Module 1

Module 2

Module 3
Game playing and knowledge structures – Games as search problem – Imperfect decisions – Evaluation functions – Alpha – Beta pruning – state of art game programs, Introduction to frames and semantic nets.

Module 4

Module 5

Text Books
Module 1, 2, 3, 4
1. Artificial Intelligence – A modern approach, Stuact Russell – Peter Narang, Pearson Education Asia

Module 5
3. Artificial Intelligence, George F Luger, Pearson Education Asia

References
1. An Introduction to Artificial Intelligence – Eugene Charniak & Drew McDermot, Pearson Education Asia
Artificial Intelligence (ref: AI by Rich & Knight)

Artificial intelligence is the study of how to make computers do things which, at the moment, people do better.

Much of the early work in AI focused on formal tasks, such as game playing and theorem proving.

Some of the game playing programs are
- Checkers playing program,
- Chess.

Some of the theorem proving programs are
- Logic theorist,
- Galernter’s theorem prover.

Another area of AI is common sense reasoning. It includes reasoning about physical objects and their relationships to each other as well as reasoning about actions and their consequences.

Eg., General problem solver (GPS)

As AI research progressed, new tasks were solved such as perception (vision and speech), natural language understanding and problem solving in domains such as medical diagnosis and chemical analysis.

Perception

Perceptual tasks are very difficult because they involve analog signals.
Natural language understanding

The problem of understanding spoken language is a perceptual problem and is hard to solve. But if we try to simplify the problem by restricting it to written language, then also it is extremely difficult. This problem is referred to as natural language understanding.

In addition to these, many people can perform specialized tasks in which expertise is necessary. Eg. Engineering design,
Scientific discovery,
Medical diagnosis,
Financial planning.

These expert tasks require knowledge that many of us do not have, they often require much less knowledge than do the mundane tasks. This knowledge is easier to represent and deal with inside programs.

As a result, the problem areas where AI is now flourishing most as a practical discipline are primarily the domains that require only specialized expertise without the assistance of commonsense knowledge. There are now thousands of programs called expert systems in day to day operation throughout all areas of industry and government.

Mundane tasks

- Perception
  - Vision
  - Speech
- Natural language
  - Understanding
  - Generation
  - Translation
- Commonsense reasoning
- Robot control
Formal tasks

- Games
  - Chess
  - Backgammon
  - Checkers
  - Go
- Mathematics
  - Geometry
  - Logic
  - Integral calculus
  - Proving properties of programs

Expert tasks

- Engineering
  - Design
  - Fault finding
  - Manufacturing planning
- Scientific analysis
- Medical diagnosis
- Financial analysis
Definitions of artificial intelligence according to 8 text books are given below.

Artificial intelligence is a system that thinks like human beings.

1. **AI is an exciting effort to make computers think… machines with minds, in the full and literal sense.**
2. **AI is the automation of activities that we associate with human thinking, activities such as decision making, problem solving, learning.**

Artificial intelligence is a system that thinks rationally.

3. **AI is the study of mental faculties through the use of computational models.**
4. **AI is the study of computations that make it possible to perceive, reason and act.**

Artificial intelligence is a system that acts like human beings.

5. **AI is the art of creating machines that perform functions that require intelligence when performed by people.**
6. **AI is the study of how to make computers do things at which, at the moment, people do better.**
Artificial intelligence is a system that acts rationally.

7. **AI is the study of the design of intelligent agents.**
8. **AI is concerned with intelligent behavior in artifacts.**

---

**AI is a system that acts like human beings**

For this, a computer would need to possess the following capabilities.

- **Natural language processing**
  - To enable it to communicate successfully in English.
- **Knowledge representation**
  - To store what it knows or hears.
- **Automated reasoning**
  - To use the stored information to answer questions and to draw new conclusions.
- **Machine learning**
  - To adapt to new circumstances and to detect and extrapolate patterns.
- **Computer vision**
  - To perceive objects.
- **Robotics**
  - To manipulate objects and move about.

**AI is a system that thinks like human beings.**
First we must have some way of determining how humans think. We need to get inside the workings of the human minds. Once we have a sufficiently precise theory of the mind, it becomes possible to express that theory using a computer program.

The field of cognitive science brings together computer models from AI and experimental techniques from psychology to try to construct precise and testable theories of the workings of the human mind.

**AI is a system that thinks rationally**

For a given set of correct premises, it is possible to yield new conclusions. For eg. “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. This resulted in a field called logic.

A precise notation for the statements about all kinds of things in the world and about relations among them are developed. Programs exist that could in principle solve any solvable problem described in logical notation.

There are 2 main obstacles to this approach. First it is not easy to take informal knowledge and state it in the formal terms required by logical notation. Second, there is a big difference between being able to solve a problem “in principle” and doing so in practice.

**AI is a system that acts rationally**

An agent is something that acts. A rational agent is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome.

We need the ability to represent knowledge and reason with it because this enables us to reach good decisions in a wide variety of situations. We need to be able to generate comprehensive sentences in natural language because saying those sentences helps us get
by in a complex society. We need learning because having a better idea of how the world works enables us to generate more effective strategies for dealing with it. We need visual perception to get a better idea of what an action might achieve.

**AI application areas** *(AI by Luger)*

The 2 most fundamental concerns of AI researchers are knowledge representation and search.

**Knowledge representation**

It addresses the problem of capturing the full range of knowledge required for intelligent behavior in a formal language, i.e. One suitable for computer manipulation. Eg. predicate calculus, LISP, Prolog

**Search**

It is a problem solving technique that systematically explores a space of problem states, i.e, successive and alternative stages in the problem solving process.
The following explains the major application areas of AI.

Game playing

Much of the early research in AI was done using common board games such as checkers, chess and the 15 puzzle. Board games have certain properties that made them ideal for AI research. Most games are played using a well defined set of rules. This makes it easy to generate the search space. The board configuration used in playing these games can be easily represented on a computer. As games can be easily played, testing a game playing program presents no financial or ethical burden.

Heuristics

Games can generate extremely large search spaces. So we use powerful techniques called heuristics to explore the problem space. A heuristic is a useful but potentially fallible problem strategy, such as checking to make sure that an unresponsive appliance is plugged in before assuming that it is broken.

Since most of us have some experience with these simple games, we do not need to find and consult an expert. For these reasons games provide a rich domain for the study of heuristic search.

**Automated reasoning and theorem proving**

Examples for automatic theorem provers are Newell and Simon’s Logic Theorist, General Problem Solver (GPS).

Theorem proving research is responsible for the development of languages such as predicate calculus and prolog.

The attraction of automated theorem proving lies in the rigor and generality of logic. A wide variety of problems can be attacked by representing the problem description as logical axioms and treating problem instances as theorems to be proved.

Reasoning based on formal mathematical logic is also attractive. Many important problems such as design and verification of logic circuits, verification of the correctness of computer programs and control of complex systems come in this category.
Expert systems

Here comes the importance of domain specific knowledge. A doctor, for example, is effective at diagnosing illness because she possesses some innate general problem solving skill; she is effective because she knows a lot about medicine. A geologist is effective at discovering mineral deposits.

Expert knowledge is a combination of theoretical understanding of the problem and a collection of heuristic problem solving rules that experience has shown to be effective in the domain. Expert systems are constructed by obtaining this knowledge from a human expert and coding it into a form that a computer may apply to similar problems.

To develop such a system, we must obtain knowledge from a human domain expert. Examples for domain experts are doctor, chemist, geologist, engineer etc.. The domain expert provides the necessary knowledge of the problem domain. The AI specialist is responsible for implementing this knowledge in a program. Once such a program has been written, it is necessary to refine its expertise through a process of giving it example problems to solve and making any required changes or modifications to the program’s knowledge.

Dendral is an expert system designed to infer the structure of organic molecules from their chemical formulas and mass spectrographic information about the chemical bonds present in the molecules.

Mycin is an expert system which uses expert medical knowledge to diagnose and prescribe treatment for spinal meningitis and bacterial infections of the blood.

Prospector is an expert system for determining the probable location and type of ore deposits based on geological information about a site.

Internist is an expert system for performing diagnosis in the area of internal medicine.

The dipmeter advisor is an expert system for interpreting the results of oil well drilling logs.

Xcon is an expert system for configuring VAX computers.
Natural language understanding and semantic modeling

One goal of AI is the creation of programs that are capable of understanding and generating human language. Systems that can use natural language with the flexibility and generality that characterize human speech are beyond current methodologies.

Understanding natural language involves much more than parsing sentences into their individual parts of speech and looking those words up in a dictionary. Real understanding depends on extensive background knowledge.

Consider for example, the difficulties in carrying out a conversation about baseball with an individual who understands English but knows nothing about the rules of the game. This person will not be able to understand the meaning of the sentence. “With none down in the top of the ninth and the go ahead run at second, the manager called his relief from the bull pen”. Even though half of the words in the sentence may individually understood, this sentence would be difficult to even the most intelligent non base ball fan.

The task of collecting and organizing this background knowledge in such a way that it may be applied to language comprehension forms the major problem in automating natural language understanding.

Modeling human performance

We saw that human intelligence is a reference point in considering artificial intelligence. It does not mean that programs should pattern themselves after the organization of the human mind. Programs that take non human approaches to solving problems are often more successful than their human counterparts. Still, the design of systems that explicitly model some aspect of human performance has been a fertile area of research in both AI and psychology.
**Planning and robotics**

Research in planning began as an effort to design robots that could perform their tasks with some degree of flexibility and responsiveness to outside world. Planning assumes a robot that is capable of performing certain atomic actions.

Planning is a difficult problem because of the size of the space of possible sequences of moves. Even an extremely simple robot is capable of generating a vast number of potential move sequences.

One method that human beings use in planning is hierarchical problem decomposition. If we are planning a trip to London, we will generally treat the problems of arranging a flight, getting to the airport, making airline connections and finding ground transportation in London separately. Each of these may be further decomposed into smaller sub problems.

Creating a computer program that can do the same is a difficult challenge.

A robot that blindly performs a sequence of actions without responding to changes in its environment cannot be considered intelligent. Often, a robot will have to formulate a plan based on the incomplete information and correct its behavior. A robot may not have adequate sensors to locate all obstacles in the way of a projected path. Organizing plans in a fashion that allows response to changing environmental conditions is a major problem for planning.

**Languages and environments for AI**

Programming environments include knowledge structuring techniques such as object oriented programming and expert systems frameworks. High level languages such as Lisp and Prolog support modular development.

Many AI algorithms are also now built in more traditional computing languages such as C++ and Java.

**Machine learning**
An expert system may perform extensive and costly computations to solve a problem. But if it is given the same or similar problem a second time, it usually does not remember the solution. It performs the same sequence of computations again. This is not the behavior of an intelligent problem solver.

The programs must learn on their own. Learning is a difficult area. But there are several programs that suggest that it is possible.

One program is AM, the automated mathematician which was designed to discover mathematical laws. Initially given the concepts and axioms of set theory, AM was able to induce important mathematical concepts such as cardinality, integer arithmetic and many of the results of number theory. AM conjectured new theorems by modifying its current knowledge base.

Early work includes Winston’s research on the induction of structural concepts such as “arch” from a set of examples in the blocks world.

The ID3 algorithm has proved successful in learning general patterns from examples.

Meta dendral learns rules for interpreting mass spectrographic data in organic chemistry from examples of data on compounds of known structure.

Teiresias, an intelligent front end for expert systems, converts high level advice into new rules for its knowledge base.

There are also now many important biological and sociological models of learning.

**Neural nets and genetic algorithms**

An approach to build intelligent programs is to use models that parallel the structure of neurons in the human brain.

A neuron consists of a cell body that has a number of branched protrusions called dendrites and a single branch called the axon. Dendrites receive signals from other neurons. When these combined impulses exceed a certain threshold, the neuron fires and an impulse or spike passes down the axon.
This description of the neuron captures features that are relevant to neural models of computation. Each computational unit computes some function of its inputs and passes the result along to connected units in the network; the final results are produced by the parallel and distributed processing of this network of neural connection and threshold weights.

**Example problems**

Problems can be classified as toy problems and real world problems.

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

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

**Toy problems**

The first example is the
The agent is in one of the 2 locations, each of which might or might not contain dirt.

Any state can be designated as the initial state.

After trying these actions (Left, Right, Suck), we get another state.

The goal test checks whether all squares are clean.

8 – puzzle problem

It consists of a 3 X 3 board with 8 numbered tiles and a blank space as shown below.
A tile adjacent to the blank space can slide into the space. The aim is to reach a specified goal state, such as the one shown on the right of the figure.

8 – Queens problem

The goal of 8-queens’ problem is to place 8 queens on a chess board such that no queen attacks any other.

(a queen attacks any piece in the same row, column or diagonal). Figure shows an attempted solution that that fails: the queen in the rightmost column is attacked by the queen at the top left.
Real world problems

Route finding problem

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.

Touring problems

They are related to route finding problems. For example, consider the figure.

Consider the problem. ‘Visit every city in the figure at least once starting and ending in Palai’. Each state must include not just the current location but also the set of cities the agent has visited.
Traveling salesperson problem (TSP)

Is a touring problem in which each city must be visited exactly once. The aim is to find the shortest tour.

VLSI layout problem

It requires positioning of millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield.

Robot navigation

It 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.

Automatic assembly sequencing of complex objects by a robot

The assembly of intricate objects such as electric motors is economically feasible. In assembly problems, the aim is to find an order in which to assemble the parts of some object. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done.

Protein design

Here the aim is to find a sequence of amino acids that will fold into a three-dimensional protein with the right properties to cure some disease.

