information,or for shopping deals. The searching techniques consider internet as a graph of nodes(pages) connected by links.

1.3.3 UNINFORMED SEARCH STRATGES

Uninformed Search Strategies have no additional information about states, except the information 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 five uninformed search strategies as given below.

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

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), so it is also called **heuristic search.**

There are following types of informed searches:

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

There are five uninformed search strategies as given below.

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

1.3.4.1 Breadth-first search

It is a simple search strategy where the root node is expanded first, then covering all other successors of the root node, further move to expand the next level nodes and the search continues until the goal node is not found.

BFS expands the shallowest (i.e., not deep) node first using FIFO (First in first out) order. Thus, new nodes (i.e., children of a parent node) remain in the queue and old unexpanded node which are shallower than the new nodes, get expanded first.
In BFS, goal test **(a test to check whether the current state is a goal state or not)** is applied to each node at the time of its generation rather when it is selected for expansion.

BFS Algorithm

- Set a variable **NODE** to the initial state, i.e., the *root node.*
- Set a variable **GOAL** which contains the value of the *goal state.*
- Loop each node by traversing level by level until the goal state is found.

- While performing the looping, start removing the elements from the queue in ***FIFO*** order.
- If the goal state is found, **return goal state** otherwise continue the search.

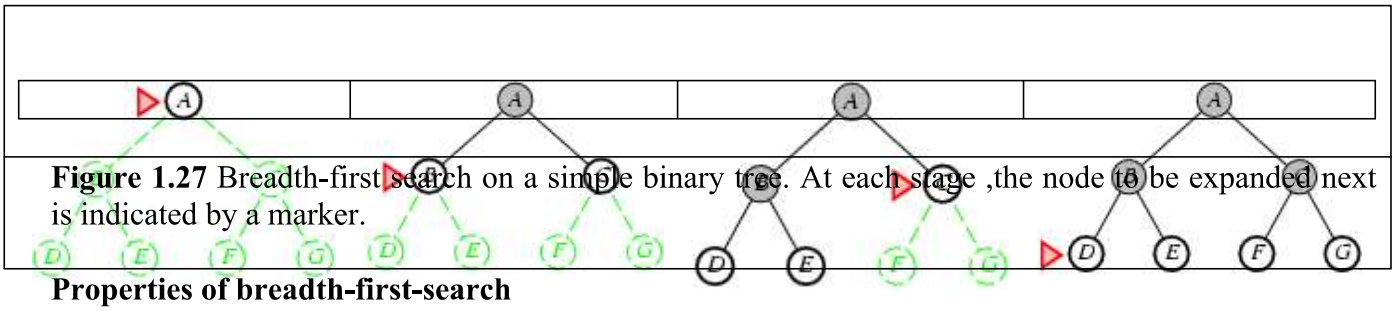


Figure 1.27 Breadth-first search on a simple binary tree. At each stage ,the node to be expanded next is indicated by a marker.

Properties of breadth-first-search

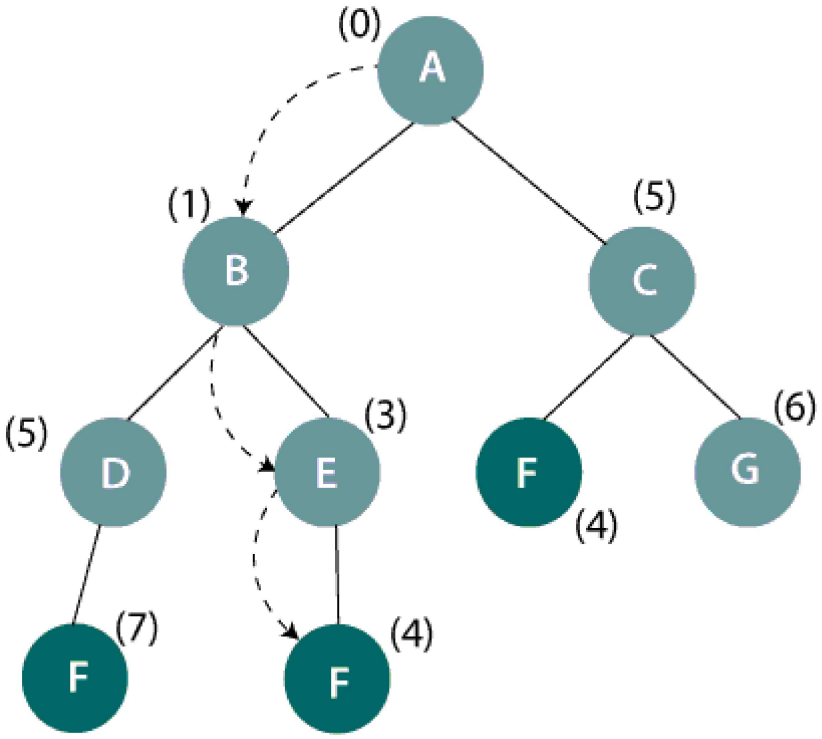
- Complete? Yes (if b is finite)
- Time? $1+b+b^2+b^3+\ldots+b^d + b^{d+1} = O(b^{d+1})$
- Space? $O(b^{d+1})$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)

Space is the bigger problem (more than time)

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

1.3.4.2 UNIFORM-COST SEARCH

Thus, uniform-cost search expands nodes in a sequence of their **optimal path cost** because before exploring any node, it searches the optimal path. Also, the step cost is positive so, paths never get shorter when a new node is added in the search.



In the above figure, it is seen that the goal-state is **F** and start/initial state is **A.** There are three paths available to reach the goal node.
A-B-D-F=13
A-B-E-F=8
A-C-F=9
We need to select an optimal path which may give the lowest total cost g(n). Therefore, **A->B->E->F** gives the optimal path cost i.e., **0+1+3+4=8.**

The performance measure of Uniform-cost search

- **Completeness:** It guarantees to reach the goal state.
- **Optimality:** It gives optimal path cost solution for the search.

2.5.1.3 DEPTH-FIRST-SEARCH

This search strategy explores the deepest node first, then backtracks to explore other nodes. It uses **LIFO (Last in First Out)** order, which is based on the **stack,** in orderto expand the unexpanded nodes in the search tree. The search proceeds to the deepest level of the tree where it has no successors. This search expands nodes till infinity, i.e., the depth of the tree.

