
CS8691 - ARTIFICIAL INTELLIGENCE

VI SEMESTER CSE

2017 Regulations

UNIT-I

UNIT I **INTRODUCTION** **9**

Introduction–Definition - Future of Artificial Intelligence – Characteristics of Intelligent Agents– Typical Intelligent Agents – Problem Solving Approach to Typical AI problems.

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	1 CPU,10^8 gates	10^{11} neurons
Storage units	10^{10} bits RAM 10^{11} bits disk	10^{11} neurons 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 "Comput-ing 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 programs. 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

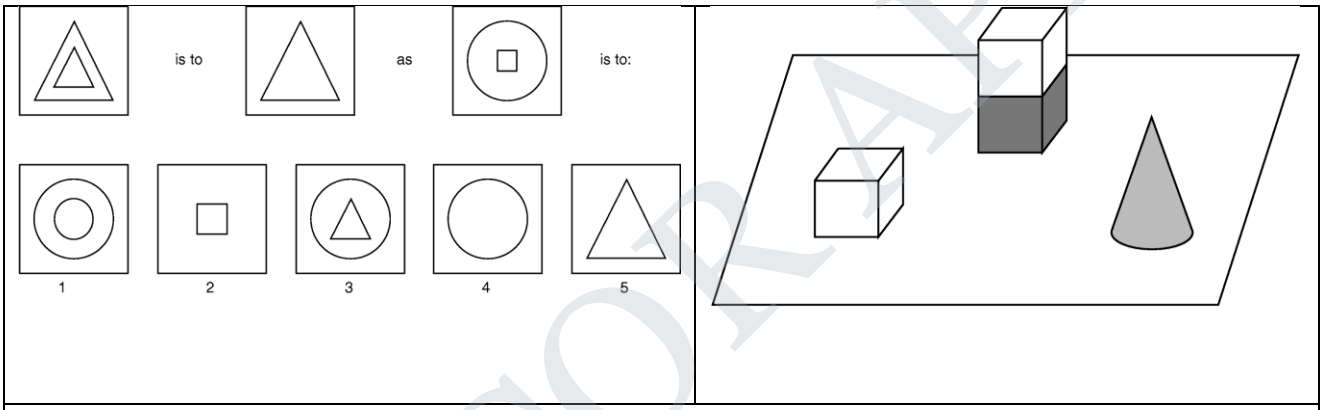


Figure 1.1 The Tom Evan's ANALOGY program could solve geometric analogy problems as shown.

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.

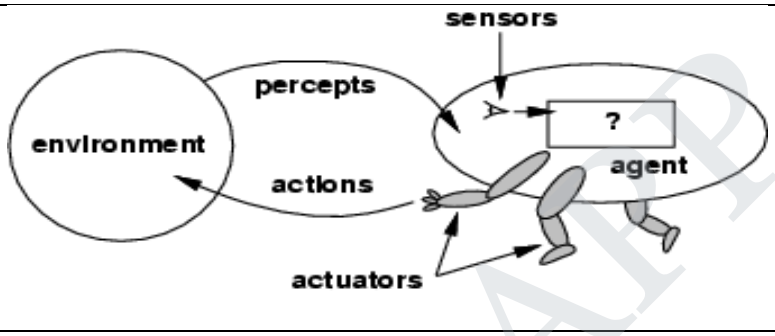


Figure 1.2 Agents interact with environments through sensors and actuators.

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.

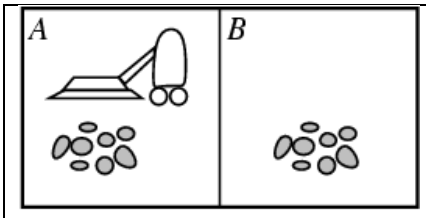


Figure 1.3 A vacuum-cleaner world with just two locations.

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, maximize profits	Roads,other traffic,pedestrians, customers	Steering,accelerator, brake, Signal,horn,display	Cameras,sonar, Speedometer,GPS, Odometer,engine sensors,keyboards, accelerometer

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

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

Figure 1.6 Examples of agent types and their PEAS descriptions.

Properties of task environments

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

Fully observable vs. **partially observable.**

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

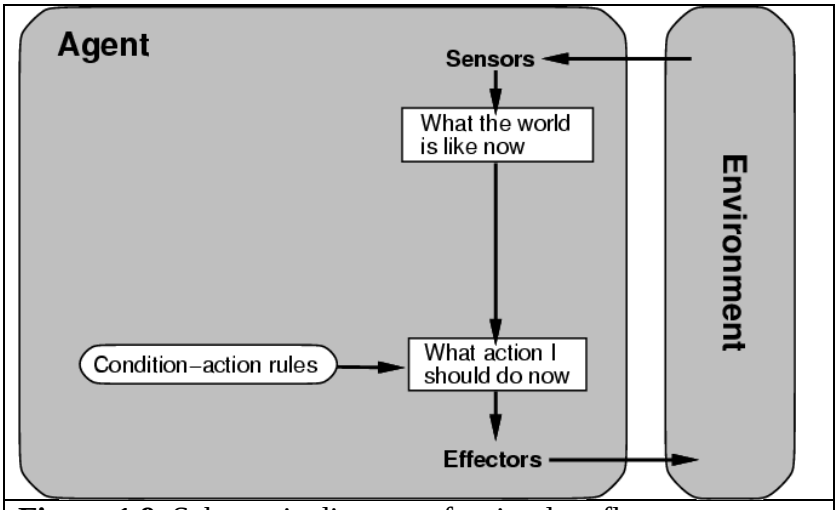


Figure 1.9 Schematic diagram of a simple reflex agent.

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.**

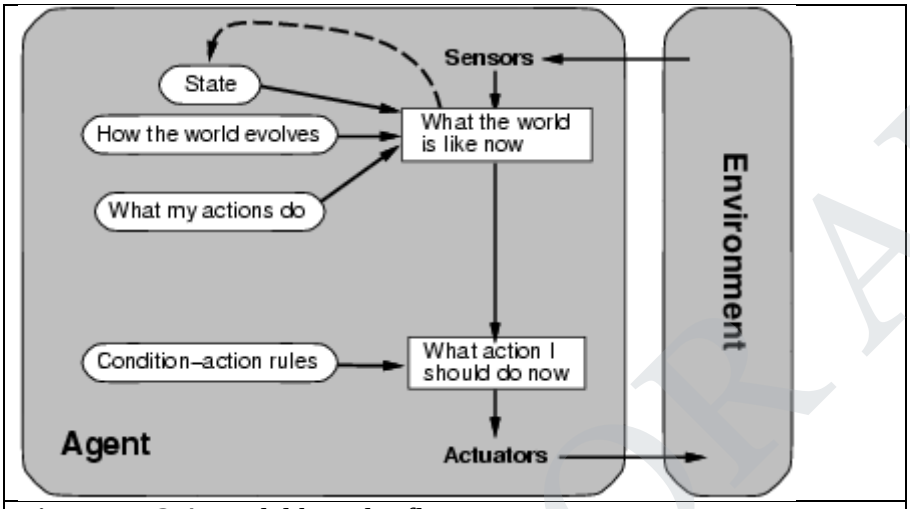


Figure 1.12 A model based reflex agent

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.

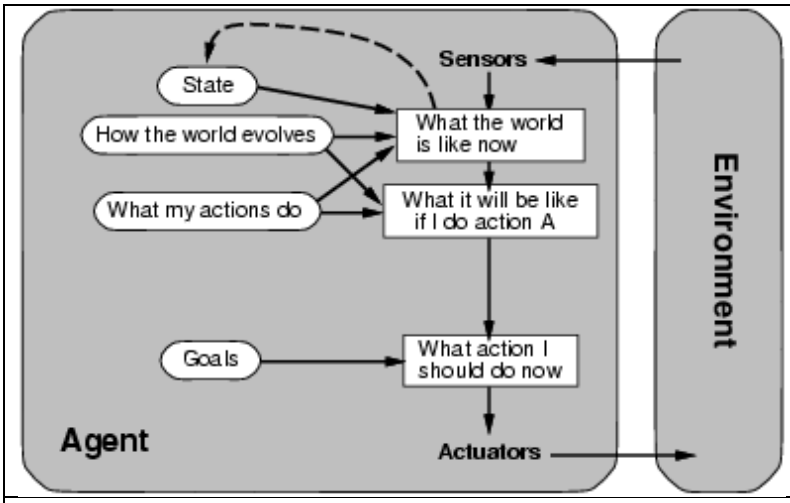


Figure 1.14 A goal based agent

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.

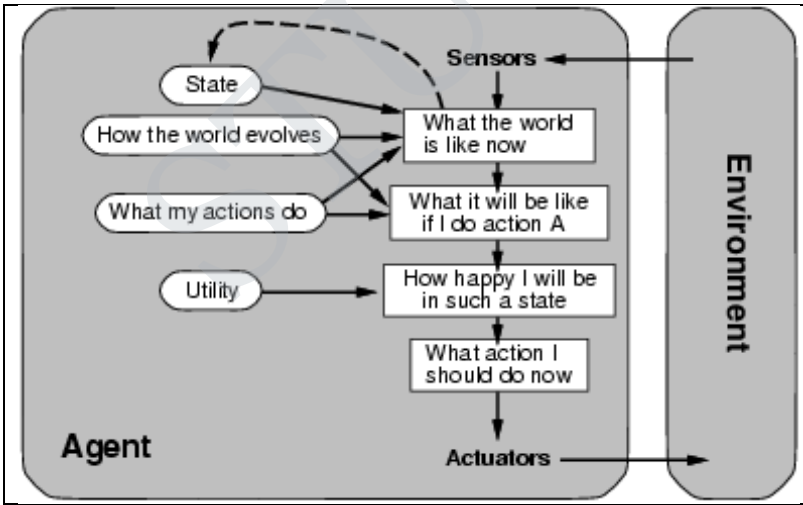


Figure 1.15 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome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

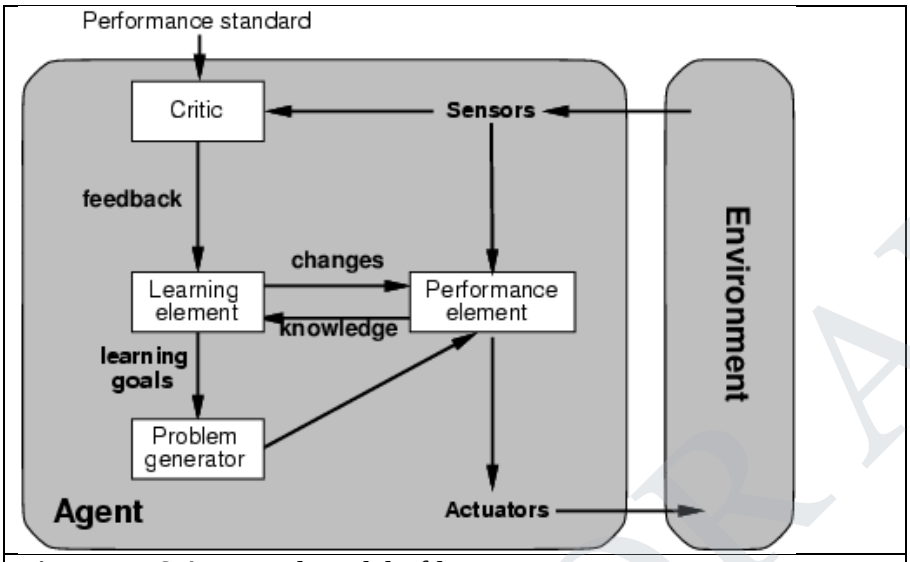


Figure 1.16 A general model of learning agents.

- 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.**

CS8691 –ARTIFICIAL INTELLIGENCE

VI SEMESTER CSE

2017 Regulations

UNIT-II

(2) SEARCHING TECHNIQUES

UNIT II PROBLEM SOLVING METHODS 9

Problem solving Methods - Search Strategies- Uninformed - Informed - Heuristics - Local Search Algorithms and Optimization Problems - Searching with Partial Observations – Constraint Satisfaction Problems – Constraint Propagation - Backtracking Search - Game Playing – Optimal Decisions in Games – Alpha - Beta Pruning - Stochastic Games

---

2.0 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.

2.0.1 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.

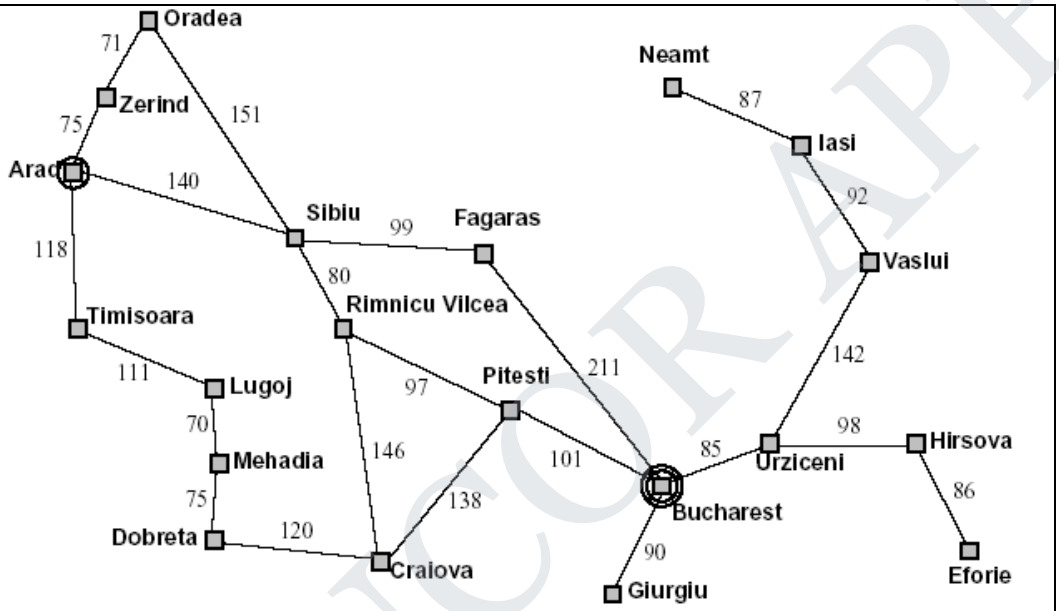


Figure 1.19 A simplified Road Map of part of Romania

2.0.2 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.

2.0.2.1 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

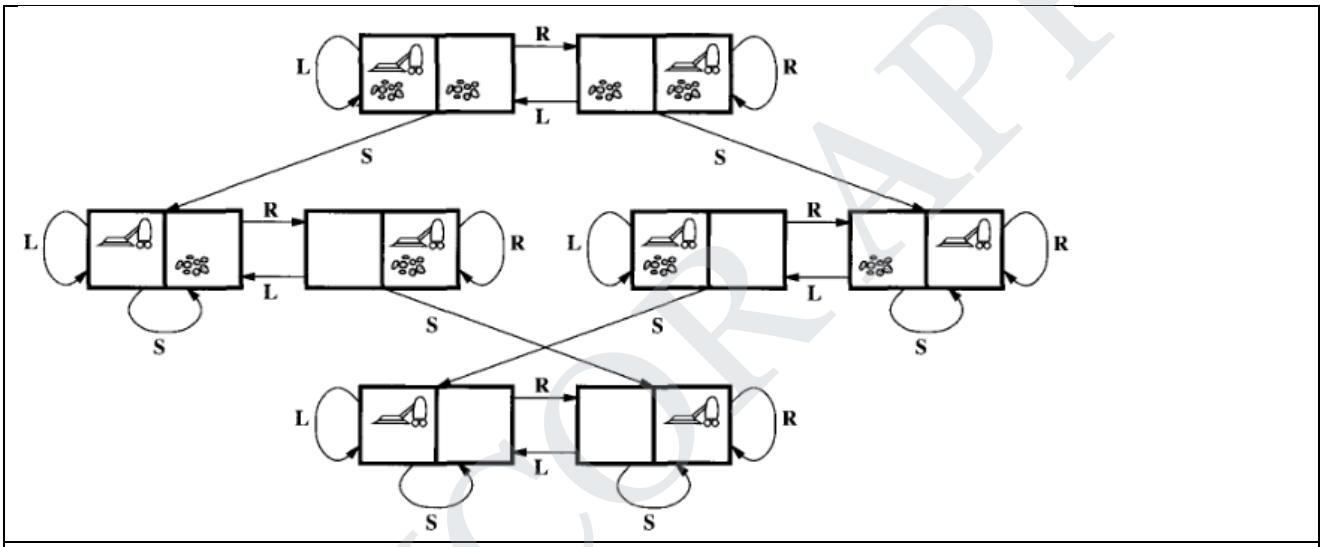


Figure 1.20 The state space for the vacuum world.
Arcs denote actions: L = Left,R = Right,S = Suck

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

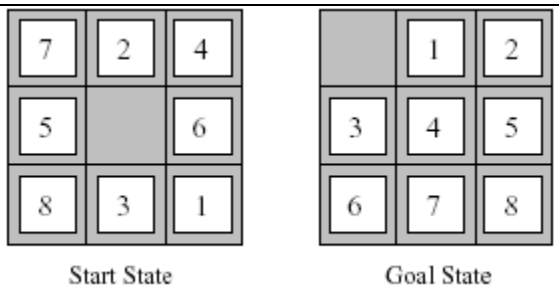


Figure 1.21 A typical instance of 8-puzzle.

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.

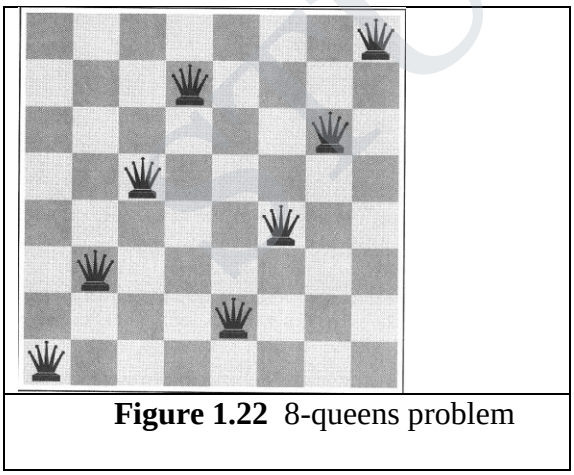


Figure 1.22 8-queens problem

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.

2.0.1 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.

2.0.2 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.

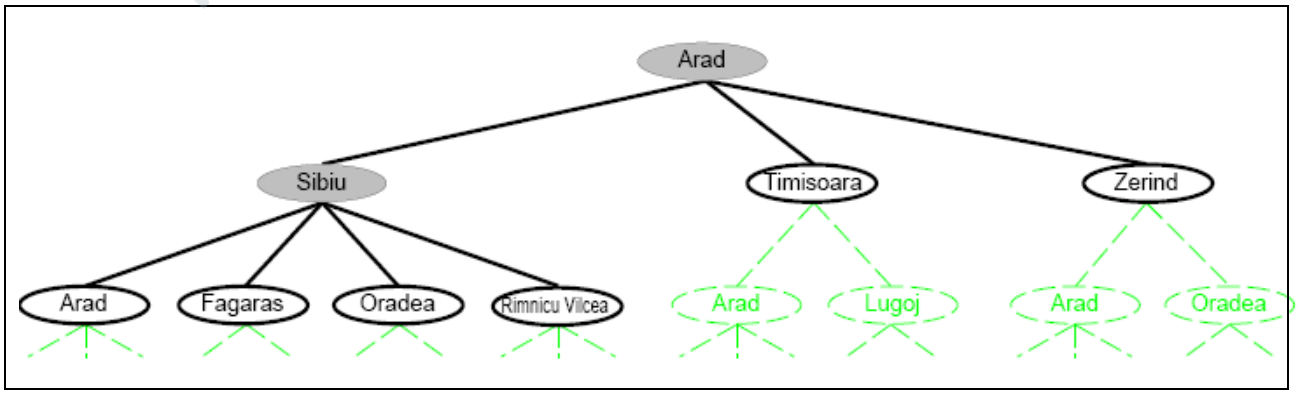


Figure 1.23 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded.; nodes that have been generated but not yet expanded are outlined in bold;nodes that have not yet been generated are shown in faint dashed line

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.

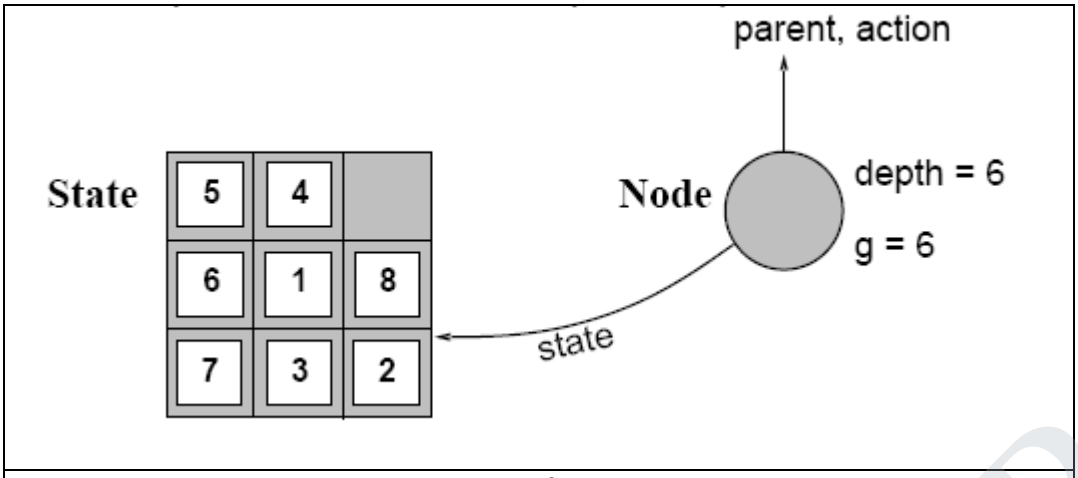


Figure 1.25 Nodes are data structures from which the search tree is constructed. Each has a parent,a state, Arrows point from child to parent.

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?

2.0.3 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

2.3.4.1 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.

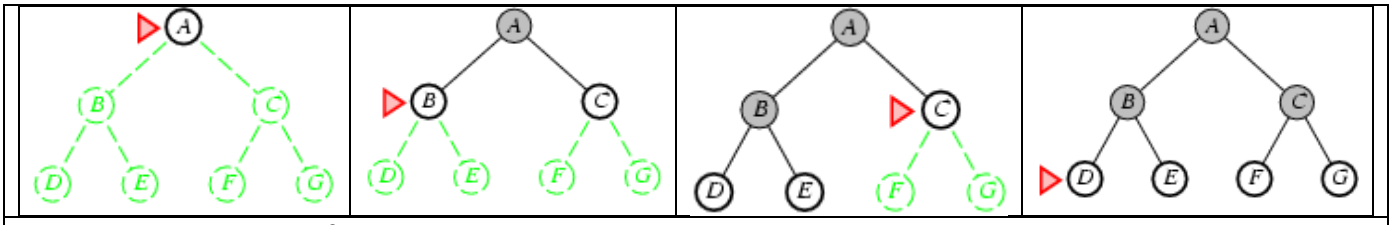


Figure 1.27 Breadth-first search on a simple binary tree. At each stage ,the node to be expanded next is indicated by a marker.

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

2.3.4.2 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.

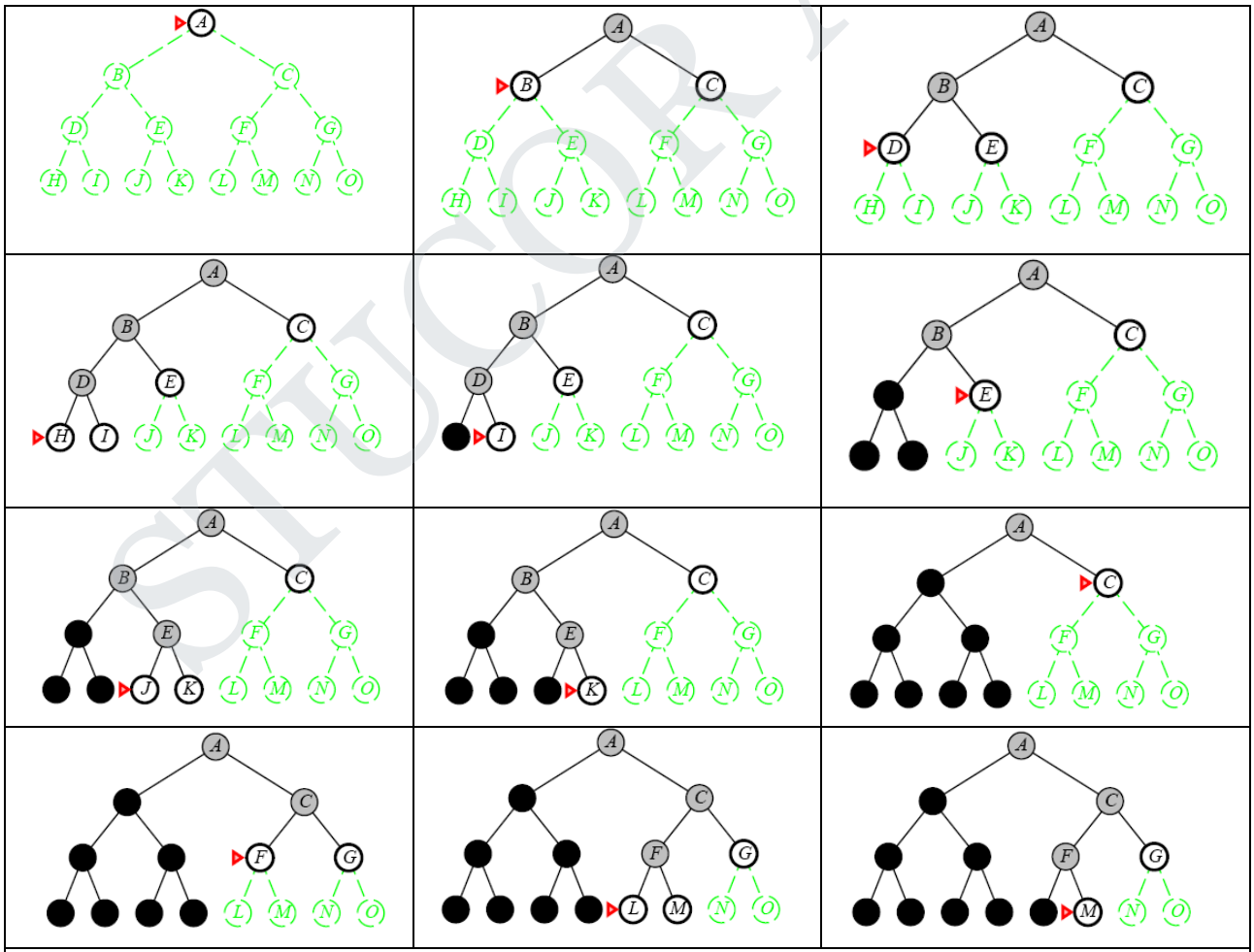


Figure 1.31 Depth-first-search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from the memory;these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

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)

2.3.4.4 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:

2.3.4.5 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.