Internet searching

It means looking for answers to questions for related information or for shopping details. Software robots are being developed for performing this internet searching.
We have seen different kinds of problems with which AI is typically concerned. To build a system to solve a particular problem, we need to do 4 things.

1. Define the problem precisely. This includes the initial state as well as the final goal state.
2. Analyze the problem.
3. Isolate and represent the knowledge that is needed to solve the problem.
4. Choose the best problem solving technique and apply it to the particular problem.

Suppose we are given a problem statement “play chess”. This now stands as a very incomplete statement of the problem we want solved.

To build a program that could “play chess”, we could first have to specify the starting position of the chess board, the rules that define the legal moves and the board positions that represent a win for one side or the other.

For this problem “play chess”, it is easy to provide a formal and complete problem description. The starting position can be described as an 8 by 8 array as follows.
We can define as our goal any board position in which the opponent does not have a legal move and his or her king is under attack.

The legal moves provide the way of getting the initial state to a goal state. They can be described easily as a set of rules consisting of 2 parts: a left side that serves as a pattern to be matched against the current board position and a right side that describes the change to be made to the board position to reflect the move.

We have defined the problem of playing chess as a problem of moving around in a state space, where each state corresponds to a legal position of the board. We can then play
chess by starting at an initial state, using a set of rules to move from one state to another, and attempting to end up in one of a set of final states.

This state space representation is useful for naturally occurring, less well structured problems. This state space representation forms the basis of most of the AI methods.

Example 2

The game of tic-tac-toe

Starting with an empty board, the first player may place an X in any one of nine places. Each of these moves yields a different board that will allow the opponent 8 possible responses and so on. We can represent this collection of possible moves and responses by regarding each board configuration as a node in a graph. The links of the graph represent legal moves from one board configuration to another. These nodes correspond to different states of the game board. The resulting structure is called a state space graph.
Example 3

Diagnosing a mechanical fault in an automobile

Here each node of the state space graph represents a partial knowledge about the automobile’s mechanical problems.

The starting node of the graph is empty, indicating that nothing is known about the cause of the problem. Each of the states in the graph has arcs that lead to states representing further accumulation of knowledge in the diagnostic process.
For example the engine trouble node has arcs to nodes labeled ‘engine starts’ and ‘engine won’t start’. From the ‘won’t start’ node, we may move to nodes labeled ‘turns over’ and ‘won’t turn over’. The ‘won’t turn over’ node has arcs to nodes labeled ‘battery ok’.

Example 3

**Water jug problem**

We are given 2 jugs, a 4 gallon and a 3 gallon one. Neither has any measuring markers on it. There is a pump that can be used to fill the jugs with water. We need to get exactly 2 gallons of water in to the 4 gallon jug.

The state space for this problem can be described as the set of ordered pairs of integers \( (x,y) \), such that \( x=0,1,2,3 \) or 4 and \( y=0,1,2 \) or 3.; \( x \) represents the number of gallons of water in the 4 gallon jug, and \( y \) represents the quantity of water in the 3 gallon jug.

The start state is \((0,0)\). The goal state is \((2,n)\) for any value of \(n\).
In order to provide a formal description of a problem, we must do the following.

1. Define a state space that contains all the possible configurations of the relevant objects.

2. Specify one or more states within that space that describe possible situations from which the problem solving process may start. These states are called the initial states.

3. Specify one or more states that would be acceptable as solutions to the problem. These states are called goal states.

4. Specify a set of rules that describe the actions available.

Problem characteristics

(AI by Ritchie & Knight)
In order to choose the most appropriate method for a particular problem, it is necessary to analyze the problem along several dimensions.

- Is the problem decomposable into a set of independent smaller or easier sub problems?

- Can solution steps be ignored or at least undone if they prove unwise?

- Is the problem’s universe predictable?

- Is a good solution to the problem obvious without comparison to all other possible solutions?

- Is the desired solution a state of the world or a path to a state?

- Is a large amount of knowledge absolutely required to solve the problem, or is knowledge important only to constrain the search?

- Can a computer that is simply given the problem return the solution, or will the solution of the problem require interaction between the computer and a person?

**Is the problem decomposable?**

Suppose we want to solve the problem of computing the expression

\[
\int (x^2 + 3x + \sin^2 x \cdot \cos^2 x) \, dx
\]

We can solve this problem by breaking it down into 3 smaller problems, each of which can then solve by using a small collection of specific rules.
\[ \int x^2 + 3x + \sin^2 x \cos^2 x \, dx \]

\[ \int x^2 \, dx \qquad \int 3x \, dx \qquad \int \sin^2 x \cos^2 x \, dx \]

\[ \frac{x^3}{3} \quad 3\int x \, dx \quad \int (1-\cos^2 x) \cos^2 x \, dx \]

\[ \frac{3x^2}{2} \quad \int \cos^2 x \, dx \quad -\int \cos^4 x \, dx \]

\[ \int \frac{1}{2} (1 + \cos 2x) \, dx \]

\[ \frac{1}{2} \int 1 \, dx \quad \frac{1}{2} \int \cos 2x \, dx \]

\[ \frac{1}{2} x \quad \frac{1}{4} \sin 2x \]

**Can solution steps be ignored or undone?**

Here we can divide problems into 3 classes.

**Ignorable**, in which solution steps can be ignored.

**Recoverable**, in which solution steps can be undone.

**Irrecoverable**, in which solution steps cannot be undone.
Ignorable problems (e.g., Theorem proving)

Here solution steps can be ignored.

Suppose we are trying to prove a mathematical theorem. We proceed by first proving a lemma that we think will be useful. Eventually we realize that the lemma is no help at all. Here the different steps in proving the theorem can be ignored. Then we can start from another rule. The former can be ignored.

Recoverable problems (e.g., 8 puzzle problem)

Consider the 8 puzzle problem.

The goal is to transform the starting position into the goal position by sliding the tiles around.

In an attempt to solve the 8-puzzle, we might make a stupid move. For example, in the game shown above, we might start by sliding tile 5 into the empty space. Having done that, we cannot change our mind and immediately slide tile 6 into the empty space since the empty space will essentially have moved. But we can backtrack and undo the 1st
move, sliding tile 5 back to where it was. Then we can move tile 6. Here mistakes can be recovered.

Irrecoverable problems (eg. Chess)

Consider the problem of playing chess. Suppose a chess playing program makes a stupid move and realizes it a couple of moves later. It cannot simply play as though it had never made the stupid move. Nor can it simply back up and start the game over from that point. All it can do is to try to make the best of the current situation and go from there.

The recoverability of a problem plays an important role in determining the complexity of the control structure necessary for the problem’s solution. Ignorable problems can be solved using a simple control structure. Recoverable problems can be solved by a slightly more complicated control strategy that does sometimes make mistakes. Irrecoverable problems will need to be solved by a system that expends a great deal of effort making each decision since the decision must be final.

Is the universe predictable?

Certain outcome problems (eg. 8 puzzle)

Suppose we are playing with the 8 puzzle problem. Every time we make a move, we know exactly what will happen. This means that it is possible to plan an entire sequence of moves and be confident that we know what the resulting state will be.

Uncertain outcome problems (eg. Bridge)

However, in games such as bridge, this planning may not be possible. One of the decisions we will have to make is which card to play on the first trick. What we would like to do is to plan the entire hand before making that first play. But now it is not possible to do such planning with certainty since we cannot know exactly where all the cards are or what the other players will do on their turns.
One of the hardest types of problems to solve is the irrecoverable, uncertain outcome. Examples of such problems are
Playing bridge,
Controlling a robot arm,
Helping a lawyer decide how to defend his client against a murder charge.

Is a good solution absolute or relative?
Any path problems
Consider the problem of answering questions based on a database of simple facts, such as the following.
1. Marcus was a man.
2. Marcus was a Pompean.
3. Marcus was born in 40 A.D.
4. All men are mortal.
5. All Pompeans died when the volcano erupted in 79 A.D.
6. No mortal lives longer than 150 years.
7. It is now 1991 A.D.
Suppose we ask the question. “Is Marcus alive?”. By representing each of these facts in a formal language, such as predicate logic, and then using formal inference methods, we can fairly easily derive an answer to the question. The following shows 2 ways of deciding that Marcus is dead.

axioms

1. Marcus was a man. 1
4. All men are mortal. 4
8. Marcus is mortal. 1,4
3. Marcus was born in 40 A.D. 3
7. It is now 1991 A.D. 7
9. Marcus’ age is 1951 years. 3,7
6. No mortal lives longer than 150 years. 6
10. Marcus is dead.  

OR

7. It is now 1991 A.D.  
5. All Pompeans died in 79 A.D.  
11. All Pompeans are dead now.  
2. Marcus was a Pompeian.  
12. Marcus is dead.  

Since all we are interested in is the answer to the question, it does not matter which path we follow.

Best path problems (eg. Traveling salesman problem)

Consider the traveling salesman problem. Our goal is to find the shortest route that visits each city exactly once. Suppose the cities to be visited and the distances between them are shown below.

<table>
<thead>
<tr>
<th></th>
<th>Boston</th>
<th>NY</th>
<th>Miami</th>
<th>Dallas</th>
<th>SF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td></td>
<td>250</td>
<td>1450</td>
<td>1700</td>
<td>3000</td>
</tr>
<tr>
<td>NY</td>
<td>250</td>
<td></td>
<td>1200</td>
<td>1500</td>
<td>2900</td>
</tr>
<tr>
<td>Miami</td>
<td>1450</td>
<td>1200</td>
<td></td>
<td>1600</td>
<td>3300</td>
</tr>
<tr>
<td>Dallas</td>
<td>1700</td>
<td>1500</td>
<td>1600</td>
<td></td>
<td>1700</td>
</tr>
<tr>
<td>SF</td>
<td>3000</td>
<td>2900</td>
<td>3300</td>
<td>1700</td>
<td></td>
</tr>
</tbody>
</table>

One place the salesman could start is Boston. In that case, one path that might be followed is the one shown below which is 8850 miles long.
But is this the solution to the problem? The answer is that we cannot be sure unless we also try all other paths to make sure that none of them is shorter.

Best path problems are computationally harder than any path problems.

Is the solution a state or a path?

Problems whose solution is a state of the world.
eg. Natural language understanding

Consider the problem of finding a consistent interpretation for the sentence,
‘The bank president ate a dish of pasta salad with the fork’.

There are several components of this sentence, each of which, in isolation, may have more than one interpretation. Some of the sources of ambiguity in this sentence are the following.

The word ‘bank’ may refer either to a financial institution or to a side of a river.
The word dish is the object of the verb ‘eat’. It is possible that a dish was eaten. But it is more likely that the pasta salad in the dish was eaten.
Pasta salad is a salad containing pasta. But there are other ways interpretations can be formed from pairs of nouns. For example, dog food does not normally contain dogs.
The phrase ‘with the fork’ could modify several parts of the sentence. In this case, it modifies the verb ‘eat’. But, if the phrase had been ‘with vegetables’, then the modification structure would be different.

Because of the interaction among the interpretations of the constituents of this sentence, some search may be required to find a complete interpretation for the sentence. But to solve the problem of finding the interpretation, we need to produce only the interpretation itself. No record of the processing by which the interpretation was found is necessary.

Problems whose solution is a path to a state?

Eg. Water jug problem

In water jug problem, it is not sufficient to report that we have solved the problem and that the final state is (2,0). For this kind of problem, what we really must report is not the final state, but the path that we found to that state.

What is the role of knowledge?
Problems for which a lot of knowledge is important only to constrain the search for a solution.

Eg. Chess

Consider the problem of playing chess. How much knowledge would be required by a perfect chess playing program? Just the rules for determining the legal moves and some simple control mechanism that implements an appropriate search procedure.

Problems for which a lot of knowledge is required even to be able to recognize a solution.

Eg. News paper story understanding

Consider the problem of scanning daily newspapers to decide which are supporting democrats and which are supporting the republicans in some upcoming election. How much knowledge would be required by a computer trying to solve this problem? Here a great deal of knowledge is necessary.

Does the task require interaction with a person?

Solitary problems

Here the computer is given a problem description and produces an answer with no intermediate communication and with no demand for an explanation for the reasoning process.

Consider the problem of proving mathematical theorems. If

1. All we want is to know that there is a proof.
2. The program is capable of finding a proof by itself.

Then it does not matter what strategy the program takes to find the proof.

Conversational problems
In which there is intermediate communication between a person and the computer, either to provide additional assistance to the computer or to provide additional information to the user.

Eg. Suppose we are trying to prove some new, very difficult theorem. Then the program may not know where to start. At the moment, people are still better at doing the high level strategy required for a proof. So the computer might like to be able to ask for advice. To exploit such advice, the computer’s reasoning must be analogous to that of its human advisor, at least on a few levels.

**Problem solving by searching**

We have seen problems and problem spaces. Search is a problem solving technique that systematically explores a space of problem states. That is to move through the problem space until a path from an initial state to a goal state is found.

Examples of problem states might include the different board configurations in a game or intermediate steps in a reasoning process. This space of alternate solutions is then searched to find a final answer.

We can contrast this with human problem solving. Humans generally consider a number of alternative strategies on their way to solving a problem. A chess player typically considers a number of alternative moves, selecting the best according to such criteria as the opponent’s possible responses. A mathematician will choose from a different but equally complex set of strategies to find a proof for a difficult theorem, a physician may systematically evaluate a number of possible diagnoses and so on.

Eg. Game of tic-tac-toe

Given any board situation, there is only a finite set of moves that a player can make. Starting with an empty board, the first player may place an X in any one of 9
places. Each of these moves yields a different board that will allow the opponent 8 possible responses, and so on. We can represent this collection of possible moves and responses as a state space graph.