DFS Algorithm

- Set a variable **NODE** to the initial state, i.e., the *root node.*
- Set a variable **GOAL** which contains the value of the *goal state.*
- Loop each node by traversing deeply in one direction/path in search of the goal node.
- While performing the looping, start removing the elements from the stack in ***LIFO*** order.
- If the goal state is found, **return goal state** otherwise backtrack to expand nodes in other direction.

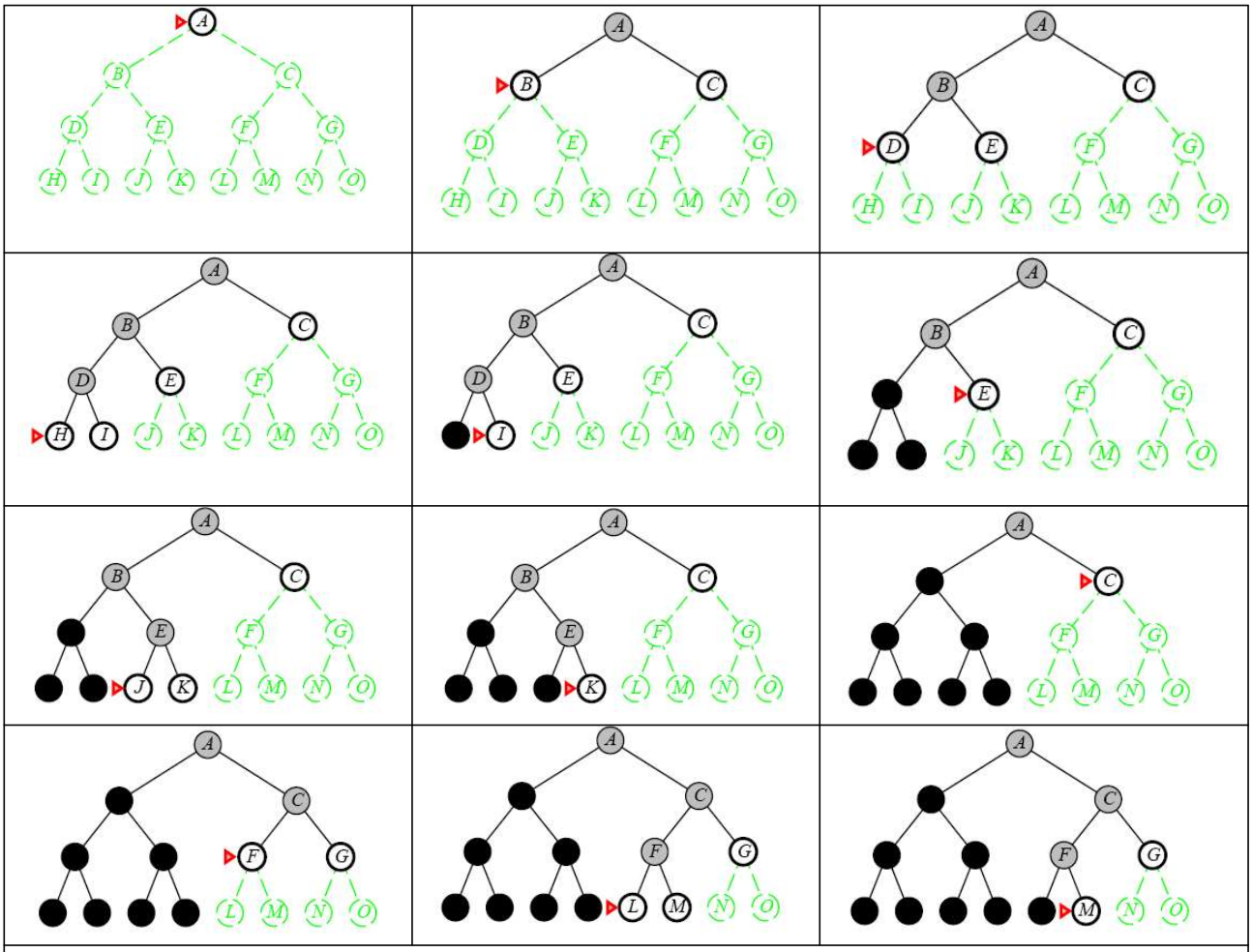


Figure 1.31 Depth-first-search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from the memory;these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue,also known as a stack.

Depth-first-search has very modest memory requirements.It needs to store only a single path from the root to a leaf node,along with the remaining unexpanded sibling nodes for each node on the path. Once the node has been expanded,it can be removed from the memory,as soon as its descendants have been fully explored(Refer Figure 2.12).

The performance measure of DFS

- **Completeness:** DFS does not guarantee to reach the goal state.
- **Optimality:** It does not give an optimal solution as it expands nodes in one direction deeply.
- **Space complexity**: It needs to store only a single path from the root node to the leaf node. Therefore, DFS has **O(bm)** space complexity where b is the **branching factor(i.e., total no. of child nodes, a parent node have)** and m is the **maximum length of any path.**
- **Time complexity**: DFS has **$O(b^m)$** time complexity.

Drawback of Depth-first-search

The drawback of depth-first-search is that it can make a wrong choice and get stuck going down very long(or even infinite) path when a different choice would lead to solution near the root of the search tree. For example ,depth-first-search will explore the entire left subtree even if node C is a goal node.

