SCSX1021

ARTIFICIAL INTELLIGENCE

UNIT: I

INTRODUCTION AND PROBLEM SOLVING

Introduction

What is artificial intelligence?

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

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 are categorized into four approaches and are summarized in the table below :

i) **Systems that think like humans**- "The exciting new effort to make computers think as if machines with minds, in the full and literal sense."

ii) **Systems that think rationally**- "The study of mental faculties through the use of computer models.

iii) **Systems that act like humans**- "The art of creating machines that perform functions that require intelligence when performed by people."

iv) **Systems that act rationally**- "Computational intelligence is the study of the design of intelligent agents."

The four approaches in more detail are as follows :

(a)**Acting humanly** : The Turing Test Approach

The Turing 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 :

i) **Natural language processing**- to enable it to communicate successfully in English.

ii) **Knowledge representation** - to store what it knows or hears.

iii) **Automated reasoning**- to use the stored information to answer questions and to draw new conclusions.

iv) **Machine learning**- to adapt to new circumstances and to detect and extrapolate patterns.

To pass the complete Turing Test, the computer will need

i) **Computer vision** to perceive the objects
ii) **Robotics** to manipulate objects and move about.

b) **Thinking humanly** : The Cognitive Modeling Approach

A machine program is constructed to think like a human but for that 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 Eg. Allen Newell and Herbert Simon, who developed GPS, the "General Problem Solver" tried to trace the reasoning steps to traces of human subjects solving the same problems.

(c) Thinking rationally: The "Laws of Thought Approach"

It is Normative (or prescriptive) rather than descriptive The Greek philosopher Aristotle was one of the first to attempt to codify "right Thinking" that is irrefutable reasoning processes. 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. An agent is something that acts. Computer agents are not mere programs, but they are expected to have the following attributes also :

(i) operating under autonomous control

(ii) perceiving their environment

(iii) persisting over a prolonged time period

(iv) adapting to change.

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

Foundations of AI

The various disciplines that contributed ideas, viewpoints, and techniques to AI are given below :

Philosophy :

i) Can formal rules be used to draw valid conclusions?

ii) How does the mental mind arise from a physical brain?
iii) Where does knowledge come from?
iv) How does knowledge lead to action?

Philosophers (going back to 400 B.C.) made AI conceivable by considering the ideas that the mind is in some ways like a machine that it operates on knowledge encoded in some internal language and that thought can be used to choose what actions to take. Aristotle (384-322 B.C.) was the first to formulate a precise set of laws governing the rational part of the mind. He developed an informal system of syllogisms for proper reasoning, which allowed one to generate conclusions mechanically, given initial premises.

Mathematics :- Formal representation and proof algorithms, computation, (un)decidability, (in)tractability , probability.

i)What are the formal rules to draw valid conclusions?

ii)What can be computed?

iii)How do we reason with uncertain information?

Economics:

i)How should we make decisions so as to maximize payoff?

ii)How should we do this when others may not go along?

iii)How should we do this when the payoff may be far in the future?

Neuroscience:

i)How do brains process information?

Psychology:

i)How do humans and animals think and act?

Psychologists adopted the idea that humans and animals can be considered information processing machines. The origin of scientific psychology are traced back to the wok of German physiologist Hermann von Helmholtz (1821-1894) and his student Wilhelm Wundt(1832 – 1920). In 1879, Wundt opened the first laboratory of experimental psychology at the University of Leipzig. In US, the development of computer modeling led to the creation of the field of cognitive science. The field can be said to have started at the workshop in September 1956 at MIT.

Computer Engineering:

i)How can we build an efficient computer?

For artificial intelligence to succeed, we need two things: intelligence and an artifact. The computer has been the artifact of choice. AI also owes a debt to the software side of

computer science, which has supplied the operating systems, programming languages, and tools needed to write modern programs. Computer engineers provided the artifacts that make AI applications possible. AI programs tend to be large, and they could not work without the great advances in speed and memory that the computer industry has provided.

Control theory and Cybernetics:

i)How can artifacts operate under their own control?

Ktesibios of Alexandria (c. 250 B.c.) built the first self-controlling machine: a water clock with a regulator that kept the flow of water running through it at a constant, predictable pace. Modern control theory, especially the branch known as stochastic optimal control, has as its goal the design of systems that maximize an objective function over time.

Linguistics

Linguists showed that language use fits into this model. Modern linguistics and AI, then, were "born" at about the same time, and grew up together, intersecting in a hybrid field called computational linguistics or natural language processing.

History of AI

The first work is now generally recognized as AI was done by Warren McCulloch and Walter Pitts(1943). They proposed a model of artificial neuron in which each neuron is characterized as being on or off. Donald Hebb(1949) demonstrated a simple updating rule for modifying the connection strengths between neurons. His rule is called as Hebb rule. There were a number of early examples of work that can be characterized as AI, but it was Alan Turing who first articulated a complete vision of A1 in his 1950 article "Computing Machinery and Intelligence." Therein, he introduced the Turing test, machine learning, genetic algorithms, and reinforcement learning.

McCarthy convinced Minsky, Claude Shannon, and Nathaniel Rochester to help him bring together U.S. researchers interested in automata theory, neural nets, and the study of intelligence.

John McCarthy(1956) coined the term "artificial intelligence" as the topic of the Dartmouth Conference, the first conference devoted to the subject. Demonstration of the first running AI program, the Logic Theorist (LT) written by Allen Newell, J.C. Shaw and Herbert Simon (Carnegie Institute of Technology, now Carnegie Mellon University).

In 1957, The General problem Solver(GPS) demonstrated by Newell, Shaw & Simon. General Problem Solver (GPS) was a computer program created in 1957 by Herbert Simon and Allen Newell to build a universal problem solver machine. The order in which the program considered subgoals and possible actions was similar to that in which humans

approached the same problems. Thus, GPS was probably the first program to embody the "thinking humanly" approach.

Lisp was invented by John McCarthy in 1958 while he was at the Massachusetts Institute of Technology (MIT). In 1963, McCarthy started the AI lab at Stanford.

Dendral was an influential pioneer project in artificial intelligence (AI) of the 1960s, and the computer software expert system that it produced. Its primary aim was to help organic chemists in identifying unknown organic molecules, by analyzing their mass spectra and using knowledge of chemistry. It was done at Stanford University by Edward Feigenbaum, Bruce Buchanan, Joshua Lederberg, and Carl Djerassi.

AI becomes an industry in 1980 to present. In 1981, the Japanese announced the "Fifth Generation" project, a 10-year plan to build intelligent computers running Prolog.

In 1985 to 1995, Neural networks return to popularity. In 1988- Resurgence of probability; general increase in technical depth Nouvelle AI": ALife, GAs, soft computing. In 1995, One of the most important environments for intelligent agents is the Internet. In 2003, Human-level AI back on the agenda.

Intelligent agent

Agents and Environment

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 the below Figure 1.1.

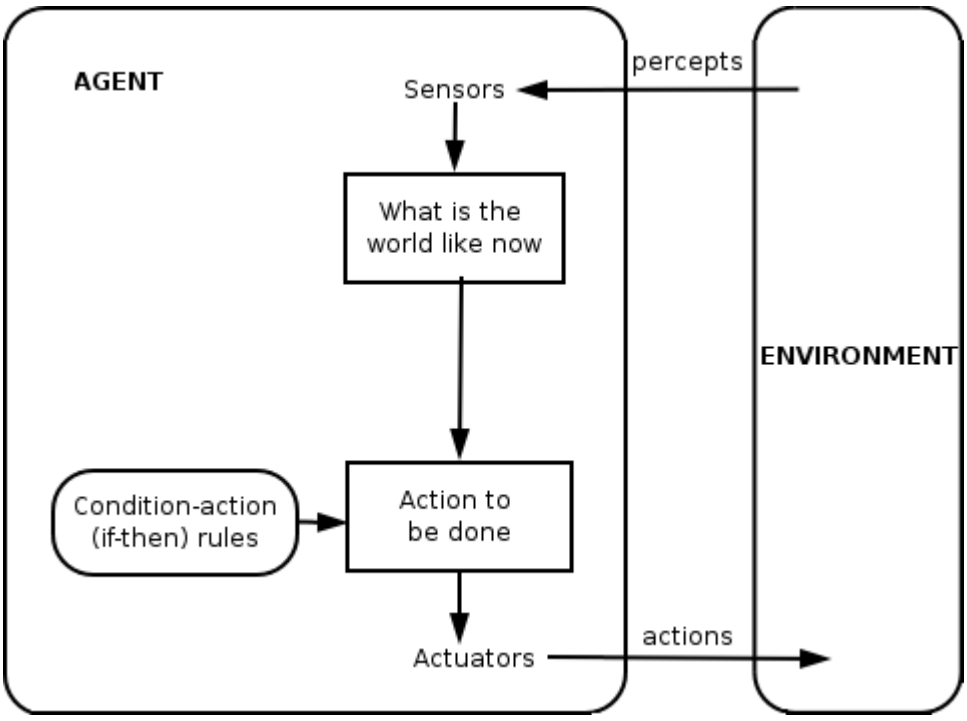


Figure 1.1 Agents interact with environments through sensors and actuators

- 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.