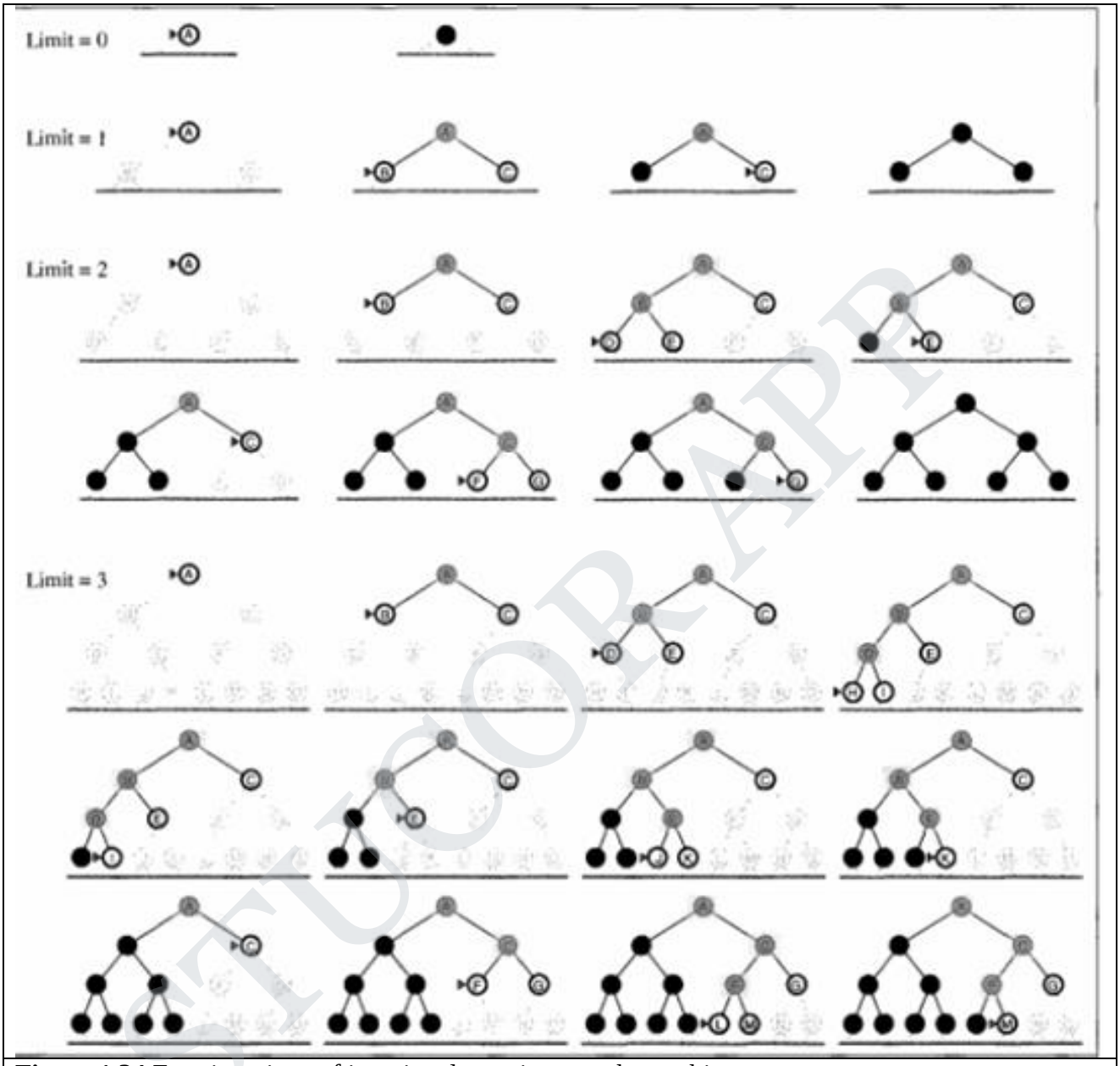


Figure 1.34 Four iterations of iterative deepening search on a binary tree

Iterative search is not as wasteful as it might seem

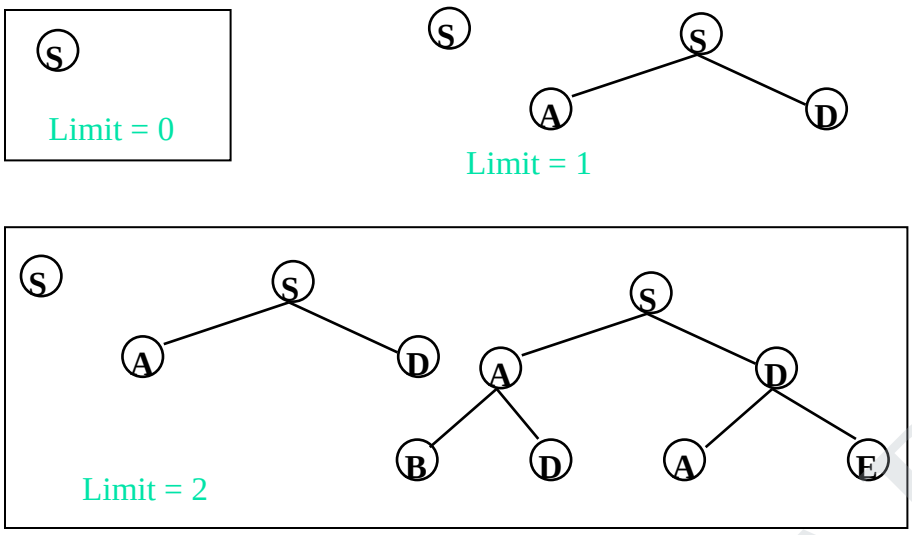


Figure 1.35

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.

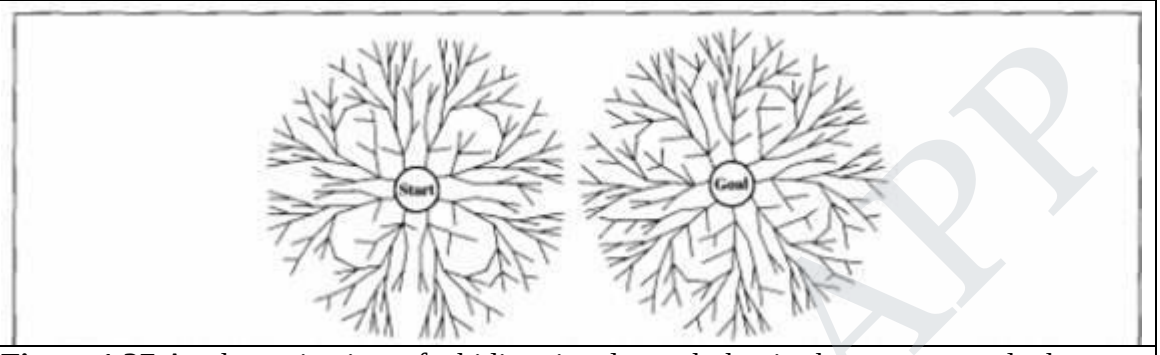


Figure 1.37 A schematic view of a bidirectional search that is about to succeed,when a Branch from the Start node meets a Branch from the goal node.

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.

2.3.2 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!

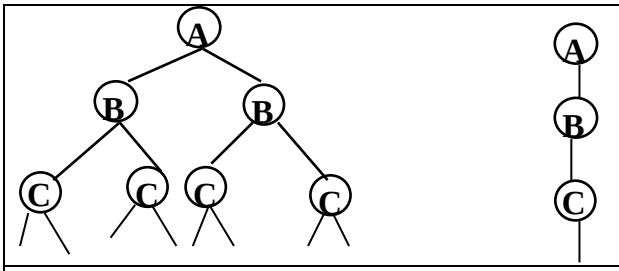


Figure 1.39

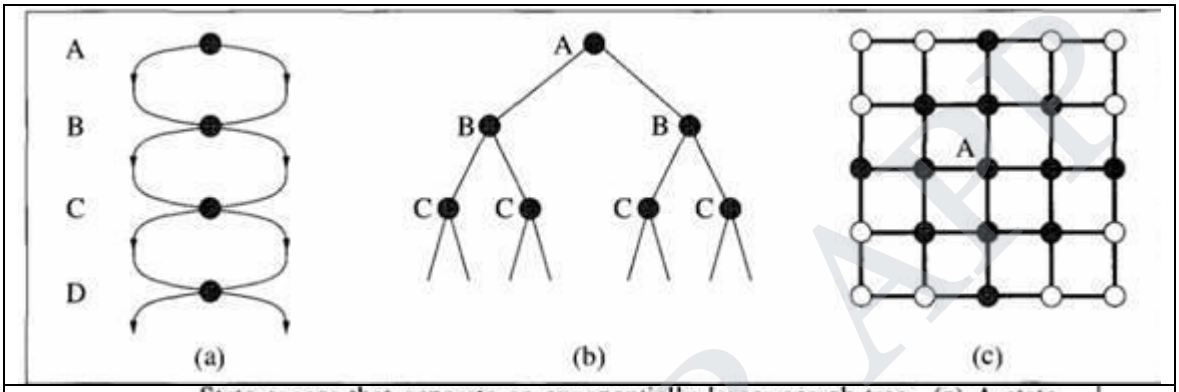


Figure 1.40 State spaces that generate an exponentially larger search tree. (a) A state space in which there are two possible actions leading from A to B, two from B to C, and so on. The state space contains $d + 1$ states, where d is the maximum depth. (b) The corresponding search tree, which has 2^d 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.

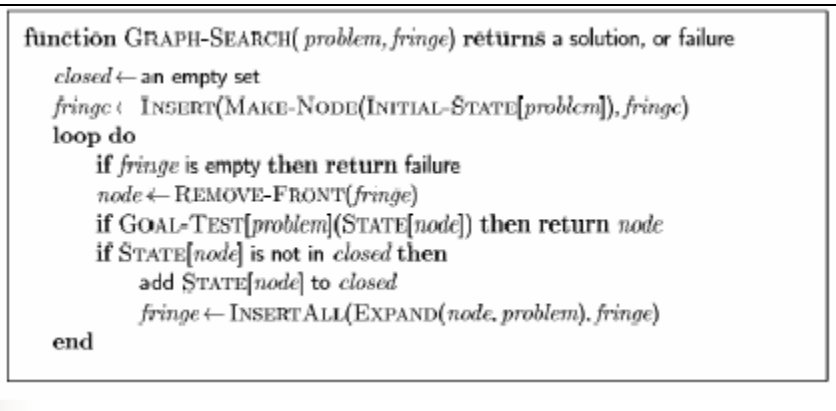


Figure 1.41 The General graph search algorithm. The set closed can be implemented with a hash table to allow efficient checking for repeated states.

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 $O(b^d)$.

2.3.2 SEARCHING WITH PARTIAL INFORMATION

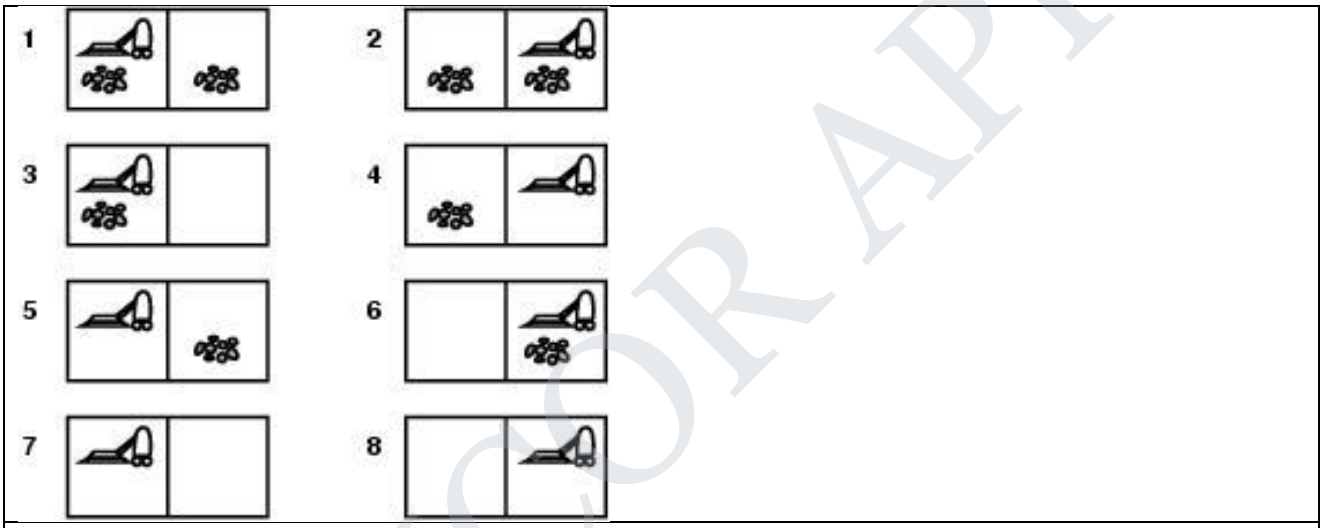
- Different types of incompleteness lead to three distinct problem types:
  - **Sensorless problems** (conformant): If the agent has no sensors at all
  - **Contingency problem**: if the environment if partially observable or if action are uncertain (adversarial)
  - **Exploration problems**: When the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

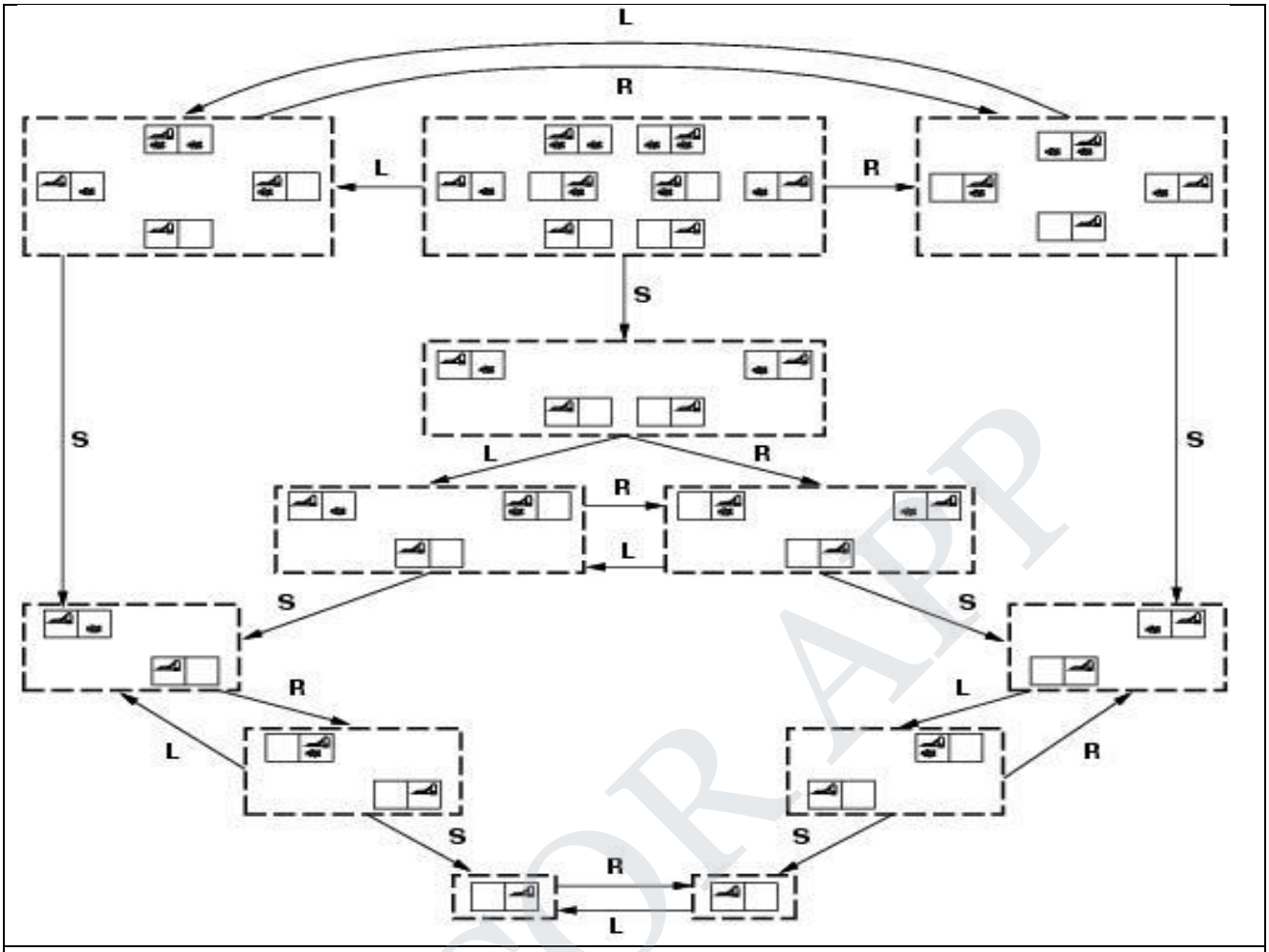
- Answer : [Right,Suck,Left,Suck] coerce the world into state 7 without any sensor
- Belief State: Such state that agent belief to be there

(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.

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.

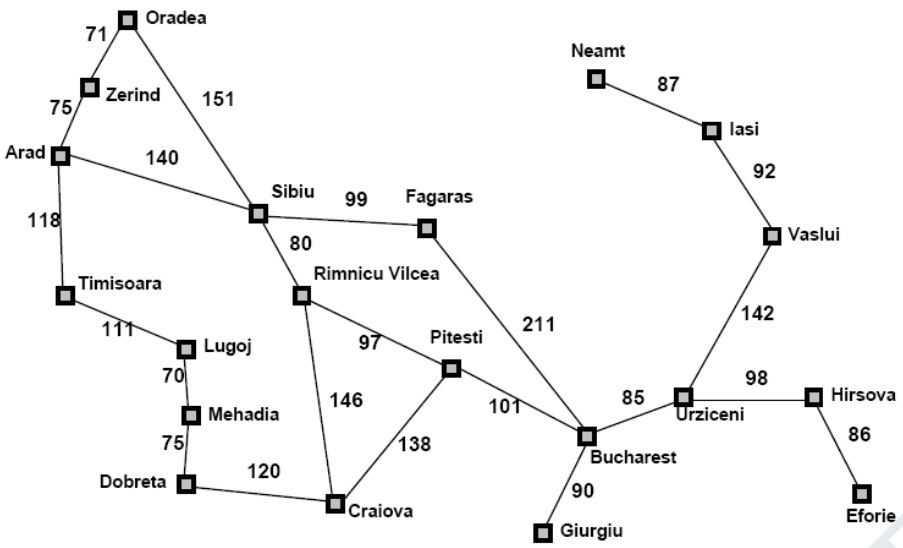
Greedy Best-first search

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.



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

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

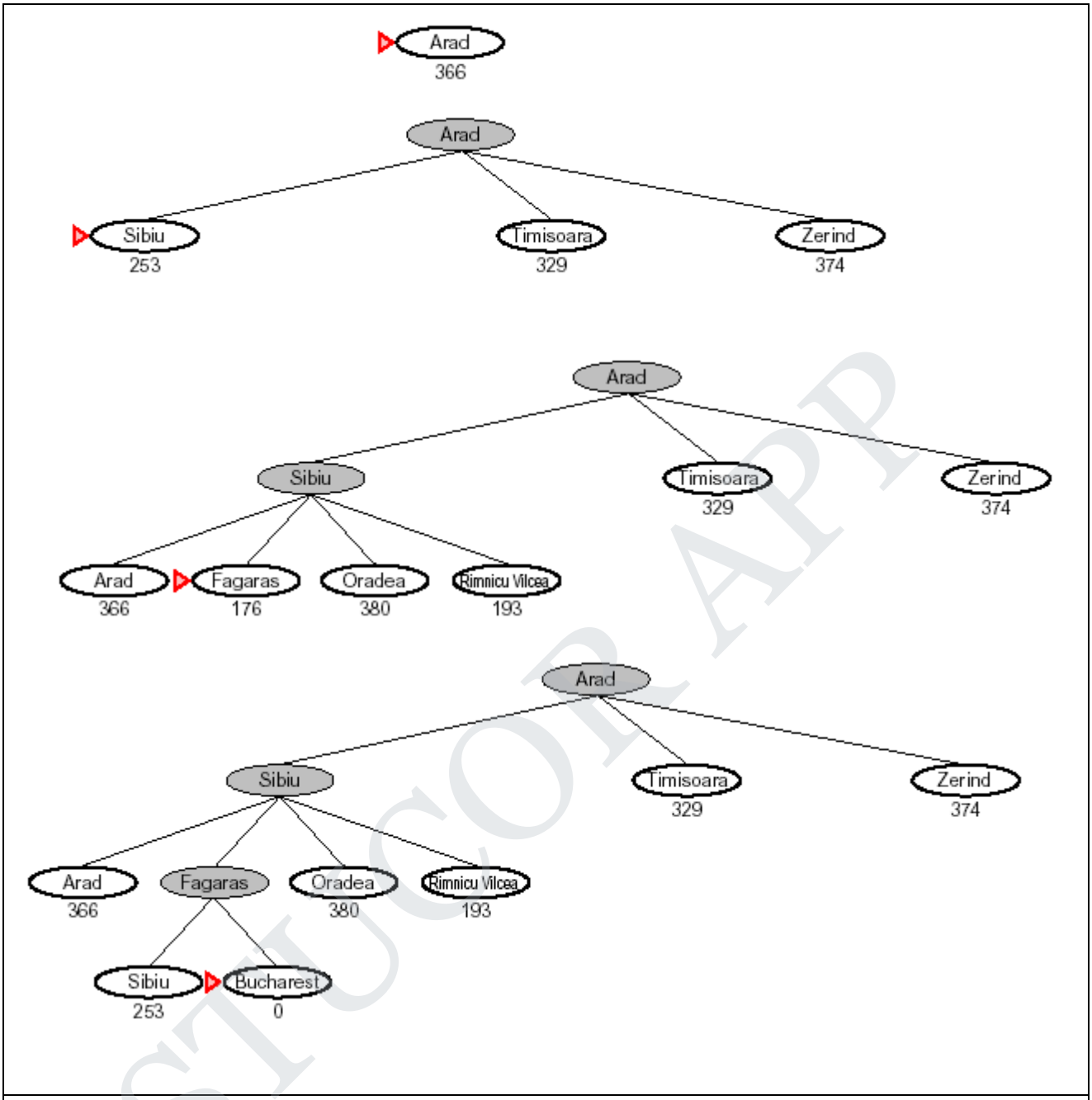


Figure 2.2 stages in greedy best-first search for Bucharest using straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

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.

