SCSX1021

ARTIFICIAL INTELLIGENCE

UNIT: I

INTRODUCTION AND PROBLEM SOLVING

Introduction

What is artificial intelligence?

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

John McCarthy, who coined the term in 1956, defines it as "the science and engineering of making intelligent machines, especially intelligent computer programs."

The definitions of AI are categorized into four approaches and are summarized in the table below :

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

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

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

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

The four approaches in more detail are as follows :

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

The Turing Test proposed by Alan Turing in 1950 .The computer is asked questions by a human interrogator. The computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or not.

Programming a computer to pass , the computer need to possess the following capabilities :

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

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

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

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

To pass the complete Turing Test, the computer will need

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

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

A machine program is constructed to think like a human but for that we need to get inside actual working of the human mind :

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

(b) through psychological experiments Eg. Allen Newell and Herbert Simon, who developed GPS, the "General Problem Solver" tried to trace the reasoning steps to traces of human subjects solving the same problems.

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

It is Normative (or prescriptive) rather than descriptive The Greek philosopher Aristotle was one of the first to attempt to codify "right Thinking" that is irrefutable reasoning processes. For example, "Socrates is a man; all men are mortal; therefore Socrates is mortal.". These laws of thought were supposed to govern the operation of the mind; their study initiated a field called logic.

(d) Acting rationally : The Rational Agent Approach Rational behavior: doing the right thing. The right thing: that which is expected to maximize goal achievement, given the available information. An agent is something that acts. Computer agents are not mere programs, but they are expected to have the following attributes also :

(i) operating under autonomous control

(ii) perceiving their environment

(iii) persisting over a prolonged time period

(iv) adapting to change.

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

Foundations of AI

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

Philosophy :

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

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

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

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

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

ii)What can be computed?

iii)How do we reason with uncertain information?

Economics:

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

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

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

Neuroscience:

i)How do brains process information?

Psychology:

i)How do humans and animals think and act?

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

Computer Engineering:

i)How can we build an efficient computer?

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

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

Control theory and Cybernetics:

i)How can artifacts operate under their own control?

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

Linguistics

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

History of AI

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

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

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

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

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

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

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

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

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

Intelligent agent

Agents and Environment

An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and Sensor acting upon that environment through **actuators.** This simple idea is illustrated in the below Figure 1.1.

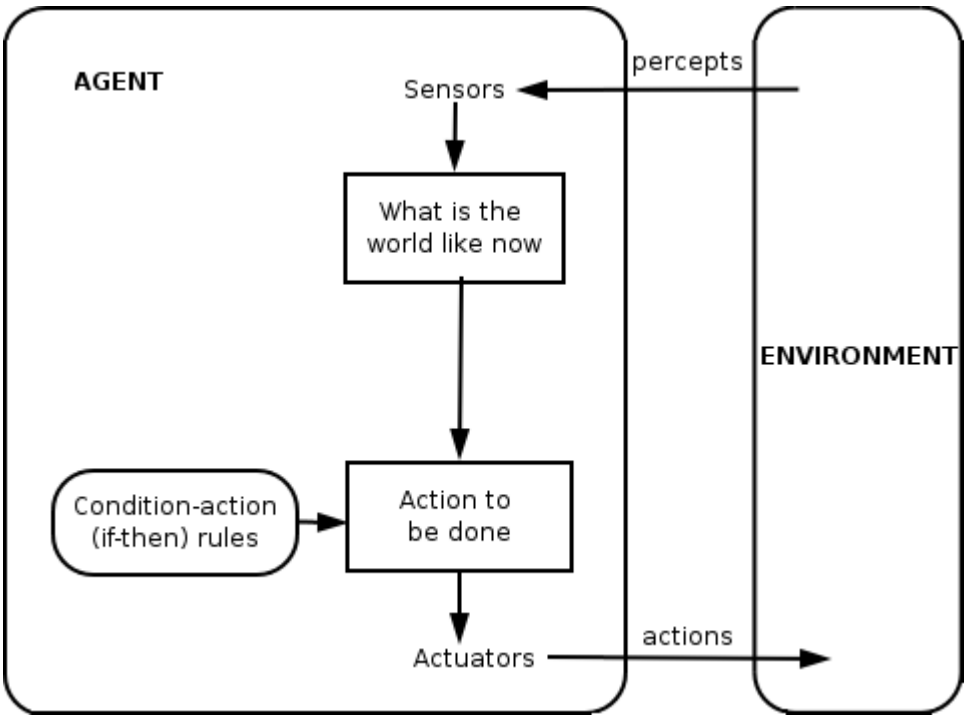


Figure 1.1 Agents interact with environments through sensors and actuators

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

Percept

Percept refers to the agent's perceptual inputs at any given instant.

Percept Sequence

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

Agent function

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

Agent program

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

To illustrate these ideas a very simple example the **vacuum-cleaner** world is shown in Figure 1.2. This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck, otherwise move to the other square. A partial tabulation of this agent function is shown in Figure 1.3.

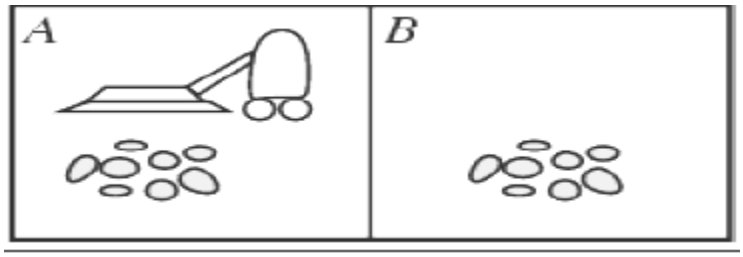


Figure 1.2 Vacuum Cleaner world with two location

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

Figure 1.3 Agent function for Vacuum Cleaner

Concept of Rationality

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

Performance measures

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

Rationality

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

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

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

Omniscience, learning, and autonomy

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

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

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

The Nature of Environment

Specifying the task environment

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

Example: Taxi Task Environment

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

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

Properties of task Environment

i) Fully observable vs. Partially observable

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

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

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

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

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

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

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

iv)Static vs. dynamic

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

E.g. Taxi driving is dynamic.

Chess is semi dynamic.

Crossword puzzles are static.

v)Discrete vs. continuous

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

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

Structure of Agents

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

Intelligent Agent = Agent Program + Architecture

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

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

Types of Agent

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

1.Simple reflex agents

2. Model-based reflex agents

3. Goal-based agents

4. Utility-based agents

1.Simple reflex agents

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

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

Condition action rule: if condition then action

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

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

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

static: rules, a set of condition-action rules

state ← INTERPRET – INPUT(percept)

rule ← RULE – MATCH(state, rules)

action ← RULE – ACTION[rule]

return action
```

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

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

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

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

Example: Medical diagnosis system

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

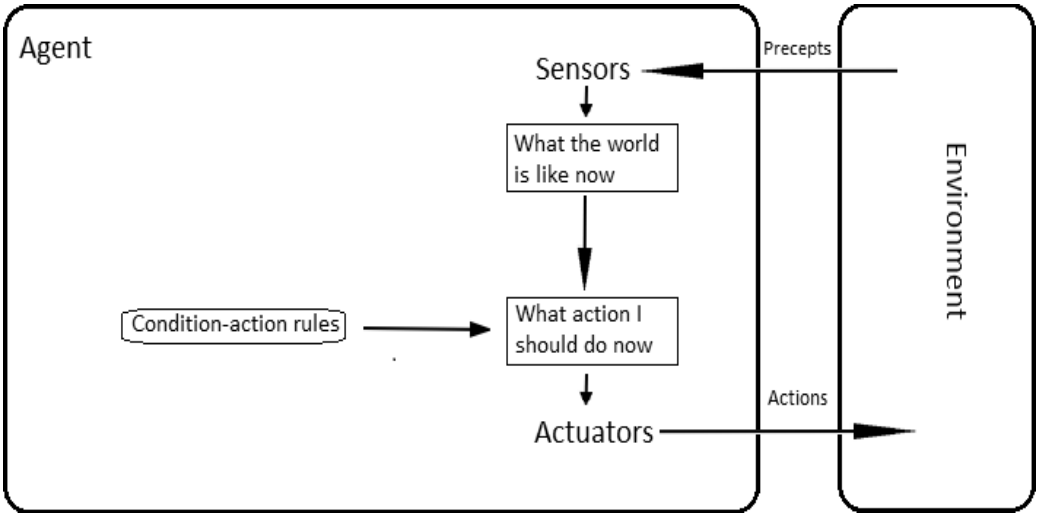


Figure 1.4 Simple Reflex agent

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

- The most effective way to handle partial observability is for the agent ―to keep track of the part of the world it can't see now.
- That is, the agent which combines the current percept with the old internal state to generate updated description of the current state.
- The current percept is combined with the old internal state and it derives a new current state is updated in the state description is also.
- This updation requires two kinds of knowledge in the agent program. First, we need some information about how the world evolves independently of the agent. Second, we need some information about how the agents own actions affect the world.

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

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

The agent program, which is shown below

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

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

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

rule ← RULE-MATCH(state, rules)

action ← RULE-ACTION[rule]
```

return action

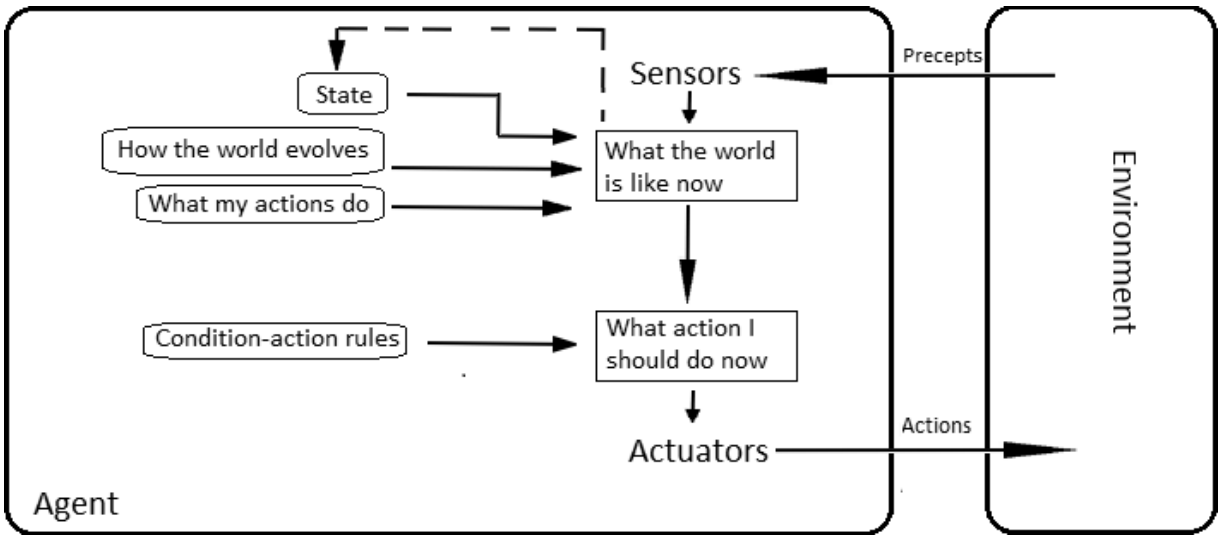


Figure 1.5 Model based Agent

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

3.Goal-based agents

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

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

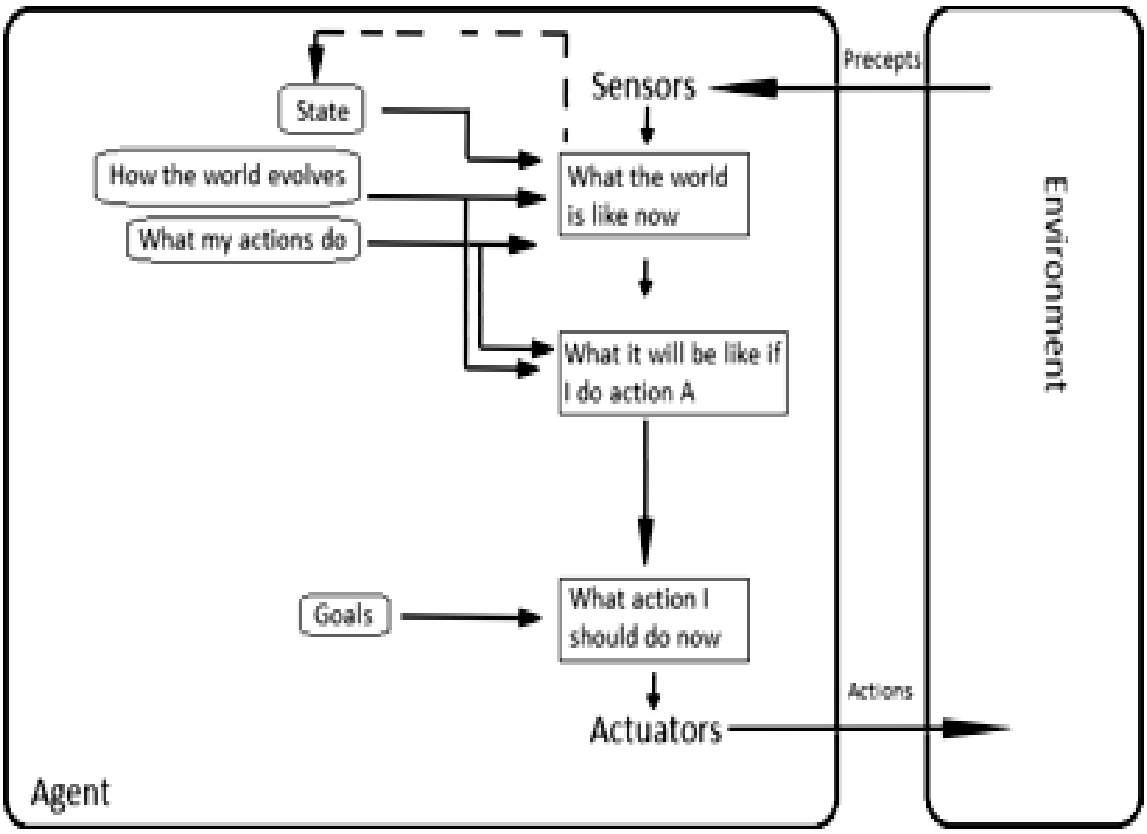


Figure 1.6 Goal based agent

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

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

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

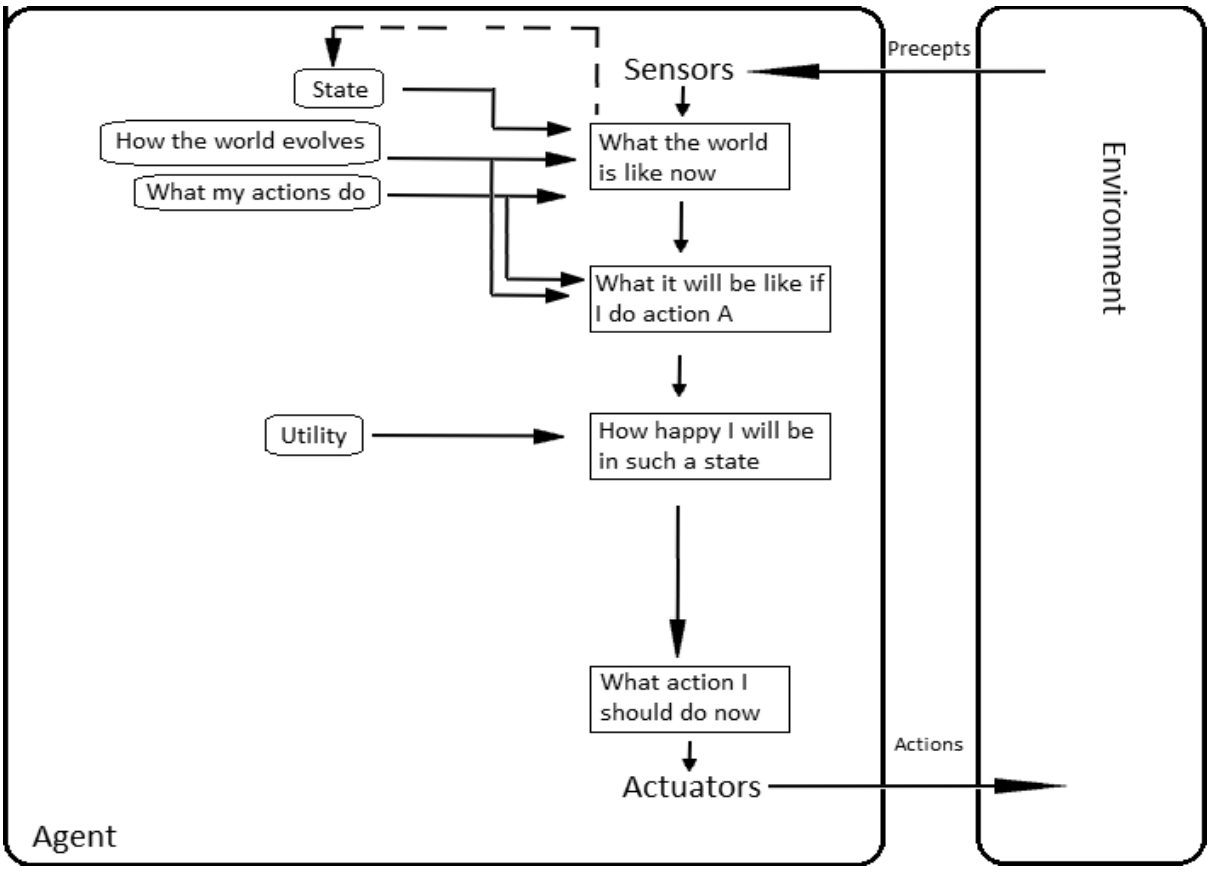


Figure 1.7 Utility based Agent

5.Learning agents

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

i)Learning element – This is responsible for making improvements. It uses the feedback from the critic on how the agent is doing and determines how the performance element should be modified to do better in the future.

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

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

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

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

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

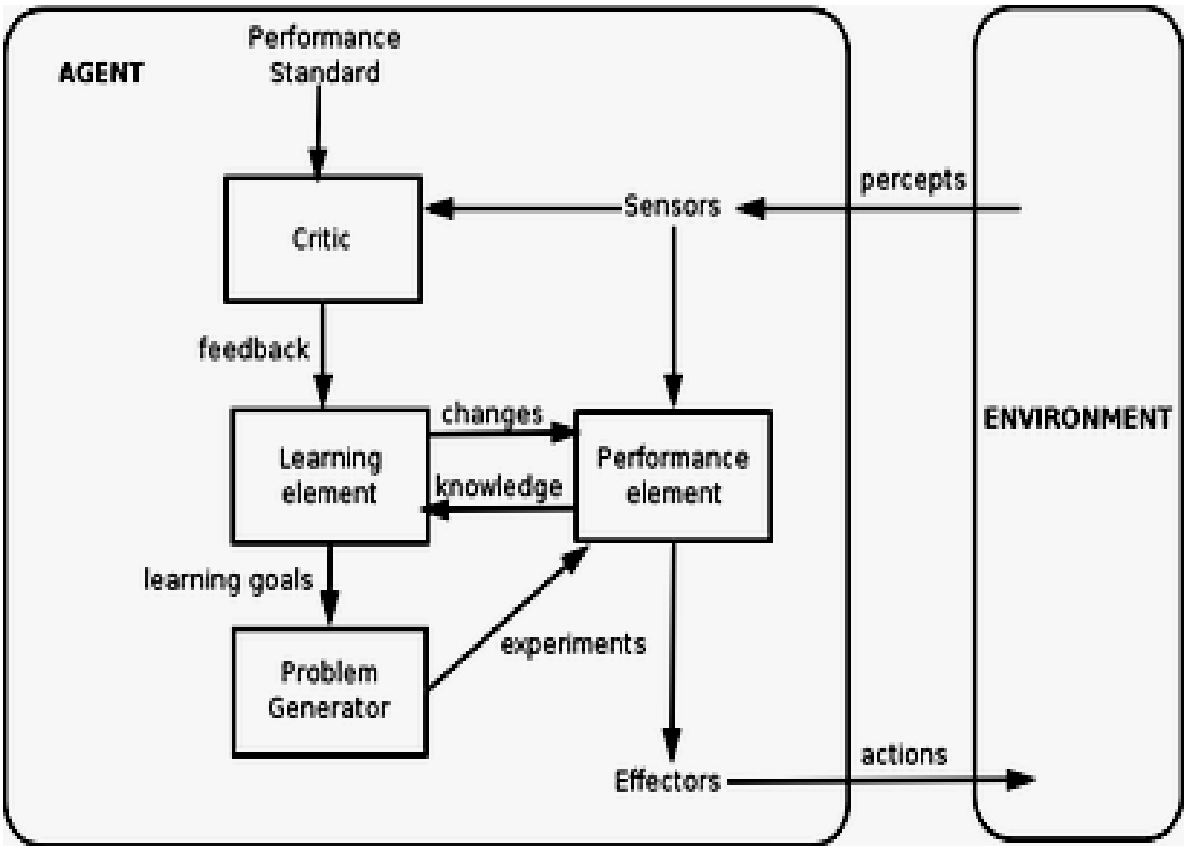


Figure 1.8 Learning Based Agent

Problem solving agents

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

Problem Formulation

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

Steps to Maximize the Performance Measure:

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

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

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

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

The below program explains about the simple problem solving agent.

```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
inputs : percept, a percept
static: seq, an action sequence, initially empty
        state, some description of the current world state
        goal, a goal, initially null
        problem, a problem formulation
state UPDATE-STATE(state, percept)
if seq is empty then do
        goal FORMULATE-GOAL(state)
        problem FORMULATE-PROBLEM(state, goal)
        seq SEARCH( problem)
action FIRST(seq);
seq REST(seq)
return action
```

The agent design assumes the Environment is

• **Static** : The entire process carried out without paying attention to changes that might be occurring in the environment.

• **Observable :** The initial state is known and the agent's sensor detects all aspects that are relevant to the choice of action

• **Discrete :** With respect to the state of the environment and percepts and actions so that alternate courses of action can be taken

• **Deterministic :** The next state of the environment is completely determined by the current state and the actions executed by the agent. Solutions to the problem are single sequence of actions

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

Well-defined problems and solutions

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

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

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

For example, from the state In(Arad),the successor function for the Romania problem would return{[Go(Sibiu),In(Sibiu)],[Go(Timisoara),In(Timisoara)],[Go(Zerind),In(Zerind)] }

i)State Space: The set of all states reachable from the initial state. The state space forms a graph in which the nodes are states and the arcs between nodes are actions.

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

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

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

(a) The **step cost** of taking action a to go from state x to state y is denoted by c(x,a,y). The step cost for Romania is shown in figure 1.9. It is assumed that the step costs are non negative.

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

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

Abstraction

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

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

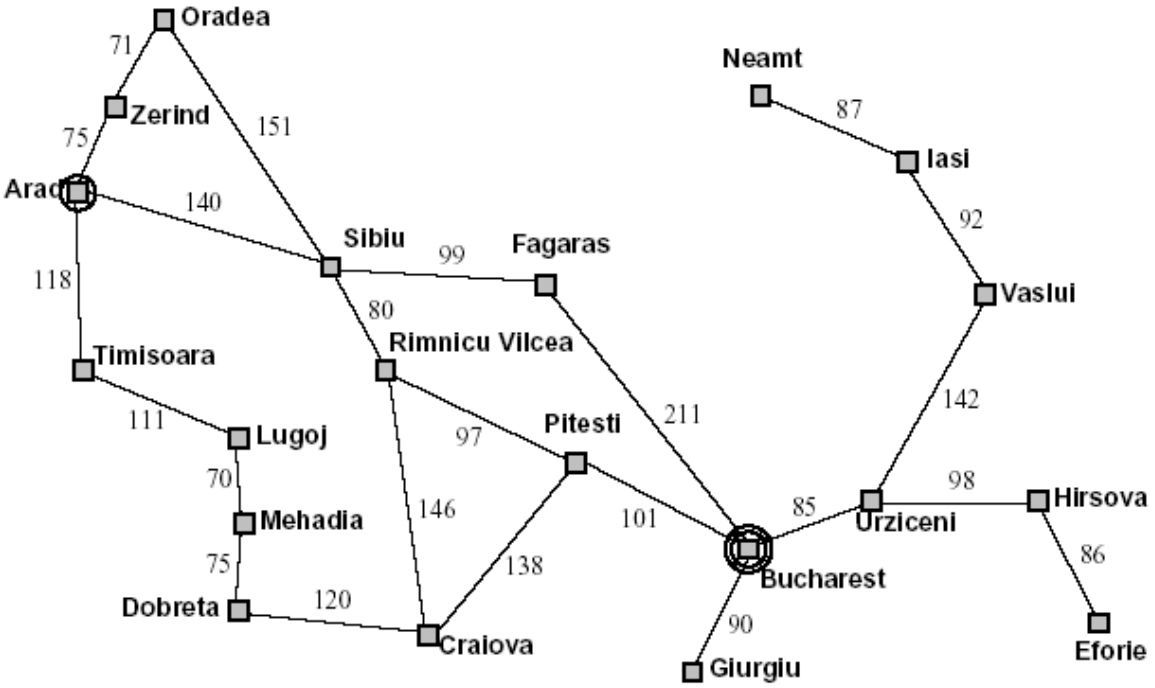


Figure 1.9 Simplified Road map of Romania

Example: Route finding problem

On holiday in Romania : currently in Arad.