Percept

Percept refers 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

It is a map from the precept sequence to an action.

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 a very simple example the **vacuum-cleaner** world is shown in Figure 1.2. 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 1.3.

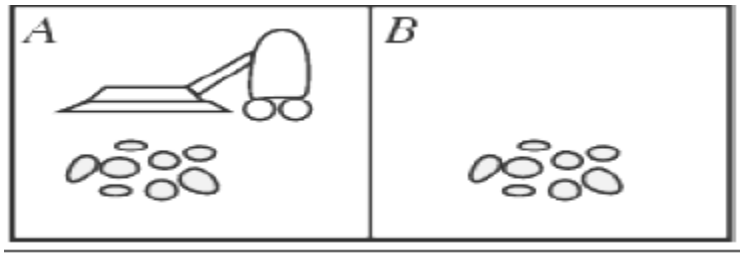


Figure 1.2 Vacuum Cleaner world with two location

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 1.3 Agent function for Vacuum Cleaner

Concept of Rationality

An agent should act as a Rational Agent. A rational agent is one that does the right thing that is the right actions will cause the agent to be most successful in the environment.

Performance measures

A performance measures embodies the criterion for success of an agent‘s behavior. As a general rule, it is better to design performance measures according to what one actually wants in the environment, rather than according to how one thinks the agent should behave.

Rationality

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

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

This leads to a **definition of a rational agent**

Omniscience, learning, and autonomy

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

A rational agent not only to gather information,but also to learn as much as possible from what it perceives. The agent‘s initial configuration could reflect some prior knowledge of the environment, but as the agent gains experience this may be modified and augmented.

Successful agents **split the task** of computing the agent function into **three different periods:** when the agent is being designed, some of the computation is done by its designers; when it is deliberating on its next action, the agent does more computation; and as it learns from experience, it does even more computation to decide how to modify its behavior.

The Nature of Environment

Specifying the task environment

A task environment specification includes the performance measure, the external environment, the actuators, and the sensors. In designing an agent, the first step must always be to specify the task environment as fully as possible. Task environments are specified as a) **PEAS (Performance, Environment, Actuators, Sensors)** description, both means the same.

Example: Taxi Task Environment

The below table describes the task environment for an automated taxi.

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

Properties of task Environment

i) Fully observable vs. Partially observable

If an agent‘s sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is fully observable. Fully observable environments are convenient because the agent need not maintain any internal state to keep track of the world.

An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data.

ii)Deterministic vs. non-deterministic (or) stochastic

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

iii)Episodic vs. non-episodic (or) sequential

In an episodic environment, the agent‘s experience is divided into atomic episodes. Each episode consists of its own percepts and actions and it does not depend on the previous episode. In sequential environments, the current decision could affect all future of decisions. Eg. Chess and taxi driving.

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

iv)Static vs. dynamic

If the environment is changing for agent‘s action then the environment is dynamic for that agent otherwise it is static. If the environment does not change for some time, then it changes due to agent‘s performance is called semi dynamic environment.

E.g. Taxi driving is dynamic.

Chess is semi dynamic.

Crossword puzzles are static.

v)Discrete vs. continuous

If the environment has limited number of distinct, clearly defined percepts and actions then the environment is discrete. E.g. Chess.

If the environment changes continuously with range of value the environment is continuous. E.g. Taxi driving.

Structure of Agents

An intelligent agent is a combination of Agent Program and Architecture.

Intelligent Agent = Agent Program + Architecture

Agent Program is a function that implements the agent mapping from percepts to actions.

Architecture is a computing device used to run the agent program.

Types of Agent

To perform the mapping task **four types of agent programs** are there. They are:

1.Simple reflex agents

2. Model-based reflex agents

3. Goal-based agents

4. Utility-based agents

1.Simple reflex agents

The simplest kind of agent is the simple reflex agent. It responds directly to percepts i.e. these agent select actions on the basis of the current percept, ignoring the rest of the percept history.

An agent describes about how the **condition action rules** allow the agent to make the connection from percept to action.

Condition action rule: if condition then action

Rectangle to denote the current internal state of the agents decision process. **Oval** to represent the background information in the process.

The agent program, which is also very simple, is shown below.

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

static: rules, a set of condition-action rules

state ← INTERPRET – INPUT(percept)

rule ← RULE – MATCH(state, rules)

action ← RULE – ACTION[rule]

return action
```

INTERRUPT-INPUT – function generates an abstracted description of the current state from the percept.

RULE-MATCH – function returns the first rule in the set of rules that matches the given state description.

RULE-ACTION – the selected rule is executed as action of the given percept.

The agent in figure 1.4 will work only ―if the correct decision can be made on the basis of only the current percept – that is, only if the environment is fully observable‖.

Example: Medical diagnosis system

If the patient has reddish brown spots **then** start the treatment for measles.

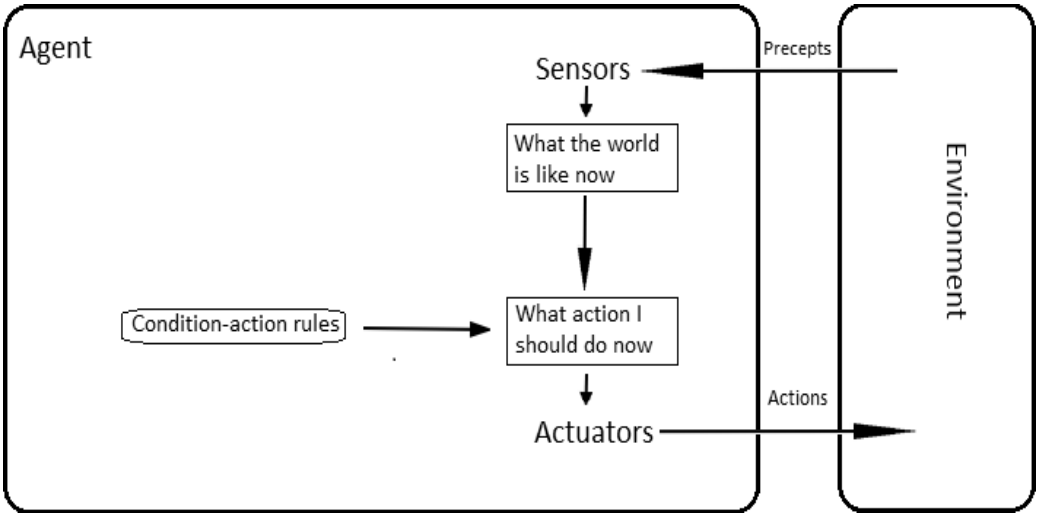


Figure 1.4 Simple Reflex agent

2.Model-based reflex agents (Agents that keep track of the world)

- 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 which combines the current percept with the old internal state to generate updated description of the current state.
- The current percept is combined with the old internal state and it derives a new current state is updated in the state description is also.
- This updation requires two kinds of knowledge in the agent program. First, we need some information about how the world evolves independently of the agent. Second, we need some information about how the agents own actions affect the world.

The above two knowledge implemented in simple Boolean circuits or in complete scientific theories is called a model of the world. An agent that uses such a model is called a model- based agent.

The figure 1.5 shows the structure of the reflex agent with internal state, showing how the current percept id combined with the old internal state to generate the updated description of the current state.

The agent program, which is shown below

