ARTIFICIAL INTELLIGENCE

VI SEMESTER CSE

UNIT-I

1.1 INTRODUCTION

1.1.1 What is AI?
1.1.2 The foundations of Artificial Intelligence.
1.1.3 The History of Artificial Intelligence
1.1.4 The state of art

1.2 INTELLIGENT AGENTS

1.2.1 Agents and environments
1.2.2 Good behavior : The concept of rationality
1.2.3 The nature of environments
1.2.4 Structure of agents

1.3 SOLVING PROBLEMS BY SEARCHING

1.3.1 Problem Solving Agents
1.3.1.1Well defined problems and solutions
1.3.2 Example problems
1.3.2.1 Toy problems
1.3.2.2 Real world problems
1.3.3 Searching for solutions
1.3.4 Uninformed search strategies
1.3.4.1 Breadth-first search
1.3.4.2 Uniform-cost search
1.3.4.3 Depth-first search
1.3.4.4 Depth limited search
1.3.4.5 Iterative-deepening depth first search

1.3.4.6 Bi-directional search
1.3.4.7 Comparing uninformed search strategies
1.3.5 Avoiding repeated states
1.3.6 Searching with partial information

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 "The exciting new effort to make computers think … machines with minds,in the full and literal sense."(Haugeland,1985)	**Systems that think rationally** "The study of mental faculties through the use of computer models." (Charniak and McDermont,1985)
Systems that act like humans The art of creating machines that perform functions that require intelligence when performed by people."(Kurzweil,1990)	**Systems that act rationally** **"**Computational intelligence is the study of the design of intelligent agents."(Poole et al.,1998)

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

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. 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

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

The Greek philosopher Aristotle was one of the first to attempt to codify "right thinking",that is irrefuatable reasoning processes. His **syllogism** provided patterns for argument structures that always yielded correct conclusions when given correct premises—for example,"Socrates is a man;all men are mortal;therefore Socrates is mortal.".
These laws of thought were supposed to govern the operation of the mind;their study initiated a field called **logic.**

(d)Acting rationally : The rational agent approach

An **agent** is something that acts. Computer agents are not mere programs ,but they are expected to have the following attributes also : (a) operating under autonomous control, (b) perceiving their environment, (c) persisting over a prolonged time period, (e) adapting to change.
A **rational agent** is one that acts so as to achieve the best outcome.

1.1.2 The foundations of Artificial Intelligence

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

Philosophy(428 B.C. – present)

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.

	Computer	Human Brain
Computational units Storage units	1 CPU,10^8 gates 10^{10} bits RAM	10^{11} neurons 10^{11} neurons

	10^{11} bits disk	10^{14} synapses
Cycle time	10^{-9} sec	10^{-3} sec
Bandwidth	10^{10} bits/sec	10^{14} bits/sec
Memory updates/sec	10^{9}	10^{14}

Table 1.1 A crude comparison of the raw computational resources available to computers(*circa* 2003) and brain. The computer's numbers have increased by at least by a factor of 10 every few years. The brain's numbers have not changed for the last 10,000 years.

Brains and digital computers perform quite different tasks and have different properties. Tablere 1.1 shows that there are 10000 times more neurons in the typical human brain than there are gates in the CPU of a typical high-end computer. Moore's Law predicts that the CPU's gate count will equal the brain's neuron count around 2020.

Psycology(1879 – present)

The origin of scientific psychology are traced back to the wok if 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 (1940-present)

For artificial intelligence to succeed, we need two things: intelligence and an artifact. The computer has been the artifact of choice.
A1 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

Control theory and Cybernetics (1948-present)

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 (1957-present)

Modem 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.**

1.1.3 The History of Artificial Intelligence

The gestation of artificial intelligence (1943-1955)

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.

The birth of artificial intelligence (1956)

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. They organized a two-month workshop at Dartmouth in the summer of 1956.
Perhaps the longest-lasting thing to come out of the workshop was an agreement to adopt McCarthy's

new name for the field: **artificial intelligence.**

Early enthusiasm, great expectations (1952-1969)

The early years of A1 were full of successes-in a limited way.

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.

At IBM, Nathaniel Rochester and his colleagues produced some of the first A1 pro-grams. Herbert Gelernter (1959) constructed the Geometry Theorem Prover, which was able to prove theorems that many students of mathematics would find quite tricky.

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.

Tom Evans's ANALOGY program (1968) solved geometric analogy problems that appear in IQ tests, such as the one in Figure 1.1

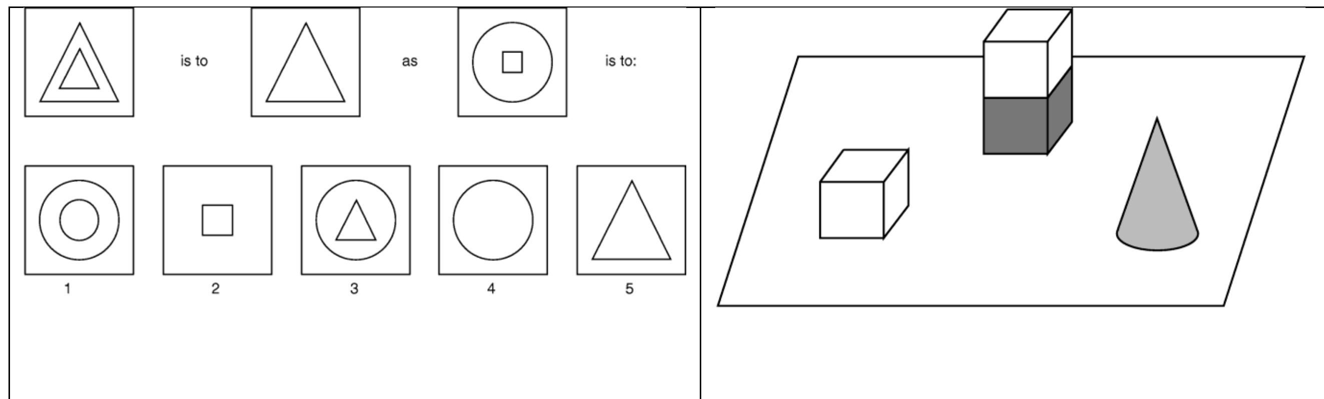


Figure 1.1 The Tom Evan's ANALOGY program could solve geometric analogy problems as shown.

A dose of reality (1966-1973)

From the beginning, AI researchers were not shy about making predictions of their coming successes. The following statement by Herbert Simon in 1957 is often quoted:

"It is not my aim to surprise or shock you-but the simplest way I can summarize is to say that there are now in the world machines that think, that learn and that create. Moreover, their ability to do these things is going to increase rapidly until-in a visible future-the range of problems they can handle will be coextensive with the range to which the human mind has been applied.

Knowledge-based systems: The key to power? (1969-1979)

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.

A1 becomes an industry (1980-present)

In 1981, the Japanese announced the "Fifth Generation" project, a 10-year plan to build intelligent computers running Prolog. Overall, the A1 industry boomed from a few million dollars in 1980 to billions of dollars in 1988.

The return of neural networks (1986-present)

Psychologists including David Rumelhart and Geoff Hinton continued the study of neural-net models of memory.

A1 becomes a science (1987-present)

In recent years, approaches based on **hidden Markov models** (HMMs) have come to dominate the area. Speech technology and the related field of handwritten character recognition are already making the transition to widespread industrial and consumer applications.
The **Bayesian network** formalism was invented to allow efficient representation of, and rigorous reasoning with, uncertain knowledge.

The emergence of intelligent agents (1995-present)

One of the most important environments for intelligent agents is the Internet.

1.1.4 The state of art

What can A1 do today?

Autonomous planning and scheduling: A hundred million miles from Earth, NASA's Remote Agent program became the first on-board autonomous planning program to control the scheduling of operations for a spacecraft (Jonsson *et* al., 2000). Remote Agent generated plans from high-level goals specified from the ground, and it monitored the operation of the spacecraft as the plans were executed-detecting, diagnosing, and recovering from problems as they occurred.
Game playing: IBM's Deep Blue became the first computer program to defeat the world champion in a chess match when it bested Garry Kasparov by a score of 3.5 to 2.5 in an exhibition match (Goodman and Keene, 1997).
Autonomous control: The ALVINN computer vision system was trained to steer a car to keep it following a lane. It was placed in CMU's NAVLAB computer-controlled minivan and used to navigate across the United States-for 2850 miles it was in control of steering the vehicle 98% of the time.
Diagnosis: Medical diagnosis programs based on probabilistic analysis have been able to perform at the level of an expert physician in several areas of medicine.
Logistics Planning: During the Persian Gulf crisis of 1991, U.S. forces deployed a Dynamic Analysis and Replanning Tool, DART (Cross and Walker, 1994), to do automated logistics planning and scheduling for transportation. This involved up to 50,000 vehicles, cargo, and people at a time, and had to account for starting points, destinations, routes, and conflict resolution among all parameters. The AI planning techniques allowed a plan to be generated in hours that would have taken weeks with older methods. The Defense Advanced Research Project Agency (DARPA) stated that this single application more than paid back DARPA's 30-year investment in AI.
Robotics: Many surgeons now use robot assistants in microsurgery. HipNav (DiGioia *et* al., 1996) is a system that uses computer vision techniques to create a three-dimensional model of a patient's internal anatomy and then uses robotic control to guide the insertion of a

hip replacement prosthesis.
Language understanding and problem solving: PROVERB (Littman *et al.,* 1999) is a computer program that solves crossword puzzles better than most humans, using constraints on possible word fillers, a large database of past puzzles, and a variety of information sources including dictionaries and online databases such as a list of movies and the actors that appear in them.

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.

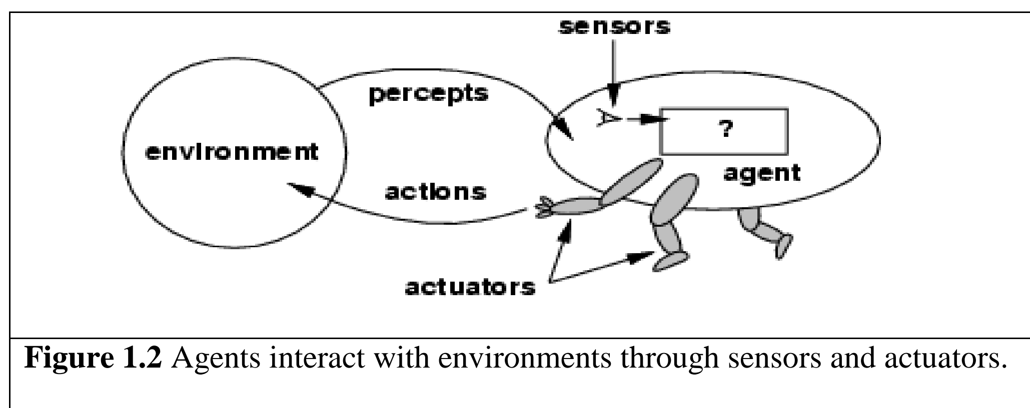


Figure 1.2 Agents interact with environments through sensors and actuators.

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.

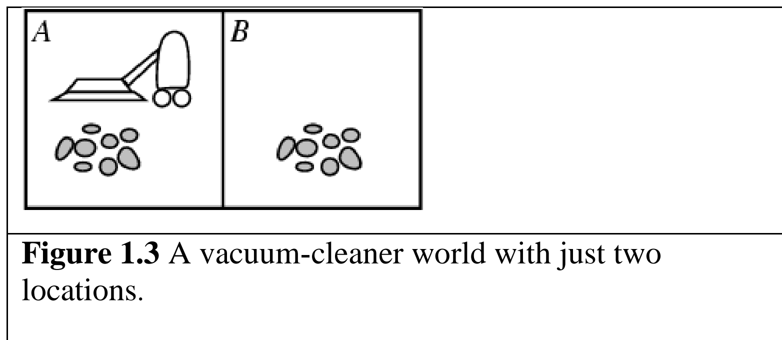


Figure 1.3 A vacuum-cleaner world with just two locations.