Given this representation, an effective game strategy will search through the graph for the paths that lead to the most wins and fewest losses and play in a way that always tries to force the game along one of the optimal paths. This searching strategy is an effective one and also it is straightforward to implement on a computer.
Searching strategies

Breadth first search (AI by Luger)

Consider the graph shown below.

States are labeled (A, B, C…).
Breadth first search explores the space in a level by level fashion. Only when there are no more states to be explored at a given level does the algorithm move on to the next level. A breadth first search of the above graph considers the states in the order A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U.

We implement breadth first search using lists, open and closed, to keep track of progress through the state space. ‘open’ lists states that have been generated but whose children have not been examined. The order in which states are removed from open determines the order of the search. ‘closed’ records states that have already been examined.

```c
void breadth_first_search ()
{
    open = [ start ];
    closed = [];
    while ( open not empty )
    {
        Remove the leftmost state from open, call it X;
        if X is a goal,
            then return SUCCESS;
        else
            {
                Generate children of X;
                Put X on closed;
                Discard children of x, if already on open or closed;
                Put remaining children on right end of open;
            }
    }
    return FAIL;
}
```
Child states are generated by inference rules, legal moves of a game or other state transition operators. Each iteration produces all children of the state $x$ and adds them to open.

Note that open is maintained as a queue (FIFO) data structure. States are added to the right of the list and removed from the left.

A trace of the breadth first search on the graph appears below.

Open | closed
--- | ---
A | empty
BCD | A
CDEF | BA
DEFGH | CBA
EFGHIJ | DCBA
FGHIJKL | EDCBA
GHJKLM | FEDCBA
HIJKLMN | GFEDCBA

And so on until open = [ ].

Because breadth first search considers every node at each level of the graph before going deeper in to the space, all states are first reached along the shortest path from the
start state. Breadth first search is therefore guaranteed to find the shortest path from the start state to goal.

**Depth first search**

Depth first search goes deeper into the search space whenever this is possible.

Consider the graph

In depth first search, the descendent states are added and removed from the left end of open. That is, open is maintained as a stack (LIFO) data structure.

```c
void depth_first_search ()
{
    open = [start];
    closed = [ ];
    while (open not empty )
    {
        remove leftmost state from open, call it X;
        if X is a goal state
            then return SUCCESS;
        else
        {
            generate children of X;
            put X on closed;
            discard children of X, if already on open or closed;
            put remaining children on left end of open;
        }
    }
```
A trace of depth first search on the above graph is shown below.

<table>
<thead>
<tr>
<th>open</th>
<th>closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>empty</td>
</tr>
<tr>
<td>B C D</td>
<td>A</td>
</tr>
<tr>
<td>E F C D</td>
<td>B A</td>
</tr>
<tr>
<td>K L F C D</td>
<td>E B A</td>
</tr>
<tr>
<td>S L F C D</td>
<td>K E B A</td>
</tr>
<tr>
<td>L F C D</td>
<td>S K E B A</td>
</tr>
<tr>
<td>T F C D</td>
<td>L S K E B A</td>
</tr>
<tr>
<td>F C D</td>
<td>T S K E B A</td>
</tr>
<tr>
<td>M C D</td>
<td>F T L S K E B A</td>
</tr>
<tr>
<td>C D</td>
<td>M F T L S K E B A</td>
</tr>
<tr>
<td>G H D</td>
<td>C M F T L S K E B A</td>
</tr>
</tbody>
</table>

And so on until open = [ ];

‘open’ records all states that are discovered and ‘closed’ contains all states that are already considered.

DFS is not guaranteed to find the shortest path to a state the first time that state is encountered.

Depth limited search

(AI by Russel)
to limit the depth of DFS, we can supply a predetermined depth limit, \( l \) to DFS. That is nodes at depth \( l \) are treated as if they have no successors. This approach is called depth limited search.

Depth first search can be viewed as a special case of depth limited search with \( l = \infty \).

Sometimes depth limit can be based on knowledge of the problem. For example, on the map of Romania, there are 20 cities. Therefore, we know that if there is a solution, it must be of length 19 at the longest, so \( l = 19 \) is a possible choice. But in fact if we studied the map carefully, we would discover that any city can be reached from any other city in at most 9 steps. This number, known as the diameter of the state space, gives us a better depth limit, which leads to a more efficient depth limited search.

**Uniform cost search**

(AI by Russell)

Breadth first search that we have learned assumes that all path costs from a state to a successor is same. It expands the shallowest unexpanded node.

But uniform costs search expands the node \( n \) with the lowest path cost, instead of expanding the shallowest node. Note that if all step costs are equal, this is identical to breadth first search.

Uniform cost search does not care about the number of steps a path has, but only about their total cost. Therefore, it will get stuck in an infinite loop if it ever expands a node that has a zero cost action leading back to the same state. We can guarantee
completeness provided the cost of every step is greater than or equal to some small positive constant. The algorithm expands nodes in order of increasing path cost.

**Bidirectional search**

(AI by Russell)

The idea behind bidirectional search is to run two simultaneous searches- one forward from the initial state and other backward from the goal, stopping when the 2 searches meet in the middle.

A schematic view of a bidirectional search that is about to succeed, when a branch from the start node meets a branch from the goal node.

Bidirectional search is implemented by having one or both of the searches check each node before it is expanded to see if it is in the fringe of the other search tree; if so, a solution has been found.

What do we mean by “the goal” in searching “backward from the goal”. For the 8-puzzle and for finding a route in Romania, there is just one goal state, so the backward search is very much like the forward search. If there are several explicitly listed goal states, then we can construct a new dummy goal state whose immediate predecessors are all the actual goal states.
The most difficult case for bidirectional search is when the goal test gives only an implicit description of some possibly large set of goal states. For example, all the states satisfying the “check mate” goal test in chess.

** Constraint satisfaction search

(AI by Ritchie and Knight)

Many problems in AI are called to be the problems of constraint satisfaction. In these problems, the goal is to discover some problem state that satisfies a given set of constraints.

Examples are cryptarithmetic puzzles, map coloring.

Crypt arithmetic problem

Here letters must be assigned particular numbers as their values. A constraint satisfaction approach to solving this problem avoids making guesses on particular assignments of numbers to letters until it has to. Instead, the initial set of constraints, which says that each number may correspond to only one letter and that the sums of the digits must be as they are given in the problem, is first augmented to include restrictions that can be inferred from the rules of arithmetic.

Constraint satisfaction is a search procedure that operates in a space of constraint sets. The initial state contains the constraints that are originally given in the problem description. A goal state is any state that has been constrained “enough”. For example, for cryptarithmetic, enough means that each letter has been assigned a unique numeric value.

Constraint satisfaction is a 2 step process. First constraints are discovered and propagated as far as possible throughout the system. Second, if there is still not a solution, search begins. A guess about something is made and added as a new constraint. Propagation can then occur with this new constraint, and so forth.
The first step, propagation, arises from the fact that there are usually dependencies among the constraints. So for example, assume we start with one constraint, \( N = E + 1 \). Then, if we added the constraint \( N = 3 \), we could propagate that to get a stronger constraint on \( E \), namely that \( E = 2 \).

Constraint propagation terminates for one of two reasons. First, a contradiction may be detected. If this happens, then there is no solution consistent with all the known constraints.

The second possible reason for termination is that the propagation has run out of stream and there are no further changes that can be made on the basis of current knowledge. If this happens, search is necessary to get the process moving again.

At this point, the second step begins. Some hypothesis about a way to strengthen the constraints must be made. In the case of the cryptarithmetic problem, for example, this usually means guessing a particular value for some letter. Once this has been done, constraint propagation can begin again from this new state. If a solution is found, it can be reported. If still more guesses are required, they can be made. If a contradiction is detected, then backtracking can be used to try a different guess and proceed with it.

Constraint satisfaction algorithm

1. propagate available constraints. To do this, first set OPEN to the set of all objects that must have values assigned to them in a complete solution. Then do until an inconsistency is detected or until OPEN is empty.
   a. Select an object OB from OPEN. Strengthen as much as possible the set of constraints that apply to OB.
   b. If this set is different from the set that was assigned the last time OB was examined or if this is the first time OB has been examined, then add to OPEN all objects that share any constraints with OB.
   c. Remove OB from OPEN.
2. If the union of the constraints discovered above defines a solution, then quit and report the solution.
3. If the union of the constraints discovered above defines a contradiction, then return failure.
4. If the neither of the above occurs, then it is necessary to make a guess at something in order to proceed. To do this, loop until a solution is found or all possible solutions have been eliminated.
   a. Select an object whose value is not yet determined and select a way of strengthening the constraints on that object.
   b. Recursively invoke constraint satisfaction with the current set of constraints augmented by the strengthening constraint just selected.

The following example describes the working of the above algorithm.
Consider the cryptarithmetic problem below.

Problem

\[
\begin{array}{cccc}
S & E & N & D \\
+ & M & O & R \\
\hline
M & O & N & E \ \\
\end{array}
\]

Initial state:
No two letters have the same value.
The sums of the digits must be as shown in the problem.

The goal state is a problem state in which all letters have been assigned a digit in such a way that all the initial constraints are satisfied.
The solution proceeds in cycles. At each cycle, 2 significant things are done.
1. Constraints are propagated by using rules that correspond to the properties of arithmetic.

2. A value is guessed for some letter whose value is not yet determined.

In the first step, it does not usually matter a great deal what order the propagation is done in, since all available propagations will be performed before the step ends. In the second step, though the order in which guesses are tried may have a substantial impact on the degree of search that is necessary. A few useful heuristics can help to select the best guess to try first.

For example, if there is a letter that has only 2 possible values and another with 6 possible values, there is a better chance of guessing right on the first then on the second. Another useful heuristic is that if there is a letter that participates in many constraints then it is a good idea to prefer it to a letter that participates in a few. A guess on such a highly constrained letter will usually lead quickly either to a contradiction or to the generation of many additional constraints.

The result of the first few cycles of processing this example is shown below.

\[
\begin{align*}
S & \quad E & \quad N & \quad D \\
+ & \quad M & \quad O & \quad R & \quad E \\
\hline
 & \quad M & \quad O & \quad N & \quad E & \quad Y
\end{align*}
\]
Initial state

\[
\begin{align*}
M &= 1 \\
S &= 8 \text{ or } 9 \\
O &= 0 \text{ or } 1 \Rightarrow O = 0 \\
N &= E \text{ or } E + 1 \Rightarrow N = E + 1 \\
C2 &= 1 \\
N + R &> 8 \\
E &\neq 9
\end{align*}
\]

\[
\begin{align*}
E &= 2
\end{align*}
\]

\[
\begin{align*}
N &= 3 \\
R &= 8 \text{ or } 9 \\
2 + D &= Y \text{ or } 2 + D = 10 + Y
\end{align*}
\]

\[
\begin{align*}
2 + D &= Y \\
N + R &= 10 + E \\
R &= 9 \\
S &= 8
\end{align*}
\]

\[
\begin{align*}
C1 &= 0 \\
C1 &= 1
\end{align*}
\]

\[
\begin{align*}
2 + D &= 10 + Y \\
D &= 8 + Y \\
D &= 8 \text{ or } 9
\end{align*}
\]

\[
\begin{align*}
Y &= 0 \\
D &= 8 \\
Y &= 1
\end{align*}
\]
Initially, rules for propagating constraints generate the following additional constraints.

\[ M = 1, \]
Since 2 single digit numbers plus a carry cannot total more than 19.

\[ S = 8 \text{ or } 9, \]
Since \( S + M + C_3 > 9 \) (to generate the carry) and \( M = 1 \), \( S + 1 + C_3 > 9 \), so \( S + C_3 > 8 \) and \( C_3 \) is at most 1.

\[ O = 0, \]
Since \( S + M (1) + C_3 (\leq 1) \) must be at least 10 to generate a carry and it can be at most 1. But \( M \) is already 1, so \( O \) must be 0.

\[ N = E \text{ or } E + 1, \]
Depending on the value of \( C_2 \). But \( N \) cannot have the same value as \( E \). So \( N = E + 1 \) and \( C_2 \) is 1.

In order for \( C_2 \) to be 1, the sum of \( N + R + C_1 \) must be greater than 9, so \( N + R \) must be greater than 8.

\[ N + R \text{ cannot be greater than } 18, \text{ even with a carry in, so } E \text{ cannot be } 9. \]

At this point, let us assume that no more constraints can be generated. Then, to make progress from here, we must guess. Suppose \( E \) is assigned the value

Now the next cycle begins. The constraint propagator now observes that

\[ N = 3, \text{ since } N = E + 1. \]

\[ R = 8 \text{ or } 9, \text{ since } R + N(3) + C_1 (1 \text{ or } 0) = 2 \text{ or } 12. \]

But since \( N \) is already 3, the sum of these nonnegative numbers cannot be less than 3. thus \( R + 3 + (0 \text{ or } 1) = 12 \) and \( R = 8 \text{ or } 9. \)

\[ 2 + D = Y \text{ or } 2 + D = 10 + Y, \text{ from the sum in the rightmost column.} \]
Again, assuming no further constraints can be generated, a guess is required. Suppose C1 is chosen to guess a value for. If we try the value 1, then we eventually reach dead ends as shown in the figure. When this happens, the process will backtrack and try C1 = 0.

Module 2


**Heuristic search (Informed search)**

In order to solve many hard problems efficiently, it is necessary to construct a control structure that is no longer guaranteed to find the best answer but that will almost always find a very good answer. Here comes the idea of a heuristic. A heuristic is a technique that improves the efficiency of a search process.