```
function REFLEX-AGENT-WITH-STATE (percept) returns an action

static: state, a description of the current world state rules, a set of condition-action rules
action, the most recent action, initially none

state ← UPDATE-STATE(state, action, percept)

rule ← RULE-MATCH(state, rules)

action ← RULE-ACTION[rule]
```

return action

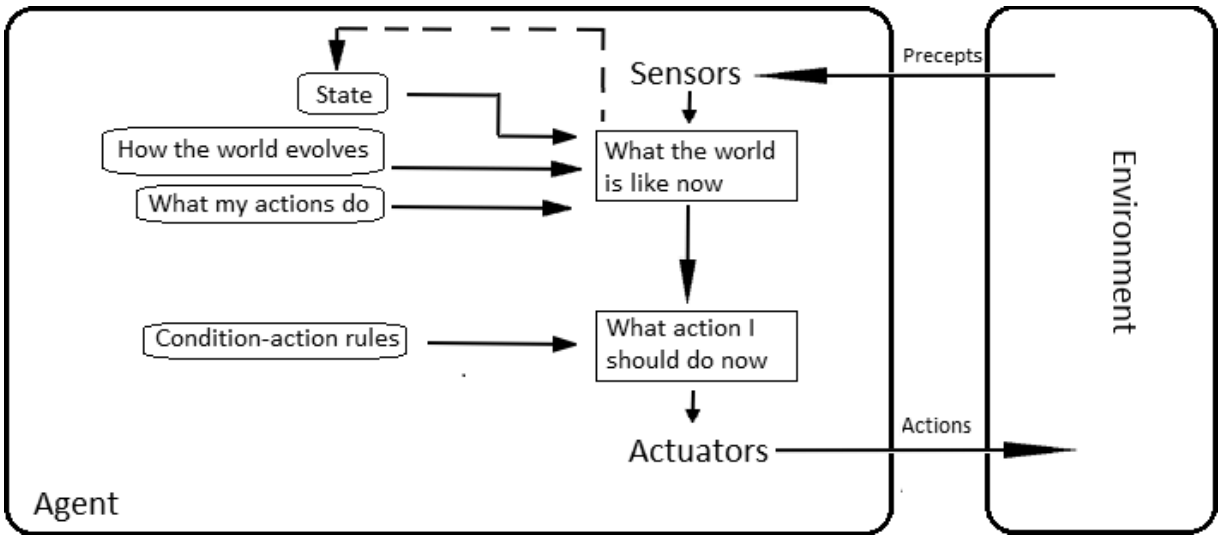


Figure 1.5 Model based Agent

UPDATE-STATE – This is responsible for creating the new internal state description by combining percept and current state description.

3.Goal-based agents

An agent knows the description of current state and also needs some sort of goal information that describes situations that are desirable. The action matches with the current state is selected depends on the goal state.

The goal-based agent appears less efficient, it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified. The goal-based agent‘s behavior can easily be changed to go to a different location. The Figure 1.6 shows the goal based agent.

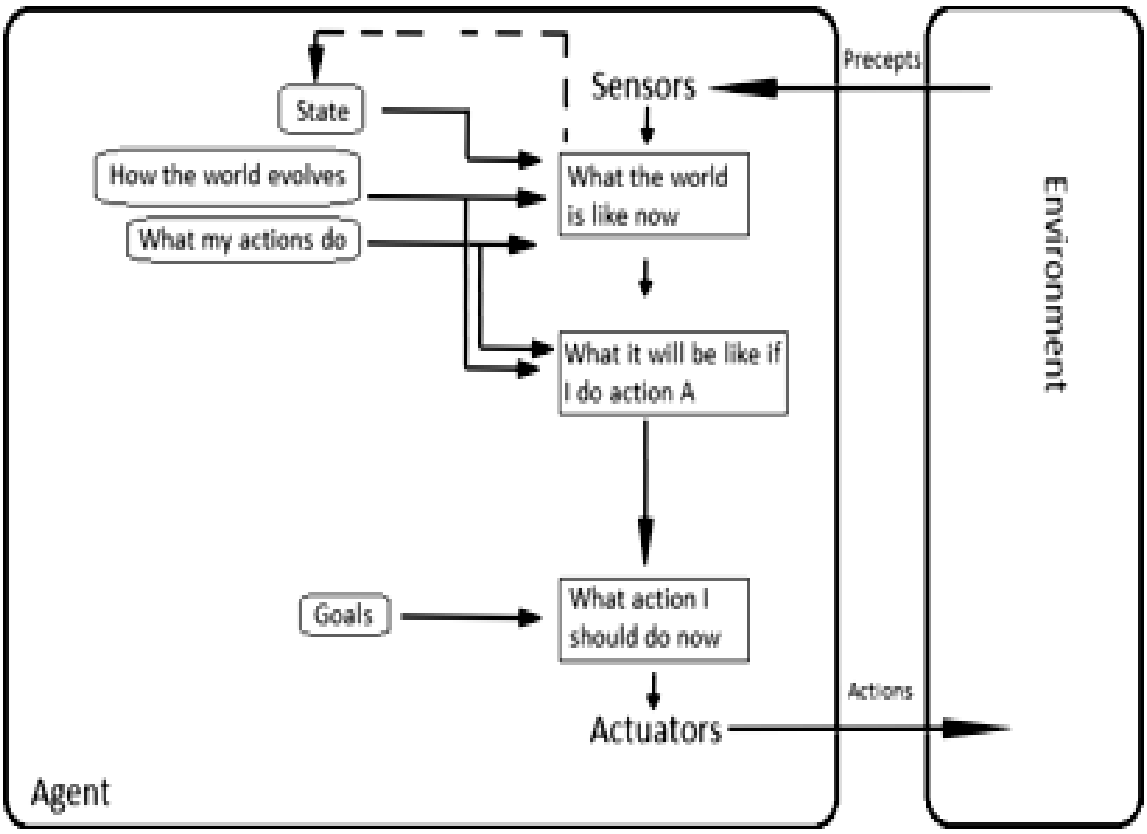


Figure 1.6 Goal based agent

4. Utility-based agents (Utility – refers to **—** the quality of being usefull)

An agent generates a goal state with high – quality behavior (utility) that is, if more than one sequence exists to reach the goal state then the sequence with more reliable, safer, quicker and cheaper than others to be selected.

A utility function maps a state (or sequence of states) onto a real number, which describes the associated degree of happiness. The utility function can be used for two different cases: First, when there are conflicting goals, only some of which can be achieved (for e.g., speed and safety), the utility function specifies the appropriate tradeoff. Second, when the agent aims for several goals, none of which can be achieved with certainty, then the success can be weighted up against the importance of the goals. The Figure 1.7 shows the utility based agent.

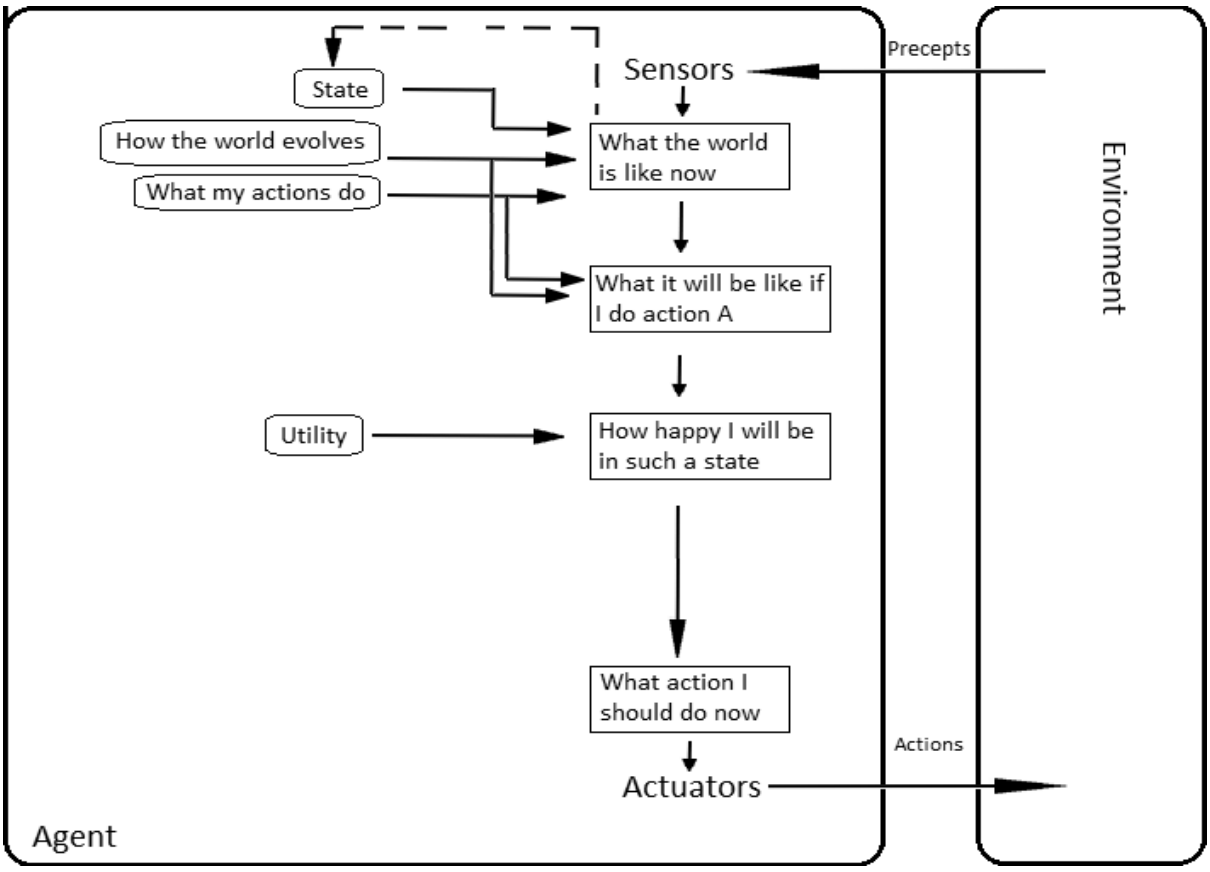


Figure 1.7 Utility based Agent

5.Learning agents

The **learning** task allows the agent to operate in initially unknown environments and to become more competent than its initial knowledge. A learning agent can be divided into four conceptual components, .

i)Learning element – This is responsible for making improvements. It uses the 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.

ii)Performance element – which is responsible for selecting external actions and it is equivalent to agent: it takes in percepts and decides on actions.