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
```

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 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,	Roads,other traffic,pedestrians,	Steering,accelerator, brake,	Cameras,sonar, Speedometer,GPS,

	maximize profits	customers	Signal,horn,display	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.**
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. A task environment is effectively fully observable if the sensors detect all aspects that are *relevant* to the choice of action;
An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simplly missing from the sensor data.

Deterministic vs. **stochastic.**

If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic; otherwise, it is stochastic.

Episodic vs. **sequential**

In an **episodic task environment**, the agent's experience is divided into atomic episodes. Each episode consists of the agent perceiving and then performing a single action. Crucially, the next episode does not depend on the actions taken in previous episodes.
For example, an agent that has to spot defective parts on an assembly line bases each decision on the current part, regardless of previous decisions;
In **sequential environments**, on the other hand, the current decision could affect all future decisions. Chess and taxi driving are sequential:

Discrete vs. **continuous.**

The discrete/continuous distinction can be applied to the *state* of the environment, to the way *time* is handled, and to the *percepts* and *actions* of the agent. For example, a discrete-state environment such as a chess game has a finite number of distinct states. Chess also has a discrete set of percepts and actions. Taxi driving is a continuous-state and continuous-time problem: the speed and location of the taxi and of the other vehicles sweep through a range of continuous values and do so smoothly over time. Taxi-driving actions are also continuous (steering angles, etc.).

Single agent vs. **multiagent.**

An agent solving a crossword puzzle by itself is clearly in a single-agent environment, whereas an agent playing chess is in a two-agent environment.
As one might expect, the hardest case is *partially observable, stochastic, sequential, dynamic, continuous,* and *multiagent.*

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.

Agent programs

The agent programs all have the same skeleton: they take the current percept as input from the sensors and return an action to the actuatom6 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

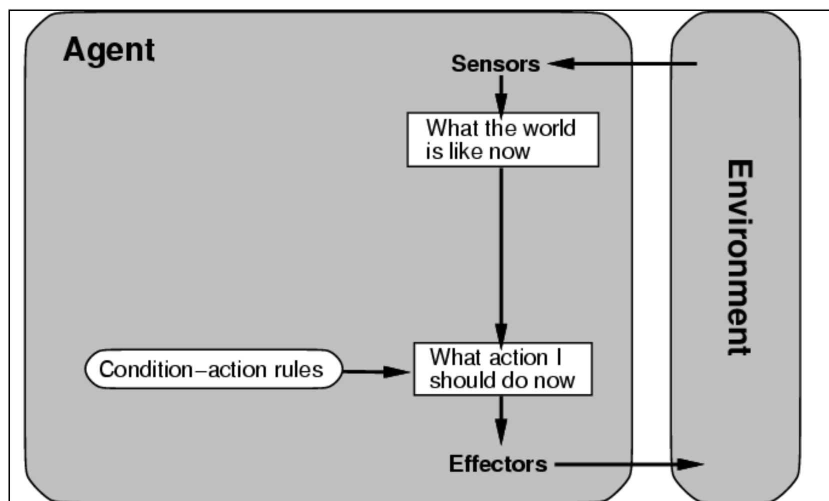


Figure 1.9 Schematic diagram of a simple reflex agent.

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.

Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program. First, we need some information about how the world evolves independently of the agent-for example, that an overtaking car generally will be closer behind than it was a moment ago. Second, we need some information about how the agent's own actions affect the world-for example, that when the agent turns the steering wheel clockwise, the car turns to the right or that after driving for five minutes northbound on the freeway one is usually about five miles north of where one was five minutes ago. This knowledge about "how the world working - whether implemented in simple Boolean circuits or in complete scientific theories-is called a **model** of the world. An agent that uses such a MODEL-BASED model is called a **model-based agent.**

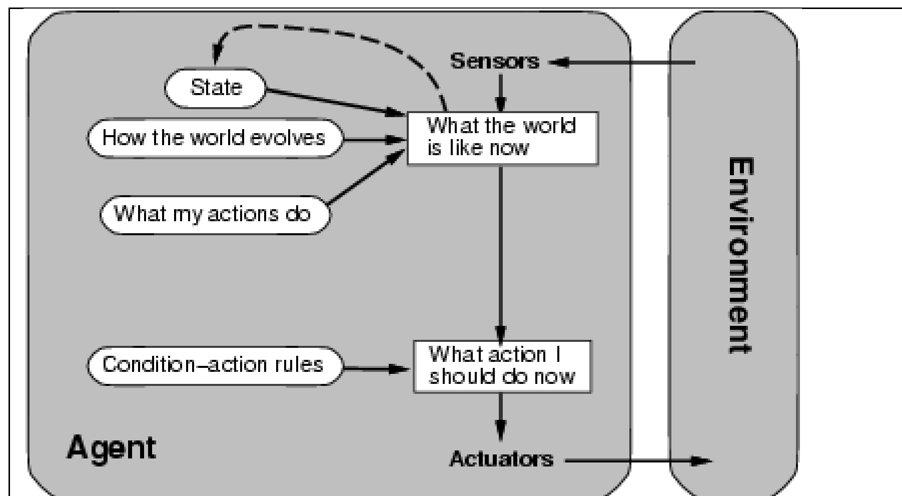


Figure 1.12 A model based reflex agent

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

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. In other words, as well as a current state description, the agent needs some sort of **goal** information that describes situations that are desirable-for example, being at the passenger's destination. The agent program can combine this with information about the results of possible actions (the same information as was used to update internal state in the reflex agent) in order to choose actions that achieve the goal. Figure 1.13 shows the goal-based agent's structure.

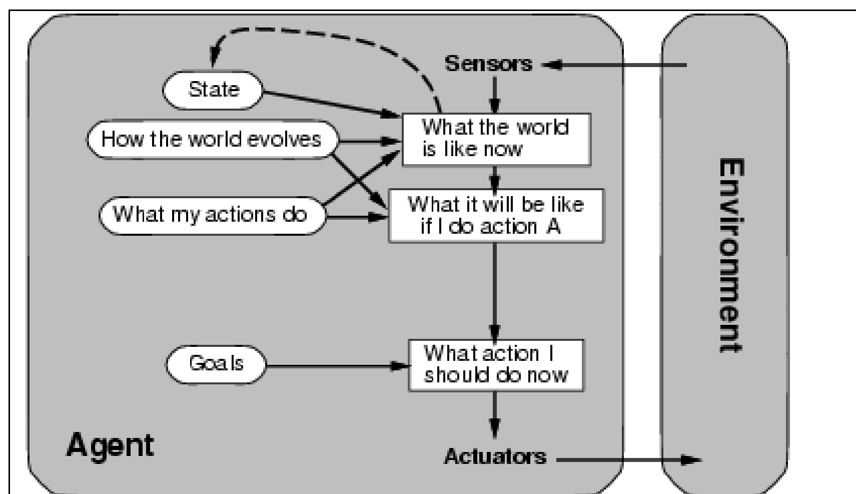


Figure 1.14 A goal based agent

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. Because "happy" does not sound very scientific, the customary terminology is to say that if one world state is preferred to another, then it has higher **utility** for the agent.

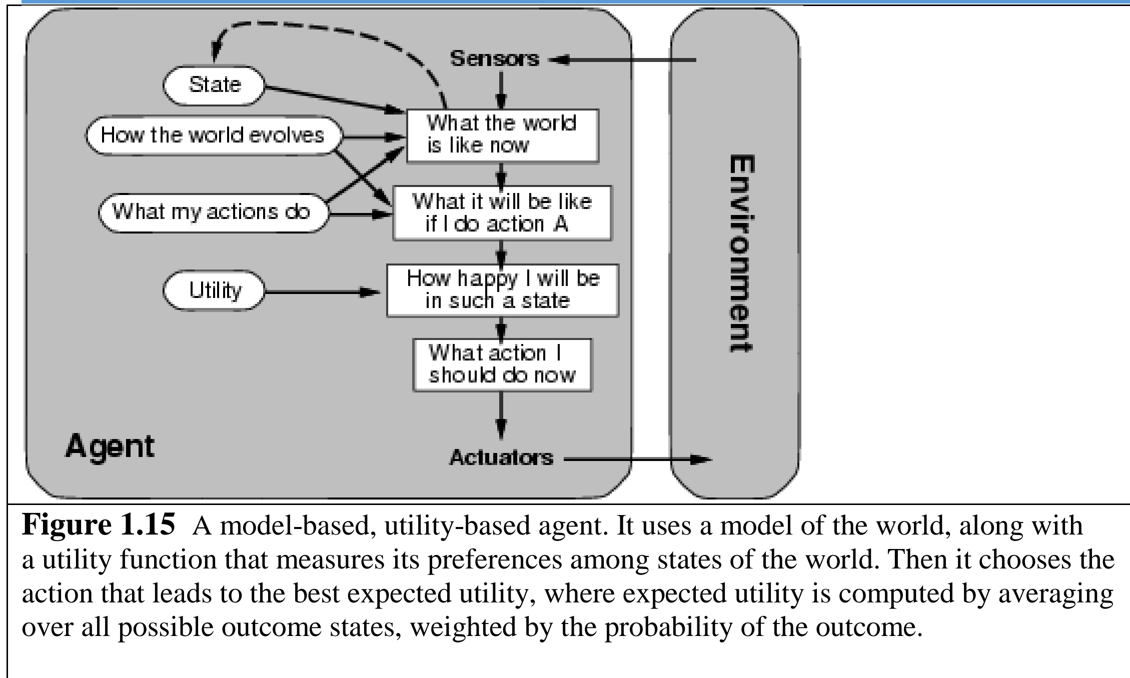


Figure 1.15 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome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

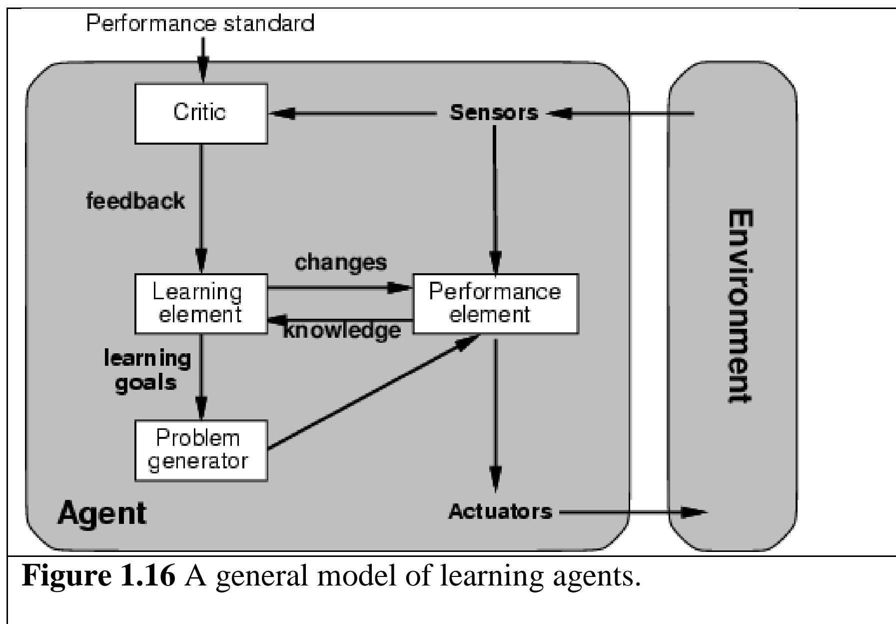


Figure 1.16 A general model of learning agents.

- 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

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.

Goal formulation,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.

Problem formulation is the process of deciding what actions and states to consider given a goal.

Example: Route finding problem
Referring to figure 1.19
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"
implicit, e.g., NoDirt(x)
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.

Figure 1.17 Goal formulation and problem formulation

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.**

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 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.

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.
- Thr **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 1.18. 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.

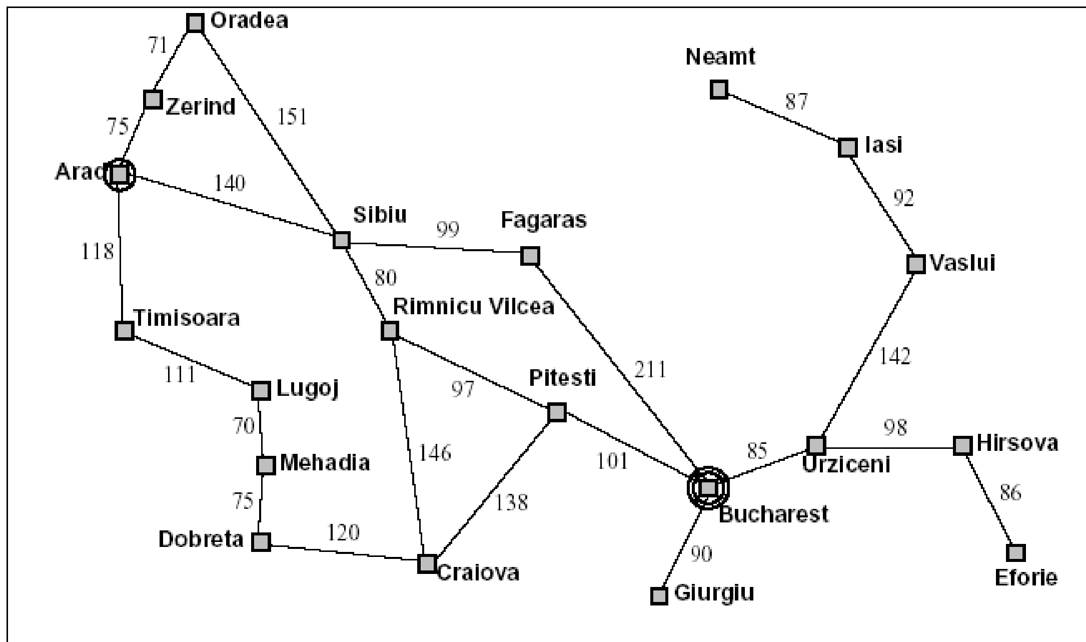


Figure 1.19 A simplified Road Map of part of Romania

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 2.3
- **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

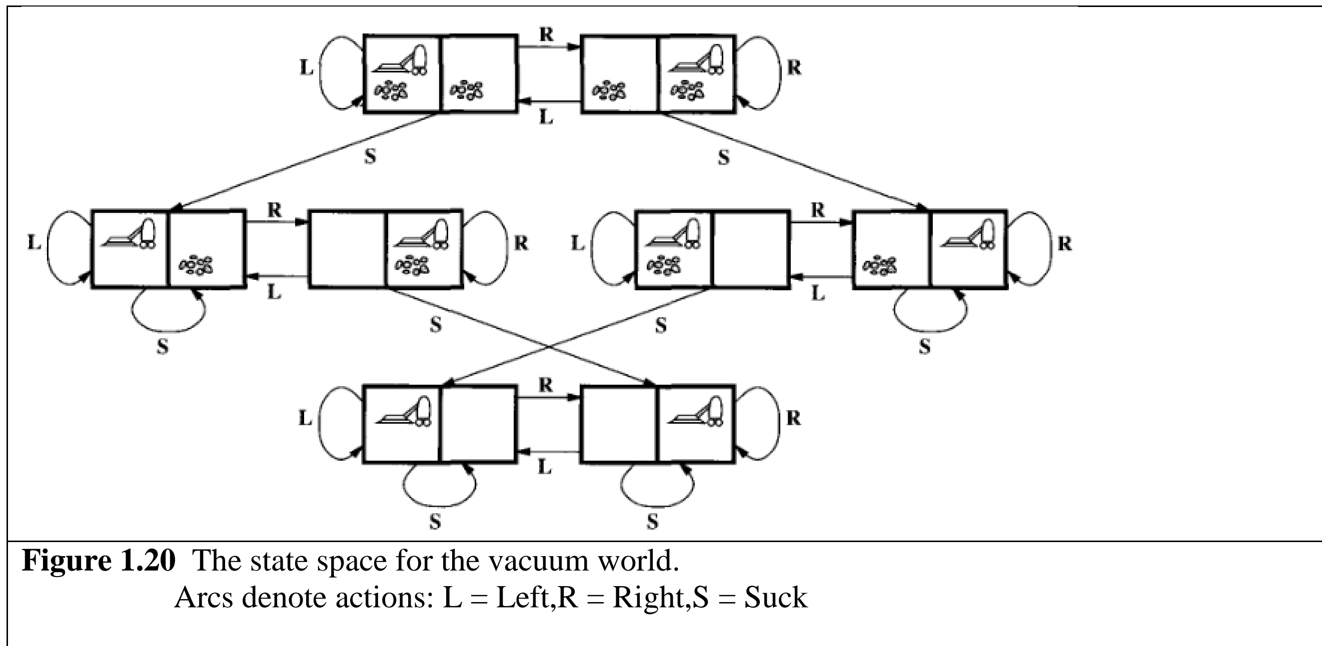


Figure 1.20 The state space for the vacuum world.
Arcs denote actions: L = Left,R = Right,S = Suck

The 8-puzzle

An 8-puzzle consists of a 3x3 board with eight numbered tiles and a blank space. A tile adjacent to the balank space can slide into the space. The object is to reach the goal state ,as shown in figure 2.4

Example: The 8-puzzle

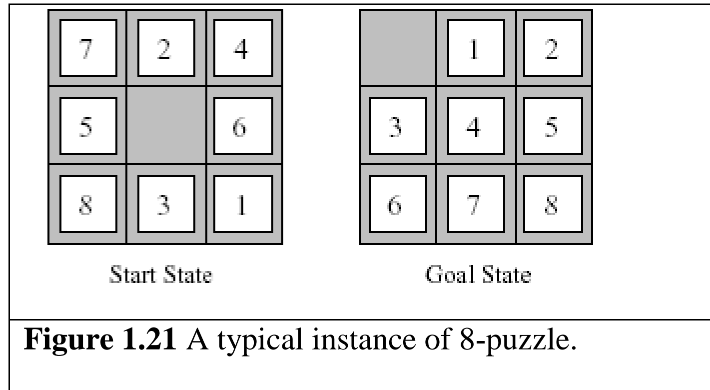


Figure 1.21 A typical instance of 8-puzzle.

The problem formulation is as follows :

- **States** : A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state** : Any state can be designated as the initial state. It can be noted that any given goal can be reached from exactly half of the possible initial states.
- **Successor function** : This generates the legal states that result from trying the four actions(blank moves Left,Right,Up or down).
- **Goal Test** : This checks whether the state matches the goal configuration shown in figure 2.4.(Other goal configurations are possible)
- **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 10^{25} states ,and random instances are still quite difficult to solve optimally with current machines and algorithms.

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 2.5 shows an attempted solution that fails: the queen in the right most column is attacked by the queen at the top left.
An **Incremental formulation** involves operators 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 move them around.
In either case the path cost is of no interest because only the final state counts.

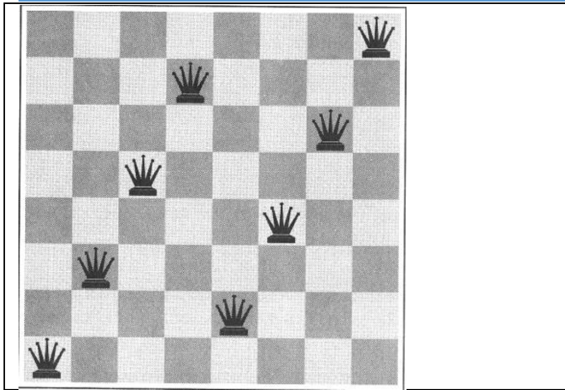


Figure 1.22 8-queens problem

The first incremental formulation one might try is the following :

- **States** : Any arrangement of 0 to 8 queens on board is a state.
- **Initial state** : No queen on the board.
- **Successor function** : Add a queen to any empty square.
- **Goal Test** : 8 queens are on the board,none attacked.

In this formulation,we have 64.63…57 = 3 x 10^{14} possible sequences to investigate.
A better formulation would prohibit placing a queen in any square that is already attacked.
:

- **States** : Arrangements of n queens (0 <= n < = 8) ,one per column in the left most columns ,with no queen attacking another are states.
- **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 10^{14} to just 2057,and solutions are easy to find.
For the 100 queens the initial formulation has roughly 10^{400} states whereas the improved formulation has about 10^{52} states. This is a huge reduction,but the improved state space is still too big for the algorithms to handle.

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.
- **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.
- **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 SEARCHING FOR SOLUTIONS

SEARCH TREE

Having formulated some **problems**,we now need to **solve** them. This is done by a **search** through the **state space**. 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 1.23 shows some of the expansions in the search tree for finding a route from Arad to Bucharest.

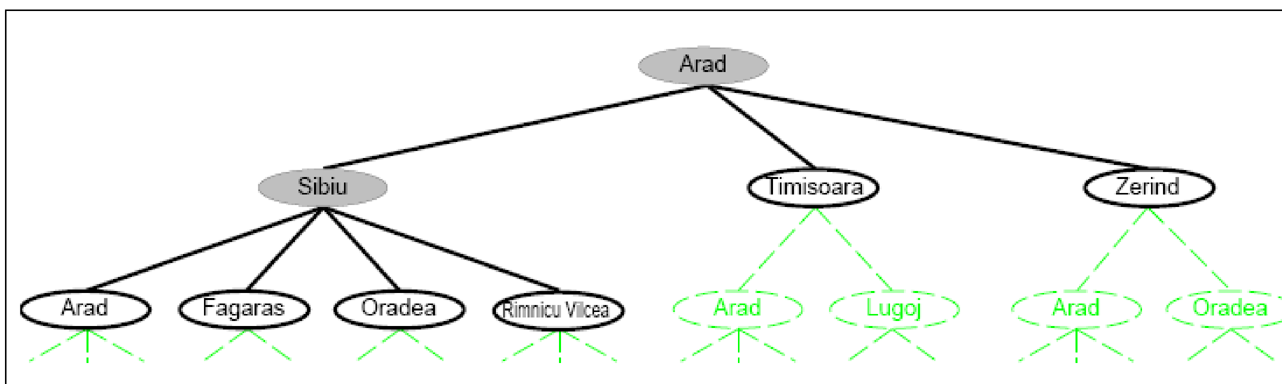


Figure 1.23 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded.; nodes that have been generated but not yet expanded are outlined in bold;nodes that have not yet been generated are shown in faint dashed line

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.

Search strategy . The general tree-search algorithm is described informally in Figure 1.24

.

Tree Search

function TREE-SEARCH(*problem*, *strategy*) **returns** a solution, or failure
  initialize the search tree using the initial state of *problem*
  loop do
    if there are no candidates for expansion **then return** failure
    choose a leaf node for expansion according to *strategy*
    if the node contains a goal state **then return** the corresponding solution
    else expand the node and add the resulting nodes to the search tree

Figure 1.24 An informal description of the general tree-search algorithm

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 :

- STATE : a state in the state space to which the node corresponds;
- PARENT-NODE : the node in the search tree that generated this node;
- ACTION : the action that was applied to the parent to generate the node;
- PATH-COST :the cost,denoted by g(n),of the path from initial state to the node,as indicated by the parent pointers; and
- 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. A state corresponds to configuration of the world.

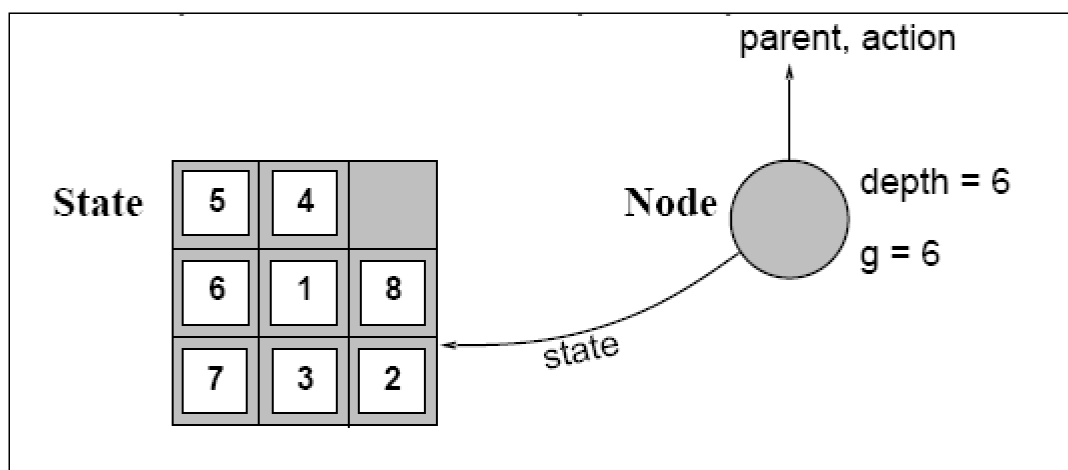


Figure 1.25 Nodes are data structures from which the search tree is constructed. Each has a parent,a state, Arrows point from child to parent.

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,that is,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.**

The general tree search algorithm is shown in Figure 2.9

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure

  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)
-------------------------------------------------------------------
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
      STATE[s] ← result
      PARENT-NODE[s] ← node
      ACTION[s] ← action
      PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
      DEPTH[s] ← DEPTH[node] + 1
      add s to successors
  return successors
```

Figure 1.26 The general Tree search algorithm

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

- **MAKE-QUEUE(element,…)** creates a queue with the given element(s).
- **EMPTY?(queue)** returns true only if there are no more elements in the queue.
- **FIRST(queue)** returns FIRST(queue) and removes it from the queue.
- **INSERT(element,queue)** inserts an element into the queue and returns the resulting queue.
- **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 :

- **Completeness** : Is the algorithm guaranteed to find a solution when there is one?
- **Optimality** : Does the strategy find the optimal solution
- **Time complexity** : How long does it take to find a solution?
- **Space complexity** : How much memory is needed to perform the search?

1.3.4 UNINFORMED SEARCH STRATGES

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.

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

1.3.4.1 Breadth-first search

Breadth-first search is a simple strategy in which the root node is expanded first,then all successors of the root node are expanded next,then their successors,and so on. In general,all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Breath-first-search is implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out(FIFO) queue,assuring that the nodes that are visited first will be expanded first. In otherwards,calling TREE-SEARCH(problem,FIFO-QUEUE()) results in breadth-first-search. The FIFO queue puts all newly generated successors at the end of the queue,which means that Shallow nodes are expanded before deeper nodes.

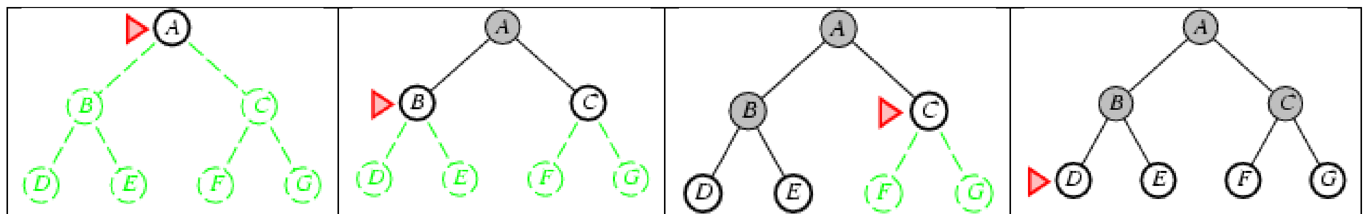


Figure 1.27 Breadth-first search on a simple binary tree. At each stage ,the node to be expanded next is indicated by a marker.

Properties of breadth-first-search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? No, unless step costs are constant

Space is the big problem; can easily generate nodes at 100MB/sec so 24hrs = 8640GB.

Figure 1.28 Breadth-first-search properties

Time and Memory Requirements for BFS – $O(b^{d+1})$

Example:
- b = 10
- 10000 nodes/second
- each node requires 1000 bytes of storage

Depth	Nodes	Time	Memory
2	1100	.11 sec	1 meg
4	111,100	11 sec	106 meg
6	10^7	19 min	10 gig
8	10^9	31 hrs	1 tera
10	10^{11}	129 days	101 tera
12	10^{13}	35 yrs	10 peta
14	10^{15}	3523 yrs	1 exa

Figure 1.29 Time and memory requirements for breadth-first-search. The numbers shown assume branch factor of b = 10 ; 10,000 nodes/second; 1000 bytes/node

Time complexity for BFS

Assume every state has b successors. The root of the search tree generates b nodes at the first level,each of which generates b more nodes,for a total of b^2 at the second level. Each of these generates b more nodes,yielding b^3 nodes at the third level,and so on. Now suppose,that the solution is at depth d. In the worst case,we would expand all but the last node at level d,generating $b^{d+1} - b$ nodes at level d+1.

Then the total number of nodes generated is

$b + b^2 + b^3 + \ldots + b^d + (b^{d+1} + b) = O(b^{d+1})$.

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

Instead of expanding the shallowest node,**uniform-cost search** expands the node n with the lowest path cost. uniform-cost search does not care about the number of steps a path has,but only about their total cost.

Expand least-cost unexpanded node

Implementation:

$fringe$ = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal?? Yes—nodes expanded in increasing order of $g(n)$

Figure 1.30 Properties of Uniform-cost-search

2.5.1.3 DEPTH-FIRST-SEARCH

Depth-first-search always expands the deepest node in the current fringe of the search tree. The progress of the search is illustrated in figure 1.31. The search proceeds immediately to the deepest level of the search tree,where the nodes have no successors. As those nodes are expanded,they are dropped from the fringe,so then the search “backs up” to the next shallowest node that still has unexplored successors.

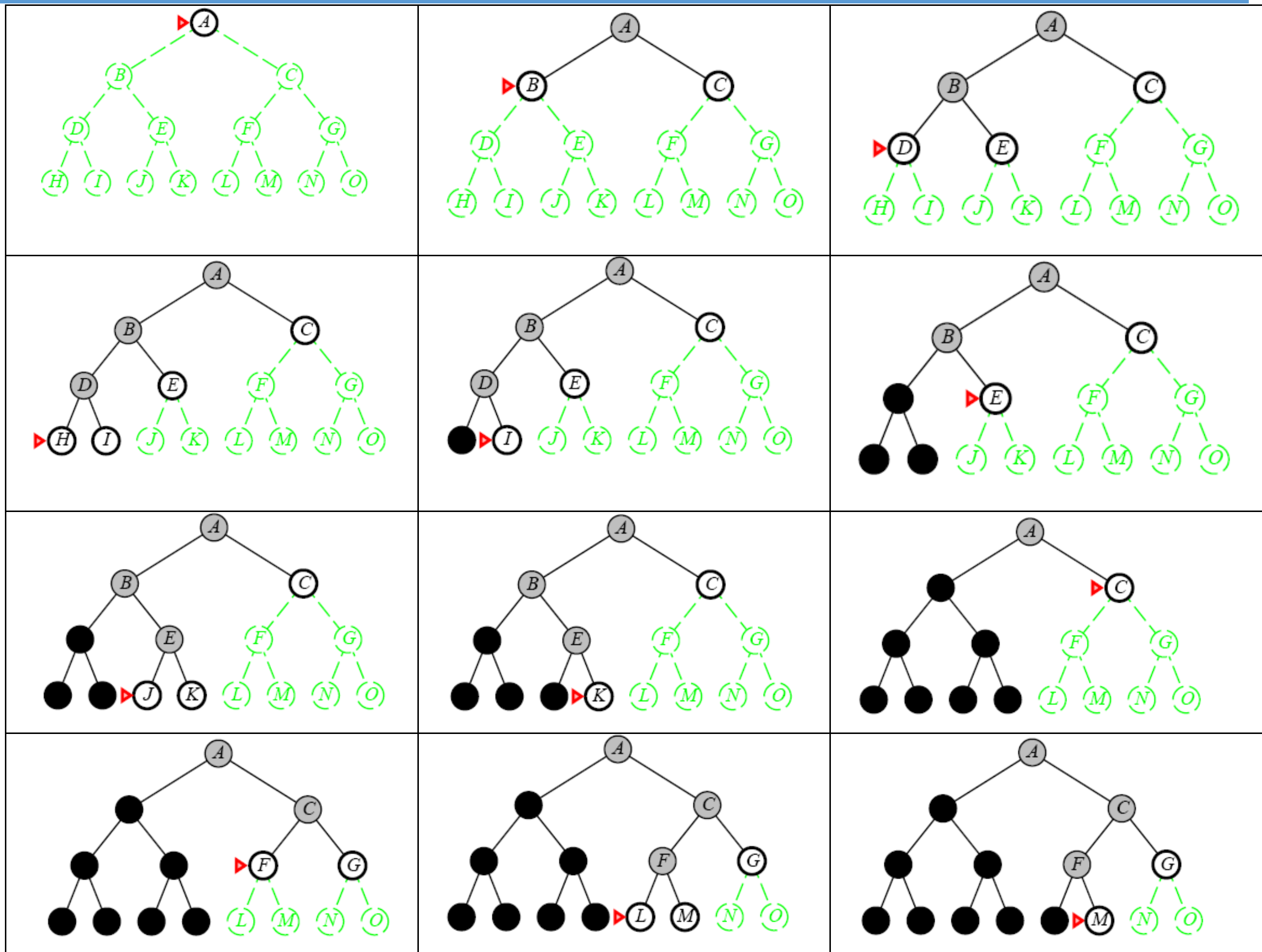


Figure 1.31 Depth-first-search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from the memory;these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

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).
For a state space with a branching factor b and maximum depth m,depth-first-search requires storage of only bm + 1 nodes.

Using the same assumptions as Figure 2.11,and assuming that nodes at the same depth as the goal node have no successors,we find the depth-first-search would require 118 kilobytes instead of 10 petabytes,a factor of 10 billion times less space.