BACKTRACKING SEARCH

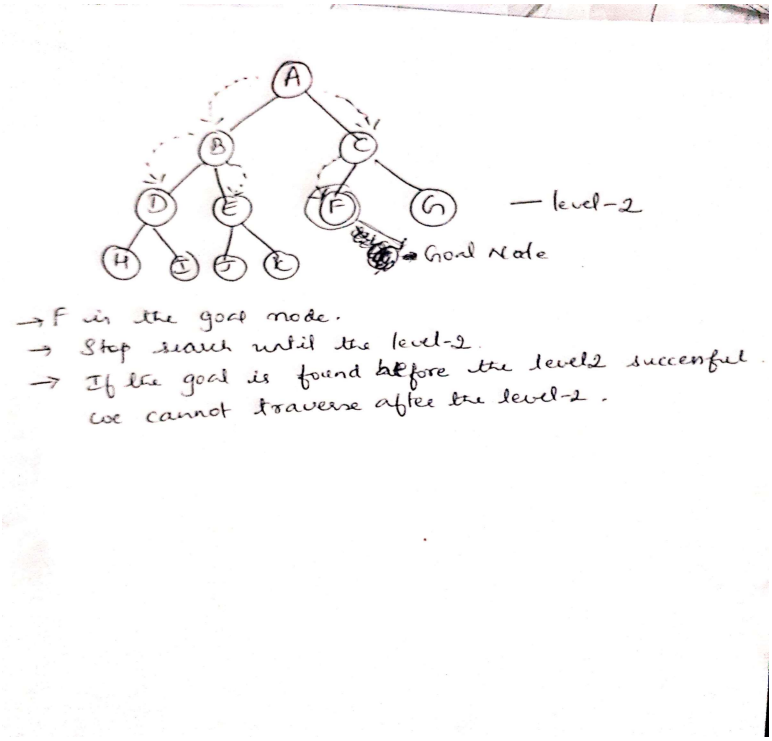
A variant of depth-first search called backtracking search uses less memory and only one successor is generated at a time rather than all successors.; Only O(m) memory is needed rather than O(bm)

1.3.4.4 DEPTH-LIMITED-SEARCH

This search strategy is similar to DFS with a little difference. The difference is that in depth-limited search, we limit the search by imposing a **depth limit *l*** to the depth of the search tree. It does not need to explore till infinity. As a result, the **depth-first search is a special case of depth-limited search.** when the limit l is infinite.

Sometimes, depth limits can be based on knowledge of the problem. For, example, on the map of Romania there are 20 cities. Therefore,we know that if there is a solution.,it must be of length 19 at the longest, So l = 10 is a possible choice. However, it can be shown that any city can be reached from any other city in at most 9 steps. This number known as the **diameter** of the state space,gives us a better depth limit.

It can be noted that the above algorithm can terminate with two kinds of failure :
a)The standard *failure* value indicates no solution;
b)the *cutoff* value indicates no solution within the depth limit.



Depth limited search (limit)
Let fringe be a list containing the initial state
Loop
if fringe is empty return failure
Node Å remove-first (fringe)
if Node is a goal
then return the path from initial state to Node
else if depth of Node = limit return cutoff
else add generated nodes to the front of fringe
End Loop

1.3.4.5 ITERATIVE DEEPENING DEPTH-FIRST SEARCH

Iterative deepening search (or iterative-deepening-depth-first-search) is a general strategy often used in combination with depth-first-search, that finds the better depth limit. It does this by gradually increasing the limit – first 0,then 1,then 2, and so on – until a goal is found. This will occur when the depth limit reaches d,the depth of the shallowest goal node. The algorithm is shown in Figure 2.14.

DFID
until solution found do
DFS with depth cutoff c
c = c+1

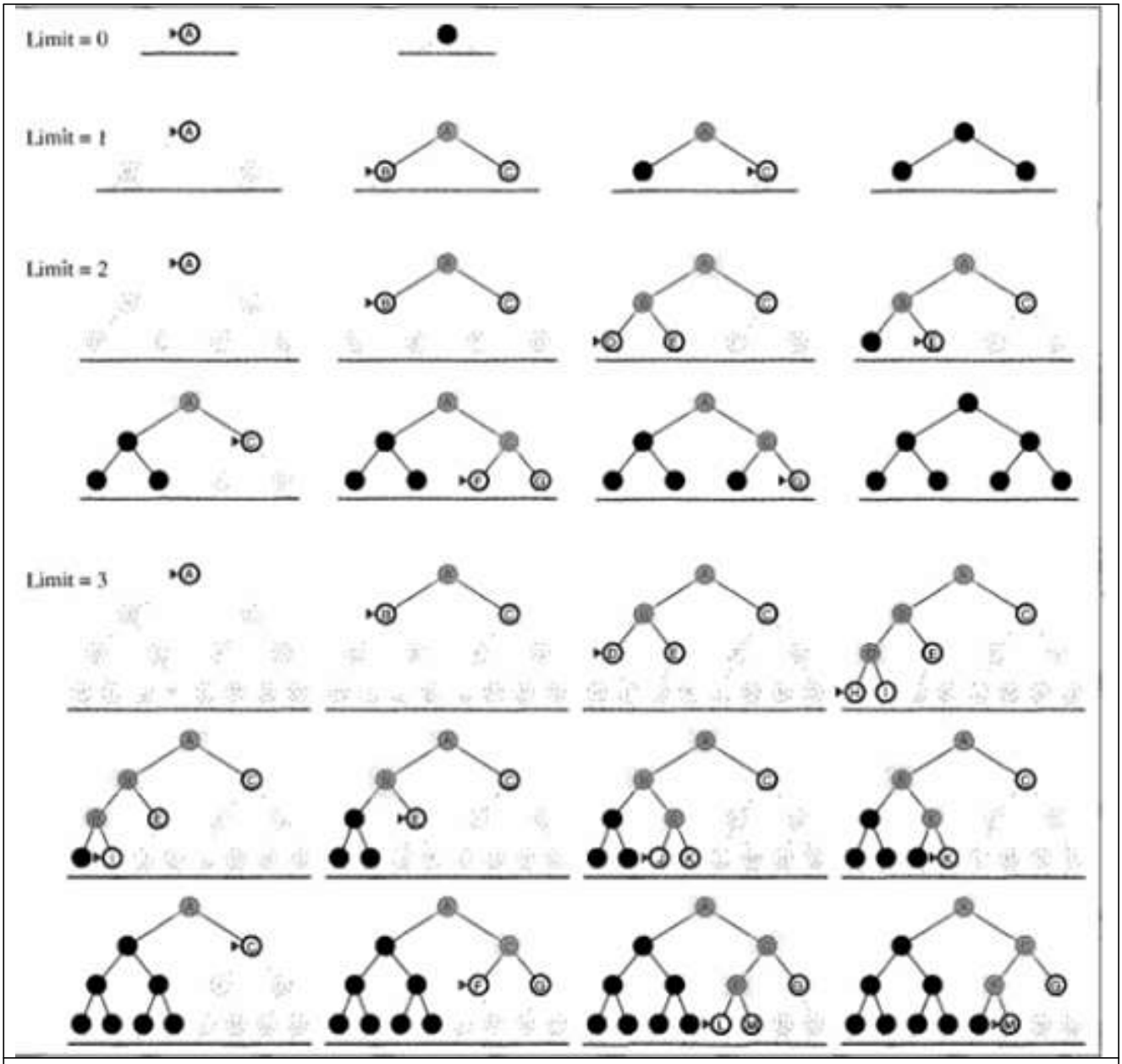
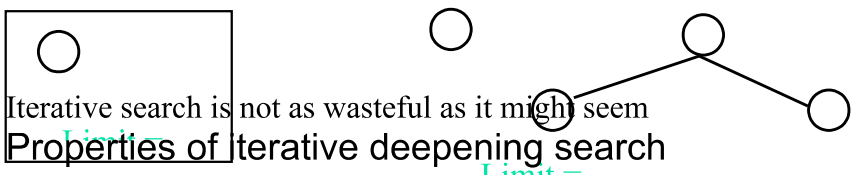