iii)Critic – It tells the learning element how well the agent is doing with respect to a fixed performance standard.

iv)Problem generator – It is responsible for suggesting actions that will lead to new and informative experiences.

In summary, agents have a variety of components, and those components can be represented in many ways within the agent program, so there appears to be great variety among learning methods. Learning in intelligent agents can be summarized as a process of modification of each component of the agent to bring the components into closer agreement with the available feedback information, thereby improving the overall performance of the

agent (All agents can improve their performance through learning). The Figure 1.8 shows the learning based agent.

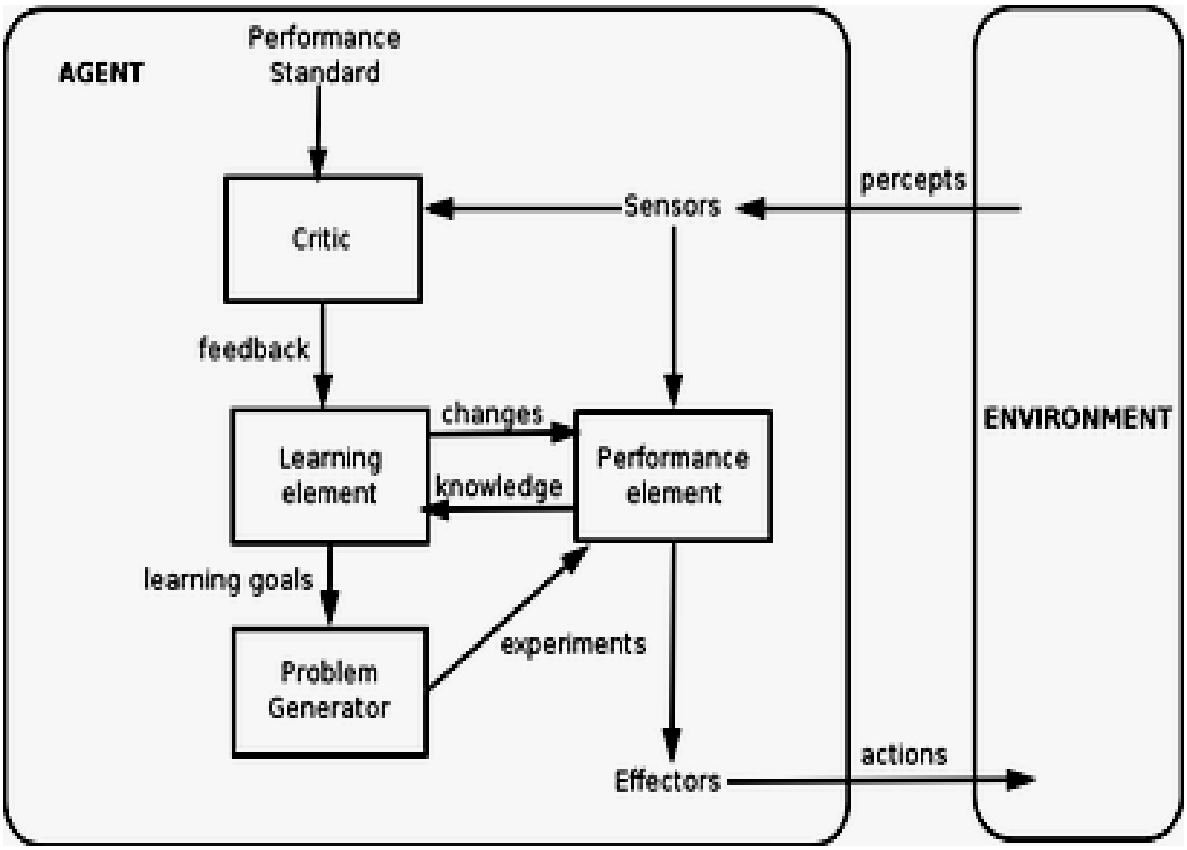


Figure 1.8 Learning Based Agent

Problem solving agents

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.

Problem Formulation

Problem solving agent is one type of goal based agent, where there are sequence of actions, the agent should select one from these actions which lead to desirable states. If the agent understands the definition of a problem, then we have to search the finding solution which implies agent should maximize the performance measures.

Steps to Maximize the Performance Measure:

1. **Goal Formulation:** It is based on the current situation and the agent's performance measure, is the first step in problem solving. The agent's task is to find out which sequence of actions will get to a goal state and the actions that result to a failure case which can be rejected without further consideration.
2. **Problem formulation:** It is the process of deciding what actions and states to consider given a goal.

3. **Search:** An agent with several immediate options of unknown value can decide what to do by examining different possible sequences of actions that leads to the states of known value, and then choosing the best sequence. The process of looking for sequences actions from the current state to reach the goal state is called **search.**

4. **Solution**: The **search algorithm** takes a **problem** as **input** and returns a **solution** in the form of **action sequence.**

5. Execution: Once a solution is found, the **execution phase** consists of carrying out the recommended action..

The below program explains about the simple problem solving agent.