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

The problem of unbounded trees can be alleviated by supplying depth-first-search with a pre-determined depth limit l.That is,nodes at depth l are treated as if they have no successors. This approach is called **depth-limited-search**. The depth limit soves the infinite path problem.
Depth limited search will be nonoptimal if we choose l > d. Its time complexity is $O(b^l)$ and its space compleiy is O(bl). Depth-first-search can be viewed as a special case of depth-limited search with l = oo
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 oocan 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.
Depth-limited-search can be implemented as a simple modification to the general tree-search algorithm or to the recursive depth-first-search algorithm. The pseudocode for recursive depth-limited-search is shown in Figure 1.32.
It can be noted that the above algorithm can terminate with two kinds of failure : the standard *failure* value indicates no solution; the *cutoff* value indicates no solution within the depth limit.
Depth-limited search = depth-first search with depth limit l,
returns cut off if any path is cut off by depth limit

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

Figure 1.32 Recursive implementation of Depth-limited-search:

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.

Iterative deepening combines the benefits of depth-first and breadth-first-search

Like depth-first-search,its memory requirements are modest;O(bd) to be precise.

Like Breadth-first-search,it is complete when the branching factor is finite and optimal when the path cost is a non decreasing function of the depth of the node.

Figure 2.15 shows the four iterations of ITERATIVE-DEEPENING_SEARCH on a binary search tree,where the solution is found on the fourth iteration.

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

Figure 1.33 The **iterative deepening search algorithm** ,which repeatedly applies depth-limited-search with increasing limits. It terminates when a solution is found or if the depth limited search resturns *failure*,meaning that no solution exists.

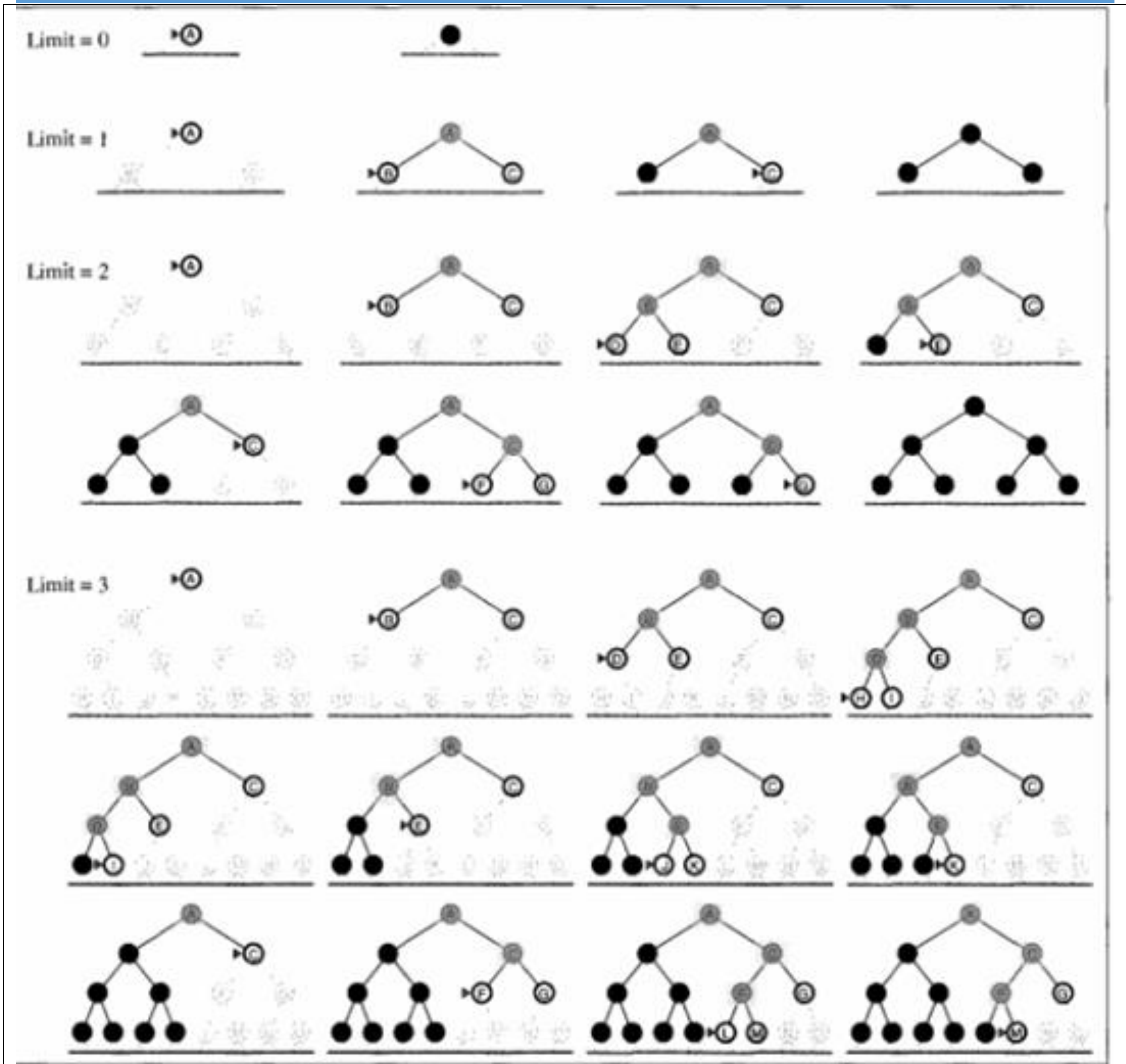


Figure 1.34 Four iterations of iterative deepening search on a binary tree

Iterative search is not as wasteful as it might seem

Iterative deepening search

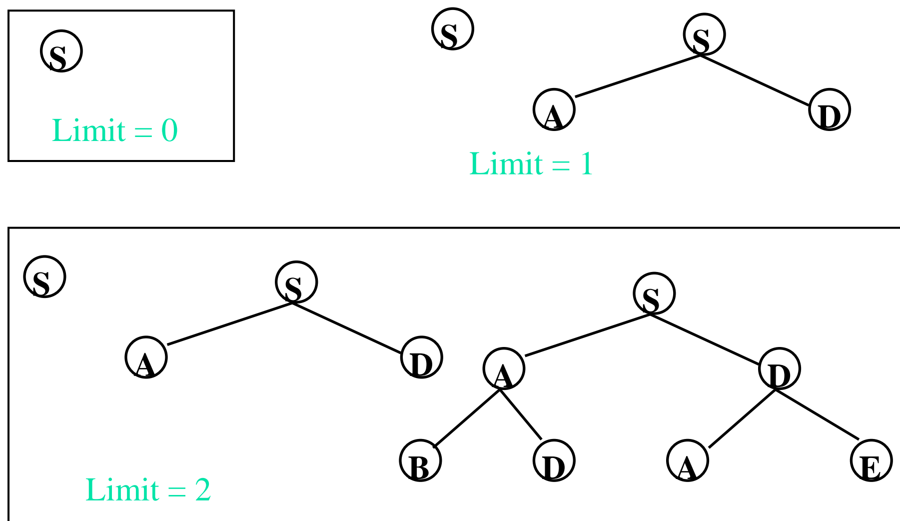


Figure 1.35

Iterative search is not as wasteful as it might seem

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 he initial state and
the other backward from the goal,
stopping when the two searches meet in the middle (Figure 1.37)

The motivation is that $b^{d/2} + b^{d/2}$ much less than ,or in the figure ,the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.

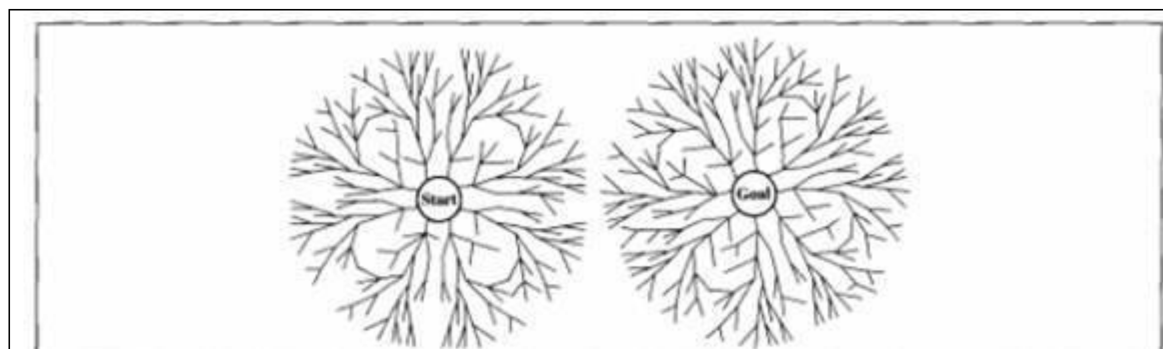


Figure 1.37 A schematic view of a bidirectional search that is about to succeed,when a Branch from the Start node meets a Branch from the goal node.

1.3.4.7 Comparing Uninformed Search Strategies

Figure 1.38 compares search strategies in terms of the four evaluation criteria .

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

Figure 1.38 Evaluation of search strategies,b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs >= E for positive E; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

1.3.5 AVOIDING REPEATED STATES

In searching,time is wasted by expanding states that have already been encountered and expanded before. For some problems repeated states are unavoidable. The search trees for these problems are infinite. If we prune some of the repeated states,we can cut the search tree down to

finite size. Considering search tree upto a fixed depth,eliminating repeated states yields an exponential reduction in search cost.

Repeated states ,can cause a solvable problem to become unsolvable if the algorithm does not detect them.

Repeated states can be the source of great inefficiency: identical sub trees will be explored many times!

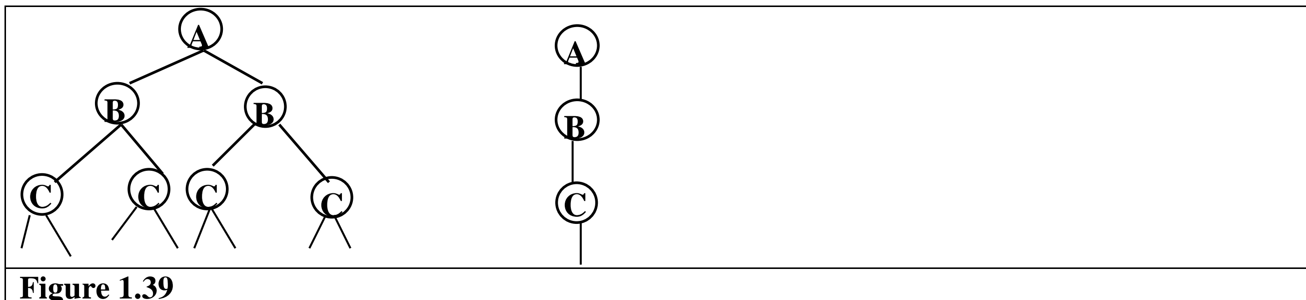


Figure 1.39

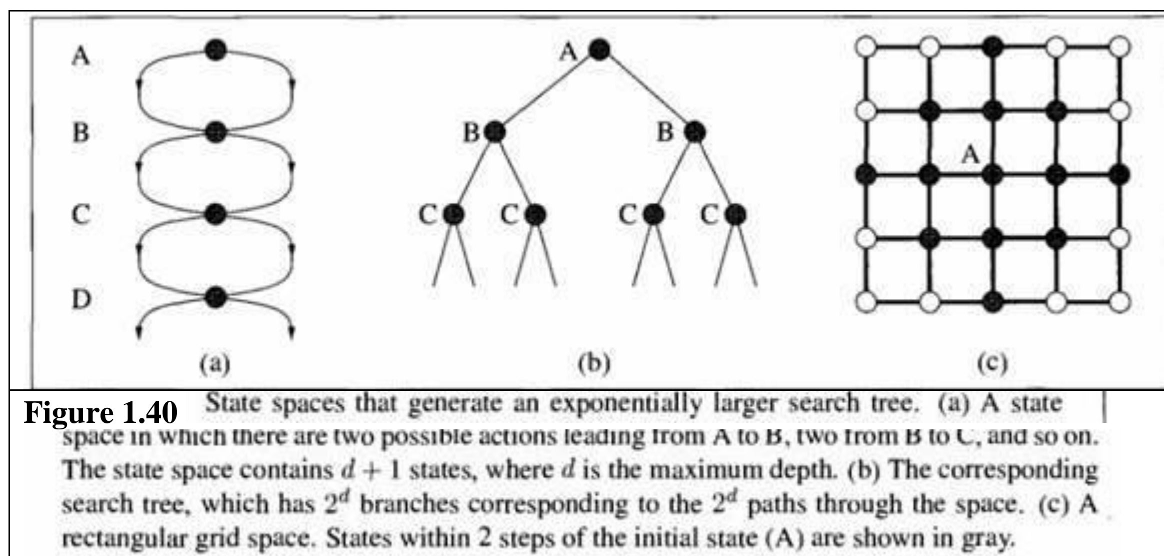


Figure 1.40 State spaces that generate an exponentially larger search tree. (a) A state space in which there are two possible actions leading from A to B, two from B to C, and so on. The state space contains $d + 1$ states, where d is the maximum depth. (b) The corresponding search tree, which has 2^d branches corresponding to the 2^d paths through the space. (c) A rectangular grid space. States within 2 steps of the initial state (A) are shown in gray.