Flight leaves tomorrow from Bucharest

Formulate goal: be in Bucharest

Formulate problem:

states: various cities

actions: drive between cities

Find solution:

sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Problem formulation

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

initial state e.g., “at Arad"

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

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

path cost (additive)

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

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

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

EXAMPLE PROBLEMS

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

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

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

i)TOY PROBLEMS

1.Vacuum World Example

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

o **Initial state**: Any state can be designated as initial state.

Successor function: This generates the legal states that results from trying the three actions (left, right, suck). The complete state space is shown in figure 2.3

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

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

Vacuum World State Space shown in figure 1.10

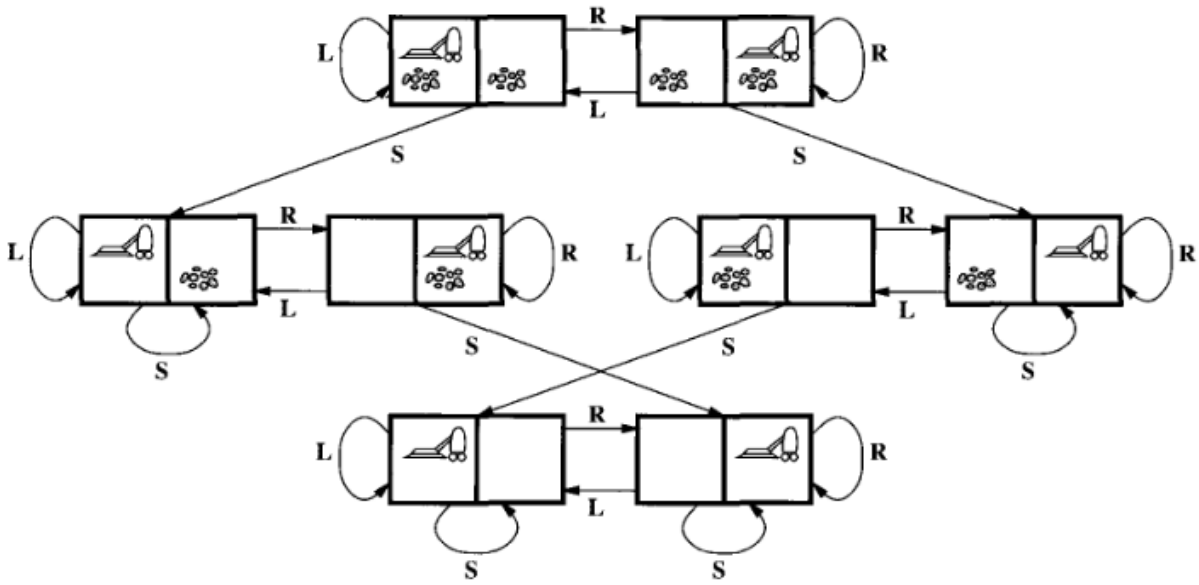


Figure 1.10 State space for the Vaccum world

2.The 8-puzzle

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

Example: The 8-puzzle shown in Figure 1.11.

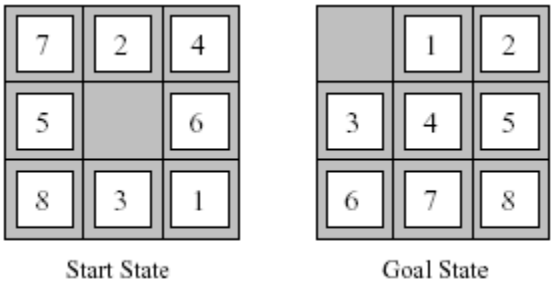


Figure 1.11 8 Puzzle

The problem formulation is as follows :

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

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

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

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

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

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

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

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

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

3.8-queens problem

The goal of 8-queens problem is to place 8 queens on the chessboard such that no queen attacks any other.(A queen attacks any piece in the same row, column or diagonal). Figure 1.12 shows an attempted solution that fails: the queen in the right most columns is attacked by the queen at the top left.

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

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

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

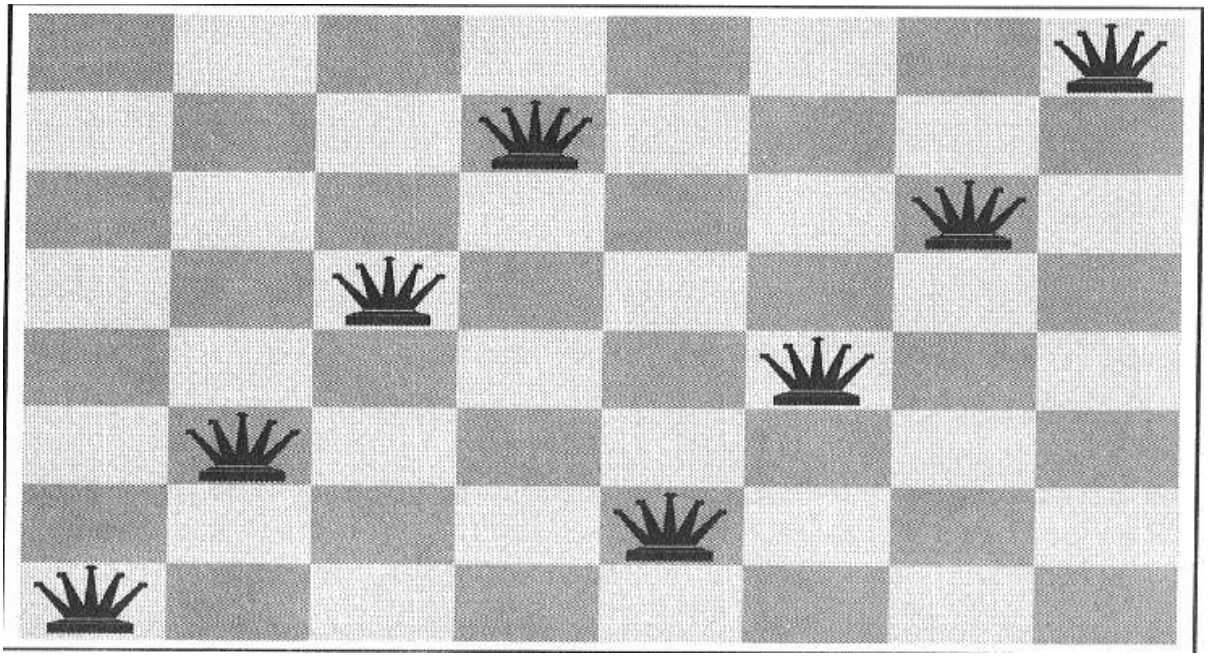


Figure 1.12 8 Queens Problem

The first incremental formulation one might try is the following :

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

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

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

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

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

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

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

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

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

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

REAL-WORLD PROBLEMS

i)ROUTE-FINDING PROBLEM

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

ii)AIRLINE TRAVEL PROBLEM

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

States: Each is represented by a location(e.g., an airport) and the current time.

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

o **Successor function:** This returns the states resulting from taking any scheduled flight (further specified by seat class and location),leaving later than the current time plus the within-airport transit time, from the current airport to another.

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

o **Path cost:** This depends upon the monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of dat, type of air plane, frequent-flyer mileage awards, and so on.

iii)TOURING PROBLEMS

Touring problems are closely related to route-finding problems, but with an important difference.

Consider for example, the problem,"Visit every city at least once" as shown in Romania map.

As with route-finding the actions correspond to trips between adjacent cities. The state space, however, is quite different.

The initial state would be "In Bucharest; visited{Bucharest}".

A typical intermediate state would be "In Vaslui; visited {Bucharest, Urziceni, Vaslui}".

The goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

iv)THE TRAVELLING SALESPERSON PROBLEM(TSP)

Is a touring problem in which each city must be visited exactly once. The aim is to find the shortest tour. The problem is known to be **NP-hard**. Enormous efforts have been expended to improve the capabilities of TSP algorithms. These algorithms are also used in

tasks such as planning movements of **automatic circuit-board drills** and of **stocking machines** on shop floors.

v)VLSI layout

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

vi)ROBOT navigation

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

vii)AUTOMATIC ASSEMBLY SEQUENCING

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

viii)INTERNET SEARCHING

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

SEARCHING FOR SOLUTIONS

SEARCH TREE

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

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

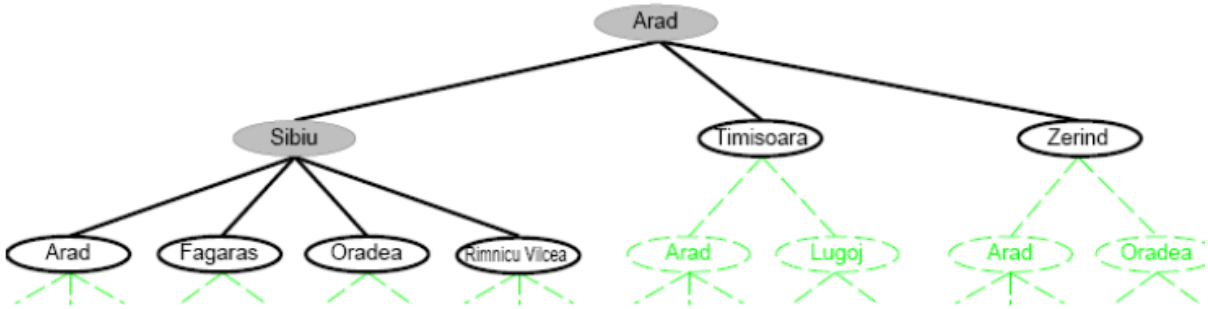


Figure 1.13 Partial search tree for finding the route

From the above diagram

- Nodes that have been expanded are shaded.;
- nodes that have been generated but not yet expanded are outlined in bold;
- nodes that have not yet been generated are shown in faint dashed line

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

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

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

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

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

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

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

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

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

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

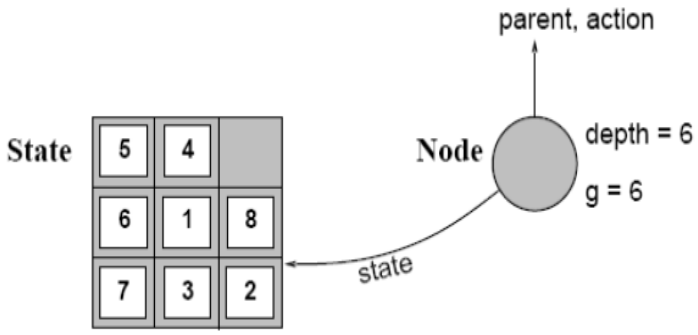


Figure 1.14 Nodes are the data structure from which the search tree is constructed.

Fringe

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

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

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

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

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

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

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

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

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

MEASURING PROBLEM-SOLVING PERFORMANCE

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

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

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

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

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

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

Uninformed search strategies

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

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

There are five uninformed search strategies as given below.

o Breadth-first search

o Uniform-cost search

o Depth-first search

o Depth-limited search

o Iterative deepening search

Breadth first search

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

Algorithm: Breadth-First Search

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

i) Apply the rule to generate a new state.

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

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

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

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

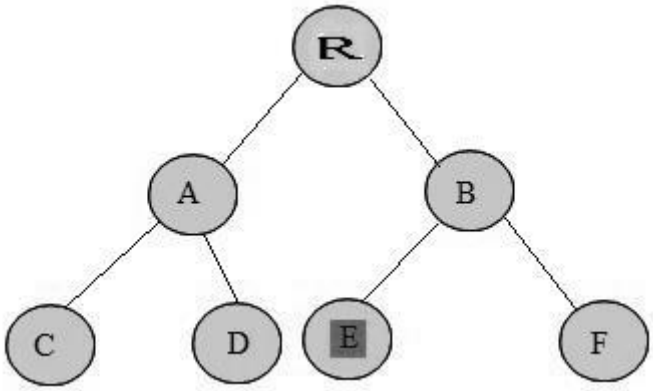


Figure 1.15 Simple binary tree for Breadth First Search

Breadth First Search

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

Advantages of Breadth-First Search

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

Disadvantages of Breadth-First Search

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

Uniform cost search

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

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

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

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

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

Depth first search

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

Algorithm: Depth First Search

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

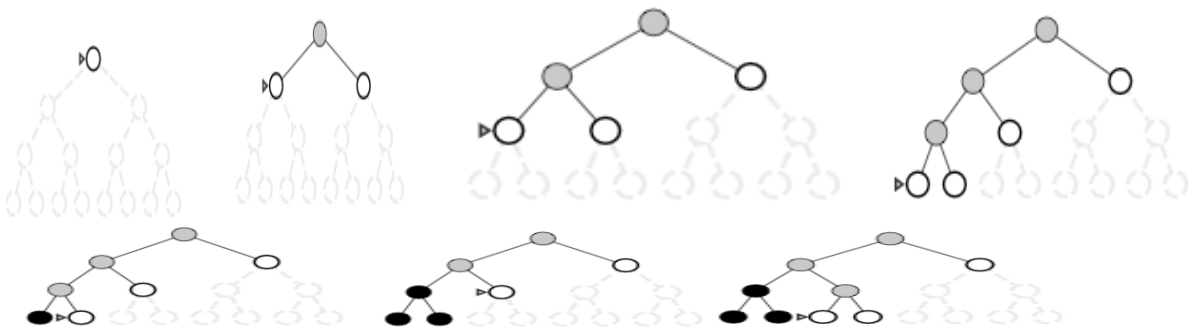
2.Otherwise, loop until success or failure is signaled.

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

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

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

Example:



Advantages of Depth-First Search

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

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

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

Disadvantages of Depth-First Search

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

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

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

Depth limited search

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

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

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

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

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

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

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

Iterative-Deepening Search

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

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

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

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

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

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

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

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

inputs: *problem*

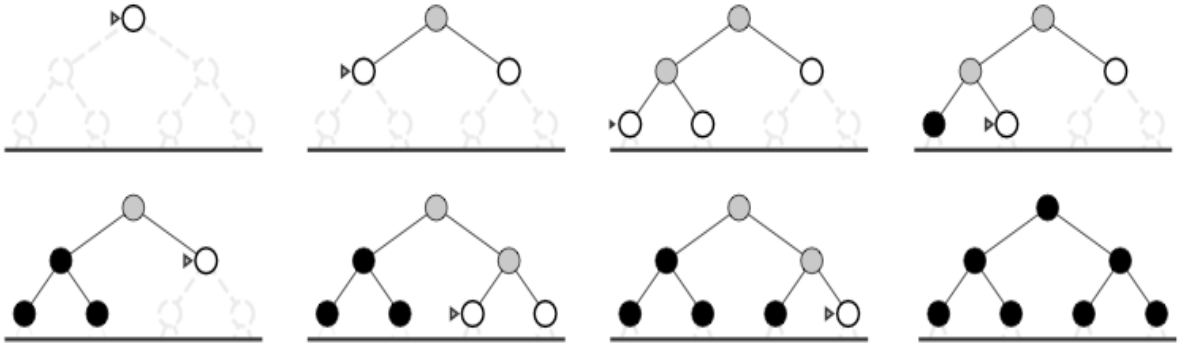
for *depth* ← 0 to ∞ **do**

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

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

Example: Limit=1

If Limit=2



Bidirectional Search

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

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

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

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

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

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

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

Time Complexity

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

Advantages

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

Disadvantages

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

Summary of algorithms

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

Searching with partial Information

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

i)sensorless problem

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

ii)contingency problem

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

iii)Exploration problems

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

Sensorless Problems

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

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

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

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

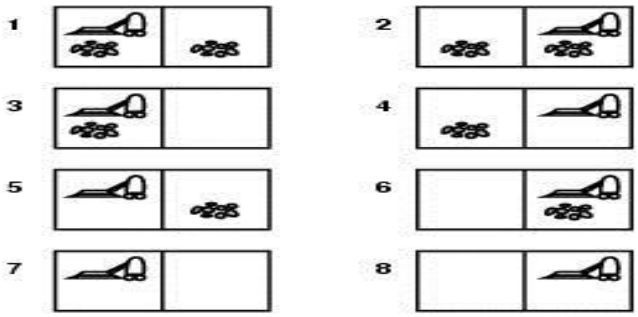


Figure 1.16 Eight possible states of Vacuum world

Partial knowledge of states and actions:

– sensorless or conformant problem

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

– contingency problem

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

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

– exploration problem

– When states and actions of the environment are unknown.

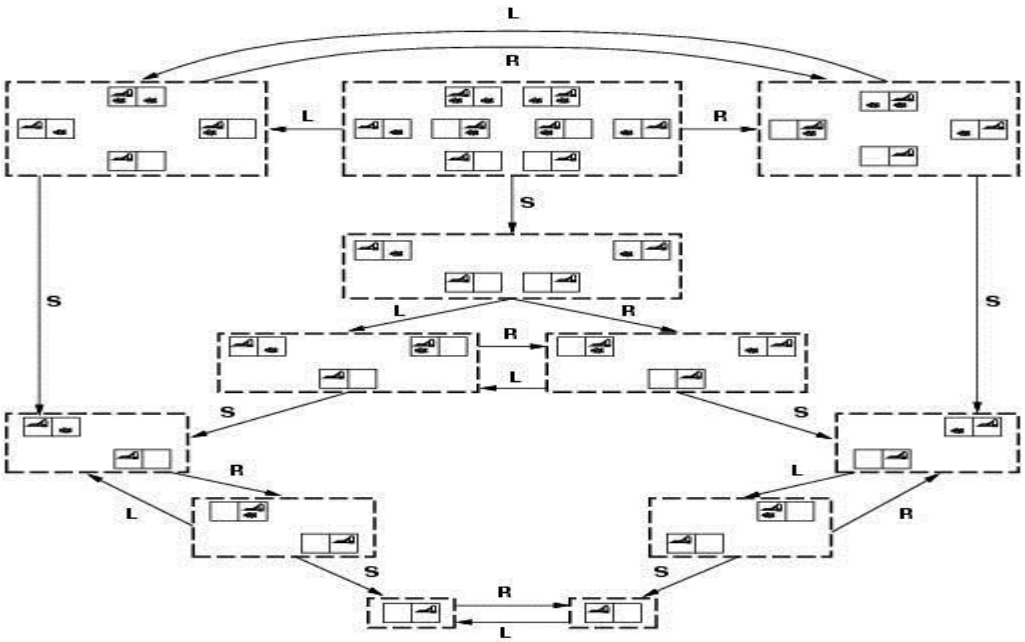


Figure 1.17 Belief State Space Search

Contingency, start in {1,3}.

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

Local sensing: dirt, location only.

– Percept = [L, Dirty] ={1,3}

– [Suck] = {5,7}

– [Right] ={6,8}

– [Suck] in {6}={8} (Success)

– BUT [Suck] in {8} = failure

Solution??

– Belief-state: no fixed action sequence guarantees solution

Relax requirement:

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

– Select actions based on contingencies arising during execution.

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

Contingency problems

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

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

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

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

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

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

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

SCSX1021 - ARTIFICIAL INTELLIGENCE

UNIT – II

Informed Search

Introduction

- Informed search strategies use problem specific knowledge beyond the definition of the problem itself.
- Use the knowledge of the problem domain to build an evaluation function f.
- For every node n is the search space $f(n)$ quantifiers the desirability of expanding n in order to reach the goal.
- To solve large problem with large number of possible states problem specific knowledge need to be added to increase the efficiency of search algorithm.
- Can find solution more efficiently than a uniformed strategy.
- A key point of informed search strategy is heuristic function. So it is called as **heuristic function.**

Strategies

Best First Search:

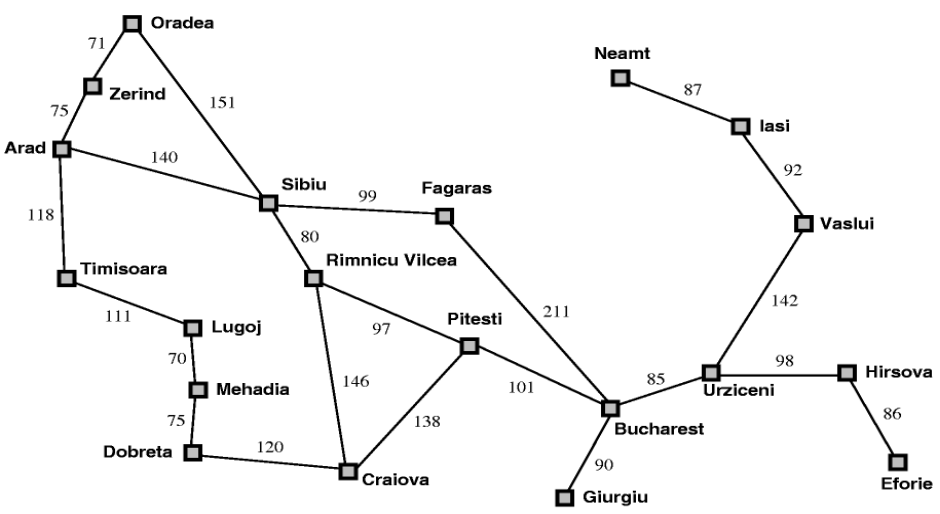
- Best First Search is an instance of the general TREE SEARCH or GRAPH SEARCH algorithm in which a node is selected for expansion based on an evaluation function$f(n)$.
- The Best First Search algorithms have different evaluation functions. A key component of these algorithms is a heuristic function denoted $h(n)$.
- $h(n)$ = estimated cost of the cheapest path from node n to a goal node.
- **For example:** From Figure 2.1**,** in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight line distance from Arad to Bucharest.
- **Heuristic functions are** the most common form in which additional knowledge of the problem is imparted to the search algorithm.
- It can implemented as:

```
function BEST-FIRST-SEARCH(problem, EVAL-FN) returns a solution sequence
  inputs: problem, a problem
          Eval-Fn, an evaluation function

  Queueing-Fn ← a function that orders nodes by EVAL-FN
  return GENERAL-SEARCH(problem, Queueing-Fn)
```

Greedy Best First Search:

- The simplest best-first strategy is to minimize the estimated cost to reach the goal, i.e., always expand the node that appears to be closest to the goal. A function that calculates cost estimates is called an **heuristic function**.
- h (n) = estimated cost of the cheapest path from the state at n to a goal state
- A best-first search that uses h to select the next node to expand is called a **greedy search**.
- To get an idea of what an heuristic function looks like, lets look at a particular problem.
- Here we can use as h the **straight-line distance** to the goal.
- To do this, we need the map co-ordinates of each city.
- This heuristic works because roads tend to head in more or less of a straight line.



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 Road Map

This figure 2.2 shows the progress of a greedy search to find a path from Arad to

Bucharest.

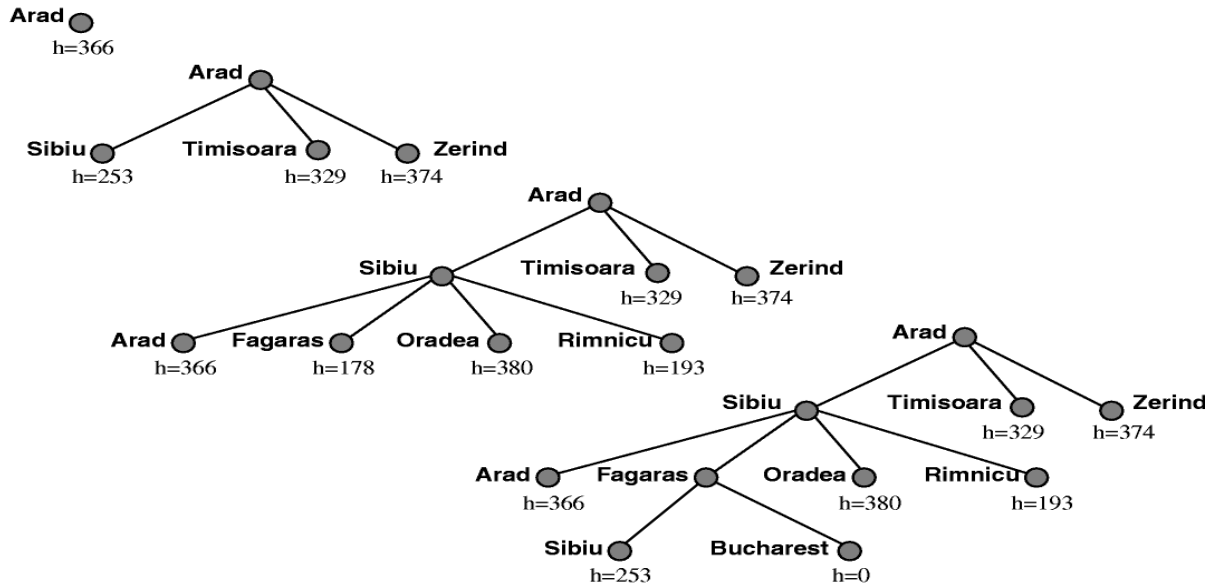


Figure 2.2 Search for Bucharest using the straight line distance heuristic

- For this problem, greedy search leads to a minimal cost search because no node off the solution path is expanded.
- However, it does not find the optimal path: the path it found via Sibiu and Fagaras to Bucharest is 32 miles longer than the path through Pimnicu Vilcea and Pitesti.
- Hence, the algorithm always chooses what looks locally best, rather than worrying about whether or not it will be best in the long run. (This is why its called greedy search.) Greedy search is susceptible to false starts.
- Consider the problem of getting from Iasi to Fagaras. h suggests that Neamt be expanded first, but it is a dead end.
- The solution is to go first to Vaslui and then continue to Urziceni, Bucharest and Fagaras. Note that if we are not careful to detect repeated states, the solution will never be found - the search will oscillate between Neamt and Iasi.
- Greedy search resembles dfs in the way that it prefers to follow a single path to the goal and backup only when a dead end is encountered.
- It suffers from the same defects as dfs -it is not optimal and it is incomplete because it can start down an infinite path and never try other possibilities.