```
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
```

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 taken

• **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.

Well-defined problems and solutions

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

1) The **initial state** that the agent starts in . The initial state for our agent of example problem is described by *In(Arad)*

2) 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)] }

i)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.

ii) A **path** in the state space is a sequence of states connected by a sequence of actions.

3) **Goal Test** : It determines whether the given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. For example, in chess, the goal is to reach a state called "checkmate," where the opponent's king is under attack and can't escape.

4) **Path Cost :** A **path cost** function assigns numeric cost to each action. For the Romania problem the cost of path might be its length in kilometers.

(a) 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 is shown in figure 1.9. It is assumed that the step costs are non negative.

(b) A **solution** to the problem is a path from the initial state to a goal state.

(c) An **optimal solution** has the lowest path cost among all solutions.

Abstraction

The process of removing detail from a representation is called abstraction. In the above route finding problem the state's In (Arad) action has so many things: the traveling companions, the scenery out of the window, how far is to the next stop, the condition of road and the weather.

The abstraction is *useful* if carrying out each of the actions in the solution is easier than the original problem; The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out.

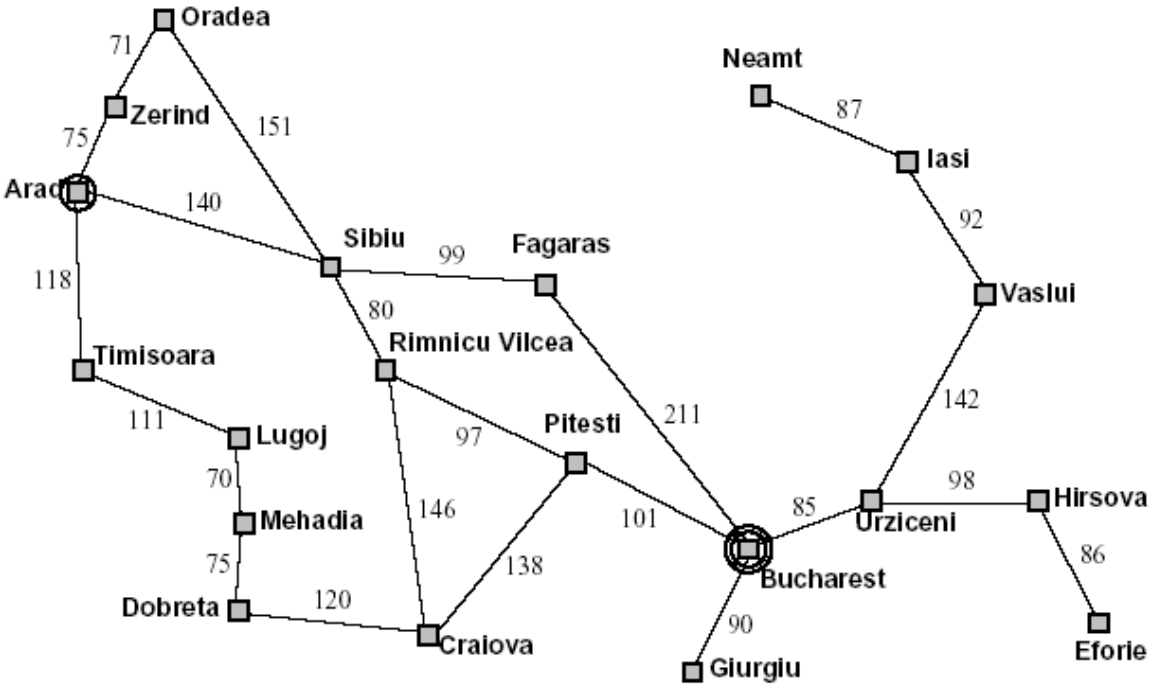


Figure 1.9 Simplified Road map of Romania

Example: Route finding problem

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

A **problem** is defined by four items:

initial state e.g., “at Arad"

successor function S(x) = set of action-state pairs e.g., S(Arad) = {[Arad -> Zerind;Zerind],….}

goal test, can be explicit, e.g., x = at Bucharest"

path cost (additive)

e.g., sum of distances, number of actions executed, etc.

c(x; a; y) is the step cost, assumed to be >= 0

A **solution** is a sequence of actions leading from the initial state to a goal state.

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

i)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.

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

i)TOY PROBLEMS

1.Vacuum World Example

o **States**: The agent is in one of two locations. each of which might or might not contain dirt. Thus there are 2 x 22 = 8 possible world states.

o **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 2.3

o **Goal Test**: This tests whether all the squares are clean.

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

Vacuum World State Space shown in figure 1.10

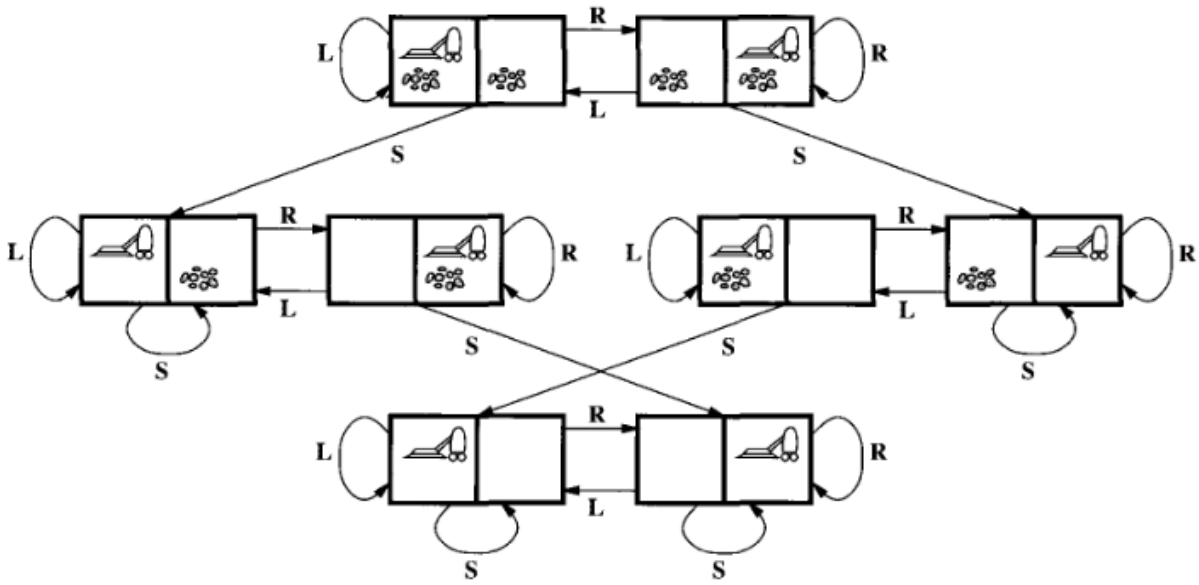


Figure 1.10 State space for the Vaccum world

2.The 8-puzzle

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

Example: The 8-puzzle shown in Figure 1.11.

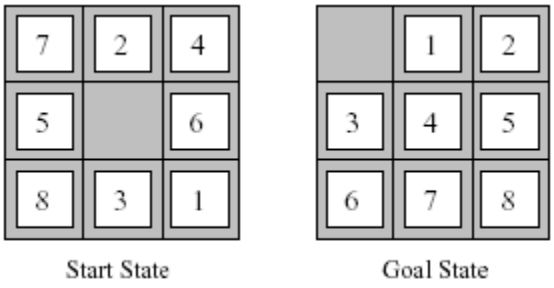


Figure 1.11 8 Puzzle

The problem formulation is as follows :

o **States** : A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.

o **Initial state** : Any state can be designated as the initial state. It can be noted that any given goal can be reached from exactly half of the possible initial states.

o **Successor function** : This generates the legal states that result from trying the four actions(blank moves Left, Right, Up or down).

o **Goal Test** : This checks whether the state matches the goal configuration shown in figure 2.4.(Other goal configurations are possible)

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

The 8-puzzle belongs to the family **of sliding-block puzzles**, which are often used as test problems for new search algorithms in AI. This general class is known as NP-complete.

The **8-puzzle** has 9!/2 = 181,440 reachable states and is easily solved.