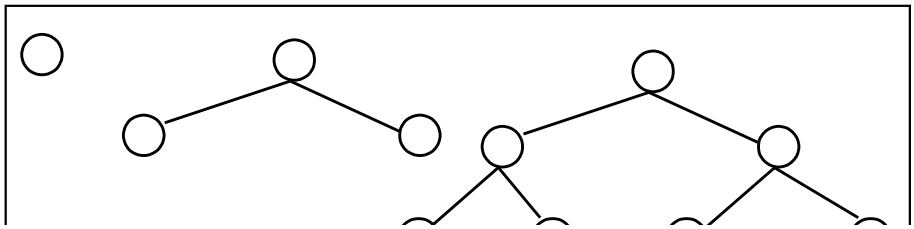
Figure 1.34 Four iterations of iterative deepening search on a binary tree

Iterative search is not as wasteful as it might seem

Figure 1.35 Iterative deepening search



Properties of Iterative deepening search



Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? No, unless step costs are constant
Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

IDS does better because other nodes at depth d are not expanded

BFS can be modified to apply goal test when a node is generated

Figure 1.36

In general, iterative deepening is the prefered uninformed search method when there is a large search space and the depth of solution is not known.

1.3.4.6 Bidirectional Search

The idea behind bidirectional search is to run two simultaneous searches –
one forward from the initial state and
the other backward from the goal,
stopping when the two searches meet in the middle (Figure 1.37).

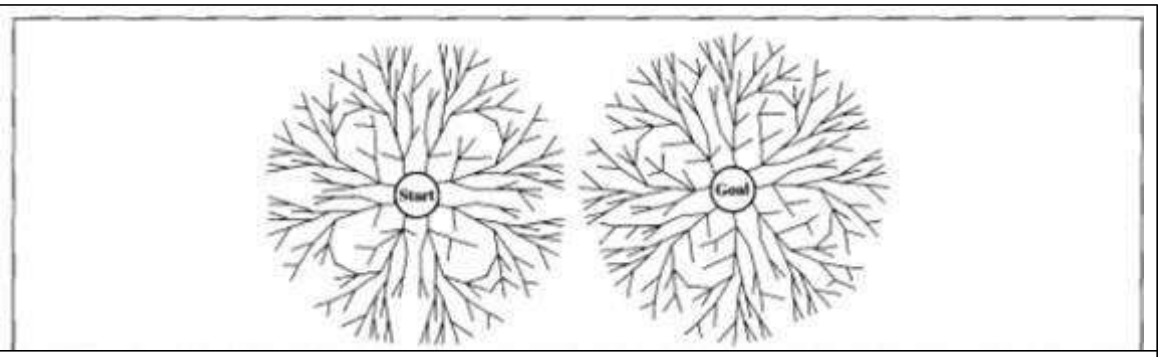


Figure 1.37 A schematic view of a bidirectional search that is about to succeed,when a Branch from the Start node meets a Branch from the goal node.

Informed Search/Heuristic Search:

Informed Search : We have seen that uninformed search methods that systematically explore the state space and find the goal. They are inefficient in most cases. Informed search methods use problem specific knowledge, and may be more efficient.

Heuristics Heuristic means “rule of thumb”. “Heuristics are criteria, methods or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal”. In heuristic search or informed search, heuristics are used to identify the most promising search path.

Example of Heuristic Function :**A heuristic function at a node n is an estimate of the optimum cost from the current node to a goal. It is denoted by h(n).**

h(n) = estimated cost of the cheapest path from node n to a goal node

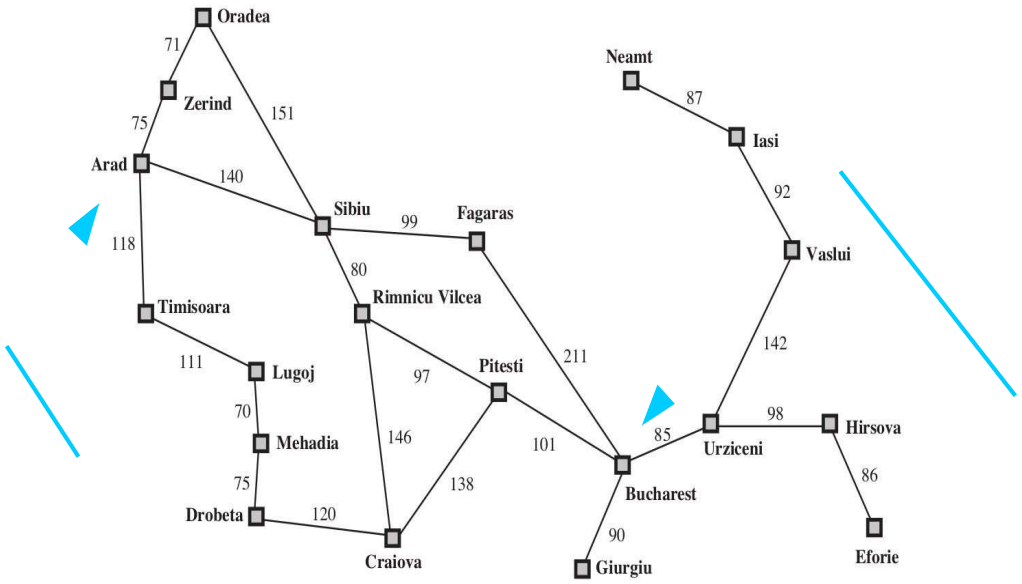
Best-first Search (Greedy search)

A best-first search is a general approach of informed search. Here, a node is selected for expansion based on an **evaluation function f(n)**, **where f(n) interprets the cost estimate value**. The evaluation function expands that node first, which has the lowest cost. A component of f(n) is **h(n)** which carries the additional information required for the search algorithm, i.e.,

h(n)= estimated cost of the cheapest path from the current node n to the goal node.