- The worst-case complexity for greedy search is $O(b^m)$, where m is the maximum depth of the search.
- Its space complexity is the same as its time complexity, but the worst case can be substantially reduced with a good heuristic function.

A* Heuristic Function:(Minimizing the total estimated solution cost)

A* Search is the most widely used form of best-first search. **A* Search** evaluates the node by combining

g(n) = the cost to reach the node, and

h(n) = the cost to get from the node to the **goal** :

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

where

f(n) -> Cheapest solution cost.

g(n) -> path cost from the start node to the node "n"

h(n) -> cheapest path cost from the node "n" to the goal node

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 hSLD.

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 hSLD that we usedin getting to Bucharest. The progress of an A* tree search for Bucharest is shown in Figure 2.3.

The values of g are computed from the step costs shown in the Romania map(figure 2.1). Also the values of hSLD are given in Figure 2.1.

An obvious example of an admissible heuristic is the straight-line distance *hSLD* that we used in getting to Bucharest. Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate.

In Figure 2.3, we show the progress of an A* tree search for Bucharest. The values of *g* are computed from the step costs in Figure 2.1, and the values of *hSLD* are given in Figure 2.3.

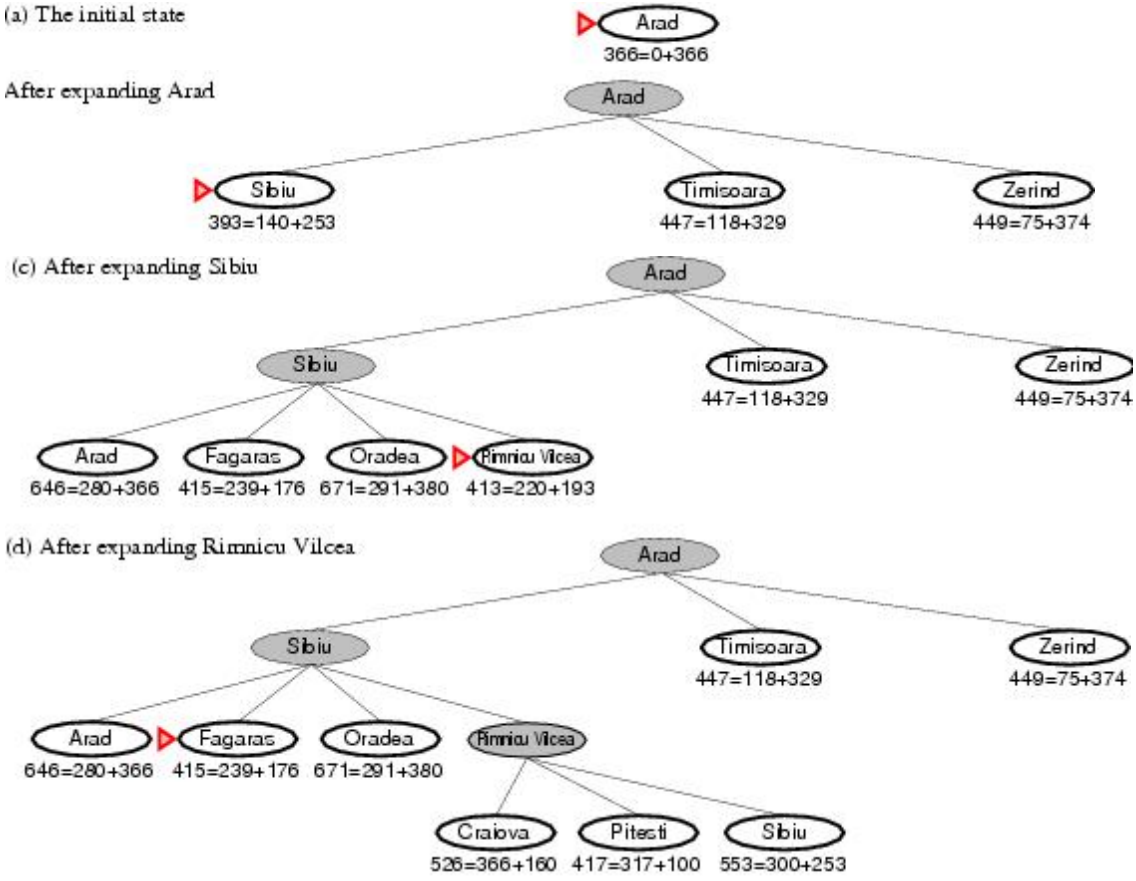
Notice in particular that Bucharest first appears on the fringe at step (e), but it is not selected for expansion because its *f-cost* (450) is higher than that of Pitesti (417). Another way to say this is that there *might* be a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450.

Optimality of A*

- Provided that h(n) never overestimates the cost to reach the goal, then in tree search A* gives the optimal solution.
- Suppose G2 is a suboptimal goal node generated to the tree.
- Let C* be the cost of the optimal solution .
- Because G2 is a goal node, it holds that h(G2) = 0, and we know that f(G2) = g(G2) > C*.
- On the other hand, if a solution exists, there must exist a node n that is on the optimal solution path in the tree.
- Because h(n) does not overestimate the cost of completing the solution path, f(n) = g(n) + h(n) C*.
- We have shown that f(n) C* < f(G2), so G2 will not be expanded and A* must return an optimal solution.
- In graph search finding an optimal solution requires taking care that the optimal solution is not discarded in repeated states .
- A particularly important special case are consistent (or monotonic) heuristics for which the triangle inequality holds in form h(n) c([n,a], n') + h(n'), where n' S(n) (the chosen action is a) and c([n,a], n') is the step cost.
- Straight-line distance is also a monotonic heuristic.
- A* using a consistent heuristic h(n) is optimal also for graph search.
- If h(n) is consistent, the values of f(n) along any path are no decreasing.
- Suppose that n' is a successor of n so that g(n') = c([n,a], n') + g(n) f(n') = g(n') + h(n') = c([n,a], n') + g(n) + h(n') • g(n) + h(n) = f(n).
- Hence, the first goal node selected for expansion (in graph search) must be an optimal solution.
- In looking for a solution, A* expands all nodes n for which f(n) < C*, and some of those for which f(n) = C* .
- However, all nodes n for which f(n) > C* get pruned.
  It is clear that A* search is complete .
- A* search is also optimally efficient for any given heuristic function, because any algorithm that does not expand all nodes with f(n) < C* runs the risk of missing the optimal solution.
- Despite being complete, optimal, and optimally efficient, A* search also has its weaknesses.

- The number of nodes for which f(n) < C* for most problems is exponential in the length of the solution.

Example:



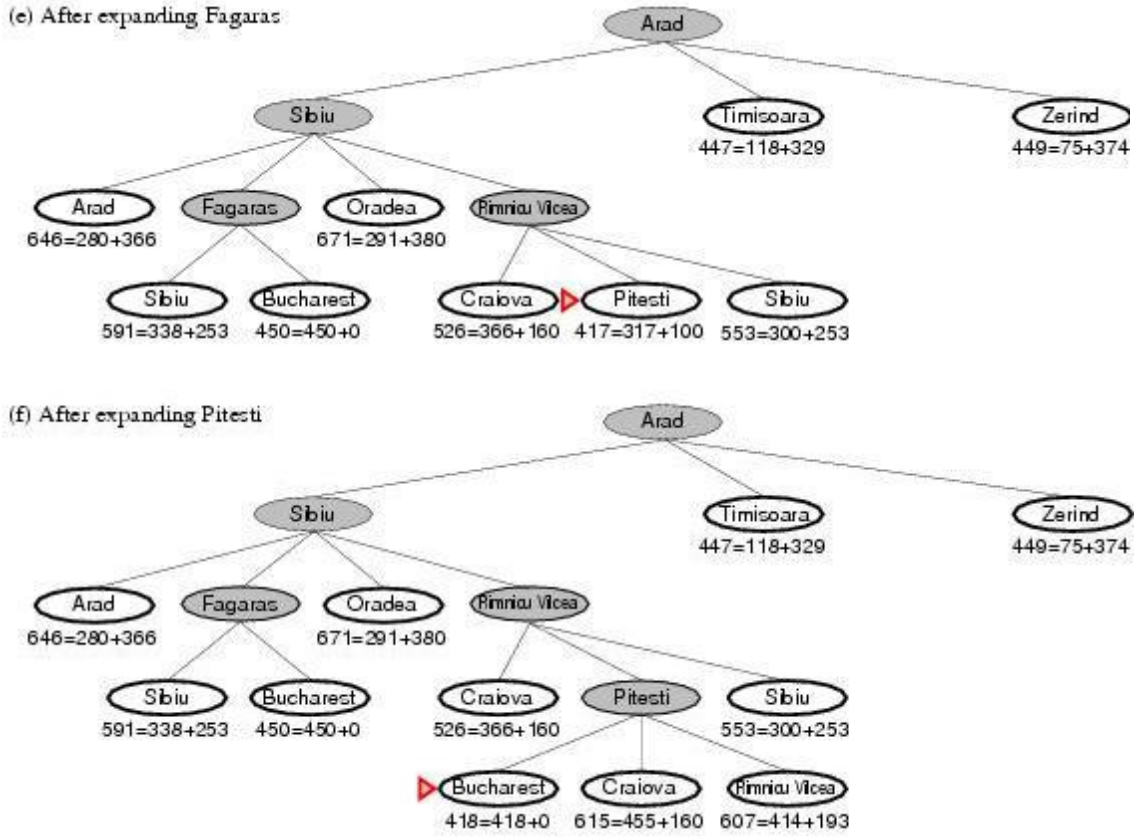


Figure 2.3 Stages in A* Search

Drawbacks

i) A*search keeps all generated nodes in memory (as do all GRAPH-SEARCH algorithms)

ii) A* usually runs out of space long before it runs out of time.

iii) A* is not practical for many large-scale problems and recently developed algorithms have overcome the space problem without sacrificing optimality or completeness, at a small cost in execution time.

Memory bounded heuristic search:

The simplest way to reduce memory requirements for A* is to adapt the idea of iterative deepening to the heuristic search context, resulting in the iterative-deepening A* (lDA*) algorithm.

The main difference between IDA* and standard iterative deepening is that the cutoff used is the *f-cost (g + h)* rather than the depth; at each iteration, the cutoff value is the smallest *f-cost* of any node that exceeded the cutoff on the previous iteration. IDA* is practical for many

problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes.

2 memory bounded algorithm:

1) RBFS (recursive best-first search).

2) MA* (Memory-bounded A*) and SMA*(simplified memory MA*)

RBFS:

- It attempts to mimic the operation of BFS.
- Suffers from using too little memory.
- Even if more memory were available , RBFS has no way to make use of it.
- 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.

```
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 fcost 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 shows how RBFS reaches Bucharest.

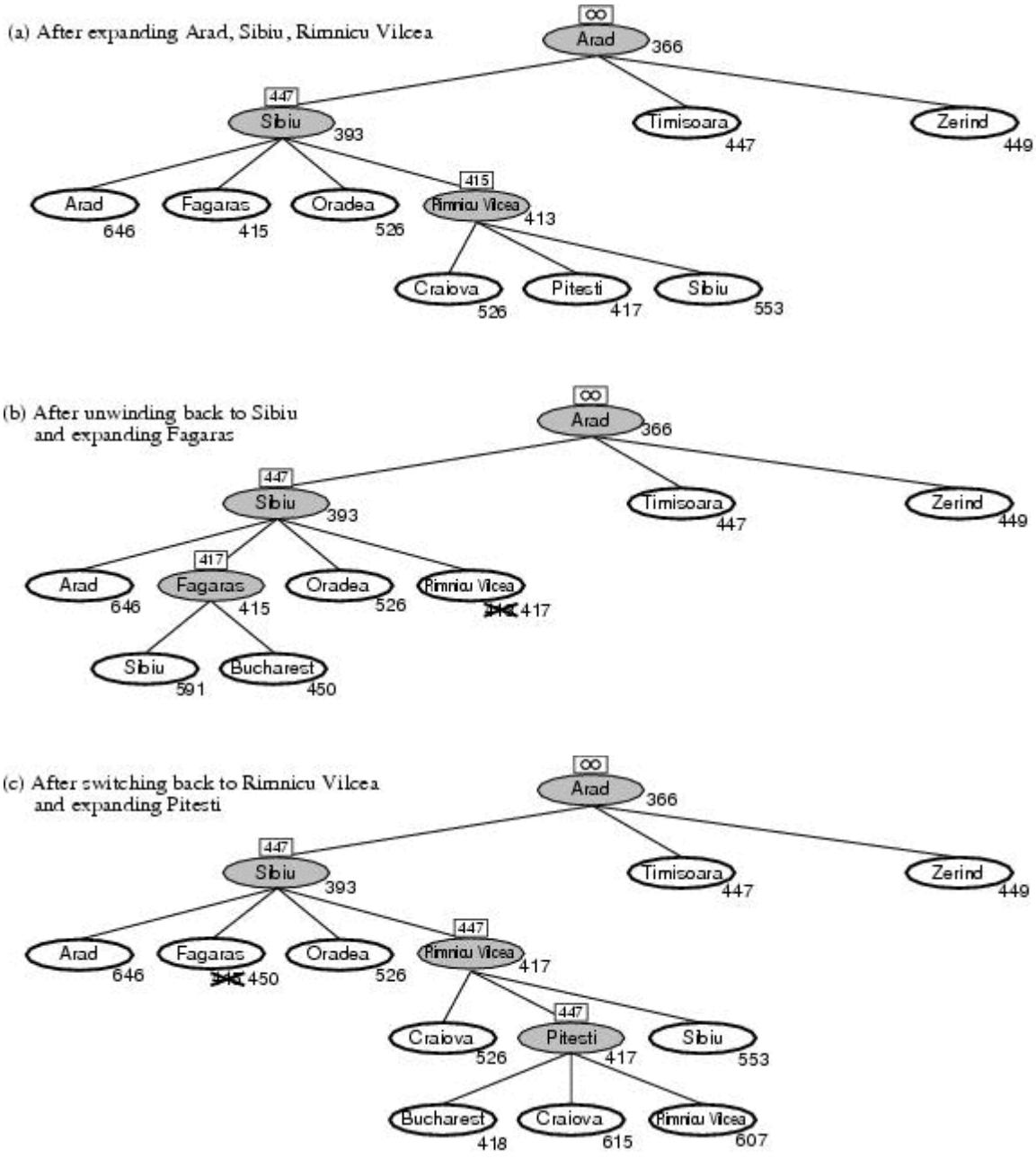


Figure 2.4 Stages in an RBFS search for the shortest route to Bucharest.

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* and RBFS suffer from using ***too little*** memory. Between iterations, IDA* retains only a single number: the current *I-cost* limit. REFS retains more information in memory, but it uses only *O(bd)* memory: even if more memory were available, RBFS has no way to make use of it. A Search technique which uses all available memory, two algorithms that do this

(1) MA* (memory-bounded A*) and

(2) SMA* (simplified MA*)

SMA*:

- Proceeds life A*,expands best leaf until memory is full.
- Cannot add new node without dropping an old one. (always drops worst one)
- Expands the best leaf and deletes the worst leaf.
- If all have same f-value-selects same node for expansion and deletion.
- SMA* is complete if any reachable solution.
- SMA* is optimal if any optimal solution is reachable; otherwise it returns the best reachable solution.SMA* might well be the best general-purpose algorithm for finding optimal solutions, particularly when the state space is a graph, step costs are not uniform, and node generation is expensive compared to the additional overhead of maintaining the open and closed lists.

Heuristic Function

A Heuristic technique helps in solving problems, even though there is no guarantee that it will never lead in the wrong direction. There are heuristics of every general applicability as well as domain specific. The strategies are general purpose heuristics. In order to use them in a specific domain they are coupler with some domain specific heuristics. There are two major ways in which domain - specific, heuristic information can be incorporated into rule-based search procedure.

Some different heuristics for the 8-puzzle problem shown in figure 2.5. A typical solution to the puzzle has around 20 steps. The branching factor is about 3. Hence, an exhaustive search to depth 20 will look at about $3^{20} = 3.5 \times 10^9$ states.

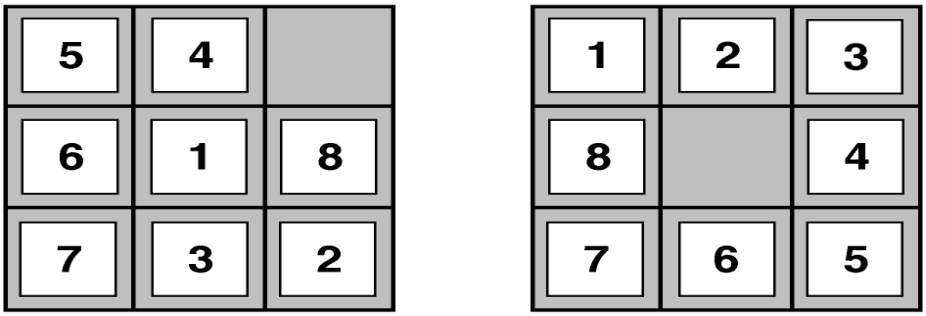


Figure 2.5 8 Puzzle Problem

By keeping track of repeated states, this number can be cut down to 9!=362,880 different arrangements of 9 squares. We need a heuristic to further reduce this number.If we want to find shortest solutions, we need a function that never overestimates the number of steps to the goal. Here are two possibilities:

- h1= the number of tiles that are in the wrong position. This is admissible because any tile that is out of place must be moved at least once.
- h2= the sum of the distances of the tiles from their goal positions. Since tiles cannot be moved diagonally, we use **city block distance**. h2 is also admissible.

The effect of heuristic accuracy on performance

One way to characterize the quality of an heuristic is the **effective branching factor** b^*. If the total number of nodes expanded 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 nodes.

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

Example if A^* finds a solution at depth 5 using 52 nodes, the effective branching factor is 1.91. The effective branching factor of an heuristic is fairly constant over problem instances and, hence, experimental measurements of b^* on a small set of problems provides a good approximation of the heuristic's usefulness.

A well designed heuristic should have a value of b^* that is close to 1. The following is a table of the performance of A^* with h1 and h2 above on 100 randomly generated problems. It shows that h2 is better than h1 and that both are much better than 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	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	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

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.

State Space Landscape

To understand local search, it is better explained using **state space landscape** as shown in figure 2.6.

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**.

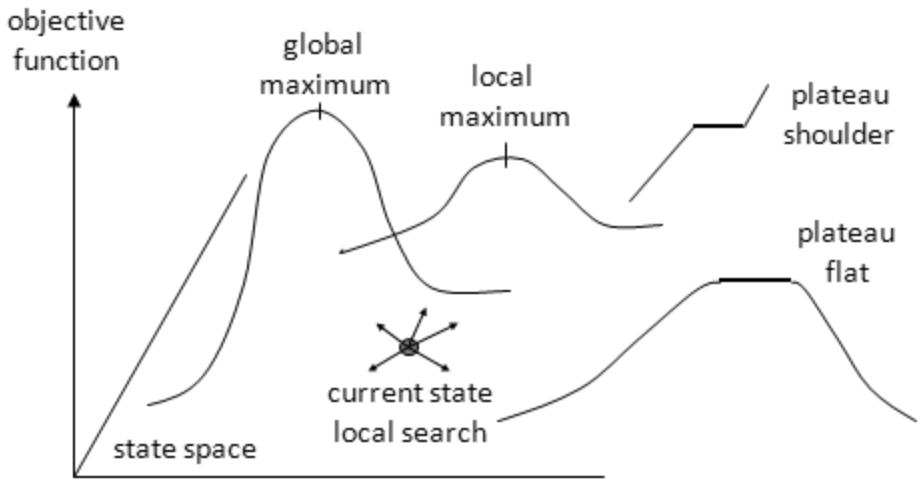


Figure 2.6 State Space Landscape

Hill Climbing

- The Hill-climbing search algorithm is a loop that continually moves in the direction of increasing value.
- The algorithm only records the state and its evaluation instead of maintaining a search tree. It takes a problem as an input, and it keeps comparing the values of the current and the next nodes.
- The next node is the highest-valued successor of the current node.
- If the value of the current node is greater than the next node, then the current node will be returned. Otherwise, it will go deeper to look at the next node of the next node.
- It 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.Hill climbing does not maintain a search tree, so the current node data structure need only record the state and its objective function value.
- Hill-climbing does not look ahead beyond the immediate neighbors of the current state.

The peaks are found on a surface of states where height is defined by Hill-climbing function.

```
function HILL-CLIMBING( problem) returns a solution state
  inputs: problem, a problem
  static: current, a node
          next, a node

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

Problems with hill-climbing

Hill-climbing often gets stuck for the following reasons.

i) Local maxima

ii) plateau

iii)Ridges

i) Local Maxima

Local maxima is a peak that is lower than the highest peak in the state space. When a local maxima is reached, the algorithm will halt even a solution has not been reached yet.

ii). Plateaux

A plateaux is an area of the state space where the neighbors are about the same height. In such a situation, a random walk will be generated.

iii). Ridges

A ridge may have steeply sloping sides towards the top, but the top only slopes gently towards a peak. In this case, the search makes little progress unless the top is directly reached, because it has to go back and forth from side to side.

It must be possible to encounter a situation that no further progress can be made from one certain starting point. If this happens, the Random restart hill-climbing is the obvious thing to do. As the name says, it randomly generates different starting points over again until it halts. It saves the best result found so far. And it can eventually find out the optimal solution if enough iteration are allowed.

As a matter of fact, and obviously, the fewer local maxima, the quicker it finds a good solution. But usually, a reasonably good solution can be found after a small number of iterations.

Variations of Hill Climbing

- Stochastic HC: chose randomly among the neighbors going uphill.
- First-choice HC: generate random successors until one is better. Good for states with high numbers of neighbors.
- Random restart: the sideway moves restart from a random state.
- Evolutionary hill-climbing: represents potential solutions as strings and performs random mutations Keeps the mutations that are better states. It's a particular case of first-choice and the ancestor of the genetic algorithms.

Simulated Annealing

- The simulated annealing takes some downhill steps to escape the local maxima, and it picks random moves instead of picking the best move.

- If the move actually improves the situation, it will keep executing the move. Otherwise, it will make the moves of a probability less than one.

- When the end of the searching is close, it starts behaving like hill-climbing.

- The word "annealing" is originally the process of cooling a liquid until it freezes. The Simulated-annealing function takes a problem and a schedule as inputs Here, schedule is a mapping determining how fast the temperature should be lowered.

- Again, the algorithm keeps comparing the values of the current and the next nodes, but here, the next node is a randomly selected successor of the current node.

- It also maintains a local variable T which is the temperature controlling the probability of downward steps.

- By subtracting the values of the current node from the next node to obtained the difference Delta-E, the algorithm can determine the probability of the next move. If Delta-E is greater than zero, then the next node will be looked at.

- Otherwise, the probability for the next node to be looked at is e to the power Delta-E over T.