The **15 puzzle** (4 x 4 board) has around 1.3 trillion states, an the random instances can be solved optimally in few milli seconds by the best search algorithms.

The **24-puzzle** (on a 5 x 5 board) has around 1025 states ,and random instances are still quite difficult to solve optimally with current machines and algorithms.

3.8-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 1.12 shows an attempted solution that fails: the queen in the right most columns is attacked by the queen at the top left.

An **Incremental formulation** involves an operator that augments the state description, starting with an empty state. for 8-queens problem, this means each action adds a queen to the state.

A **complete-state formulation** starts with all 8 queens on the board and moves them around.

In either case the path cost is of no interest because only the final state counts.

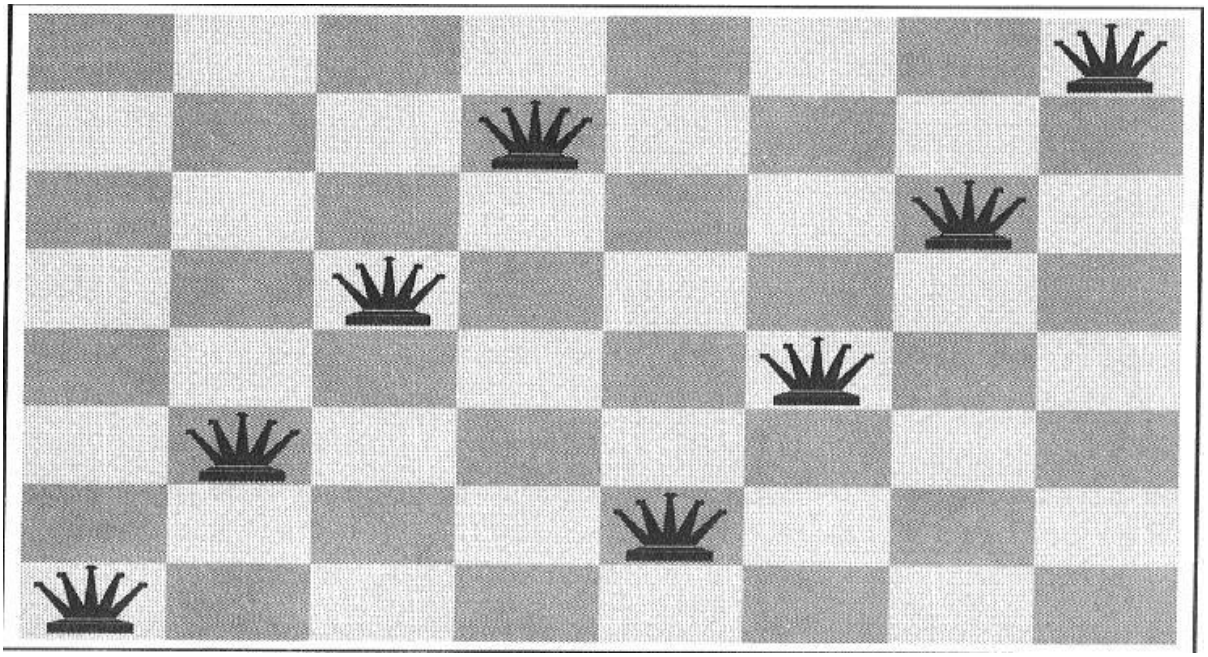


Figure 1.12 8 Queens Problem

The first incremental formulation one might try is the following :

o **States** : Any arrangement of 0 to 8 queens on board is a state.

o **Initial state** : No queen on the board.

o **Successor function** : Add a queen to any empty square.

o **Goal Test** : 8 queens are on the board, none attacked.

In this formulation, we have 64.63…57 = 3 x 1014 possible sequences to investigate.

A better formulation would prohibit placing a queen in any square that is already attacked. :

o **States** : Arrangements of n queens (0 <= n < = 8) ,one per column in the left most columns ,with no queen attacking another are states.

o **Successor function** : Add a queen to any square in the left most empty column such that it is not attacked by any other queen.

This formulation reduces the 8-queen state space from 3 x 1014 to just 2057,and solutions are easy to find.

For the 100 queens the initial formulation has roughly 10400 states whereas the improved formulation has about 1052 states. This is a huge reduction, but the improved state space is still too big for the algorithms to handle.

REAL-WORLD PROBLEMS

i)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.

ii)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.

o **Initial state:** This is specified by the problem.

o **Successor function:** 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.

o **Goal Test :** Are we at the destination by some pre specified time?

o **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.

iii)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.

iv)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.

v)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**.

vi)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.

vii)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 chosen, there will be no way to add some part later without undoing some work 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.

viii)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.

SEARCHING FOR SOLUTIONS

SEARCH TREE

A **search tree** is generated by the **initial state** and the **successor function** that together define the **state space**. In general, we may have a *search graph* rather than a *search tree*, when the same state can be reached from multiple paths.

Figure 1.13 shows some of the expansions in the search tree for finding a route from Arad to Bucharest.

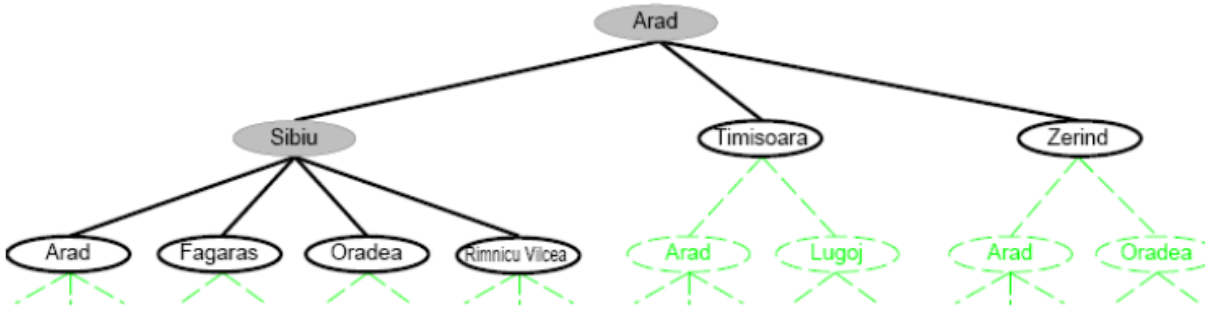


Figure 1.13 Partial search tree for finding the route

From the above diagram

- 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 line

The root of the search tree is a **search node** corresponding to the initial **state**, In(Arad). The first step is to test whether this is a **goal state**. The current state is expanded by applying the successor function to the current state, thereby generating a new set of states. In this case, we get three new states: In(Sibiu),In(Timisoara),and In(Zerind). Now we must choose which of these three possibilities to consider further. This is the essense of search-following up one option now and putting the others aside for latter, in case the first choice does not lead to a solution.