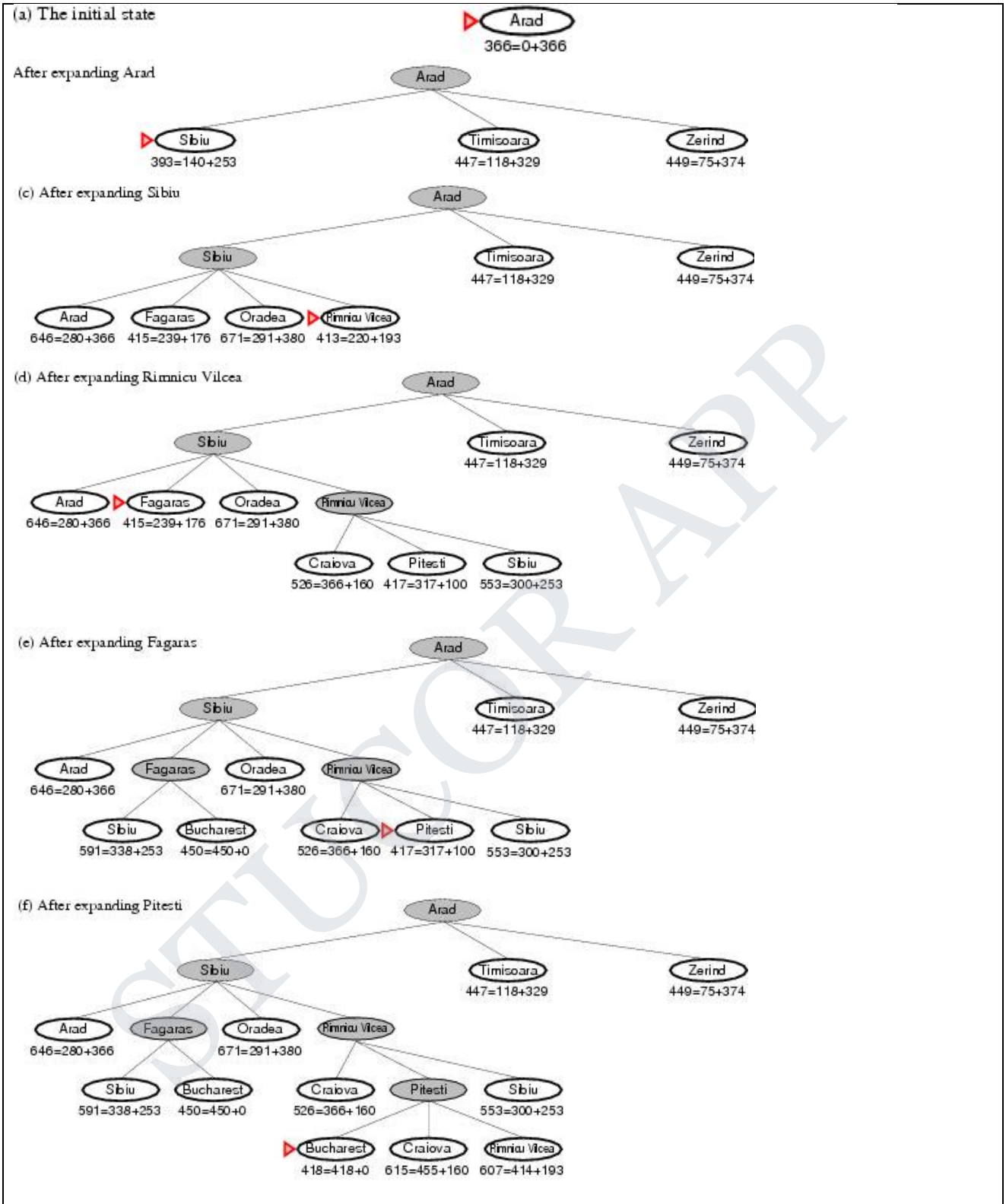


Figure 2.3 Stages in A* Search for Bucharest. Nodes are labeled with f = g + h . The h-values are the straight-line distances to Bucharest taken from figure 2.1

```
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

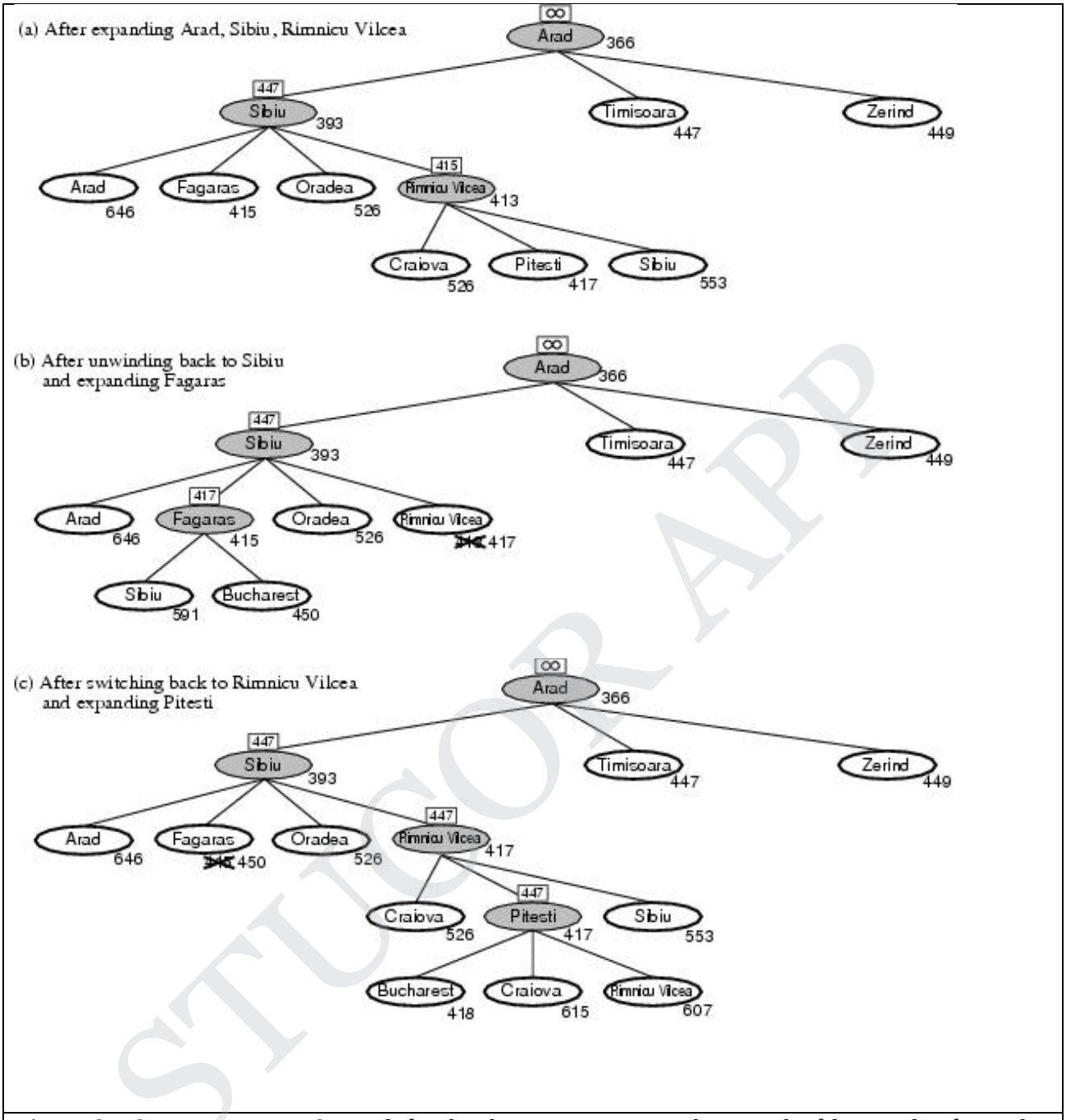


Figure 2.5 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (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

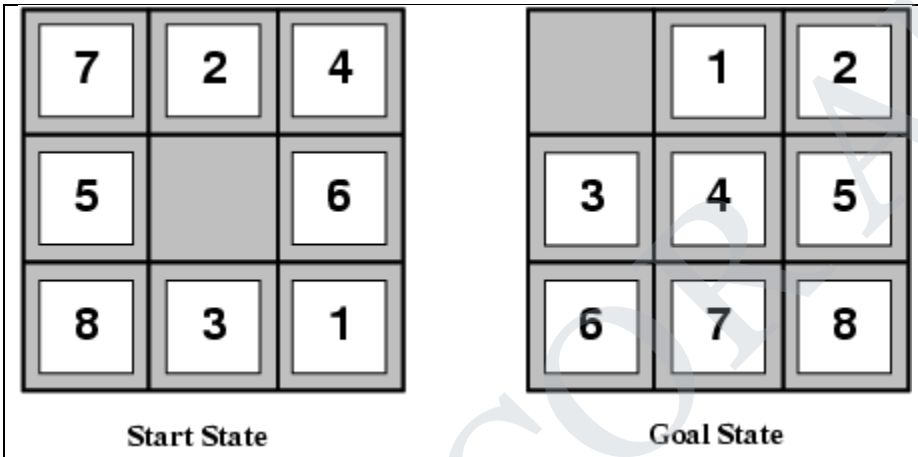


Figure 2.6 A typical instance of the 8-puzzle.
The solution is 26 steps long.

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**.

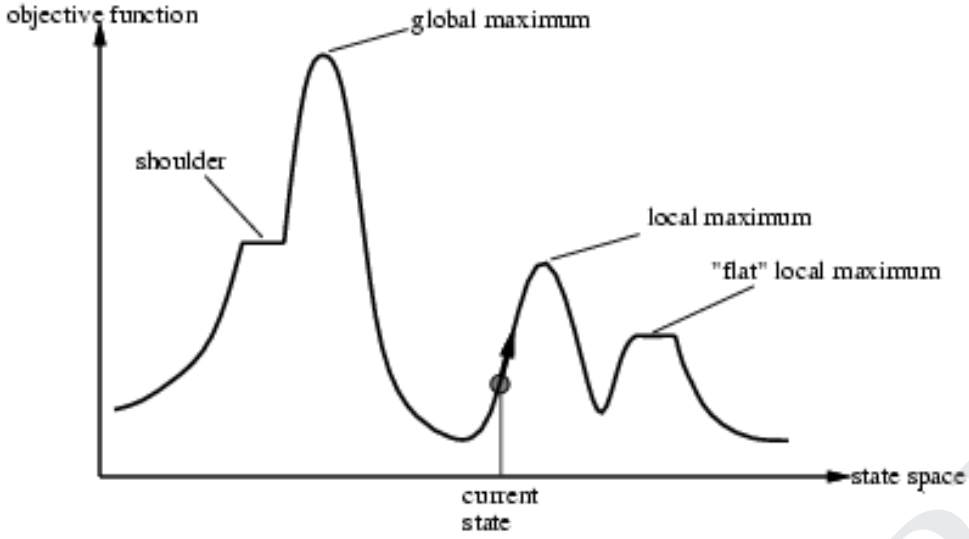


Figure 2.8 A one dimensional **state space landscape** in which elevation corresponds to the **objective function**. The aim is to find the global maximum. Hill climbing search modifies the current state to try to improve it ,as shown by the arrow. The various topographic features are defined in the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

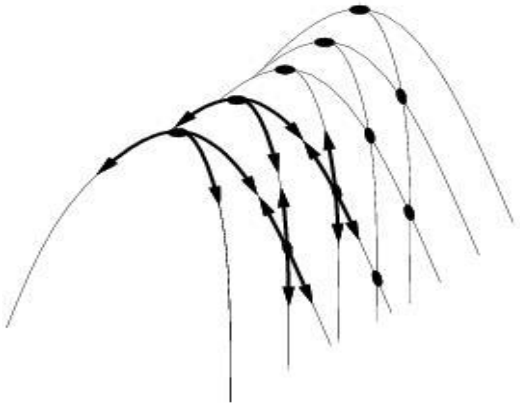


Figure 2.10 Illustration of why ridges cause difficulties for hill-climbing. The grid of states(dark circles) is superimposed on a ridge rising from left to right,creating a sequence of local maxima that are not directly connected to each other. From each local maximum,all th available options point 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 $\triangle$ 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 x $\log_2$ 8 = 24 bits.

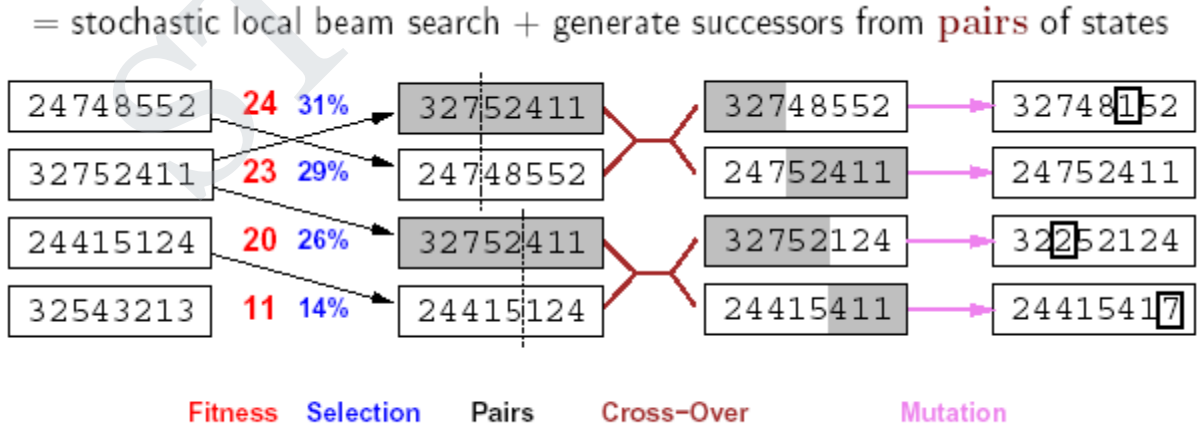


Figure 2.12 The genetic algorithm. The initial population in (a) is ranked by the fitness function in (b),resulting in pairs for mating in (c). They produce offspring in (d),which are subjected to

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.

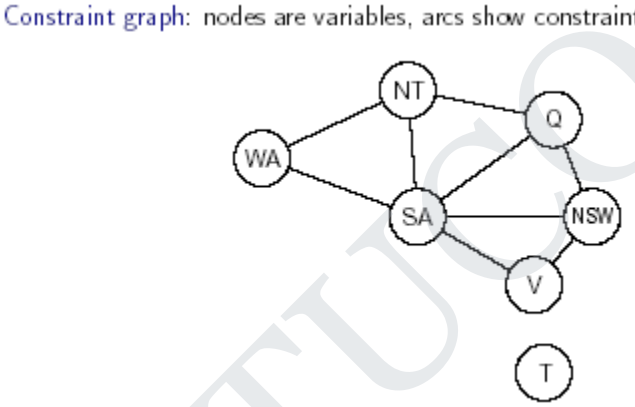


Figure 2.15 (b) The map coloring problem represented as a constraint graph.

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.

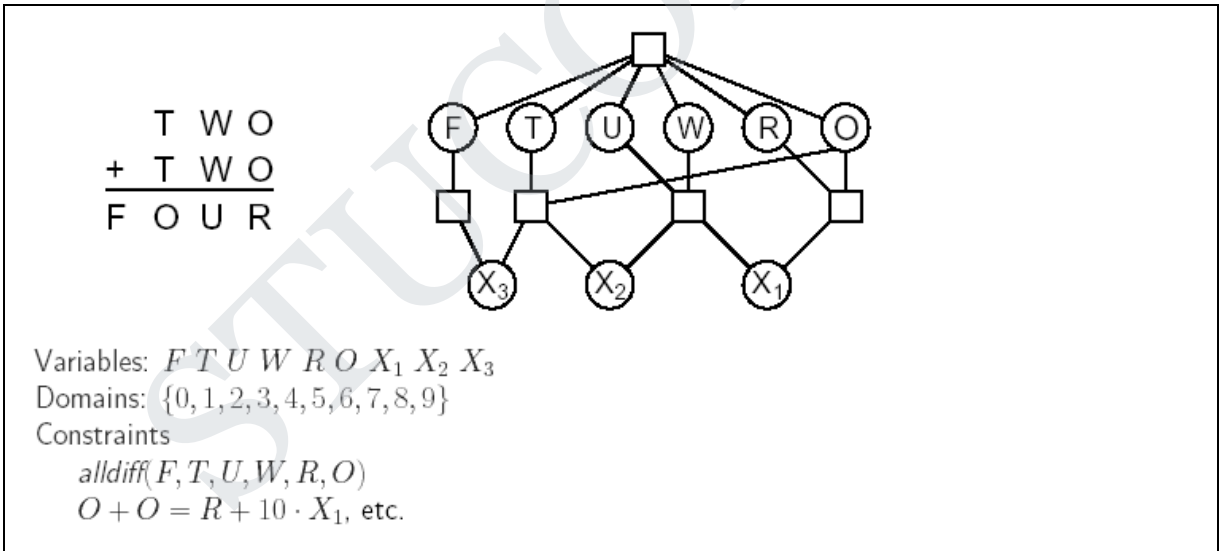


Figure 2.16 (a) Cryptarithmetic problem. Each letter stands for a distinct digit;the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct,with the added restriction that no leading zeros are allowed. (b) The constraint hypergraph for the cryptarithmetic 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

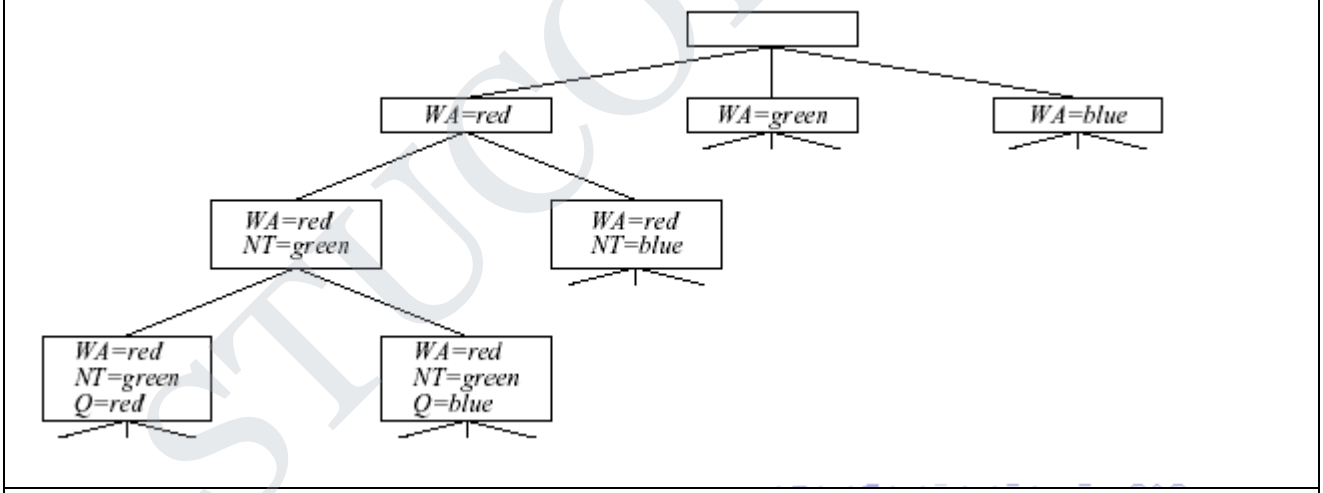


Figure 2.17(b) Part of search tree generated by simple backtracking for the map coloring problem.

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*	(R)	G B	R G B	R G B	R G B	G B	R G B
After *Q*=*green*	(R)	B	(G)	R B	R G B	B	R G B
After *V*=*blue*	(R)	B	(G)	R	(B)		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

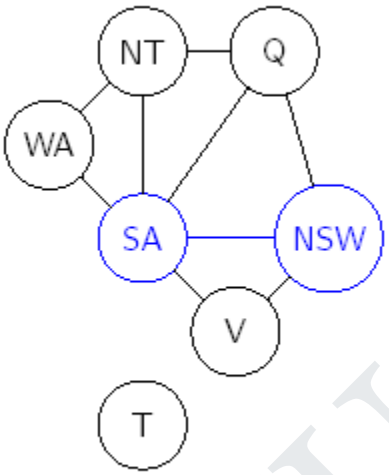


Figure: Australian Territories

- 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

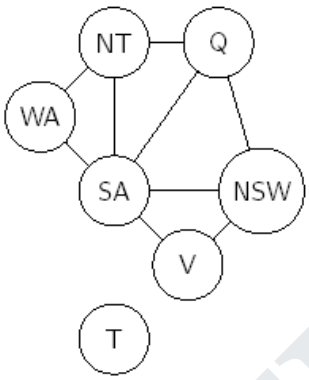


Figure: Australian Territories

- 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

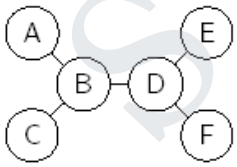


Figure: Tree-Structured CSP

- 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

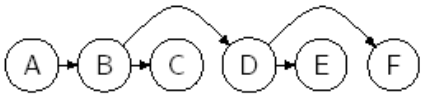


Figure: Linear ordering

2.4 ADVERSARIAL SEARCH

Competetive environments,in which the agent's goals are in conflict,give rise to **adversarial search** problems – often known as **games.**

2.4.1 Games

Mathematical **Game Theory**,a branch of economics,views any **multiagent environment** as a **game** provided that the impact of each agent on the other is "significant",regardless of whether the agents are cooperative or competitive. In,**AI**,"games" are deterministic,turn-taking,two-player,zero-sum games of perfect information. This means deterministic,fully observable environments in which there are two agents whose actions must alternate and in which the **utility values** at the end of the game are always equal and opposite. For example,if one player wins the game of chess(+1),the other player necessarily loses(-1). It is this opposition between the agents' utility functions that makes the situation **adversarial.**

Formal Definition of Game

We will consider games with two players,whom we will call **MAX** and **MIN**. MAX moves first,and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A **game** can be formally defined as a **search problem** with the following components :

- The **initial state**,which includes the board position and identifies the player to move.
- A **successor function**,which returns a list of (*move,state*) pairs,each indicating a legal move and the resulting state.
- A **terminal test**,which describes when the game is over. States where the game has ended are called **terminal states**.
- A **utility function** (also called an objective function or payoff function),which give a numeric value for the terminal states. In chess,the outcome is a win,loss,or draw,with values +1,-1,or 0. he payoffs in backgammon range from +192 to -192.

Game Tree

The **initial state** and **legal moves** for each side define the **game tree** for the game. Figure 2.18 shows the part of the game tree for tic-tac-toe (noughts and crosses). From the initial state,MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing a 0 until we reach leaf nodes corresponding to the terminal states such that one player has three in a row or all the squares are filled. He number on each leaf node indicates the utility value of the terminal state from the point of view of MAX;high values are assumed to be good for MAX and bad for MIN. It is the MAX's job to use the search tree(particularly the utility of terminal states) to determine the best move.

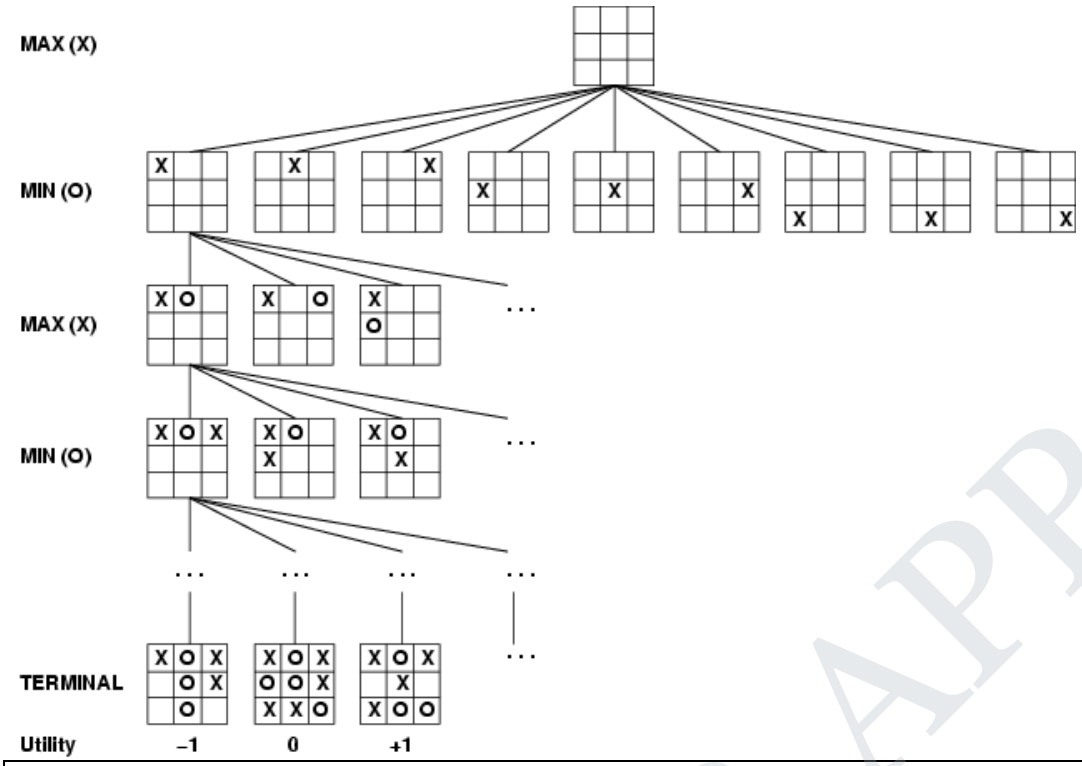


Figure 2.18 A partial search tree . The top node is the initial state,and MAX move first,placing an X in an empty square.

2.4.2 Optimal Decisions in Games

In normal search problem,the **optimal solution** would be a sequence of move leading to a **goal state** – a terminal state that is a win. In a game,on the other hand,MIN has something to say about it,MAX therefore must find a contingent **strategy**,which specifies MAX's move in the **initial state**,then MAX's moves in the states resulting from every possible response by MIN,then MAX's moves in the states resulting from every possible response by MIN those moves,and so on. An **optimal strategy** leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.

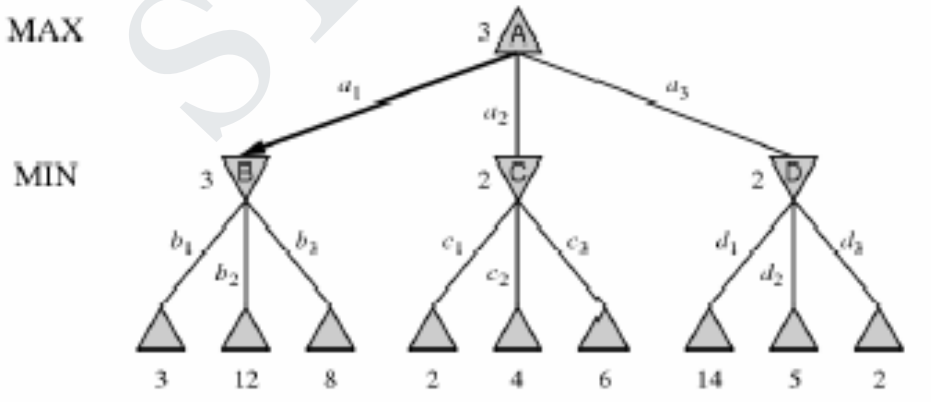


Figure 2.19 A two-ply game tree. The △ nodes are "MAX nodes",in which it is AMX's turn to move,and the ▽ nodes are "MIN nodes". The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1,because it leads to the successor with the highest minimax value,and MIN's best reply is b_1,because it leads to the successor with the lowest minimax value.

Minimax Search: Algorithm

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game

  v ← MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value                    For MAX Node
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s))
  return v
```

```
function MIN-VALUE(state) returns a utility value                    For MIN Node
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s))
  return v
```

Figure 2.20 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move,that is,the move that leads to the outcome with the best utility,under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree,all the way to the leaves,to determine the backed-up value of a state.

The minimax Algorithm

The minimax algorithm(Figure 2.20) computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state,directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree ,and then the minimax values are **backed up** through the tree as the recursion unwinds**.** For example in Figure 2.19,the algorithm first recourses down to the three bottom left nodes,and uses the utitliy function on them to discover that their values are 3,12,and 8 respectively. Then it takes the minimum of these values,3,and returns it as the backed-up value of node B. A similar process gives the backed up values of 2 for C and 2 for D. Finally,we take the maximum of 3,2,and 2 to get the backed-up value of 3 at the root node. The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is *m*,and there are b legal moves at each point,then the time

complexity of the minimax algorithm is $O(b^m)$. The space complexity is O(bm) for an algorithm that generates successors at once.

2.4.3 Alpha-Beta Pruning

The problem with minimax search is that the number of game states it has to examine is **exponential** in the number of moves. Unfortunately,we can't eliminate the exponent,but we can effectively cut it in half. By performing **pruning,**we can eliminate large part of the tree from consideration. We can apply the technique known as **alpha beta pruning** ,when applied to a minimax tree ,it returns the same move as **minimax** would,but **prunes away** branches that cannot possibly influence the final decision.

Alpha Beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:

- α : the value of the best(i.e.,highest-value) choice we have found so far at any choice point along the path of MAX.
- β: the value of best (i.e., lowest-value) choice we have found so far at any choice point along the path of MIN.

Alpha Beta search updates the values of α and β as it goes along and prunes the remaining branches at anode(i.e.,terminates the recursive call) as soon as the value of the current node is known to be worse than the current α and β value for MAX and MIN,respectively. The complete algorithm is given in Figure 2.21.

The effectiveness of alpha-beta pruning is highly dependent on the order in which the successors are examined. It might be worthwhile to try to examine first the successors that are likely to be the best. In such case,it turns out that alpha-beta needs to examine only $O(b^{d/2})$ nodes to pick the best move,instead of $O(b^d)$ for minimax. This means that the effective branching factor becomes sqrt(b) instead of b – for chess,6 instead of 35. Put anotherway alpha-beta cab look ahead roughly twice as far as minimax in the same amount of time.

```
function ALPHA-BETA-SEARCH(state) returns an action
    inputs: state, current state in game
    v ← MAX-VALUE(state, −∞, +∞)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s, α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 2.21 The alpha beta search algorithm. These routines are the same as the minimax routines in figure 2.20,except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β

2.4.4 Imperfect ,Real-time Decisions

The minimax algorithm generates the entire game search space,whereas the alpha-beta algorithm allows us to prune large parts of it. However,alpha-beta still has to search all the way to terminal states for atleast a portion of search space. Shannon's 1950 paper,Programming a computer for playing chess,proposed that programs should **cut off** the search earlier and apply a heuristic **evaluation function** to states in the search,effectively turning nonterminal nodes into terminal leaves. The basic idea is to alter minimax or alpha-beta in two ways :

(1) The utility function is replaced by a heuristic evaluation function EVAL,which gives an estimate of the position's utility,and

(2) the terminal test is replaced by a **cutoff test** that decides when to apply EVAL.

2.4.5 Games that include Element of Chance

Evaluation functions

An evaluation function returns an estimate of the expected utility of the game from a given position,just as the heuristic function return an estimate of the distance to the goal.

Games of imperfect information

- Minimax and alpha-beta pruning require too much leaf-node evaluations.
  May be impractical within a reasonable amount of time.
- SHANNON (1950):
  - Cut off search earlier (replace TERMINAL-TEST by CUTOFF-TEST)
  - Apply heuristic evaluation function EVAL (replacing utility function of alpha-beta)

Cutting off search

Change**:**

- **if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

  into
- **if** CUTOFF-TEST(*state,depth*) **then return** EVAL(*state*)

Introduces a fixed-depth limit *depth*

- Is selected so that the amount of time will not exceed what the rules of the game allow.

When cuttoff occurs, the evaluation is performed.

Heuristic EVAL

Idea: produce an estimate of the expected utility of the game from a given position.

Performance depends on quality of EVAL.

Requirements:

- EVAL should order terminal-nodes in the same way as UTILITY.
- Computation may not take too long.
- For non-terminal states the EVAL should be strongly correlated with the actual chance of winning.

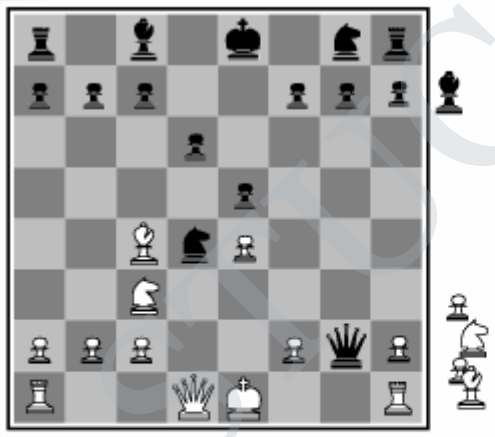
Only useful for quiescent (no wild swings in value in near future) states

Weighted Linear Function

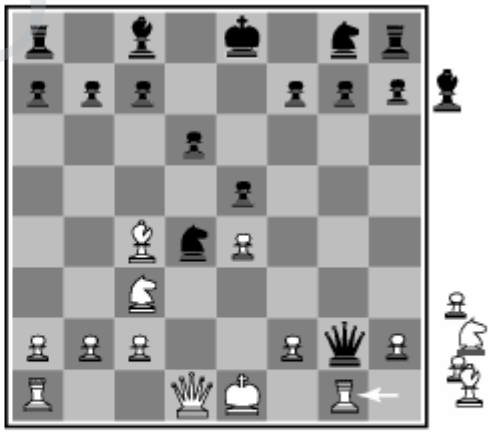
The introductory chess books give an approximate material value for each piece : each pawn is worth 1,a knight or bishop is worth 3,a rook 3,and the queen 9. These feature values are then added up toobtain the evaluation of the position. Mathematically,these kind of evaluation fuction is called weighted linear function,and it can be expressed as :

$$Eval(s) = w1\ f1(s) + w2\ f2(s) + \ldots + wn\ fn(s)$$

- e.g., w1 = 9 with
  f1(s) = (number of white queens) – (number of black queens), etc.



(a) White to move



(b) White to move

(a) Black has an advantage of a knight and two pawns and will win the game.
(b) Black will lose after white captures the queen.

Games that include chance

In real life,there are many unpredictable external events that put us into unforeseen situations. Many games mirror this unpredictability by including a random element,such as throwing a dice. **Backgammon** is a typical game that combines luck and skill. Dice are rolled at the beginning of player's turn to determine the legal moves. The backgammon position of Figure 2.23,for example,white has rolled a 6-5,and has four possible moves.

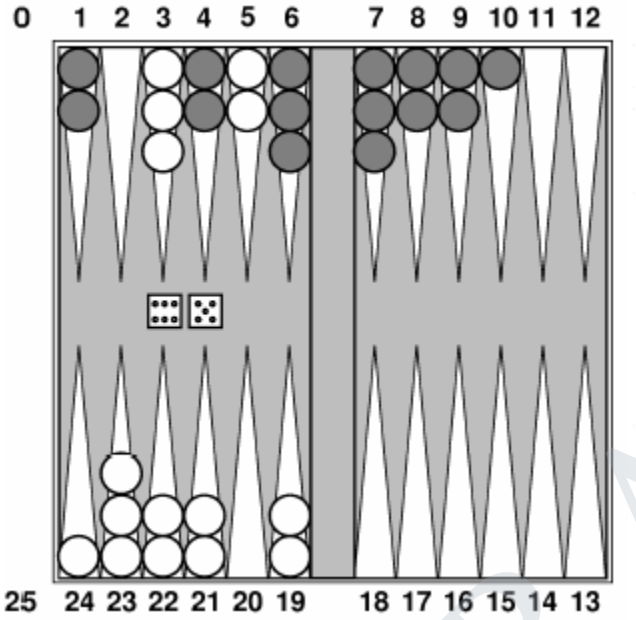


Figure 2.23 A typical backgammon position. The goal of the game is to move all one's pieces off the board. White moves clockwise toward 25,and black moves counterclockwise toward 0. A piece can move to any position unless there are multiple opponent pieces there; if there is one opponent ,it is captured and must start over. In the position shown,white has rolled 6-5 and must choose among four legal moves (5-10,5-11),(5-11,19-24),(5-10,10-16),and (5-11,11-16)

- White moves clockwise toward 25
- Black moves counterclockwise toward 0
- A piece can move to any position unless there are multiple opponent pieces there; if there is one opponent, it is captured and must start over.
- White has rolled 6-5 and must choose among four legal moves: (5-10, 5-11), (5-11, 19-24) (5-10, 10-16), and (5-11, 11-16)

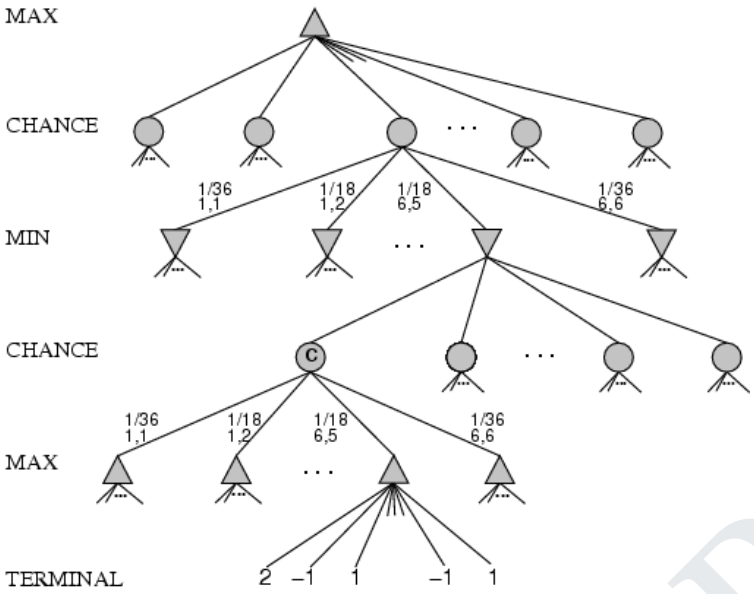


Figure 2-24 Schematic game tree for a backgammon position.

Expected minimax value

EXPECTED-MINIMAX-VALUE(*n*)=

UTILITY(*n*) If *n* is a terminal

max*s* ∈ *successors(n)* MINIMAX-VALUE(*s*) If *n* is a max node

min*s* ∈ *successors(n)* MINIMAX-VALUE(*s*) If *n* is a max node

$\sum s \in successors(n)\ P(s)$. EXPECTEDMINIMAX(*s*) If *n* is a chance node

These equations can be backed-up recursively all the way to the root of the game tree.

Anna University – 2017 regulations
Computer Science and Engineering

VI SEM CSE CS8691 Artificial Intelligence

UNIT III KNOWLEDGE REPRESENTATION 9

First Order Predicate Logic – Prolog Programming – Unification – Forward Chaining-Backward Chaining – Resolution – Knowledge Representation - Ontological Engineering-Categories and Objects – Events - Mental Events and Mental Objects - Reasoning Systems for Categories - Reasoning with Default Information

UNIT-III Question and Answers

(1) How Knowledge is represented?

A variety of ways of knowledge(facts) have been exploited in AI programs.
Facts : truths in some relevant world. These are things we want to represent.

(2) What is propositional logic?

It is a way of representing knowledge.
In logic and mathematics, a **propositional calculus** or **logic** is a formal system in which formulae representing *propositions* can be formed by combining atomic propositions using *logical connectives*
Sentences considered in propositional logic are not arbitrary sentences but are the ones that are either true or false, but not both. This kind of sentences are called **propositions**.

Example

Some facts in propositional logic:
It is raning. - RAINING
It is sunny - SUNNY
It is windy - WINDY

If it is raining ,then it is not sunny - RAINING -> ¬ SUNNY

(3) What are the elements of propositional logic?

Simple sentences which are true or false are basic propositions. Larger and more complex sentences are constructed from basic propositions by combining them with **connectives**. Thus **propositions** and **connectives** are the basic elements of propositional logic. Though there are many connectives, we are going to use the following **five basic connectives** here:

NOT, AND, OR, IF_THEN (or IMPLY), IF_AND_ONLY_IF.

They are also denoted by the symbols:

¬, ∧, ∨, →, ↔, respectively.

(4) What is inference?

Inference is deriving new sentences from old.

(5) What are modus ponens?

There are standard patterns of inference that can be applied to derive chains of conclusions that lead to the desired goal. These patterns of inference are called **inference rules.** The best-known rule is called **Modus Ponens** and is written as follows:

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$$

The notation means that, whenever any sentences of the form $\alpha \Rightarrow \beta$ and α are given, then the sentence β can be inferred. For example, if $(WumpusAhead \wedge WumpusAlive) \Rightarrow Shoot$ and $(WumpusAhead \wedge WumpusAlive)$ are given, then $Shoot$ can be inferred.

Another useful inference rule is **And-Elimination**, which says that, from a conjunction, any of the conjuncts can be inferred:

$$\frac{\alpha \wedge \beta}{\alpha}$$

For example, from $(WumpusAhead \wedge WumpusAlive)$, $WumpusAlive$ can be inferred.

(6) What is entailment?

Propositions tell about the notion of truth and it can be applied to logical reasoning. We can have logical entailment between sentences. This is known as entailment where a sentence follows logically from another sentence. In mathematical notation we write :

$$\alpha \models \beta$$

(7) What are knowledge based agents?

The central component of a knowledge-based agent is its knowledge base, or KB. Informally, a knowledge base is a set of sentences. Each sentence is expressed in a language called a knowledge representation language and represents some assertion about the world.