- In other word, the variable Delta-E is actually the amount by which the evaluation is worsened

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  static: current, a node
          next, a node
          T, a "temperature" controlling the probability 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^(ΔE/T)
```

Applications of Simulated Annealing:

- Traveling sales man problem.
- VLSI Design.
- Production Scheduling.
- Timetable problem.
- Image Processing.

Local beam search

Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations. The local beam search algorithm 10 keeps track of *k* states rather than just one. It begins with *k* randomly generated states. At each step, all the successors of all *k* states are generated. If anyone is a goal, the algorithm halts. Otherwise, it selects the *k* best successors from the complete list and repeats.

A local beam search with *k* states runs *k* random restarts in parallel instead of in sequence. In a random-restart search, each search process runs independently of the others. In a local beam search, useful information is passed among the k parallel search threads.

Stochastic Beam Search

This search helps to avoid the above problem. Instead of choosing the best k from the pool of successor stochastic beam search chooses k successors at random, with the probability that choosing a successor is an increasing function of its value**.**

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, GA 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-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires 8 x $\log_2$ 8 = 24 bits.

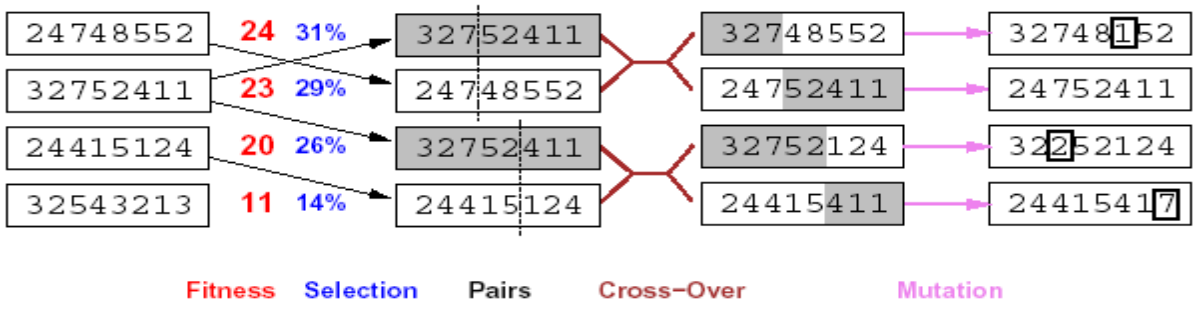


Figure 2.7 Genetic Algorithm

Fitness function : In figure 2.7 each state is rated by the evaluation function or the **fitness function**. This function should return higher values for better states so, for 8 queens pbm, we use number of non attacking pairs of queens, which has a value of 28 for a solution. The value of the four states are 24,23,20,22.

Selection : In Figure 2.7 ,a random choice of two pairs is selected for reproduction, in accordance with the probabilities in figure 2.7. One individual is selected twice and not at all once.

Cross over : For each pair to be mated, a crossover point is randomly chosen from the positions in the string. In Figure 2.7 the crossover points are after the third digit in the first pair and after the fifth digit in the second pair. The first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent.

Mutation : Each location is subject to random **mutation** with a small independent probability. One digit was mutated in the first, third, and fourth offspring. In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column.

Advantages of GA

Advantage comes from the ability of crossover to combine large blocks of letters that have evolved independently to perform useful functions, thus raising the level of granularity at which the search operates.

Schema

Substring in which some of the positions can be left unspecified is called Schema.

Strings that match the schema (such as 24613578) are called instances of the schema.

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
inputs: population, a set of individuals
FITNESS-FN, a function that measures the fitness of an individual
repeat
new.populatioti +- 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 .populaiion
until some individual is fit enough, or enough time has elapsed
return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(X, y) returns an individual
inputs: x, y, parent individuals
n +- LENGTH(X)
c +- random number from 1 to n
return ApPEND(SUBSTRING(X,1, c), SUBSTRING(y, c + 1, n))
```

Constraint Specification Problem

- 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_i of possible values. Each constraint C_i involves some subset of the 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 so- lution to a CSP is a complete assignment that satisfies all the constraints.
- Some CSPs also require a solution that maximizes an objective function.

Suppose that, having tired of Romania, we are looking at a map of Australia showing each of its states and territories, and that we are given the task of coloring each region either red, green, or blue in such a way that no neighboring regions have the same color. To formulate this as a CSP, we define the variables to be the regions shown in figure 2.8: 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)}. 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.

The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color. The map coloring problem represented as a constraint graph.

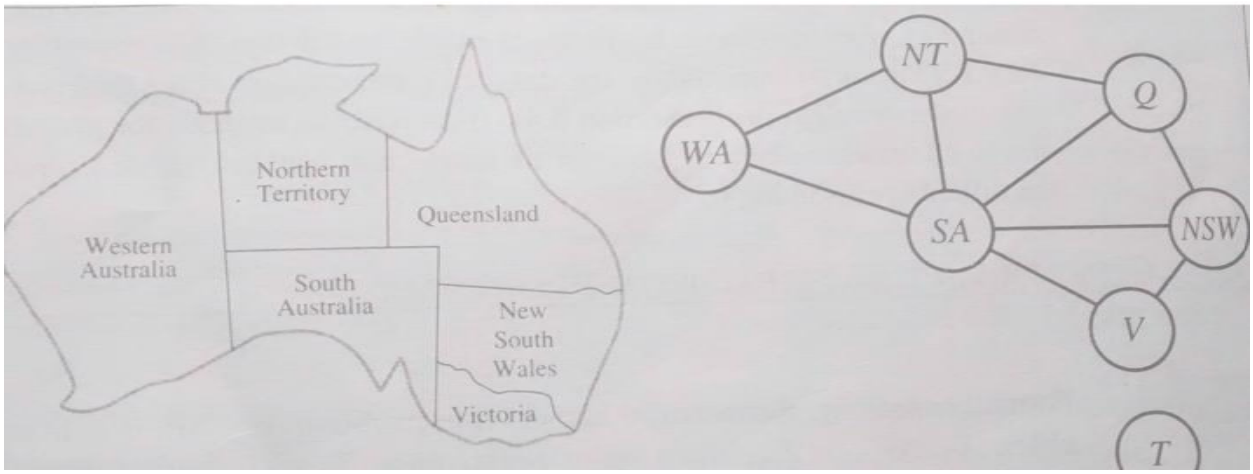


Figure 2.8 Road Map

CSP can be given an incremental formulation as a standard search problem as follows:

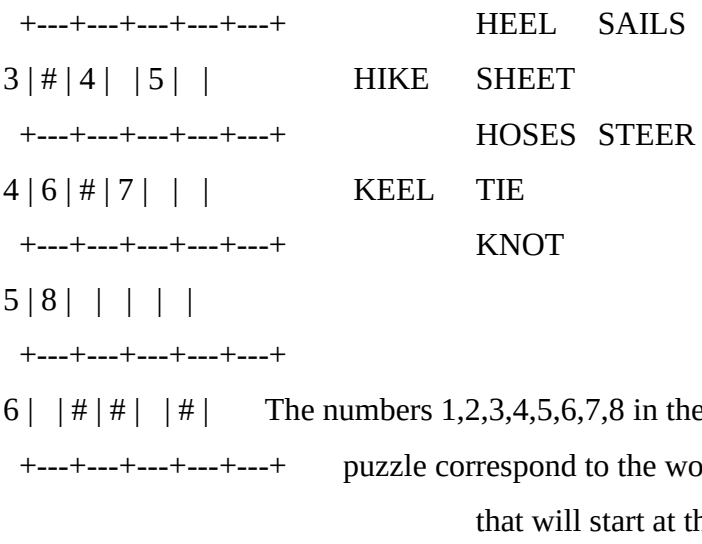
- **Initial state:** The empty assignment{}, in which all variables are unassigned.
- **Successor Function:** A value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.
- **Goal Test:** The current assignment is complete.
- **Path Cost:** A constant cost for every step.

Examples of constraint satisfaction problems:

Example 1: The n-Queen problem: The local condition is that no two queens attack each other, i.e. are on the same row, or column, or diagonal.

Example 2: A crossword puzzle: We are to complete the puzzle

```
   1   2   3   4   5
 +---+---+---+---+---+       Given the list of words:
1|1|  |2|  |3|              AFT      LASER
 +---+---+---+---+---+              ALE      LEE
2|#|#|  |#|  |              EEL      LINE
```



Example 3: A cryptography problem: In the following pattern

```
    S E N D
    M O R E
 ========
M O N E Y
```

We have to replace each letter by a distinct digit so that the resulting sum is correct.

Example 4: A map coloring problem: We are given a map, i.e. a planar graph, and we are told to color it using three colors, green, red, and blue, so that no two neighboring countries have the same color.

All these examples are instances of the same pattern, captured by the following definition:

A Constraint Satisfaction Problem is characterized by:

- a set of variables {x1, x2, .., xn},
- for each variable xi a domain Di with the possible values for that variable, and
- a set of constraints, i.e. relations, that are assumed to hold between the values of the variables. [These relations can be given intentionally, i.e. as a formula, or extensionally, i.e. as a set, or procedurally, i.e. with an appropriate generating or recognising function.] We will only consider constraints involving one or two variables.

The constraint satisfaction problem is to find, for each i from 1 to n, a value in Di for xi so that all constraints are satisfied.

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.

ALGORITHM

```
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
```

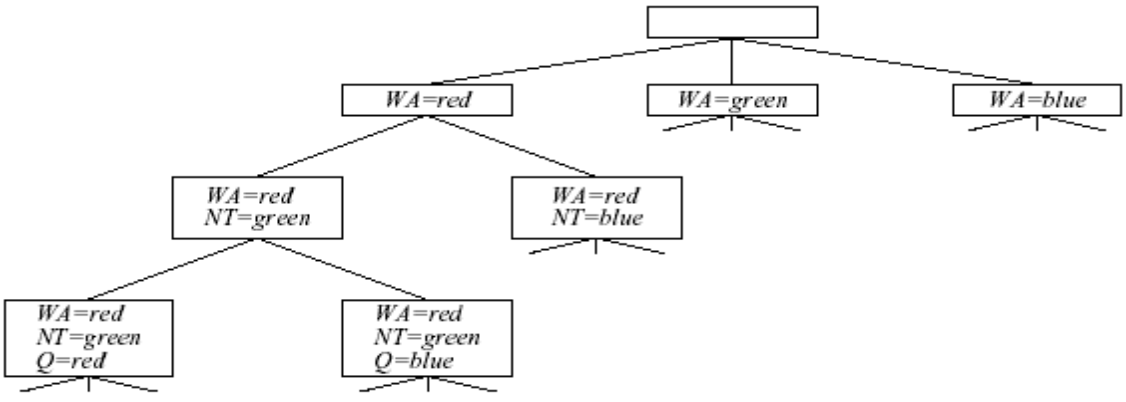


Figure 2.9 Simple backtracking for map colouring

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.

	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 2.10 Map coloring with forward checking

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

It provides a fast method of constraint propagation that is substantially stronger than forward checking. Here arc refers to a directed arc in the constraint graph. Such as the arc from SA to NSW. Given the current domains of SA and NSW,the arc is consistent if, for *every* value x of SA, there is *some* value y of NSW that is consistent with x. In the third row the current

domains of SA and NSW are fblueg and fred; blueg respectively. For SA=blue, there is a consistent assignment for NSW, namely, NSW =red; therefore, the arc from SA to NSW is consistent. On the other hand, the reverse arc from NSW to SA is not consistent: for the assignment NSW =blue, there is no consistent assignment for SA. The arc can be made consistent by deleting the value blue from the domain of NSW.

We can also apply arc consistency to the arc from SA to NT at the same stage in the search process. The third row of the table shows that both variables have the domain fblueg. The result is that blue must be deleted from the domain of SA, leaving the domain empty. Thus, applying arc consistency has resulted in early detection of an inconsistency that is not detected by pure forward checking.

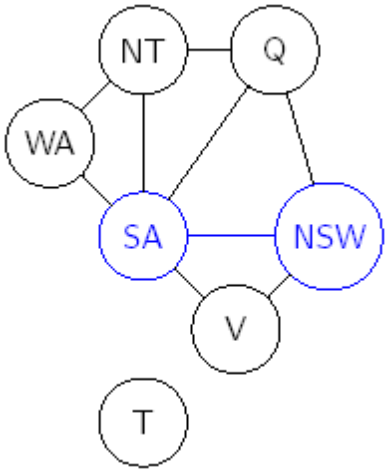


Figure 2.11 Arc consistency

function AC-3(csp) **returns** the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables fX1; X2; : : : ; Xng
local variables: queue, a queue of arcs, initially all the arcs in csp
while queue is not empty **do**
(Xi; Xj) REMOVE-FIRST(queue)
if REMOVE-INCONSISTENT-VALUES(Xi; Xj) **then**
for each Xk **in** NEIGHBORS[Xi] **do**
add (Xk; Xi) to queue
function REMOVE-INCONSISTENT-VALUES(Xi; Xj) **returns** true iff we remove a value
removed false

for each x **in** DOMAIN[Xi] **do**
if no value y in DOMAIN[Xj] allows (x ,y) to satisfy the constraint between Xi and Xj
then delete x from Domain[Xi]
return removed

Intelligent backtracking: looking backward

The BACKTRACKING-SEARCH algorithm has a very simple policy for what to do when a branch of the search fails: back up to the preceding variable and try a different value for it. This is called **chronological backtracking**, because the *most recent* decision point is revisited. In this subsection, we will see that there are much better ways.

Consider what happens when we apply simple backtracking with a fixed variable ordering Q, NSW, V , T, SA, WA, NT. Suppose we have generated the partial assignment fQ=red;NSW =green; V =blue; T =redg. When we try the next variable, SA, we see that every value violates a constraint. We back up to T and try a new color for Tasmania! Obviously this is silly—recoloring Tasmania cannot resolve the problem with South Australia.

A more intelligent approach to backtracking is to go all the way back to one of the CONFLICT SET set of variables that *caused the failure*. This set is called the **conflict set**; here, the conflict set for SA is fQ;NSW; V g. In general, the conflict set for variable X is the set of previously assigned variables that are connected to X by constraints. The **backjumping** method backtracks to the *most recent* variable in the conflict set

LOCAL SEARCH FOR CSPS

- Local search using the min-conflicts heuristic has been applied to constraint satisfaction problems with great success. They use a complete-state formulation: the initial state assigns a value to every variable, and the successor function usually works by changing the value of one variable at a time.
- In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variables—the min-conflicts heuristic.

- Min-conflicts is surprisingly effective for many CSPs, particularly when given a reasonable initial state. Amazingly, on then-queens problem, if you don‘t count the initial placement of queens, the runtime of minconflicts is roughly independent of problem size

```
function MIN-CONFLICTS(csp, max steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
    max steps, the number of steps allowed before giving up
current an initial complete assignment for csp

for i = 1 to max steps do

      if current is a solution for csp then return current
      var a randomly chosen, conflicted variable from VARIABLES[csp]
      value the value v for var that minimizes CONFLICTS(var, v, current, csp)
      set var =value in current

return failure
```

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 decomposotion into subproblems.

The algorithm has the following steps:

1. Choose any variable as the root of the tree, and order the variables from the root to the leaves in such a way that every node’s parent in the tree precedes it in the ordering. (See Figure 2.12(b).) Label the variables X1; : : : ;Xn in order. Now, every variable except the root has exactly one parent variable.

2. For j from n down to 2, apply arc consistency to the arc (Xi;Xj), where Xi is the parent of Xj , removing values from DOMAIN[Xi] as necessary.

3. For j from 1 to n, assign any value for Xj consistent with the value assigned for Xi,

- where Xi is the parent of Xj .

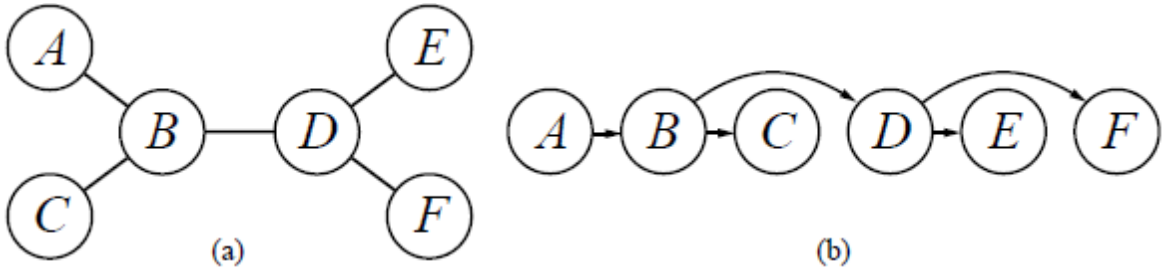


Figure 2.12 Constraint graph of a tree structured CSP

There are two key points to note. First, after step 2 the CSP is directionally arc-consistent,so the assignment of values in step 3 requires no backtracking. Second, by applying the arc-consistency checks in reverse order in step 2, the algorithm ensures that any deleted values cannot endanger the consistency of arcs that have been processed already. The complete algorithm runs in time $O(nd^2)$.

Tree decomposition

Each subproblem is solved independently, and the resulting solutions are then combined. Like most divide-and-conquer algorithms, this works well if no subproblem is too large. Figure 2.13 shows a tree decomposition of the mapcoloring problem into five subproblems. A tree decomposition must satisfy the following three requirements:

_ Every variable in the original problem appears in at least one of the subproblems.

_ If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.

_ If a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems.

The first two conditions ensure that all the variables and constraints are represented in the decomposition. The third condition seems rather technical, but simply reflects the constraint that any given variable must have the same value in every subproblem in which it appears; the links

joining subproblems in the tree enforce this constraint. For example, SA appears in all four of the connected subproblems in Figure 2.13

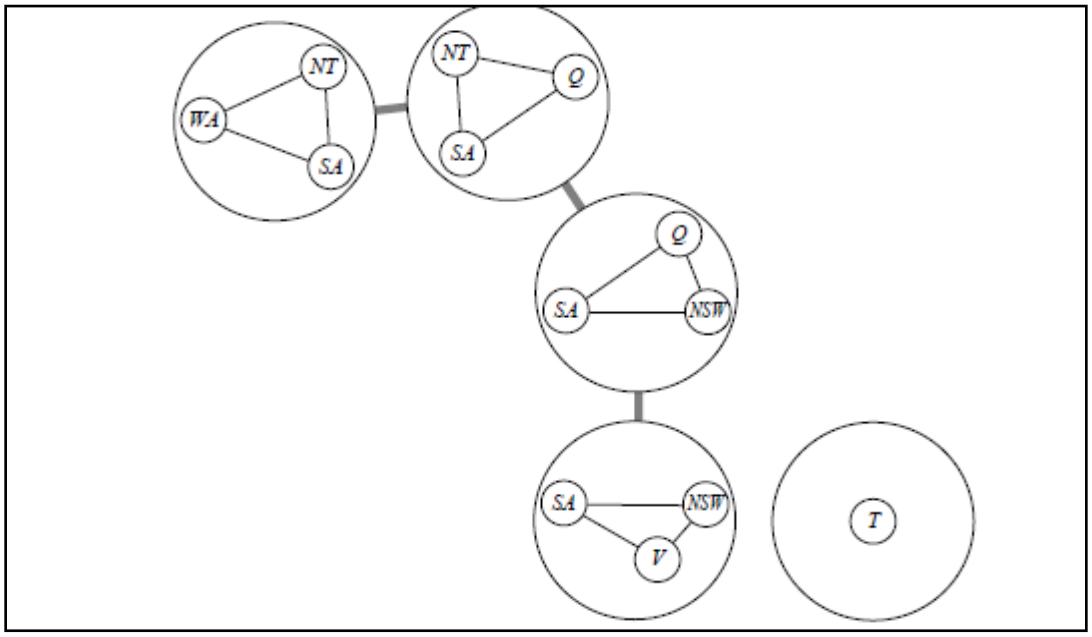


Figure 2.13 Tree Decomposition

ADVERSARIAL SEARCH

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

GAMES

- Mathematical Game Theory, a branch of economics, views any multi agent 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.

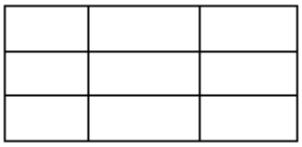
Optimal Decisions in Game

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. His 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 shows the part of the game tree for tic-tac-toe (nougats 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.

x	0	x
	0	x
	0	

Won by MIN

x	0	x
0	0	x
x	x	0

Draw between MIN and MAX

x	0	x
	x	x
x	0	0

Won by MAX

- The 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. The below figure 2.14 show about the search tree for the game of tic-tac-toe.

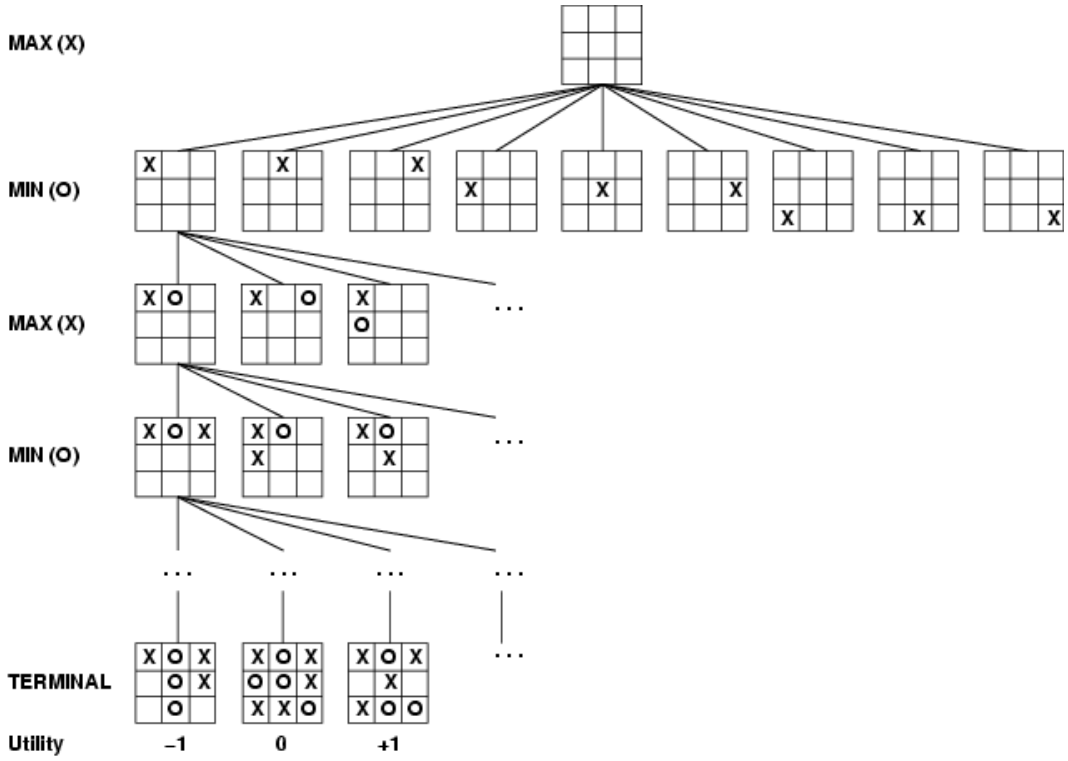


Figure 2.14 search tree for the game of tic-tac-toe.

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.

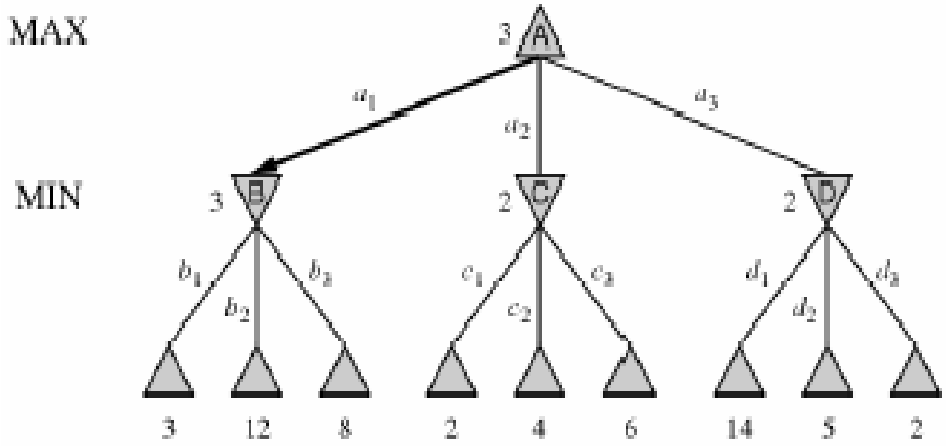


Figure 2.15 Two ply game tree

Figure 2.15 shows A Two ply game tree. The nodes are "MAX nodes", in which it is MAX'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.