Note: If the current node n is a goal node, the value of h(n) will be 0.

Best-first search is known as a greedy search because it always tries to explore the node which is nearest to the goal node and selects that path, which gives a quick solution. Thus, it evaluates nodes with the help of the heuristic function, i.e., **f(n)=h(n).**



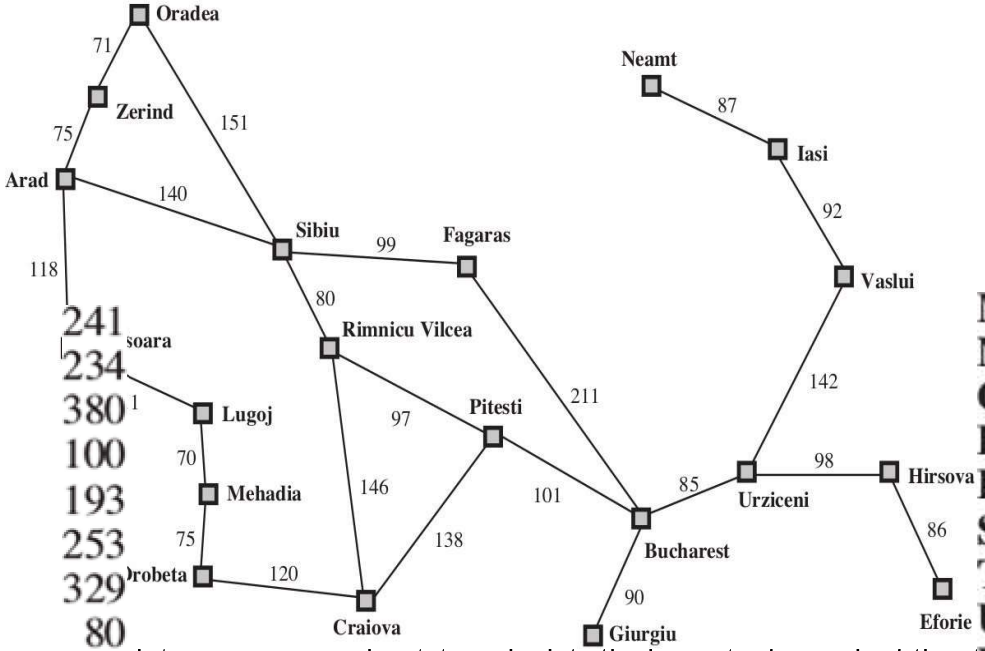
It is calculated based on Straight line distance

Is it possible to use additional information to decide which direction to search in?

YES

Instead of searching in all directions, let's bias search in the direction of the goal

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244



Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

ic)

ch time you expand a state, calculate the heuristic for each of the st e

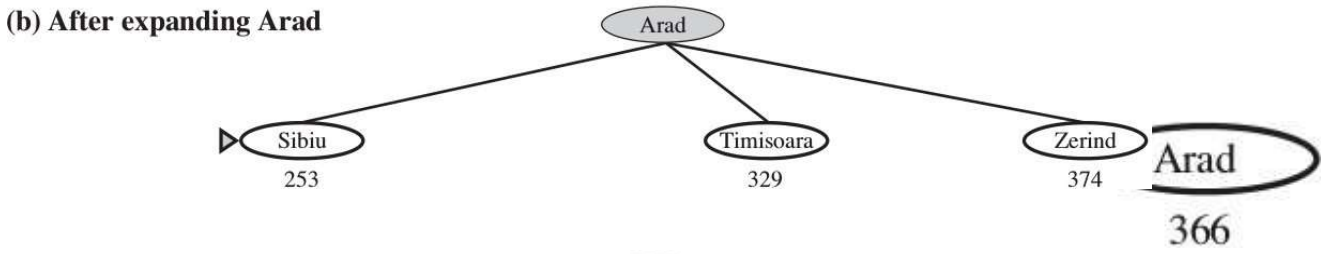
e

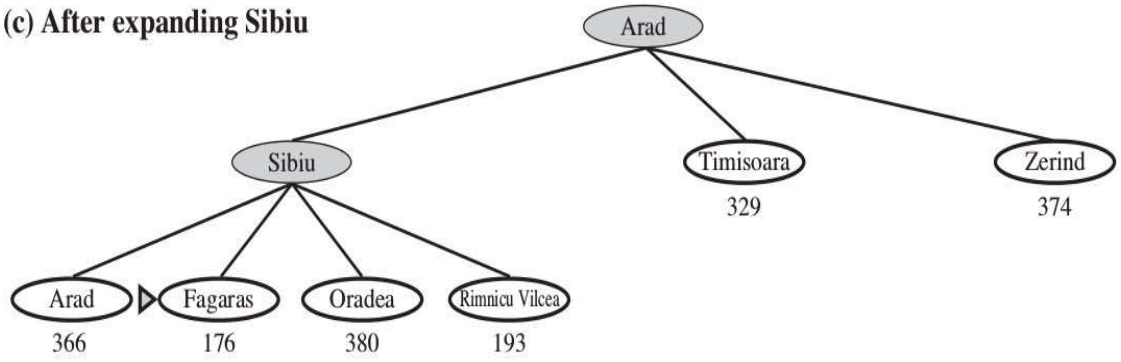
is like a guess about how far the state is from the goal)