```
function KB-AGENT(percept) returns an action
  static: KB, a knowledge base
          t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION- QUERY(^))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

Figure 7.1 A generic knowledge-based agent.

Figure 7.1 shows the outline of a knowledge-based agent program. Like all our agents, it takes a percept as input and returns an action. The agent maintains a knowledge base, KB, which may initially contain some **background knowledge.** Each time the agent program is called, it does three things. First, it TELLS the knowledge base what it perceives. Second, it ASKS the knowledge base what action it should perform. In the process of answering this

query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on.

(8) **Explain in detail the connectives used in propositional logic.**

The **syntax** of propositional logic defines the allowable sentences. The **atomic sentences**- the indivisible syntactic elements-consist of a single **proposition symbol.** Each such symbol stands for a proposition that can be true or false. We will use uppercase names for symbols: P, Q, R, and so on.

Complex sentences are constructed from simpler sentences using **logical connectives.** There are five connectives in common use:

¬ (not). A sentence such as $\neg W_{1,3}$ is called the **negation** of $W_{1,3}$. **A literal** is either an atomic sentence (a **positive literal)** or a negated atomic sentence (a **negative literal).**

∧ (and). A sentence whose main connective is ∧, such as $W_{1,3} \wedge P_{3,1}$, is called a **conjunction;** its parts are the **conjuncts.** (The ∧ looks like an "A" for "And.")

∨ (or). A sentence using ∨, such as $(W_{1,3} \wedge P_{3,1}) \vee W_{2,2}$, is a **disjunction** of the **disjuncts** $(W_{1,3} \wedge P_{3,1})$ and $W_{2,2}$. (Historically, the ∨ comes from the Latin "vel," which means "or." For most people, it is easier to remember as an upside-down ∧.)

⇒ (implies). A sentence such as $(W_{1,3} \wedge P_{3,1}) \Rightarrow \neg W_{2,2}$ is called an **implication** (or conditional). Its **premise** or **antecedent** is $(W_{1,3} \wedge P_{3,1})$, and its **conclusion** or **consequent** is $\neg W_{2,2}$. Implications are also known as **rules** or **if–then** statements. The implication symbol is sometimes written in other books as ⊃ or →.

⇔ (if and only if). The sentence $W_{1,3} \Leftrightarrow \neg W_{2,2}$ is a **biconditional.**

Figure 7.7 gives a formal grammar of propositional logic;

```
Sentence        → AtomicSentence | ComplexSentence
AtomicSentence  → True | False | Symbol
Symbol          → P | Q | R | ...
ComplexSentence → ¬ Sentence
                | ( Sentence ∧ Sentence )
                | ( Sentence ∨ Sentence )
                | ( Sentence ⇒ Sentence )
                | ( Sentence ⇔ Sentence )
```

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic.

(9) Define First order Logic?

Whereas propositional logic assumes the world contains facts,
first-order logic (like natural language) assumes the world contains

Objects: people, houses, numbers, colors, baseball games, wars, …
Relations: red, round, prime, brother of, bigger than, part of, comes between, …

Functions: father of, best friend, one more than, plus, …

(10) Specify the syntax of First-order logic in BNF form.

$$
\begin{aligned}
Sentence &\rightarrow AtomicSentence \\
&\mid (\ Sentence\ Connective\ Sentence\ ) \\
&\mid Quantifier\ Variable,\ldots\ Sentence \\
&\mid \neg\ Sentence \\
\\
AtomicSentence &\rightarrow Predicate(Term,\ldots) \mid Term = Term \\
\\
Term &\rightarrow Function(Term,\ldots) \\
&\mid Constant \\
&\mid Variable \\
\\
Connective &\rightarrow \Rightarrow \mid \wedge \mid \vee \mid \Leftrightarrow \\
Quantifier &\rightarrow \forall \mid \exists \\
Constant &\rightarrow A \mid X_1 \mid John \mid \ldots \\
Variable &\rightarrow a \mid x \mid s \mid \ldots \\
Predicate &\rightarrow Before \mid HasColor \mid Raining \mid \ldots \\
Function &\rightarrow Mother \mid LeftLeg \mid \ldots
\end{aligned}
$$

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form. (See page 984 if you are not familiar with this notation.) The syntax is strict about parentheses; the comments about parentheses and operator precedence on page 205 apply equally to first-order logic.

(11) Compare different knowledge representation languages.

Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief $\in [0, 1]$
Fuzzy logic	facts with degree of truth $\in [0, 1]$	known interval value

Figure 8.1 Formal languages and their ontological and epistemological commitments.

(12) **What are the syntactic elements of First Order Logic**?

The basic syntactic elements of first-order logic are the symbols that stand for objects,
relations, and functions. The symbols,come in three kinds:
a) constant symbols, which stand for objects;
b) predicate symbols, which stand for relations;
c) and function symbols, which stand for functions.
We adopt the convention that these symbols will begin with uppercase letters.
Example:
Constant symbols :
Richard and John;

predicate symbols :
Brother, OnHead, Person, King, and Crown;
function symbol :
LeftLeg.

(13) What are quantifiers?

There is need to express properties of entire collections of objects,instead of enumerating the objects by name. Quantifiers let us do this.
FOL contains two standard quantifiers called
a) Universal (∀**)** and
b) Existential (∃**)**

Universal quantification
(∀x) P(x) : means that P holds for **all** values of x in the domain associated with that variable
E.g., (∀x) dolphin(x) => mammal(x)
Existential quantification
(∃ x)P(x) means that P holds for **some** value of x in the domain associated with that variable
E.g., (∃ x) mammal(x) ^ lays-eggs(x)
Permits one to make a statement about some object without naming it

(14) Explain Universal Quantifiers with an example.

Rules such as "All kings are persons," is written in first-order logic as
∀x King(x) => Person(x)
where ∀ is pronounced as " For all .."

Thus, the sentence says, "For all *x*, if *x* is a king, then z is a person."
The symbol x is called a variable(lower case letters)
The sentence ∀**x** P,where P is a logical expression says that P is true for every object x.

(15) Explain Existential quantifiers with an example.

Universal quantification makes statements about every object.

It is possible to make a statement about some object in the universe without naming it,by using an existential quantifier.

Example

"King John has a crown on his head"

∃x Crown(x) ^ OnHead(x,John)

∃x is pronounced "There exists an x such that .." or " For some x .."

(16) What are nested quantifiers?

Nested quantifiers

We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type. For example, "Brothers are siblings" can be written as

$$\forall x \ \forall y \ \ Brother(x, y) \Rightarrow Sibling(x, y).$$

Example-2

"Everybody loves somebody" means that

for every person,there is someone that person loves

∀x ∃ y Loves(x,y)

(17) Explain the connection between ∀ and ∃

"Everyone likes icecream " is equivalent

"there is no one who does not like ice cream"

This can be expressed as :

∀x Likes(x,IceCream) is equivalent to

¬ ∃ ¬Likes(x,IceCream)

(18) What are the steps associated with the knowledge Engineering process?

Discuss them by applying the steps to any real world application of your choice.

Knowledge Engineering

The general process of knowledge base constructiona process is called knowledge engineering.

A knowledge engineer is someone who investigates a particular domain, learns what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain. We will illustrate the knowledge engineering process in an electronic circuit domain that should already be fairly familiar,

The steps associated with the knowledge engineering process are :

1. ***Identfy the task***.

. The task will determine what knowledge must be represented in order to connect problem instances to answers. This step is analogous to the PEAS process for designing agents.
2. ***Assemble the relevant knowledge***. The knowledge engineer might already be an expert in the domain, ***or*** might need to work with real experts to extract what they know-a process called **knowledge acquisition.**
3. ***Decide on a vocabulary of predicates, functions, and constants***. That is, translate the important domain-level concepts into logic-level names.
Once the choices have been made. the result is a vocabulary that is known as the **ontology** of the domain. The word *ontology* means a particular theory of the nature of being or existence.
4. Encode general /cnowledge about the domain. The knowledge engineer writes down the axioms for all the vocabulary terms. This pins down (to the extent possible) the meaning of the terms, enabling the expert to check the content. Often, this step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.
5. Encode a description of the specijic problem insttznce.
For a logical agent, problem instances are supplied by the sensors, whereas a "disembodied" knowledge base is supplied with additional sentences in the same way that traditional programs are supplied with input data.
6. *Pose queries to the inference procedure and get answers.* This is where the reward is: we can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing.
7. Debug the knowledge base.
∀x NumOfLegs(x,4) => Mammal(x)
Is false for reptiles ,amphibians.

To understand this seven-step process better, we now apply it to an extended example-the domain of electronic circuits.

The electronic circuits domain

We will develop an ontology and knowledge base that allow us to reason about digital circuits of the kind shown in Figure 8.4. We follow the seven-step process for knowledge engineering.

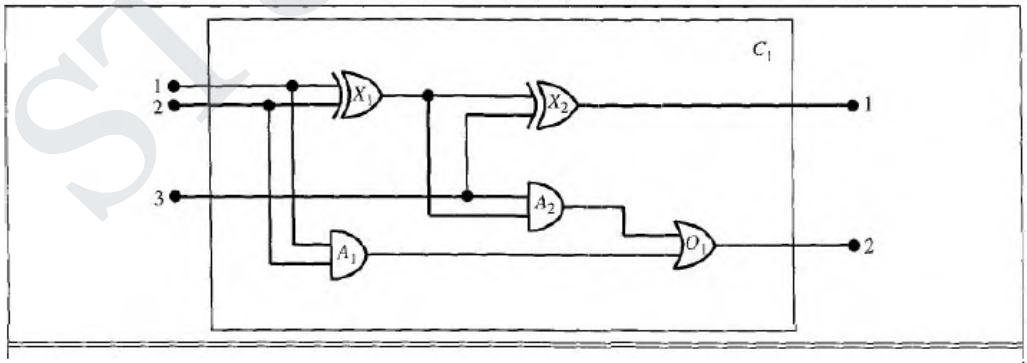


Figure 8.4 A digital circuit C1, purporting to be a one-bit full adder. The first two inputs are the two bits to be added and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two AND gates and one OR gate.

Identify the task

There are many reasoning tasks associated with digital circuits. At the highest level, one analyzes the circuit's functionality. For example, what are all the gates connected to the first input terminal? Does the circuit contain feedback loops? These will be our tasks in this section.

Assemble the relevant knowledge

What do we know about digital circuits? For our purposes, they are composed of wires and gates. Signals flow along wires to the input terminalls of gates, and each gate produces a signal on the output terminal that flows along another wire.

Decide on a vocabulary

We now know that we want to talk about circuits, terminals, signals, and gates. The next step is to choose functions, predicates, and constants to represent them. We will start from individual gates and move up to circuits.

First, we need to be able to distinguish a gate from other gates. This is handled by naming gates with constants: *X I , X2*, and so on

Encode general knowledge of the domain

One sign that we have a good ontology is that there are very few general rules which need to be specified. A sign that we have a good vocabulary is that each rule can be stated clearly and concisely. With our example, we need only seven simple rules to describe everything we need to know about circuits:

1. If two terminals are connected, then they have the same signal:
2. The signal at every terminal is either 1 or 0 (but not both):
3. Connected is a commutative predicate:
4. An OR gate's output is 1 if and only if any of its inputs is 1:
5. An **A.ND** gate's output is 0 if and only if any of its inputs is 0:
6. An XOR gate's output is 1 if and only if its inputs are different:
7. A NOT gate's output is different from its input:

Encode the specific problem instance

The circuit shown in Figure 8.4 is encoded as circuit *C1* with the following description. First, we categorize the gates:

Type(X1)= XOR Type(X2)= XOR

Pose queries to the inference procedure

What combinations of inputs would cause the first output of *Cl* (the sum bit) to be 0 and the second output of *C1* (the carry bit) to be l?

Debug the knowledge base

We can perturb the knowledge base in various ways to see what kinds of erroneous behaviors emerge.

(19) Give examples on usage of First Order Logic.

The best way to find usage of First order logic is through examples. The examples can be taken from some simple **domains.** In knowledge representation, a domain is just some part of the world about which we wish to express some knowledge.

Assertions and queries in first-order logic

Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called **assertions.**

For example, we can assert that John is a king and that kings are persons:

TELL(KB, King (John)) .

Where KB is knowledge base.
TELL(KB, ∀x King(x) => Person(x)).

We can ask questions of the knowledge base using ASK. For example,

$$\text{ASK}(KB, King(John))$$

returns *true.*

Questions asked using ASK are called **queries** or **goals**
ASK(KB,Person(John))
Will return true.
(ASK KBto find whther Jon is a king)
ASK(KB, ∃**x** person(x))

The kinship domain

The first example we consider is the domain of family relationships, or kinship.
This domain includes facts such as
"Elizabeth is the mother of Charles" and
"Charles is the father of William" and rules such as
"One's grandmother is the mother of one's parent."
Clearly, the objects in our domain are people.
We will have two unary predicates, *Male* and *Female.*
Kinship relations-parenthood, brotherhood, marriage, and so on-will be represented by binary predicates: *Parent, Sibling, Brother, Sister, Child, Daughter,Son, Spouse, Husband, Grandparent, Grandchild, Cousin, Aunt,* and *Uncle.*
We will use functions for *Mother* and *Father.*

(20) What is universal instantiation?

Inference rules for quantifiers

Let us begin with universal quantifiers. Suppose our knowledge base contains the standard folkloric axiom stating that all greedy kings are evil:

$$\forall x \;\; King(x) \wedge Greedy(x) \Rightarrow Evil(x) .$$

Then it seems quite permissible to infer any of the following sentences:

$$King(John) \wedge Greedy(John) \Rightarrow Evil(John).$$
$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard) .$$
$$King(Father(John)) \wedge Greedy(Father(John)) \Rightarrow Evil(Father(John)).$$

The rule of **Universal Instantiation** (UI for short) says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable. [1] To write out the inference rule formally, we use the notion of **substitutions** introduced in Section 8.3. Let $\text{SUBST}(\theta, \alpha)$ denote the result of applying the substitution θ to the sentence α. Then the rule is written

$$\frac{\forall v \;\; \alpha}{\text{SUBST}(\{v/g\}, \alpha)}$$

for any variable v and ground term g. For example, the three sentences given earlier are obtained with the substitutions $\{x/John\}$, $\{x/Richard\}$, and $\{x/Father(John)\}$.

The corresponding **Existential Instantiation** rule: for the existential quantifier is slightly more complicated. For any sentence α, variable v, and constant symbol k that does not appear elsewhere in the knowledge base,

$$\frac{\exists v \;\; \alpha}{\text{SUBST}(\{v/k\}, \alpha)} .$$

Universal instantiation (UI)

- Every instantiation of a universally quantified sentence is entailed by it:

$$\frac{\forall v \; \alpha}{\text{Subst}(\{v/g\}, \alpha)}$$

for any variable *v* and ground term *g*

- E.g., $\forall x \; King(x) \wedge Greedy(x) \Rightarrow Evil(x)$ yields:
  $King(John) \wedge Greedy(John) \Rightarrow Evil(John)$
  $King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$
  $King(Father(John)) \wedge Greedy(Father(John)) \Rightarrow Evil(Father(John))$
  .
  .
  .

Existential instantiation (EI)

- For any sentence α, variable *v*, and constant symbol *k* that does not appear elsewhere in the knowledge base:

$$\frac{\exists v \; \alpha}{\text{Subst}(\{v/k\}, \alpha)}$$

- E.g., $\exists x \; Crown(x) \wedge OnHead(x, John)$ yields:

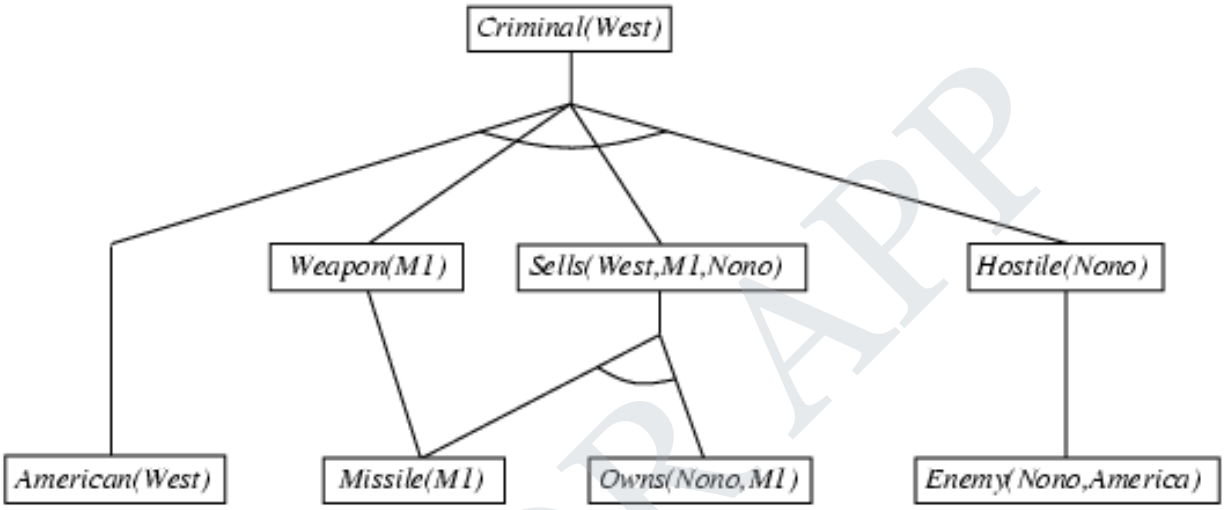
$$Crown(C_1) \wedge OnHead(C_1, John)$$

provided C_1 is a new constant symbol, called a Skolem constant

(21) What is forward chaining? Explain with an example.

Using a deduction to reach a conclusion from a set of antecedents is called forward chaining. In other words,the system starts from a set of facts,and a set of rules,and tries to find the way of using these rules and facts to deduce a conclusion or come up with a suitable couse of action. This is known as data driven reasoning.

EXAMPLE



The proof tree generated by forward chaining.
Example knowledge base

- The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

- Prove that Col. West is a criminal

... it is a crime for an American to sell weapons to hostile nations:
$American(x) \land Weapon(y) \land Sells(x,y,z) \land Hostile(z) \Rightarrow Criminal(x)$
Nono … has some missiles, i.e., $\exists x\ Owns(Nono,x) \land Missile(x)$:
$Owns(Nono,M_1)$ and $Missile(M_1)$
… all of its missiles were sold to it by Colonel West
$Missile(x) \land Owns(Nono,x) \Rightarrow Sells(West,x,Nono)$
Missiles are weapons:
$Missile(x) \Rightarrow Weapon(x)$
An enemy of America counts as "hostile“:
$Enemy(x,America) \Rightarrow Hostile(x)$
West, who is American …
$American(West)$
The country Nono, an enemy of America …
$Enemy(Nono,America)$

Note:

(a) The initial facts appear in the bottom level
(b) Facts inferred on the first iteration is in the middle level
(c) The facts inferered on the 2nd iteration is at the top level

Forward chaining algorithm

```
function FOL-FC-ASK(KB, α) returns a substitution or false
  repeat until new is empty
    new ← { }
    for each sentence r in KB do
      (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-APART(r)
      for each θ such that (p1 ∧ ... ∧ pn)θ = (p'1 ∧ ... ∧ p'n)θ
                  for some p'1, ..., p'n in KB
        q' ← SUBST(θ, q)
        if q' is not a renaming of a sentence already in KB or new then do
          add q' to new
          φ ← UNIFY(q', α)
          if φ is not fail then return φ
    add new to KB
  return false
```

(22) **What is backward chaining ? Explain with an example.**

Forward chaining applies a set of rules and facts to deduce whatever conclusions can be derived.

In **backward chaining** ,we start from a **conclusion**,which is the hypothesis we wish to prove,and we aim to show how that conclusion can be reached from the rules and facts in the data base.

The conclusion we are aiming to prove is called a goal ,and the reasoning in this way is known as **goal-driven**.

Backward chaining example

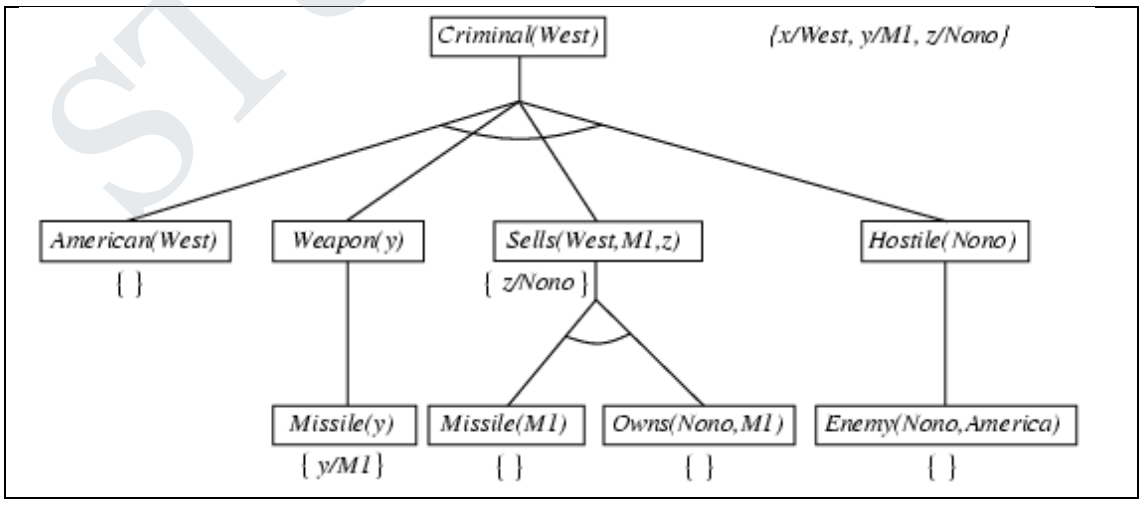


Fig : Proof tree constructed by backward chaining to prove that West is criminal.

Note:
(a) To prove Criminal(West) ,we have to prove four conjuncts below it.
(b) Some of which are in knowledge base,and others require further backward chaining.

(23) Explain conjunctive normal form for first-order logic with an example.

Conjunctive normal form for first-order logic

As in the propositional case, first-order resolution requires that sentences be in **conjunctive normal form** (CNF)—that is, a conjunction of clauses, where each clause is a disjunction of literals.[6] Literals can contain variables, which are assumed to be universally quantified. For example, the sentence

$$\forall x \;\; American(x) \land Weapon(y) \land Sells(x, y, z) \land Hostile(z) \Rightarrow Criminal(x)$$

becomes, in CNF,

$$\neg American(x) \lor \neg Weapon(y) \lor \neg Sells(x, y, z) \lor \neg Hostile(z) \lor Criminal(x).$$

Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence. In particular, the CNF sentence will be unsatisfiable just when the original sentence is unsatisfiable, so we have a basis for doing proofs by contradiction on the CNF sentences. Here we have to eliminate existential quantifiers. We will illustrate the procedure by translating the sentence "Everyone who loves all animals is loved by someone," or

$$\forall x \;\; [\forall y \;\; Animal(y) \Rightarrow Loves(x, y)] \Rightarrow [\exists y \;\; Loves(y, x)]$$

The steps are as follows:

◇ Eliminate implications:

$$ti\, x \;\; [\neg \forall y \;\; \neg Animal(y) \lor Loves(x, y)] \; \mathbf{v} \; [\exists y \;\; Loves(y, x)]$$

◇ Move ¬ inwards: In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have

$\neg \forall x \;\; p$ becomes $3x \;\; \neg p$
$\neg \exists x \;\; p$ becomes $\forall x \;\; \neg p$.

Our sentence goes through the following transformations:

$$\forall x \;\; [\exists y \;\; \neg(\neg Animal(y) \lor Loves(x, y))] \lor [\exists y \;\; Loves(y, x)].$$
$$\forall x \;\; [\exists y \;\; \neg\neg Animal(y) \land \neg Loves(x, y)] \lor [\exists y \;\; Loves(y, x)].$$
$$\forall x \;\; [\exists y \;\; Animal(y) \land \neg Loves(x, y)] \; \mathbf{v} \; [\exists y \;\; Loves(y, x)].$$

Notice how a universal quantifier $(\forall y)$ in the premise of the implication has become an existential quantifier. The sentence now reads "Either there is some animal that x doesn't love, or (if this is not the case) someone loves x." Clearly, the meaning of the original sentence has been preserved.

(24) What is Ontological Engineering?

Ontology refers to organizing every thing in the world into hierarch of categories. Representing the abastract concepts such as Actions,Time,Physical Objects,and Beliefs is called Ontological Engineering.

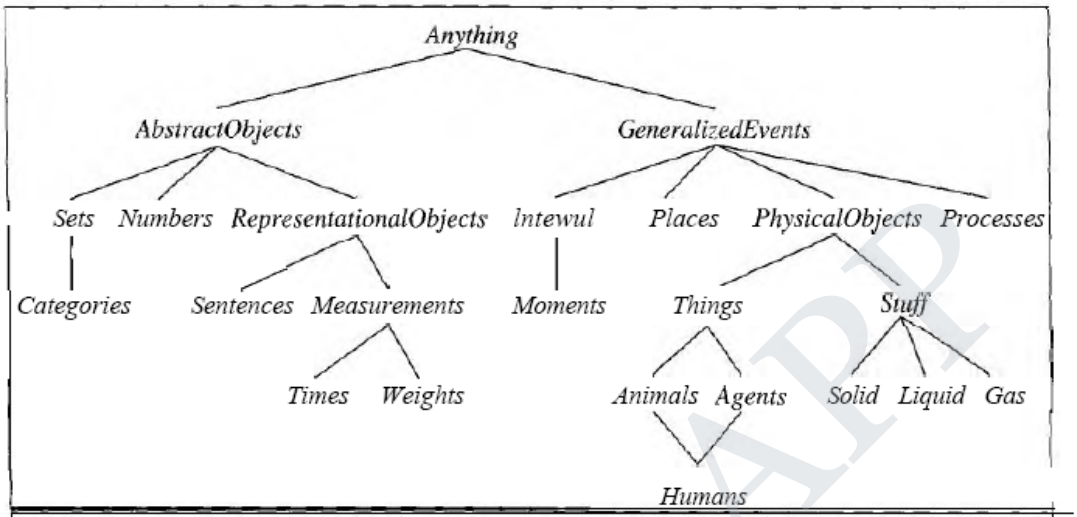


Figure 10.1 The upper ontology of the world, showing the topics to be covered later in the chapter. Each arc indicates that the lower concept is a specialization of the upper one.

(25) How categories are useful in Knowledge representation?

CATEGORIES AND OBJECTS

The organization of objects into **categories** is a vital part of knowledge representation. Although interaction with the world takes place at the level of individual objects, *much reasoning takes place at the level of categories.*

(26) What is taxonomy?

Subclass relations organize categories into a taxonomy, or taxonomic hierarchy. Taxonomies have been used explicitly for centuries in technical fields. For example, systematic biology aims to provide a taxonomy of all living and extinct species; library science has developed a taxonomy of all fields of knowledge, encoded as the Dewey Decimal system; and
tax authorities and other government departments have developed extensive taxoriornies of occupations and commercial products. Taxonomies are also an important aspect of general commonsense knowledge.
First-order logic makes it easy to state facts about categories, either by relating objects to categories or by quantifying over their members:

- An object is a member of a category. For example:
  $BB_9 \in Basketballs$
- A category is a subclass of another category. For example:
  $Basketballs \subset Balls$
- All members of a category have some properties. For example:
  $x \in Basketballs \Rightarrow Round(x)$
- Members of a category can be recognized by some properties. For example:
  $Orange(x) \wedge Round(z) \wedge Diameter(x) = 9.5'' \wedge x \in Balls \Rightarrow x \in Basketballs$
- A category as a whole has some properties. For example:
  $Dogs \in DomesticatedSpecies$

:Notice that because *Dogs* is a category and is a member of *DomesticatedSpecies*, the latter must be a category of categories. One can even have categories of categories of categories, but they are not much use.