Using good heuristics, we can hope to get good solutions to hard problems in less exponential time. A heuristic is a strategy for selectively searching a problem space. It guides our search along lines that have a high probability of success while avoiding wasted or apparently stupid efforts. Human beings use a large number of heuristics in problem solving. If you ask a doctor what could cause nausea and stomach pains, he might say it is “probably either stomach flu or food poisoning”. Heuristics are not fool proof. Even the best game strategy can be defeated, diagnostic tools developed by expert physicians sometimes fail; experienced mathematicians sometimes fail to prove a difficult theorem.

State space search gives us a means of formalizing the problem solving process, and heuristics allow us to infuse that formalism with intelligence.

There are two major ways in which domain specific, heuristic knowledge can be incorporated in to a rule based search procedure.

1. In the rules themselves. For example the rules for a chess playing system might describe not simply the set of legal moves but rather a set of sensible moves.
2. As a heuristic function that evaluates individual problem states and determines how desirable they are.
A heuristic function
It is a function that maps from problem state descriptions to measures of desirability, usually represented as numbers.

The following gives an example for evaluating a state with a heuristic function.

Eg. Consider the 8-puzzle problem.
The object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration.

Best first search (A* algorithm)
Constraint satisfaction search

The first two techniques, breadth first search and depth first search we have already learned.

![Start state](image1.png)  ![Goal state](image2.png)

The above figure shows the start state and goal state for the 8-puzzle.
In the above diagram, the start state and the first set of moves are shown.

Consider a heuristic function for evaluating each of the states. The number of tiles that are out of place in each state when it is compared with the goal state.

Consider the states b, c and d in the above diagram.

Compare the state b with the goal state. We will get the heuristic function value for b as 5.

In the same way, compare c and d with the goal state.

The heuristic function value for c is 3. The heuristic function value for d is 5.

This example demonstrates the use of a heuristic function to evaluate the states. Which aspects of the problem state are considered, how those aspects are evaluated, and the weights given to individual aspects are chosen in such a way that the value of the heuristic function at a given node in the search process gives as good an estimate as possible of whether that node is on the desired path to a solution.

Well designed heuristic functions can play an important part efficiently guiding a search process towards a solution. Sometimes very simple heuristic functions can provide a fairly good
estimate of whether a path is any good or not. In other situations, more complex heuristic functions should be employed.

The purpose of a heuristic function is to guide the search process in the most profitable direction by suggesting which path to follow first when more than one is available. The more accurately the heuristic function estimates the true merits of each node in the search graph, the more direct the solution process.

Let us consider heuristic functions in detail

The 8-puzzle was one of the earliest heuristic search problems.

Consider two heuristics for the 8 puzzle problem.

\( h_1 = \) the number of misplaced tiles

For the above figure, for the start state value of \( h_1 \) is 8.

\( h_2 = \) the sum of the distances of the tiles from their goal positions

For the above figure, for the start state the value of

\[ h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18. \]

The average solution cost for a randomly generated 8-puzzle instance is about 22 steps. The branching factor is about 3. (Branching factor means average number of successors of a state or the number of branches from a state).

Consider two heuristics for the 8 puzzle problem.

\( h_1 = \) the number of misplaced tiles

For the above figure, for the start state value of \( h_1 \) is 8.

\( h_2 = \) the sum of the distances of the tiles from their goal positions

For the above figure, for the start state the value of

\[ h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18. \]
One way to characterize the quality of a heuristic is the effective branching factor, \( b^* \).

If the total number of nodes generated by A* algorithm for a particular problem is \( N \), and the solution depth is \( d \), then \( b^* \) is the branching factor that a uniform tree of depth \( d \) would have to have in order to contain \( N + 1 \) nodes. Thus

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

For example if A* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92.

A well designed heuristic would have a value of \( b^* \) close to 1.

To see the effect of heuristic functions on the 8 puzzle problem, see the table below. Different instances of 8-puzzle problem are solved using iterative deepening search and A* search using \( h_1 \) and \( h_2 \).

<table>
<thead>
<tr>
<th>( d )</th>
<th>IDS</th>
<th>( A^* (h_1) )</th>
<th>( A^* (h_2) )</th>
<th>IDS</th>
<th>( A^* (h_1) )</th>
<th>( A^* (h_2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>6</td>
<td>6</td>
<td>2.45</td>
<td>1.79</td>
<td>1.79</td>
</tr>
<tr>
<td>4</td>
<td>112</td>
<td>13</td>
<td>12</td>
<td>2.87</td>
<td>1.48</td>
<td>1.45</td>
</tr>
<tr>
<td>6</td>
<td>680</td>
<td>20</td>
<td>18</td>
<td>2.73</td>
<td>1.34</td>
<td>1.30</td>
</tr>
<tr>
<td>8</td>
<td>6384</td>
<td>39</td>
<td>25</td>
<td>2.80</td>
<td>1.33</td>
<td>1.24</td>
</tr>
<tr>
<td>10</td>
<td>47127</td>
<td>93</td>
<td>39</td>
<td>2.79</td>
<td>1.38</td>
<td>1.22</td>
</tr>
<tr>
<td>12</td>
<td>3644035</td>
<td>227</td>
<td>73</td>
<td>2.78</td>
<td>1.42</td>
<td>1.24</td>
</tr>
</tbody>
</table>

The results show that \( h_2 \) is better than \( h_1 \), and is far better than iterative deepening search.

Inventing heuristic functions

We have seen two heuristic functions for the 8 puzzle problem, \( h_1 \) and \( h_2 \).

\( h_1 \) = the number of misplaced tiles
\( h_2 \) = the sum of the distances of the tiles from their goal positions

Also we found that \( h_2 \) is better. Is it possible for a computer to invent such heuristic functions mechanically? Yes, it is possible.
If the rules of the puzzle were changed so that a tile could move anywhere, then h1 would give the exact number of steps in the shortest solution.

Similarly if a tile could move one square in any direction, then h2 would give the exact number of steps in the shortest solution.

Suppose we write the definition of 8-puzzle problem as

A tile can move from square A to square B if

- A is horizontally or vertically adjacent to B and B is blank.

From this statement we can generate three statements.

a. A tile can move from square A to Square B if A is adjacent to B.

b. A tile can move from square A to square B if B is blank.

c. A tile can move from square A to square B.

From a, we can derive h2. This is because h2 would be a proper score if we move each tile to its destination.

From c we can derive h1. This is because h1 would be a proper score, if tiles could move to their intended destinations in one step.

A program called ABSOLVER can generate heuristics automatically from problem definitions.

If a collection of heuristics h1, h2, h3… hm is available for a problem, then which one we should choose? We would choose

\[ h(n) = \max \{ h1, h2, \ldots, hm \} \]

**Heuristic search techniques- Heuristic for constraint satisfaction problem**

Many of the problems that come in artificial intelligence are too complex to be solved by direct techniques. They must be attacked by appropriate search methods in association with whatever direct techniques are available to guide the search. In this topic, we will learn some general purpose search techniques. These methods are all varieties of heuristic search. These techniques form the core of most AI systems.

The following are some of the search strategies.
Depth first search, 
Breadth first search, 
Hill climbing, 

**Iterative deepening search**

Iterative deepening search (or iterative deepening depth first search) is a general strategy used in combination with depth first search. It uses a depth bound on depth first search. It does by gradually increasing the limit – first 0, then 1, then 2 and so on – until a goal is found.

Figure shows four iterations of iterative deepening search on a binary search tree. Here the solution is found on the 4th iteration.

Consider a state space graph shown below.

The iterative deepening search on the above graph generates states as given below.

States generated in the order

Limit = 0 A
Iterative deepening search performs a depth first search of the space with a depth bound of 1. If it fails to find a goal, it performs another depth first search with a depth bound of 2. This continues, increasing the depth bound by 1 at each iteration. At each iteration, the algorithm performs a complete depth first search to the current depth bound.

Algorithm
1. Set depth limit = 0.
2. Conduct a depth first search to a depth of depth limit. If a solution path is found, then return it.
3. Otherwise, increment depth limit by 1 and go to step 2.

Iterative deepening search continues the benefits of depth first and breadth first search. It is the preferred search method when there is a large search space and the depth of the solution is not known.

For example, a chess program may be required to complete all its moves within 2 hours. Since it is impossible to know in advance how long a fixed depth tree search will take, a program may find itself running out of time. With iterative deepening, the current search can be aborted at any time and the best move found by the previous iteration can be played. Previous iterations can provide invaluable move ordering constraints.

Hill climbing

Hill climbing strategies expand the current state in the search and evaluate its children. The best child is selected for further expansion; neither its siblings nor its parent is retained. Search halts when it reaches a state that is better than any of its children. Hill climbing is named for the strategy that might be used by an eager, but blind mountain climber: go uphill along the
steepest possible path until you can go no farther. Because it keeps no history, the algorithm cannot recover from failures of its strategy.

There are three various strategies for hill climbing. They are

Simple hill climbing,
Steepest ascent hill climbing and
Simulated annealing.

**Simple hill climbing**

The simplest way to implement hill climbing is as follows.

Algorithm

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise continue with the initial state as the current state.

2. Loop until a solution is found or until there are no new operators left to be applied in the current state:
   a. Select an operator that has not yet been applied to the current state and apply it to produce a new state.
   b. Evaluate the new state,
      i. If it is a goal state, then return it and quit.
      ii. If it is not a goal state, but it is better than the current state, then make it the current state.
      iii. If it is not better than the current state, then continue in the loop.

Example:
A problem is given. Given the start state as ‘a’ and the goal state as ‘f’.
Suppose we have a heuristic function $h(n)$ for evaluating the states. Assume that a lower value of heuristic function indicates a better state.
Here a has an evaluation value of 5. a is set as the current state.

Generate a successor of a. Here it is b. The value of b is 3. It is less than that of a. that means b is better than the current state a. So set b as the new current state.

Generate a successor of b. it is d. D has a value of 4. It is not better than the current state b (3). So generate another successor of b. it is e. It has a value of 6. It is not better than the current state b (3). Then generate another successor of b. We get k. It has an evaluation value of 2. k is better than the current state b (3). So set k as the new current state.

Now start hill climbing from k. Proceed with this, we may get the goal state f.

Steepest ascent hill climbing

A useful variation on simple hill climbing considers all the moves from the current state and selects the best one as the next state.

Algorithm
1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.

2. Loop until a solution is found or until a complete iteration produces no change to current state.
   a. Let SUCC be a state such that any possible successor of the current will be better than SUCC.
   b. For each operator that applies to the current state, do:
      i. Apply the operator and generate a new state.
      ii. Evaluate the new state. If it is a goal state, then return it and quit. If not, compare it to SUCC. If it is better, then set SUCC to this state. If it is not better, leave SUCC alone.
   c. if the SUCC is better than current state, then set current state to SUCC.

An example is shown below.

Consider a problem. Initial state is given as a. the final state is m. Let h(n) be a heuristic function for evaluating the states.
In this, all the child states (successors) of a are generated first. The state d has the least value of heuristic function. So d is the best among the successors of a. then d is better than the current state a, so make d as the new current state. Then start hill climbing from d.

Both basic and steepest ascent hill climbing may fail to find a solution. Either algorithm may stop not by finding a goal state but by getting to a state from which no better states can be generated. This will happen if the program has reached either a local maximum, a plateau or a ridge.
Here in problem solving, our aim is to find the global maximum.

A local maximum is a state that is better than all its neighbors but is not better than some other states farther away.

A plateau is a flat area of the search space in which a whole set of neighboring states have the same value. On a plateau, it is not possible to determine the best direction in which to move by making local comparisons.

A ridge is a special kind of local maximum. It is an area of the search space that is higher than surrounding areas and that itself has a slope.

There are some ways of dealing with these problems.

Backtrack to some earlier node and try going in a different direction. This is a fairly good way of dealing with local maxima.

Make a big jump in some direction to try to get to a new section of the search space. This is a good way of dealing with plateaus.

Apply 2 or more rules before doing the test. This corresponds to moving in several directions at once. This is a good way for dealing with ridges.
Simulated annealing

We have seen that hill climbing never makes a down hill move. As a result, it can stuck on a local maximum. In contrast, a purely random walk- that is, moving to a successor chosen uniformly at random from the set of successors- is complete, but extremely inefficient.

Simulated annealing combines hill climbing with a random walk. As a result, it yields efficiency and completeness.

Simulated annealing is a variation of hill climbing in which, at the beginning of the process, some downhill moves may be made. The idea is to do enough exploration of the whole space early on so that the final solution is relatively insensitive to the starting state. This should lower the chances of getting caught at a local maximum, a plateau or a ridge.

In metallurgy, annealing is the process in which metals are melted and then gradually cooled until some solid state is reached. It is used to tamper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to coalesce in to a low energy crystalline state.

Physical substances usually move from higher energy configurations to lower ones. But there is some probability that a transition to a higher energy state will occur. This probability is given by the function

\[ P = e^{-\Delta E / kT} \]

Where \( \Delta E \) is positive change in the energy level, \( T \) is the temperature and \( k \) is Boltzmann’s constant.

Physical annealing has some properties. The probability of a large uphill move is lower than the probability of a small one. Also, the probability that an uphill move will be made decreases as the temperature decreases. Large uphill moves may occur early on, but as the
process progresses, only relatively small upward moves are allowed until finally the process converges to a local maximum configuration.

If cooling occurs so rapidly, stable regions of high energy will form. If however, a slower schedule is used, a uniform crystalline structure which corresponds to a global minimum is more likely to develop. If the schedule is too slow, time is wasted.

These properties of physical annealing can be used to define the process of simulated annealing. In this process, $\Delta E$ represents change in the value of heuristic evaluation function instead of change in energy level. $k$ can be integrated into $T$. Hence we use the revised probability formula

$$P' = e^{-\Delta E / T}$$

The algorithm for simulated annealing is as follows.