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
```

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

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

The minimax algorithm 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 the previous fig ,the algorithm first recourses down to the three bottom left nodes, and uses the utility 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.

ALPHA-BETA PRUNING

Pruning: The process of eliminating a branch of the search tree from consideration without examining is called pruning. The two parameters of pruning technique are:

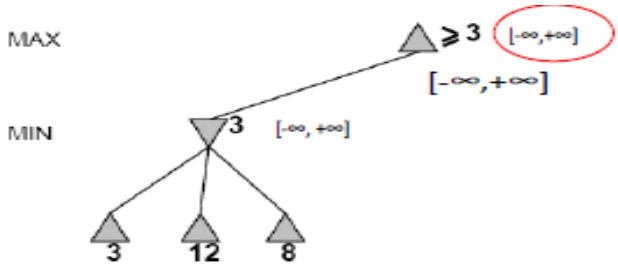
1.**Alpha (α)**: Best choice for the value of MAX along the path or lower bound on the value that on maximizing node may be ultimately assigned.

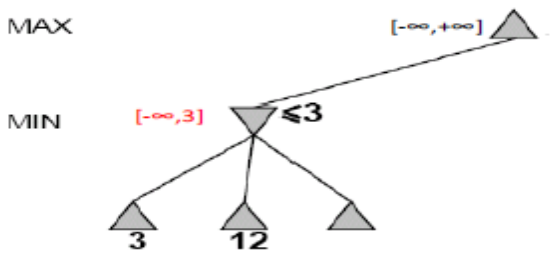
2.**Beta (β):** Best choice for the value of MIN along the path or upper bound on the value that a minimizing node may be ultimately assigned.

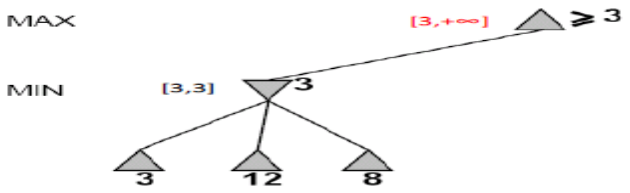
Alpha-Beta Pruning: The alpha and beta values are applied to a minimax tree, it returns the same move as minimax, but prunes away branches that cannot possibly influence the final decision is called Alpha-Beta pruning or Cutoff.

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.

Consider the two ply game tree from figure 2.16 . The different Stage of the calculation for optimal decision for the game tree.

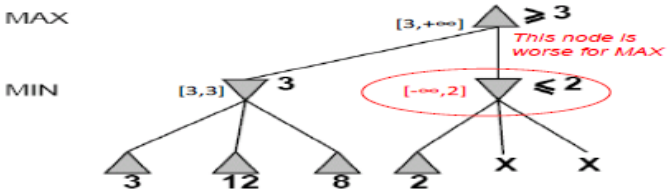


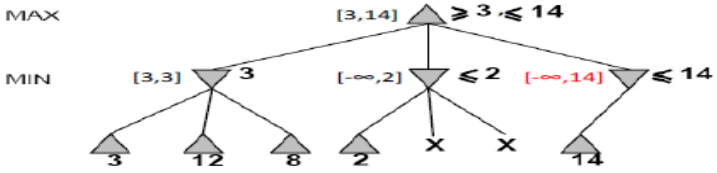




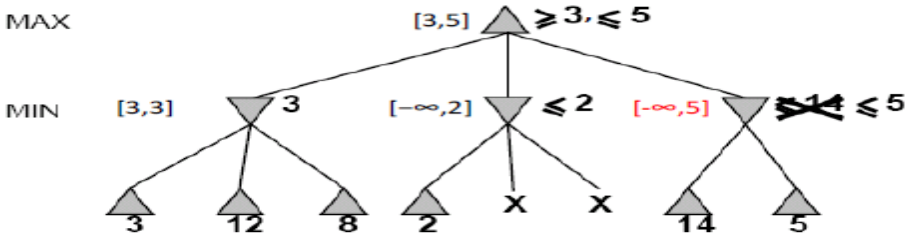
avks

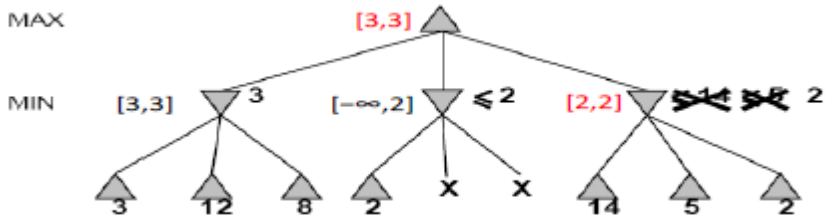
Alpha-Beta Example (continued)





avks 28





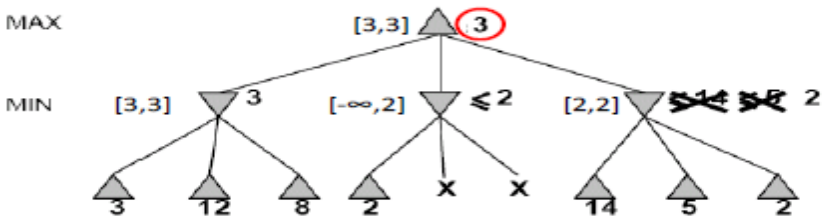


Figure 2.16 Alpha Beta Pruning

Stages in the calculation of the optimal decision for the game tree in Figure 2.16. At each point, we show the range of possible values for each node.

(a) The first leaf below *B* has the value 3. Hence, *B,* which is a MIN node, has a value of *at most* 3.

(b) The second leaf below *B* has a value of 12; MIN would avoid this move, so the value of *B* is still at most 3.

(c) The third leaf below *B* has a value of 8; we have seen all *B's* successors, so the value of *B* is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root.

(d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that *B* is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successors of C. This is an example of alpha-beta pruning.

(e) The first leaf below *D* has the value 14, so *D* is worth *at most* 14. This is still higher than MAX'S best alternative (i.e., 3), so we need to keep exploring *D's* successors. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14.

(f) The second successor of *D* is worth 5, so again we need to keep exploring. The third successor is worth 2, so now *D* is worth exactly 2. MAX'S decision at the root is to move to *B,* giving a value of 3

Consider a node *n* in the tree ---

• If player has a better choice at:

– Parent node of n

– Or any choice point further up

• Then *n* will never be reached in play.

• Hence, when that much is known about *n*, it can be pruned.

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.

Alpha –beta pruning can be applied to trees of any depth, and it is often possible to prune entire sub trees rather than just leaves.

```
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
```

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 at least 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 non terminal 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.

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.

Evaluation function Design :-

(i) The evaluation function should order the *terminal* states in the same way as the true utility function;

(ii)The computation must not take too long.

(iii) For non terminal states, the evaluation function should be strongly correlated with the actual chances of winning.

Example:

In chess, each material has its own value, that is called Material Value (i.e. each pawn is worth 1, a knight or bishop is worth 3, a rook 5, and the queen 9). Other features such as "good pawn structure" and "king safety" might be worth half a pawn, say. These feature values are then simply added up to obtain the evaluation of the position. A secure advantage equivalent to a pawn gives a substantial likelihood of winning, and a secure advantage equivalent to three pawns should give almost certain victory.

Mathematically, this kind of evaluation function is called a **weighted linear function,** because it can be expressed as *n*

EVAL(s) = *wlh(s) + w2h(s) + ... + wnfn(',) =* $\sum_{i=1}^{n} wifi(s)$,

where each *Wi* is a weight and each *fi* is a feature of the position. For chess, the *fi* could be the numbers of each kind of piece on the board, and the *Wi* could be the values of the pieces

Cutting off search

To perform a cut-off test, an evaluation function should be applied to positions that are quiescent – unlikely to exhibit wild swings in value in the search tree.

The next step is to modify ALPHA-BETA-SEARCH so that it will call the heuristic EVAL function when it is appropriate to cut off the search.

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

We also must arrange for some bookkeeping so that the current *depth* is incremented on each recursive call. The most straightforward approach to controlling the amount of search is to set a fixed depth limit, so that CUTOFF-*TEST(state, depth)* returns *true* for all *depth* greater than some fixed depth *d.* (It must also return *true* for all terminal states, just as TERMINAL-TEST did.) The depth *d* is chosen so that the amount of time used will not exceed what the rules of the game allow.

QUIESCENCE :

It is restricted to consider only certain types of moves, such as capture moves, that will quickly resolve the uncertainties in the position.

HORIZON EFFECT:

It arises when the program is facing a move by the opponent that causes serious damage and is ultimately unavoidable.

Over the Search Horizon :

Move to a place where it cannot be detected.

Singular Extension :

It will avoid the horizon effect without adding too much search cost. It is a move that is clear than all other moves in a given position. It‘s branching factor is 1.

Forward Pruning :

It means some moves at a given node are pruned immediately without further consideration.

Games that include an element of 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. For example, white has rolled a 6-5, and has four possible moves shown in figure 2.17 . White knows what his or her own legal moves are, White doesnot know what black is going to roll and thus does not know what balck legal moves will be.

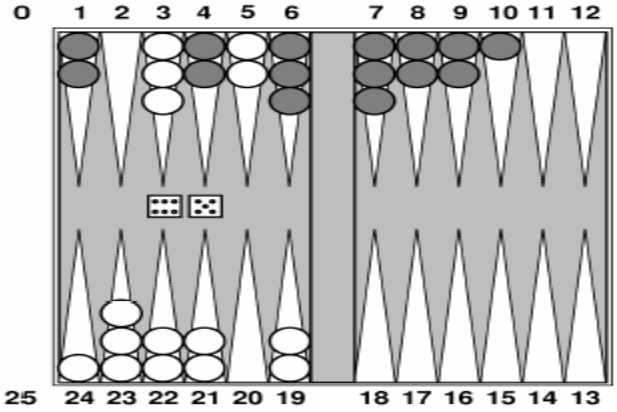


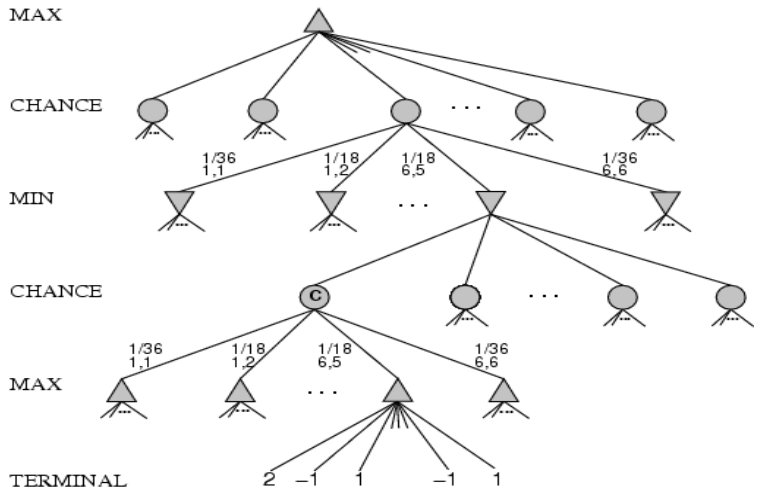
Figure 2.17 Backgammon Position

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)

A game tree in backgammon must include chance nodes in addition to MAX and MIN nodes. In the below figure chance nodes are shown as circles. The branches leading from each chance node denotes the possible dice rolls, and each is labeled with the roll and the chance node denote the possible dice rolls.



Expected minimax value

EXPECTED-MINIMAX-VALUE(n)=

UTILITY(n) If n is a terminal

$max_s \in$ successors(n) MINIMAX-VALUE(s) If n is a max node

$min_s \in$ 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.

Position evaluation in games with chance nodes

- Due to the presence of chance nodes means that one has to be more careful about what the evaluation values mean.
- The program behaves totally differently if there is a change in the scale of some evaluation values.
- The evaluation function must be a *positive linear* transformation of the probability of winning from a position

Complexity of Expectiminimax

If the program knew in advance all the dice rolls that would occur for the rest of the game, solving a game with dice would be just like solving a game without dice, which minimax does in *O(bm)* time. Because expectiminimax is also considering all the possible dice-roll sequences, it will take 0 *(bmnm),* where *ti* is the number of distinct rolls, d is the depth. Card game is also an example that takes an element of chance.

SCSX1021 ARTIFICIAL INTELLIGENCE

Unit:III

KNOWLEDGE AND REASONING

Logical Agents

The **representation** of knowledge and the **reasoning** processes that bring knowledge to life are central to the entire field of artificial intelligence.

• Knowledge and reasoning are enable successful behaviors that would be very hard to achieve.

• Knowledge and reasoning also play a crucial role in dealing with **partially observable environments.**

• A knowledge-based agent can combine general knowledge with current percepts to infer hidden aspects of the current state prior to selecting actions.

KNOWLEDGE BASED AGENTS

• The central component of a knowledge-based agent is its knowledge base, or KB.

• A knowledge base is a set of **sentences**.

• **Knowledge Representation Language:-**Each sentence is expressed in a language called a Knowledge Representation Language and represents some assertion about the world.

• **Inferenc**e :- There is a way to add new sentences to the knowledge base is **TELL** and to query what is know is **ASK**.

TELL :- A way to add new sentences to the knowledge base

ASK :- A way to query what is known.

Both tasks may involve **INFERENCE** that is, deriving new sentences from old.

In LOGICAL AGENTS, inference must obey the fundamental requirement that when one ASKS a question of the knowledge base, the answer should follow from what has been told (or rather, TELLED) to the knowledge base previously.

Knowledge Based Agent Program

```
function KB-AGEN T(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(t))
        TELL(KB, MAKE-ACTION-SENTENCE(action,t))
        T ←t+1
return action
```

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.
  - Third, the agent records its choice with TELL and executes the action. The TELL is necessary to let the knowledge base know that the hypothetical action has actually been executed.

The details of the Representation Language are hidden inside three functions that implement the interface between the sensors and actuators and the core representation and reasoning system.

- MAKE-PERCEPT-SENTENCE takes a percept and a time and returns a sentence asserting that the agent perceived the given percept at the given time.
- MAKE-ACTION-QUERY takes a time as input and returns a sentence that asks what action should be done at the current time.

• MAKE-ACTION-SENTENCE constructs a sentence asserting that the chosen action was executed. The details of the inference mechanisms are hidden inside TELL and ASK.

Knowledge level

The knowledge-based agent is not an arbitrary program for calculating actions. It is amenable to a description at the knowledge level, where we need specify only what the agent knows and what its goals are, in order to fix its behavior.

Implementation level

For example : an automated taxi might have the goal of dropping a passenger to Marin County and might know that it is in San Francisco and that the Golden Gate Bridge is the only link between the two locations. Then we can expect it to cross the Golden Gate Bridge because it knows that that will achieve its goal. This analysis is independent of how the taxi works at the implementation level. It doesn't matter whether its geographical knowledge is implemented as linked lists or pixel maps, or whether it reasons by manipulating strings of symbols stored in registers or by propagating noisy signals in a network of neurons.

• One can build a knowledge-based agent simply by Telling it what it needs to know.

• The agent's initial program, before it starts to receive percepts, is built by adding one by one the sentences that represent the designer's knowledge of the environment.

• Designing the representation language to make it easy to express this knowledge in the form of sentences simplifies the construction problem enormously. This is called the **declarative approach** to system building.

• The **procedural approach** encodes desired behaviors directly as program code; minimizing the role of explicit representation and reasoning can result in a much more efficient system

The Wumpus world environment

The **wumpus world** is a cave consisting of rooms connected by passageways.

• Lurking somewhere in the cave is the wumpus, a beast that eats anyone who enters its room.

• The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in).

• The only mitigating feature of living in this environment is the possibility of finding a heap of gold.

• Although the wumpus world is rather tame by modern computer game standards, it makes an excellent test bed environment for intelligent agents.

• Michael Genesereth was the first to suggest this.

A Sample Wumpus world is shown in the figure 3.1.

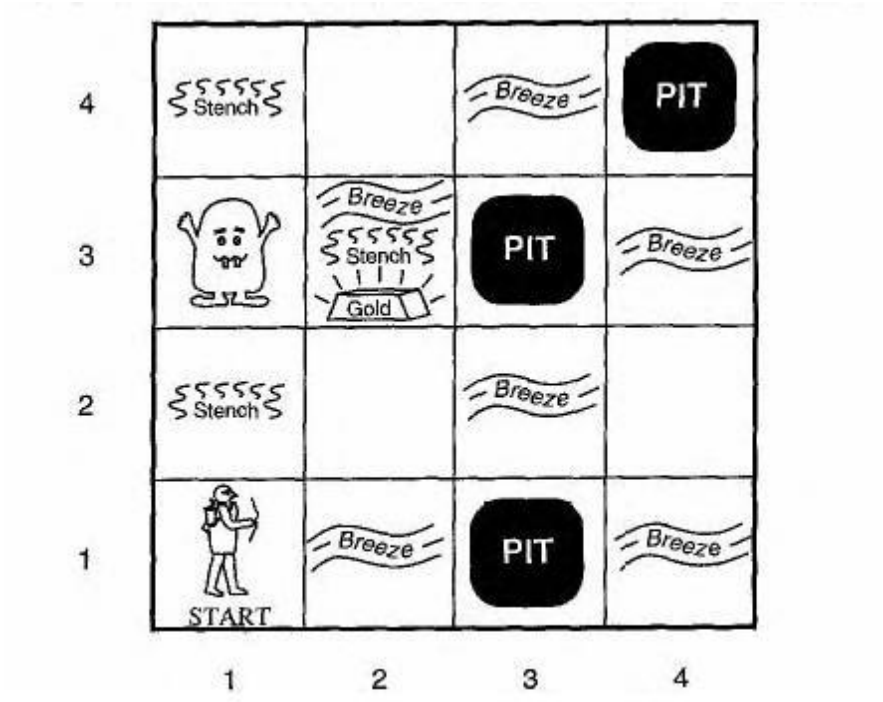


Figure 3.1 Wumpus World

The precise definition of the task environment is given by the PEAS description:

• **Performance measure: +I000 for picking up the gold, -1000 for falling into a pit or** being eaten by the wumpus, -1 for each action taken and -10 for using up the arrow.

• **Environment: A 4 x 4 grid of rooms. The agent always starts in the square labeled** [1,1], facing to the right. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.

• **Actuators:** The agent can move forward, turn left by 90°, or turn right by 90°. The agent dies a miserable death if it enters a square containing a pit or a live wumpus. Moving forward has no effect if there is a wall in front of the agent. The action Grab can be used to pick up an object that

is in the same square as the agent. The action Shoot can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits (and hence kills) the wumpus or hits a wall. The agent only has one arrow, so only the first Shoot action has any effect.

• **Sensors:** The agent has five sensors, each of which gives a single bit of information.

1. In the square containing the wumpus and in the directly (not diagonally) adjacent squares the agent will perceive a stench.
2. In the squares directly adjacent to a pit, the agent will perceive a breeze.
3. In the square where the gold is, the agent will perceive a glitter.
4. When an agent walks into a wall, it will perceive a bump.
5. When the wumpus is killed, it emits a woeful scream that can be perceived anywhere in the cave.

The percepts will be given to the agent in the form of a list of five symbols; for example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent will receive the

percept [Stench, Breeze, None, None, None].

In most instances of the wumpus world, it is possible for the agent to retrieve the gold safely. Occasionally, the agent must choose between going home empty-handed and risking death to find the gold. About 21% of the environments are utterly unfair, because the gold is in a pit or surrounded by pits.

The agent's initial knowledge base contains the rules of the environment, as listed above; in particular, it knows that it is in [l,l] and that [l,l] is a safe square.

• The first percept is [None, None, None, None, None], from which the agent can conclude that its neighboring squares are safe. Figure 3.2(a) shows the agent's state of knowledge at this point. We list the sentences in the kowledge base using letters such as B (breezy) and OK (safe, neither pit nor wumpus) marked in the appropriate squares.

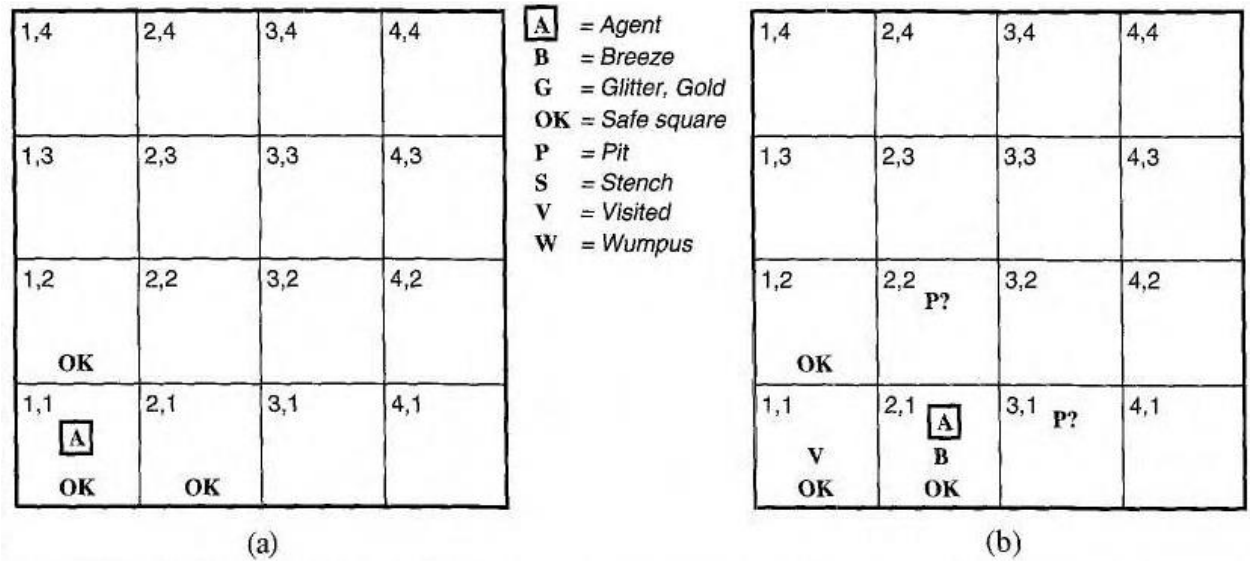