(27) What is physical composition?

Physical composition

The idea that one object can be part of another is a familiar one. One's nose is part of one's head, Romania is part of Europe, and this chapter is part of this book. We use the general *PartOf* relation to say that one thing is part of another. Objects can be grouped into *PartOf* hierarchies, reminiscent of the *Subset* hierarchy:

$PartOf(Bucharest, Romania)$
$PartOf(Romania, EasternEurope)$
$PartOf(EasternEurope, Europe)$
$PartOf(Europe, Earth)$.

The *PartOf* relation is transitive and reflexive; that is,

$PartOf(x,y) \wedge PartOf(y,z) \Rightarrow PartOf(x,z)$.
$PartOf(x,x)$.

Therefore, we can conclude $PartOf(Bucharest, Earth)$.

(28) **Explain the Ontology of Situation calculus**.

Situations are logical terms consisting of the initial situation (usually called *So)* and all situations that are generated by applying an action to a situation. The function *Result(a, s)* (sometimes called *Do)* names the situation that results when action *a* is executed in situation *s.* Figure 10.2 illustrates this idea.

Fluents are functions and predicates that vary from one situation to the next, such as the location of the agent or the aliveness of the wumpus. The dictionary says a fluent is something that fllows, like a liquid. In this use, it means flowing or changing across situations. By convention, the situation is always the last argument of a fluent. For example, *lHoldzng(G1, So)* says that the agent is not holding the gold GI in the initial situation *So. Age(Wumpus, So)* refers to the wumpus's age in *So.*

Atemporal or **eternal** predicates and functions are also allowed. Examples include the predicate Gold (GI) and the function *LeftLeg Of* (*Wumpus).*

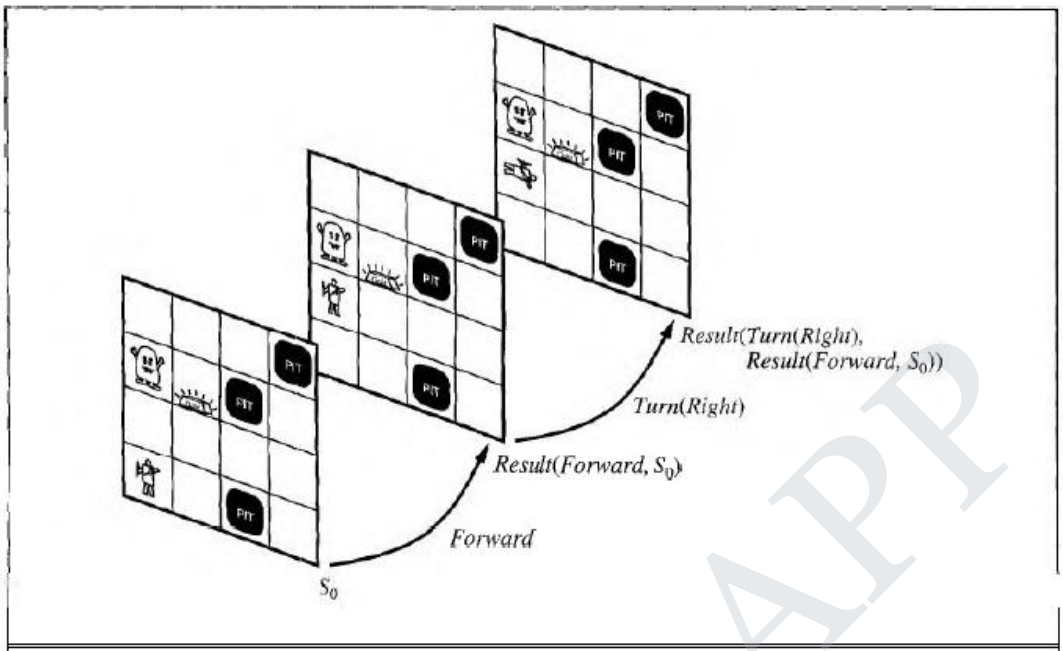


Figure 10.2 In situation calculus, each situation (except S_0) is the result of an action.

(29) What is event calculus?

Time and event calculus

Situation calculus works well when there is a single agent performing instantaneous, discrete actions. When actions have duration and can overlap with each other, situation calculus becomes somewhat awkward. Therefore, we will cover those topics with an alternative for- EVENTCALCULUS malism known as **event calculus,** which is based on points in time rather than on situations.

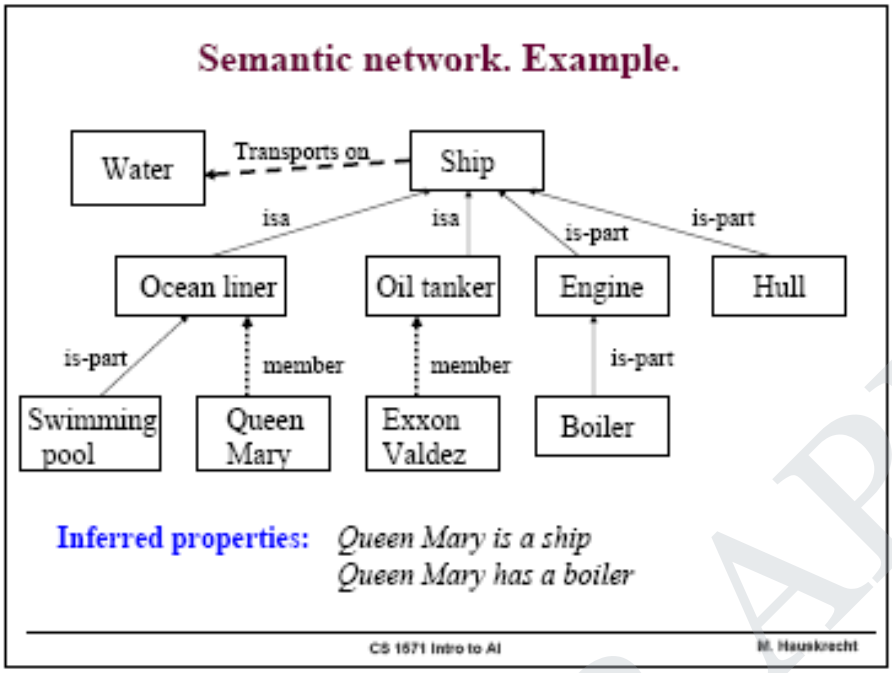
(The terms "event" and "action" may be used interchangeably. Informally, "event" connotes a wider class of actions, including ones with no explicit agent. These are easier to handle in event calculus than in situation calculus.)

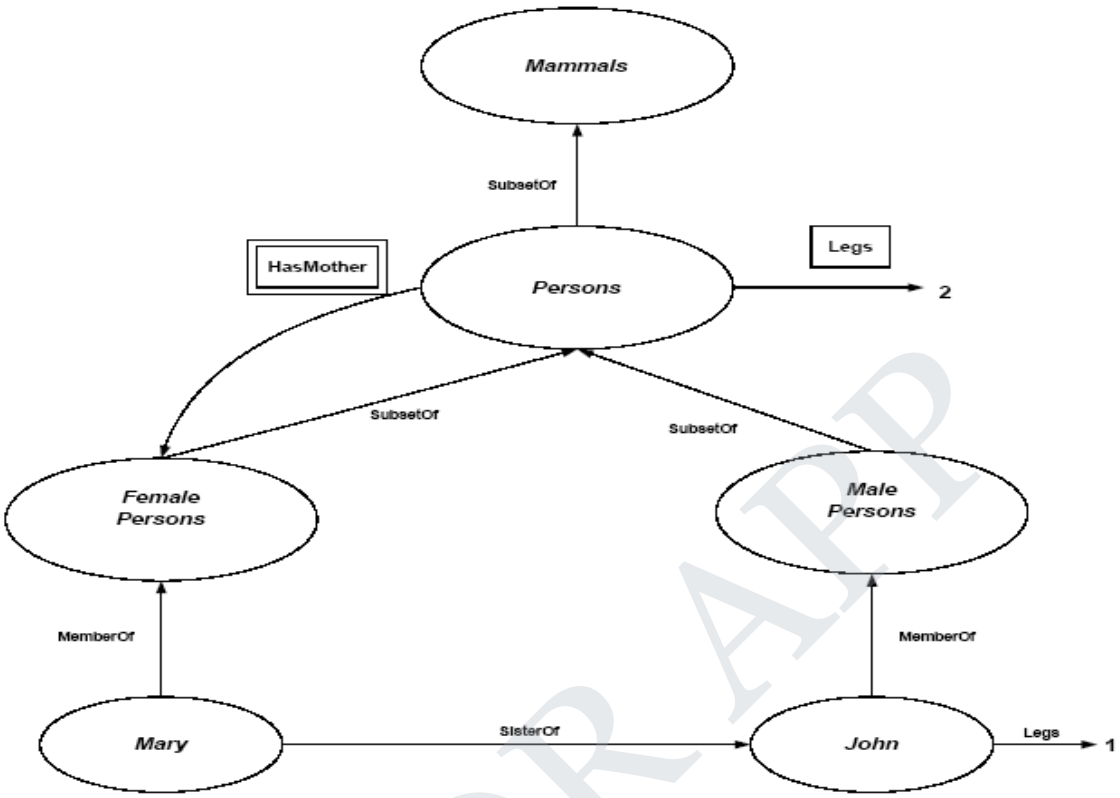
In event calculus, fluents hold at points in time rather than at situations, and the calculus is designed to allow reasoning over intervals of time. The event calculus axiom says that a fluent is true at a point in time if the fluent was initiated by an event at some time in the past and was not terminated by an intervening event. The *Initiates* and *Terminates* relations play a role similar to the *Result* relation in situation calculus; *Initiates(e, f , t)* means that the occurrence of event *e* at time *t* causes fluent *f* to become true, while *Terminates* (w , f, *t)* means that *f* ceases to be true. We use *Happens(e, t)* to mean that event *e* happens at time *t,*

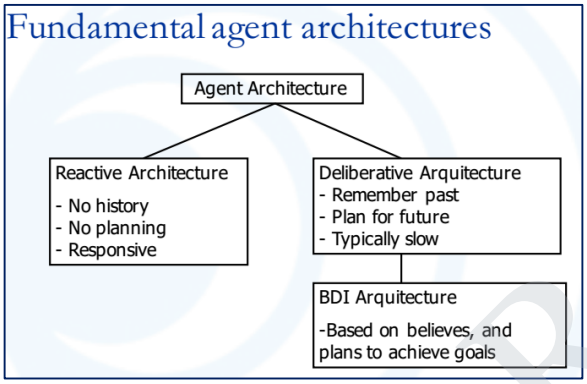
(30) What are semantic networks?

(31) Semantic networks are capable of representing individual objects,categories of objects,and relation among objects. Objects or Ctegory names are represented in ovals and are connected by labeled arcs.

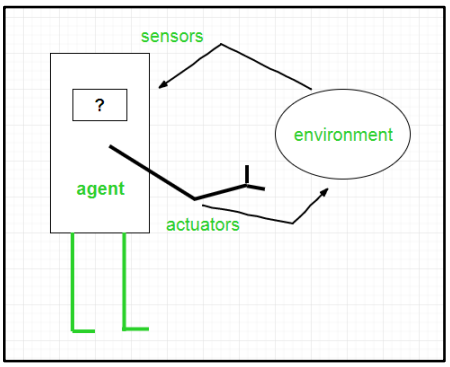
Semantic network example

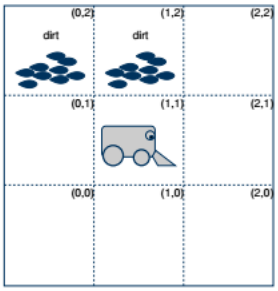




CS8691 Artificial Intelligence – 2017 Regulations	
UNIT IV - SOFTWARE AGENTS	
Architecture for Intelligent Agents – Agent communication – Negotiation and Bargaining – Argumentation among Agents – Trust and Reputation in Multi-agent systems.	
PART - A	
Q.No	**Questions**
1.	Define Purely Reactive Agents.  A *reactive* system is one that maintains an ongoing interaction with its environment, and responds to changes that occur in it (in time for the response to be useful) Purely reactive agents ▪ Some agents decide what to do without reference to their history — they base their decision making entirely on the present, with no reference at all to the past. ▪ We call such agents *purely reactive*: $action : E \rightarrow Ac$ ▪ A thermostat is a purely reactive agent $action(e) = \begin{cases} \text{off} & \text{if } e = \text{temperature OK} \\ \text{on} & \text{otherwise.} \end{cases}$
2.	What are the two types of information source? Data-mining agents This agent uses information technology to find trends and patterns in an abundance of information from many different sources. The user can sort through this information in order to find whatever information they are seeking. A data mining agent operates in a data warehouse discovering information. A 'data warehouse' brings together information from lots of different sources. "Data mining" is the process of looking through the data warehouse to find information that you can use to take action, such as ways to increase sales or keep customers who are considering defecting.

3.	What are characteristics of the subsumption architecture? **Subsumption architecture** is a reactive robotic architecture heavily associated with behavior-based robotics which was very popular in the 1980s and 90s. The term was introduced by Rodney Brooks and colleagues in 1986.[1][2][3] Subsumption has been widely influential in autonomous robotics and elsewhere in real-time AI. • Situatedness – A major idea of situated AI is that a robot should be able to react to its environment within a human-like time-frame. According to Brooks, situated mobile robot should not represent the world via an internal set of symbols and then act on this model. But "the world is its own best model", which means that proper perception-to-action setups can be used to directly interact with the world as opposed to modelling it. • **Embodiment** – Building an embodied agent accomplishes two things. **(1)** The designer to test and create an integrated physical control system, not theoretic models or simulated robots that might not work in the physical world. **(2)** Directly coupling sense-data to meaningful actions. • **Intelligence** – Developing perceptual and mobility skills are a necessary foundation for human-like intelligence. The intelligence is determined by the dynamics of interaction with the world. • **Emergence** – Conventionally, individual modules are not considered intelligent by themselves. It is the interaction of such modules, evaluated by observing the agent and its environment, that is usually deemed intelligent (or not).
4.	State the advantage of vertically layered architecture. **Advantages** Low complexity. If there are n layers there are n-1 interfaces between them. If each layer is capable of suggesting m possible actions then there are at most m2(n-1) interactions • No central control, no bottleneck in the agent's decision making
5.	**Explore some interesting properties of agents and perception.** Artificial intelligence is defined as a study of rational agents. A rational agent could be anything which makes decisions, as a person, firm, machine, or software. It carries out an action with the best outcome after considering past and current percepts(agent's perceptual inputs at a given instance). An AI system is composed of an **agent and its environment**. The agents act in their environment. The environment may contain other agents. An agent is anything that can be viewed as : • perceiving its environment through **sensors** and • acting upon that environment through **actuators**

	
6.	What are four classes of agents? Types of Agents Agents can be grouped into four classes based on their degree of perceived intelligence and capability : • Simple Reflex Agents • Model-Based Reflex Agents • Goal-Based Agents • Utility-Based Agents • Learning Agent
7.	What are logical formulae and logical deduction? **Logical Deduction:** The phenomenon of deriving a conclusion from a single proposition or a set of given propositions, is known as **logical deduction**. The given propositions are also referred to as the **premises**. Logical Deduction is reasoning which constructs or evaluates deductive arguments. Deductive arguments are attempts to show that a conclusion necessarily follows from a set of premises or hypotheses. A deductive argument is valid if the conclusion does follow necessarily from the premises, i.e., the conclusion must be true provided that the premises are true. A deductive argument is sound if it is valid and its premises are true. Deductive arguments are valid or invalid, sound or unsound. Deductive reasoning is a method of gaining knowledge. Deductive reasoning = specific kind of symbolic approach where representations are **logical formulae** and syntactic manipulation used is **logical deduction (theorem proving)** Example: the vacuum world • A small robot to help with housework • Perception: dirt sensor, orientation (north, south, east, west) • Actions: suck up dirt, step forward, turn right by 90 degrees • Starting point (0; 0), robot cannot exit room

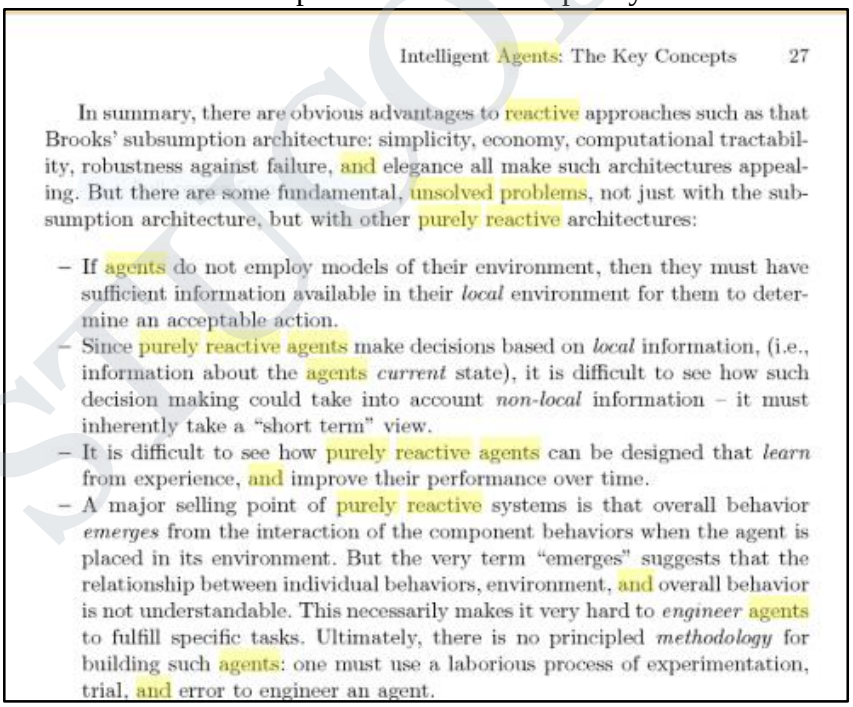


• Goal: traverse the room continually, search for and remove dirt

Example: the vacuum world

• Formulate this problem in logical terms:
• Percept is *dirt* or *null*, actions *forward*, *suck* or *turn*
• Domain predicates *In*(*x; y*), *Dirt*(*x; y*), *Facing*(*d*)
• *next* function must update internal (belief) state of agent correctly
• *old*(Δ) := *fP*(*t*1 : : : *tn*)*jP 2 fIn; Dirt; Facingg* ^ *P*(*t*1 : : : *tn*) *2* Δ*g*
• Assume *new* : *D* × *Per ! D* adds new predicates to database
(what does this function look like?)
• Then, *next*(Δ*; p*) = (Δ*nold*(Δ)) *[new*(Δ*; p*)
• Agent behaviour specified by (hardwired) rules, e.g.
In(*x; y*) ^ *Dirt*(*x; y*) *) Do*(*suck*)
In(0*;* 0) ^ *Facing*(*north*) ^ *:Dirt*(0*;* 0) *) Do*(*forward*)
In(0*;* 1) ^ *Facing*(*north*) ^ *:Dirt*(0*;* 1) *) Do*(*forward*)
In(0*;* 2) ^ *Facing*(*north*) ^ *:Dirt*(0*;* 2) *) Do*(*turn*)
In(0*;* 2) ^ *Facing*(*east*) *) Do*(*forward*)

8. What are the unsolved problems with other purely reactive architectures?



Intelligent Agents: The Key Concepts 27

In summary, there are obvious advantages to reactive approaches such as that Brooks' subsumption architecture: simplicity, economy, computational tractability, robustness against failure, and elegance all make such architectures appealing. But there are some fundamental, unsolved problems, not just with the subsumption architecture, but with other purely reactive architectures:

- If agents do not employ models of their environment, then they must have sufficient information available in their *local* environment for them to determine an acceptable action.
- Since purely reactive agents make decisions based on *local* information, (i.e., information about the agents *current* state), it is difficult to see how such decision making could take into account *non-local* information – it must inherently take a "short term" view.
- It is difficult to see how purely reactive agents can be designed that *learn* from experience, and improve their performance over time.
- A major selling point of purely reactive systems is that overall behavior *emerges* from the interaction of the component behaviors when the agent is placed in its environment. But the very term "emerges" suggests that the relationship between individual behaviors, environment, and overall behavior is not understandable. This necessarily makes it very hard to *engineer* agents to fulfill specific tasks. Ultimately, there is no principled *methodology* for building such agents: one must use a laborious process of experimentation, trial, and error to engineer an agent.

9. Define belief-desire-intention (BDI) architectures

BDI agents

A BDI agent is a particular type of bounded rational software agent, imbued with particular *mental attitudes*, viz: Beliefs, Desires and Intentions (BDI).

	Belief-Desire-Intention (BDI) Architecture
	The BDI architecture is based on practical reasoning by Bratman's philosophical emphasis on intentional stance (Bratman, 1987). Practical reasoning is reasoning toward actions - the process of figuring out what to do. This is different from the theoretical reasoning process as it derives knowledge or reaches conclusions by using one's beliefs and knowledge.
	
	Architecture
	
	The following defines the idealized architectural components of a BDI system.
	
	• **Beliefs**: Beliefs represent the informational state of the agent, in other words its beliefs about the world (including itself and other agents). Beliefs can also include inference rules, allowing forward chaining to lead to new beliefs. Using the term *belief* rather than *knowledge* recognizes that what an agent believes may not necessarily be true (and in fact may change in the future).
	  ○ **Beliefset**: Beliefs are stored in database (sometimes called a *belief base* or a *belief set*), although that is an implementation decision.
	• **Desires**: Desires represent the motivational state of the agent. They represent objectives or situations that the agent *would like* to accomplish or bring about. Examples of desires might be: *find the best price*, *go to the party* or *become rich*.
	  ○ **Goals**: A goal is a desire that has been adopted for active pursuit by the agent. Usage of the term *goals* adds the further restriction that the set of active desires must be consistent. For example, one should not have concurrent goals to go to a party and to stay at home – even though they could both be desirable.
	• **Intentions**: Intentions represent the deliberative state of the agent – what the agent *has chosen* to do. Intentions are desires to which the agent has to some extent committed. In implemented systems, this means the agent has begun executing a plan.
	  ○ **Plans**: Plans are sequences of actions (recipes or knowledge areas) that an agent can perform to achieve one or more of its intentions. Plans may include other plans: my plan to go for a drive may include a plan to find my car keys.
	• **Events**: These are triggers for reactive activity by the agent. An event may update beliefs, trigger plans or modify goals.

10.	What are the two types of control flow within layered architectures? There are two types of vertical layered architectures namely one-pass and two-pass control architectures. In one-pass architecture, control flows from the initial layer that gets data from sensors to the final layer that generates action output (see Figure 7). In two-pass architecture, data flows up the sequence of layers and control then flows back down (see Figure 8).

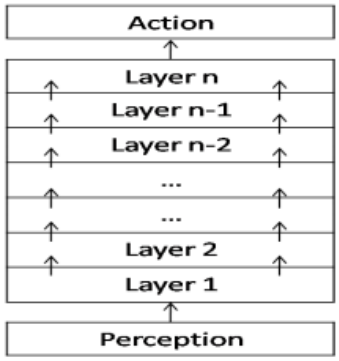


Figure 7: Vertical layer architecture: one-pass

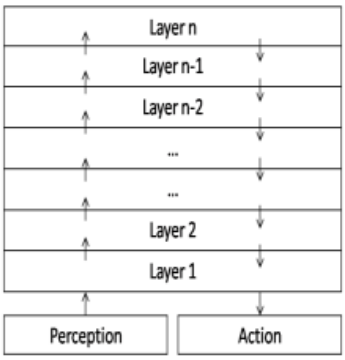


Figure 8: Vertical layer architecture: two-pass

11. State the advantage of horizontal layered architectures.

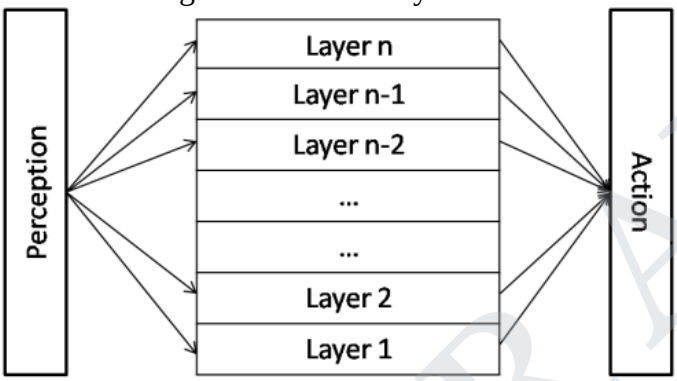


Figure 5: Horizontal Layer Architecture

The advantage of horizontal layer architecture is that only *n* layers are required for mapping to *n* different types of behaviours.

12. Define Agent Communication.

Components of communicating agents

communicating consists of the speaker and the hearer. Because for communication to take place, the agent must be able to perform both these tasks. Both these components can be further explained as follows on the basis of their functioning:

Speaker	Hearer
1. Intention	1. Perception
2. Generation	2. Analysis
3. Synthesis	3. Disambiguation
	4. Incorporation

13. Define Coherence.

Coherence

We review the notion of coherence as used in linguistics. Intuitively, a discourse (text or dialogue) can be called coherent when its parts 'belong together'. Coherence has been studied in natural language semantics and pragmatics under the header of discourse structure. Aspects of coherence that have to do with form are also called *cohesion* [13]. In natural language, cohesion shows by the use of a consistent vocabulary, a consistent style and parallel syntactic constructions. The use of anaphora and ellipsis to refer back to objects mentioned earlier gives the impression of a coherent discourse. Coherence is strongly related with the topic structure. A discourse of which the topics of each of the

utterances are related, for example becausethey are subtopics, makes a more coherent impression than a text with frequent topic shifts. A common approach to analyze coherence is rhetorical structure
theory [19]. The content expressed by difierent utterances is related by rhetorical relations, such as elaboration, explanation or contrast. Rhetorical relations are also called *coherence relations*. They are typically marked by adverbials like 'because' (explanation), or 'however' (contrast). An example is given on the left of flgure 1. If no rhetorical relation can be found to link utterances, the discourse is incoherent. This also relates to the function of utterances. If each utterance contributes to a single purpose, for example to convince the reader or explain
something, this increases coherence.

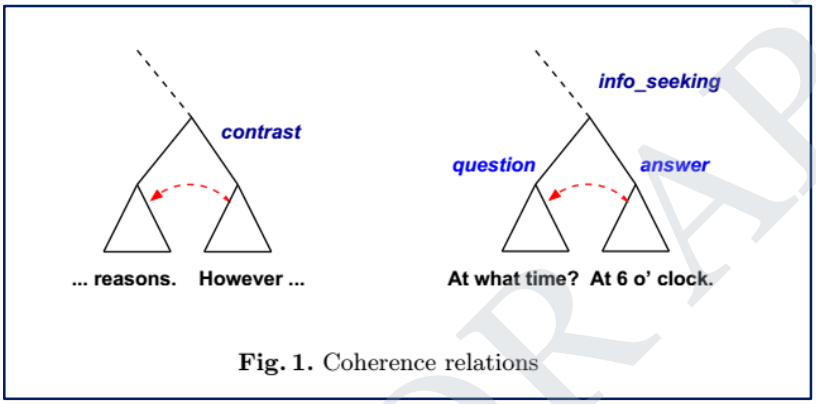


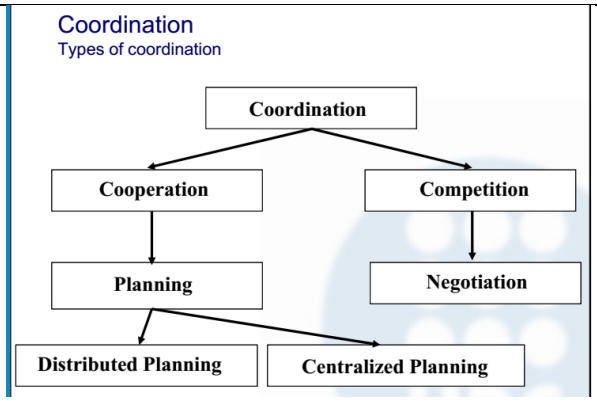
Fig. 1. Coherence relations

A so called discourse context records the contributions of each of the utterances to the over-all meaning of a discourse. By means of a context, the global notion of coherence can now be reduced to a local notion of coherence with
respect to the context.

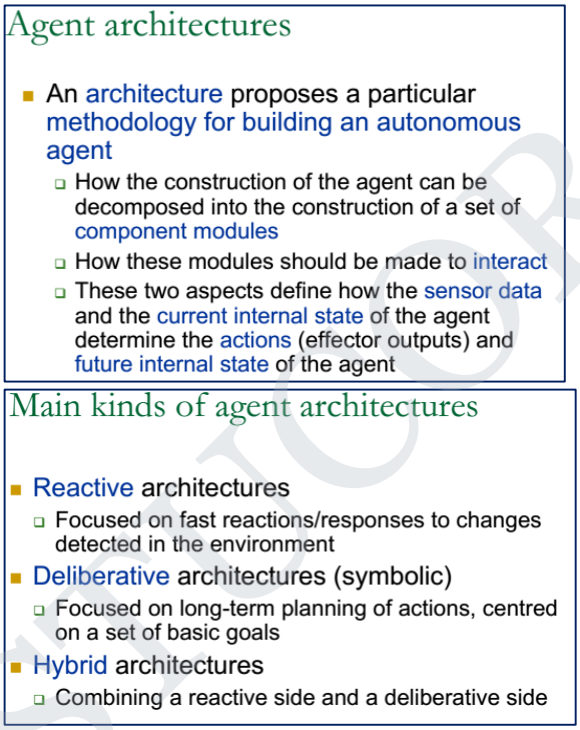
1. An utterance U is coherent with context C iff a coherence relation R can be found that connects U with some part U' of C.
2. In this case, a new context $C' = C + U$ is created which adds the content of U to C in a way that depends on $R(U, U')$.
3. A sequence of utterances $U_1, ..., U_n$ is called coherent, iff each U_k is coherent with discourse context C_k, which represents the contributions of $U_1, ..., U_{k-1}$.