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
  closed ← an empty set
  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 node
     if STATE[node] is not in closed then
        add STATE[node] to closed
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
  end
```

Figure 1.41 The General graph search algorithm. The set closed can be implemented with a hash table to allow efficient checking for repeated states.

Do not return to the previous state.
• Do not create paths with cycles.
• Do not generate the same state twice.
- Store states in a hash table.

- Check for repeated states.

  - Using more memory in order to check repeated state
    - *Algorithms that forget their history are doomed to repeat it.*
    - Maintain Close-List beside Open-List(fringe)

Strategies for avoiding repeated states
We can modify the general TREE-SEARCH algorithm to include the data structure called the **closed list** ,which stores every expanded node. The fringe of unexpanded nodes is called the **open list.**
If the current node matches a node on the closed list,it is discarded instead of being expanded.
The new algorithm is called GRAPH-SEARCH and much more efficient than TREE-SEARCH. The worst case time and space requirements may be much smaller than $O(b^d)$.

1.3.6 SEARCHING WITH PARTIAL INFORMATION

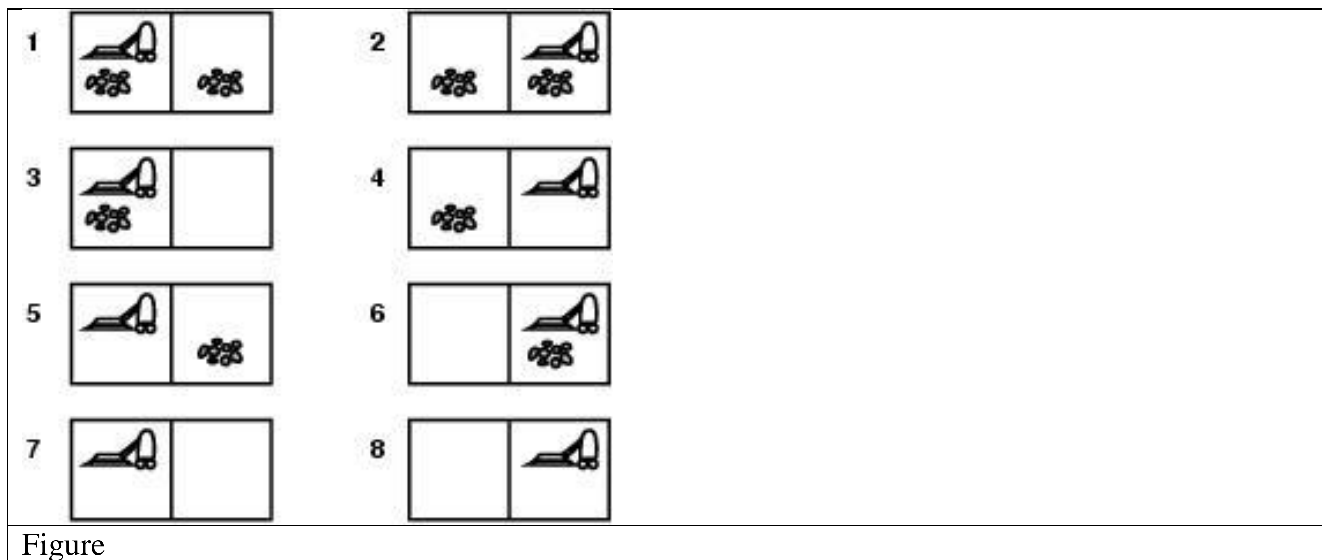
- Different types of incompleteness lead to three distinct problem types:
  - **Sensorless problems** (conformant): If the agent has no sensors at all
  - **Contingency problem**: if the environment if partially observable or if action are uncertain (adversarial)
  - **Exploration problems**: When the states and actions of the environment are unknown.

  - 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)

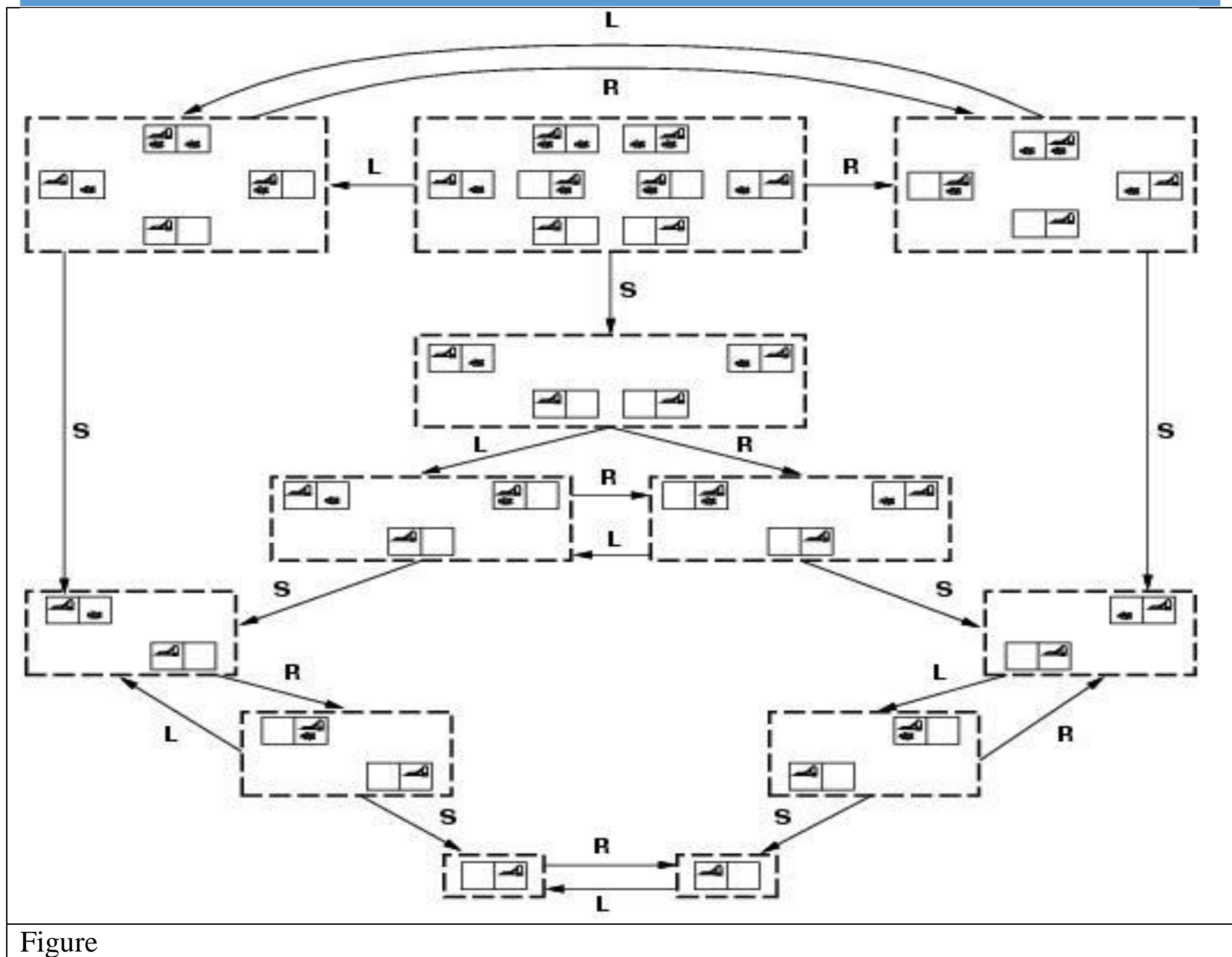
- Answer : [Right,Suck,Left,Suck] coerce the world into state 7 without any sensor
- Belief State: Such state that agent belief to be there

(SLIDE 7) 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



Figure

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.

Time and space complexity are always considered with respect to some measure of the problem difficulty. In theoretical computer science ,the typical measure is the size of the state space.

In AI,where the graph is represented implicitly by the initial state and successor function,the complexity is expressed in terms of three quantities:
b,the **branching factor** or maximum number of successors of any node;
d,the **depth** of the **shallowest goal node**; and
m,the **maximum length** of any path in the state space.

Search-cost - typically depends upon the time complexity but can also include the term for memory usage.
Total–cost – It combines the search-cost and the path cost of the solution found.

UNIT-II
(2) SEARCHING TECHNIQUES

2.1 INFORMED SEARCH AND EXPLORATION

2.1.1 Informed(Heuristic) Search Strategies
2.1.2 Heuristic Functions
2.1.3 Local Search Algorithms and Optimization Problems
2.1.4 Local Search in Continuous Spaces
2.1.5 Online Search Agents and Unknown Environments

2.2 CONSTRAINT SATISFACTION PROBLEMS(CSP)

2.2.1 Constraint Satisfaction Problems
2.2.2 Backtracking Search for CSPs
2.2.3 The Structure of Problems

2.3 ADVERSARIAL SEARCH

2.3.1 Games
2.3.2 Optimal Decisions in Games
2.3.3 Alpha-Beta Pruning
2.3.4 Imperfect ,Real-time Decisions
2.3.5 Games that include Element of Chance

2.1 INFORMED SEARCH AND EXPLORATION

2.1.1 Informed(Heuristic) Search Strategies

Informed search strategy is one that uses problem-specific knowledge beyond the definition of the problem itself. It can find solutions more efficiently than uninformed strategy.

Best-first search

Best-first search is an instance of general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function** f(n). The node with lowest evaluation is selected for expansion,because the evaluation measures the distance to the goal.
This can be implemented using a priority-queue,a data structure that will maintain the fringe in ascending order of f-values.

2.1.2. Heuristic functions

A **heuristic function** or simply a **heuristic** is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search.

The key component of Best-first search algorithm is a **heuristic function**,denoted by h(n):

h(n) = extimated cost of the **cheapest path** from node n to a **goal node**.

For example,in Romania,one might estimate the cost of the cheapest path from Arad to Bucharest via a **straight-line distance** from Arad to Bucharest(Figure 2.1).
Heuristic function are the most common form in which additional knowledge is imparted to the search algorithm.

Greedy Best-first search

Greedy best-first search tries to expand the node that is closest to the goal,on the grounds that this is likely to a solution quickly.
It evaluates the nodes by using the heuristic function f(n) = h(n).
Taking the example of **Route-finding problems** in Romania , the goal is to reach Bucharest starting from the city Arad. We need to know the straight-line distances to Bucharest from various cities as shown in Figure 2.1. For example, the initial state is In(Arad) ,and the straight line distance heuristic h_{SLD}(In(Arad)) is found to be 366.
Using the **straight-line distance** heuristic $\mathbf{h_{SLD}}$ **,**the goal state can be reached faster.

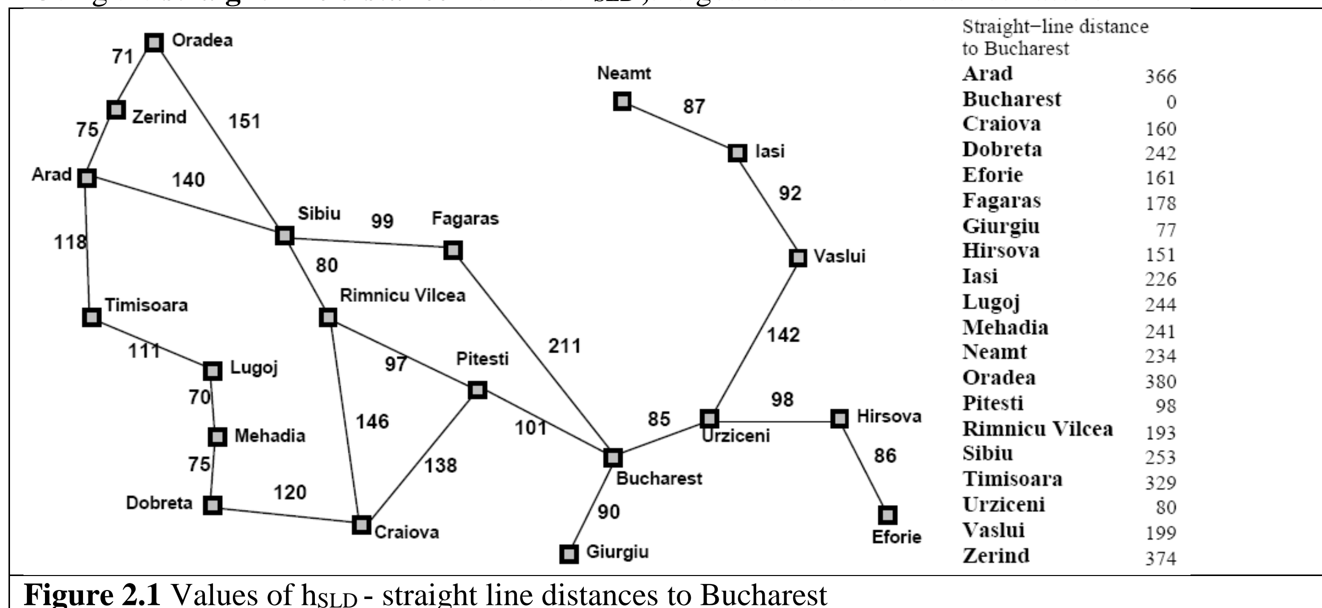


Figure 2.1 Values of h_{SLD} - straight line distances to Bucharest

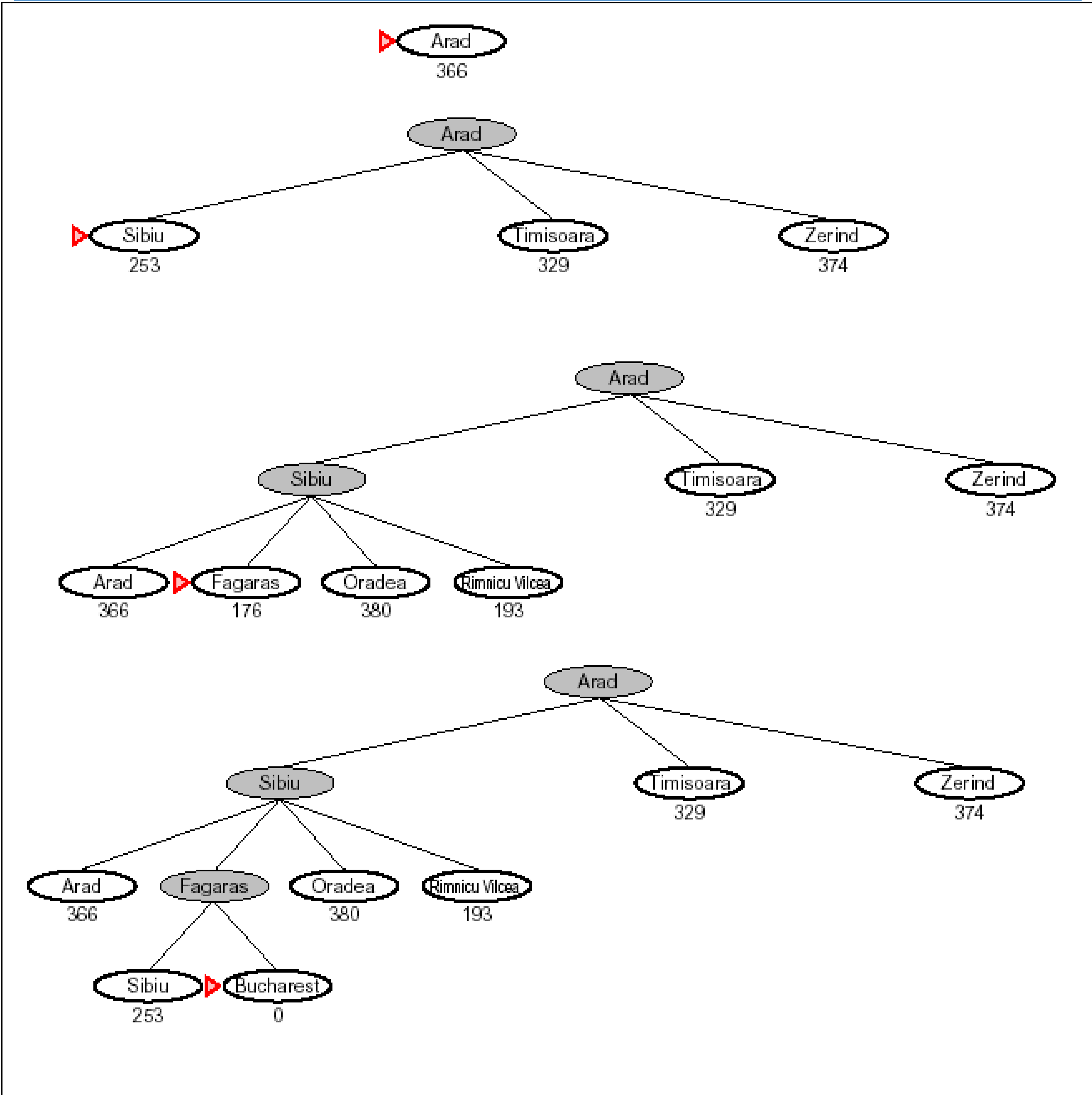


Figure 2.2 stages in greedy best-first search for Bucharest using straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Figure 2.2 shows the progress of greedy best-first search using h_{SLD} to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu,because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras,because it is closest. Fagaras in turn generates Bucharest,which is the goal.

Properties of greedy search

- **Complete??** No–can get stuck in loops, e.g.,
  Iasi ! Neamt ! Iasi ! Neamt !
  Complete in finite space with repeated-state checking
- **Time??** O(bm), but a good heuristic can give dramatic improvement
- **Space??** O(bm)—keeps all nodes in memory
- **Optimal??** No

Greedy best-first search is not optimal,and it is incomplete.
The worst-case time and space complexity is $O(b^m)$,where m is the maximum depth of the search space.

A^* Search

A^* Search is the most widely used form of best-first search. The evaluation function f(n) is obtained by combining

(1) **g(n)** = the cost to reach the node,and
(2) **h(n)** = the cost to get from the node to the **goal** :

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

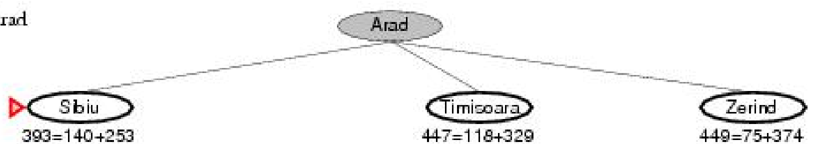
A^* Search is both optimal and complete. A^* is optimal if h(n) is an admissible heuristic. The obvious example of admissible heuristic is the straight-line distance h_{SLD}. It cannot be an overestimate.
A^* Search is optimal if h(n) is an admissible heuristic – that is,provided that h(n) never overestimates the cost to reach the goal.
An obvious example of an admissible heuristic is the straight-line distance h_{SLD} that we used in getting to Bucharest. The progress of an A^* tree search for Bucharest is shown in Figure 2.2.
The values of 'g ' are computed from the step costs shown in the Romania map(figure 2.1). Also the values of h_{SLD} are given in Figure 2.1.

Recursive Best-first Search(RBFS)

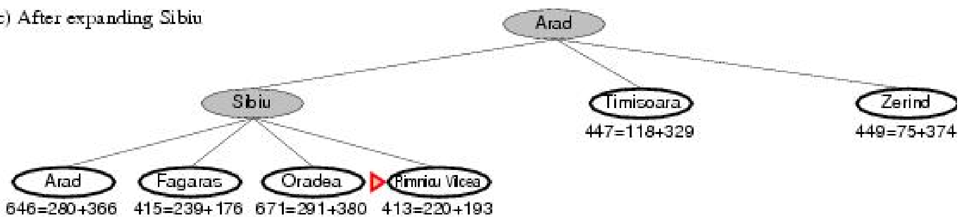
Recursive best-first search is a simple recursive algorithm that attempts to mimic the operation of standard best-first search,but using only linear space. The algorithm is shown in figure 2.4.
Its structure is similar to that of recursive depth-first search,but rather than continuing indefinitely down the current path,it keeps track of the f-value of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit,the recursion unwinds back to the alternative path. As the recursion unwinds,RBFS replaces the f-value of each node along the path with the best f-value of its children.
Figure 2.5 shows how RBFS reaches Bucharest.

(a) The initial state

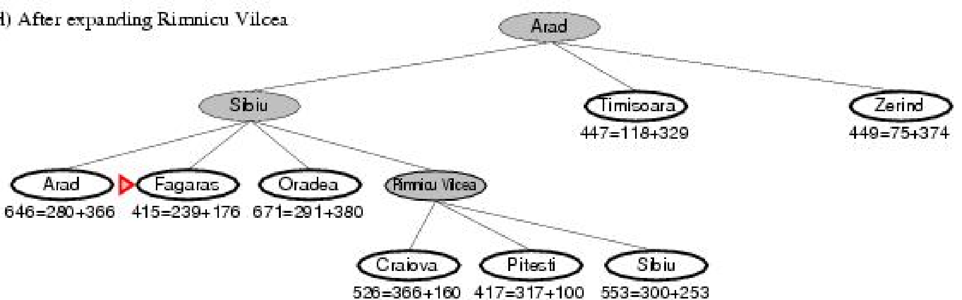
After expanding Arad



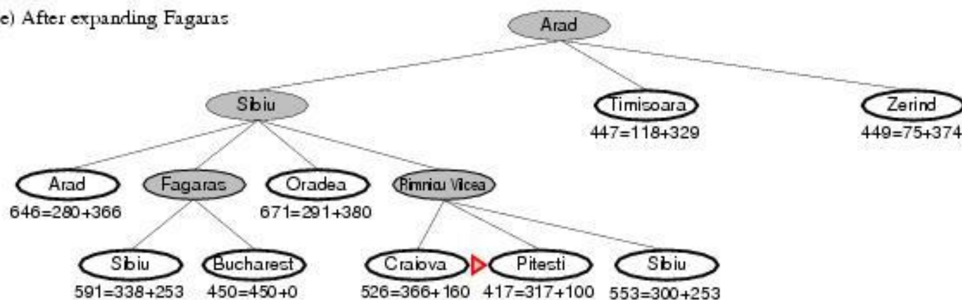
(c) After expanding Sibiu



(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



(f) After expanding Pitesti

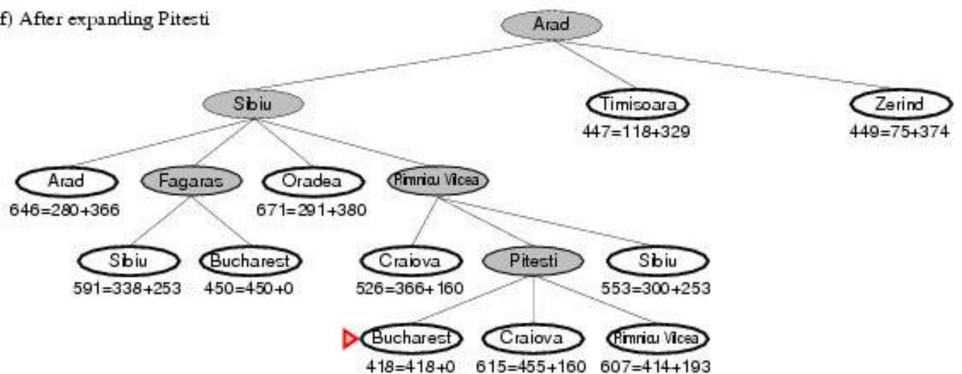


Figure 2.3 Stages in A* Search for Bucharest. Nodes are labeled with f = g + h . The h-values are the straight-line distances to Bucharest taken from figure 2.1

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) return a solution or failure
    return RFBS(problem,MAKE-NODE(INITIAL-STATE[problem]),∞)

function RFBS( problem, node, f_limit) return a solution or failure and a new f-
    cost limit
    if GOAL-TEST[problem](STATE[node]) then return node
    successors ← EXPAND(node, problem)
    if successors is empty then return failure, ∞
    for each s in successors do
        f [s] ← max(g(s) + h(s), f [node])
    repeat
        best ← the lowest f-value node in successors
        if f [best] > f_limit then return failure, f [best]
        alternative ←  the second lowest f-value among successors
        result, f [best] ←  RBFS(problem, best, min(f_limit, alternative))
        if result ≠ failure then return result
```

Figure 2.4 The algorithm for recursive best-first search

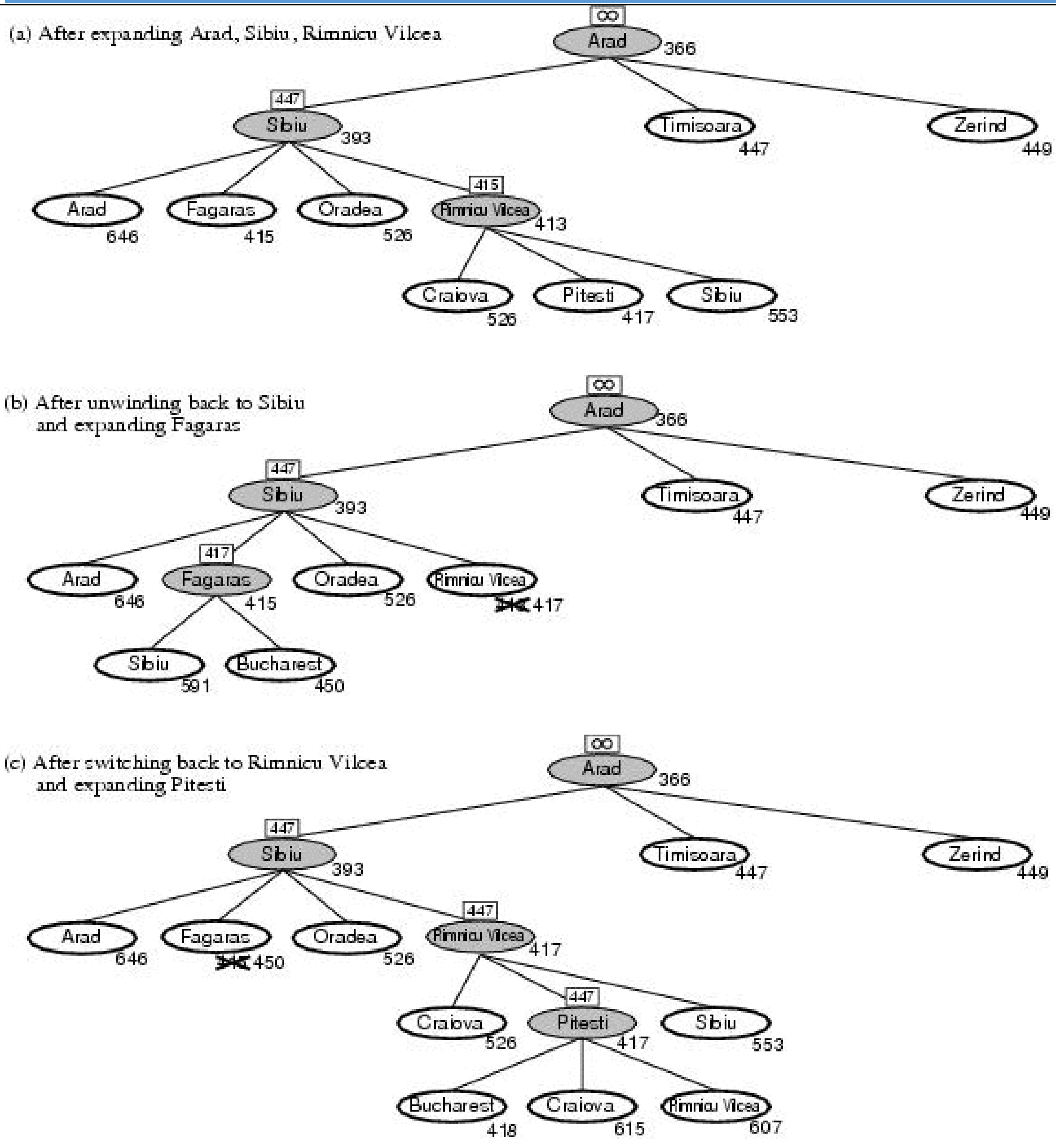


Figure 2.5 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea;then Fagaras is expanded,revealing a best leaf value of 450.
(c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed upto Fagaras; then Rimni Vicea is expanded. This time because the best alternative path(through Timisoara) costs atleast 447,the expansion continues to Bucharest

RBFS **Evaluation** :

RBFS is a bit more efficient than IDA*
- Still excessive node generation (mind changes)

Like A*, optimal if *h(n)* is admissible
Space complexity is *O(bd)*.
- IDA* retains only one single number (the current f-cost limit)

Time complexity difficult to characterize
- Depends on accuracy if h(n) and how often best path changes.

IDA* en RBFS suffer from ***too little*** memory.

2.1.2 Heuristic Functions

A **heuristic function** or simply a heuristic is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search

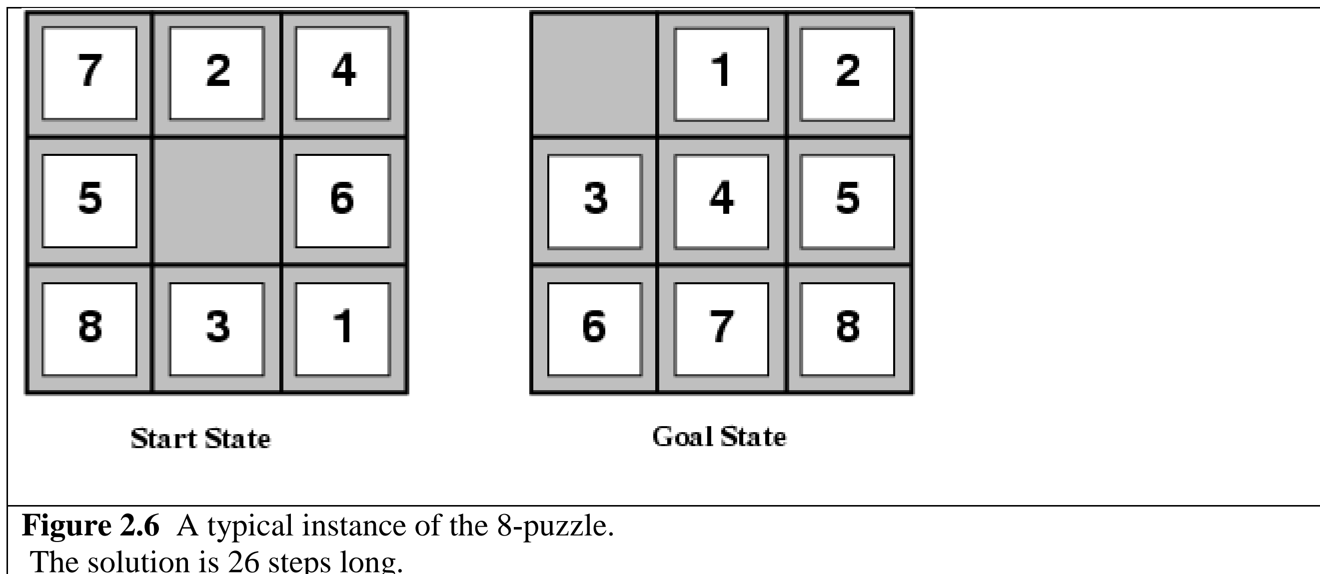


Figure 2.6 A typical instance of the 8-puzzle.
The solution is 26 steps long.

The 8-puzzle
The 8-puzzle is an example of Heuristic search problem. The object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration(Figure 2.6)
The average cost for a randomly generated 8-puzzle instance is about 22 steps. The branching factor is about 3.(When the empty tile is in the middle,there are four possible moves;when it is in the corner there are two;and when it is along an edge there are three). This means that an exhaustive search to depth 22 would look at about 3^{22} approximately = 3.1 X 10^{10} states.
By keeping track of repeated states,we could cut this down by a factor of about 170,000,because there are only 9!/2 = 181,440 distinct states that are reachable. This is a manageable number ,but the corresponding number for the 15-puzzle is roughly 10^{13}.
If we want to find the shortest solutions by using A^*,we need a heuristic function that never overestimates the number of steps to the goal.
The two commonly used heuristic functions for the 15-puzzle are :

(1) h_1 = the number of misplaced tiles.

For figure 2.6 ,all of the eight tiles are out of position,so the start state would have $h_1 = 8$. h_1 is an admissible heuristic.

(2) h_2 = the sum of the distances of the tiles from their goal positions. This is called **the city block distance** or **Manhattan distance.**

h_2 is admissible ,because all any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in start state give a Manhattan distance of

$$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18.$$

Neither of these overestimates the true solution cost ,which is 26.

The Effective Branching factor

One way to characterize the **quality of a heuristic** is the **effective branching factor b*.** If the total number of nodes generated by A* for a particular problem is **N**,and the **solution depth** is **d,**then b^* is the branching factor that a uniform tree of depth d would have to have in order to contain N+1 nodes. Thus,

$$N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$$

For example,if A^* finds a solution at depth 5 using 52 nodes,then effective branching factor is 1.92. A well designed heuristic would have a value of b^* close to 1,allowing failru large problems to be solved.