Figure 3.2 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [None, None, None, None, None]. (b) After one move, with percept [None, Breeze, None, None, None].

From the fact that there was no stench or breeze in [1,1], the agent can infer that [1,2] and [2,1] are free of dangers. They are marked with an OK to indicate this. A cautious agent will move only into a square that it knows is OK. Let us, suppose the agent decides to move forward to [2,1], giving the scene in Figure 3.2(b).

• The agent detects a breeze in [2,1], so there must be a pit in a neighboring square. The pit cannot be in [1, 1], by the rules of the game, so there must be a pit in [2,2] or [3,1] or both. The notation P? in Figure 3.2(b) indicates a possible pit in those squares. At this point, there is only one known square that is OK and has not been visited yet. So the prudent agent will turn around, go back to [1, 1] and then proceed to [1,2].

• The new percept in [1,2] is [Stench, None, None, None, None], resulting in the state of knowledge shown in Figure 3.2(a). The stench in [1,2] means that there must be a wumpus nearby. But the wumpus cannot be in [1,1], by the rules of the game, and it cannot be in [2,2] (or the agent would have detected a stench when it was in [2,1]).

Therefore, the agent can infer that the wumpus is in [1,3]. The notation W! indicates this. Moreover, the lack of a Breeze in [1,2] implies that there is no pit in [2,2]. Yet we already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1]. This is a fairly difficult inference, because it combines knowledge gained at different times in different

places and relies on the lack of a percept to make one crucial step. The inference is beyond the abilities of most animals, but it is typical of the kind of reasoning that a logical agent does.

• The agent has now proved to itself that there is neither a pit nor a wumpus in [2,2], so it is OK to move there. We will not show the agent's state of knowledge at [2,2]; we just assume that the agent turns and moves to [2,3], giving us Figure 3.3(b). In [2,3], the agent detects a glitter, so it should grab the gold and thereby end the game.

• In each case where the agent draws a conclusion from the available information, that conclusion is guaranteed to be correct the available information is correct. This is a fundamental property of logical reasoning.

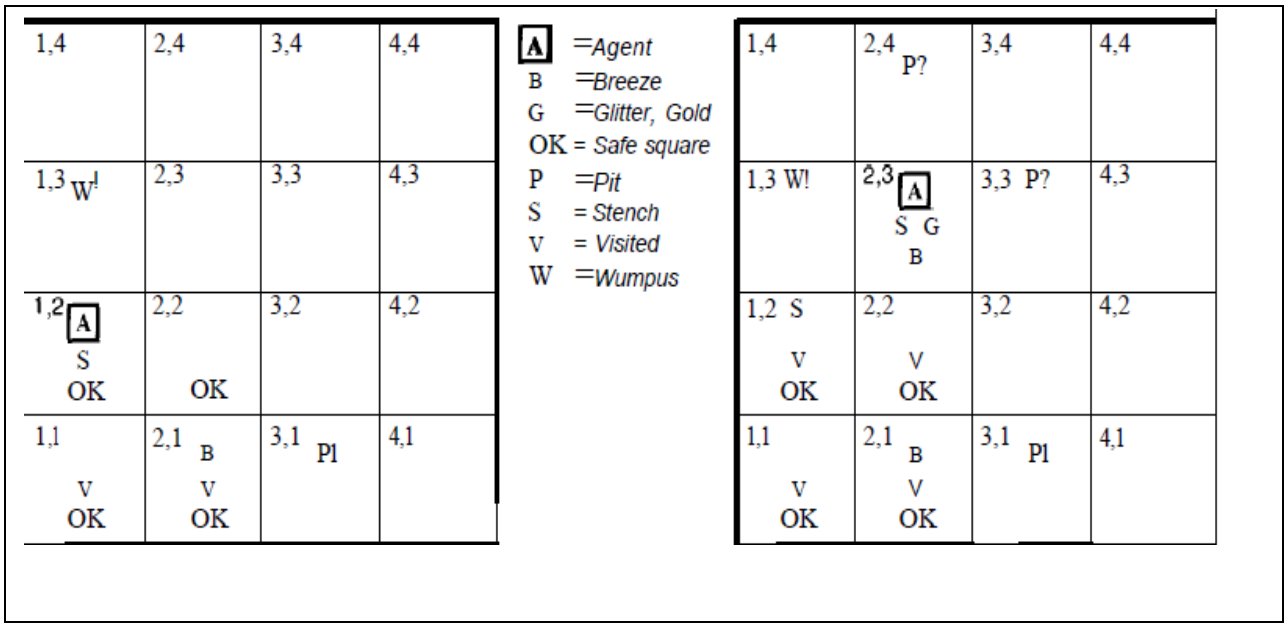


Fig 3.3 Two later stages in the progress of the agent. (a) After the third move, with percept [Stench, None, None, None, None]. (b) After the fifth move, with percept [Stench, Breeze, Glitter, None, None].

LOGIC

• Knowledge bases consists of sentences. These sentences are expressed according to the syntax of the representation language, which specifies all the sentences that are well formed.

• The syntax is clear enough in ordinary arithmetic:

"x + y = 4" is a well-formed sentence, whereas "x2y+ =" is not.

semantics

• A logic must also define the semantics of the language.

• Semantics means "meaning" of sentences. In logic, the definition is more precise.

• The semantics of the language defines the truth of each sentence with respect to each possible world.

• For example, "x + y = 4" is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1.1 .

In standard logics, every sentence must be either true or false in each possible world-there is no "in between.

Example : Assume x and y as the number of men and women sitting at a table playing bridge, for example, and the sentence x + y = 4 is true when there are four in total; formally, the possible models are just all possible assignments of numbers to the variables x and y. Each such assignment fixes the truth of any sentence of arithmetic whose variables are x and y.

Entailment

• Entailment means that one thing follows from another. Relation of logical entailment between sentences is involved that a sentence follows logical from another sentence.

• KB ⊨ α Knowledge base KB entails sentence α if and only if α is true in all worlds where KB is true

– E.g., the KB containing “the Giants won” and “the Reds won” entails “Either the Giants won or the Reds won”.

– E.g., x+y = 4 entails 4 = x+y.

– Entailment is a relationship between sentences (i.e., syntax) that is based on semantics.

Models

• Logicians typically think in terms of models, which are formally structured worlds with respect to which truth can be evaluated.

m is a model of a sentence α if α is true in m

M(α) is the set of all models of α

Then KB ⊨ α if M(KB) ⊆ M(α)

– E.g. KB=Giants won and Reds won α = Giants won

Consider the situation in Figure 3.2(b): the agent has detected nothing in [1,1] and a breeze in [2,1]. These percepts, combined with the agent's knowledge of the rules of the wumpus world, constitute the KB. The agent is interested (among other things) in whether the adjacent squares [1,2], [2,2], and [3,1] contain pits. Each of the three squares might or might not contain a pit, so (for the purposes of this example) there are $2^3 = 8$ possible models. These are shown in Figure 3.3.

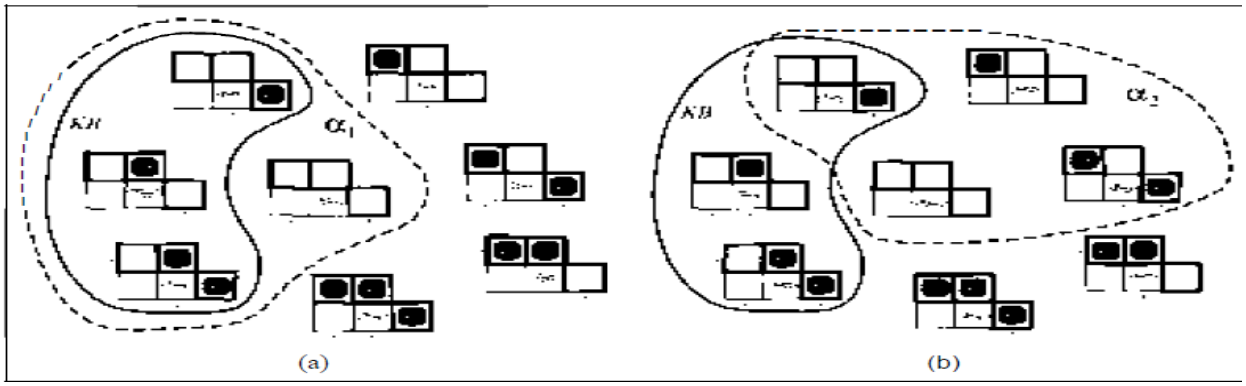


Figure 3.3 : Possible models for the presence of pits in squares [1,2], [2,2], and [3,1], given observations of nothing in [1,1] and a breeze in [2,1]. (a) Models of the knowledge base and αl (no pit in [1,2]). (b) Models of the knowledge base and α2 (no pit in [2,2]).

Conclusion

• The KB is false in models that contradict what the agent knows-for example, the KB is false in any model in which [l,2] contains a pit, because there is no breeze in [1,1].

• Now let us consider two possible conclusions:

• α1= "There is no pit in [1,2].“

• α2= "There is no pit in [2,2].“

• In every model in which KB is true, α1 is also true.

• Hence, KB ⊨ α1 : there is no pit in [1,2].

• In some models in which KB is true, α2 is false.

• Hence, KB ¥ α2 :so, the agent cannot conclude that there is no pit in [2,2].

Logical Inference : The above shown example illustrates entailment and also show how the definition of entailment can be applied to derived conclusions to carry out logical inference.

Model Checking : The inference algorithm illustrated in Figure 3.3 is called model checking, because it enumerates all possible models to check the α is true in all models in which KB is true.

If an inference algorithm i can derive α from KB, then

KB $\vdash_i \alpha$,

Pronunciation for above notation is :

"α is derived from KB by i" (or) " i derives α from KB".

Sound : An inference algorithm that derives only entailed sentences is called sound or truth preserving.

• i is sound if whenever KB $\vdash_i \alpha$, it is also true that KB $\models \alpha$

Soundness is a highly desirable property. An unsound inference procedure essentially makes things up as it goes along-it announces the discovery of nonexistent needles.

Complete: an inference algorithm is complete if it can derive any sentence that is entailed.

• i is complete if whenever KB $\models \alpha$, it is also true that KB $\vdash_i \alpha$

PROPOSITIONAL LOGIC

• Propositional logic is the simplest logic.

• The syntax of propositional logic and its semantics-the way in which the truth of sentences is determined.

Syntax -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.

Rules:

i. Uppercase names are used for symbols (ie) P,Q,R and so on.

ii. Names are Arbitrary

Complex Sentences : Complex sentences are constructed from simpler sentences using logical connections. There are five connectives.

- If S is a sentence, ¬S is a sentence (negation)
- If S1 and S2 are sentences, S1 ∧S2 is a sentence (conjunction)
- If S1 and S2 are sentences, S1 ∨S2 is a sentence (disjunction)
- If S1 and S2 are sentences, S1 ⇒ S2 is a sentence (implication)
- If S1 and S2 are sentences, S1 ⇔ S2 is a sentence (biconditional / IF AND ONLY IF)

¬ (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 $W_{1,3} /\backslash P_{3,1}$, is called a conjunction; its parts are the conjuncts. (The /\ looks like an "N' for "And.")

V (or). A sentence using V, such as $(W_{1,3}/\backslash P_{3,1}) VW_{2,2}$, is a disjunction of the Disjuncts $(W_{1,3} /\backslash P_{3,1})$ and $W_{2,2}$.

⇒ (implies). A sentence such as $(W_{1,3}/\backslash P_{3,1}) \Rightarrow \neg W_{2,2}$ is called an implication (or conditional). Its premise or antecedent is $(W_{1,3}/\backslash P_{3,1})$,and its conclusion or consequent is $W_{2,2}$. Implications are also known as rules or if-then statements.

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

BNF(Backus-Naur Form): The figure 3.4 shows about the BNF grammar of sentence in propositional logic.

Sentence	→	*Atomicbentence* | *ComplexSentence*
Atomicilentence	→	True | False | *Symbol*
Symbol	→	P | Q | R | .. ,
ComplexSentence	→	¬ *Sentence*
		(Sentence 1\ Sentence)
		(Sentence V Sentence)
		(Sentence ⇒ Sentence)
		(Sentence ⇔ Sentence)

Figure 3.4 A BNF (Backus-Naur Form) grammar of sentences in propositional logic.

Every sentence constructed with binary connectives must be enclosed in parentheses. This ensures that we have to write ((A /\ B) ⇒ C) instead of A /\ B ⇒C.

• Order of precedence for the connectives is similar to the precedence used in arithmetic.

For example, ab+ c is read as ((ab) + c) rather than a(b + c) because multiplication has higher precedence than addition.

The order of precedence in propositional logic

• The order of precedence in propositional logic is (from highest to lowest) ¬, /\, V, ⇒, ⇔.

• Hence, the sentence ¬P V Q /\ R⇒ S is equivalent to the sentence ((¬P) V (Q /\ R)) ⇒ S.

• Precedence does not resolve ambiguity in sentences such as A /\ B /\ C, which could be read as ((A /\ B) /\ C) or as (A /\ (B /\ C)). These two readings mean the same thing according to the semantics.

Semantics

• The semantics defines the rules for determining the truth of a sentence with respect to a particular model.

• In propositional logic, a model simply fixes the truth value, true or false for every proposition symbol.

• For example, if the sentences in the knowledge base make use of the proposition symbols $P_{l,2}$, $P_{2,2}$, and $P_{3,1}$, then one possible model is

$m1 = \{P_{1,2} = \text{false}, P_{2,2} = \text{false}, P_{3,1} = \text{true}\}$.

• With three proposition symbols, there are $2^3 = 8$ possible models.

• The semantics for propositional logic must specify how to compute the truth value of any sentence, given a model. This is done recursively. All sentences are constructed from atomic sentences and the five connectives; therefore, we need to specify how to compute the truth of atomic sentences and how to compute the truth of sentences formed with each of the five connectives.

Atomic sentences

Atomic sentences are easy:

• True is true in every model and False is false in every model.

• The truth value of every other proposition symbol must be specified directly in the model.

• For example, in the model m1 given earlier, $P_{1,2}$ is false.

Complex sentences

• For any sentence s and any model m, the sentence ¬s is true in m if and only if s is false in m.

• Such rules reduce the truth of a complex sentence to the truth of simpler sentences.

• The rules for each connective can be summarized in a truth table that specifies the truth value of a complex sentence for each possible assignment of truth values to its components.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
$false$	$false$	$true$	$false$	$false$	$true$	$true$
$false$	$true$	$true$	$false$	$true$	$true$	$false$
$true$	$false$	$false$	$false$	$true$	$false$	$false$
$true$	$true$	$false$	$true$	$true$	$true$	$true$

Figure 3.5 Truth tables for the logical connectives

Using the above figure 3.5, the truth value of any sentence S can be computed with respect to any model m by a simple process of recursive evaluation. Each model specifies true/false for each proposition symbol

E.g. $P_{1,2}$ $P_{2,2}$, $P_{3,1}$

false true false

With these symbols, 8 possible models, can be enumerated automatically.

Rules for evaluating truth with respect to a model *m*:

¬S is true if S is false

S1/\ S2 is true if S1 is true and S2 is true

S1 v S2 is true if S1is true or S2 is true

S1 ⇒ S2 is true if S1 is false or S2 is true

i.e., is false if S1 is true and S2 is false

S1 ⇔ S2 is true if S1⇒S2 is true and S2⇒S1 is true

Simple recursive process evaluates an arbitrary sentence, e.g.,

¬P1,2 ∧(P2,2)∨ (P3,1) = true∧(true V false) = true ∨ true = true

Confusion:

(i) *P*V*Q* is true when *P* is true or *Q* is true *or both.* There is a different connective called "exclusive or" ("xor" for short) that yields false when both disjuncts are true. There is no consensus on the symbol for exclusive or; two choices are

(ii) Any implication is true whenever its antecedent is false.

For example,

(a) "5 is even implies Sam is smart" is true, regardless of whether Sam is smart.

(b) *"P ⇒Q"* saying that "If *P* is true, then *Q* is true".

This sentence could be *false* if *P* is true but *Q* is false.

From the truth table, Bidirectional P ⇔ Q, it is true whenever both P ⇒ Q and Q ⇒ P are true (ie) "P if and only if Q" or "P if Q".

Example (Wumpus World)

A square is breezy *if* a neighboring square has a pit, and a square is breezy *only if* a neighboring square has a pit. So we need biconditionals such as

$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$ that $B_{1,1}$ means that there is a breeze in [1,1], $P_{1,2}$ means Pit in [1,2].

From the truth table, that the one-way implication

$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$ is true but incomplete. $B_{1,1}$, is false and $P_{1,1}$ is true, which violate the rules of the Wumpus world.

To solve this, implication requires the presence of pits if there is a breeze, whereas the biconditional also requires the absence of pits if there is no breeze.

A Simple knowledge base

We have defined the semantics for propositional logic, we can construct a knowledge base for the wumpus world.

Let $P_{i,j}$ be true if there is a pit in [i, j].

Let $B_{i,j}$ be true if there is a breeze in [i, j].

The knowledge base includes the following sentences, each one labeled for convenience:

• There is no pit in [l,l]:

R1: ¬P1,1

• A square is breezy if and only if there is a pit in a neighboring square. This has to be stated for each square; for now, we include just the relevant squares:

R2 : B1,1 ⇔ (P1,2 V P2,1)

R3: B2,1 ⇔ (P1,1 VP2,2 V P3,1)

• The preceding sentences are true in all wumpus worlds. Now we include the breeze percepts for the first two squares visited in the specific world the agent is in, leading up to the situation.

R4: ¬B1,1

R5: B2,1

The knowledge base, then, consists of sentences *R*1 through *R5*.

INFERENCE RULES

Logical inference is to decide whether $KB \models a$ a for some sentence *a*. Inference algorithm for will be a direct implementation of the definition of entailment: enumerate the models, and check that a is true in every model in which *KB* is true.

Example : Wumpus World

Propositional symbols for Wumpus are B1,1, *B2,1,* P1,1, *P1,2*, *P2,1, P2,2,* and *P3,1*. With the above symbols are 2^7 = 128 possible models. KB is true in any one of these three which is shown in the below Figure 3.6.

BI,I	$B_{2,1}$	PI,1	$P_{I,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_I	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

Figure 3.6 . A truth table constructed for the knowledge base given in the text.

From the above table, *KB* is true if *R1* through *R 5* are true, which occurs in just 3 of the 128 rows. In all 3 rows, *P1 ,2* is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

The algorithm for deciding entailment in propositional logic is shown below.