14. Define the property of Coordination

- **Coordination** is a desired property in a Multiagent System whose agents should perform complex tasks in a shared environment.
- The **degree of coordination** in a Multiagent System depends on:
  - -The inability of each individual agent to achieve the whole task(s)
  - -The dependency of one agent on others to achieve the tasks
  - -The need to reduce/optimize resource usage
  - -The need to avoid system halts The need to keep some conditions holding

	
15.	What are the three aspects to the formal study of communication? **Communications Process** Communications is a continuous process which mainly involves three elements viz. sender, message, and receiver. The elements involved in the communication process are explained below in detail: 1. **Sender** The sender or the communicator generates the message and conveys it to the receiver. He is the source and the one who starts the communication 2. **Message** It is the idea, information, view, fact, feeling, etc. that is generated by the sender and is then intended to be communicated further. **Receiver** He is the person who is last in the chain and for whom the message was sent by the sender.
16.	What are the fields Used in protocol? Protocols play a central role in agent communication. A protocol specifies the rules of interaction between two or more communicating agents by restricting the range of allowed follow-up utterances for each agent at any stage during a communicative interaction (dialogue). Such a protocol may be imposed by the designer of a particular system or it may have been agreed upon by the agents taking part in a particular communicative interaction before that interaction takes place. **EXAMPLE : Auction protocol** **Auction:** An *auction* is defined as an interaction between any number of buyers and a single seller that lasts for a predetermined time, mediated by a broker. Technically, the auction is regarded as a single-item, first-price, open-cry, ascending auction (Parsons et al. 2011; Harris and Raviv 1981). An auction is started as soon as the seller accepts the proposal from the broker to host it, and during its lifecycle

	the broker receives bids from any buyer agent. The broker does not interact with the seller during this time, and therefore can accept or reject an offer based on whether or not the offering agent has violated any norms. Once a buyer has its offer accepted by the broker, the following norms are created amongst the buyer, seller, and broker. a(buyer,broker,true,bid) (3) c(buyer,seller,highest_bid,payment) (4) c(seller,buyer,payment,delivery) (5) Norm 3 states that all buyers are authorized to make bids on the auctioned item. Norm 4 states that the buyer with the highest bid is committed to sending the payment to the seller. This commitment ensures that there is no way for a buyer to retract a bid (that has not been outbid) without violating their commitment. Norm 5 is the same commitment from the direct sales protocol (Norm 2) that handles delivery of the item once it is paid for.
17.	Define Ontology. ➢ A conceptualization is a map from the problem domain into the representation. ➢ A conceptualization specifies: o I What sorts of individuals are being modeled o I The vocabulary for specifying individuals, relations and properties o I The meaning or intention of the vocabulary If more than one person is building a knowledge base, they must be able to share the conceptualization. ➢ An ontology is a specification of a conceptualization. ➢ An ontology specifies the meanings of the symbols in an information system.
18.	Define bargaining. A bargaining problem deals with a situation where some players negotiate over sharing a fixed sum of resources. There are two approaches to analyzing a bargaining problem, namely the cooperative approach and the non-cooperative approach. One well-known and widely adopted cooperative bargaining solution is the Nash (1950) bargaining solution. An equally popular and important non-cooperative bargaining solution is the subgame perfect equilibrium in Rubinstein's (1982) bilateral bargaining model. The Model A finite number of players, called players 1, 2,...,n, negotiate how to split a pie of size 1 via (n − 1) bilateral bargaining sessions. In each bilateral bargaining session, two players negotiate a partial and bilateral agreement that specifies the share of the pie for one of the players who then leaves the game. After a partial agreement, the other player continues to negotiate with the rest of the players over the remainder of the pie. The (n − 1) bilateral bargaining sessions determine (n−1) players' shares of the pie and hence all n players' shares of the pie.

19.	Give the Diagrammatic Representation of Trust and Reputation Models for Multiagent Systems. ➢ "Trust begins where knowledge [certainty] ends: trust provides a basis dealing with uncertain, complex, and threatening images of the future." (Luhmann,1979) ➢ "Trust is the outcome of observations leading to the belief that the actions of another may be relied upon, without explicit guarantee, to achieve a goal in a risky situation." (Elofson, 2001) ➢ Reputation is one of the elements that allows us to build trust. ➢ Reputation has also a social dimension. It is not only useful for the individual but also for the society as a mechanism for social order.
20.	Define agent architecture. 
PART - B	
1.	What are Abstract Architectures for Intelligent Agents.(13) **Logic-Based Architecture** Logic-based architecture also known as the symbolic-based or deliberative architecture is one the earliest agent architecture that rests on the physical-symbol systems hypothesis (Newell & Simon, 1976). This classical architecture is based on the traditional artificial symbolic approach by representing and modeling the environment and the agent behavior with symbolic representation. Thus, the agent behavior is based on the

manipulation of the symbolic representation.
Building agent in logic-based approach is viewed as a deduction process. An agent is encoded as a logical theory by using specification and the process of selecting the action is through deduction process that reduces the problem to a solution such as in theorem proving.

Reactive Architecture
Reactive agent architecture is based on the direct mapping of situation to action. It is different from the logic-based architecture where no central symbolic world model and complex symbolic reasoning are used. Agent responses to changes in the environment in a stimulus-response based. The reactive architecture is realized through a set of sensors and effectors, where perceptual input is mapped to the effectors to changes in the environment. Brook's subsumption architecture is known as the best pure reactive architecture (Brooks, 1986). This architecture was developed by Brook who has critiqued on many of the drawbacks in logic-based architecture.

One of the advantages of reactive architecture is that it is less complicated to design and implement than logic-based architecture. An agent's behaviour is computationally tractable. The robustness of reactive architecture against failure is another advantage. Complex behaviours can be achieved from the interaction of simple ones. The disadvantages of reactive architecture include (1) insufficient information about agent's current state to determine an activation action due to modelling of environment available, (2) the processing of the local information limits the planning capabilities in long term or bigger picture and hence, learning is difficult to be achieved, (3) emergent behaviour which is not yet fully understood making it even more intricate to engineer. Therefore, it is difficult to build task-specific agents and one of the solutions is to evolve the agents to perform certain tasks
(Togelius, 2003). The work in this domain is referred to as artificial life.

Belief-Desire-Intention (BDI) Architecture
The BDI architecture is based on practical reasoning by Bratman's philosophical emphasis on intentional stance (Bratman, 1987). Practical reasoning is reasoning toward actions - the process of figuring out what to do. This is different from the theoretical reasoning process as it derives knowledge or reaches conclusions by using one's beliefs and knowledge. Human practical reasoning involves two activities namely deliberation and means-end reasoning. Deliberation decides what state of affairs needs to be achieved while means-end reasoning decides how to achieve these states of affairs. In BDI architecture, agent consists of three logic components referred as mental states/mental attitudes namely beliefs, desires and intentions. Beliefs are the set of information an agent has about the world. Desires are the agent's motivation or possible options to carry out the actions. Intentions

are the agent's commitments towards its desires and beliefs. Intentions are key component in practical reasoning. They describe states of affairs that the agent has

	committed to bringing about and as a result they are action-inducing. Forming the intentions is critical to an agent's success. **Layered (Hybrid) Architecture** Layered (hybrid) architecture is an agent architecture which allows both reactive and deliberate agent behavior. Layered architecture combines both the advantages of reactive and logic-based architecture and at the same time alleviates the problems in both architectures. Subsystems are decomposed into a layer of hierarchical structure to deal with different behaviours. There are two types of interaction that flow between the layer namely horizontal and vertical. In the horizontal layer architecture, each layer is directly connected to the sensory input and action output (see Figure 5). Each layer is like an agent mapping the input to the action to be performed. The advantage of horizontal layer architecture is that only *n* layers are required for mapping to *n* different types of behaviours. However, a mediator function is used to control the inconsistent actions between layer interactions. Another complexity is the large number of possible interactions between horizontal layers–*mn* (where m is the number of actions per layer). Vertical layer architecture eliminates some of these issues as the sensory input and action output are each dealt with by at most one layer each (creating no inconsistent action suggestions)
2.	Write briefly on Concrete Architectures for Intelligent Agents.(13) *Concrete architecture.* Concrete architecture is defined by starting from an abstract architecture: each component is assigned a type chosen among the ones the language provides; each macro-instruction of the engine is implemented. *Agent class.* A class is defined by instantiating the components in the concrete architecture which contain the program of the agent.

3.3 Concrete Architecture Definition

In the definition of the concrete architecture, all the components are assigned a type, the global variables are initialized, and the definitions of partially specified procedures are completed. To illustrate two different implementation choices, we consider two concrete BDI architectures, bdi_1 and bdi_2, obtained from the previously defined abstract BDI architecture *bdi*.

In concrete architecture bdi_1, *plans_component* is assigned a type `stack`, *beliefs_component* has type `set`, *goals_component* has type `set` and *intentions_component* has a type `queue`. External events are either events generated by the environment or messages sent by other agents in the system. An agent which is implemented using this architecture will have both reactivity and social ability.

In the architecture bdi_2, *plans_component*, *beliefs_component* and *goals_component* are `sets`, while *intentions_component* is a `stack`[2]. The only perceived events are those generated by the environment. This architecture gives origin to strongly reactive agents without the ability to receive messages. In both concrete architectures, perceived events are collected in the global variable *event* which has type queue.

The implementation of the BDI concrete architectures is depicted in Figure 4 and 5. In bdi_1, an event is perceived from the environment by means of the **perceive**(e_1) procedure. The global variable *event* is updated by inserting the perceived event into it. A message from *sender* is received in parallel with the perception of the environment, and the received message is also put into the event queue. In bdi_2, only events taking place in the environment are perceived and inserted in the event queue.

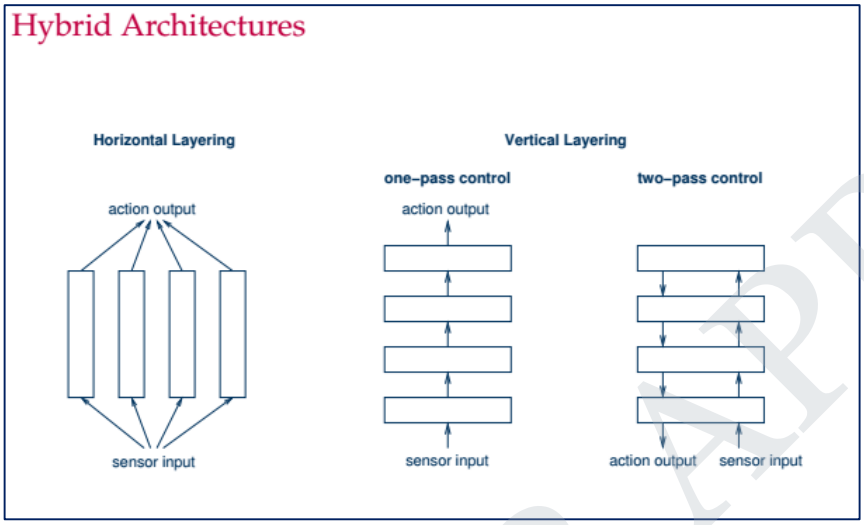
```
architecture {bdi1} is a {bdi} {
  components {
      plans_component: stack;
      belief_component, goals_component: set;
      intention_component: queue
  };
  init_global_vars { ... };
  procedures {
      perceive_event() {
          decl sender, e_1, e_2;
          ( perceive(e_1); event := insqueue(event, e_1)) ||
          ( get_belief_component("sender", !sender); rec(sender, e_2);
          event := insqueue(event, e_2) )
      };
      ...
  }
}
```

Fig. 4. Definition of concrete architecture bdi_1.

3. Write a short note on Layered architectures. (13)

Hybrid Architectures

- *Meta-level control of interactions between these components becomes a key issue in hybrid architectures*
- *Commonly used:* **layered** *approaches*
- *Horizontal layering:*
- *All layers are connected to sensory input/action output*
- *Each layer produces an action, different suggestions have to be reconciled*

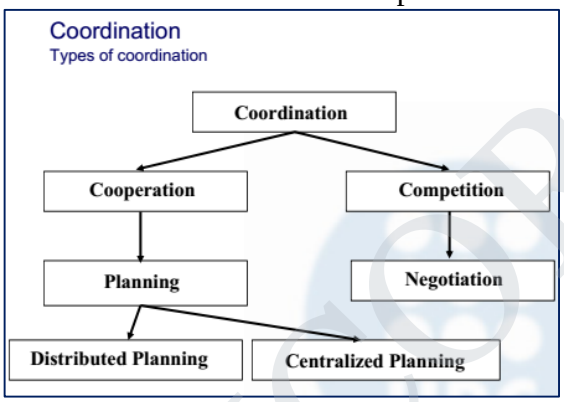
	• *Vertical layering:* • *Only one layer connected to sensors/effectors* • *Filtering approach (one-pass control): propagate intermediate decisions from one layer to another* • *Abstraction layer approach (two-pass control): different layers make decisions at different levels of abstraction*  Touring Machines • Horizontal layering architecture • Three sub-systems: Perception sub-system, control sub-system and action sub-system • Control sub-system consists of • Reactive layer: situation-action rules • Planning layer: construction of plans and action selection • Modelling layer: contains symbolic representations of mental states of other agents • The three layers communicate via explicit **control rules**
4.	Define Agent Communication. Write a short note on coordination, Dimensions of meaning and Message types.(13) A multi-agent system (MAS) may be seen as a collection of collaborative agents „ They can communicate and cooperate with other agents, while keeping their autonomy „ They usually *negotiate* with their peers to reach mutually acceptable agreements during *cooperative problem solving* They normally have limited learning capabilities „ Collaborative agents are usually *deliberative* agents (e.g. BDI model), with some reasoning capabilities ‰ Reactive agents can hardly communicate and collaborate (only through actions that modify the common environment) „ They are usually static, complex agents

Coordination

Wooldridge and Jennings define an *Agent* as a computer program capable of taking its own decisions with no external control (*autonomy*), based on its perceptions of the environment and the objectives it aims to satisfy. An agent may take actions in response to changes in the environment (*reactivity*) and also it may take initiatives (*proactivity*).

A further attribute of agents is their ability to communicate with other agents (*social ability*), not only to share information but, more important, to **coordinate actions** in order to achieve goals for which agents do not have plans they can fulfil on their own, solving even more complex problems.

- **Coordination** is a desired property in a Multiagent System whose agents should perform complex tasks in a shared environment
- The **degree of coordination** in a Multiagent System depends on:
  - The inability of each individual agent to achieve the whole task(s)
  - The dependency of one agent on others to achieve the tasks
  - The need to reduce/optimize resource usage
  - The need to avoid system halts
  - The need to keep some conditions holding



Types of Coordination

Competition and Negotiation

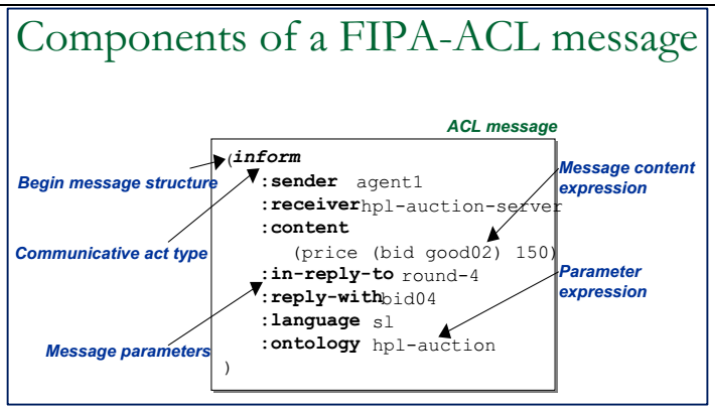
Competition is kind of coordination between antagonist agents which compete with each other or that are selfish

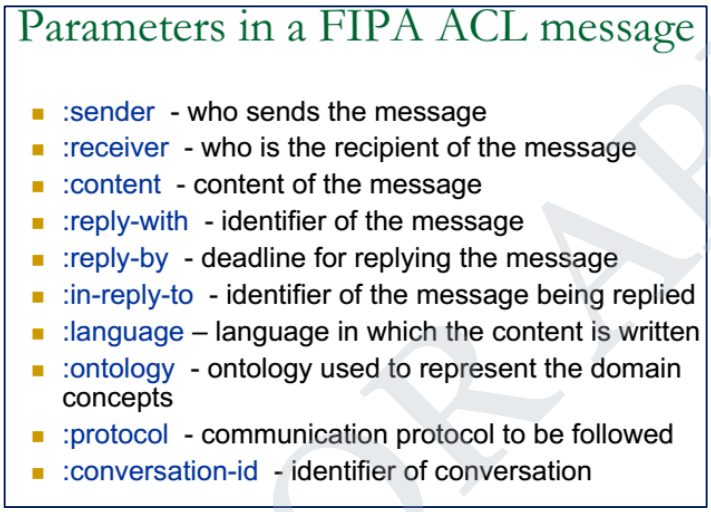
.We will be more interested in **Negotiation**, as it is a kind of competition that involves some higher level of intelligence.

The degree of success in negotiation (for a given agent can be measure by)

The capability of this agent to maximize its own benefit

The capability of not taking into account the other agents' benefit or even trying to minimize other agents' benefit.





5.

Explain Negotiation in detail. (13)

Need of negotiation in MAS

- Agents may have incompatible goals, and resources to achieve these goals may be limited; in such cases competition and conflicts may arise
- The effects of the agents' antagonistic behaviour needs to be limited
- Agents must be able to reach compromises, resolve conflicts, allocate goods and resources by way of an agreement
- Agents' interactions are governed by a set of rules: an interaction protocol

Negotiation protocol elements (I)

Public elements

-Negotiation set which represents the space of possible offers/proposals that the agents can make

-The protocol rules which govern the agents' interactions

Negotiation protocol elements (II)

Private elements

- The set of strategies that the agents can use to participate in the negotiation process:
- They are not dictated by the protocol itself
- May take into account the other agents' strategies

- Learning mechanisms

Protocol rules (I)

Admission rules

When an agent can participate in a negotiation
(e.g. eligibility criteria)

Interaction rules

Sequence of admissible/valid actions (e.g. moments in which bids are allowed)

Validity rules

What constitutes a legal offer/proposal (e.g. a new bid must be higher than the last bid)

Outcome determination rules

When an agreement has been reached

Protocol rules (II)

Withdrawal rules

When an agent can withdraw from the negotiation

Termination rules

When a negotiation ends unsuccesfully

Commitment rules

How the commitments that agents make
during the negotiation are managed

Negotiation factors

Number of attributes: one, many

[Multi-attribute auctions]

Number of agents:

One-to-one

One-to-many

Many-to-many

Number of units: one, many

[Multi-unit auction]

Interrelated goods: one good or a number of goods that are substitutable or interdependent

[Combinatorial auctions]

Protocol evaluation criteria (I)

Social welfare – the sum of all agent's payoffs or utilities in a given solution

Pareto efficiency – a solution x is Pareto optimal if there's no other solution x' such that at least one agent is better off in x' than in x, and no agent is worst off in x' than in x

Individual rationality – an agent should not lose out by participating in a negotiation

Protocol evaluation criteria (II)

Stability – mechanism should be designed to be non-manipulable: motivate each agent to behave in the desired manner

Computational efficiency – mechanisms
should be designed so that when agents use them, as little computation is needed as possible

Distribution and communication efficiency –
distributed protocol vs. minimum communication (time, money,…)

6.	Explain Bargaining theories in detail. (13) **Bargaining** or **haggling** is a type of negotiation in which the buyer and seller of a good or service debate the price and exact nature of a transaction. If the bargaining produces agreement on terms, the transaction takes place. Bargaining is an alternative pricing strategy to fixed prices. Optimally, if it costs the retailer nothing to engage and allow bargaining, they can deduce the buyer's willingness to spend. It allows for capturing more consumer surplus as it allows price discrimination, a process whereby a seller can charge a higher price to one buyer who is more eager (by being richer or more desperate). # Bargaining Theories ## Behavioral theory The personality theory in bargaining emphasizes that the type of personalities determine the bargaining process and its outcome. A popular behavioral theory deals with a distinction between hard-liners and soft-liners. Various research papers refer to hard-liners as warriors, while soft-liners are shopkeepers. It varies from region to region. ## Game theory Bargaining games refer to situations where two or more players must reach an agreement regarding how to distribute an object or monetary amount. Each player prefers to reach an agreement in these games, rather than abstain from doing so. However, each prefers that the agreement favor their interests. Examples of such situations include the bargaining involved in a labor union and the directors of a company negotiating wage increases, the dispute between two communities about the distribution of a common territory, or the conditions under which two countries agree on nuclear disarmament. Analyzing these kinds of problems looks for a solution that specifies which component in dispute corresponds to each party involved. Players in a bargaining problem can bargain for the objective as a whole at a precise moment in time. The problem can also be divided so that parts of the whole objective become subject to bargaining during different stages. ## Bargaining and posted prices in retail markets Retailers can choose to sell at posted prices or allow bargaining: selling at a public posted price commits the retailer not to exploit buyers once they enter the retail store, making the store more attractive to potential customers, while a bargaining strategy has the advantage that it allows the retailer to price discriminate between different types of customer. ## Processual theory This theory isolates distinctive elements of the bargaining chronology in order to better understand the complexity of the negotiating process. Several key features of the processual theory include: • Bargaining range • Critical risk • Security point ## Integrative theory Integrative bargaining (also called "interest-based bargaining," "win-win bargaining") is a negotiation strategy in which parties collaborate to find a "win-win" solution to their dispute. This strategy focuses on developing mutually beneficial agreements based on the interests of the disputants.

	Interests include the needs, desires, concerns, and fears important to each side. They are the underlying reasons why people become involved in a conflict. **Narrative theory** A very different approach to conceptualizing bargaining is as co-construction of a social narrative, where narrative, rather than economic logic drives the outcome.
7.	Narrate Argumentation among Agents in detail.(13) What is Argumentation? What philosophers call it! Arguing with Others "A verbal and social activity of reason aimed at increasing (or decreasing) the acceptability of a controversial standpoint for the listener or reader, by putting forward a constellation of propositions (i.e. arguments) intended to justify (or refute) the standpoint before a rational judge" [van Eemeren et al] "the giving of reasons to support or criticize a claim that is questionable, or open to doubt" [Walton] What is an Argument? Arguments as Chained Inference Rules Arguments as Instances of Schemes Arguments as Graphs
8.	Briefly explain (i). Communication Levels (4) (a) Sender & Receiver (b) Medium (c) Messge (d) Feedback (ii). Speech Acts (3) **Speech acts** are defined in terms of the effects of the cognitive state of the hearer that are intended by the speaker. They are seen as parts of plans that the participants find and execute Most other **Artificial Intelligence** (**AI**) work on **speech acts** is in the area of Distributed **AI**. In linguistics, a **speech act** is an utterance defined in terms of a speaker's intention and the effect it has on a listener. Essentially, it is the action that the speaker hopes to provoke in his or her audience. **Speech acts** might be requests, warnings, promises, apologies, greetings, or any number of **declarations**. (iii). Knowledge Query and Manipulation Language (KQML)(3) The **Knowledge Query and Manipulation Language**, or **KQML**, is a language and protocol for communication among software agents and knowledge-based systems.[1] It was developed in the early 1990s as part of the DARPA knowledge Sharing Effort, which was aimed at developing techniques for building large-scale knowledge bases which are

	shareable and reusable. While originally conceived of as an interface to knowledge based systems, it was soon repurposed as an Agent communication language. Work on KQML was led by Tim Finin of the University of Maryland, Baltimore County and Jay Weber of EITech and involved contributions from many researchers. The KQML message format and protocol can be used to interact with an intelligent system, either by an application program, or by another intelligent system. KQML's "performatives" are operations that agents perform on each other's knowledge and goal stores. Higher-level interactions such as contract nets and negotiation are built using these. KQML's "communication facilitators" coordinate the interactions of other agents to support knowledge sharing. (iv). Knowledge Interchange Format (KIF)(3) **Knowledge Interchange Format** (**KIF**) is a computer language designed to enable systems to share and re-use information from knowledge-based systems. KIF is similar to frame languages such as KL-One and LOOM but unlike such language its primary role is not intended as a framework for the expression or use of knowledge but rather for the interchange of knowledge between systems. The designers of KIF likened it to PostScript. PostScript was not designed primarily as a language to store and manipulate documents but rather as an interchange format for systems and devices to share documents. In the same way KIF is meant to facilitate sharing of knowledge across different systems that use different languages, formalisms, platforms, etc. KIF has a declarative semantics. It is meant to describe facts about the world rather than processes or procedures. Knowledge can be described as objects, functions, relations, and rules. It is a formal language, i.e., it can express arbitrary statements in first order logic and can support reasoners that can prove the consistency of a set of KIF statements. KIF also supports non-monotonic reasoning. KIF was created by Michael Genesereth, Richard Fikes and others participating in the DARPA knowledge Sharing Effort.[1]
9.	With diagrammatic representation, explain Trust and Reputation in Multi-agent systems in detail.(13) Trust is a multi-dimension entity which concerns various attributes such as reliability, dependability, security and honesty, among others. Trust can be basically defined as "a particular level of the subjective probability with which an agent assesses that another agent or group of agents will perform a particular action" What is Trust? It depends on the level we apply it: **User confidence** • Can we trust the user behind the agent? – Is he she a trustworthy source of some kind of knowledge? (e.g. an expert in a field) – Does he/she acts in the agent system (through his agents in a trustworthy way? **Trust of users in agents** • Issues of autonomy: the more autonomy, less trust • How to create trust? – Reliability testing for agents – Formal methods for open MAS– Security and verifiability **Trust of agents in agents** • Reputation mechanisms

	• Contracts • Norms and Social Structures Why Trust? (I) In closed environments, cooperation among agents is included as part of the designing process: - the multi-agent system is usually built by a single developer or a single team of developers and the chosen, option to reduce complexity is to ensure cooperation among the agents they build including it as an important system requirement. - *Benevolence assumption*: an agent *ai* requesting information or a certain service from agent *aj* can be sure that such agent will answer him if *aj* has the capabilities and the resources needed, otherwise *aj* will inform *ai* that it cannot perform the action requested. It can be said that in closed environments **trust is implicit.** Why Trust? (II) However, in an open environment *trust* is not easy to achieve, as - Agents introduced by the system designer can be expected to be nice and trustworthy but this cannot be ensured for alien agents out of the designer control These *alien* agents may give incomplete or false information to other agents or betray them if such actions allow them to fulfill their individual goals. In such scenarios developers use to create *competitive systems* where each agent seeks to maximize its own expected utility at the expense of other agents But, what if solutions can only be constructed by means of *cooperative problem solving*? Agents should try to cooperate, even if there is some uncertainty about the other agent's behaviour. That is, to have some explicit represent How to compute trust • Trust values can be externally defined ▪ by the system designer: the trust values are pre-defined ▪ By the human user: he can introduce his trust values about the humans behind the other agents • Trust values can be inferred from some existing representation about the interrelations between the agents ▪ Communication patterns, cooperation history logs, e-mails, webpage connectivity mapping… • Trust values can be learnt from current and past experiences ▪ Increase trust value for agent a_i if behaves properly with us ▪ Decrease trust value for agent a_i if it fails/defects us • Trust values can be propagated or shared through a MAS ▪ Recommender systems, Reputation mechanisms.
10.	Compare and contrast about the negotiation and bargaining.(13) **<u>An Argument Framework for Negotiation</u>**

Bargaining is negotiation of price alone. But negotiation may apply to much more than price, and may not include price at all. However, all negotiation involves an exchange of value, and agreements and promises of performance. Bargaining is often done verbally. Negotiation often involves written records.

Negotiation is a central process in an agent society where autonomous agents have to cooperate in order to resolve conflicting interests and yet compete to divide limited resources. A direct dialogical exchange of information between agents usually leads to competitive
forms of negotiation where the most powerful agents win. Alternatively, an intelligent mediated interaction may better achieve the goal of reaching a common agreement and supporting cooperative negotiation.
In both cases argumentation is the reference framework to rationally manage conflicting knowledge or objectives, a framework which provides the fundamental abstraction "argument" to exchange pieces of information.

Need for argumentation

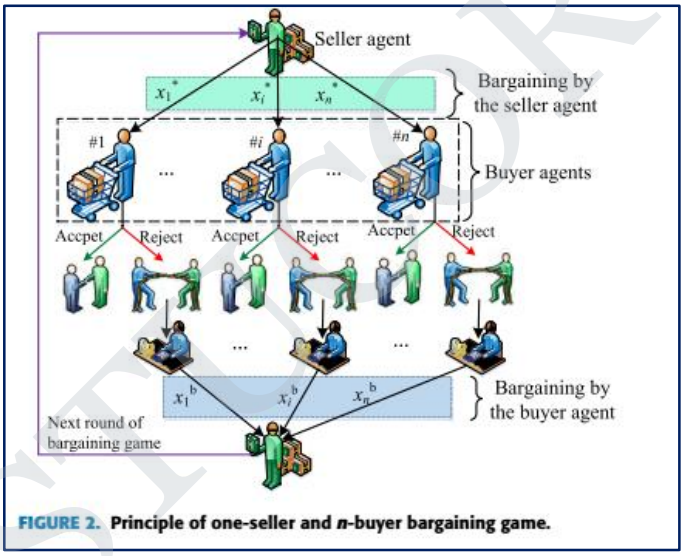
A society mainly evolves through interaction and communication among participating entities. Within a society, people argue and negotiate in order to solve problems, to resolve or reduce conflicts, to exchange information, and to inform each other of pertinent facts. In particular, argumentation is a useful feature of human intelligence that enables us to deal with incomplete and inconsistent information. People usually have only partial knowledge about
the world (they are not omniscient) and often they have to manage conflicting information.