To test the heuristic functions h_1 and h_2,1200 random problems were generated with solution lengths from 2 to 24 and solved them with iterative deepening search and with A^* search using both h_1 and h_2. Figure 2.7 gives the averaghe number of nodes expanded by each strategy and the effective branching factor.

The results suggest that h_2 is better than h_1,and is far better than using iterative deepening search. For a solution length of 14,A^* with h_2 is 30,000 times more efficient than uninformed iterative deepening search.

	Search Cost			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Figure 2.7 Comparison of search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A^* Algorithms with h_1,and h_2. Data are average over 100 instances of the 8-puzzle,for various solution lengths.

Inventing admissible heuristic functions

- **Relaxed problems**

- A problem with fewer restrictions on the actions is called a ***relaxed problem***
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- If the rules of the 8-puzzle are relaxed so that a tile can move *anywhere*, then *h1(n)* gives the shortest solution
- If the rules are relaxed so that a tile can move to *any adjacent square,* then *h2(n)* gives the shortest solution

2.1.3 LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- For example,in the 8-queens problem,what matters is the final configuration of queens,not the order in which they are added.

- In such cases, we can **use local search algorithms.** They operate using a **single current state**(rather than multiple paths) and generally move only to neighbors of that state.
- The important applications of these class of problems are (a) integrated-circuit design,(b)Factory-floor layout,(c) job-shop scheduling,(d)automatic programming,(e)telecommunications network optimization,(f)Vehicle routing,and (g) portfolio management.

Key advantages of Local Search Algorithms
(1) They use very little memory – usually a constant amount; and
(2) they can often find reasonable solutions in large or infinite(continuous) state spaces for which systematic algorithms are unsuitable.

OPTIMIZATION PROBLEMS
Inaddition to finding goals,local search algorithms are useful for solving pure **optimization problems**,in which the aim is to find the **best state** according to an **objective function**.

State Space Landscape
To understand local search,it is better explained using **state space landscape** as shown in figure 2.8.
A landscape has both "**location**" (defined by the state) and "**elevation**"(defined by the value of the heuristic cost function or objective function).
If elevation corresponds to **cost**,then the aim is to find the **lowest valley** – a **global minimum**; if elevation corresponds to an **objective function**,then the aim is to find the **highest peak** – a **global maximum.**
Local search algorithms explore this landscape. A complete local search algorithm always finds a **goal** if one exists; an **optimal** algorithm always finds a **global minimum/maximum**.

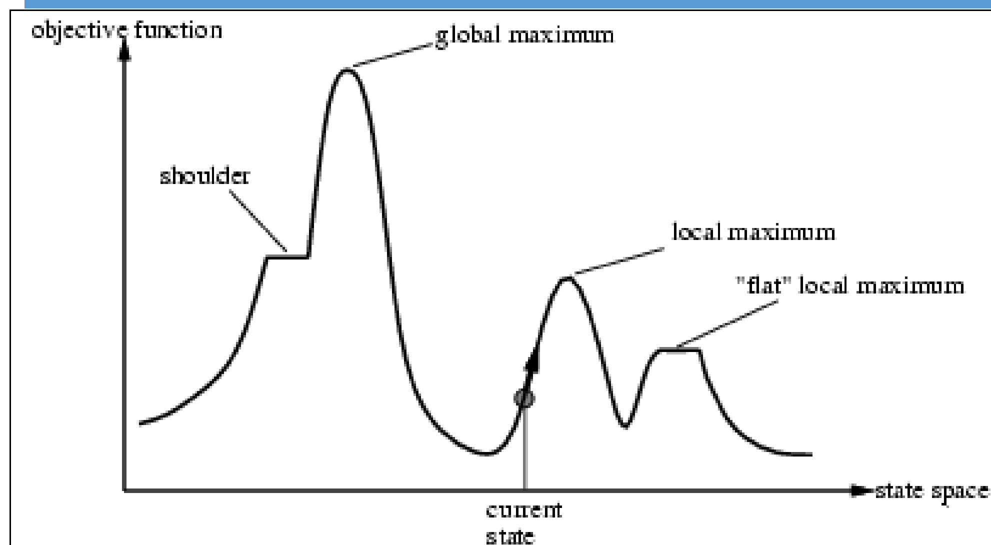


Figure 2.8 A one dimensional **state space landscape** in which elevation corresponds to the **objective function**. The aim is to find the global maximum. Hill climbing search modifies the current state to try to improve it ,as shown by the arrow. The various topographic features are defined in the text

Hill-climbing search

The **hill-climbing** search algorithm as shown in figure 2.9, is simply a loop that continually moves in the direction of increasing value – that is,**uphill**. It terminates when it reaches a "**peak**" where no neighbor has a higher value.

```
function HILL-CLIMBING( problem) return a state that is a local maximum
    input: problem, a problem
    local variables: current, a node.
                     neighbor, a node.

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest valued successor of current
        if VALUE [neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
```

Figure 2.9 The hill-climbing search algorithm (steepest ascent version),which is the most basic local search technique. At each step the current node is replaced by the best neighbor;the neighbor with the highest VALUE. If the heuristic cost estimate h is used,we could find the neighbor with the lowest h.

Hill-climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next. Greedy algorithms often perform quite well.

Problems with hill-climbing

Hill-climbing often gets stuck for the following reasons :

- **Local maxima** : a local maximum is a peak that is higher than each of its neighboring states,but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upwards towards the peak,but will then be stuck with nowhere else to go
- **Ridges** : A ridge is shown in Figure 2.10. Ridges results in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
- **Plateaux** : A plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum,from which no uphill exit exists,or a shoulder,from which it is possible to make progress.

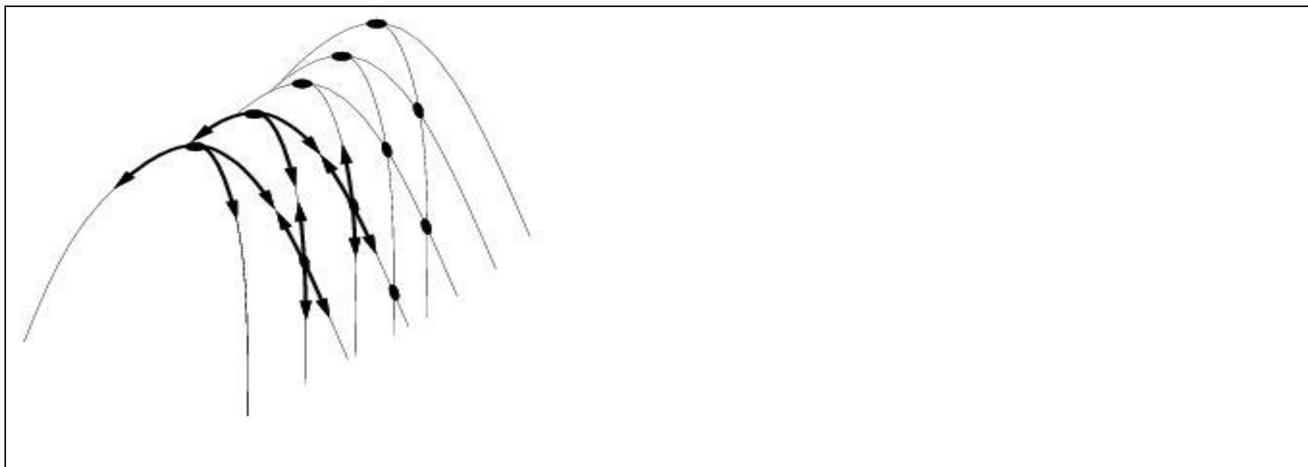


Figure 2.10 Illustration of why ridges cause difficulties for hill-climbing. The grid of states(dark circles) is superimposed on a ridge rising from left to right,creating a sequence of local maxima that are not directly connected to each other. From each local maximum,all th available options point downhill.

Hill-climbing variations

- **Stochastic hill-climbing**
  - Random selection among the uphill moves.
  - The selection probability can vary with the steepness of the uphill move.
- **First-choice hill-climbing**
  - cfr. stochastic hill climbing by generating successors randomly until a better one is found.
- **Random-restart hill-climbing**
  - Tries to avoid getting stuck in local maxima.

Simulated annealing search

A hill-climbing algorithm that never makes “downhill” moves towards states with lower value(or higher cost) is guaranteed to be incomplete,because it can stuck on a local maximum.In contrast,a purely random walk –that is,moving to a successor choosen uniformly at random from the set of successors – is complete,but extremely inefficient.

Simulated annealing is an algorithm that combines hill-climbing with a random walk in someway that yields both efficiency and completeness.

Figure 2.11 shows simulated annealing algorithm. It is quite similar to hill climbing. Instead of picking the best move,however,it picks the random move. If the move improves the situation,it is

always accepted. Otherwise,the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move – the amount E by which the evaluation is worsened.

Simulated annealing was first used extensively to solve VLSI layout problems in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks.

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps

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

Figure 2.11 The simulated annealing search algorithm,a version of stochastic hill climbing where some downhill moves are allowed.

Genetic algorithms

A Genetic algorithm(or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states,rather than by modifying a single state.

Like beam search,Gas begin with a set of k randomly generated states,called the population. Each state,or individual,is represented as a string over a finite alphabet – most commonly,a string of 0s and 1s. For example,an 8 8-quuens state must specify the positions of 8 queens,each in acolumn of 8 squares,and so requires $8 \times \log_2 8 = 24$ bits.

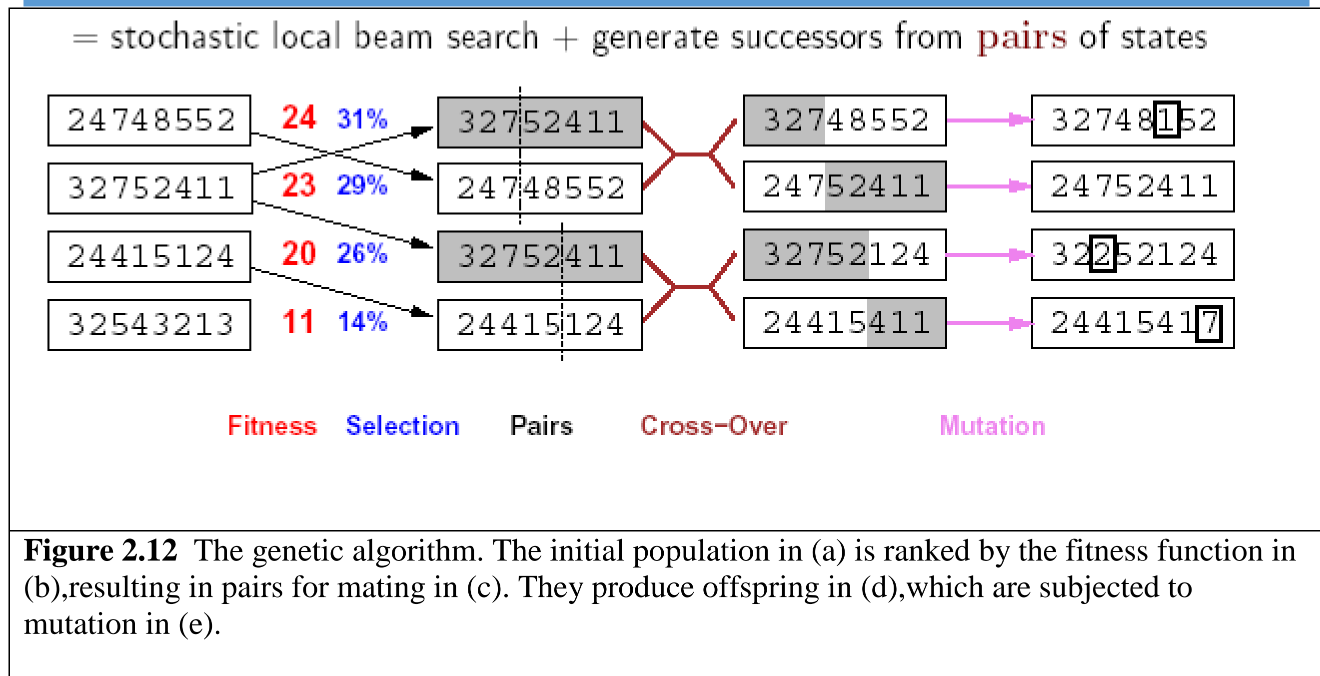


Figure 2.12 The genetic algorithm. The initial population in (a) is ranked by the fitness function in (b),resulting in pairs for mating in (c). They produce offspring in (d),which are subjected to mutation in (e).

Figure 2.12 shows a population of four 8-digit strings representing 8-queen states. The production of the next generation of states is shown in Figure 2.12(b) to (e).
In (b) each state is rated by the evaluation function or the **fitness function**.
In (c),a random choice of two pairs is selected for reproduction,in accordance with the probabilities in (b).
Figure 2.13 describes the algorithm that implements all these steps.

function GENETIC_ALGORITHM(*population,* FITNESS-FN) **return** an individual
  input: *population*, a set of individuals
    FITNESS-FN, a function which determines the quality of the individual
  repeat
    new_population ← empty set
    loop for i **from** 1 **to** SIZE(*population*) **do**
      x ← RANDOM_SELECTION(*population*, FITNESS_FN)
  y ← RANDOM_SELECTION(*population*, FITNESS_FN)
      child ← REPRODUCE(*x,y*)
      if (small random probability) **then** *child* ← MUTATE(*child*)
      add *child* to *new_population*
    population ← *new_population*
  until some individual is fit enough or enough time has elapsed
  return the best individual

Figure 2.13 A genetic algorithm. The algorithm is same as the one diagrammed in Figure 2.12,with one variation:each mating of two parents produces only one offspring,not two.

2.1.4 LOCAL SEARCH IN CONTINUOUS SPACES

- We have considered algorithms that work only in discrete environments, but real-world environment are continuous
- Local search amounts to maximizing a continuous objective function

in a multi-dimensional vector space.

- This is hard to do in general.
- Can immediately retreat
  - Discretize the space near each state
  - Apply a discrete local search strategy (e.g., stochastic hill climbing, simulated annealing)
- Often resists a closed-form solution
  - Fake up an empirical gradient
  - Amounts to greedy hill climbing in discretized state space
- Can employ Newton-Raphson Method to find maxima
- Continuous problems have similar problems: plateaus, ridges, local maxima, etc.

2.1.5 Online Search Agents and Unknown Environments

Online search problems

- Offline Search (all algorithms so far)
  - Compute complete solution, ignoring environment Carry out action sequence
- Online Search
  - Interleave computation and action
  - Compute—Act—Observe—Compute—·
- Online search good
  - For dynamic, semi-dynamic, stochastic domains
  - Whenever offline search would yield exponentially many contingencies
- Online search necessary for exploration problem
  - States and actions unknown to agent
  - Agent uses actions as experiments to determine what to do

Examples

Robot exploring unknown building
Classical hero escaping a labyrinth

- Assume agent knows
  - Actions available in state s
    Step-cost function $c(s,a,s')$
    State s is a goal state
- When it has visited a state s previously Admissible heuristic function h(s)
- Note that agent doesn't know outcome state (s ′) for a given action (a) until it tries the action (and all actions from a state s)
- Competitive ratio compares actual cost with cost agent would follow if it knew the search space
- No agent can avoid dead ends in all state spaces
  - Robotics examples: Staircase, ramp, cliff, terrain
- Assume state space is safely explorable—some goal state is always reachable

Online Search Agents

- Interleaving planning and acting hamstrings offline search
  - A* expands arbitrary nodes without waiting for outcome of action Online algorithm can expand only the node it physically occupies Best to explore nodes in physically local order
  - Suggests using depth-first search
  - Next node always a child of the current
- When all actions have been tried, can't just drop state
  Agent must physically backtrack
- Online Depth-First Search
  - May have arbitrarily bad competitive ratio (wandering past goal) Okay for exploration; bad for minimizing path cost
- Online Iterative-Deepening Search
  - Competitive ratio stays small for state space a uniform tree

Online Local Search

- Hill Climbing Search
  - Also has physical locality in node expansions
    Is, in fact, already an online search algorithm
  - Local maxima problematic: can't randomly transport agent to new state in effort to escape local maximum
- Random Walk as alternative
  - Select action at random from current state
  - Will eventually find a goal node in a finite space
  - Can be very slow, esp. if "backward" steps as common as "forward"
- Hill Climbing with Memory instead of randomness
  - Store "current best estimate" of cost to goal at each visited state Starting estimate is just h(s)
  - Augment estimate based on experience in the state space Tends to "flatten out" local minima, allowing progress Employ optimism under uncertainty
  - Untried actions assumed to have least-possible cost Encourage exploration of untried paths

Learning in Online Search

- Rampant ignorance a ripe opportunity for learning Agent learns a "map"
  of the environment
- Outcome of each action in each state
- Local search agents improve evaluation function accuracy
- Update estimate of value at each visited state
- Would like to infer higher-level domain model
- Example: "Up" in maze search increases y -coordinate Requires
- Formal way to represent and manipulate such general rules (so far, have hidden rules within the successor function)
- Algorithms that can construct general rules based on observations of the effect of actions

2.2 CONSTRAINT SATISFACTION PROBLEMS(CSP)

A **Constraint Satisfaction Problem**(or CSP) is defined by a set of **variables ,**$X_1, X_2, \ldots X_n$,and a set of constraints $C_1, C_2, \ldots, C_m$. Each variable X_i has a nonempty **domain** D,of possible **values**. Each constraint C_i involves some subset of variables and specifies the allowable combinations of values for that subset.
A **State** of the problem is defined by an **assignment** of values to some or all of the variables,$\{X_i = v_i, X_j = v_j, \ldots\}$. An assignment that does not violate any constraints is called a **consistent** or **legal assignment.** A complete assignment is one in which every variable is mentioned,and a **solution** to a CSP is a complete assignment that satisfies all the constraints.
Some CSPs also require a solution that maximizes an **objective function**.

Example for Constraint Satisfaction Problem :

Figure 2.15 shows the map of Australia showing each of its states and territories. We are given the task of coloring each region either red,green,or blue in such a way that the neighboring regions have the same color. To formulate this as CSP ,we define the variable to be the regions :WA,NT,Q,NSW,V,SA, and T. The domain of each variable is the set {red,green,blue}.The constraints require neighboring regions to have distinct colors;for example,the allowable combinations for WA and NT are the pairs
{(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}.
The constraint can also be represented more succinctly as the inequality WA not = NT,provided the constraint satisfaction algorithm has some way to evaluate such expressions.) There are many possible solutions such as
{ WA = red, NT = green,Q = red, NSW = green, V = red ,SA = blue,T = red}.
It is helpful to visualize a CSP as a constraint graph,as shown in Figure 2.15(b). The nodes of the graph corresponds to variables of the problem and the arcs correspond to constraints.

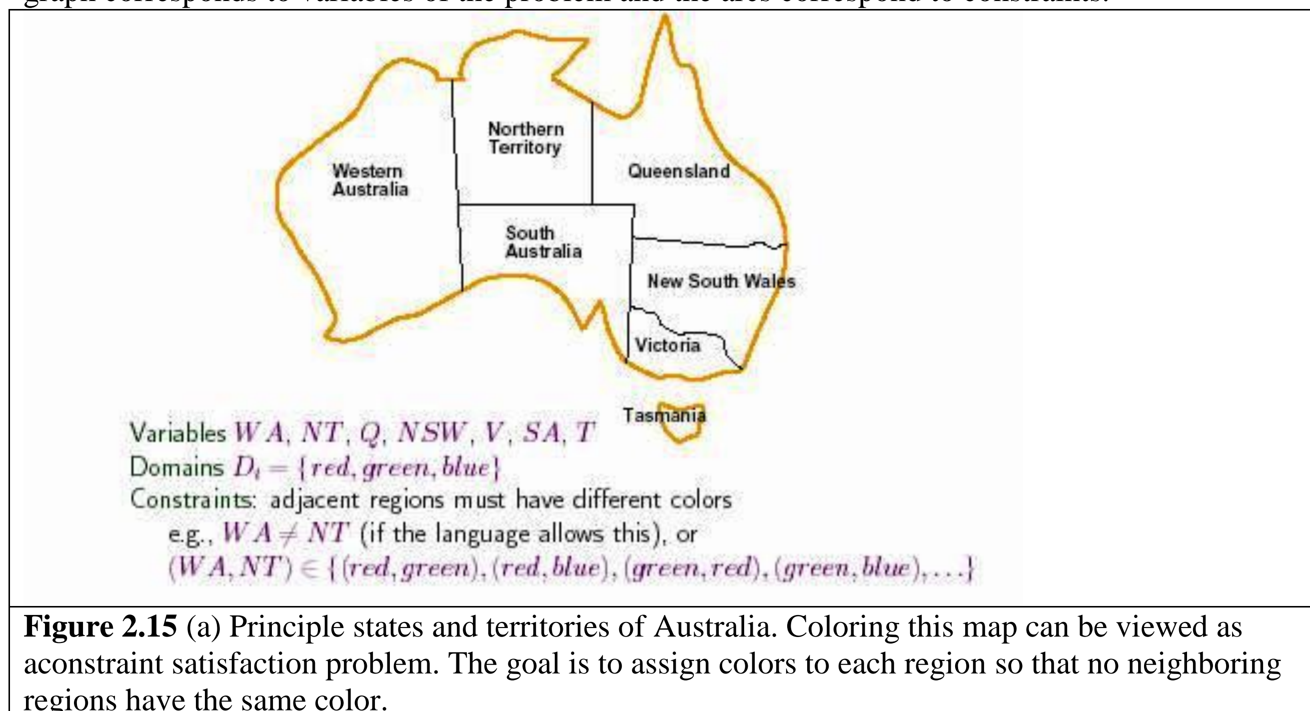


Figure 2.15 (a) Principle states and territories of Australia. Coloring this map can be viewed as aconstraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color.

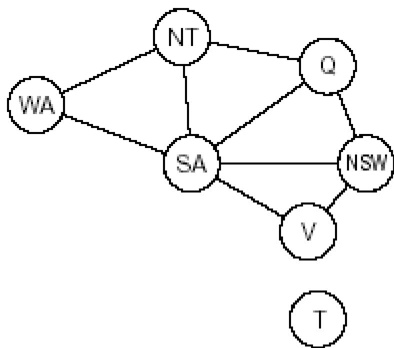


Figure 2.15 (b) The map coloring problem represented as a constraint graph.

CSP can be viewed as a standard search problem as follows :

- **Initial state** : the empty assignment { },in which all variables are unassigned.
- **Successor function** : a value can be assigned to any unassigned variable,provided that it does not conflict with previously assigned variables.
- **Goal test** : the current assignment is complete.
- **Path cost** : a constant cost(E.g.,1) for every step.

Every solution must be a complete assignment and therefore appears at depth n if there are n variables.
Depth first search algorithms are popular for CSPs

Varieties of CSPs

(i) Discrete variables

Finite domains

The simplest kind of CSP involves variables that are **discrete** and have **finite domains.** Map coloring problems are of this kind. The 8-queens problem can also be viewed as finite-domain CSP,where the variables Q_1,Q_2,…..Q_8 are the positions each queen in columns 1,….8 and each variable has the domain {1,2,3,4,5,6,7,8}. If the maximum domain size of any variable in a CSP is d,then the number of possible complete assignments is O(d^n) – that is,exponential in the number of variables. Finite domain CSPs include **Boolean CSPs**,whose variables can be either *true* or *false*.