Algorithm

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Initialise BEST-SO-FAR to the current state.
3. Initialize $T$ according to the annealing schedule.
4. Loop until a solution is found or until there are no new operators left to be applied in the current state.
   a. Select an operator that has not yet been applied to the current state and apply it to produce a new state.
   b. Evaluate the new state. Compute
      $$\Delta E = (\text{value of current}) - (\text{value of new state})$$
      - If the new state is a goal state, then return it and quit.
      - If it is not a goal state, but is better than the current state, then make it the current state. Also set BEST-SO-FAR to this new state.
If it is not better than the current state, then make it the current state with probability \( p' \) as defined above. This step is usually implemented by invoking a random number generator to produce a number in the range \([0, 1]\). If that number is less than \( p' \), then the move is accepted. Otherwise, do nothing.

c. Revise \( T \) as necessary according to the annealing schedule.

5. Return BEST- SO- FAR as the answer.

To implement this algorithm, it is necessary to maintain an annealing schedule. That is first we must decide the initial value to be used for temperature. The second criterion is to decide when the temperature of the system should be reduced. The third is the amount by which the temperature will be reduced each time it is changed.

In this algorithm, instead of picking the best move, it picks a random move. If the move improves the situation, it is always accepted. Otherwise the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the amount \( \Delta E \) by which the evaluation is worsened. The probability also decreases as the temperature \( T \) goes down. Thus bad moves are more likely to be allowed at the start when the temperature is high, and they become more unlikely as \( T \) decreases.

Simulated annealing was first used extensively to solve VLSI layout problems.

**Informed search**

These search strategies use problem specific knowledge for finding solutions.

**Best first search** (A* algorithm)
Best first search is a general informed search strategy. Here a node is selected for an expansion based on an evaluation function, \( f(n) \). Traditionally, the node with the lowest evaluation is selected for expansion. It is implemented using a priority queue.

Best first search combines breadth first and depth first search. That is to follow a single path at a time, but switch paths whenever some competing path looks more promising than the current one does.

Best first search uses 2 lists, open and closed. Open to keep track of the current fringe of search and closed to record states already visited.

Algorithm

```plaintext
function best_first_search ( )
{
    open = [start];
    closed = [ ];
    while (open not empty)
    {
        remove the left most state from open, call it X;
        If X = goal then return the path from start to X;
        Else
        {
            generate children of X;
            for each child of X do
            {
                case
                the child is not in open or closed :
                {
                    assign the child a heuristic value;
                    add the child to open;
                }
            }
            case
```
the child is already on open:
{
    if the child was reached by a shorter path
    then give the state on open the shorter path;
}

case
the child is already on closed :
{
    if the child was reached by a shorter path
    then
    {
        remove the state from closed;
        add the child to open;
    }

} /*end of for */

put X on closed;
reorder states on open by heuristic merit;
} /* end of else */
return FAIL;
}

Here open acts as a priority queue. Algorithm orders the states on open according to some heuristic estimate of their “closeness” to a goal. Each iteration of the loop considers the most promising state on the open list. At each iteration, best first search removes the first element from the open list. If it meets the goal conditions, the algorithm returns the solution path that led to the goal. Each state retains ancestor information to determine, if it had previously been reached by a shorter path and to allow the algorithm to return the final solution path.

If the first element on open is not a goal, the algorithm applies all matching production rules or operators to generate its descendents. If a child state is already on open or closed, the algorithm checks to make sure that the state records the shorter of the 2 partial solution paths.
Duplicate states are not retained. By updating the ancestor history of nodes on open and closed when they are rediscovered, the algorithm is more likely to find a shorter path to a goal.

Best first search applies a heuristic evaluation to the states on open, and the list is sorted according to the heuristic values of these states. This brings the best states to the front of open.

Eg.

Figure shows a state space with heuristic evaluations. Evaluations are attached to some of its states.

A trace of the execution of best first search on this graph appears below. Suppose P is the goal state.

1. Open
   A5
   closed
   empty

2. Evaluate A5
   B4, C4, D6
   A5

3. Evaluate B4
   C4, E5, F5, D6
   B4, A5

4. Evaluate C4
   H3, G4, E5, F5, D6
   C4, B4, A5

5. Evaluate H3
   O2, P3, G4, E5, F5, D6
   H3, C4, B4, A5

6. Evaluate O2
   P3, G4, E5, F5, D6
   O2, H3, C4, B4, A5

7. Evaluate P3
   the solution is found.
The best first search algorithm always selects the most promising state on open for further expansion. It does not abandon all other states but maintains them on open. In the event a heuristic leads the search down a path that proves incorrect, the algorithm will eventually retrieve some previously generated, next best state from open and shifts its focus to another part of the space. In the figure, after the children of state B were found to have poor heuristic evaluations the search shifted its focus to state c. the children of B were kept on open in case the algorithm needed to return them later.

**Implementing heuristic evaluation functions**

We need a heuristic function that estimates a state. We call this function $f'$. It is convenient to define this function as the sum of 2 components, $g$ and $h'$.
The function $g$ is the actual cost of getting from the initial state (a) to the current state (p). The function $h'$ is an estimate of the cost of getting from the current state (p) to a goal state (m). Thus

$$f' = g + h'$$

the function $f'$, then, represents an estimate of the cost of getting from the initial state (a) to a goal state (m) along the path that generated the current state (p).

if this function $f'$ is used for evaluating a state in the best first search algorithm, the algorithm is called A* algorithm.

We now evaluate the performance of a heuristic for solving 8 puzzle problem. Figure shows the start and goal states, along with the first 3 states generated in the search.
We consider a heuristic that counts the tiles out of place in each state when it is compared with the goal. The following shows the result of applying this heuristic to the 3 child states of the above figure.
The distance from staring state to its descendents (g) can be measured by maintaining a depth count for each state. This count is 0 for the beginning state and is incremented by 1 for each level of the search. It records the actual number of moves that have been used to go from the start state in a search to each descendent. The following shows the f values.
The best first search of 8 puzzle graph using f as defined above appears below.
Each state is labeled with a letter and its heuristic weight, $f'(n) = g(n) + h'(n)$. 
Open  closed
1       a4
2       c4, b6, d6       a4
3       c5, f5, b6, d6, g6   a4, c4
4       f5, h6, b6, d6, g6, i7   a4, c4, e5
5       j5, h6, b6, d6, g6, k7, i7   a4, c4, e5, f5
6       l5, h6, b6, d6, g6, k7, i7   a4, c4, e5, f5, j5
7       m5, h6, b6, d6, g6, n7, k7, i7   a4, c4, e5, f5, j5, l5
8       success, m= goal

Notice the opportunistic nature of best first search.

The g(n) component of the evaluation function gives the search move of a breadth first flavor. This prevents it from being misled by an erroneous evaluation; if a heuristic continuously returns ‘good’ evaluations for states along a path that fails to reach a goal, the g value will grow to dominate h and force search back to a shorter solution path. This guarantees that the algorithm will not become permanently lost, descending an infinite branch.

**A* algorithm**

The following shows an expanded version of the above best first search algorithm called A* algorithm.

1.open = [start];
   Set the start node’s g = 0;
   h’ value to whatever it is.
   f’ = g + h’ = 0 + h’;
   f’ = h’

closed = [ ];
2. Until a goal node is found, repeat the following procedure.

\{
    if (open = [ ] ) return failure;
else

    \{
        Pick the node on open with the lowest \( f^* \) value. Call it BESTNODE;
        Remove it from open; Place it on closed;
        If (BESTNODE is a goal node) then return success;
        Else

        \{
            Generate the successors of BESTNODE;
            for each SUCCESSOR

            \{
                a. set SUCCESSOR to point back to BESTNODE;
                    These backward links will make it possible to recover the
                    path once a solution is found.
                b. compute \( g(SUCCESSOR) = g(BESTNODE) + \) the cost of getting
                    from BESTNODE to SUCCESSOR.
                c. See if SUCCESSOR is same as any node on open. That is it has
                    already been generated but not processed. If so, call that node OLD.
                    Since the node already exists in the graph, we can throw SUCCESSOR
                    away and add OLD to the list of BESTNODE’s successors. Now we
                    must decide whether OLD’s parent link should be reset to point to
                    BESTNODE. It should be if the path we have just found to
                    SUCCESSOR is cheaper than the current best path to OLD. To see
                    whether it is cheaper to get to OLD via its current parent or to
                    SUCCESSOR via BESTNODE by comparing their g values. If OLD is
                    cheaper, then we need do nothing. If SUCCESSOR is cheaper, then
                    reset OLD’s parent link to point to BESTNODE, record the new
                    cheaper path in \( g(OLD) \), and update \( f'(OLD) \).
    \}
\}
d. If SUCCESSOR was not in open, see if it is in closed. If so, call the
node on closed OLD and add OLD to the list of BESTNODE’s
successors. Check to see if the new path or the old path is better just as
in step 2c, and set the parent link and g and f values appropriately. If
we have just found a better path to OLD, we must propagate the
improvement to OLD’s successors. This is a bit tricky. OLD points to
its successors. Each successor in turn points to its successors, and so
forth, until each branch terminates with a node that either is still on
open or has no successors. So to propagate the new cost downwards,
do a depth first traversal of the tree starting at OLD, changing each
node’s g value and also its f value, terminating each branch when you
reach either a node with no successors or a node to which an
equivalent or better path has already been found. This condition is easy
to check for. Each node’s parent link points back to its best known
parent. As we propagate down to a node, see if its parent points to the
node we are coming from. If so, continue the propagation. If not, then
its g value already reflects the better path of which it is part. So the
propagation may stop here. But it is possible that with the new value of
g being propagated downward, the path we are following may become
better than the path through the current parent. So compare the two. If
the path through the current parent is still better, stop the propagation.
If the path we are propagating through is now better, reset the parent
and continue propagation.

e. If successor was not already on either open or closed, then put it on
open, and add it to the list of BESTNODE’s successors. Compute f’
(successor) = g (successor) + h’ (successor).
Game playing

In many environments, there are multiple agents. In this, a given agent must consider the actions of other agents. If the agents are competing with each other, then such an environment is called a competitive environment. Competitive environments in which the agents goals are in conflict, results in problems known as games.

Consider the game of chess. If one player wins the game of chess, then utility function value is +1. in this the opposite player loses. His utility function value is -1. if the game ends in a draw, the utility function value is 0. Like this, in AI, games are turn taking, 2 player, zero sum games.

For AI researchers, the nature of games makes them an attractive subject for study. The state of a game is easy to represent and agents are limited to a small number of actions. Precise rules are there for making these actions.

By 1950, chess playing program was developed by Zuse, Shannon, Wiener and Turing. After that systems for playing checkers, Othello, Backgammon and Go were developed.

Games are interesting because they are too hard to solve. For example, chess has around $35^{100}$ states. Generating all these states is impossible. Therefore game playing research has brought a number of interesting ideas on how to make the best possible use of time.

Minimax search procedure

We will consider games with 2 players. The opponents in a game are referred to as MIN and MAX. MAX represents the player trying to win or to MAXimize his advantage. MIN is the opponent who attempts to MINimize MAX’s score. 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 kind of search problem with the following components.

The initial state, which identifies 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 determines when the game is over. States where the game has ended are called terminal states.

A utility function (also called an objective function) which gives a numeric value for the terminal states. In chess, the outcome is a win, loss or draw with values +1, -1 or 0. Some games have a wider variety of possible outcomes.

The initial state and the legal moves for each side define the game tree for the game. Figure below shows part of the game tree for tic-tac-toe.
The top node is the initial state, and MAX moves first, placing an X in an empty square. The figure shows part of the search tree giving alternating moves by MIN (O) and MAX (X), until we reach terminal states, which can be assigned utilities according to the rules of the game.

Play alternates between MAX’s placing an X and MIN’s placing an O until we reach leaf nodes corresponding to terminal states such that one player has there in a row or all the aquare are filled. 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 MAX’s job to use the search tree to determine the best move.
Search strategies

In a normal search problem, the best solution is a sequence of moves leading to a goal state. In a game, on the other hand, MIN has a role. MAX therefore must find a contingent strategy. We will see how to find this optimal strategy.

Even a simple game like tic-tac-toe is too complex for us to draw the entire game tree. So we will use a game tree as shown below.

The ∆ nodes are MAX nodes, in which it is MAX’s turn to move and the ▽ nodes are MIN nodes. The terminal states show the utility values for MAX.

The possible moves for MAX at the root node are labeled a1, a2, a3. the possible replies to a1 for MIN are b1, b2, b3 and so on. This game ends after one move each by MAX and MIN.

Given a game tree, the optimal strategy can be determined by examining the minimax value of each node, which we write as minimax-value (n). The minimax-value of a node is the utility for MAX of being in the corresponding state. The minimax-value of a terminal state is just
its utility. Given a choice, MAX will prefer to move to a state of maximum value, whereas MIN prefers a state of minimum value. So we have

Minimax-value (n) =
\[
\begin{cases}
    \text{Utility (n), if } n \text{ is a terminal state} \\
    \max_{s \in \text{successors (n)}} \min \text{minimax-value (s)}, & \text{If } n \text{ is a MAX node} \\
    \min_{s \in \text{successors (n)}} \text{minimax-value (s)}, & \text{If } n \text{ is a MIN node}
\end{cases}
\]

Let us apply these definitions to the above game tree. The terminal nodes on the bottom level are labeled with their utility values. The first MIN node, labeled B, has 3 successors with values 3, 12 and 8. So its minimax-value is 3. Similarly C has a minimax-value 2 and D has minimax-value 2. The root node is a MAX node; its successors have minimax values 3, 2 and 2. So it has a minimax value of 3.