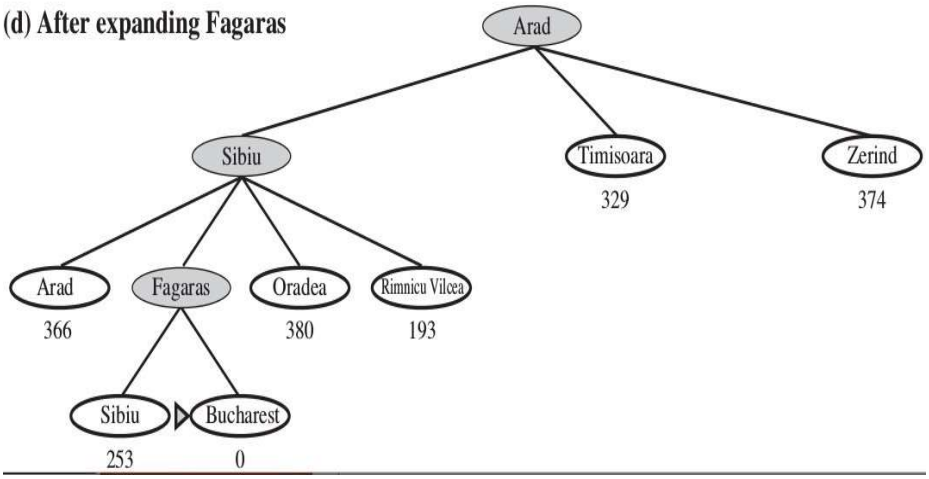
d)on each step, choose to expand the state with the lowest heuristic value.

The performance measure of Best-first search Algorithm:

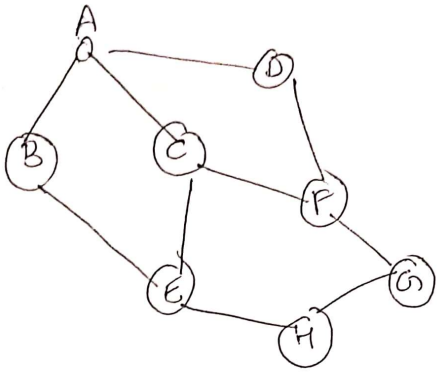
- **Completeness:** Best-first search is incomplete even in finite state space.
- **Optimality:** It does not provide an optimal solution.
- **Time and Space complexity:** It has **$O(b^m)$** worst time and space complexity, where **m is the maximum depth of the search tree.**







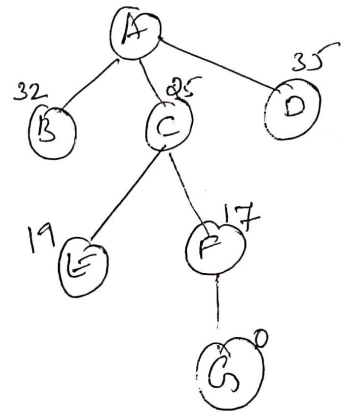
PATH IS -Arad-Sibiu-Fagars-Bucharest



Heuristic Value

Straight line distance

A → G = 40
B → G = 32
C → G = 25
D → G = 35
E → G = 19
F → G = 17
H → G = 10
G → G = 0



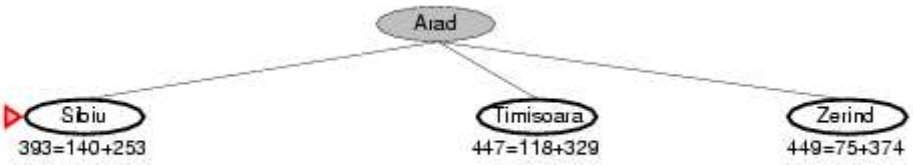
A → C → F → G

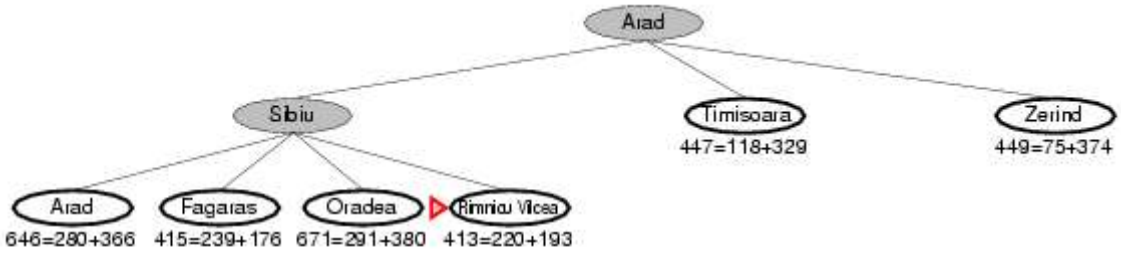
A* Search Algorithm

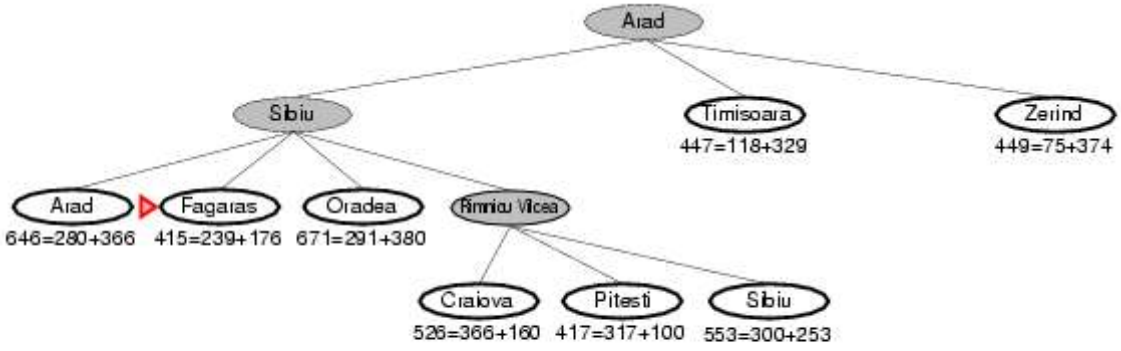
A* search is the most widely used informed search algorithm where a node n is evaluated by combining values of the functions **g(n)**and **h(n).** The function **g(n) is the path cost from the start/initial node to a node n** and **h(n) is the estimated cost of the cheapest path from node n to the goal node.** Therefore, we have