Negotiation

The main form of communication to resolve conflict in human and artificial society is negotiation. Concretely, negotiation is an argumentative process where the participants compete for limited resources or collaborate to find common agreement over their division or allocation.
In the context of multi-agent systems there exist several approaches to realise automated forms of negotiation, through heuristics, game theory and argumentation. Because argumentation involves the requesting, provision and consideration of reasons for claims, it is the most sophisticated of these different forms of interaction for negotiation.
However, providing agents with appropriate conceptual models and related software architectures to fully automate argumentation and negotiation in generic (as distinct from particular
well-defined) domains is still an unsolved research challenge.

Argumentation

Argumentation is a verbal and social activity of reason aimed at increasing (or decreasing) the acceptability of a controversial standpoint for the listener or reader, by

	putting forward a constellation of propositions intended to justify (or refute) the standpoint before a rational judge. In summary, argumentation can be seen as the principled interaction of different, potentially conflicting arguments, for the sake of arriving at a consistent conclusion. Perhaps the most crucial aspect of argumentation is the interaction between arguments. Bargaining is negotiation of price alone. But negotiation may apply to much more than price, and may not include price at all. However, all negotiation involves an exchange of value, and agreements and promises of performance. Bargaining is often done verbally. Negotiation often involves written records. **MULTI-AGENT BARGAINING LEARNING** *A. BARGAINING GAME* In a basic two-player bargaining game, the seller agent will firstly make an offer to the buyer agent, if the offer is accepted by the buyer agent, then an bargaining equilibrium (i.e., the strategy of the offer) can be determined, otherwise, the bargaining role will be shifted to be on the buyer agent in the next period until they reach an agreement on the offer. Enlighted by this game, a novel cooperative one-seller and *n*buyer bargaining game is proposed for achieving an efficient coordination between different players.  **FIGURE 2.** Principle of one-seller and *n*-buyer bargaining game.
11.	Examine the Argumentation among Agents.(13) What is Argumentation? "A verbal and social activity of reason aimed at increasing (or decreasing) the acceptability of a controversial standpoint for the listener or reader, by putting forward a constellation of propositions (i.e. arguments) intended to justify (or refute) the standpoint before a rational judge" [van Eemeren et al] "the giving of reasons to support or criticize a claim that is questionable, or open to doubt" [Walton]

Argumentation versus Reasoning

If you are the judge, argumentation becomes (nonmonotonic) reasoning

Process of Argumentation

Constructing arguments (in favor of / against a "statement") from available information.

A: "Tweety is a bird, so it flies"

B: "Tweety is just a cartoon!"

Determining the different conflicts among the arguments.

"Since Tweety is a cartoon, it cannot fly!" (B attacks A)

Evaluating the acceptability of the different arguments.

"Since we have no reason to believe otherwise, we'll assume Tweety is a cartoon." (accept B). "But then, this means despite being a bird he cannot fly." (reject A).

4 Concluding, or defining the justified conclusions.

"We conlcude that Tweety cannot fly!"

Argumentation System

Example model by Prakken

A tuple $(\mathcal{L}, -, \mathcal{R}_s, \mathcal{R}_d, \leq)$

- a logical language $\mathcal{L}$,
- strict rules $\mathcal{R}_s$
- defeasible rules $\mathcal{R}_d$
- partial order $\leq$ over $\mathcal{R}_d$
- *Contrariness function* $- : \mathcal{L} \to 2^{\mathcal{L}}$ captures conflict between formulas

  Classical negation $\neg$ *captured by* $\neg\varphi \in \overline{\varphi}$ *and* $\varphi \in \overline{\neg\varphi}$.

A particular knowledge base $(\mathcal{K}, \leq')$ with:

- $\mathcal{K} \subseteq \mathcal{L}$ divided into:
  - $\mathcal{K}_n$ are necessary *axioms* (cannot be attacked);
  - $\mathcal{K}_p$ are *ordinary premises*;
  - $\mathcal{K}_a$ are *assumptions*;
  - $\mathcal{K}_i$ are *issues*.
- $\leq'$ is a partial order on $\mathcal{K} \backslash \mathcal{K}_n$.

Inference rules:

- *defeasible* rule $\varphi_1, \ldots, \varphi_n \Rightarrow \varphi$ means conclusion φ follows *presumably* from the premises $\varphi_1, \ldots, \varphi_n$
- *strict* rule $\varphi_1, \ldots, \varphi_n \to \varphi$ stands for classical implication

Functions *Perm*(A), *Conc*(A) and *Sub*(A) returns premises, conclusion, and *sub-arguments* of argument A respectively.

Argument

An argument is any of the following:

- $\varphi \in \mathcal{K}$, where $Prem(A) = \{\varphi\}$, $Conc(A) = \varphi$, and $Sub(A) = \{\varphi\}$.
- $A_1, \ldots, A_n \to \psi$, where $A_1, \ldots, A_n$ are arguments, and there exists in $\mathcal{R}_s$ a strict rule $Conc(A_1), \ldots, Conc(A_n) \to \psi$.
- $A_1, \ldots, A_n \Rightarrow \psi$, where $A_1, \ldots, A_n$ are arguments, and there exists in $\mathcal{R}_d$ a defeasible rule $Conc(A_1), \ldots, Conc(A_n) \to \psi$.

where

- $Prem(A) = Prem(A_1) \cup \cdots \cup Prem(A_n)$
- $Sub(A) = Sub(A_1) \cup \cdots \cup Sub(A_n) \cup \{A\}$

Argumentation Scheme

Argumentation schemes are forms (or categories) of argument, representing stereotypical ways of drawing inferences from particular patterns of premises to conclusions in a particular domain (e.g. reasoning about action).

For each scheme, we list:

- Premises
- Conclusion
- A set of critical questions that can be used to scrutinize the argument by questioning explicit or implicit premises.

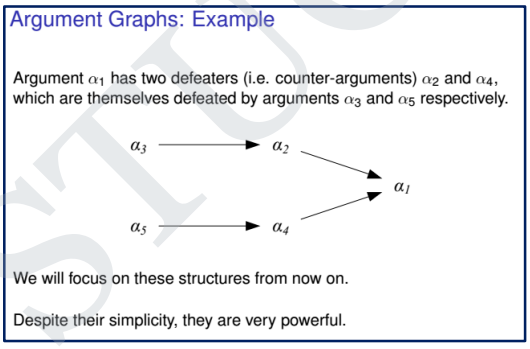
Argumentation Scheme Example

Walton's "sufficient condition scheme for practical reasoning":

In the current circumstances R
We should perform action A
Which will result in new circumstances S
Which will realise goal G
Which will promote some value V.

Associated critical questions include:

CQ1: *Are the believed circumstances true?*
CQ2: *Does the action have the stated consequences?*
CQ3: *Assuming the circumstances and that the action has the stated consequences, will the action bring about the desired goal?*
CQ4: *Does the goal realise the value stated?*
CQ5: *Are there alternative ways of realising the same consequences?*



12.	Describe the trust and reputation in multi-agent systems.(13)

What is Trust?

- It depends on the level we apply it:
  - User confidence
    - Can we trust the user behind the agent?
      - Is he/she a trustworthy source of some kind of knowledge? (e.g. an expert in a field)
      - Does he/she acts in the agent system (through his agents in a trustworthy way?
  - Trust of users in agents
    - Issues of autonomy: the more autonomy, less trust
    - How to create trust?
      - Reliability testing for agents
      - Formal methods for open MAS
      - Security and verifiability
  - Trust of agents in agents
    - Reputation mechanisms
    - Contracts
    - Norms and Social Structures
- Def: Gambetta defines *trust* as ***a particular level of subjective probability with which an agent a_j will perform a particular action both before [we] can monitor such action … and in a context in which it affects [our] own action.***
- Trust is subjective and contingent on the uncertainty of future outcome (as a result of trusting).

How to compute trust?

- Trust value can be assigned to an agent or to a group of agents
- Trust value is an asymmetrical function between agent `a1` and `a2`
  - `trust_val(a1,a2)` does not need to be equal to `trust_val(a2,a1)`
- Trust can be computed as
  - A binary value (1='I do trust this agent', 0='I don't trust this agent')
  - A set of qualitative values or a discrete set of numerical values (e.g. 'trust always', 'trust conditional to X', 'no trust') (e.g. '2', '1', '0', '-1', '-2')
  - A continuous numerical value (e.g. [-300..300])
  - A probability distribution
  - Degrees over underlying beliefs and intentions (cognitive approach)

Trust and Reputation

- Most authors in literature make a mix between trust and reputation
- Some authors make a distinction between them
  - **Trust** is an individual measure of confidence that a given agent has over other agent(s)
  - **Reputation** is a social measure of confidence that a group of agents or a society has over agents or groups
  - (social) Reputation is one mechanism to compute (individual) Trust
    - I will trust more an agent that has good reputation
    - My reputation clearly affects the amount of trust that others have towards me.
    - Reputation can have a ***sanctioning*** role in social groups: a bad reputation can be very costly to one's future transactions.

13. Explain about Planning and acting in the real world .(13)

Planning

The task of coming up with a sequence of actions that will achieve a goal is called **planning.**

- ***Planning*** *is a search problem that requires to find an efficient* ***sequence of actions*** *that transform a system from a given starting state to the* ***goal state***

Classical Planning Environments

we consider only environments that are fully observable, deterministic, finite, static (change happens only when the agent acts), and discrete (in time, action, objects, and effects). These are called **classical planning** environments. In contrast, nonclassical planning is for partially observable or stochastic environments and involves a different set of algorithms and agent designs.

Action Schema

An **action schema** represents a number of different actions that can be derived by instantiating the variables p, from, and to to different constants.
In general, an action schema consists of three parts:

(1) The **action name** and **parameter list**-
for example, Fly(p, from, to) - serves to identify the action.

(2) The **precondition** is a conjunction of function-free positive literals stating wlliat must be true in a state before the action can be executed. Any variables in the precondition must also appear in the action's parameter list.

(3) The **effect** is a conjunction of function-free literals describing how the state changes when the action is executed. A positive literal P in the effect is asserted to be true in the state resulting from the action, whereas a negative literal *P* is asserted to be false. Variables in the effect must also appear in the action's parameter list.

Example: The blocks world
One of the most famous planning domains is known as the **blocks world.** This domain consists of a set of cube-shaped blocks sitting on a table. The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it to another position, either on the table or on top of another block. The arm can pick up only one block at a time, so it cannot pick up a block that has another one on it. The goal will always be to build one or more stacks of blocks, specified in terms of what blocks are on top of what other blocks. For example, a goal might be to get block A on B and block *C* on D.
We will use *On(b, x)* to indicate that block *b* is on *x,* where *x* is either another block or the table. The action for moving block *b* from the top of *x* to the top of y will be *Move(b, x,* y).
Now, one of the preconditions on moving *b* is that no other block be on it. In first-order logic, this would be 13 *x On (x , b)* or, alternatively, 'd *x 1 On(x, b).* These could be stated as

	preconditions in ADL. We can stay within the STRIPS language, however, by introducing a new predicate, *Clear(x),* that is true when nothing is on *x.* The action *Move* moves a block *b* from *x* to *y* if both *b* and *y* are clear. After the move is made, *x* is clear but *y* is not. *A* formal description of *Move* in STRIPS is *Action (Move (b,* z, *y)* , PRECOND: *On(b, x)* A *Clear(b) A Clear(y),* EFFECT: *On(b, y)* A *Clear(x) A lOn(b, x) A lClear(y))* . Unfortunately, this action does not maintain *Clear* properly when *x* or y is the table. When x = *Table,* this action has the effect *Clear(Table),* but the table should not become clear, and when *y = Table,* it has the precondition *Clear(Table),* but the table does not have to be clear to move a block onto it. To fix this, we do two things. First, we introduce another action to move a block *b* from x to the table: *Action(MoveTo Table(b, x)* , PRECOND: *On(b, x)* A *Clear(b),* EFFECT: *On(b, Table)* A *Clear(x) A lOn(b, x))* . Second, we take the interpretation of *Clear(b)* to be "there is a clear space on *b* to hold a block." Under this interpretation, *Clear(Table)* will always be true. The only problem is that nothing prevents the planner from using *Move (b, x, Table)* instead of *Move To Table (b, x)* . We could live with this problem-it will lead to a larger-than-necessary search space, but will not lead to incorrect answers-or we could introduce the predicate *Block* and add *Bloctk(b)* A *Block(y)* to the precondition of *Move.* Finally, there is the problem of spurious actions such as *Move(B, C,* C), which should be a no-op, but which has contradictory effects. It is common to ignore such problems, because they seldom cause incorrect plans to be produced. The correct approach is add inequality preconditions as shown in Figure 11.4. *Init*(*On*(*A*, *Table*) ∧ *On*(*B*, *Table*) ∧ *On*(*C*, *Table*) ∧ *Block*(*A*) ∧ *Block*(*B*) ∧ *Block*(*C*) ∧ *Clear*(*A*) ∧ *Clear*(*B*) ∧ *Clear*(*C*)) *Goal*(*On*(*A*, *B*) ∧ *On*(*B*, *C*)) *Action*(*Move*(*b*, *x*, *y*), PRECOND: *On*(*b*, *x*) ∧ *Clear*(*b*) ∧ *Clear*(*y*) ∧ *Block*(*b*) ∧ (*b* ≠ *x*) ∧ (*b* ≠ *y*) ∧ (*x* ≠ *y*), EFFECT: *On*(*b*, *y*) ∧ *Clear*(*x*) ∧ ¬ *On*(*b*, *x*) ∧ ¬ *Clear*(*y*)) *Action*(*MoveToTable*(*b*, *x*), PRECOND: *On*(*b*, *x*) ∧ *Clear*(*b*) ∧ *Block*(*b*) ∧ (*b* ≠ *x*), EFFECT: *On*(*b*, *Table*) ∧ *Clear*(*x*) ∧ ¬ *On*(*b*, *x*)) **Figure 11.4** A planning problem in the blocks world: building a three-block tower. One solution is the sequence [*Move*(*B*, *Table*, *C*), *Move*(*A*, *Table*, *B*)].
14.	How do you execute the planning in solving problems? (13)
PART-C	

1	Create and design the architecture of intelligence agent with an example. (15) ## Pedagogical Agents Pedagogical agents have come a long way in 20 years. During this time, they have evolved from largely expressionless, robotic characters to empathetic and caring supporters of learning. The trend to make machines more human-like seems to be having a positive influence on learning outcomes, at least in particular ways, and in certain situations. They have progressed from simple demonstrations, to the focus of large-scale studies that address cognitive and noncognitive outcomes. Despite advances in animation and video-game technologies, pedagogical agents still remain far behind in terms of their use of nonverbal behaviors when compared with expert human teachers. The use of gestures during teaching can reinforce concepts and support comprehension, and when used appropriately, can have a direct impact on learning outcomes (Alibali et al., 2013). Thus, it is likely that the full potential of pedagogical agents has not yet been explored, specifically to explore how they might promote learning via nonverbal signals and appropriate use of gestures. **Strengthen Links Between Agent Behaviors and Learner Emotions** The pedagogical agents provide unique opportunities (over non-agent enabled learning environments) to increase engagement and connect with learners emotionally. The argument rests on the emotional potential of simulating meaningful social interactions and has roots in social agency theory. The pedagogical agents should strive to carefully manage the emotional states of learners, helping those on the brink of frustration and disengagement and challenging those who may be approaching boredom. Using conversational and nonverbal strategies, pedagogical agents have a wide range of communicative strategies available to achieve such a balance. Build Real Relationships How successful teachers connect with students, whether it be through effective nonverbal communication or proper interpretation of student emotions, can act as a blueprint for future pedagogical agent research. ## Teaching Knowledge Pedagogical agents originate from research efforts into affective computing (personal systems able to sense, recognize, and respond to human emotions), artificial intelligence (simulating human intelligence, speech recognition, deduction, inference, and creative response), and gesture and narrative language (how artifacts, agents, and toys can be designed with psychosocial competencies). solving context.

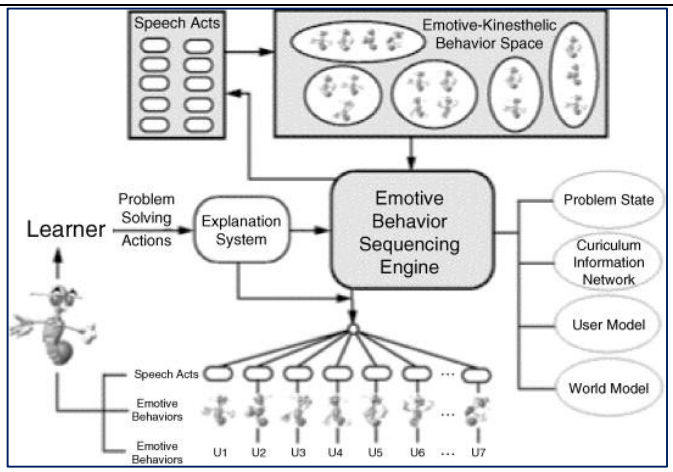


Figure 4.13. An emotive-kinesthetic behavior sequencing architecture used with Cosmos (Lester et al., 1999b).

Pedagogical agents have many liabilities. They are complex to create, text-to-speech with a robotic voice can be annoying to learners, speech recognition technology is not strong enough for widespread use, and text input through natural language understanding (NLU) technology is in its infancy (see Sections 5.5 and 5.6). Animated pedagogical agents are better suited to teach objective information with clear right and wrong answers rather than material based in theory or discussion (Slater, 2000).

4.4.2.1 Emotive Agents

Pedagogical agents often appear to have emotion along with an understanding of the student's problems, providing contextualized advice and feedback similar to a personal tutor (Lester et al., 1997a, 1999a). Human-like attributes can enhance agents' communication skills (i.e., agents rationally respond to the student's emotions or affect). Agents assume a lifelike real-time quality while interacting through a mixed-initiative graphical dialogue. Reacting in real time means the processing time for a tutor to respond to a student appears negligible or the response is immediate as it would be in conversation with another human.

4.4.2.2 Life Quality

Building *life quality* into agents means that the characters' movements, if humanoid, follow a strict adherence to the laws of biology and physics. This implies that the character's musculature and kinesthetics are defined by the physical principles that govern the structure and movement of human and animal bodies (Towns et al., 1988). Facial expressions may be modeled from a human subject. For example, when a character becomes excited, it raises its eyebrows and its eyes widen. In the stylized traditional animation mode, an excited character might bulge out its eyes and leap off the ground.

Promoting metacognition

The use of adaptive scaffolding and pedagogical agents represent cutting-edge metacognitive interventions in open learning environments. Azevedo et al.

	(2004) demonstrated the effectiveness of adaptive scaffolding in monitoring college students' understanding in hypermedia and providing subsequent support.
2.	Explain about the agent communication. (15) Why do we need Agent Communication? „ *Multi agent systems allow distributed problem solving* „ *This requires the agents to coordinate their actions* „ *Agent communication facilitates this by allowing individual agents to interact* ‰ *allows cooperation* ‰ *allows information sharing* Speech Acts „ *A speech act is an act of communication* „ *Speech does not imply any particular communication media* „ *There are various types of speech act* „ *By using the various types of speech act, agents can interact effectively* Communication protocols „ *There are many situations in which agents engaged in a dialogue with a certain purpose exchange the same sequence of messages* ‰ *When an agent makes a question to another* ‰ *When an agent requests a service from another* ‰ *When an agent looks for help from other agents* „ *To ease the management of this typical message interchanges we can use predefined* **protocols** **Communicating agents** Communicating consists of the speaker and the hearer. Because for communication to take place, the agent must be able to perform both

these tasks. Both these components can be further explained as follows on the basis of their functioning:

Speaker	Hearer
1. Intention 2. Generation 3. Synthesis	1. Perception 2. Analysis 3. Disambiguation 4. Incorporation

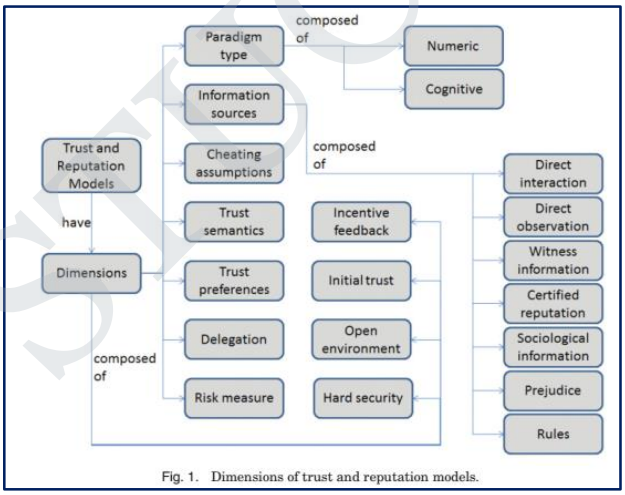
Speaker

1. **Intention:**
   Before speaking anything, we know the intention of what we want to convey to the other person. The same thing is implemented in the communicating systems. This makes communication valid and relevant from the side of the communicating system.
2. **Generation:**
   After knowing the intention of what is to be conveyed, the system must gather words so that the information can be reached to the user in his very own communicating language. So, the generation of relevant words is done by the system after the intention process.
3. **Synthesis:**
   Once the agent has all the relevant words, yet they have to be uttered in a way that they have some meaning. So, after the generation of words, the formation of meaningful sentences takes places and finally, the agent speaks them out to the user.

Hearer

1. **Perception:**
   In the perception phase, the communicating system perceives what the user has spoken to it. This is a sort of an audio input signal which the agent receives from the user and then this signal is sent for the further processing by the system.
2. **Analysis:**
   After getting the audio input from the user which is a sequence of sentences and phrases, the system tries to analyze them by extracting the meaningful terms out of the sentences by removing

	the articles, connectors and other words which are there only for the sake of sentence formation. 3. **Disambiguation:** This is the most important thing that a communicating system carries out. After the analyzing process, the agent must understand the meaning of the sentences that the user have spoken. So, this understanding phase in which the system tries to derive the meaning of the sentences by removing various ambiguities and errors is known as disambiguation. This is done by understanding the Syntax, Semantics, and Pragmatics of the sentences. 4. **Incorporation:** In incorporation, the system figures out whether the understanding that it has derived out of the audio signal is correct or not. Whether it is meaningful, whether the system should consider it or ask the user for further input for resolving any sort of ambiguity.
3.	Develop the trust and reputation in multi-agent systems and make an effective analysis over it. (15) ➢ Trust and reputation concepts are widely used in various fields of computer science, such as evaluation systems, P2P networks, grid computing, game theory, e-commerce, semantic web, software engineering, web services, and recommendation systems. ➢ Another field in which these techniques have been gaining importance is multiagent systems (MASs), which are formed by autonomous agents that interact to achieve their own goals. To achieve their goals, agents must engage in some social activities, such as cooperation, coordination, negotiation, and conflict resolution. ➢ The execution of such activities can bring many problems if agent A establishes a contract with agent B and B does not do it or executes the task dishonestly. ➢ A trusting relationship must exist between these agents when one needs to delegate a task to another. This relationship is addressed by Castelfranchi and Falcone [1998], who state that the confidence an agent has in the other's behavior is a Mental attitude that will influence future decisions. ➢ Another field in which these techniques have been gaining importance is multiagent systems (MASs), which are formed by autonomous agents that interact to achieve their own goals. ➢ To achieve their goals, agents must engage in some social activities, such as cooperation, coordination, negotiation, and conflict resolution [Wooldridge 2009].

	➢ The execution of such activities can bring many problems if agent A establishes a contract with agent B and B does not do it or executes the task dishonestly. A trusting relationship must exist between these agents when one needs to delegate a task to another. This relationship is addressed by Castelfranchi and Falcone [1998], who state that the confidence an agent has in the other's behavior is a mental attitude that will influence future decisions. ➢ In the e-commerce scenario, the human customer can delegate the negotiation authority to his or her personal agent, who will interact and negotiate with other agents or people to reach an agreement. It is necessary to trust that the agent understands the consumer's needs and has the trade competence, ensuring that he or she will not be exploited or cheated by other agents. In electronic auctions, bidders can collude and pay a low price for products, and afterward, they can resell them for a higher price. On the other hand, in a Vickrey auction, the auctioneer can lie to the winner about the price of the second-highest bidder, forcing him or her to pay more than he or she should [Wooldridge 2009]. As we can realize, trust plays an important role in these scenarios, and trust and reputation mechanisms were built to decrease risk in these kinds of interactions. There are several trust definitions in the literature, and one of the most accepted is given by Gambetta [1988], who defines it as a subjective probability that an agent will perform a particular task as expected.  Fig. 1. Dimensions of trust and reputation models.
4.	Analyse about the planning and acting in the real world is happens and explain it. (15) **Planning** The task of coming up with a sequence of actions that will achieve a goal is called **planning.**

- ***Planning*** *is a search problem that requires to find an efficient* ***sequence of actions*** *that transform a system from a given starting state to the* ***goal state***

Classical Planning Environments

we consider only environments that are fully observable, deterministic, finite, static (change happens only when the agent acts), and discrete (in time, action, objects, and effects). These are called **classical planning** environments. In contrast, nonclassical planning is for partially observable or stochastic environments and involves a different set of algorithms and agent designs.

Action Schema

An **action schema** represents a number of different actions that can be derived by instantiating the variables p, from, and to to different constants.
In general, an action schema consists of three parts:

(1) The **action name** and **parameter list**-
for example, Fly(p, from, to) - serves to identify the action.

(2) The **precondition** is a conjunction of function-free positive literals stating wlliat must be true in a state before the action can be executed. Any variables in the precondition must also appear in the action's parameter list.

(3) The **effect** is a conjunction of function-free literals describing how the state changes when the action is executed. A positive literal P in the effect is asserted to be true in the state resulting from the action, whereas a negative literal *P* is asserted to be false. Variables in the effect must also appear in the action's parameter list.

Example: The blocks world
One of the most famous planning domains is known as the **blocks world.** This domain consists of a set of cube-shaped blocks sitting on a table. The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it to another position, either on the table or on top of another block. The arm can pick up only one block at a time, so it cannot pick up a block that has another one on it. The goal will always be to build one or more stacks of blocks, specified in terms of what blocks are on top of what other blocks. For example, a goal might be to get block A on B and block *C* on D.
We will use *On(b, x)* to indicate that block *b* is on *x,* where *x* is either another block or the table. The action for moving block *b* from the top of *x* to the top of y will be *Move(b, x,* y).
Now, one of the preconditions on moving *b* is that no other block be on it. In first-order logic, this would be 13 *x On (x , b)* or, alternatively, 'd *x 1 On(x, b).* These could be stated as

preconditions in ADL. We can stay within the STRIPS language, however, by introducing a
new predicate, *Clear(x),* that is true when nothing is on *x.*
The action *Move* moves a block *b* from *x* to *y* if both *b* and *y* are clear. After the move is made, *x* is clear but *y* is not. *A* formal description of *Move* in STRIPS is
Action (Move (b, z, *y)* ,
PRECOND: *On(b, x)* A *Clear(b) A Clear(y),*
EFFECT: *On(b, y)* A *Clear(x) A lOn(b, x) A lClear(y))* .
Unfortunately, this action does not maintain *Clear* properly when *x* or y is the table. When
x = *Table,* this action has the effect *Clear(Table),* but the table should not become clear, and
when *y = Table,* it has the precondition *Clear(Table),* but the table does not have to be clear
to move a block onto it. To fix this, we do two things. First, we introduce another action to
move a block *b* from x to the table:
Action(MoveTo Table(b, x) ,
PRECOND: *On(b, x)* A *Clear(b),*
EFFECT: *On(b, Table)* A *Clear(x) A lOn(b, x))* .
Second, we take the interpretation of *Clear(b)* to be "there is a clear space on *b* to hold a block." Under this interpretation, *Clear(Table)* will always be true. The only problem is that
nothing prevents the planner from using *Move (b, x, Table)* instead of *Move To Table (b, x)* .
We could live with this problem-it will lead to a larger-than-necessary search space, but will not lead to incorrect answers-or we could introduce the predicate *Block* and add *Bloctk(b)* A*Block(y)* to the precondition of *Move.*
Finally, there is the problem of spurious actions such as *Move(B, C,* C), which should be a no-op, but which has contradictory effects. It is common to ignore such problems, because they seldom cause incorrect plans to be produced. The correct approach is add inequality
preconditions as shown in Figure 11.4.

$$
\begin{array}{l}
Init(On(A, Table) \wedge On(B, Table) \wedge On(C, Table) \\
\quad \wedge Block(A) \wedge Block(B) \wedge Block(C) \\
\quad \wedge Clear(A) \wedge Clear(B) \wedge Clear(C)) \\
Goal(On(A, B) \wedge On(B, C)) \\
Action(Move(b, x, y), \\
\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge \\
\qquad (b \neq x) \wedge (b \neq y) \wedge (x \neq y), \\
\quad \text{EFFECT: } On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)) \\
Action(MoveToTable(b, x), \\
\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x), \\
\quad \text{EFFECT: } On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))
\end{array}
$$

Figure 11.4 A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[Move(B, Table, C), Move(A, Table, B)]$.

Anna University
B.E.(Computer Science and Engineering) – 2017 Regulations
VI SEM CSE CS8691 Artificial Intelligence

UNIT V APPLICATIONS 9

AI applications – Language Models – Information Retrieval- Information Extraction – Natural Language Processing - Machine Translation – Speech Recognition – Robot – Hardware – Perception – Planning – Moving

UNIT-V Question and Answers

(1) Define communication.