That is

Minimax-value(A) = \max \left[ \min (3,12,8), \min (2,4,6), \min (14,5,2) \right]

= \max \left[ 3, 2, 2 \right] = 3

As a result, action a1 is the optimal choice for MAX because it leads to the successor with the highest minimax-value. This is the minimax decision at the root.
The definition of optimal play for MAX assumes that MIN also
plays optimally – it maximizes the worst outcome for MAX. Suppose MIN does not play
optimally. Then it is easy to show that MAX will do even better.

**The minimax algorithm**

The minimax algorithm given below computes the minimax
decision from the current state.

function minimax-decision (state )
returns an action
{
    v = max-value (state);
    return an action in successors (state) with value v;
}

function max-value (state)
returns a utility value
{
    if terminal-test (state) then
        return utility (state);

    v = - α ;

    for a,s in successors (state)
        { 
            v = max (v, min-value (s));
        }

    return v;
}

function min-value (state)
returns a utility value
{
    If terminal-test (state) then
        return utility (state);

    \[ v = + \alpha \];

    for a,s in successors (state)
    {
        \[ v = \min (v, \max-value (s)); \]
    }

    return v;
}

The above procedure uses a simple recursive computation of the minimax values of each successor state.

The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are backed up through the tree.

For example, for the above game tree, 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 for the root node.
Imperfect real time decisions

The minimax search searches the entire game tree, whereas the alpha beta algorithm helps us to prune large parts of it. But here also, alpha beta has to search all the way to terminal states for at least a portion of the search space. This is not practical. This is because moves in a 2 player game must be made within a reasonable amount of time.

It is proposed that programs should cut off the search earlier and apply a heuristic evaluation function to states in the search. Here the alpha beta search is altered in 2 ways. The utility function is replaced by a heuristic evaluation function EVAL. It gives an estimate of the position’s utility. The terminal test is replaced by a cut off test.

Evaluation functions

An evaluation function returns an estimate of the expected utility of the game from a given position. For centuries, chess players have developed ways of judging the value of a position. The performance of a game playing program is dependent on the quality of its evaluation function. How do we design good evaluation functions?

The computation of the evaluation function for a state must not take too long.

Consider the game of chess. If we are cutting off the search at non terminal states, the evaluation algorithm is uncertain about the final outcomes of those states.
Most evaluation functions work by calculating various features of the states. For example the number of pawns possessed by each side in a game of chess. Most evaluation functions compute separate numerical contributions from each feature and then combine them to find the total value.

For example, chess books give an approximate material value for each piece; 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 good. These feature values are then simply added up to obtain the evaluation of the position. This kind of evaluation function is called a weighted linear function because it is expressed as

$$\text{Eval (s)} = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

Where each $w_i$ is a weight and each $f_i$ is a feature of the position. For chess, the $f_i$ could be the numbers of each kind of piece on the board, and $w_i$ could be the values of the pieces. (1 for pawn, 3 for bishop etc.)

From this it is seen that the contribution of each feature is independent of the values of the other features. For example, we have given a value 3 to a bishop. It ignores the fact
that bishops are more powerful in the end game. Because of this, current programs for chess and other games also use non-linear combinations of features. For example, a pair of bishops might be worth more than twice the value of a single bishop, and a bishop is worth more in the end game than at the beginning.

Given the linear form of the evaluation, the feature and weights result in the best approximation to the true ordering of states by value.

**Cutting off search**

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.

We use the following line,

```
If cutoff-test (state, depth) then
    return eval (state);
```

Here instead of terminal-test (state), we use cut-off-test (state, depth). The approach is to control the amount of search to set a fixed depth limit \( d \). The depth \( d \) is chosen so that the amount of time used will not exceed what the rules of the game allow.

Again consider the game of chess. An approach is to apply iterative deepening. When time runs out, the program returns the move selected by the deepest completed search.

But this approach can lead to errors. Consider again the simple evaluation function for chess. Suppose the program searches to the depth limit, reaching the position shown below.
white to move

Here black is ahead by a knight and 2 pawns. It would generate the evaluation function value, and it declares that black is going to win. But from the figure, white’s next move will capture black queen. As a result, the position is really a won for white. But this can be seen only by looking ahead one more step.

Quiescent states

Hence, a more sophisticated cut off test is needed. The evaluation function should be applied only to quiescent positions. Quiescent position is a position which is unlikely to exhibit wild swings in value in the near future. Non-quiescent positions can be expanded further until quiescent positions are reached. This additional search is called quiescence search.

Horizon effect

Another problem that can occur is horizon effect. Consider the following board configuration.
Horizon effect arises when the program is facing a move by the opponent that causes serious damage. In the above state, black is ahead in material, but if white advances its pawn from the 7th row to the 8th, the pawn will become a queen and create an easy win for white.

The use of singular extensions has been quite effective in avoiding the horizon effect. A singular extension is a move that is clearly better than all other moves in a given position.

Forward pruning

It is possible to do forward pruning. It means that some moves at a given node are pruned immediately without further consideration. Most humans playing chess only consider a few moves from each position. This is dangerous because sometimes the best move will be pruned away. So normally this is not applied near root. Forward pruning can be used safely in special situations. For example when 2 moves are symmetric or equivalent, only one of them need be considered.

Combining all these techniques results in a program that can play chess. The branching factor for chess is about 35. If we used minimax search, we could look ahead only about 5 moves. Such a program can be defeated by an average human chess player who can plan 6 or 8 moves ahead.

With alpha beta search, we get to about 10 moves. To reach grandmaster status, we need an extensively tuned evaluation function. We need a super computer to run the program.

Alpha- Beta pruning

The problem with minimax search is that a large number of states need to be examined. We can effectively cut it in half using a technique called alpha beta pruning.

Here the trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree.

Consider again the game tree shown below.
Let us calculate the minimax value at a. One way is to simplify the formula for minimax value. Let 2 successors of node C have values x and y. The value of the root node is given by:

\[
\text{Minimax-value (A)} = \max \left[ \min (3, 12, 8), \min (2, x, y), \min (14, 5, 2) \right]
\]

Suppose minimum of x and y is z.

Then \( \text{minimax-value (A)} = \max \left[ 3, \min (2, x, y), 2 \right] \) if \( z \leq 2 \).

\[
= \max [3, \min (2, x, y), 2] \]

If \( z \leq 2 \).
= max [ 3, z, 2 ]

= 3

From this, it is clear that the value of the root node A and hence the minimax decision are independent of the values of the pruned leaves x and y.

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.

The general principle is this. Consider a node n somewhere in the tree; such that player has a choice of moving to that node. If player has a better choice m either at the parent node of n or at any choice further up, then n will never be reached in actual play.

Alpha beta pruning gets its name from the following 2 parameters, α and β.
The algorithm for alpha beta search is given below.

function alpha-beta-search (state)
    returns an action
    {
        v = max-value (state, -α, +α );
        return the action in successors (state) with value v;
    }

function max-value (state, α, β )
    returns a utility value
    {
        if terminal-test (state) then
            return utility (state);
        v = -α ;
        for a, s in successors (state)
            {
                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
    {
        if terminal-test (state) then
return utility (state);

v = +α ;
for a, s in successors (state)
{
    v = min (v, max-value (s, α, β) );
    if v <= α then
        return v;

    β = min (β, v);
}
return v;

\( \alpha \) – the value of the best (i.e. Highest value) choice we have found so far at any choice point along the path for MAX.

\( \beta \) – the value of the best (i.e. Lowest value) choice we have found so far at any choice point along the path for MIN.

alpha-beta search updates the values of \( \alpha \) and \( \beta \) as it goes along and prunes the remaining branches at a node as soon as the value of the current node is known to be worse than the current \( \alpha \) or \( \beta \) value for MAX or MIN, respectively.

**State of the art game programs**

Some researchers believe that game playing has no importance in main stream AI. But game playing programs continue to generate excitement and a steady stream of innovations that have been adopted by a wider community.

Chess
Deep blue program

In 1997, the Deep Blue program defeated world chess champion, Garry Kasparov. Deep Blue was developed by Campbell, Hsu and Hoane at IBM. The machine was a parallel computer with 30 IBM processors and 480 VLSI chess processors. Deep Blue used iterative deepening alpha beta search procedure. Deep Blue searched 126 million nodes per second on average. Search reached depth 14 routinely. The evaluation function had over 8000 features.

The success of Deep Blue shows that progress in computer game playing has come from powerful hardware, search extensions and good evaluation function.

Fritz

In 2002, the program Fritz played against world champion Vladimir Kramnik. The game ended in a draw. The hardware was an ordinary PC.

Checkers

Arthur Samuel of IBM, developed a checkers program. It learned its own evaluation function by playing itself thousands of times. In 1962, it defeated Nealy, a champion in checkers. Chinook was developed by Schaeffer. Chinook played against world champion Dr. Tinsley in 1990. Chinook won the game.

Othello

Is a popular computer game. It has only 5 to 15 moves. In 1997, the Logistello program defeated human world champion, Murakami.

Backgammon

Garry Tesauro developed the program TD-gammon. It is ranked among the top 3 players in the world.

Go

It is the most popular board game in Asia. The programs for playing Go are Geomate and Go4++.
Bridge

Bridge is a multiplayer game with 4 players. Bridge Baron program won the 1997 bridge championship.

GIB program won the 2000 championship.

Knowledge representation

Knowledge is very important in artificial intelligence systems. Knowledge is to be represented properly. It becomes clear that particular knowledge representation models allow for more powerful problem solving mechanisms that operate on them. Two of the knowledge representation schemes are semantic nets and frames.

Semantic nets

The following diagram shows a semantic network in which some knowledge is stored. Here it shows how some knowledge on cricket is represented.
Here boxed nodes represent objects and values of attributes of objects. These values can also be viewed as objects with attributes and values, and so on. The arrows on the lines point from an object to its value along the corresponding attribute line.

All of the objects and most of the attributes shown in this example have been chosen to correspond to the game of cricket. They have no general importance. The 2 exceptions to this are the attribute isa, which is being used to show class inclusion, and the attribute instance, which is being used to show class membership. Using this technique, the knowledge base can support retrieval of both of facts that have been explicitly stored and of facts that can be derived from those that are explicitly stored.

From the above semantic network, we can derive answers to the following questions,

Team (Dravid) = India

This attribute Dravid had a value stored explicitly in the knowledge base.

Batting average (Dravid) = 36

Since there is no value for batting average stored explicitly for Dravid, we follow the instance attribute to batsman and extract the value stored there.

Height (Akhtar) = 6-1

This represents another default inference. Notice here that because we get to it first, the more specific fact about the height of cricket players overrides a more general fact about the height of adult males.

Bats (Dravid) = right
To get a value for the attribute ‘bats’ required going up the isa hierarchy to the class ‘cricket player’. But what we found there was not a value but a rule for computing a value. This rule required another value as input. So the entire process must be begun again recursively to find a value for ‘handed’. This time it is necessary to go all the way up to ‘person’ to discover that the default value for handedness for people is ‘right’. Now the rule for ‘bats’ can be applied, producing the result ‘right’.

The main idea behind semantic nets is that the meaning of a concept comes from, the ways in which it is connected to other concepts. In a semantic net, information is represented as a set of nodes connected to each other by a set of labeled arcs, which represent relationships among the nodes.

Consider another example.

one of the early ways that semantic nets were used to find relationships among objects by spreading activation out from each of 2 nodes and seeing where the activation met. This process is called intersection search. Using this, it is possible to use the above semantic network to answer questions such as

“What is the connection between India and blue?”

Representing non-binary predicates
Some of the arcs from the above figure can be represented in logic as:

- isa (person, mammal)
- instance (Dravid, person)
- team (Dravid, India)
- uniform-color (Dravid, Blue)

The above are binary predicates.

Three or more place predicates can also be converted to a binary form by creating one new object representing the entire predicate statement and then introducing binary predicates to describe the relationship to this new object of each of the original arguments.

For example, suppose we know that:

score (Pakistan, India, 230-240)

This can be represented in a semantic net by creating a node to represent the specific game and then relating each of the 3 pieces of information to it.

This produces the semantic net shown below.
This technique is particularly useful for representing the contents of a sentence that describes several aspects of a particular event. The sentence

“John gave the book to Mary”.

This is represented as follows.

Additional points on semantic nets

There should be a difference between a link that defines a new entity and one that relates existing entities. Consider the semantic net

Both nodes represent objects that independently of their relationship to each other.

Suppose we want to represent the fact that “John is taller than Bill” using the net
The nodes H1 and H2 are new concepts representing John’s height and Bill’s height respectively.

We may use the following net to represent the fact

“John is 6 feet tall and that he is taller than Bill”.

The operations on these nets can exploit the fact that some arcs such as height, define new entities, while others, such as greater than and value describe relationships among existing entities.

**Partitioned semantic nets**

Consider the fact

“The dog bit the postman”.

---

greater than
Consider another fact

“Every dog has bitten a postman”.

This is a quantified expression. That is

\[ \forall x: \text{Dog}(x) \rightarrow \exists y: \text{postman}(y) \land \text{Bite}(x,y) \]

Here we need to partition the semantic net into a set of spaces, each of which corresponds to the scope of one or more variables.

To represent this fact, it is necessary to encode the scope of the universally quantified variable \( x \). This can be done using partitioning as shown below.
Node g is an instance of special class GS of general statements about the world. (ie. With universal quantifier, $\forall$. Every element of GS has at least 2 attributes.

1. A form which states the relation to be asserted.
2. One or more $\forall$ connections, one for each of the universally quantified variables.

In this example, there only one such universally quantified variable, ie d.