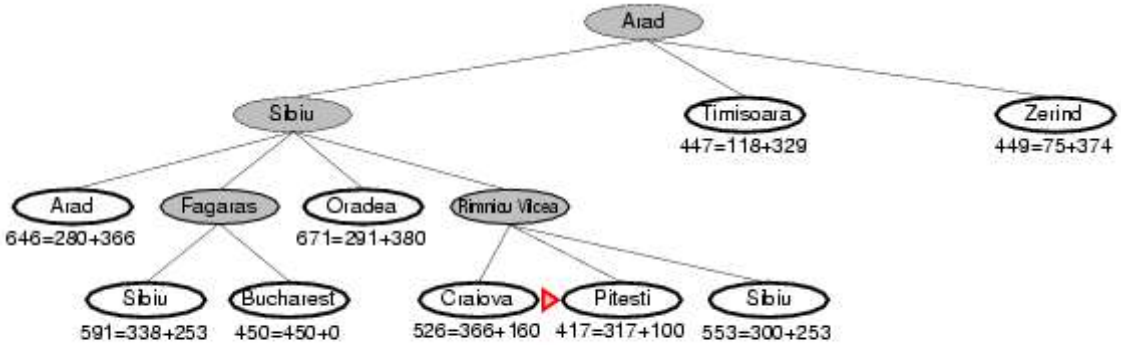
f(n)=g(n)+h(n)

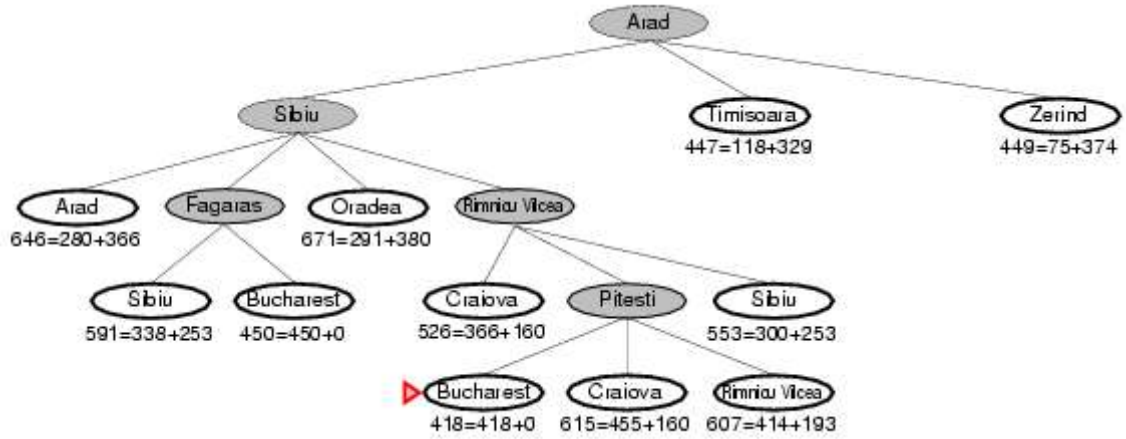
where **f(n) is the estimated cost of the cheapest solution through n.**
So, in order to find the cheapest solution, try to find the lowest values of f(n)




Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253



Calculation of f(n) for node S:

f(A)=(distance from node A to A) + h(A)

- 0+366=366.

Calculation of f(n) for node T:
f(A)=(distance from node A to S)+h(S)

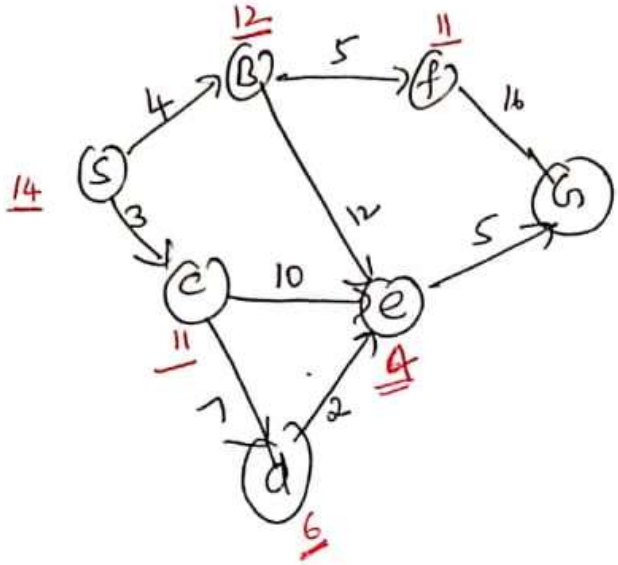
- 140+253=393

Calculation of f(n) for node B:
f(B)=(distance from node A to T)+h(T)

- 118+329=447

Therefore, node S has the lowest f(n) value. Hence, node S will be explored to its next level and again calculate the lowest f(n) value.

A* Algorithm



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

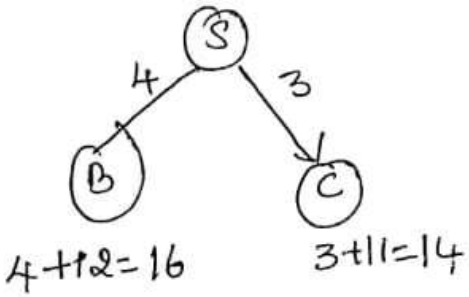
↓ ↓ Estimated

Actual cost from start node to N | cost from N to goal node

$f(B) = 4 + 12 = 16$

$f(C) = 3 + 11 = 14.$

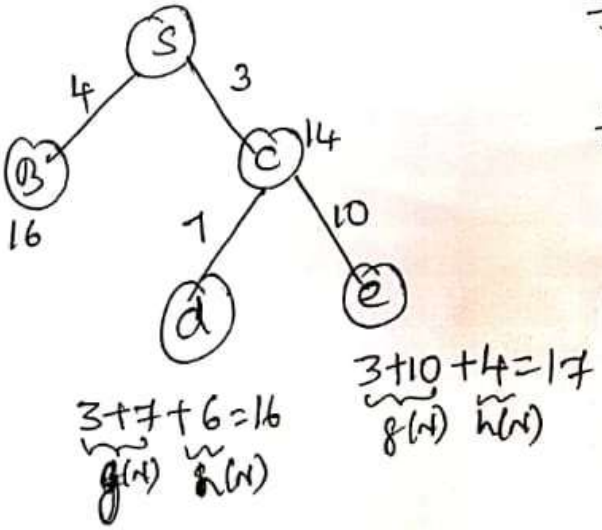
1)