Communication is the intentional exchange of information brought about by the production and perception of **signs** drawn from a shared system of conventional signs. Most animals use signs to represent important messages: food here, predator nearby etc. In a partially observable world, communication can help agents be successful because they can learn information that is observed or inferred by others.

(2) What is speech act?

What sets humans apart from other animals is the complex system of structured messages known as **language** that enables us to communicate most of what we know about the world. This is known as speech act.

Speaker, hearer, and **utterance** are generic terms referring to any mode of communication. The term **word** is used to refer to any kind of conventional communicative sign.

(3) **What are the capabilities gained by an agent from speech act?**.

- **Query** other agents about particular aspects of the world. This is typically done by asking questions: *Have you smelled the wumpus anywhere?*
- **Inform** each other about the world. This is done by making representative statements: *There's a breeze here in 3 4.* Answering a question is another Pund of informing.
- **Request** other agents to perform actions: *Please help me carry the gold.* Sometimes **indirect speech act** (a request in the form of a statement or question) is considered more polite: *I could use some help carrying this.* An agent with authority can give commands *(Alpha go right; Bravo and Charlie go lejl),* and an agent with power can make a threat *(Give me the gold, or else).* Together, these kinds of speech acts are called **directives.**
- **Acknowledge** requests: **OK.**
- **Promise** or commit to a plan: *I'll shoot the wumpus; you grab the gold.*

(4) Define formal language.

A formal language is defined as a (possibly infinite) set of' **strings.** Each string is a concatenation of **terminal symbols,** sometimes called words. For example, in the language of first-order logic, the terminal symbols include A and P, and a typical string is "P A Q." . Formal languages such as first-order logic and Java have strict mathematical definitions. This is in contrast to **natural languages,** such as Chinese, Danish, and English, that have no strict definition but are used by a community.

(5) Define a grammar.

A **grammar** is a finite set of rules that specifies a language. Formal languages always have an official grammar, specified in manuals or books. Natural languages have no official grammar, but linguists strive to discover properties of the language by a process of scientific

inquiry and then to codify their discoveries in a grammar.

(6) What are the component steps of communication? Explain with an example.

The component steps of communication

A typical communication episode, in which speaker S wants to inform hearer H about proposition P using words W, is composed of seven processes:

1) Intention. Somehow, speaker S decides that there is some proposition P that is worth saying to hearer H. For our example, the speaker has the intention of having the hearer know that the wumpus is no longer alive.

2) Generation. The speaker plans how to turn the proposition P into an utterance that makes it likely that the hearer, upon perceiving the utterance in the current situation, can infer the meaning P (or something close to it). Assume that the speaker is able to come up with the words "The wumpus is dead," and call this *W*.

3) Synthesis. The speaker produces the physical realization *W'* of the words *W*. This can be via ink on paper, vibrations in air, or some other medium. In Figure 22.1, we show the agent synthesizing a string of sounds *W'* written in the phonetic alphabet defined on page 569: "[thaxwahmpaxsihzdehd]." The words are run together; this is typical of quickly spoken speech.

4) Perception. H perceives the physical realization W' as *Wi* and decodes it as the words *W2*. When the medium is speech, the perception step is called **speech recognition;** when it is printing, it is called **optical character recognition.**

5) Analysis. H infers that *W2* has possible meanings *PI, . . . , P,*.

We divide analysis into three main parts:

a) **syntactic interpretation (or parsing**),

b) **Semantic interpretation,** and

c)**Pragmatic interpretation**.

Parsing is the process of building a **parse tree** for an input string, as shown in Figure 22.1. The interior nodes of the parse tree represent phrases and the leaf nodes represent words.

Semantic interpretation is the process of extracting the meaning of an utterance as an expression in some representation language. Figure 22.1 shows two possible semantic interpretations: that the wumpus is not alive and that it is tired (a colloquial meaning of dead). Utterances with several possible interpretations are said to be **ambiguous.**

Pragmatic interpretation takes into account the fact that the same words can have different meanings in different situations.

6) Disambiguation. H infers that *S* intended to convey P, (where ideally *P,* = P).

Most speakers are not intentionally ambiguous, but most utterances have several feasible interpretations.. Analysis generates possible interpretations;if more than one interpretation is found, then disambiguation chooses the one that is best.

7) **Incorporation. H** decides to believe *P,* (or not). A totally naive agent might believe everything it hears, but a sophisticated agent treats the speech act as evidence for P,, not confirmation of it.

Putting it all together, we get the agent program shown in Figure 22.2. Here the agent acts as a robot slave that can be commanded by a master. On each turn, the slave will answer a question or obey a command if the master has made one, and it will believe any statements made by the master. It will also comment (once) on the current situation if it has nothing more pressing to do, and it will plan its own action if left aloine.

Here is a typical dialog:

ROBOT SLAVE MASTER
I feel a breeze. Go to 12.
Nothing is here. Go north.
I feel a breeze and I smell a stench
and I see a glitter. Grab the gold.
Fig 22.1 shows the seven processes involved in communication,using the example sentence “The wumpus is dead”.

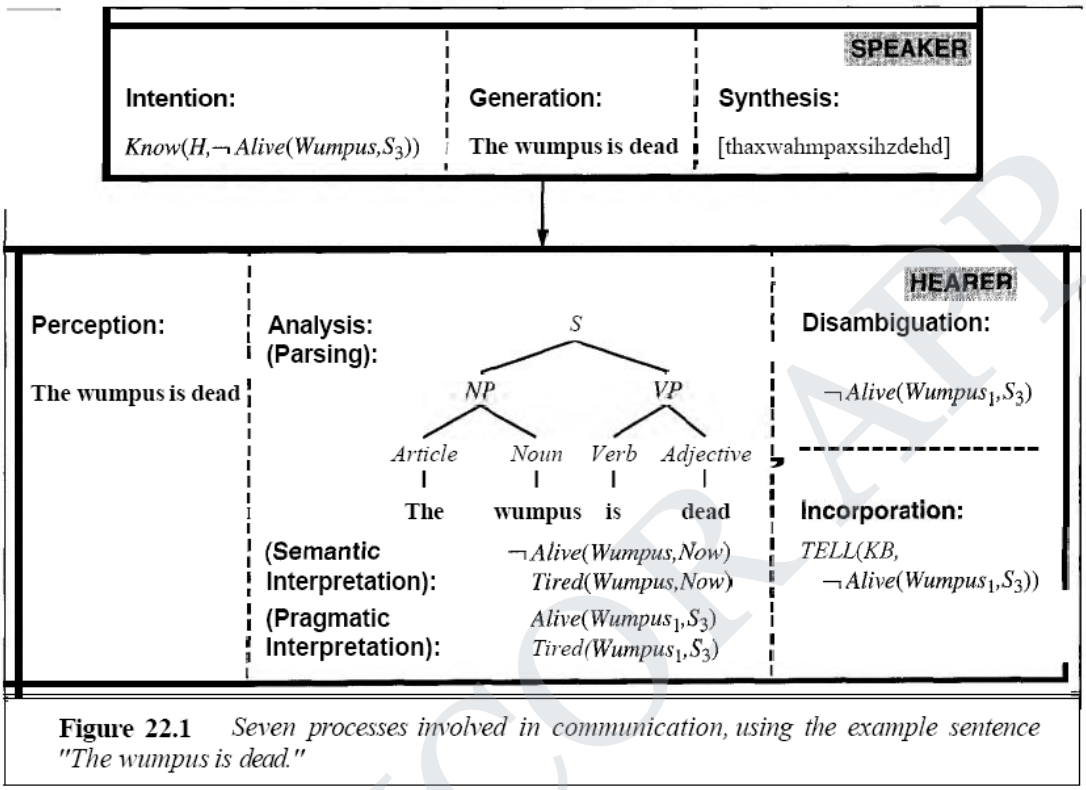


Figure 22.1 *Seven processes involved in communication, using the example sentence "The wumpus is dead."*

(7) Define a Lexicon and grammar for language consisting of a small fragment of English.

The Lexicon of $\mathcal{E}_0$

First we define the **lexicon**, or list of allowable words. The words are grouped into the categories or parts of speech familiar to dictionary users: nouns, pronouns, and names to denote

things, verbs to denote events, adjectives to modify nouns, and adverbs to modify verbs. Categories that are perhaps less familiar to some readers are articles (such as the), prepositions (in), and conjunctions (and). Figure 22.3 shows a small lexicon.

Noun	→	stench | breeze | glitter | **nothing** | agent | wumpus | pit | **pits**| **gold** | east | ...
Verb	→	is | see | smell | shoot | feel | stinks | go | grab | carry | kill | turn | ...
Adjective	→	right | left | east | dead | back | smelly | ..
Adverb	→	here | there | nearby | ahead | right | left | east | south | **back** | ...
Pronoun	→	me | you | I | it | ...
Name	→	John | Mary | Boston | **Aristotle** | ...
Article	→	the | a | an | ...
Preposition	→	to | in | on | near | ...
Conjunction	→	and | or | but | ...
Digit	→	**0**| **1**| **2**| **3**| **4**| **5**| **6**| **7**| **8**| **9**

Figure 22.3 The lexicon for $\mathcal{E}_0$.

The Grammar of $\mathcal{E}_0$

The next step is to combine the words into phrases. We will use five nonterminal symbols to define the different kinds of phrases: sentence *(S),* noun phrase (NP), verb phrase (VP), prepositional phrase (PP), and relative clause e el clause).' Figure 22.4 shows a grammar for l o , with an example for each rewrite rule. lo generates good English sentences such as the following:

John is in the pit
The wumpus that stinks is in 2 2

S	→	*NP VP*	*I* + feel a breeze
	|	*S Conjunction S*	I feel a breeze + and + I smell a wumpus
NP	→	*Pronoun*	*I*
	|	*Name*	John
	|	*Noun*	pits
	|	*Article Noun*	the + wumpus
	|	*Digit Digit*	3'4
	|	*NP PP*	the wumpus + to the east
	|	*NP RelClause*	the wumpus + that is smelly
VP	→	*Verb*	stinks
	|	*VP NP*	feel + a breeze:
	|	*VP Adjective*	is + smelly
	|	*VP PP*	turn + to the east
	|	*VP Adverb*	go + ahead
PP	→	*Preposition NP*	to + the east
RelClause	→	that *VP*	that + is smelly

Figure 22.4 The grammar for $\mathcal{E}_0$, with example phrases for each rule.

(8) What is parsing? Explain the top down parsing method.

Parsing is defined as the process of finding a **parse tree** for a given input string.

That is, a call to the parsing function PARSE, such as

PARSE("the wumpus is dead", $\mathcal{E}_0$, S)

should return a parse tree with root S whose leaves are "the wumpus is dead" and whose internal nodes are nonterminal symbols from the grammar Eo.

Parsing can be seen as a process of searching for a parse tree.

There are two extreme ways of specifying the search space (and many variants in between).

First, we can start with the *S* symbol and search for a tree that has the words as its leaves. This is called **top-down parsing**

Second, we could start with the words and search for a tree with root S. This is called **bottom-up parsing.**

Top-down parsing can be precisely defined as a search problem as follows:

- The initial state is a parse tree consisting of the root S and unknown children: [S: ?I.

  In general, each state in the search space is a parse tree.

The successor function selects the leftmost node in the tree with unknown children. It then looks in the grammar for rules that have the root label of the node on the left-hand side. For each such rule, it creates a successor state where the ? is replaced by a list corresponding to the right-hand side of the rule..

(9) Formulate the bottom-up parsing as a search problem.

The formulation of bottom-up parsing as a search is as follows:

The **initial state** is a list of the words in the input string, each viewed as a parse tree that is just a single leaf node-for example; **[the, wumlpus, is, dead].** In general, each state in the search space is a list of parse trees.

The **successor function** looks at every position ***i*** in the list of trees and at every righthand side of a rule in the grammar. If the subsequence of the list of trees starting at *i* matches the right-hand side, then the subsequence is replaced by a new tree whose category is the left-hand side of the rule and whose children are the subsequence. By "matches," we mean that the category of the node is the same as the element in the righthand side. For example, the rule *Article* + **the** matches the subsequence consisting of the first node in the list **[the, wumpus, is, dead],** so a successor state would be *[[Article:***the], wumpus, is, dead].**

The **goal test** checks for a state consisting of a single tree with root S.

See Figure 22.5 for an example of bottom-up parsing.

step	*list of nodes*	*subsequence*	*rule*
INIT	**the wumpus is dead**	**the**	Article → **the**
2	*Article* **wumpus is dead**	**wumpus**	Noun → **wumpus**
3	*Article Noun* **is dead**	*Article Noun*	*NP* → *Article Noun*
4	*NP* **is dead**	**is**	*Verb* → **is**
5	*NP Verb* **dead**	**dead**	*Adjective* → **dead**
6	**NP** *Verb Adjective*	*Verb*	*VP* → *Verb*
7	*NP VP Adjective*	*VP Adjective*	*VP* → *VP Adjective*
8	*NP VP*	*NP VP*	*S* → *NP VP*
GOAL	S		

Figure 22.5 Trace of a bottom up parse on the string "The wumpus is dead." We start with a list of nodes consisting of words. Then we replace subsequences that match the right-hand side of a rule with a new node whose root is the left-hand side. For example, in the third line the Article and Noun nodes are replaced by an NP node that has those two nodes as children. The top-down parse would produce a similar trace, but in the opposite direction.

(10) What is dynamic programming?

Forward Chaining on graph search problem is an example of dynamic programming. Solutions to the sub problems are constructed incrementally from those of smaller sub problems and are cached to avoid recomputation.

(11) Construct a parse tree for "You give me the gold" showing the sub categories of the verb and verb phrase.

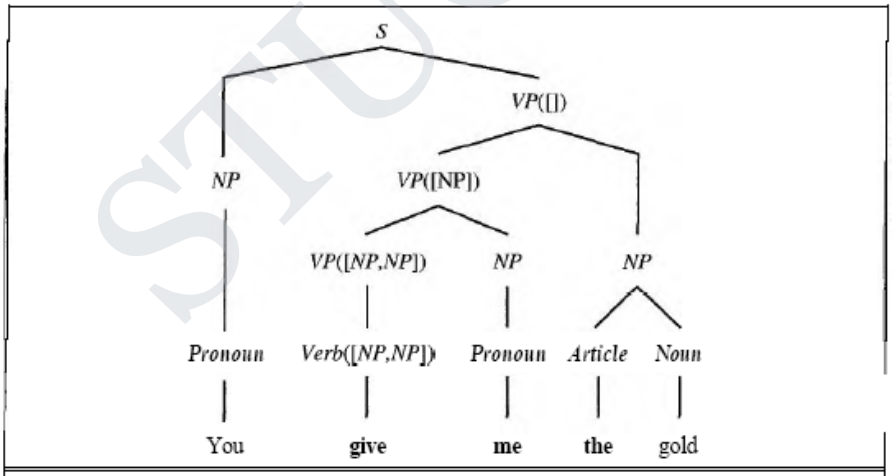


Figure 22.13 Parse tree for "You give me the gold" showing subcategorization of the verb and verb phrase.

(12) What is semantic interpretation? Give an example.

Semantic interpretation is the process of associating an FOL expression with a phrase.

$$Exp(x) \rightarrow Exp(x_1)\ Operator(op)\ Exp(x_2)\ \{x = Apply(op, x_1, x_2)\}$$
$$Exp(x) \rightarrow (\ Exp(x)\ )$$
$$Exp(x) \rightarrow Number(x)$$
$$Number(x) \rightarrow Digit(x)$$
$$Number(x) \rightarrow Number(x_1)\ Digit(x_2)\ \{x = 10 \times x_1 + x_2\}$$
$$Digit(x) \rightarrow x\ \{0 \leq x \leq 9\}$$
$$Operator(x) \rightarrow x\ \{x \in \{+, -, \div, \times\}\}$$

Figure 22.14 A grammar for arithmetic expressions, augmented with semantics. Each variable x_i represents the semantics of a constituent. Note the use of the $\{test\}$ notation to define logical predicates that must be satisfied, but that are not constituents.

(13) Construct a grammar and sentence for "John loves Mary"

$$S(rel(obj)) \rightarrow NP(obj)\ VP(rel)$$
$$VP(rel(obj)) \rightarrow Verb(rel)\ NP(obj)$$
$$NP(obj) \rightarrow Name(obj)$$

$$Name(John) \rightarrow \textbf{John}$$
$$Name(Mary) \rightarrow \textbf{Mary}$$
$$Verb(\lambda y\ \lambda x\ Loves(x, y)) \rightarrow \textbf{loves}$$

Figure 22.16 A grammar that can derive a parse tree and semantic interpretationfor "John loves Mary" (and three other sentences). Each category is augmented with a single argument representing the semantics.

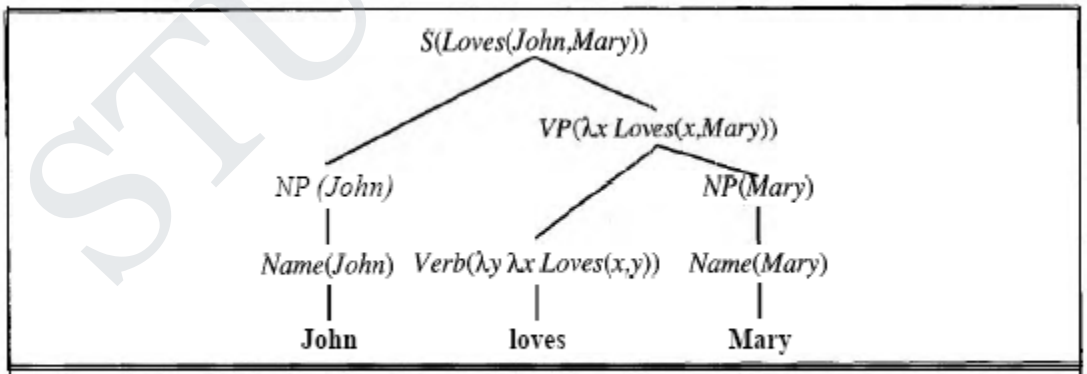


Figure 22.17 A parse tree with semantic interpretations for the string "John loves Mary".

(14) Construct a parse tree for the sentence "Every agent smells a Wumpus"

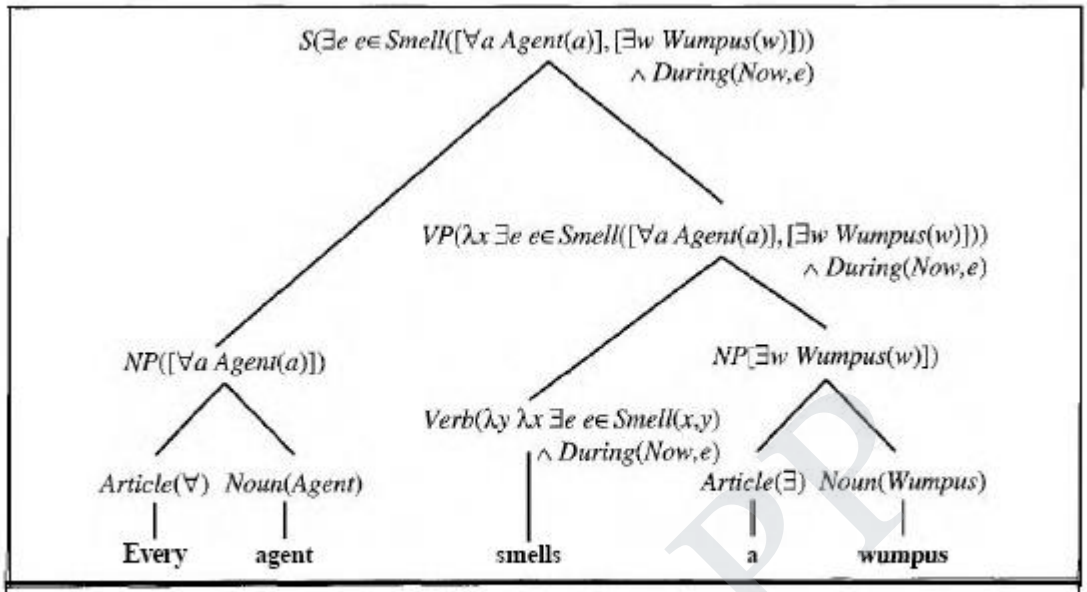


Figure 22.19 Parse tree for the sentence "Every agent smells a wumpus," showing both syntactic structure and semantic interpretations.

(15) Define lexical,syntactic, and semantic ambiguity.

Lexical ambiguity, in which a word has more than one meaning. Lexical ambiguity is quite common; "back" can be an adverb (go back), an adjective (back door), a noun (the back of the room) or a verb (back up your files). "Jack" can be a name, a noun (a playing card, a six-pointed metal game piece, a nautical flag, a fish, a male donkey, a socket, or a device for raising heavy objects), or a verb (to jack up a car, to hunt with a light, or to hit a baseball hard).

Syntactic ambiguity (also known as structural ambiguity) can occur with or without lexical ambiguity. For example, the string "I smelled a wumpus in *2,2"* has two parses: one where the prepositional phrase "in 2,2" modifies the noun and one where it modifies the verb. The syntactic ambiguity leads to a semantic ambiguity, because one parse means that the wumpus is in 2,2 and the other means that a stench is in *2,2.* In this case, getting the wrong interpretation could be a deadly mistake.

Semantic ambiguity can occur even in phrases with no lexical or syntactic ambiguity. For example, the noun phrase "cat person" can be someone who likes felines or the lead of the movie Attack of the Cat People. A "coast road" can be a road that follows the coast or one that leads to it.

(16) What is disambiguation?

Disambiguation

Disambiguation is a question of diagnosis. The speaker's intent to communicate is an unobserved cause of the words in the utterance, and the hearer's job is to work backwards from the words and from knowledge of the situation to recover the most likely intent of the speaker.. Some sort of preference is needed because syntactic and semantic interpretation rules alone cannot identify a unique correct interpretation of a phrase or sentence. So we divide the work: syntactic and semantic interpretation is responsible for enumerating a set of candidate interpretations, and the disambiguation process chooses the best one.

(17) What is discourse?

A discourse is any string of language-usually one that is more than one sentence long. Textbooks, novels, weather reports and conversations are all discourses. So far we have

largely ignored the problems of discourse, preferring to dissect language into individual sentences that can be studied *in vitro.* We will look at two particular subproblems: reference resolution and coherence.

Reference resolution

Reference resolution is the interpretation of a pronoun or a definite noun phrase that refers to an object in the world. The resolution is based on knowledge of the world and of the previous parts of the discourse. Consider the passage "John flagged down the waiter. He ordered a hani sandwich."

To understand that "he" in the second sentence refers to John, we need to have understood that the first sentence mentions two people and that John is playing the role of a customer and hence is likely to order, whereas the waiter is not.

The structure of coherent discourse

If you open up this book to 10 random pages, and copy down the first sentence from each page. The result is bound to be incoherent. Similarly, if you take a coherent 10-sentence passage and permute the sentences, the result is incoherent. This demonstrates that sentences in natural language discourse are quite different from sentences in logic. In logic, if we TELL sentences A, *B* and *C* to a knowledge base, in any order, we endl up with Ithe conjunction *A* A *B A C.* In natural language, sentence order matters; consider th~ed ifference between "Go two blocks. Turn right." and "Turn right. Go two blocks."

(18) What is grammar induction?

Grammar induction is the task of learning a grammar from data. It is an obvious task to attempt, given that it has proven to be so difficult to construct a grammar by hand and that billions of example utterances are available for free on the Internet. It is a difficult task because the space of possible grammars is infinite and because verifying that a given grammar generates a set of sentences is computationally expensive.

Grammar induction can learn a grammar from examples, although there are limitations on how well the grammar will generalize.

(19) What is information retrieval?

Information retrieval is the task of finding documents that are relevant to a user's need for information. The best-known examples of information retrieval systems are search engines on the World Wide Web. A Web user can type a query such as [AI book] into a search engine and see a list of relevant pages. An information retrieval (henceforth IR) system can be characterized by:

1) **A document collection.** Each system must decide what it wants to treat as a document: a paragraph, a page, or a multi-page text.

2) A query posed in a query language. The query specifies what the user wants to know. The query language can be just a list of words, such as [AI book]; or it can specify a phrase of words that must be adjacent, as in ["AI book"]; it can contain Boolean operators as in **[AI** *AND* book]; it can include non-Boolean operators such as [AI book SITE:www.aaai.org].

3) A result set. This is the subset of documents that the IR system judges to be **relevant** to the query. By relevant, we mean likely to be of use to the person who asked the query, for the particular information need expressed in the query.

4) **A presentation of the result set.** This can be as simple as a ranked list of document titles or as complex as a rotating color map of the result set projected onto a three dimensional space.

(20) What is clustering?

Clustering is an unsupervised learning problem. Unsupervised clustering is the problem of discerning multiple categories in a collection of objects. The problem is unsupervised because the category labels are not given.

Examples

We are familiar with terms such as “red giant” and “white dwarf”,but the stars do not carry these labels – astronomers had to perform unsupervised clustering to identify these categories.

(21) What is agglomerative clustering?

Agglomerative clustering creates a tree of clusters going all the way down to the individual documents. We begin by considering each document as a separate cluster. Then we find the two clusters that are closest to each other according to some distance measure and merge these clusters into one. The distance measure between two clusters can be the distance to the median of the cluster. Agglomerative clustering takes time $O(n^2)$,where n is the number of documents.

(22) What is K-means clustering?

K-means clustering creates a flat set of exactly k-categories.It works as follows :

i) Pick K documents at random to represent the K categories.

ii) Assign every document to the closest category.

iii) Compute the mean of each cluster and use K-means to represent the new value of the K categories.

iv) Repeat the steps (ii) and (iii) until convergence.

K-means takes O(n)

(23) What is information extraction?

Information extraction is the process of creating database entries by skimming a text and looking for occurrences of a particular **class of object or event** and for **relationships** among those **objects and events**.

We could be trying to extract instances of addresses from web pages, with database fields for street, city, state, and zip code; or instances of storms from weather reports, with fields for temperature, wind speed, and precipitation. Information extraction systems are mid-way between information retrieval systems and full-text parsers, in that they need to do more than consider a document as a bag of words, but less than completely analyze every sentence.

The simplest type of information extraction system is called an ***attribute-based*** **system** because it assumes that the entire text refers to a single object and the task is to extract attributes of that object.

For example, the problem of extracting from the text "17in SXGA Monitor for only $249.99" the database relations given by

3 m m E *ComputerMonitors A Size(m, Inches(l7)) A Price(m, $(249.99)) A Resolution(m, 1280 x 1024)* .

Some of this information can be handled with the help of regular expressions, which define a regular grammar in a single text string. Regular expressions are used in Unix commands such as grep, in programming languages such as Perl, and in word processors such as Microsoft Word.

(24) What is machine translation? Explain different types.

Machine translation is the automatic translation of text from one natural language (the source) to another (the target). This process has proven to be useful for a number of tasks, including the following:

1. Rough translation, in which the goal is just to get the gist of a passage. Ungrammatical and inelegant sentences are tolerated as long as the meaning is clear. For example, in Web surfing, a user is often happy with a rough translation of a foreign web page.

Sometimes a monolingual human can post-edit the output without having to read the source. This type of machine-assisted translation saves money because such editors can be paid less than bilingual translators.

2**. Restricted-source translation**, in which the subject matter and format of the source text are severely limited. One of the most successful examples is the TAUM-METEO system, which translates weather reports from English to French. It works because the language used in weather reports is highly stylized and regular.

3. **Pre-edited translation**, in which a human preedits the source document to make it conform to a restricted subset of English (or whatever the original language is) before machine translation. This approach is particularly cost-effective when there is a need to translate one document into many languages, as is the case for legal documents in the European Community or for companies that sell the same product in many countries. Restricted languages are sometimes called "Caterpillar English," because Caterpillar Corp. was the first firm to try writing its manuals in this form. Xerox defined a language for its maintenance manuals which was simple enough that it could be translated by machine into all the languages Xerox deals with. As an added benefit, the original English manuals became clearer as well.

4**. Literary translation**, in which all the nuances of the source text are preserved. This is currently beyond the state of the art for machine translation.

(25) Draw a schematic for a machine translation system for English to French.

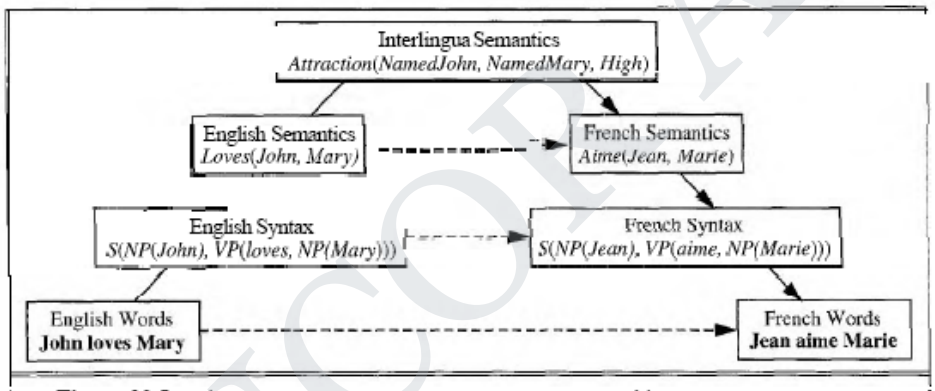


Figure 23.5 A schematic diagram of the choices for a machine translation system. We start with English text at the top. An interlingua-based system follows the solid lines, parsing English first into a syntactic form, then into a semantic representation and an interlingua representation, and then through generation to a semantic, syntactic, and lexical form in French. A transfer-based system uses the dashed lines as a shortcut. Different systems make the transfer at different points; some make it at multiple points.