Consider another fact, 

“Every dog has bitten the postman”.

www.lectnote.blogspot.com
In this, node C representing the victim lies outside the form of the general statement. It is not viewed as an existentially quantified variable. Instead, it is interpreted as standing for a specific entity.

Consider another fact

“Every dog has bitten every postman”.

The semantic net for this is shown below.
In this case, g has 2 links, one pointing to d which represents any dog, and one pointing to m, representing any postman.

In the above net, space S1 is included in space SA.

Conclusion

The idea of semantic net is to simply represent labeled connections among entities. But as we expand the range of problem solving tasks, the semantic net begins to become more complex. As a result, it is necessary to assign more structure to nodes as well as to links. As a result, frames were evolved.

Frames

A frame is a collection of attributes (slots) and associated values that describe some entity in the world.

Consider a node in the semantic network below.

```
Cricket player
  isa : adult male
  bats : (EQUAL handed)
```

```
Adult male
  isa

Cricket player
  bats
  height
  batting average

6-1
48
```
height: 6 - 1
batting average: 48

There is so much flexibility in this frame representation, since this can be used to solve particular representation problems.

A single frame taken alone has not much use. Instead, we build frame systems out of collections of frames that are connected to each other. Frame systems are used to encode knowledge and support reasoning.

Consider the semantic network shown below.

```
person
    \--- handed \- Right
        \- isa
          \- Adult
              \- height \- 5 - 10
                  \- isa
                      \- Cricket player
                          \- height \- 6 - 1
                              \- bats avrg \- 58
                                  \- isa
                                      \- batsman
                                          \- batting avrg \- 36
                                              \- isa
                                                  \- bowler
                                                      \- instance
                                                          \- team
                                                              \- India
                                                                  \- Dravid
                                                                      \- Pakistan
                                                                          \- Akhtar
```

The frame system corresponding to the above semantic network is shown below.
isa : mammal
*handed : right

adult male
isa : person
*height : 5 – 10

cricket player
isa : adult male
*height : 6 – 1
*bats : (equal to) handed
*batting avrg : 48
*team :
*uniform color:

batsman
isa : cricket player
*batting avrg : 52

Dravid
instance : batsman
team : India

bowler
isa : cricket player
batting avrg : 40

Akhtar
instance : bowler
team : Pakistan
In this example, the frames ‘person’, ‘adult male’, ‘cricket player’, ‘bowler’, ‘batsman’ are all classes.

The frames ‘Dravid’, ‘Akhtar’, are instances.

Set theory provides a good basis for understanding frames. Here classes can be called as sets. Instances mean elements of a set.

The ‘isa’ relation is in fact a subset relation. The set of adultmales is a subset of all persons. The set of cricket players is a subset of the set of adult males and so on.

Our instance relation corresponds to the relation element of. ‘Dravid’ is an element of the set of ‘batsman’. Thus he is an element of all the supersets of ‘batsman’, including ‘cricket player’ and ‘person’.

There are 2 kinds of attributes associated with a class (set). They are attributes about the set itself and attributes that are to be inherited by each element of the set.

The attributes that are inherited by each element of the set is indicated by an asterisk (*). For example, consider the class cricket player. We have shown only one property of it as a set. It is a subset of the set of ‘adult male’. We have listed 5 properties that all cricket players have (height, bats, batting average, team and uniform color), and we have specified default values for the first three of them.

Consider the net shown below.
Here the distinction between a set and an instance may not seem clear. For example, ‘India’ is an instance of cricket team could be thought of as a set of players. Here notice that value of the slot ‘players’ is a set. Suppose we want to represent ‘India’ as a class instead of an instance. Then its instances would be the individual players.

A class can be viewed as 2 things simultaneously:
A subset (isa) of a larger class that also contains its elements and
An instance of a class of sets from which it inherits set level properties.

It is useful to distinguish between regular classes, whose elements are individual entities and meta classes, which are special classes whose elements are themselves classes. A class is now an element of (instance) some class as well as a sub class (isa) of one or more classes.

Consider the example shown below:

class

| instance | : | class |
| isa      | : | class |

team

| instance | : | class |
| isa      | : | class |
| *team size | : |

cricket team

| instance | : | class |
| isa      | : | team |
| *team size | : | 14 |
| *manager | : |
India

instance : cricket team
isa : cricket player
team size : 13
manager : chapel
*uniform color: blue

Dravid

instance : India
instance : batsman
uniform color : blue
batting avrg : 53

The most basic meta class is the ‘class’ all classes are instances of it. In the example, ‘team’ is a sub class of class ‘class’ and ‘cricket team’ is a sub class of ‘team’.

‘India’ is an instance of ‘cricket team’. It is not an instance of ‘class’ because its elements are individuals, not sets. ‘India’ is a sub class of ‘cricket player’, since all of its elements are also elements of that set.

Finally, ‘Dravid’ is an instance of ‘India’. This makes him also, by traversing ‘isa’ links, an instance of ‘cricket player’. We had a class ‘batsman’, to which we attached the fact that ‘batsman’ have above average batting averages. To allow that here, we make ‘Dravid’ an instance of ‘batsman’ as well. He thus inherits properties from both ‘India’ and from ‘batsman’, as well as the classes above these.

Other class relations

Classes can be related in many ways.

One such relation ship is ‘mutually disjoint with’. It relates a class to one or more other classes that are guaranteed to have no elements in common with it.

Another relationship is ‘is covered by’. It relates a class to a set of subclasses, the union of which is equal to it. If a class ‘is covered by’ a set S of mutually disjoint classes, then s is called a partition of the class.
The examples are shown below. Consider the classes in the following semantic net.

- **Cricket player**
  - is covered by \{ batsman, bowler, fielder \}
  - \{ national cricketer, county cricketer \}

- **batsman**
  - isa : cricket player
  - mutually disjoint with : \{ bowler, fielder \}

- **bowler**
  - isa : cricket player
  - mutually disjoint with : \{ batsman, fielder \}
fielder

isa : cricket player
mutually disjoint with : { batsman, bowler }

Dravid

instance : batsman
instance : national cricketer

Frame languages

To represent knowledge, we are using frames. A number of frame oriented knowledge representation languages have been developed. Examples of such languages are

KRL,
FRL,
RLL,
KL-ONE,
KRYPTON,
NIKL,
CYCL,
Conceptual graphs,
THEO and
FRAMEKIT.

Module 4

Knowledge and Reasoning – Review of representation and reasoning with Logic
Propositional calculus

Propositional calculus is a language. Using their words, phrases and sentences, we can represent and reason about properties and relationships in the world. Propositional logic is simple
to deal with. We can easily represent real world facts as logical propositions written as well formed formulas (wff’s) in propositional logic.

Propositional calculus symbols

The symbols of propositional calculus are the

propositional symbols:

P, Q, R, S,….

truth symbols:

True, False

and connectives:

∩, U, ¬, →, ≡

propositional calculus sentences

the following are valid sentences;
true,

P,
Q,
R,
¬, ¬ false,
P U ¬P,
P ∩ ¬P,
P → Q,
P U Q ≡ R

Conjunct
In the sentence $P \cap Q$, $P$ and $Q$ are called conjuncts.

Disjunct

In the sentence $P \cup Q$, $P$ and $Q$ are called disjuncts.

Equivalence

Two expressions in propositional calculus are equivalent if they have the same value under all truth value assignments.

\begin{align*}
\neg(\neg P) & \equiv P \\
P \rightarrow Q & \equiv \neg P \cup Q \\
\neg(P \cup Q) & \equiv \neg P \cap \neg Q \\
\neg(P \cap Q) & \equiv \neg P \cup \neg Q \\
P \cap Q & \equiv Q \cap P \\
P \cup Q & \equiv Q \cup P \\
(P \cap Q) \cap R & \equiv P \cap (Q \cap R) \\
(P \cup Q) \cup R & \equiv P \cup (Q \cup R) \\
P \cup (Q \cap R) & \equiv (P \cup Q) \cap (P \cup R) \\
P \cap (Q \cup R) & \equiv (P \cap Q) \cup (P \cap R)
\end{align*}

These identities can be used to change propositional calculus expressions into a syntactically different but logically equivalent form.

Semantics of propositional calculus

A proposition symbol corresponds to a statement about the world. For example, $P$ may denote the statement “it is raining” or $Q$, the statement “I live in a wooden house”. A proposition may be either true or false given some state of the world.

Inference in first order logic
Predicate calculus

In propositional calculus, each atomic symbol P, Q etc. denotes a proposition of some complexity. There is no way to access the components of an individual assertion. Predicate calculus provides this ability. For eg. Instead of letting a single propositional symbol, P, denote the entire sentence “Marcus is a man”, we can create a predicate man that describes the relationship between man and Marcus.

Man (Marcus)

Through inference rules we can manipulate predicate calculus expressions.

Predicate calculus allows expressions to contain variables.

The syntax of predicates

The symbols of predicate calculus consists of

1. The set of letters of the English alphabet. (both upper case and lower case)
2. The set of digits, 0, 1, 2….9.
3. The underscore, _.

Symbols are used to denote objects, properties or relations in a world.

Predicate calculus terms

The following are some statements represented in predicate calculus.

f (X, Y)
father (david)
price (bananas)
likes (george, Susie)
friends (bill, george)
likes (X, george)

The predicate symbols in these expressions are f, father, price, likes, friends.

In the predicates above, david, bananas, george, Susie, bill are constant symbols.
The following represents some facts in the real world and corresponding predicate calculus expressions.

1. Marcus was a man.
   \( \text{man (Marcus)} \)

2. Marcus was a Pompeian.
   \( \text{pompeian (Marcus)} \)

3. All Pompeians were Romans.
   \( \forall x \, \text{Pompeian}(x) \implies \text{Roman}(x) \)

4. Caesar was a ruler.
   \( \text{Ruler (Caesar)} \)

5. All Romans were either loyal to Caesar or hated him.
   \( \forall x \, \text{Roman}(x) \implies \text{loyal}(x, \text{Caesar}) \lor \text{hate}(x, \text{Caesar}) \)

6. Everyone is loyal to someone.
   \( \forall x \, \exists y \, \text{loyal}(x, y) \)

7. People only try to assassinate rulers they are not loyal to.
   \( \forall x \, \forall y \, \text{person}(x) \land \text{ruler}(y) \land \text{tryassasinate}(x, y) \implies \neg \text{loyal}(x, y) \)

8. Marcus tried to assassinate Caesar.
   \( \text{Tryassasinate (Marcus, Caesar)} \)

**Inference rules involving quantifiers**

**Quantifiers**

Predicate calculus includes 2 symbols, the \( \forall \) and \( \exists \).

For example,

\[ \exists \forall \]

\[ \exists Y \, \text{friends (Y, peter)} \]

\[ \forall X \, \text{likes(X, icecream)} \]

The universal quantifier, \( \forall \), indicates that the sentence is true for all values of the variable. The existential quantifier, \( \exists \), indicates that the sentence is true for at least one value in the domain. Some relationships between universal and existential quantifiers are given below.
\[ \neg \exists x \ p(x) \equiv \forall x \ \neg p(x) \]

\[ \neg \forall x \ p(x) \equiv \exists x \ \neg p(x) \]

\[ \exists x \ p(x) \equiv \forall y \ p(y) \]

\[ \forall x \ q(x) \equiv \forall y \ q(y) \]

\[ \forall x \ (p(x) \cap q(x)) \equiv \forall x \ p(x) \cap \forall y \ q(y) \]

\[ \exists x \ (p(x) \cup q(x)) \equiv \exists x \ p(x) \cup \exists y \ q(y) \]

Inference rules

**Modus ponens**

If the sentences \( P \) and \( P \rightarrow Q \) are known to be true, then modus ponens lets us infer \( Q \).

Example

Assume the following observations.

1. “if it is raining, then the ground will be wet”.
2. “it is raining”.

If \( P \) denotes “it is raining” and
\( Q \) denotes “the ground is wet”;
Then 1 becomes
\[ P \rightarrow Q \]
2 becomes
\[ Q \]
Through the application of modus ponens we can infer \( Q \) that is “ground is wet”.
Example 2

Consider the statements
“Socrates is a man”. And
“all men are mortal”.

These can be represented in predicate calculus.

“Socrates is a man” can be represented by
$$\text{man}(\text{Socrates})$$

“all men are mortal” can be
$$\forall X (\text{man}(X) \rightarrow \text{mortal}(X))$$

Since the variable $X$ in the implication is universally quantified, we may substitute any value in the domain for $x$. By substituting “Socrates” for $X$ in the implication, we infer the expression,
$$\text{man}(\text{socrates}) \rightarrow \text{mortal}(\text{socrates})$$

By considering the predicates
$$\text{man}(\text{socrates}) \text{ and } \text{man}(\text{socrates}) \rightarrow \text{mortal}(\text{socrates})$$

Applying modus ponens, we get
$$\text{mortal}(\text{socrates})$$

That is “Socrates is mortal”.
**Unification**

To apply inference rules such as modus ponens, an inference system must be able to determine when 2 expressions are the same or match. In predicate calculus, the process of matching 2 sentences is complicated by the existence of variables in the expressions.

Unification is an algorithm for determining the substitutions needed to make 2 predicate calculus expressions match. For example, in the earlier section, we substitute ‘socrates’ for X in the expression

\[ \forall X (\text{man}(X) \rightarrow \text{mortal}(X)) \]

This allowed the application of modus ponens to infer

\[ \text{mortal}(\text{socrates}) \]

**Example**

\[ \text{man}(\text{socrates}) \]

\[ \forall X (\text{man}(X) \rightarrow \text{mortal}(X)) \]

Here we substituted Socrates for X. This substitution is denoted by \{ socrates / X \}. \{ socrates / X \} is called a unifier.