```
function TREE-SEARCH(problem, fringe) returns solution
        fringe  := INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
         loop do
    if EMPTY?(fringe) then return failure
    node  := REMOVE-FIRST(fringe)
    if GOAL-TEST[problem] applied to STATE[node] succeeds
      then return SOLUTION(node)
    fringe       := INSERT-ALL(EXPAND(node, problem), fringe)
```

The choice of which state to expand is determined by the **search strategy**. There are an infinite number paths in this state space, so the search tree has an infinite number of **nodes.**

A **node** is a data structure with five components :

o STATE : a state in the state space to which the node corresponds;

o PARENT-NODE : the node in the search tree that generated this node;

o ACTION : the action that was applied to the parent to generate the node;

o PATH-COST :the cost, denoted by g(n),of the path from initial state to the node, as indicated by the parent pointers; and

o DEPTH : the number of steps along the path from the initial state.

It is important to remember the **distinction between nodes and states**. A node is a book keeping data structure used to represent the search tree in the below figure 1.14 denotes, a state corresponds to configuration of the world. Nodes are on particular paths, as defined by PARENT-NODE pointers, whereas states are not.

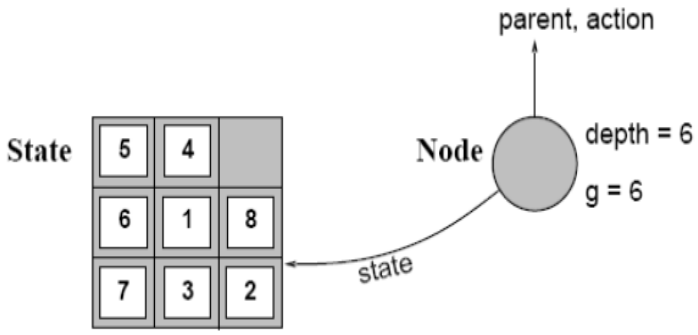


Figure 1.14 Nodes are the data structure from which the search tree is constructed.

Fringe

Fringe is a collection of nodes that have been generated but not yet been expanded. Each element of the fringe is a leaf node, thatis, a node with no successors in the tree. The fringe of each tree consists of those nodes with bold outlines. The collection of these nodes is implemented as a **queue.**

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

The operations specified in Figure 1.26 on a queue are as follows:

o **MAKE-QUEUE(element,…)** creates a queue with the given element(s).

o **EMPTY?(queue)** returns true only if there are no more elements in the queue.

o **FIRST(queue)** returns FIRST(queue) and removes it from the queue.

o **INSERT(element, queue)** inserts an element into the queue and returns the resulting queue.

o **INSERT-ALL(elements, queue)** inserts a set of elements into the queue and returns the resulting queue.

MEASURING PROBLEM-SOLVING PERFORMANCE

The output of problem-solving algorithm is either failure or a solution.(Some algorithms might struck in an infinite loop and never return an output.

The algorithm's performance can be measured in four ways :

o **Completeness** : Is the algorithm guaranteed to find a solution when there is one?

o **Optimality** : Does the strategy find the optimal solution

o **Time complexity** : How long does it take to find a solution?

o **Space complexity** : How much memory is needed to perform the search?

Uninformed search strategies

Uninformed Search Strategies have no additional information about states beyond that provided in the **problem definition**.

Strategies that know whether one non goal state is "more promising" than another are called **Informed search or heuristic search** strategies.

There are five uninformed search strategies as given below.

o Breadth-first search

o Uniform-cost search

o Depth-first search

o Depth-limited search

o Iterative deepening search

Breadth first search

Breadth First Search (BFS) searches breadth-wise in the problem space. Breadth-First search is like traversing a tree where each node is a state which may a be a potential candidate for solution. It expands nodes from the root of the tree and then generates one level of the tree at a time until a solution is found. It is very easily implemented by maintaining a queue of nodes. Initially the queue contains just the root. In each iteration, node at the head of the queue is removed and then expanded. The generated child nodes are then added to the tail of the queue.

Algorithm: Breadth-First Search

1. Create a variable called NODE-LIST and set it to the initial state.
2. Loop until the goal state is found or NODE-LIST is empty.
    a. Remove the first element, say E, from the NODE-LIST. If NODE-LIST was empty then quit.
    b. For each way that each rule can match the state described in E do:

i) Apply the rule to generate a new state.

ii) If the new state is the goal state, quit and return this state.

iii) Otherwise add this state to the end of NODE-LIST

Since it never generates a node in the tree until all the nodes at shallower levels have been generated, *breadth-first search* always finds a shortest path to a goal. Since each node can be generated in constant time, the amount of time used by Breadth first search is proportional to the number of nodes generated, which is a function of the branching factor b and the solution d. Since the number of nodes at level d is b^d, the total number of nodes

generated in the worst case is $b + b^2 + b^3 + \ldots + b^d$ i.e. $O(b^d)$, the asymptotic time complexity of breadth first search.

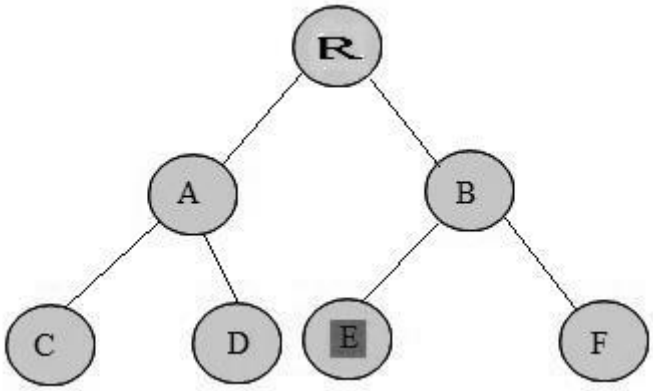


Figure 1.15 Simple binary tree for Breadth First Search

Breadth First Search

Look at the above figure 1.15 with nodes starting from root node, R at the first level, A and B at the second level and C, D, E and F at the third level. If we want to search for node E then BFS will search level by level. First it will check if E exists at the root. Then it will check nodes at the second level. Finally it will find E a the third level.

Advantages of Breadth-First Search

1. Breadth first search will never get trapped exploring the useless path forever.
2. If there is a solution, BFS will definitely find it out.
3. If there is more than one solution then BFS can find the minimal one that requires less number of steps.

Disadvantages of Breadth-First Search

1. The main drawback of Breadth first search is its memory requirement. Since each level of the tree must be saved in order to generate the next level, and the amount of memory is proportional to the number of nodes stored, the space complexity of BFS is $O(b^d)$. As a result, BFS is severely space-bound in practice so will exhaust the memory available on typical computers in a matter of minutes.
2. If the solution is farther away from the root, breath first search will consume lot of time.

Uniform cost search

i) If all the edges in the search graph do not have the same cost then breadth-first search generalizes to uniform-cost search.

ii)Instead of expanding nodes in order of their depth from the root, uniform-cost search expands nodes in order of their cost from the root.

iii)At each step, the next step n to be expanded is one whose cost g(n) is lowest where g(n) is the sum of the edge costs from the root to node n. The nodes are stored in a priority queue. This algorithm is also known as Dijkstra's single-source shortest algorithm.

iv)Whenever a node is chosen for expansion by uniform cost search, a lowest-cost path to that node has been found.

v)The worst case time complexity of uniform-cost search is $O(b^c/m)$, where c is the cost of an optimal solution and m is the minimum edge cost.

Depth first search

Depth First Search (DFS) searches deeper into the problem space. Breadth-first search always generates successor of the deepest unexpanded node. It uses last-in first-out stack for keeping the unexpanded nodes. More commonly, depth-first search is implemented recursively, with the recursion stack taking the place of an explicit node stack.

Algorithm: Depth First Search

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

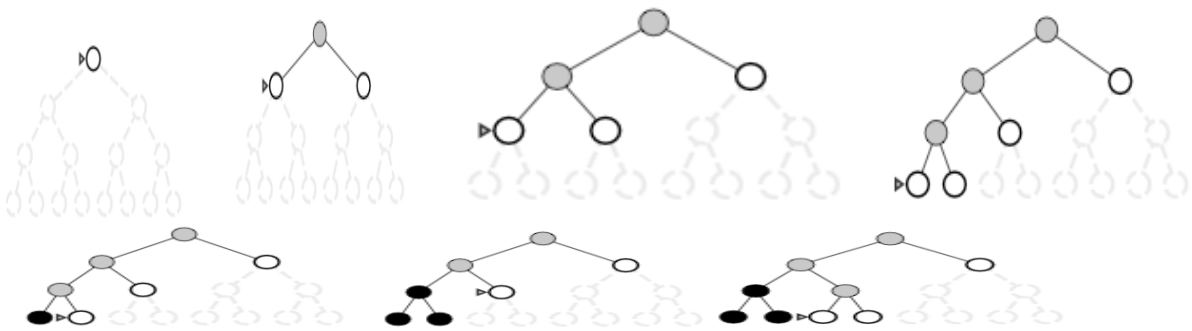
2.Otherwise, loop until success or failure is signaled.

a) Generate a state, say E, and let it be the successor of the initial state. If there is no successor, signal failure.

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

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

Example:



Advantages of Depth-First Search

- The advantage of depth-first Search is that memory requirement is only linear with respect to the search graph. This is in contrast with breadth-first search which requires more space. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node.

- The time complexity of a depth-first Search to depth d is O(b^d) since it generates the same set of nodes as breadth-first search, but simply in a different order. Thus practically depth-first search is time-limited rather than space-limited.

- If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.

Disadvantages of Depth-First Search

- The disadvantage of Depth-First Search is that there is a possibility that it may go down the left-most path forever. Even a finite graph can generate an infinite tree. One solution to this problem is to impose a cutoff depth on the search. Although the ideal cutoff is the solution depth d and this value is rarely known in advance of actually solving the problem. If the chosen cutoff depth is less than d, the algorithm will fail to find a solution, whereas if the cutoff depth is greater than d, a large price is paid in execution time, and the first solution found may not be an optimal one.

- Depth-First Search is not guaranteed to find the solution.

- And there is no guarantee to find a minimal solution, if more than one solution exists.

Depth limited search

Depth-first search will not find a goal if it searches down a path that has infinite length. So, in general, depth-first search is not guaranteed to find a solution, so it is not complete.

This problem is eliminated by limiting the depth of the search to some value l. However, this introduces another way of preventing depth-first search from finding the goal: if the goal is deeper than l it will not be found.

How would you make an intelligent guess for l for a given search problem?

Its time complexity is $O(bl)$ and its space complexity is $O(bl)$. What would the space complexity be of the backtracking version of this search?

Regular depth-first search is a special case, for which $l=\infty$. The algorithm for Depth limited search is given below.