Infinite domains

Discrete variables can also have **infinite domains** – for example,the set of integers or the set of strings. With infinite domains,it is no longer possible to describe constraints by enumerating all allowed combination of values. Instead a constraint language of algebric inequalities such as $Startjob_1 + 5 <= Startjob_3$.

(ii) CSPs with continuous domains

CSPs with continuous domains are very common in real world. For example ,in operation research field,the scheduling of experiments on the Hubble Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical,precedence and power constraints. The best known category of continuous-domain CSPs is that of **linear programming** problems,where the constraints must be linear inequalities forming a *convex* region. Linear programming problems can be solved in time polynomial in the number of variables.

Varieties of constraints :

(i) unary constraints involve a single variable.

Example : SA # green

(ii) Binary constraints involve paris of variables.
Example : SA # WA
(iii) Higher order constraints involve 3 or more variables.
Example : cryptarithmetic puzzles.

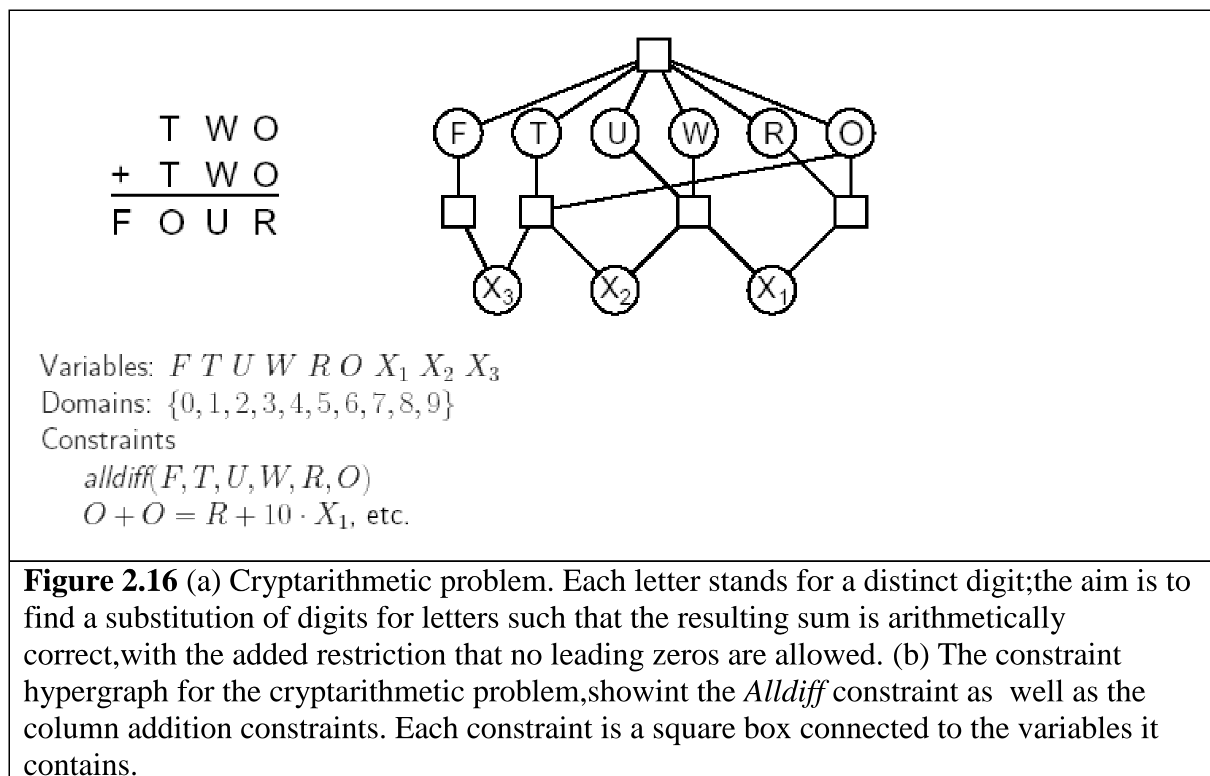


Figure 2.16 (a) Cryptarithmetic problem. Each letter stands for a distinct digit;the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct,with the added restriction that no leading zeros are allowed. (b) The constraint hypergraph for the cryptarithmetic problem,showint the *Alldiff* constraint as well as the column addition constraints. Each constraint is a square box connected to the variables it contains.

2.2.2 Backtracking Search for CSPs

The term **backtracking search** is used for depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in figure 2.17.