```
function TT-ENTAILS?(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  symbols ← a list of the proposition symbols in KB and α
  return TT-CHECK-ALL(KB, α, symbols, [])
```

```
function TT-CHECK-ALL(KB, α, symbols, model) returns true or false
  if EMPTY?(symbols) then
      ifPL-TRUE?(KB, model) then return PL-TRuE?(a, model)
      else return true
  else do
      P ← FIRST(symbols); rest ← REST(symbols)
      return TT-CHECK-ALL(KB, α, rest, EXTEND(P, true, model) and
             TT-CHECK-ALL(KB, α, rest, EXTEND(P, false, model)
```

Equivalence, validity, and satisfiability

Logical equivalence: two sentences α and β are logically equivalent if they are true in the same set of models. We write this as a $\alpha \Leftrightarrow \beta$.

Example : Two sentences α and β are logically equivalent if they are true in same models:

• $\alpha \equiv ß$ if and only if $\alpha \models \beta$ and $\beta \models \alpha$.

• Equivalence for any two sentences α and β is $\alpha \equiv ß$ if and only if $\alpha \models \beta$ and $\beta \models \alpha$

$$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha) \quad \text{commutativity of } \wedge$$
$$(\alpha \vee \beta) \equiv (\beta \vee \alpha) \quad \text{commutativity of } \vee$$
$$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma)) \quad \text{associativity of } \wedge$$
$$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma)) \quad \text{associativity of } \vee$$
$$\neg(\neg\alpha) \equiv \alpha \quad \text{double-negation elimination}$$
$$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha) \quad \text{contraposition}$$
$$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta) \quad \text{implication elimination}$$
$$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) \quad \text{biconditional elimination}$$
$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \quad \text{de Morgan}$$
$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \quad \text{de Morgan}$$
$$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \quad \text{distributivity of } \wedge \text{ over } \vee$$
$$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) \quad \text{distributivity of } \vee \text{ over } \wedge$$

The symbols *α,* β and γ stands for arbitrary sentences of propositional logic.

Validity

A sentence is valid if it is true in all models.

e.g., A V¬ A, A ⇒ A, (A ∧ (A ⇒ B)) ⇒ B. These are valid / true statements.

Valid sentences are also known as tautologies-they are *necessarily* true. Because the sentence *True* is true in all models, every valid sentence is logically equivalent to *True.*

For any sentences KB and α , *KB* ⊨ α if and only if (*KB* ⇒ α) is valid. Every valid implication sentence describes an inference.

Satisfiability

A sentence is satisfiable if it is true in *some* model.

For example, the knowledge base given earlier, *(R1 ∧ R2 ∧ R3 ∧ R4 ∧ R s),* is satisfiable because there are three models in which it is true.

If a sentence a is true in a model m, then we say that m satisfies α, or that m is a model of α. Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence.

FIRST-ORDER LOGIC

FOL, a representation language of knowledge which is powerful than propositional logic (ie)Boolean logic. FOL is an expressive.

- It follows Procedural approach
- It derive facts from other facts(ie.) dependent

First-order logic (like natural language) assumes the world contains

- Noun- refers to Objects. Eg.Name, place, thing.
- Verb refers to relations
- Some relations are functions
- Relation has only one value

Functions have many values assigned to it.

- Objects: people, houses, numbers, colors, baseball games, wars,
- Properties: red, round, prime, Small
- Relations: bigger than, part of, comes between, …
- Functions: father of, best friend, one more than, plus, …

Example :

1. “Evil King John ruled England in 1200”

   Objects: John, England, 1200; Relation: ruled; Properties: evil, king.

2. "One plus two equals three"

   Objects: one, two, three, one plus two; Relation: equals;

   Function: plus. ("One plus two" is a name for the object that is obtained by applying the function "plus" to the objects "one" and "two." Three is another name for this object.)

3. "Squares neighboring the wumpus are smelly."

   Objects: wumpus, squares; Property: smelly; Relation: neighboring.

Ontological Commitment

• The primary difference between propositional and first-order logic lies in the **ontological commitment** made by each language-that is, what it assumes about the **nature of reality**. Special Purpose logic makes faster Ontological commitment.

Temporal Logic: It assumes that facts hold at particular times and that those times (intervals) are ordered (arranged).

High order logic: It is more expressive than FOL. It allows one to make assertions about all relations.

Epistemological Commitments

A logic that allows the possible states of knowledge that it allows with respect to each fact. In first order logic, a sentence represents a fact and the agent believes the sentence to be true, believes it to be false, or has no opinion. These logics therefore have three possible states of knowledge regarding any sentence. Systems using probability theory, can have any *degree of belief*, ranging from 0 (total disbelief) to 1 (total belief).

Ontological Commitment and Epistemological Commitment of different logics:

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

SYNTAX AND SEMANTICS

Syntax → Procedures

Semantics →Meanings

a) Models for FOL

- Models for propositional logic are just sets of truth values for the proposition symbols (ie 0's and 1's).
- Models for first-order logic are represented in terms of objects and predicates on objects (ie) properties of objects (or) relation between objects.
- The **domain** of a model is the set of objects it contains. These objects are sometimes called **domain elements**.
- **Relation** → related set of tuples of objects
- **Tuple** → is a collection of objects arranged in a fixed order and is written with angle brackets surrounding the objects.

Example: Richard the Lionheart, King of England from 1189 to 1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown.

• { (Richard the Lionheart, King John), (King John, Richard the Lionheart) }

From the above example, the underlined words are the objects.

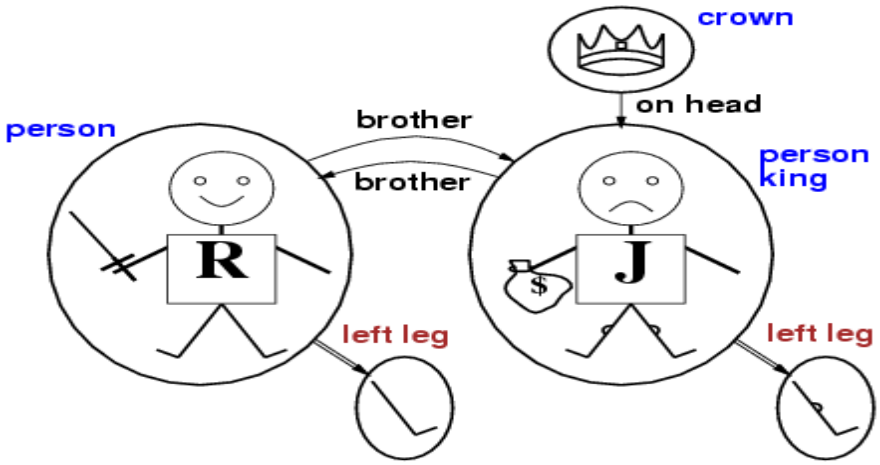


Figure 3.7 A model containing five objects

The above figure 3.7 contains

1) Five objects – Richard, John, Left, Legs, Crown

2) Two binary relations – "Brothers", "on head"

3) Three unary relations (indicated by labels on the objects), - **"Person"** is Richard & John, **"King"** is John and **Crown**

4) one unary function- left-leg - < Richard the LionHeart> → Richard's left leg

< King John> → John's Left leg

<Richard, John> → It is tuple which consists of set of objects in arranged order with angle brackets.

Total functions: Models In FOL requires total function (ie) every input tuple must have a value.

Symbols and interpretations

The basic syntactic elements of first-order logic are the symbols that stand for **objects, relations, and functions.**

The symbols, are in three kinds:

- Constant Symbols, which stand for objects; (*Richard, John)*
- Predicate symbols, which stand for relations; (*Brother, OnHead, Person, King, Crown)*
- Function symbols, which stand for functions. (*LeftLeg)*

- The semantics must relate sentences to models in order to determine truth. For this to happen, we need an interpretation.
- **Interpretation** that specifies exactly which objects, relations, and functions are referred to by the constant, predicate, and function symbols.

Intended Interpretation

Example:

a) **Object** :- Richard refers to Richard the Lionheart and John refers to the evil King John.

b) **Relation** :- Brother refers to the brotherhood relation between Richard and John, that is, the set of tuples of objects given in the above model;

OnHead refers to the "on head" relation that holds between the crown and King John; Person, King, and Crown refer to the sets of objects that are persons, kings, and crowns.

c) **Function** :- LeftLeg refers to the "left leg" function, that is, the mapping given in above model.

For example, one interpretation maps Richard to the crown and John to King John's left leg. There are five objects in the model, so there are 25 possible interpretations just for the constant symbols Richard and John.

Syntax of first-order logic with equality, specified in Backus-Naur form(BNF)

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

Detailed Explanation for all the above elements

1. Terms:

• A term is a logical expression that refers to an object

• To build a term, we use constant symbols, variables and function symbols.

• To build a sentence, Quantifiers and Predicate symbols are used.

• Example: Constant symbol-"King John's left leg" rather than giving a name to his leg.

Function symbol - *LeftLeg(John)*

• Term →f(t1, t2,…tn)

• The function symbol refers to some function in the model. The argument terms t1, t2,…tn refers to objects in the domain d1, d2,…dn.

• Eg. i) John refers to King John.

• ii) Left Leg(John) refers to King John's left leg

2. Atomic Sentences

• Constant symbols and function symbols refers to objects and predicate symbols which further refers to relation is called Atomic Sentence which state facts.

• An Atomic Sentence is formed from a predicate symbol followed by parenthesized list of terms.

Atomic sentence = *predicate* (*term1,...,term n*) or *term1 = term2*

Term = *function* (*term1,...,term n*) or *constant* or *variable*

• Eg. 1) William is the brother of Richard

Brother (William, Richard)

2) Richard's father is married to William's mother

Married (Father(Richard), Mother(William))

• An atomic sentence is true in a given model, under a given interpretation, if the relation referred to by the predicate symbol holds among the objects referred to by the arguments.

3) Complex Sentences

Logical Connectives are used to construct more complex sentences, in which meaning of given sentences has to be satisfied.

• Complex sentences are made from atomic sentences using connectives.

$\neg S$, $S1 \wedge S2$, $S1 \vee S2$, $S1 \Leftrightarrow S2$, $S1 \Rightarrow S2$,

E.g. 1)*Sibling(King John, Richard)* $\Rightarrow$ *Sibling(Richard, King John)*

2) Either Richard is king or John is King.

King(Richard) V King (John)

3) Richard is not king, so it implies John is king

$\neg$King(Richard) $\Rightarrow$ King(John)

4) Quantifiers

• To express properties of entire collections of objects, instead of enumerating the objects by name.

• First-order logic contains two standard quantifiers, called ***Universal*** **(∀) and E*xistential* (∃).**

Universal quantification(∀)

- This quantifier is usually denoted as ∀ and Pronounced as "For All".
- Logical Expression is true for all objects x(variable) in the universe.
- Eg. "All kings are persons" is written in first-order logic as
- ∀x *King (x) Person(x). (ie)* "For all *x*, if *x* is a king, then *x* is a person". X → Variable.
- The symbol x is called **variable**. Eg. LeftLeg(x)
- Ground Term- A term with no variable is called a ground term.
- The sentence ∀*x P*, where *P* is any logical expression, says that *P* is true for every object *x*.

Extended Interpretations

• ∀*x P* is true in a given model under a given interpretation if *P* is true in all possible extended interpretations constructed from the given interpretation, where each extended interpretation specifies a domain element to which *x* refers.

• We can extend the interpretation in five ways:

• *x* → Richard the Lionheart,

• *x* → King John,

• *X* → Richard's left leg,

• $x \rightarrow$ John's left leg,

• $x \rightarrow$ the crown.

• The universally quantified sentence

$\forall x$ *King (x)* $\Rightarrow$ *Person(x)* is true under the original interpretation if the sentence

King (x) $\Rightarrow$ *Person(x)* is true in each of the five extended interpretations.

• The universally quantified sentence is equivalent to asserting the following five sentences:

• Richard the Lionheart is a king $\Rightarrow$ Richard the Lionheart is a person. - True

• King John is a king $\Rightarrow$ King John is a person. - True

• Richard's left leg is a king $\Rightarrow$ Richard's left leg is a person. False

• John's left leg is a king $\Rightarrow$ John's left leg is a person. - False

• The crown is a king $\Rightarrow$ the crown is a person. - False

4.2 Existential quantification($\exists$)

• It makes a statement about **some** object in the universe without naming it, by using an existential quantifier.

• Pronounced as "There Exists" logical expression which is true for some objects x (variable) in the universe.

• Eg.1 "Leesa has a brother who is a dog" which is expressed as

$\exists$x brother(x, Lessa) $\rightarrow$ Dog(x) (ie) Leesa's brother is a dog $\Rightarrow$ Leesa is also a dog and hence x may be replaced by brother of Lessa if it exists.

Eg2: a) Z is a Dog

b) Z is a brother of N

a. Dog (Z)

b. Brother (Z, N)

Eg3. $\exists$ *X Crown(x)* $\Rightarrow$ *OnHead(x, John)*

• Applying the semantics, we see that the sentence says that at least one of the following assertions is true:

• Richard the Lionheart is a crown $\Rightarrow$ Richard the Lionheart is on John's head;

• King John is a crown $\Rightarrow$King John is on John's head;

• Richard's left leg is a crown ⇒ Richard's left leg is on John's head;

• Now an implication is true if both premise and conclusion are true, or if its premise is false**.**

So, in the first assertion, if Richard the LionHeart is not a crown, then the first assertion is true and the existential is satisfied.

Nested Quantifiers

• The sentences are represented using both quantifiers (ie) universal and existential are called Nested Quantifiers. There are two types of sentences in which different nested quantifiers are used.

• Simple sentences use same type of quantifiers.

• Eg: a) Same type : "Brothers are siblings" Note : x,y are brothers

∀x ∀y *Brother(x,y) ⇒Sibling(x,y).*

implies

∀x,y sibiling(x,y) ⇔ sibling(y,x)

equivalence

Example:

"Everybody loves somebody"

∀x∃y Loves(x,y)

"There is someone who is loved by everyone", → ∃y ∀x Loves(x,y)

"Everyone will be loved by some body", → ∀*x (*∃*y Loves(x, y))*

*"*Some one will be loved by everybody" → ∃x(∀y Loves(x,y))

Connections between ∀ and ∃

- The two quantifiers are actually intimately connected with each other, through negation.
- Eg: "Everyone likes ice cream"means that there is no one who does not like ice cream:

∀x *Likes(x,IceCream)* ⇔ ∃⌐x ⌐Likes (x, IceCream).

All X Nobody Dislikes

Because ∀ is a conjunction and ∃ is a disjunction.

De Morgan's rules

∀ is a conjunction and ∃ is a disjunction and its not quantified

Quantified Sentences	Unquantified Sentences
$\forall x \neg P = \neg \exists x P$	$\neg (P \wedge Q) = \neg P \vee \neg Q$
$\neg \forall x P = \exists x \neg P$	$\neg P \wedge \neg Q = \neg (P \vee Q)$
$\forall x P = \neg \exists x \neg P$	$(P \wedge Q) = \neg (\neg P \vee \neg Q)$
$\exists x P = \neg \forall x \neg P$	$(P \vee Q) = \neg (\neg P \wedge \neg Q)$

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)) . Kb – Knowledge Base

TELL(KB, $\forall$x King(x) $\Rightarrow$Person(x)) .

We can ask questions of the knowledge base using ASK.

For example, ASK(KB, King(John))

Questions asked using ASK are called **queries or goals**

(a) ASK(KB, Person(John)) (ie) to find whether John is a person. It is true.

(b) ASK(KB, $\exists$ x Person(x)). It may be True (or) false.

(ie) ASK KB, that there may be some x who is a person and we solve it by providing such an x, that is called substitution list (or) Binding list.

If there is more than one answer, a list of substitutions can be returned.

The kinship domain

• The domain of family relationships is called kinship domain.

• Eg. "E is the mother of C" and "C is the father of W" and rules such as "One's grandmother is the mother of one's parent.“ (ie) “W’s grandmother is the mother of W’s parent”.

• Kinship domain consists of

(a) Objects – people

(b) Unary predicate – Male and Female

(c) Binary predicate – Parent, Brother, Sister

(d) Function – Father. Mother (Every person has exactly one of these)

(e) Relation – Brotherhood, Sisterhood

For example :

• one's mother is one's female parent:

$\forall m,c$ Mother(c)=m $\Leftrightarrow$ Female(m) Λ Parent(m, c). (ie) m is a mother of c $\Leftrightarrow$m is a Female AND m is a parent of C

• One's husband is one's male spouse:

$\forall$ w, h Husband(h, w) $\Leftrightarrow$ Male(h) Λ Spouse(h, w).

(ie) h is a husband (h,w) $\Leftrightarrow$h is male AND h is a spouse of W

• Male and female are disjoint categories:

$\forall x$ Male(x) $\Leftrightarrow \neg$ Female(x). (ie) x is male $\Leftrightarrow$ x is not a Female

• Parent and child are inverse relations:

$\forall$ p, c Parent (p, c) $\Leftrightarrow$ Child (c, p) .(ie) p is parent of c $\Leftrightarrow$c is child of p

Numbers, sets, and lists

Numbers are perhaps the most vivid example of how a large theory can be built up from a tiny kernel of axioms. 0 is natural numbers or nonnegative integers. We need a predicate NatNum that will be true of natural numbers; we need one constant symbol, 0 and we need one function symbol, S (successor).

PEANOAXIOMS : The Peano axioms define natural numbers and addition. Natural numbers are defined recursively: NatNum(O) .

$\forall n$ NatNum(n) $\Rightarrow$ NatNum(S(n)) .

That is, 0 is a natural number, and for every object n, if ri is a natural number then S(n) is a natural number. So the natural numbers are 0, S(O), S(S(O)), and so on.

We also need **axioms to constrain the successor function**:

n O $\neq$ S(n). (ie) Zero will not be equal to Natural number.

$\forall m,n$ m $\neq$ n$\Rightarrow$ S(m) $\neq$ S(n).

Now we can define addition in terms of the successor function:

$\forall m$ NatNum(m) $\Rightarrow$ + (0, m) = m

$\forall m,n$ NatNum(m) Λ NatNum(n) $\Rightarrow$ + (S(m),n) = S(+(m,n)).

Generally, adding 0 to any natural number m gives m itself but when we use the binary function symbol "+" in the term + (m, 0). In ordinary mathematics, the term would be written m

+ 0 using infix notation. To make our sentences about numbers easier to read, we will allow the use of infix notation.

SYNTACTIC SUGAR : An extension to or abbreviation of the standard syntax that does not change the semantics.

SETS :

• The domain of mathematical set representation, which consists of

(a) Constant – Empty Set { }

(b) Predicates – Member and subset

(c) Functions – Intersection, Union, and Adjoin

{(s1 ∩ s2) , (s1 ∪ s2), x|s2}

One possible set of axioms is as follows:

1. The only sets are the empty set and those made by adjoining something to a set:

$$\forall s\ Set(s) \Leftrightarrow (s = \{\}) \vee (\exists x,s2\ Set(s2) \wedge s = \{x|s2\}) .$$

2. The empty set has no elements adjoined into it, in other words, there is no way to decompose *EmptySet* into a smaller set and an element:

$$\neg\exists x,s\ \{x|s\} = \{\}$$

3. Adjoining an element already in the set has no effect:

$$\forall x,s\ x \in s \Leftrightarrow s = \{x|s\}$$

4. The only members of a set are the elements that were adjoined into it.

$$\forall x,s\ x \in s \Leftrightarrow [\exists y,s2\}\ (s = \{y|s2\} \wedge (x = y \vee x \in s2))]$$

5. A set is a subset of another set if and only if all of the first set's members are members of the second set:

$$\forall\ s1,s2\ s1 \subseteq s2 \Leftrightarrow (\forall x\ x \in s1 \Rightarrow x \in s2)$$

6. Two sets are equal if and only if each is a subset of the other:

$$\forall s1,s2\ (s1 = s2) \Leftrightarrow (s1 \subseteq s2 \wedge s2 \subseteq s1)$$

7. An object is in the intersection of two sets if and only if it is a member of both sets:

$$\forall x,s1,s2\ x \in (s1 \cap s2) \Leftrightarrow (x \in s1 \wedge x \in s2)$$

8. An object is in the union of two sets if and only if it is a member of either set:

$$\forall x,s1,s2\ x \in (s1 \cup s2) \Leftrightarrow (x \in s1 \vee x \in s2)$$

Lists

Lists are similar to sets. The element in the list can appear multiple times. The lists are ordered. Constant list is *NIL,* which has no element. *Cons, Append, first and rest* are the functions. *Find* is the predicate.

The empty list is []. The term *Cons(x, y),* where *y* is a nonempty list, is written *[xly]*.

The term *Cons(x, Nil),*is written as *[x]*.

A list of several elements, such as *[A, B,* CJ, corresponds to the nested term *Cons(A, Cons(B, Cons(C, Nil))).*

SITUATION CALCULUS

- The Situation Calculus is a logic formalism designed for representing and reasoning about **dynamical domains**.
    - McCarthy, Hayes 1969
    - Reiter 1991
- In First-Order Logic, sentences are either true or false and stay that way. Nothing is corresponding to any sort of change.
- SitCalc represents changing scenarios as a set of SOL formulae.
- A domain is encoded in SOL by three kind of formulae
    - **Action precondition axioms** and **action effects axioms**
    - **Successor state axioms**, one for each fluent
    - **The foundational axioms** of the situation calculus
- **Situation Calculus : An Example**
- **World**:
    - robot
    - items
    - locations (x,y)
    - moves around the world
    - picks up or drops items
    - some items are too heavy for the robot to pick up
    - some items are fragile so that they break when they are dropped
    - robot can repair any broken item that it is holding
- **Actions**

- **move(x, y):** robot is moving to a new location **(x, y)**
- **pickup(o):** robot picks up an object **o**
- **drop(o):** robot drops the object **o** that holds

- **Situations**
  - Initial situation $\mathbf{S_0}$: no actions have yet occurred
  - A new situation, resulting from the performance of an action a in current situation **s**, is denoted using the function symbol **do(a, s)**.
    - **do(move(*2*, *3*), $\mathbf{S_0}$):** denotes the new situation after the performance of action **move(*2*, *3*)** in initial situation $\mathbf{S_0}$.
    - **do(pickup(*Ball*), do(move(*2*, *3*), $\mathbf{S_0}$))**
    - **do(a,s)** is equal to **do(a',s')** **s=s' and a=a'**
- **Fluents: "properties of the world"**
  - **Relational fluents**
    - Statements whose truth value may change
    - They take a situation as a final argument
    - **is_carrying(o, s)**: robot is carrying object **o** in situation **s**
  - **E.g. Suppose that the robot initially carries nothing**
    - **is_carrying(*Ball*, $\mathbf{S_0}$) : FALSE**
    - **is_carrying(*Ball*, do(pickup(*Ball*), $\mathbf{S_0}$)) : TRUE**
- **Action Preconditions Axioms**
  - Some actions may not be executable in a given situation
  - **Poss(a,s):** special binary predicate
    - denotes the executability of action **a** in situation **s**
  - **Examples:**
  - **Poss(drop(o),s) ↔ is_carrying(o,s)**
  - **Poss(pickup(o),s) ↔ (∀z ¬is_carrying(z,s) ∧¬heavy(o))**
- **Action Effects Axioms**
  - Specify the effects of an action **on the fluents**
  - **Examples:**
  - **Poss(pickup(o),s) → is_carrying(o,do(pickup(o),s))**
  - **Poss(drop(o),s) ∧ fragile(o) → broken(o,do(drop(o),s))**

- Is that enough? No, because of the **frame problem**

BULIDING A KNOWLEDGE BASE

Knowledge engineering projects vary widely in content, scope, and difficulty, but all such projects include the following steps:

1. **Identify the task** – The knowledge engineer must delineate (ie) describe the range of questions that the knowledge base will support and the kinds of facts that will be available for each specific problem instance.

2. **Assemble the relevant knowledge** – The knowledge engineer might already be an expert in the domain to extract what they know this process is called Knowledge Acquisition. At this stage, the knowledge is not represented and the main idea is is to understand the scope of the knowledge base, as determined by the task, and to understand how the domain actually works.

3. **Decide on a vocabulary of predicates, functions, and constants** - 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. The ontology determines what kinds of things exist, but does not determine their specific properties and interrelationships.

4. **Encode general knowledge about the domain** - The knowledge engineer writes down the axioms for all the vocabulary terms (ie) enabling the expert to check the content. 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 specific problem instance** - It will involve writing simple atomic sentences about instances of concepts that are already part of the ontology

6. **Pose queries to the inference procedure and get answers** - 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** – Answers will be correct for the knowledge base as written, assuming that the inference procedure is sound, but they will not be the ones that the user is expecting. A debugging process could confirm missing axioms or axioms that are too weak can be identified easily by noticing places where the chain of reasoning stops unexpectedly. 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 3.8.

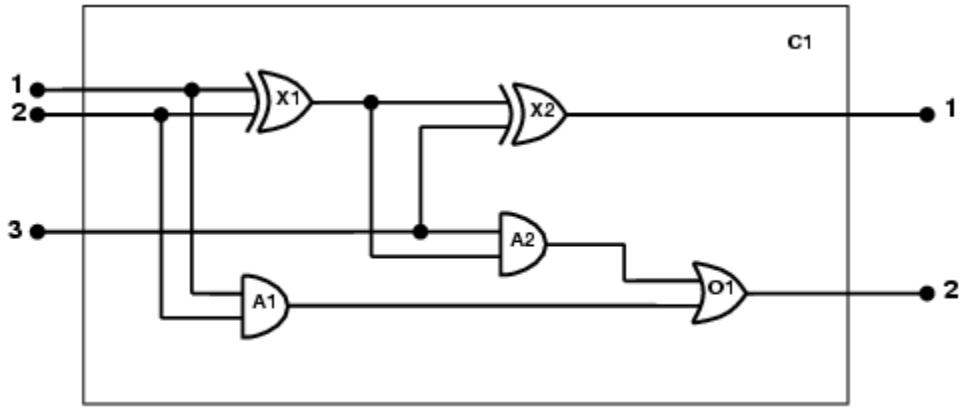


Figure 3.8 A digital circuit C1, purporting to be a one-bit full adder

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.

We follow the seven-step process for knowledge engineering

1. Identify the task

   – Does the circuit actually add properly? (circuit verification)

2. Assemble the relevant knowledge

   – Composed of wires and gates; Types of gates (AND, OR, XOR, NOT)

   – Irrelevant: size, shape, color, cost of gates

3. Decide on a vocabulary

   Type(X1) = XOR

   Type(X1, XOR)

   XOR(X1)

4. Encode general knowledge of the domain

   – $\forall t1,t2$ Connected(t1, t2) $\Rightarrow$ Signal(t1) = Signal(t2)

   – $\forall t$ Signal(t) = 1 V Signal(t) = 0

   – $1 \neq 0$

   – $\forall t1,t2$ Connected(t1, t2) $\Leftrightarrow$ Connected(t2, t1)

– ∀g Type(g) = OR ⇒ Signal(Out(1,g)) = 1 ⇔ ∃n Signal(In(n,g)) = 1

– ∀g Type(g) = AND ⇒ Signal(Out(1,g)) = 0 ⇔ ∃n Signal(In(n,g)) = 0

– ∀g Type(g) = XOR ⇒ Signal(Out(1,g)) = 1 ⇔ Signal(In(1,g)) ≠ Signal(In(2,g))

– ∀g Type(g) = NOT ⇒ Signal(Out(1,g)) ≠ Signal(In(1,g))

5. Encode the specific problem instance

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

Type(A1) = AND Type(A2) = AND

Type(O1) = OR

Connected(Out(1,X1),In(1,X2)) Connected(In(1,C1),In(1,X1))

Connected(Out(1,X1),In(2,A2)) Connected(In(1,C1),In(1,A1))

Connected(Out(1,A2),In(1,O1)) Connected(In(2,C1),In(2,X1))

Connected(Out(1,A1),In(2,O1)) Connected(In(2,C1),In(2,A1))

Connected(Out(1,X2),Out(1,C1)) Connected(In(3,C1),In(2,X2))

Connected(Out(1,O1),Out(2,C1)) Connected(In(3,C1),In(1,A2))

6. Pose queries to the inference procedure

What are the possible sets of values of all the terminals for the adder circuit?

∃i1,i2,i3,o1,o2 Signal(In(1,C1)) = i1 Λ Signal(In(2,C1)) = i2 Λ Signal(In(3,C1)) = i3 Λ Signal(Out(1,C1)) = 0 Λ Signal(Out(2,C1)) = 1.

7. Debug the knowledge base May have omitted assertions like 1 ≠ 0

Suddenly, the system will be unable to prove any outputs for the circuit, except for the input cases 000 and 110. We can pinpoint the problem by asking for the outputs of each gate.

ONTOLOGY

Ontology means Remaining. Representing the abstract concepts is sometimes called **ontological engineering**-it is related to the knowledge engineering process.

In "toy" domains, the choice of representation is not that important; it is easy to come up with a consistent vocabulary.

Real world problems such as shopping on the Internet or controlling a robot in a changing physical environment require more general and flexible representations, we have many choice of representation like Actions, Time, Physical objects and Beliefs so this occur in different domains.

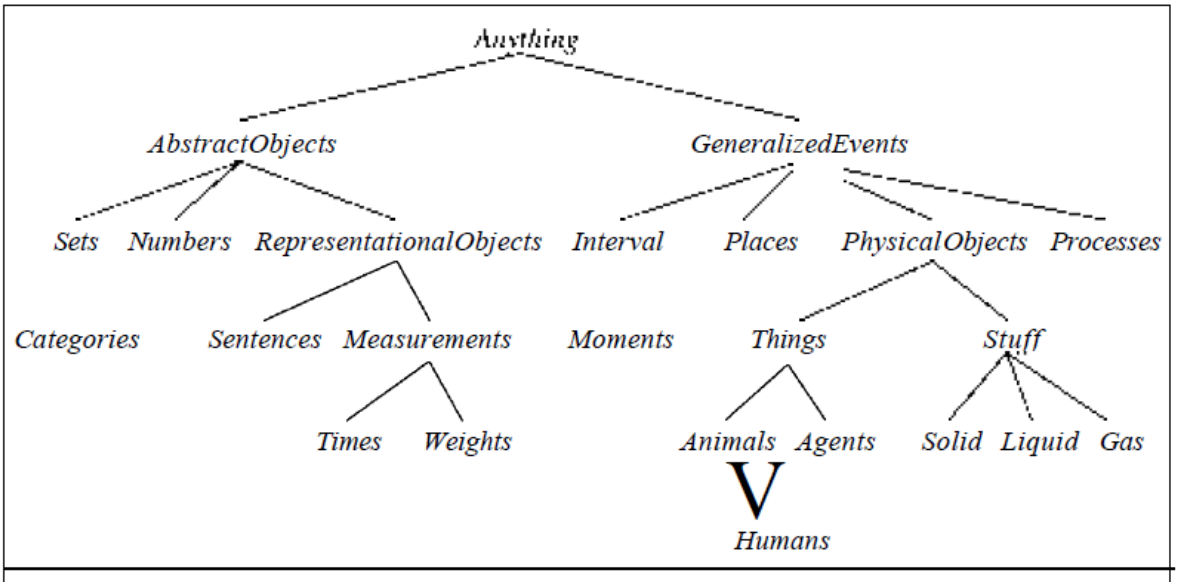


Figure 3.9 The upper ontology of the world

The general framework of concepts is called an **upper ontology,** because of the convention of drawing graphs with the general concepts at the top and the more specific concepts below them, as in above Figure 3.9.

Characteristics of General purpose ontology.

1. A general purpose ontology should be applicable in more or less special purpose domain.
2. In any demanding domain, different areas of knowledge must be unified.

FORWARD AND BACKWARD CHAINING

Forward chaining is one of the two main methods of reasoning when using an inference engine and can be described logically as repeated application of *modus ponens*. Forward chaining is a popular implementation strategy for expert systems, business and production rule systems. The opposite of forward chaining is backward chaining.

Forward chaining starts with the available data and uses inference rules to extract more data (from an end user, for example) until a goal is reached. An inference engine using forward chaining searches the inference rules until it finds one where the antecedent (**If** clause) is known to be true. When such a rule is found, the engine can conclude, or infer, the consequent (**Then** clause), resulting in the addition of new information to its data.

Inference engines will iterate through this process until a goal is reached.

Example:1, suppose that the goal is to conclude the color of a pet named Fritz, given that he croaks and eats flies, and that the rule base contains the following four rules:

1. **If** X croaks and X eats flies - **Then** X is a frog
2. **If** X chirps and X sings - **Then** X is a canary
3. **If** X is a frog - **Then** X is green
4. **If** X is a canary - **Then** X is yellow

Let us illustrate forward chaining by following the pattern of a computer as it evaluates the rules. Assume the following facts:

- Fritz croaks
- Fritz eats flies

With forward reasoning, the inference engine can derive that Fritz is green in a series of steps:

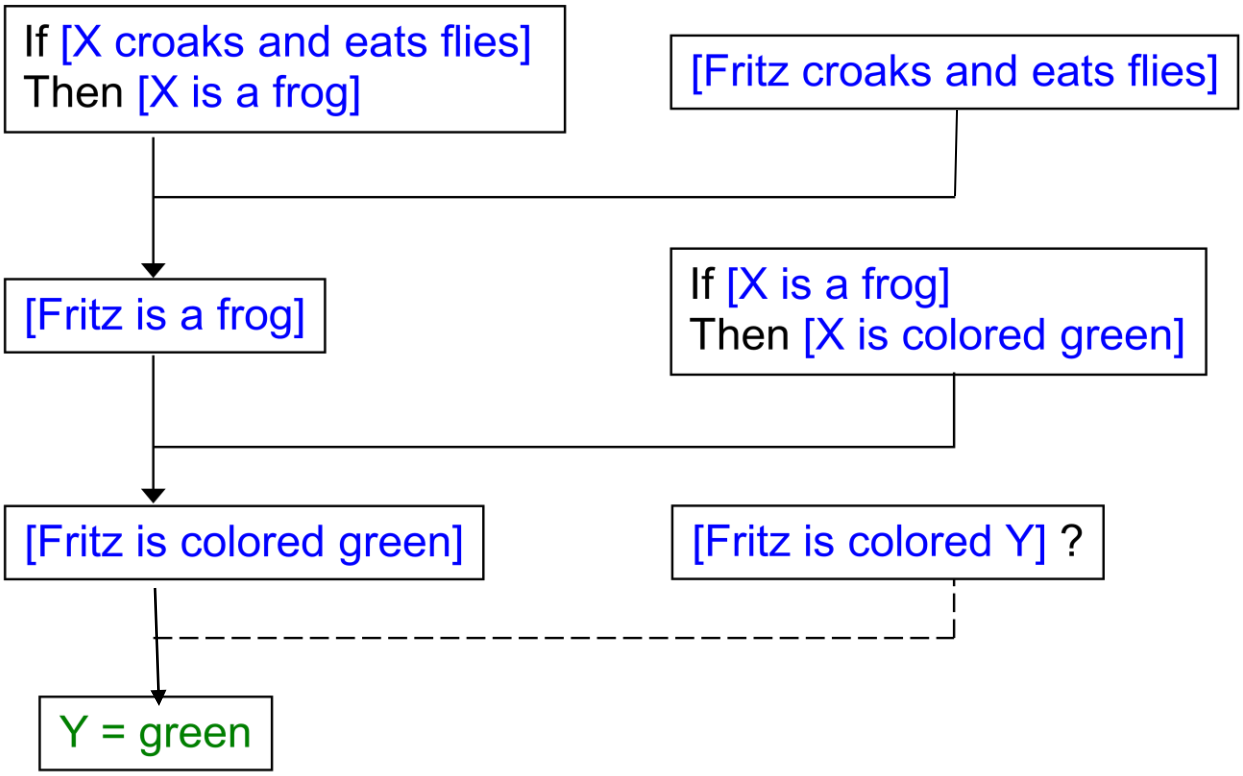
1. Since the base facts indicate that "Fritz croaks" and "Fritz eats flies", the antecedent of rule #1 is satisfied by substituting Fritz for X, and the inference engine concludes:

Fritz is a frog

2. The antecedent of rule #3 is then satisfied by substituting Fritz for X, and the inference engine concludes:

Fritz is green

The name "forward chaining" comes from the fact that the inference engine starts with the data and reasons its way to the answer, as opposed to backward chaining, which works the other way around. In the derivation, the rules are used in the opposite order as compared to backward chaining. In this example, rules #2 and #4 were not used in determining that Fritz is green.



Example:2

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.

To prove:

"West is a criminal".

Proof :

First, represent these facts as first-order definite clauses. The forward-chaining algorithm solves this problem.

Step 1: " it is a crime for an American to sell weapons to hostile nations"

American(x) $\wedge$ Weapon(y) $\wedge$ Sells(x,y,z) $\wedge$ Hostile(z) $\Rightarrow$ Criminal(x)

Step 2: " Nono … has some missiles", Missile(x): is transformed in to 2 definite clauses :

$\wedge$x Owns(Nono,x) $\exists$i.e., Owns(Nono,M1) and Missile(M1)

where M1 is a new constant

Step 3: "… all of its missiles were sold to it by Colonel West"

Sells(West, x, Nono)$\Rightarrow$ Owns(Nono ,x) $\wedge$Missile(x)

Step 4: Missiles are weapons:

$$Weapon(x) \Rightarrow Missile(x)$$

Step 5:An enemy of America counts as "hostile“:

$$Hostile(x) \Rightarrow Enemy(x,America)$$

Step 6: West, who is American …

$$American(West)$$

Step 7: The country Nono, an enemy of America …

$$Enemy(Nono, America)$$

This knowledge base contains no function symbols and is therefore an instance of the class of Datalog knowledge bases-that is, sets of first-order definite clauses with no function symbols. The diagrammatic representation of Forward Chaining shown in the below figure 3.10.

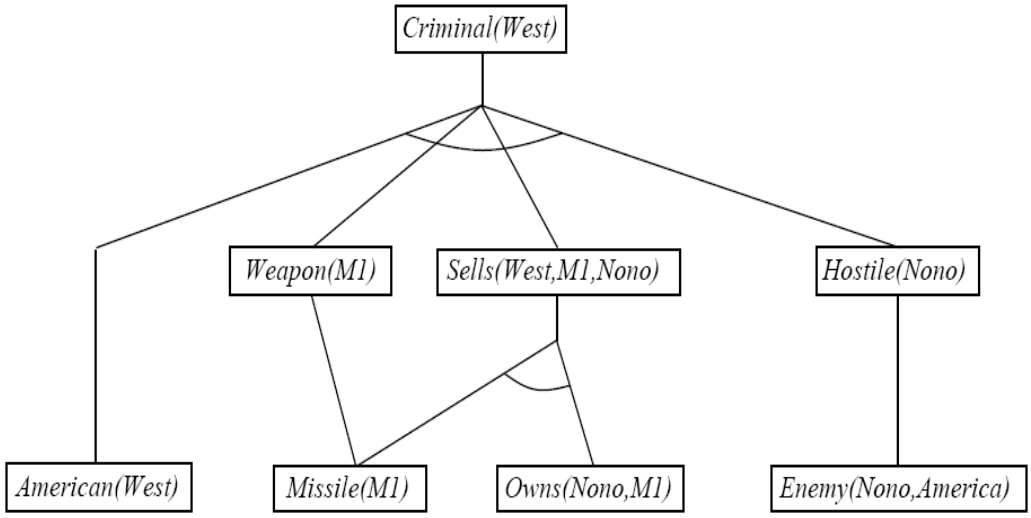


Figure 3.10 Example for forward Chaining

A simple forward-chaining algorithm, Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions to the known facts. The process repeats until the query is answered or no new facts are added. A fact is not "new" if it is just a renaming of a known fact.

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
```

Backward chaining

Backward chaining (or **backward reasoning**) is an inference method that can be described (in lay terms) as working backward from the goal(s). It is used in automated theorem provers, inference engines, proof assistants and other artificial intelligence applications.