```
function Depth-Limited-Search( problem, limit) returns a solution/fail/cutoff
    return Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)
function Recursive-DLS(node, problem, limit) returns solution/fail/cutoff
    cutoff-occurred? false
    if Goal-Test(problem,State[node]) then return Solution(node)
    else if Depth[node] = limit then return cutoff
    else for each successor in Expand(node, problem) do
        result Recursive-DLS(successor, problem, limit)
        if result = cutoff then cutoff_occurred? true
        else if result not = failure then return result

    if cutoff_occurred? then return cutoff else return failure
```

Iterative-Deepening Search

If a depth-limited depth-first search limited to depth l does not find the goal, try it again with limit at $l+1$. Continue this until goal is found.

Make depth-limited depth-first search complete by repeatedly applying it with greater values for the depth limit l.

Feels like breadth-first search, in that a level is fully explored before extending the search to the next level. But, unlike breadth-first search, after one level is fully explored, all nodes already expanded are thrown away and the search starts with a clear memory.

Seems very wasteful. Is it really? How many nodes are generated at the final level d? bd

How many nodes are expanded in the tree on your way to the final level, down to depth $d-1$?

$b+b2+\cdots+bd-1=O(bd-1)$

How much of a waste is it to throw away those $O(bd-1)$ nodes? Say b=10 and d=5. We are throwing away on the order of 104=1,000 nodes, regenerating them and then generating bd=105=10,000 new nodes. Regenerating those 1,000 nodes seems trivial compared to making the 10,000 new ones.

function ITERATIVE_DEEPENING_SEARCH(*problem*) **return** a solution or failure

inputs: *problem*

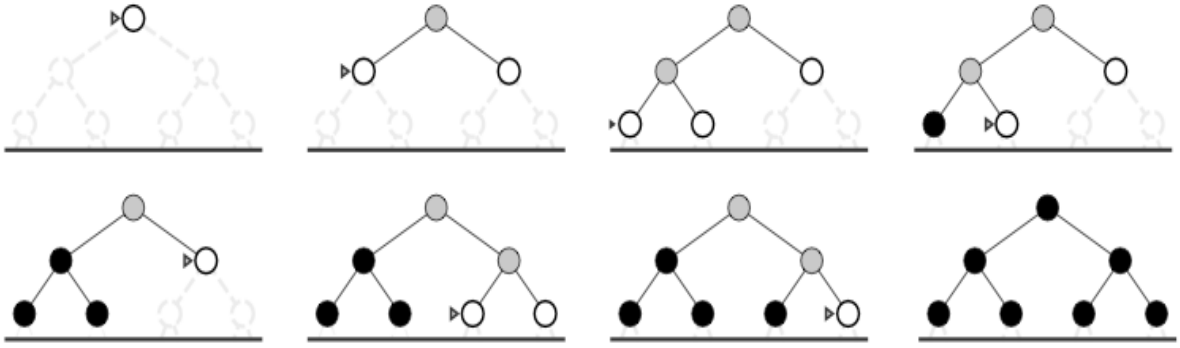
for *depth* ← 0 to ∞ **do**

result ← DEPTH-LIMITED_SEARCH(*problem*, *depth*)

if *result* ≠ *cuttoff* **then return** *result*

Example: Limit=1

If Limit=2



Bidirectional Search

i) Bidirectional Search, as the name implies, searches in two directions at the same time: one forward from the initial state and the other backward from the goal.

ii) This is usually done by expanding tree with branching factor b and the distance from start to goal is d.

iii) The search stops when searches from both directions meet in the middle.

iv) Bidirectional search is a brute-force search algorithm that requires an explicit goal state instead of simply a test for a goal condition.

v) Once the search is over, the path from the initial state is then concatenated with the inverse of the path from the goal state to form the complete solution path.

vi) **Bidirectional search** still guarantees optimal solutions.

vii) Assuring that the comparisons for identifying a common state between the two frontiers can be done in constant time per node by hashing.

Time Complexity

The time complexity of Bidirectional Search is O(b^d/2) since each search need only proceed to half the solution path. Since at least one of the searches must be breadth-first in order to find a common state, the space complexity of bidirectional search is also O(b^d/2). As a result, it is space bound in practice.

Advantages

1. The merit of bidirectional search is its speed. Sum of the time taken by two searches (forward and backward) is much less than the $O(b^d)$ complexity.
2. It requires less memory.

Disadvantages

1. Implementation of bidirectional search algorithm is difficult because additional logic must be included to decide which search tree to extend at each step.
2. One should have known the goal state in advance.
3. The algorithm must be too efficient to find the intersection of the two search trees.
4. It is not always possible to search backward through possible states.

Summary of algorithms

Criterion	**Breadth-First**		**Depth-First**	**Depth-limited**	**Iterative deepening**	**Bidirectional search**
Complete?	YES*		NO	YES, if $l \geq d$	YES	YES*
Time	b^{d+1}		b^m	b^l	b^d	$b^{d/2}$
Space	b^{d+1}		bm	bl	bd	$b^{d/2}$
Optimal?	YES*		NO	NO	YES	YES

Searching with partial Information

If the Knowledge of states or actions is incomplete, it leads to three distinct problem type:

i)sensorless problem

If an agent has no sensors at all, it could be in one of several possible initial states, and each action might therefore lead to one of several possible successor states.

ii)contingency problem

If the environment is partially observable or if action are uncertain (**adversarial**) then the agent's percepts provide new information after each action. Each possible percept defines a contingency that must be planned for. A problem is called **adversarial** if the uncertainty is caused by the actions of another agent .

iii)Exploration problems

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

Sensorless Problems

Suppose that the vaccum agent knows all the effects of its action, but has no sensors.

The eight possible states of the vacuum world shown in figure 1.16.

- No sensor
- Initial State(1,2,3,4,5,6,7,8)
- After action [Right] the state (2,4,6,8)
- After action [Suck] the state (4, 8)
- After action [Left] the state (3,7)
- After action [Suck] the state (8)
- Answer : [Right, Suck, Left, Suck] coerce the world into state 7 without any sensor

o Belief State: Such state that agent belief to be there

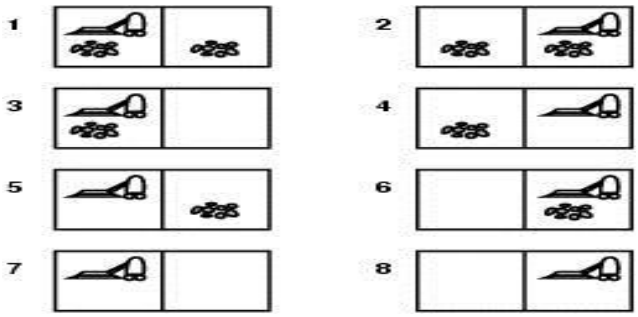


Figure 1.16 Eight possible states of Vacuum world

Partial knowledge of states and actions:

– sensorless or conformant problem

– Agent may have no idea where it is; solution (if any) is a sequence.

– contingency problem

– Percepts provide new information about current state; solution is a tree or policy; often interleave search and execution.

– If uncertainty is caused by actions of another agent: adversarial problem

– exploration problem

– When states and actions of the environment are unknown.

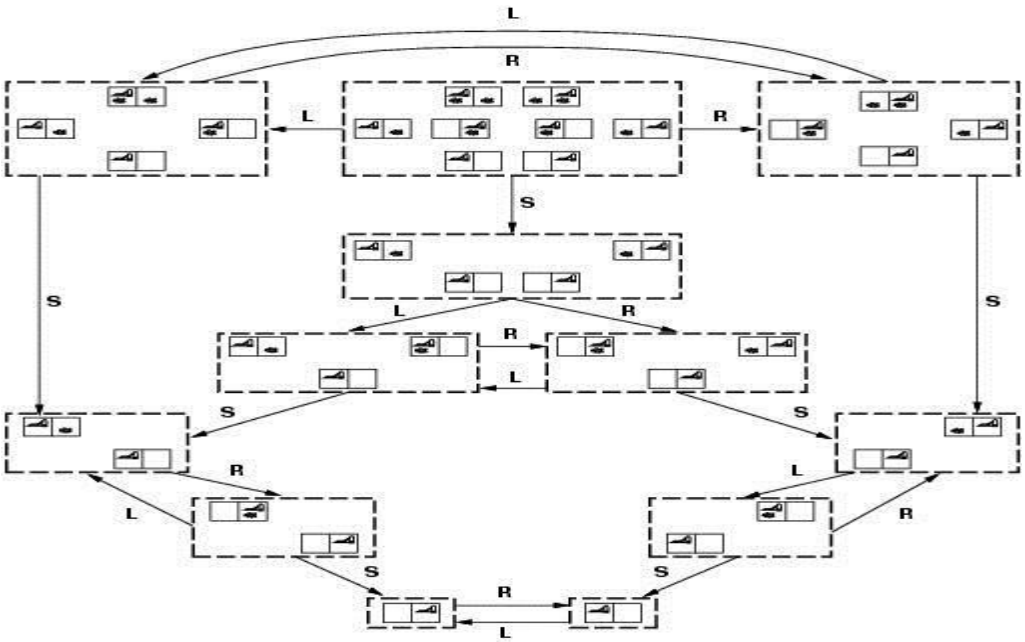


Figure 1.17 Belief State Space Search

Contingency, start in {1,3}.

Murphy's law, Suck *can* dirty a clean carpet.

Local sensing: dirt, location only.

– Percept = [L, Dirty] ={1,3}

– [Suck] = {5,7}

– [Right] ={6,8}

– [Suck] in {6}={8} (Success)

– BUT [Suck] in {8} = failure

Solution??

– Belief-state: no fixed action sequence guarantees solution

Relax requirement:

– [*Suck, Right,* if *[R ,dirty]* then *Suck*]

– Select actions based on contingencies arising during execution.

The Figure 1.17 shows the reachable belief state space, but the entire belief state space contains every possible set of physical states, 2^8=256 belief states.

Contingency problems

When the environment is such that the agent can obtain new information from its sensors after acting, the agent faces a contingency problem. The solution to a contingency problem often takes the form of a *tree,* where each branch may be selected depending on the percepts received up to that point in the tree.

For example, suppose that the agent is in the Murphy's Law world and that it has a position sensor and a local dirt sensor, but no sensor capable of detecting dirt in other squares. Thus, the percept *[L, Dirty]* means that the agent is in one of the states {I, 3}.

The agent might formulate the action sequence *[Suck, Right, Suck]*. Sucking would change the state to one of {5, 7}, and moving right would then change the state to one of {6, 8}. Executing the final *Suck* action in state 6 takes us to state 8, a goal, but executing it in state 8 might take us back to state 6 (by Murphy's Law), in which case the plan fails.

By examining the belief-state space for this version of the problem, it can easily be determined that no fixed action sequence guarantees a solution to this problem. There is, however, a solution if we don't insist on *a fixed* action sequence:

[Suck, Right, if [R,Dirty] then Suck] .

This extends the space of solutions to include the possibility of selecting actions based on contingencies arising during execution. Contingency problems ***sometime****s* allow purely sequential solutions. For example, consider a ***fully observable*** Murphy's Law world. Contingencies arise if the agent performs a *Suck* action in a clean square, because dirt might

or might not be deposited in the square. As long as the agent never does this, no contingencies arise and there is a sequential solution from every initial state