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
       if value is consistent with assignment given CONSTRAINTS[csp] then
           add {var = value} to assignment
           result ← RECURSIVE-BACKTRACKING(assignment, csp)
           if result ≠ failure then return result
           remove {var = value} from assignment
   return failure
```

Figure 2.17 A simple backtracking algorithm for constraint satisfaction problem. The algorithm is modeled on the recursive depth-first search

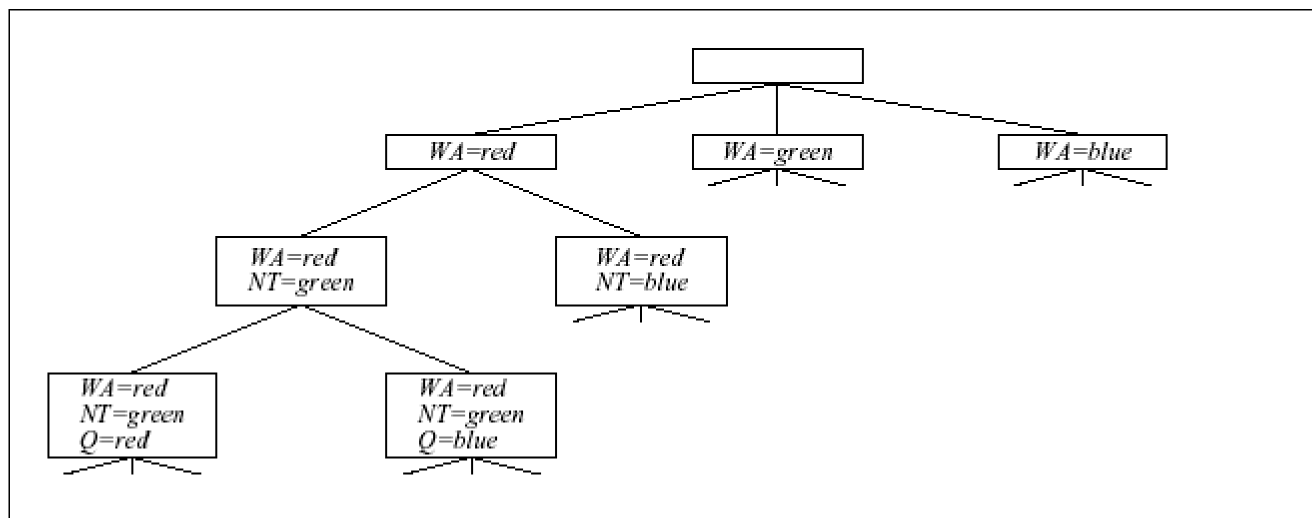


Figure 2.17(b) Part of search tree generated by simple backtracking for the map coloring problem.

Propagating information through constraints

So far our search algorithm considers the constraints on a variable only at the time that the variable is chosen by SELECT-UNASSIGNED-VARIABLE. But by looking at some of the constraints earlier in the search, or even before the search has started, we can drastically reduce the search space.

Forward checking

One way to make better use of constraints during search is called **forward checking**. Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y 's domain any value that is inconsistent with the value chosen for X. Figure 5.6 shows the progress of a map-coloring search with forward checking.

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		R G B

Figure 5.6 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes red from the domains of the neighboring variables NT and SA. After $Q = green$, $green$ is deleted from the domains of NT, SA, and NSW. After $V = blue$, $blue$ is deleted from the domains of NSW and SA, leaving SA with no legal values.

Constraint propagation

Although forward checking detects many inconsistencies, it does not detect all of them.

Constraint propagation is the general term for propagating the implications of a constraint on one variable onto other variables.

Arc Consistency

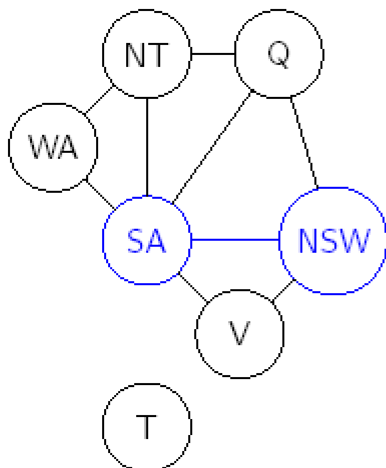


Figure: Australian Territories

- One method of constraint propagation is to enforce **arc consistency**
  - Stronger than forward checking
  - Fast
- *Arc* refers to a *directed* arc in the constraint graph
- Consider two nodes in the constraint graph (e.g., *SA* and *NSW*)
  - An arc is **consistent** if
  - For every value *x* of *SA*
  - There is some value *y* of *NSW* that is consistent with *x*
- Examine arcs for consistency in *both* directions

k-Consistency

- Can define stronger forms of consistency

k-Consistency

A CSP is **k-consistent** if, for any consistent assignment to $k-1$ variables, there is a consistent assignment for the k-th variable

- **1-consistency (node consistency)**
  - Each variable by itself is consistent (has a non-empty domain)
- **2-consistency (arc consistency)**
- **3-consistency (path consistency)**
  - Any pair of adjacent variables can be extended to a third

Local Search for CSPs

- Local search algorithms good for many CSPs
- Use complete-state formulation
  - Value assigned to every variable
  - Successor function changes one value at a time
- Have already seen this
  - Hill climbing for 8-queens problem (AIMA § 4.3)
- Choose values using **min-conflicts** heuristic
  - Value that results in the minimum number of conflicts with other variables

2.2.3 The Structure of Problems
Problem Structure

- Consider ways in which the structure of the problem's constraint graph can help find solutions
- Real-world problems require decomposition into subproblems

Independent Subproblems

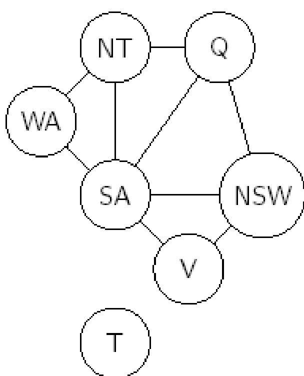


Figure: Australian Territories

- T is not connected
- Coloring T and coloring remaining nodes are **independent subproblems**
- *Any* solution for T combined with *any* solution for remaining nodes solves the problem
- Independent subproblems correspond to **connected components** of the constraint graph
- Sadly, such problems are rare

Tree-Structured CSPs

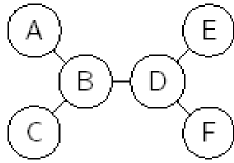


Figure: Tree-Structured CSP

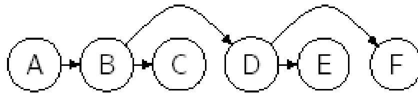


Figure: Linear ordering

- In most cases, CSPs are connected
- A simple case is when the constraint graph is a **tree**
- Can be solved in time linear in the number of variables
  - Order variables so that each parent precedes its children
  - Working "backward," apply arc consistency between child and parent
  - Working "forward," assign values consistent with parent

UNIT III

PLANNING

3.1 INTRODUCTION

The purpose of planning is to find a sequence of actions that achieves a given goal when performed starting in a given state. In other words, given a set of operator instances (defining the possible primitive actions by the agent), an initial state description, and a goal state description or predicate, the planning agent computes a plan.

There are two type of planning

1) CLASSICAL PLANNING: The environment which are fully observable, deterministic, finite, static and discrete.
2) NON CLASSICAL PLANNING: The environment which is partially observable or stochastic environment

3.1.2 WHAT HAVE WE DONE SO FAR ?

- Earlier we saw that **problem-solving agents** are able to plan ahead - to consider the consequences of *sequences* of actions - before acting.
- We also saw that a **knowledge-based agents** can select actions based on explicit, logical representations of the current state and the effects of actions. This allows the agent to succeed in complex, inaccessible environments that are too difficult for a problem-solving agent
- **Problem Solving Agents + Knowledge-based Agents = Planning Agents**

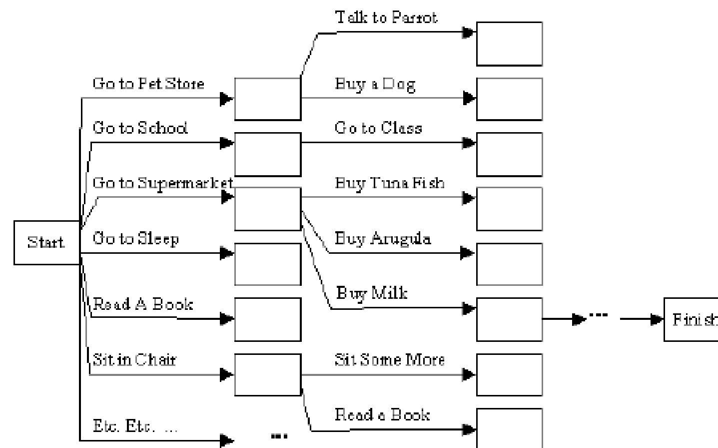
3.1.3 SIMPLE PLANNING ALGORITHM

1. Generate a goal to achieve
2. Construct a plan to achieve goal from current state
3. Execute plan until finished
4. Begin again with new goal

Functions used in the algorithm:
STATE-DESCRIPTION: uses a percept as input and returns the description of the initial state in a format required for the planner.
IDEAL-PLANNER: is the planning algorithm
MAKE-GOAL-QUERY: asks the knowledge base what the next goal will be.

3.1.4 PROBLEM SOLVING vs PLANNING

Consider the task of getting milk or banana ect form super market… the standard search algorithm fails miserably



In searches (problem solving), operators are used simply to generate successor states and we can not look "inside" an operator to see how it's defined.
The goal-test predicate also is used as a "black box" to test if a state is a goal or not.
The search cannot use properties of how a goal is defined in order to reason about finding path to that goal. *Hence this approach is all algorithm and representation weak.*

Planning is considered different from problem solving because of the difference in the way they represent states, goals, actions, and the differences in the way they construct action sequences.

Problems with Problem solving agent:

- It is evident from the above figure that the actual branching factor would be in the thousands or millions. The heuristic evaluation function can only choose states to determine which one is closer to the goal. It cannot eliminate actions from consideration. The agent makes guesses by considering actions and the evaluation function ranks those guesses. The agent picks the best guess, but then has no idea what to try next and therefore starts guessing again.

- It considers sequences of actions beginning from the initial state. The agent is forced to decide what to do in the initial state first, where possible choices are to go to any of the next places. Until the agent decides how to acquire the objects, it can't decide where to go.

Planning emphasizes what is in operator and goal representations. There are three key ideas behind planning:

- *to "open up" the representations* of state, goals, and operators so that a reasoner can more intelligently select actions when they are needed
- *the planner is free to add actions to the plan* wherever they are needed, rather than in an incremental sequence starting at the initial state
- *most parts of the world are independent of most other parts* which makes it feasible to take a conjunctive goal and solve it with a divide-and-conquer strategy

3.1.5 SITUATION CALCULS

Situation calculus is a version of first-order-logic (FOL) that is augmented so that it can reason about actions in time.

$PlanResult(p, s)$ is the situation resulting from executing p in s

$$PlanResult([\,], s) = s$$
$$PlanResult([a|p], s) = PlanResult(p, Result(a, s))$$

Initial state $At(Home, S_0) \wedge \neg Have(Milk, S_0) \wedge \ldots$

Actions as Successor State axioms

$Have(Milk, Result(a, s)) \Leftrightarrow$
$[(a = Buy(Milk) \wedge At(Supermarket, s)) \vee (Have(Milk, s) \wedge a \neq \ldots)]$

Query

$$s = PlanResult(p, S_0) \wedge At(Home, s) \wedge Have(Milk, s) \wedge \ldots$$

Solution

$$p = [Go(Supermarket), Buy(Milk), Buy(Bananas), Go(HWS), \ldots]$$

3.1.6 STRIPS (Stanford Research Institute Problem Solver)

Classical Planners use the STRIPS (Stanford Research Institute Problem Solver) language to describe states and operators. It is an efficient way to represent planning algorithms.

States are represented by conjunctions of function-free ground literals, that is, predicates applied to constant symbols, possibly negated.

- An example of an initial state is:
  At(Home) /\ -Have(Milk) /\ -Have(Bananas) /\ -Have(Drill) /\ ...
- Goals are a conjunction of literals. Therefore the goal is
  At(Home) /\ Have(Milk) /\ Have(Bananas) /\ Have(Drill)
- Goals can also contain variables. Being at a store that sells milk is equivalent to
  At(x) /\ Sells(x,Milk)

Strips operators consist of three components

- **action description**: what an agent actually returns to the environment in order to do something.
- **precondition**: conjunction of atoms (positive literals), that says what must be true before an operator can be applied.
- **effect of an operator**: conjunction of literals (positive or negative) that describe how the situation changes when the operator is applied.

Example:
An example action of going from one place to another:

Op(ACTION:*Go(there)*, PRECOND:*At(here) /\ Path(here, there)* EFFECT:*At(there) /\ -At(here)*)

3.2 PLANNING AS SEARCH

There are two main approaches to solving planning problems, depending on the kind of search space that is explored:

1. Situation-space search
2. Planning-space search

PLAN SPACE SEARCH

- the search space is the space of all possible plans
- a node corresponds to a partial plan
- initially we will specify an "initial plan" which is one node in this space
- a goal node is a node containing a plan which is complete, satisfying all of the goals in the goal state
- the node itself contains all of the information for determining a solution plan (e.g. sequence of actions)

3.2.1 SITUATION SPACE SEARCH

In situation space search

- The search space is the space of all possible states or situations of the world
- Initial state defines one node
- A goal node is a state where all goals in the goal state are satisfied
- A solution plan is the sequence of actions (e.g. operator instances) in the path from the start node to a goal node

There are 2 approaches to situation-space planning:

1. Progression situation-space planning
2. Regression situation-space planning

3.2.1.1 FORWARD CHAINING OR PROGRESSION SITUATION SPACE PLANNING

- *Forward-chaining* from initial state to goal state
- Looks just like a state-space search except STRIPS operators are specified instead of a set of next-move functions
- You can use any search method you like (i.e. BFS, DFS, A*)
- **Disadvantage**: huge search space to explore, so usually very inefficient, high branching factor!

Algorithm

1. Start from initial state
2. Find all operators whose preconditions are true in the initial state
3. Compute effects of operators to generate successor states
4. Repeat steps #2-#3 until a new state satisfies the goal conditions

Formulation as state-space search problem:

- Initial state = initial state of the planning problem
  Literals not appearing are false
- Actions = those whose preconditions are satisfied
- Add positive effects, delete negative
- Goal test = does the state satisfy the goal
- Step cost = each action costs 1

3.2.1.2 BACKWARD CHAINING OR REGRESSION SITUATION SPACE PLANNING

- *Backward-chaining* from goal state to initial state
- Regression situation-space planning is usually more efficient than progression because many operators are applicable at each state, yet only a small number of operators are applicable for achieving a given goal

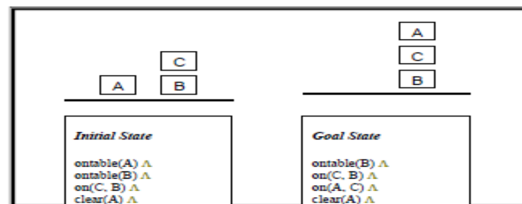
- Hence, regression is more goal-directed than progression situation-space planning
- **Disadvantage***:* cannot always find a plan even if one exists!

Algorithm:

1. Start with goal node corresponding to goal to be achieved
2. Choose an operator that will *add* one of the goals
3. Replace that goal with the operator's preconditions
4. Repeat steps #2-#3 until you have reached the initial state
5.

While backward-chaining is performed by STRIPS in terms of the generation of goals, sub-goals, sub-sub-goals, etc., operators are used in the forward direction to generate successor states, starting from the initial state, until a goal is found.

3.2.2 EXAMPLE: BOX WORLD



Definitions of Descriptors:

ontable(x): block *x* is on top of the table

on(x,y): block *x* is on top of block *y*

clear(x): there is nothing on top of block *x*; therefore it can be picked up
handempty: you are not holding any block

Definitions of Operators:

Op{ACTION: **pickup(x)**
PRECOND: ontable(x), clear(x), handempty
EFFECT: holding(x), ~ontable(x), ~clear(x), ~handempty }

Op{ACTION: **putdown(x)**
PRECOND: holding(x)
EFFECT: ontable(x), clear(x), handempty, ~holding(x) }

Op{ACTION: **stack(x,y)**
PRECOND: holding(x), clear(y)
EFFECT: on(x,y), clear(x), handempty, ~holding(x), ~clear(y) }

Op{ACTION: **unstack(x,y)**
PRECOND: clear(x), on(x,y), handempty
EFFECT: holding(x), clear(y), ~clear(x), ~on(x,y), ~handempty) }

FORWARD CHANNING

Step	State	Applicable Operators	Operator Applied
#1	ontable(A) Λ ontable(B) Λ on(C, B) Λ clear(A) Λ clear(C) Λ handempty	pickup(A) unstack(C,B)	pickup(A)
#2	ontable(A) Λ ontable(B) Λ on(C, B) Λ ~clear(A) Λ clear(C) Λ holding(A)	putdown(A) stack(A,C)	stack(A,C)
#3	ontable(B) Λ on(C, B) Λ on(A, C) Λ clear(A) Λ ~clear(C) Λ handempty Λ ~holding(A)	Matches goal state so	

BACKWARD CHANNING

Step	State	Stack	Plan	Note
#1	ontable(A) Λ ontable(B) Λ on(C, B) Λ clear(A) Λ clear(C) Λ handempty	achieve(on(A,C))		Stack contains original goal. State contains the initial state description.
#2	Same.	achieve(clear(C), holding(A), apply(Stack(A,C)) achieve(on(A,C))		Choose operator Stack to solve goal popped from top of goal stack.
#3	Same.	achieve(holding(A)) achieve(clear(C)) achieve(clear(C), holding(A), apply(Stack(A,C)) achieve(on(A,C))		Order sub-goals arbitrarily.
#4	Same.	achieve(ontable(A), clear(A), handempty), apply(pickup(A)) achieve(holding(A)) achieve(clear(C)) achieve(clear(C), holding(A), apply(Stack(A,C)) achieve(on(A,C))		Choose operator *pickup* to solve goal popped from top of goal stack.
#5	ontable(B) Λ on(C, B) Λ clear(C) Λ holding(A)	achieve(clear(C)) achieve(clear(C), holding(A), apply(Stack(A,C)) achieve(on(A,C))	Pickup(A)	Top goal is true in current state, so pop it and apply operator *pickup(A)*.
#6	ontable(B) Λ on(C, B) Λ on(A,C) Λ clear(A) Λ handempty	achieve(on(A,C))	pickup(A) stack(A,C)	Top goal *achieve(C)* true so pop it. Re-verify that goals that are the preconditions of the *stack(A,C)* operator still true, then pop that and the operator is applied.

#7	Same.	<empty>		Re-verify that original goal is true in current state, then pop and halt with empty goal stack and state description satisfying original goal.

3.2.3 HEURISTICS FOR STATE-SPACE SEARCH

• Neither progression or regression are very efficient without a good heuristic.

How many actions are needed to achieve the goal?

Exact solution is NP hard, find a good estimate

Two approaches to find admissible heuristic:

• The optimal solution to the relaxed problem.

Remove all preconditions from actions

• The subgoal independence assumption:

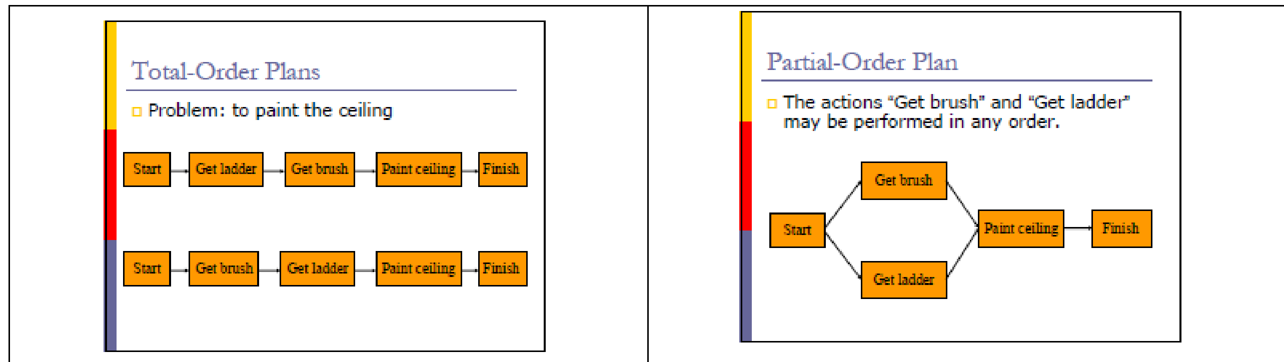
The cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving the subproblems independently.

3.4 PARTIAL ORDER PLANNING

TOTAL ORDER vs PARTIAL ORDER PLANNER

- Any planner that maintains a partial solution as a totally ordered list of steps found so far is called a **total-order planner**, or a **linear planner**.
- Alternatively, if we only represent partial-order constraints on steps, then we have a **partial-order planner**, which is also called a **non-linear planner**.
- In this case, we specify a set of temporal constraints between pairs of steps of the form S1 < S2 meaning that step S1 comes before, but not necessarily immediately before, step S2. We also show this temporal constraint in graph form as

  S1 +++++++++> S2
- STRIPS is a total-order planner, as are situation-space progression and regression planners
- Partial-order planners exhibit the property of least commitment because constraints ordering steps will only be inserted when necessary. On the other hand, situation-space progression planners make commitments about the order of steps as they try to find a solution and therefore may make mistakes from poor guesses about the right order of steps.



Differences between situation-space algorithm and plan-space algorithm

- An alternative is to search through the space of *plans* rather than a space of *situations*.
- That is, we start with a simple, incomplete plan, which we call a **partial plan**.
- Then we consider ways of expanding the partial plan until we come up with a complete plan that solves the problem.
- We use this approach when the ordering of sub-goals affects the solution.

Two types of operators are used:

- Refinement operators take a partial plan and add constraints to it. They eliminate some plans from the set and they never add new plans to it.

- A modification operator debugs incorrect plans that the planner may make, therefore we can worry about bugs later.

Key Difference Between Plan-Space Planning and Situation-Space Planning In Situation-Space planners all operations, all variables, and all orderings must be fixed when each operator is applied. Plan-Space planners make commitments (i.e., what steps in what order) only as necessary. Hence, Plan-Space planners do least-commitment planning.

HOW TO REPRESENT PLAN

A **plan** is formally defined as a data structure consisting of the following 4 components:

1. A set of plan steps
2. A set of step ordering constraints
3. A set of variable binding constraints
4. A set of causal links

Example: *Plan*(
STEPS:{S_1:*Op(*ACTION: *Start),*
S_2:*Op*(ACTION: *Finish*,
PRECOND: *Ontable*(c), *On*(b,c), *On*(a,b) },
ORDERINGS: {$S_1 < S_2$},
BINDINGS: {},

LINKS: {})

UNIT IV UNCERTAIN KNOWLEDGE AND REASONING 9
Uncertainty – review of probability - probabilistic Reasoning – Bayesian networks – inferences in Bayesian networks – Temporal models – Hidden Markov models

1) What is uncertainity?

The logical agents make the epistemological commitment that propositions are true, false, or unknown. When an agent knows enough facts about its environment, the logical approach enables it to derive plans that are guaranteed to work. This is a good thing. Unfortunately, *agents almost never have access to the whole truth about their erzvironment.* Agents must, therefore, act under **uncertainty.** For example, an agent in the wumpus world has sensors that report only local information; most of the world is not immediately observable. A wumpus agent often will find itself unable to discover which of two squares contains a pit. If those squares are *en route* to the gold, then the agent might have to take a chance and enter one of the two squares.
The real world is far more complex than the wumpus world. For a logical agent, it might be impossible to construct a complete and correct description of how its actions will work. Suppose, for example, that the agent wants to drive someone to the airport to catch a flight and is considering a plan, Ago, that involves leaving home 90 minutes before the flight departs and driving at a reasonable speed. Even though the airport is only about 15 miles away, the agent will not be able to conclude with certainty that "Plan A90 will get us to the airport in time." Instead, it reaches the weaker conclusion "Plan A90 will get us to the airport in time, as long as my car doesn't break down or run out of gas, and I don't get into an accident, and there are no accidents on the bridge, and the plane doesn't leave early, and . . . ." None of these conditions can be deduced, so the plan's success cannot be inferred.
If a logical agent cannot conclude that any particular course of action achieves its goal, then it will be unable to act. Conditional planning can overcome uncertainty to some extent, but only if the agent's sensing actions can obtain the required information and only if there are not too many different contingencies.

Uncertain knowledge

$\forall$p symptom(p, Toothache) $\rightarrow$ disease(p,cavity)

$\forall$p sympt(p,Toothache) $\rightarrow$

disease(p,cavity) $\vee$ disease(p,gum_disease) $\vee$…

2) Define Prior Probability.

Prior probability

The **unconditional** or **prior probability** associated with a proposition *a* is the degree of belief accorded to it in the absence of any other information; it is written as $P(a)$.
For example, if the prior probability that I have a cavity is 0.1, then we would write

$P(Cavity = true) = 0.1$ or $P(cavity) = 0.1$.

Sometimes, we will want to talk about the probabilities of all the possible values of a random variable. In that case, we will use an expression such as **P(** *Weather),* which denotes

a ***vector*** of values for the probabilities of each individual state of the weather. Thus, instead

of writing the four equations
P(Weather = sunny) = 0.7
P(TNeather = rain) = 0.2
P(Weather = cloudy) = 0.08
P(Weather = snow) = 0.02 .
we may simply write
P(Weather) = (0.7,0.2,0.08,0.02) .
PROBABILITY
DISTRIBUTION This statement defines a prior **probability distribution** for the random variable *Weather.*

3) What is conditional probability?

Conditional probability

Once the agent has obtained some evidence concerning the previously unknown random variables making up the domain, prior probabilities are no longer applicable. Instead, we use

conditional or **posterior** probabilities. The notation used is *P(alb),* where *a* and *b* are any

proposition^.^ This is read as "the probability of *a,* given that *all* we know is *b."* For example,
P(cavity1 toothache) = 0.8
indicates that if a patient is observed to have a toothache and no other information is yet available, then the probability of the patient's having a cavity will be 0.8. A prior probability, such as *P(cavity),* can be thought of as a special case of the conditional probability *P(cavity(*),
where the probability is conditioned on no evidence.
Conditional probabilities can be defined in terms of unconditional probabilities. The defining equation is which holds whenever *P(b)* > 0.
This equation can also be written as
P(a A b) = P(a(b) P(b)
which is called the **product rule.** The product rule is perhaps easier to remember: it comes from the fact that, for *a* and *b* to be true, we need *b* to be true, and we also need *a* to be true given *b.* We can also have it the other way around:
P(a A b) = P(b(a)P(a.)

4) What are the axioms of probability?

So far, we have defined a syntax for propositions and for prior and conditional probability statements about those propositions. Now we must provide some sort of semantics for probability statements.
We begin with the **basic axioms** that serve to define the probability scale and its endpoints:

1. All probabilities are between 0 and 1. For any proposition **a**,

$$0 \leq P(a) \leq 1$$

2. Necessarily true (i.e., valid) propositions have probability 1, and necessarily false (i.e., unsatisfiable) propositions have probability 0.

$$P(true) = 1 \qquad P(false) = 0 .$$

Next, we need an axiom that connects the probabilities of logically related propositions. The simplest way to do this is to define the probability of a disjunction as follows:

3. The probability of a disjunction is given by

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b) .$$

These three axioms are often called **Kolmogorov's axioms** in honor of the Russian mathematician Andrei Kolmogorov, who showed how to build up the rest of probability theory from this simple foundation.

5) Define Baye's Rule.

13.6 BAYES' RULE AND ITS USE

On page 470, we defined the **product rule** and pointed out that it can be written in two forms because of the commutativity of conjunction:

$$P(a \wedge b) = P(a|b)P(b)$$
$$P(a \wedge b) = P(b|a)P(a) .$$

Equating the two right-hand sides and dividing by $P(a)$, we get

$$P(b|a) = \frac{P(a|b)P(b)}{P(a)} \qquad (13.9)$$

BAYES' RULE

This equation is known as **Bayes' rule** (also Bayes' law or Bayes' theorem).[9] This simple equation underlies all modern AI systems for probabilistic inference. The more general case of multivalued variables can be written in the **P** notation as

$$\mathbf{P}(Y|X) = \frac{\mathbf{P}(X|Y)\mathbf{P}(Y)}{\mathbf{P}(X)}$$

where again this is to be taken as representing a set of equations, each dealing with specific values of the variables. We will also have occasion to use a more general version conditionalized on some background evidence **e**:

$$\mathbf{P}(Y|X, \mathbf{e}) = \frac{\mathbf{P}(X|Y, \mathbf{e})\mathbf{P}(Y|\mathbf{e})}{\mathbf{P}(X|\mathbf{e})} \qquad (13.10)$$

6) Give an example for application of Baye's Rule.

Applying Bayes' rule: The simple case

On the surface, Bayes' rule does not seem very useful. It requires three terms-a conditional probability and two unconditional probabilities-just to compute one conditional probability. Bayes' rule is useful in practice because there are many cases where we do have good probability estimates for these three numbers and need to compute the fourth. In a task such as medical diagnosis, we often have conditional probabilities on causal relationships and want to derive a diagnosis. A doctor knows that the disease meningitis causes the patient to have a stiff neck, say, 50% of the time. The doctor also knows some unconditional facts: the prior probability that a patient has meningitis is 1150,000, and the prior probability that any patient has a stiff neck is 1120. Letting *s* be the proposition that the patient has a stiff neck and *rn* be the proposition that the patient has meningitis, we have
That is, we

$$\begin{aligned} P(s|m) &= 0.5 \\ P(m) &= 1/50000 \\ P(s) &= 1/20 \\ P(m|s) &= \frac{P(s|m)P(m)}{P(s)} = \frac{0.5 \times 1/50000}{1/20} = 0.0002 . \end{aligned}$$

That is, we expect only 1 in 5000 patients with a stiff neck to have meningitis. Notice that, even though a stiff neck is quite strongly indicated by meningitis (with probability 0.5), the probability of meningitis in the patient remains small. This is because the prior probability on stiff necks is much higher than that on meningitis.

7) What are Bayesian Networks?

A Bayesian network is a directed graph in which each node is annotated with quantitative probability information. The full specification is as follows:

1. A set of random variables makes up the nodes of the network. Variables may be discrete or continuous.
2. A set of directed links or arrows connects pairs of nodes. If there is an arrow from node X to node Y, X is said to be aparent of Y.
3. Each node X_i has a conditional probability distribution $P(X_i|Parents(X_i))$ that quantifies the effect of the parents on the node.
4. The graph has no directed cycles (and hence is a directed, acyclic graph, or DAG).

8) Explain a Simple Bayesian Network with an example.

Consider the simple world consisting of the variables *Toothache, Cavityy, Catch,* and *Weather.* We argued that *Weather* is independent of the other variables; furthermore, we argued that *Toothache* and *Catch* are conditionally independent, given *Cavity.* These relationships are represented by the Bayesian network structure shown in Figure 14.1. Formally, the conditional independence of *Toothache* and *Catch* given *Cavity* is indicated by the absence of a link between *Toothache* and *Catch.* Intuitively, the network represents the fact that *Cavity* is a direct cause of *Toothache* and *Catch,* whereas no direct causal relationship exists between *Toothache* and *Catch.*

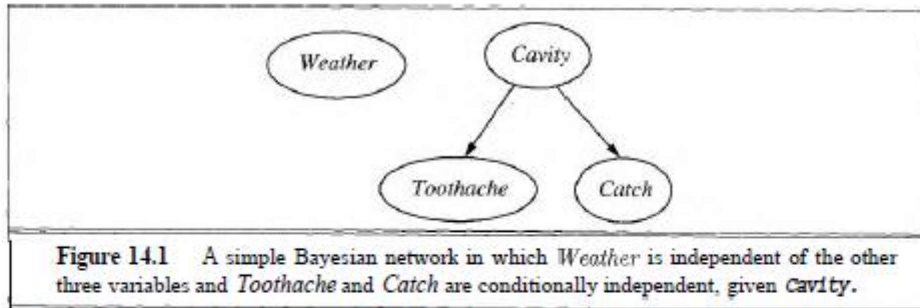


Figure 14.1 A simple Bayesian network in which *Weather* is independent of the other three variables and *Toothache* and *Catch* are conditionally independent, given *Cavity*.

9) Explain a Bayesian Network with an example showing both topology and Conditional probability tables(CPTs)

For the moment, let us ignore the conditional distributions in the figure and concentrate on the topology of the network. In the case of the burglary network, the topology shows that burglary and earthquakes directly affect the probability of the alann's going off, but whether John and Mary call depends only on the alarm. The network thus represents our assumptions that they do not perceive any burglaries directly, they do not notice the minor earthquakes, and they do not confer before calling.

Now let us turn to the conditional distributions shown in Figure 14.2. In the figure, each distribution is shown as a **conditional probability table,** or CPT. (This form of table can be used for discrete variables; other representations, including those suitable for continuous variables, are described in Section 14.2.) Each row in a CPT contains the conditional CONDITIONING CASE probability of each node value for a **conditioning case.** A conditioning case is just a possible combination of values for the parent nodes-a miniature atomic event, if you like. Each row must sum to 1, because the entries represent an exhaustive set of cases for the variable. For Boolean variables, once you know that the probability of a true value is *p,* the probability of false must be 1 - *p,* so we often omit the second number, as in Figure 14.2. In general, a table for a Boolean variable with *k* Boolean parents contains *2k* independently specifiable probabilities. A node with no parents has only one row, representing the prior probabilities of each possible value of the variable.

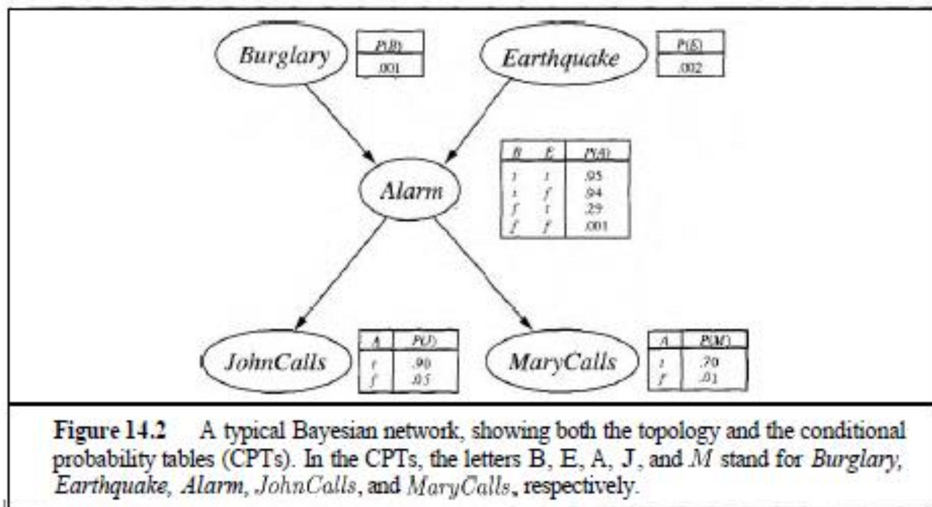


Figure 14.2 A typical Bayesian network, showing both the topology and the conditional probability tables (CPTs). In the CPTs, the letters B, E, A, J, and *M* stand for *Burglary, Earthquake, Alarm, JohnCalls*, and *MaryCalls*, respectively.

10) Give an example of probabilistic Inference.

In this section we will describe a simple method for probabilistic inference-that is, the computation from observed evidence of posterior probabilities for query propositions. We will use the full joint distribution as the "knowledge base" fi-om which answers to all questions may be derived. Along the way we will also inlroduce several useful techniques for manipulating equations involving probabilities.

We begin with a very simple example: a domain consisting of just the three Boolean variables *Toothache, Cavity,* and *Catch* (the dentist's nasty steel probe catches in my tooth). The full joint distribution is a 2 x 2 x 2 table as shown Iln Figure 13.3.

	toothache		*¬toothache*	
	catch	*¬catch*	*catch*	*¬catch*
cavity	0.108	0.012	0.072	0.008
¬cavity	0.016	0.064	0.144	0.576

Figure 13.3 A full joint distribution for the Toothache, Cavity, *Catch* world.

Notice that the probabilities in the joint distribution sum to 1, as required by the axioms of probability. Notice also that Equation (13.2) gives us a direct way to calculate the probability of any proposition, simple or complex: We slimply identify those atomic events in which the proposition is true and add up their probabilities. For example, there are six atomic events in which *cavity* $\vee$ *toothache* holds:

$P(cavity \vee toothache)$ = 0.108 + 0.012 + 0.072 + 0.008 + 0.016 + 0.064 == 0.28

One particularly common task is to extract the distribution over some subset of variables or a single variable. For example, adding the entries in the first row gives the uncondiltional or MARGINAL PROBABILITY marginal probability8 of *cavity:*

$P(cavity)$ = 0.108 + 0.012 + 0.072 + 0.008 = 0.2 .

Conditional probabilities can be defined in terms of unconditional probabilities. The defining equation is

$$P(a|b) = \frac{P(a \wedge b)}{P(b)} \qquad (13.1)$$

which holds whenever $P(b) > 0$. This equation can also be written as

$$P(a \wedge b) = P(a|b)P(b)$$

which is called the **product rule.** The product rule is perhaps easier to remember: it comes from the fact that, for a and b to be true, we need b to be true, and we also need a to be true given b. We can also have it the other way around:

$$P(a \wedge b) = P(b|a)P(a) \ .$$

In some cases, it is easier to reason in terms of prior probabilities of conjunctions, but for the most part, we will use conditional probabilities as our vehicle for probabilistic inference.

In most cases, we will be interested in computing *conditional* probabilities of some variables, given evidence about others. Conditional probabilities can be found by first using Equation (13.1) to obtain an expression in terms of unconditional probabilities and then evaluating the expression from the full joint distribution. For example, we can compute the probability of a cavity, given evidence of a toothache, as follows:

$$P(cavity|toothache) = \frac{P(cavity \wedge toothache)}{P(toothache)} = \frac{0.108 + 0.012}{0.108 + 0.012 + 0.016 - 0.064} = 0.6 \ .$$

Just to check, we can also compute the probability that there is no cavity, given a toothache:

$$P(\neg cavity|toothache) = \frac{P(\neg cavity \wedge toothache)}{P(toothache)} = \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} = 0.4 \ .$$

UNIT-V Learning

Learning from Observations – Inductive Learning – Decision Trees – Explanation based Learning – Statistical Learning Methods – Reinforcement Learning

UNIT-V Question and Answers

(1) What is machine learning?

Machine learning

- Like human learning from past experiences,a computer does not have “experiences”.
- A computer system learns from data, which represent some “past experiences” of an application domain.
- Objective of machine learning : learn a target function that can be used to predict the values of a discrete class attribute, e.g., approve or not-approved, and high-risk or low risk.
- The task is commonly called: **Supervised learning, classification, or inductive learning**

(2) Define Supervised learning .

Supervised learning is a machine learning technique for learning a function from training data. The training data consist of pairs of input objects (typically vectors), and desired outputs. The output of the function can be a continuous value (called regression), or can predict a class label of the input object (called classification). The task of the supervised learner is to predict the value of the function for any valid input object after having seen a number of training examples (i.e. pairs of input and target output). To achieve this, the learner has to generalize from the presented data to unseen situations in a "reasonable" way.

Another term for supervised learning is **classification**. Classifier performance depend greatly on the characteristics of the data to be classified. There is no single classifier that works best on all given problems. Determining a suitable classifier for a given problem is however still more an art than a science. The most widely used classifiers are the **Neural Network** (Multi-layer Perceptron), **Support Vector Machines, k-Nearest Neighbors, Gaussian Mixture Model, Gaussian, Naive Bayes, Decision Tree and RBF classifiers.**

(3) **Compare Supervised learning and unsupervised learning.**.

Supervised vs. unsupervised Learning

- **Supervised learning**:

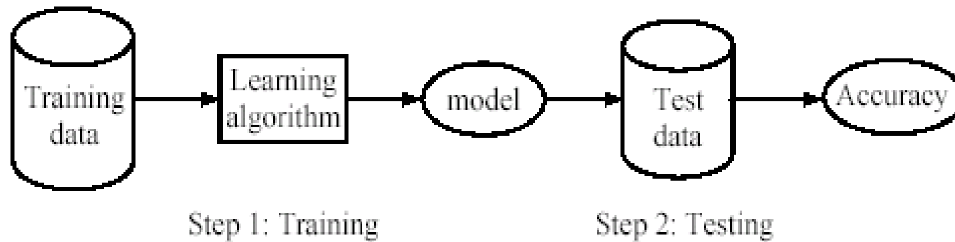
  classification is seen as supervised learning from examples.
  - **Supervision**: The data (observations, measurements, etc.) are labeled with pre-defined classes. It is like that a “teacher” gives the classes (supervision).
  - **Test data** are classified into these classes too.
- **Unsupervised learning** (clustering)
  - Class labels of the data are unknown
  - Given a set of data, the task is to establish the existence of classes or clusters
  - in the data

(4) Explain the steps in Supervised learning process.

Supervised learning process: two steps

- **Learning** (training): Learn a model using the training data
- **Testing**: Test the model using unseen test data to assess the model accuracy

$$Accuracy = \frac{\text{Number of correct classifications}}{\text{Total number of test cases}},$$



(5) What is a decision tree?

A decision tree takes as input an object or situation described by a set of attributes and returns a "decision" – the predicted output value for the input.

A decision tree reaches its decision by performing a sequence of tests.

Example : "HOW TO" manuals (for car repair)

(6) **Explain the Decision Tree learning with an example.**

A decision tree reaches its decision by performing a sequence of tests. Each internal node in the tree corresponds to a test of the value of oine of the properties, and the branches from the node are labeled with the possible values of the test. Each leaf node in the tree specifies the value to be returned if that leaf is reached. The decision tree representation seems to be very natural for humans; indeed, many "How To" manuals (e.g., for car repair) are written entirely as a single decision tree stretching over hundreds of pages.

A somewhat simpler example is provided by the problem of whether to wait for a table at a restaurant. The aim here is to learn a definition for the **goal predicate** Will Wait. In setting this up as a learning problem, we first have to state what attributes are available to describe examples in the domain.

we will see how to automate this task; for now, let's suppose we decide on the following list of attributes:

1. Alternate: whether there is a suitable alternative restaurant nearby.
2. Bar: whether the restaurant has a comfortable bar area to wait in.
3. Fri/Sat: true on Fridays and Saturdays.
4. *Hungry:* whether we are hungry.
5. Patrons: how many people are in the restaurant (values are None, Some, and Full).
6. Price: the restaurant's price range ($, $$, $$$).
7. Raining: whether it is raining outside.
8. Reservation: whether we made a reservation.
9. Type: the kind of restaurant (French, Italian, Thai, or burger).
10. WaitEstimate: the wait estimated by the host (0-10 minutes, 10-30, 30-60, >60).

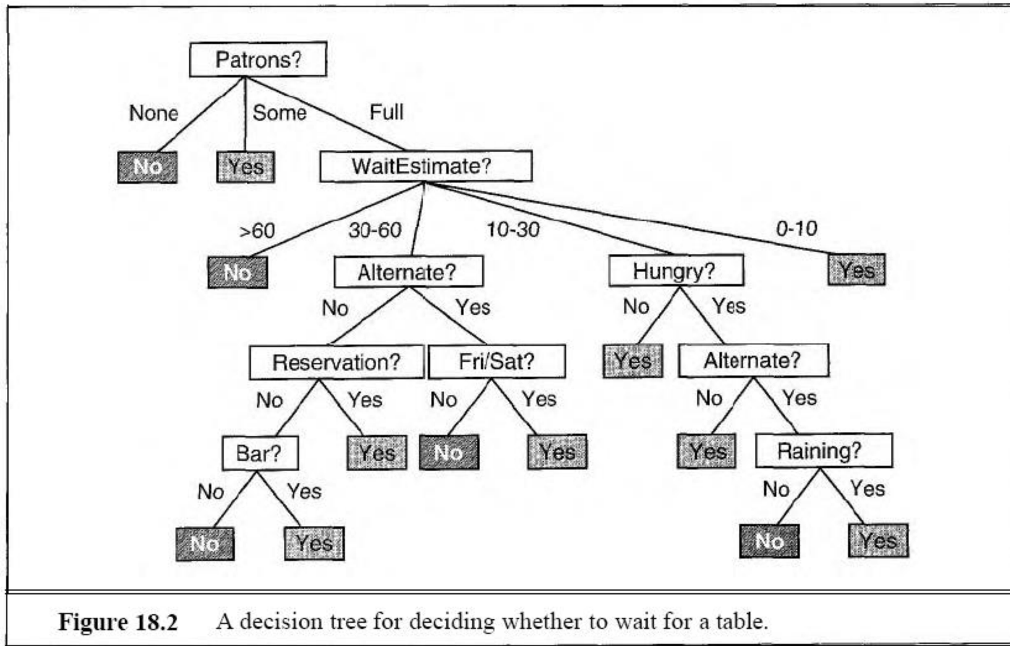


Figure 18.2 A decision tree for deciding whether to wait for a table.

(7) Give an example of decision tree induction from examples.

An example for a Boolean decision tree consists of a vector of' input attributes, X, and a single Boolean output value y. A set of examples (X1,Y1) . . . , (X2, y2) is shown in Figure 18.3. The positive examples are the ones in which the goal *Will Wait* is true (XI, ***X3,*** . . .); the negative examples are the ones in which it is false (X2, *X5,* . . .). The complete set of examples is called the **training set.**

Example	Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	*Yes*
X_2	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*3040*	*No*
X_3	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	*Yes*
X_4	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	*Yes*
X_5	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	*No*
X_6	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	*Yes*
X_7	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	*No*
X_8	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	*Yes*
X_9	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	*No*
X_{10}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	*No*
X_{11}	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	*No*
X_{12}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	*Yes*

Figure 18.3 Examples for the restaurant domain.

(8) Explain the Decision Tree Algorithm for an example problem.

The basic idea behind the Decision-Tree-Learning-Algorithm is to test the most important attribute first. By "most important," we mean the one that makes the most difference to the classification of an example. That way, we hope to get to the correct classification with a small number of tests, meaning that all paths in the tree will be short and the tree as a whole will be small.