Backward chaining starts with a list of goals (or a hypothesis) and works backwards from the consequent to the antecedent to see if there is data available that will support any of these consequents.[3] An inference engine using backward chaining would search the inference rules until it finds one which has a consequent (**Then** clause) that matches a desired goal. If the antecedent (**If** clause) of that rule is not known to be true, then it is added to the list of goals (in order for one's goal to be confirmed one must also provide data that confirms this new rule).

For example, suppose a new pet, Fritz, is delivered in an opaque box along with two facts about Fritz:

- Fritz croaks
- Fritz eats flies

The goal is to decide whether Fritz is green, based on a rule base containing the following four rules:

1. **If** X croaks and X eats flies – **Then** X is a frog
2. **If** X chirps and X sings – **Then** X is a canary
3. **If** X is a frog – **Then** X is green
4. **If** X is a canary – **Then** X is yellow

With backward reasoning, an inference engine can determine whether Fritz is green in four steps. To start, the query is phrased as a goal assertion that is to be proved: "Fritz is green".

1. Fritz is substituted for X in rule #3 to see if its consequent matches the goal, so rule #3 becomes:

If Fritz is a frog – **Then** Fritz is green

Since the consequent matches the goal ("Fritz is green"),the rules engine now needs to see if the antecedent ("If Fritz is a frog") can be proved. The antecedent therefore becomes the new goal:

Fritz is a frog

2. Again substituting Fritz for X, rule #1 becomes:

If Fritz croaks and Fritz eats flies – **Then** Fritz is a frog

Since the consequent matches the current goal ("Fritz is a frog"), the inference engine now needs to see if the antecedent ("If Fritz croaks and eats flies") can be proved. The antecedent therefore becomes the new goal:

Fritz croaks and Fritz eats flies

3. Since this goal is a conjunction of two statements, the inference engine breaks it into two sub-goals, both of which must be proved:

Fritz croaks

Fritz eats flies

4. To prove both of these sub-goals, the inference engine sees that both of these sub-goals were given as initial facts. Therefore, the conjunction is true:

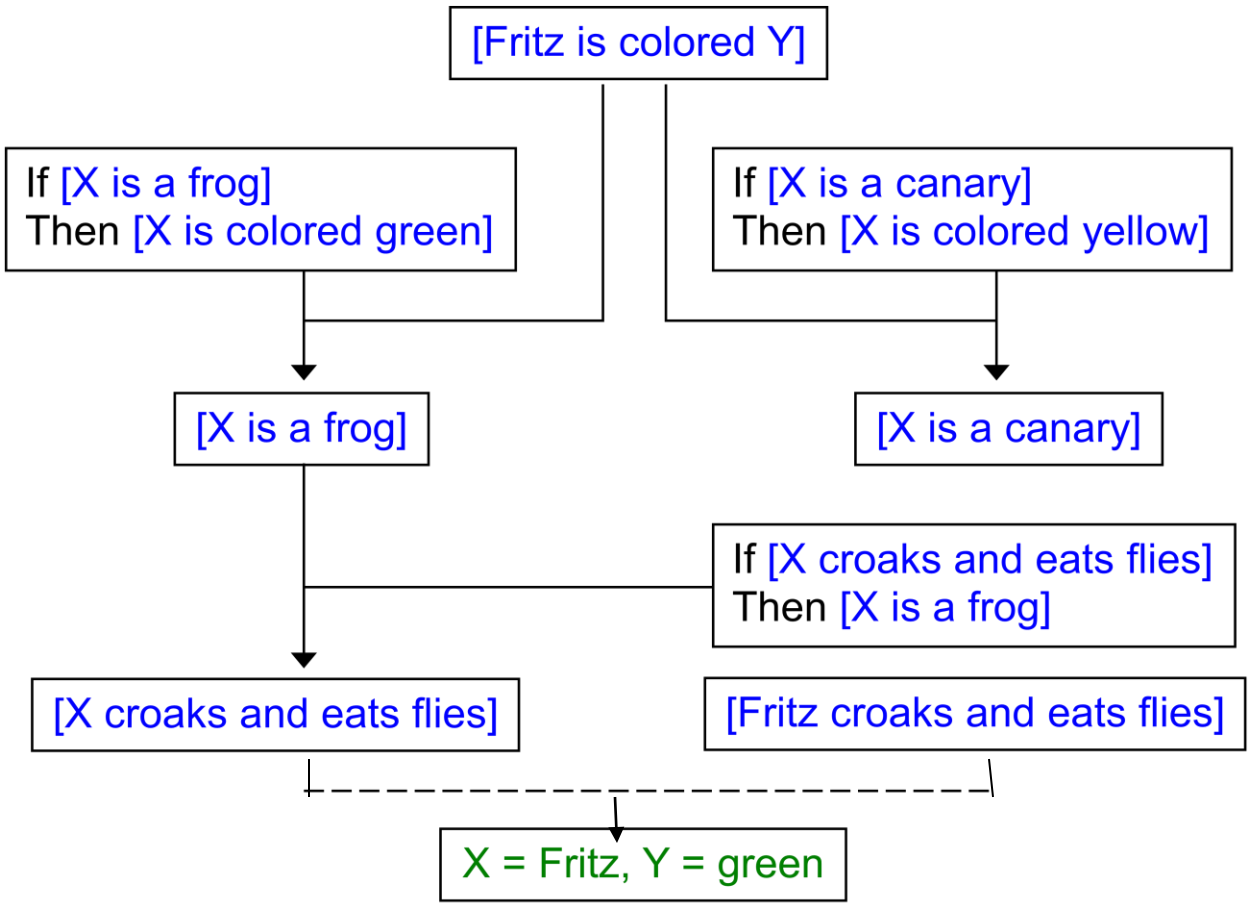
Fritz croaks and Fritz eats flies therefore the antecedent of rule #1 is true and the consequent must be true:

Fritz is a frog

Therefore the antecedent of rule #3 is true and the consequent must be true:

Fritz is green

This derivation therefore allows the inference engine to prove that Fritz is green. Rules #2 and #4 were not used.



Difference

Backward chaining (a la Prolog) is more like finding what initial conditions form a path to your goal. At a very basic level it is a backward search from your goal to find conditions that will fulfil it.

Backward chaining is used for interrogative applications (finding items that fulfil certain criteria) - one commercial example of a backward chaining application might be finding which insurance policies are covered by a particular reinsurance contract.

Forward chaining (a la CLIPS) matches conditions and then generates inferences from those conditions. These conditions can in turn match other rules. Basically, this takes a set of initial conditions and then draws all inferences it can from those conditions.

Algorithm

function FOL-BC-ASK(KB, goals, θ) **returns** a set of substitutions
  inputs: KB, a knowledge base
    $goals$, a list of conjuncts forming a query
    θ, the current substitution, initially the empty substitution $\{\}$
  local variables: ans, a set of substitutions, initially empty

  if $goals$ is empty **then return** $\{\theta\}$
  $q' \leftarrow$ SUBST(θ, FIRST($goals$))
  for each r **in** KB where STANDARDIZE-APART(r) $= (p_1 \wedge \ldots \wedge p_n \Rightarrow q)$
    and $\theta' \leftarrow$ UNIFY(q, q') succeeds
   $ans \leftarrow$ FOL-BC-ASK(KB, $[p_1, \ldots, p_n | \text{REST}(goals)]$, COMPOSE($\theta', \theta$)) $\cup$ ans
  return ans

RESOLUTION

In 1930, the German mathematician Kurt Godel proved the first completeness theorem for first-order logic, showing that any entailed sentence has a finite proof. In 1931, Godel proved an even more famous incompleteness theorem.

- The theorem states that a logical system that includes the principle of induction without which very little of discrete mathematics can be constructed-is necessarily incomplete. Hence, there are sentences that are entailed, but have no finite proof within the system.
- Resolution-based theorem provers have been applied widely to derive mathematical theorems, including several for which no proof was known previously. Theorem provers have also been used to verify hardware designs and to generate logically correct programs, among other applications like
  - Conjunctive normal form for first-order logic.
  - The resolution inference rule

- Completeness of Resolution
- Dealing with equality
- Resolution strategies
- Theorem provers

Conjunctive normal form for first-order logic

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.

Literals can contain variables, which are assumed to be universally quantified. For example, the sentence

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

Becomes in CNF,

¬American(x) V ¬Weapon(y))V¬ Sells(x, y, z) V¬Hostile(z) V Criminal(x)

Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence. The procedure for conversion to CNF is very similar to the propositional case.

Eg: **Everyone who loves all animals is loved by someone**

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

The steps are as follows:

1. **Eliminatebiconditionals and implications:**

   $\forall x$ [¬$\forall y$¬ Animal(y) V Loves(x,y)] V[$\exists y$ Loves(y,x)]

2.Move ¬ inwards:

   ¬$\forall x$ p becomes $\exists x$ ¬p

   ¬$\exists x$ p becomes $\forall x$ ¬p

The transforms are

   $\forall x$ [$\exists y$¬(¬ Animal(y) V Loves(x,y))] V[$\exists y$ Loves(y,x)]

   $\forall x$ [$\exists y$¬¬ Animal(y) $\wedge$¬ Loves(x,y)] V[$\exists y$ Loves(y,x)]

   $\forall x$ [$\exists y$ Animal(y) $\wedge$¬ Loves(x,y)] V[$\exists y$ Loves(y,x)]

3. Standardize variables: For sentences like (Vx P(x)) V (:3x Q(x)) which use the same variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers. Thus, we have each quantifier should use a different one

   $\forall x$ [$\exists y$ Animal(y) $\wedge$¬ Loves(x,y)] V[$\exists z$ Loves(z,x)]

4. Skolemize: a more general form of existential instantiation. Skolemization is the process of removing existential quantifiers by elimination. Each existential variable is replaced by a Skolem function of the enclosing universally quantified variables:

$$\forall x\ [\ Animal(A) \wedge \neg\ Loves(x,A)]\ V\ Loves(B,x)$$

5. Drop universal quantifiers: At this point, all remaining variables must be universally quantified. Moreover, the sentence is equivalent to one in which all the universal quantifiers have been moved to the left. We can therefore drop the universal quantifiers.

$$Animal(F(x)) \wedge \neg\ Loves(x,F(x))]\ V\ Loves(G(x),x)$$

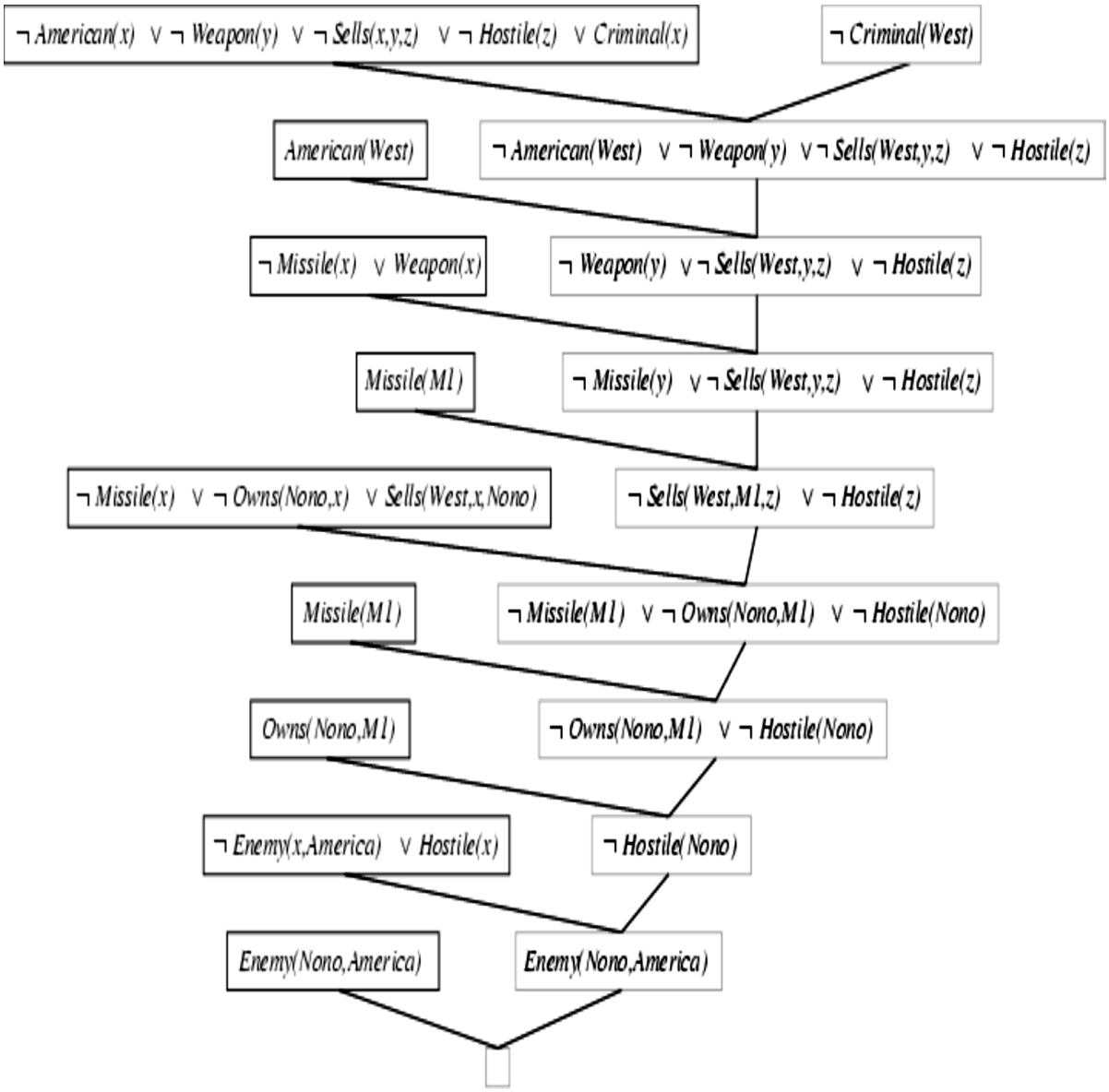
6.**Distribute $\wedge$ over V:**

$$[Animal(F(x))\ V\ \ Loves(G(x),x)] \wedge [\ \neg Loves(x,F(x))VLoves(G(x),x)]$$

This step may also require flattening out nested conjunctions and disjunctions. The sentence is now in CNF and consists of two clauses.

1. F(x) refers to the animal potentially unloved by x
2. G(x) refers to someone who might love x

Example



Example

This makes use of Skolemization and involves clauses that are not definite clauses. This results in a somewhat more complex proof structure. In English, the problem is as follows:

Everyone who loves all animals is loved by someone.

Anyone who kills an animal is loved by no one.

Jack loves all animals.

Either Jack or Curiosity killed the cat, who is named Tuna.

Did Curiosity kill the cat?

First, we express the original sentences, some background knowledge, and the negated goal G in first-order logic:

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

B. $\forall x\ [\exists y\ Animal(y) \wedge Kills(x,y)] \Rightarrow [\forall z\ \neg Loves(z, x)]$

C. $\forall x\ Animal(x) \Rightarrow Loves(Jack, x)$

D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$

E. $Cat(Tuna)$

F. $\forall x\ Cat(x) \Rightarrow Animal(x)$

-.G. $\neg Kills(Curiosity, Tuna)$

Now we apply the conversion procedure to convert each sentence to CNF:

AI. $Animal(F(x)) \vee Loves(G(x), x)$

A2. $\neg Loves(x, F(x)) \vee Loves(G(x), x)$

B. $\neg Animal(y) \vee \neg Kills(x, y) \vee \neg Loves(z, x)$

C. $\neg Animal(x) \vee Loves(Jack, x)$

D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$

E. $Cat(Tuna)$

F. $\neg Cat(x) \vee Animal(x)$

-,G. $\neg Kills(Curiosity, Tuna)$

Example:

1.Jack owns a dog.

2.Every dog owner is an animal lover.

3.No animal lover kills an animal.

4.Either Jack or Curiosity killed the cat, who is named Tuna.

5.Did Curiosity kill the cat?

1. $\exists x : Dog(x) \wedge Owns(Jack, x)$

2. $\forall x; \quad (\exists y \quad Dog(y) \wedge Owns(x, y)) \rightarrow AnimalLover(x)$

3. $\forall x; \quad AnimalLover(x) \rightarrow (\forall y \quad Animal(y) \rightarrow \neg Kills(x, y))$

4. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$

5. $Cat(Tuna)$

6. $\forall x : \quad Cat(x) \rightarrow Animal(x)$

Conjunctive Normal Form

$Dog(D)$

$Owns(Jack, D)$

$\neg Dog(y) \vee \neg Owns(x, y) \vee AnimalLover(x)$

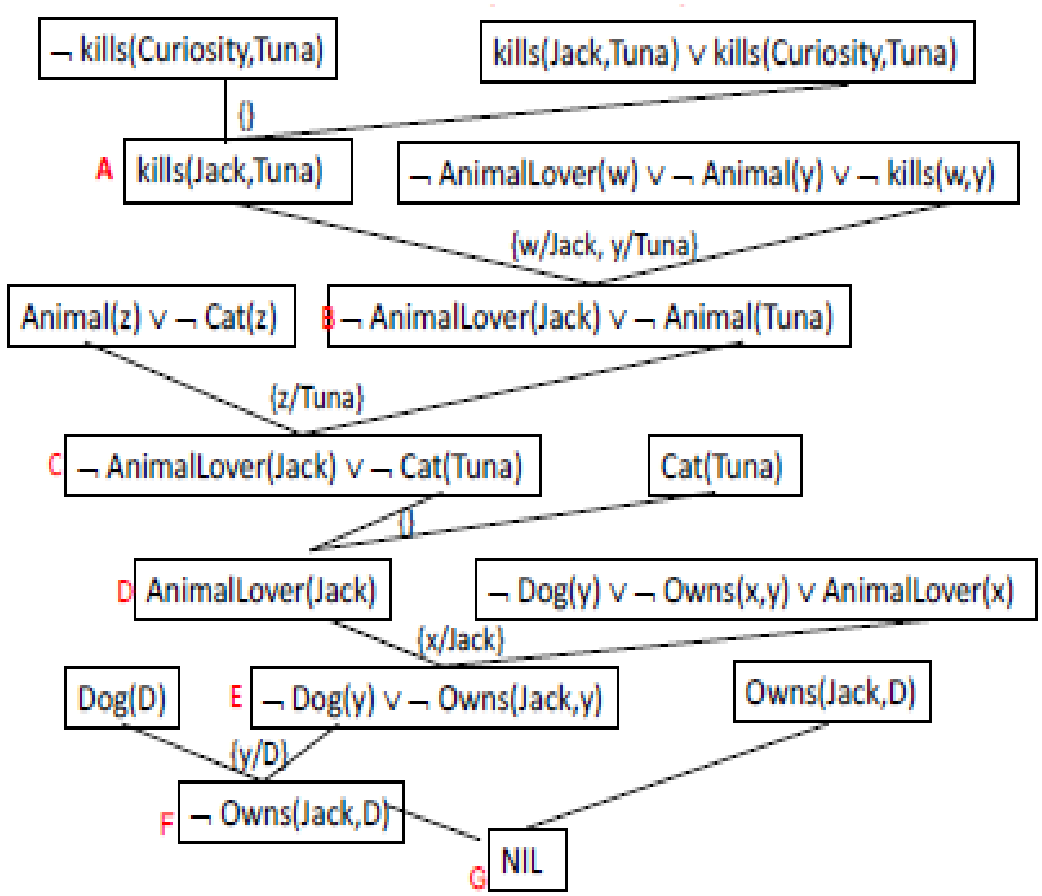
$\neg AnimalLover(w) \vee \neg Animal(y) \vee \neg Kills(w, y)$

$Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$

$Cat(Tuna)$

$\neg Cat(z) \vee Animal(z)$

$\neg Kills(Curiosity, Tuna)$



Resolution Strategies

Unit resolution: prefer to perform resolution if one clause is just a literal yields shorter sentences.

•**Set of support** : identify a subset of the KB (hopefully small); every resolution will take a clause from the set and resolve it with another sentence, then add the result to the set of support.

•**Input resolution**: always combine a sentence from the query or KB with another sentences.

Linear resolution: resolve P and Q if P is in the original KB or is an ancestor of Q in the proof tree.

• **Subsumption**: eliminate all sentences more specific than a sentence already in the KB

Demodulation

Demodulation is typically used for simplifying expressions using collections of assertions such as $x + 0 = x$, $x1 = x$, and so on. The rule can also be extended to handle non-unit clauses in which an equality literal appears:

Paramodulation

Unlike demodulation, paramodulation yields a complete inference procedure for first-order logic with equality.

TRUTH MAINTENANCE SYSTEM

Many of the inferences drawn by a knowledge representation system will have only default status, rather than being absolutely certain. Inevitably, some of these inferred facts will turn out to be wrong and will have to be retracted in the face of new information. This process is called **belief revision**.

Truth Maintenance Systems (TMS) have been developed as a means of implementing Non-Monotonic Reasoning Systems.

Basically TMSs:

- all do some form of dependency directed backtracking
- assertions are connected via a network of dependencies.

Justification-Based Truth Maintenance Systems (JTMS)

- This is a simple TMS in that it does not know anything about the structure of the assertions themselves.
- Each supported belief (assertion) in has a justification.
- Each justification has two parts:
    - An IN-List -- which supports beliefs held.
    - An OUT-List -- which supports beliefs not held.
- An assertion is connected to its justification by an arrow.
- One assertion can feed another justification thus creating the network.
- Assertions may be labelled with a belief status.
- An assertion is valid if every assertion in the IN-List is believed and none in the OUT-List are believed.
- An assertion is non-monotonic is the OUT-List is not empty or if any assertion in the IN-List is non-monotonic.

Logic-Based Truth Maintenance Systems (LTMS)

Similar to JTMS except:

- Nodes (assertions) assume no relationships among them except ones explicitly stated in justifications.
- JTMS can represent P and ¬P simultaneously. An LTMS would throw a contradiction here.
- If this happens network has to be reconstructed.

Assumption-Based Truth Maintenance Systems (ATMS)

- JTMS and LTMS pursue a single line of reasoning at a time and backtrack (dependency-directed) when needed -- depth first search.
- ATMS maintain alternative paths in parallel -- breadth-first search
- Backtracking is avoided at the expense of maintaining multiple contexts.
- However as reasoning proceeds contradictions arise and the ATMS can be pruned
    - Simply find assertion with no valid justification.