$f(d) = \overset{3+7}{10} + 6 = 16$

$f(e) = \overset{3+10}{13} + 4 = 17.$

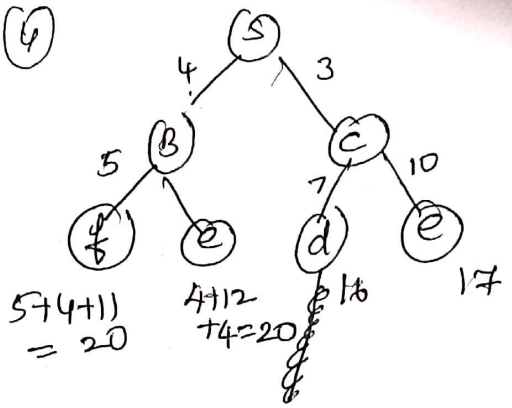
2)



Note heuristic values

S = 14
B = 12
C = 11
d = 6
e = 4
f = 11
G = 0

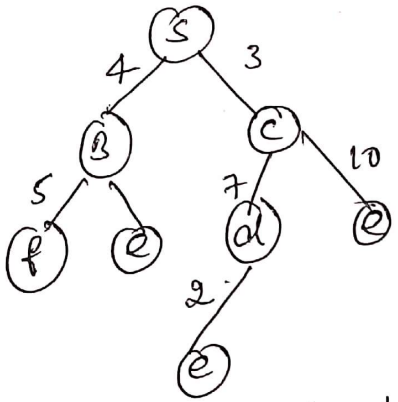
(4)



5+4+11 = 20

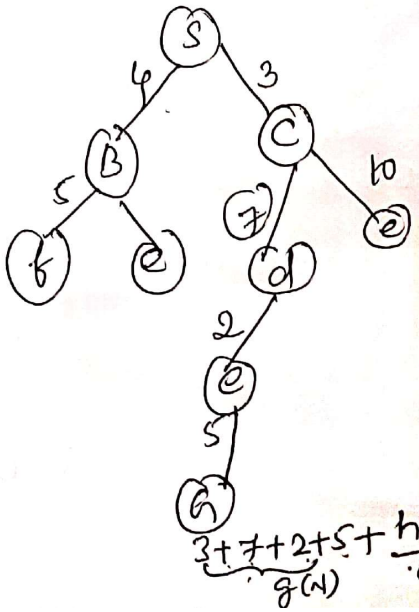
4+12 +4=20

(5)



$\underbrace{3+7+2}_{g(N)} + \underbrace{4}_{h(N)} = 16$

(6)



$\underbrace{3+7+2+5}_{g(N)} + \underbrace{h(N)}_{0}^{G} = 17$

The performance measure of A* search

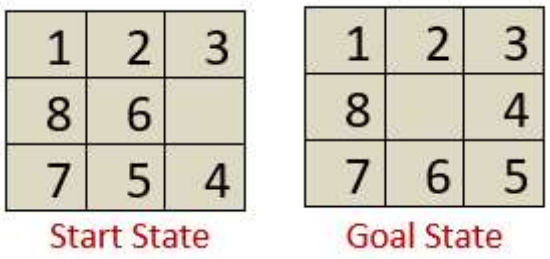
- **Completeness:** The star(*) in A* search guarantees to reach the goal node.
- **Optimality:** An underestimated cost will always give an optimal solution.
- **Space and time complexity:** A* search has **$O(b^d)$ space and time** complexities.

Heuristic Functions

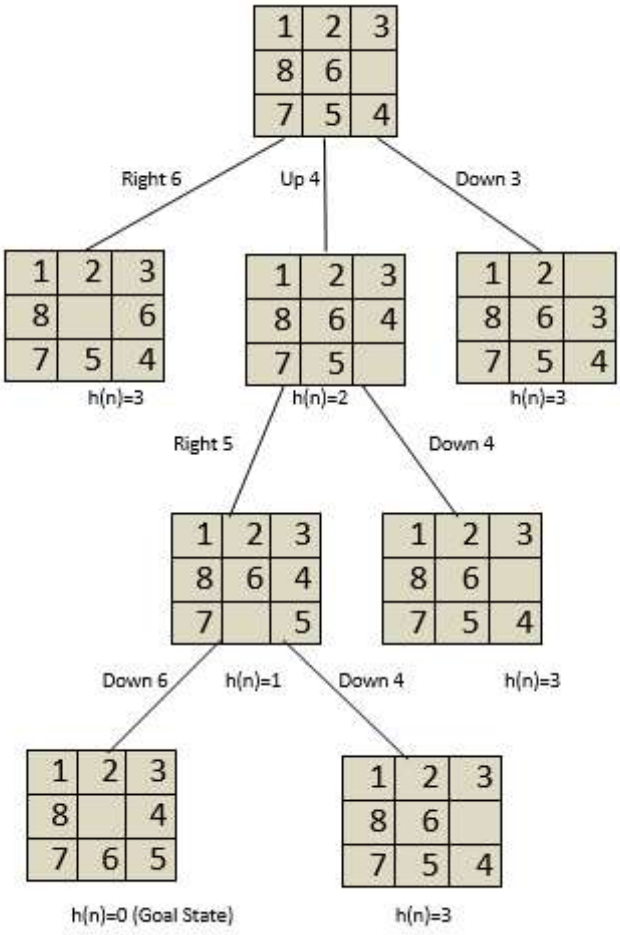
As we have already seen that an informed search make use of heuristic functions in order to reach the goal node in a more prominent way. Therefore, there are several pathways in a search tree to reach the goal node from the current node. The selection of a good heuristic function matters certainly. *A good heuristic function is determined by its efficiency. More is the information about the problem, more is the processing time.*

Some toy problems, such as 8-puzzle, 8-queen, tic-tac-toe, etc., can be solved more efficiently with the help of a heuristic function. Let's see how:

Consider the following 8-puzzle problem where we have a start state and a goal state. Our task is to slide the tiles of the current/start state and place it in an order followed in the goal state. There can be four moves either **left, right, up, or down.** There can be several ways to convert the current/start state to the goal state, but, we can use a heuristic function h(n) to solve the problem more efficiently.



We can construct a state-space tree to minimize the h(n) value to 0as shown below:



,

It is seen from the above state space tree that the goal state is minimized from h(n)=3 to h(n)=0. However, we can create and use several heuristic functions as per the requirement.

Properties of a Heuristic search Algorithm

- **Admissible Condition:** An algorithm is said to be admissible, if it returns an optimal solution.
- **Completeness:** An algorithm is said to be complete, if it terminates with a solution (if the solution exists).