```
function DECISION-TREE-LEARNING(examples, attribs, default) returns a decision tree
  inputs: examples, set of examples
          attrzbs, set of attributes
          default, default value for the goal predicate

  if examples is empty then return default
  else if all examples have the same classification then return the classification
  else if attrzbs is empty then return MAJORITY-VALUE(examples)
  else
     best ← CHOOSE-ATTRIBUTE(attribs, examples)
     tree ← a new decision tree with root test best
     m ← MAJORITY-VALUE(examples)
     for each value v_i of best do
        examples_i ← {elements of examples with best = v_i}
        subtree ← DECISION-TREE-LEARNING(examples_i, attribs – best, m)
        add a branch to tree with label v_i and subtree subtree
     return tree
```

Figure 18.5 The decision tree learning algorithm.

(9) How the performance of learning algorithm is assessed?

Assessing the performance of the learning algorithm

A learning algorithm is good if it produces hypotheses that do a good job of predicting the classifications of unseen examples.

Obviously, a prediction is good if it turns out to be true, so we can assess the quality of a hypothesis by checking its predictions against the correct classification once we know it. We do this on a set of examples known as the **test set.** If we train on all our available examples, then we will have to go out and get some more to test on, so often it is more convenient to adopt the following methodology:

1. Collect a large set of examples.
2. Divide it into two disjoint sets: the **training set** and the **test set.**
3. Apply the learning algorithm to the training set, generating a hypothesis h.
4. Measure the percentage of examples in the test set that are correctly classified by h.
5. Repeat steps 2 to 4 for different sizes of training sets and different randomly selected training sets of each size.

The result of this procedure is a set of data that can be processed to give the average prediction quality as a function of the size of the training set. This function can be plotted on a graph, giving what is called the **learning curve** for the algorithm on the particular domain.

(10) What are learning curves?

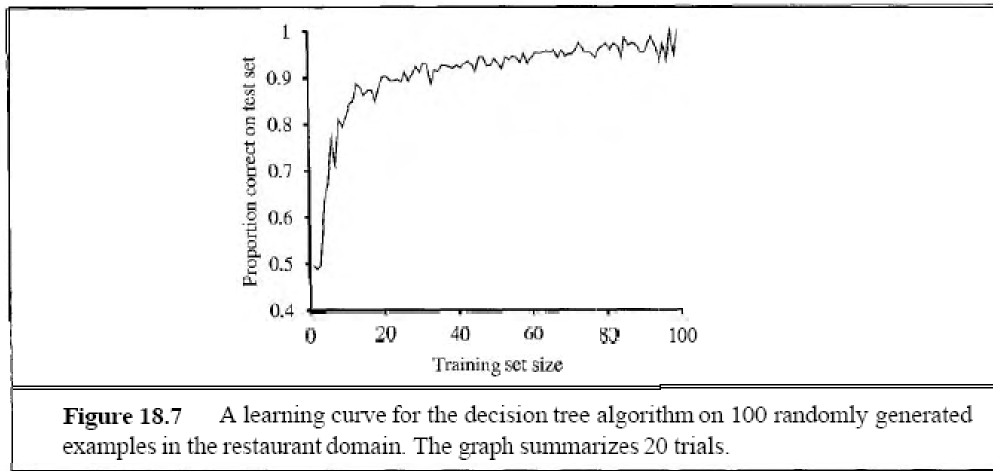


Figure 18.7 A learning curve for the decision tree algorithm on 100 randomly generated examples in the restaurant domain. The graph summarizes 20 trials.

.The average predicting quality of an algorithm can be plotted as a function of the size of the training set as shown in Fig 18.7. This is called a learning curve.
It can be noticed that, as the training set grows, the prediction quality increases. (For this reason, such curves are also called **happy graphs.**) This is a good sign that there is indeed some pattern in the data and the learning algorithm is picking it up.

(11) **What is Ensemble Learning?**

Ensemble means a group producing a single effect.
Ensemble learning is the process by which multiple models, such as classifiers or experts, are strategically generated and combined to solve a particular computational intelligence problem. Ensemble learning is primarily used to improve the (classification, prediction, function approximation, etc.) performance of a model, or reduce the likelihood of an unfortunate selection of a poor one.
Recently in the area of machine learning the concept of combining classifiers is proposed as a new direction for the improvement of the performance of individual classifiers. These classifiers could be based on a variety of classification methodologies, and could achieve different rate of correctly classified individuals. The goal of classification result integration algorithms is to generate more certain, precise and accurate system results.

Ensemble Classification

Aggregation of predictions of multiple classifiers with the goal of improving accuracy.

(12) What is Explanation based Learning?

Explanation based learning is a method for extracting **general rules** from individual observations. The basic idea behind EBL is first construct an explanation of the observation using prior knowledge.
EXAMPLE – A caveman toasting a lizard on the end of a pointed stick:
In the case of lizard toasting,the cavemen generalize by explaining the success of the pointed stick : It supports the lizard while keeping the hands away from the fire. From this explanation,they infer a general rule : that any long,rigid,sharp object can be used to toast small,soft-bodies edibles.
The general rules follows logically from the background knowledge possessed by the cavemen.

(13) What is Relevance Based Learning?

Relevance-based learning (RBL) uses prior knowledge in the form of determinations to identify the relevant attributes, thereby generating a reduced hypothesis space and speeding up learning. RBL also allows deductive generalizations from single examples.

EXAMPLE

Consider the case of the traveler to Brazil meeting her first Brazilian. On hearing him speak Portugese,she immediately conclude that Brazilians speak Portugese. The inferen ce was based on her Background Knowledge,namely,that people in a given country speak the same language. The same is expressed in First Order Predicate Logic as follows :

$$Nationality(x,n) \wedge Nationality(y,n) \wedge Language(x,l) \Rightarrow Language(y,l) \,. \quad \text{(A)}$$

"If *x* and y have the same nationality *n* and *x* speaks language *1,* then y also speaks it."

Sentences such as (A) express a strict form of relevance: given nationality, language is fully determined. (Put another way: language is a function of nationality.) These sentences are called **functional dependencies or determinations.** They occur so commonly in certain kind of applications (e.g., defining database designs)

(14) Explain with a diagram the Cumulative Learning Process.

The use of prior knowledge in learning leads to a picture of **cumulative learning,** in which learning agents improve their learning ability as they acquire more knowledge.

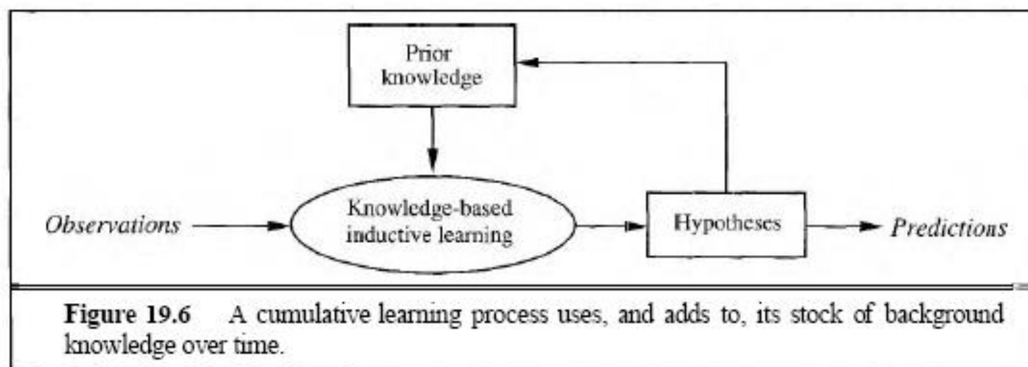


Figure 19.6 A cumulative learning process uses, and adds to, its stock of background knowledge over time.

The modem approach is to design agents that *already know something* and are trying to learn some more The general idea is shown schematically in Figure 19.6.

(15) **What is inductive logic programming?**

Inductive Logic Programming(ILP) combines inductive methods with the power of First Order Representation**.** The object of Inductive learning program is to come up with a set of sentences for the Hypotheses such that entailment constraint is satisfied.

EXAMPLE

The family tree example:

The descriptions

Father(Philips,Charles)

Father(Philips,Anne)

Parent(x,y) ⇔ [Mother(x,y) v Father(x,y)]
Grandparent(x,y) ⇔ [∃z Parent(x,z) ^ Parent(z,y)]

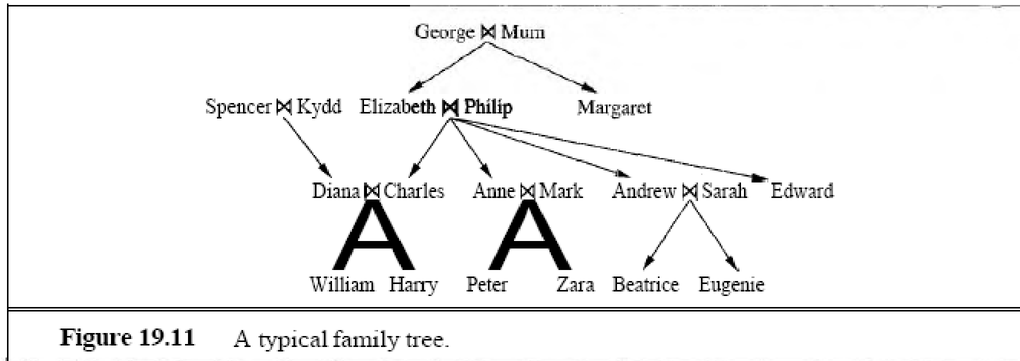


Figure 19.11 A typical family tree.

(16) What is Bayesian Learning?

Bayesian learning simply calculates the probability of each hypothesis,given the data,and makes predictions on that basis. That is,the predictions are made by using all the hypotheses,weighted by their probabilities,rather than using just ahigh "best" hypothesis. In this way learning is reduced to probabilistic inference.

(17) What are neural networks?

Traditionally, the term **neural network** had been used to refer to a network or circuit of biological neurons[citation needed]. The modern usage of the term often refers to artificial neural networks, which are composed of artificial neurons or nodes.

Artificial neural networks are made up of interconnecting artificial neurons (programming constructs that mimic the properties of biological neurons). Artificial neural networks may either be used to gain an understanding of biological neural networks, or for solving artificial intelligence problems.

In the artificial intelligence field, artificial neural networks have been applied successfully to speech recognition, image analysis and adaptive control, in order to construct software agents (in computer and video games) or autonomous robots. Most of the currently employed artificial neural networks for artificial intelligence are based on statistical estimation, optimization and control theory.

A **Neural network** is an artificial system (made of artificial neuron cells). It is modeled after the way the human brain works. Several computing cells work in parallel to produce a result. This is usually seen as one of the possible ways Artificial intelligence can work. Most neural network can tolerate if one or more of the processing cells fail.

What is important in the idea of neural networks is that they are able to learn by themselves, an ability which makes them remarkably distinctive in comparison to normal computers, which cannot do anything for which they are not programmed.

(18) Explain the units in the neural network.

Units in neural networks

Neural networks are composed of nodes or **units** (see Figure 2:0.15) connected by directed **links. A** link from unit j to unit i serves to propagate lhe **activation** *aj* from **j** to *i.* Each link also has a numeric **weight** *Wj,i* ssociated with it, which determine the strength and sign of the connection. Each unit i first computes a weighted sum of its inputs:

Then it applies an **activation function** g to this sum to derive the output:

$$a_i = g(in_i) = g\left(\sum_{j=0}^{n} W_{j,i}a_j\right) . \qquad (20.10)$$

Notice that we have included a **bias weight** $W_{0,i}$ connected to a fixed input $a_0 = -1$. We will explain its role in a moment.

The activation function g is designed to meet two desiderata. **First,** we want the unit to be "active" (near +1) when the "right" inputs are given, and "inactive" (near **0**) when the "wrong" inputs are given. Second, the activation needs to be nonlinear, otherwise the entire neural network collapses into a simple linear function (see Exercise 20.17). Two choices for g

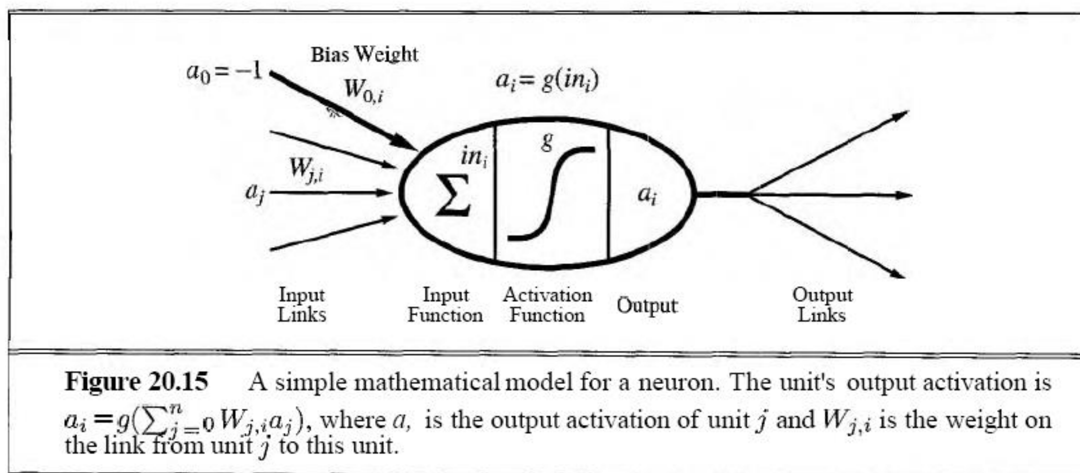


Figure 20.15 A simple mathematical model for a neuron. The unit's output activation is $a_i = g(\sum_{j=0}^{n} W_{j,i}a_j)$, where a_j is the output activation of unit j and $W_{j,i}$ is the weight on the link from unit j to this unit.

are shown in Figure 20.16: the **threshold** function and the **sigmoid function** (also known as the **logistic function).** The sigmoid function has the advantage of being differentiable, which we will see later is important for the weight-learning algorithm. Notice that both functions have a threshold (either hard or soft) at zero; the bias weight $W_{0,i}$ sets the *actual* threshold for the unit, in the sense that the unit is activated when the weighted sum of "real" inputs $\sum_{j=1}^{n} W_{j,i}a_j$ (i.e., excluding the bias input) exceeds $W_{0,i}$.

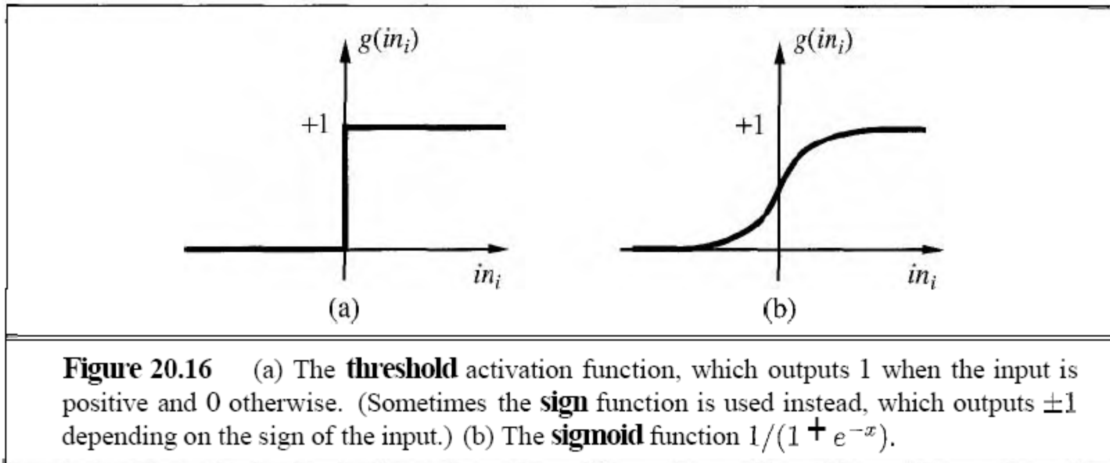


Figure 20.16 (a) The **threshold** activation function, which outputs 1 when the input is positive and 0 otherwise. (Sometimes the **sign** function is used instead, which outputs ±1 depending on the sign of the input.) (b) The **sigmoid** function $1/(1+e^{-x})$.

(19) What is reinforcement learning?

Reinforcement learning refers to a class of problems in machine learning which postulate an agent exploring an environment in which the agent perceives its current state and takes actions. The environment, in return, provide a a reward (which can be positive or negative). Reinforcement learning algorithms attempt to find a policy for maximising cumulative reward for the agent over the curse of the problem (Wikipedia).

Examples

Playing chess:

Reward comes at end of game

Ping-pong:

Reward on each point scored

Animals:

Hunger and pain - negative reward
food intake – positive reward

(20) What is passive reinforcement learning?

Passive Reinforcement Learning

- Assume fully observable environment.
- Passive learning:
  - Policy is fixed (behavior does not change).

  - The agent learns how good each state is.
- Similar to policy evaluation, but:
  - Transition function and reward function or unknown.
- Why is it useful?
  - For future policy revisions.

(21) What is active reinforcement learning?

Active Reinforcement Learning

- Using passive reinforcement learning, utilities of states and transition probabilities are learned.
- Those utilities and transitions can be plugged into Bellman equations.
- Problem?
  - Bellman equations give optimal solutions given correct utility and transition functions.
  - Passive reinforcement learning produces approximate estimates of those functions.

(22) What is machine learning?

Machine learning is the subfield of artificial intelligence that is concerned with the design and development of algorithms that allow computers to improve their performance over time based on data, such as from sensor data or databases. A major focus of machine learning research is to automatically produce (induce) models, such as rules and patterns, from data. Hence, machine learning is closely related to fields such as data mining, statistics, inductive reasoning, pattern recognition, and theoretical computer science.