Next let us see the unification algorithm. To simplify the manipulation of expressions, the algorithm assumes a slightly modified syntax. By representing an expression as a list with the predicate or function names as the first element followed by its arguments, we simplify the manipulation of expressions.

<table>
<thead>
<tr>
<th>PC syntax</th>
<th>LIST syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>p (a,b)</td>
<td>(p a b)</td>
</tr>
<tr>
<td>friends (george, tom)</td>
<td>(friends george tom)</td>
</tr>
<tr>
<td>equal (eve, mother (cain) )</td>
<td>(equal eve (mother cain))</td>
</tr>
<tr>
<td>man (socrates)</td>
<td>(man socrates)</td>
</tr>
</tbody>
</table>
Unification algorithm

function unify (E1, E2)
{
  case
  {
    both E1 and E2 are constants or the empty list:
      if E1 = E2 then return { };
      else return FAIL;
    E1 is a variable:
      if E1 occurs in E2 then return FAIL;
      else return {E2 / E1};
    E2 is a variable:
      if E2 occurs in E1 then return FAIL;
      else return {E1 / E2};
    either E1 or E2 are empty:
      then return FAIL;
  otherwise:
  {
    HE1 = first element of E1;
    HE2 = first element of E2;
    SUBS1 = unify (HE1, HE2);
    If (SUBS1 = fail) then return FAIL;
TE1 = apply (SUBS1, rest of E1);
TE2 = apply (SUBS1, rest of E2);
SUBS2 = unify (TE1, TE2);
if (SUBS2 = fail) then return FAIL;
else return composition (SUBS1, SUBS2);

Example
Consider an example. Given 2 predicates.
1. parents (X, father (X), mother (bill))
2. parents (bill, father (bill), Y)
These predicates can be represented in list syntax as
1 → (parents X father X) (mother bill))
2 → (parents bill (father bill) Y)
To find out the unifier that match the above expressions call unify( );
unify ( (parents x (father X) (mother bill)), (parents bill (father bill) Y))
A trace of the execution of the above call is shown below.
1. unify (parents X (father X) (mother bill), (parents bill (father bill) Y))

\[ \text{return } \{(\text{mother bill})/Y, \text{bill}/X\} \]

2. unify (parents, parents)

\[ \text{return } \{\} \]

3. unify (X (father X) (mother bill), (bill (father bill) Y))

\[ \text{return } \{(\text{mother bill})/Y\} \]

4. unify (X, bill)

\[ \text{return } \{\text{bill}/X\} \]

5. unify ((father bill) (mother bill)), ((father bill) Y))

\[ \text{return } \{(\text{mother bill})/Y\} \]

6. unify ((father bill), (father bill))

\[ \text{return } \{\} \]

11. unify(((mother bill)), (Y))
After the execution of the unification algorithm, we get the result as \{bill/X, (mother bill) / Y\}.

Resolution

Resolution is a technique for proving theorems in the propositional or predicate calculus. Resolution proves a theorem by negating the statement to be proved and adding this negated goal to the set of axioms.

Resolution proofs involve the following steps.

1. put the premises or axioms in to clause form.
2. add the negation of what is to be proved, in clause form, to the set of axioms.
3. resolve these clauses together, producing new clauses that logically follow from them.
4. produce a contradiction by generating the empty clause.
5. the substitutions used to produce the empty clause are those under which the opposite of the negated goal is true.
Resolution requires that the axioms and the negation of the goal be placed in a normal form called clause form. Clause form represents the logical database as a set of disjunctions of literals.

Producing the clause form

The resolution procedure requires all statements in the database to be converted to a standard form called clause form. The form is referred to as conjunction of disjuncts.

The following is an example of a fact represented in clause form.

\((-\text{dog}(X) \cup \text{animal}(X)) \cap (-\text{animal}(Y) \cup \text{die}(Y)) \cap \text{dog}(\text{fido})\)

The following is the algorithm for reducing any set of predicate calculus statements to clause form.

1. First we eliminate the → by using the equivalent form. For example \(a \rightarrow b \equiv \neg a \cup b\).
2. Next we reduce the scope of negation.
   \[
   \neg (\neg a) \equiv a \\
   \neg (\forall X) a(X) \equiv (\exists X) \neg a(X) \\
   \neg (\exists X) b(X) \equiv (\forall X) \neg b(X) \\
   \neg (a \cap b) \equiv \neg a \cup \neg b \\
   \neg (a \cup b) \equiv \neg a \cap \neg b
   \]
3. Standardize by renaming all variables so that variables bound by different quantifiers have unique names.
   If we have a statement
   \[
   ((\forall X) a(X) \cup \forall X b(X)) \equiv (\forall X) a(X) \cup (\forall Y) b(Y)
   \]
4. Move all quantifiers to the left without changing their order.
5. Eliminate all existential quantifiers by a process called skolemization.
   \[
   (\forall X) (\exists Y) (\text{mother}(X, Y)) \text{ is replaced by } (\forall X) \text{ mother}(X, m(X))
   \]
   \[
   (\forall X) (\forall Y) (\exists Z) (\forall W) (\text{foo}(X, Y, Z, W)) \text{ is replaced with } (\forall X) (\forall Y) (\forall W) (\text{foo}(X, Y, f(X, Y), W))
   \]
6. Drop all universal quantifiers.
7. Convert the expression to the conjunct of disjuncts form using the following equivalences.
   \[ a \lor (b \lor c) \equiv (a \lor b) \lor c \]
   \[ a \land (b \land c) \equiv (a \land b) \land c \]
   \[ a \land (b \lor c) \text{ is already in clause form.} \]
   \[ a \lor (b \land c) \equiv (a \lor b) \land (a \lor c) \]
8. Call each conjunct a separate clause.
   For eg.
   \[ (a \lor b) \land (a \lor c) \]
   Separate each conjunct as
   \[ a \lor b \text{ and} \]
   \[ a \lor c \]
9. Standardize the variables apart again.
   \[ \exists X (a(X) \land b(X)) \equiv (\exists X) a(X) \land (\exists Y) b(Y) \]

After performing these nine steps, we will get the expression in clause form.

Example
Consider the following expression.

\[ \exists X ( [a (X) \land b (X)] \rightarrow [c (X, I) \land (\exists Y) ((\exists Z) [c(Y, Z) \rightarrow d (X, Y)])]) \lor (\exists X) (e(X)) \]

Convert this expression to clause form.

Step 1.
Eliminate the \( \rightarrow \).

\[ \exists X ( [\neg(a (X) \land b (X)] \lor [c (X, I) \land (\exists Y) ((\exists Z) [\neg c(Y, Z) \lor d (X, Y)])]) \lor (\exists X) (e(X)) \]

Step 2:
Reduce the scope of negation.

\[ \exists X ( [\neg(a (X) \lor \neg b (X)] \lor [c (X, I) \land (\exists Y) ((\exists Z) [\neg c(Y, Z) \lor d (X, Y)])]) \lor (\exists X) (e(X)) \]
step 3:
standardize by renaming the variables.

\((\forall X) \ (\neg (a (X) U \neg b (X)) U [c (X, I) \cap (\exists Y) ((\exists Z) \neg c(Y, Z) U d (X, Y))]) U (\forall W) (e(W))\)

step 4:
Move all quantifiers to the left.

\((\forall X) (\exists Y) (\exists Z) (\forall W) \ (\neg (a (X) U \neg b (X)) U [c (X, I) \cap (\neg c(Y, Z) U d (X, Y))]) U (e(W))\)

step 5:
Eliminate existential quantifiers.

\((\forall X) (\forall W) \ (\neg (a (X) U \neg b (X)) U [c (X, I) \cap (\neg c(f(X), g(X)) U d (X, f(X))]) U (e(W))\)

step 6:
Drop all universal quantifiers.

\(\neg (a (X) U \neg b (X)) U [c (X, I) \cap (\neg c(f(X), g(X)) U d (X, f(X))]) U (e(W))\)

step 7:
Convert the expression to conjunct of disjuncts form.

\[\neg (a (X) U \neg b (X)) U [c (X, I) U (e(W)) \cap [\neg (a (X) U \neg b (X)) U \neg c(f(X), g(X)) U d (X, f(X)) U e(W)]\]

step 8:
Call each conjunct a separate clause.

1 \(\rightarrow\) \(\neg (a (X) U \neg b (X)) U c (X, I) U (e(W))\)

2 \(\rightarrow\) \(\neg (a (X) U \neg b (X)) U \neg c(f(X), g(X)) U d (X, f(X)) U e(W)\)

Step 9:
Standardize the variables apart again.

1 \(\rightarrow\) \(\neg (a (X) U \neg b (X)) U c (X, I) U (e(W))\)

2 \(\rightarrow\) \(\neg (a (V) U \neg b (V)) U \neg c(f(V), g(V)) U d (V, f(V)) U e(Z)\)

This is the clause form generated.
The resolution proof procedure

Suppose we are given the following axioms.

1. \( b \cup c \rightarrow a \)
2. \( b \)
3. \( d \cap e \rightarrow c \)
4. \( e \cup f \)
5. \( d \cap \neg f \)

We want to prove ‘a’ from these axioms.

First convert the above predicates to clause form.

We get

1. \( \neg (b \cap c) \cup a \)
2. \( \neg b \cup \neg c \cup a \)
3. \( a \cup \neg b \cup \neg c \)

3. \( \neg (d \cap e) \rightarrow c \)
   \( e \cup \neg d \cup \neg e \)

We get the following clauses.

\( a \cup \neg b \cup \neg c \)
\( b \)
\( c \cup \neg d \cup \neg e \)
\( e \cup f \)
\( d \)
\( \neg f \)

The goal to be proved, a, is negated and added to the clause set.

Now we have

\( a \cup \neg b \cup \neg c \)
\( b \)
\( c \cup \neg d \cup \neg e \)
\( e \cup f \)
After resolving the clauses we get the empty clause. This means that a is true.

Example 2:
Consider the facts.
Anyone passing history exams and winning the lottery is happy. But anyone who studies or is lucky can pass all his exams. John did not study but he is lucky. Anyone who is lucky wins the lottery. Is John happy?

First the sentences to predicate form:

Anyone passing his history exams and winning the lottery is happy.

\[ \forall X \ (\text{pass}(X, \text{history}) \cap \text{win}(X, \text{lottery}) \rightarrow \text{happy}(X)) \]

Anyone who studies or is lucky can pass all his exams.

\[ \forall X \ \forall Y \ (\text{study}(X) \cup \text{lucky}(X) \rightarrow \text{pass}(X, Y)) \]

John did not study but he is lucky.

\[ \neg \text{study}(\text{john}) \cap \text{lucky}(\text{john}) \]

Anyone who is lucky wins the lottery.

\[ \forall X \ (\text{lucky}(X) \rightarrow \text{win}(X, \text{lottery})) \]

After changing these 4 predicate statements to clause form, we get

\[ \neg \text{pass}(X, \text{history}) \cup \neg \text{win}(X, \text{lottery}) \cup \text{happy}(X) \]

\[ \neg \text{study}(Y) \cup \text{pass}(Y, Z) \]

\[ \neg \text{lucky}(V) \cup \text{pass}(V, W) \]

\[ \neg \text{study}(\text{john}) \]

\[ \text{lucky}(\text{john}) \]

\[ \neg \text{lucky}(U) \cup \text{win}(U, \text{lottery}) \]

Into these clauses is entered, in clause form, the negation of the conclusion.

\[ \neg \text{happy}(\text{john}) \]
The derivation of the contradiction is given below.

\[
\neg \text{pass}(X, \text{history}) \cup \neg \text{win}(X, \text{lottery}) \cup \text{happy}(X) \quad \neg \text{lucky}(U) \cup \text{win}(U, \text{lottery})
\]

\[
\{U/X\}
\]

\[
\neg \text{pass}(U, \text{history}) \cup \text{happy}(U) \cup \neg \text{lucky}(U) \quad \neg \text{happy}(\text{john})
\]

\[
\{\text{john}/U\}
\]

\[
\text{lucky}(\text{john}) \quad \neg \text{pass}(\text{john, history}) \cup \neg \text{lucky}(\text{john})
\]

\[
\{}
\]

\[
\neg \text{pass}(\text{john, history}) \quad \neg \text{lucky}(V) \cup \text{pass}(V, W)
\]

\[
\{\text{john}/V, \text{history}/W\}
\]

\[
\neg \text{lucky}(\text{john}) \quad \text{lucky}(\text{john})
\]

\[
\{}
\]

We get the empty clause. This proves that John is happy.
Example 3:
Consider the following set of facts.

All people who are not poor and are smart are happy. Those people who read are not stupid. John can read and is wealthy. Happy people have exciting lives. Can anyone be found with an exciting life?

We assume $\forall X (\text{smart}(X) \equiv \neg \text{stupid}(X))$ and $\forall Y (\text{wealthy}(Y) \equiv \neg \text{poor}(Y))$

$$\forall X (\neg \text{poor}(X) \cap \text{smart}(X) \rightarrow \text{happy}(X))$$
$$\forall Y (\text{read}(Y) \rightarrow \text{smart}(Y))$$
$$\text{read}(\text{john}) \cap \neg \text{poor}(\text{john})$$
$$\forall Z (\text{happy}(Z) \rightarrow \text{exciting}(Z))$$

the negation of the conclusion is:
$$\neg \exists W (\text{exciting}(W))$$

After transforming these in to clause form we get,

poor $(X) \cup \neg \text{smart}(X) \cup \text{happy}(X)$

$\neg \text{read}(Y) \cup \text{smart}(Y)$

read $(\text{john})$

$\neg \text{poor}(\text{john})$

$\neg \text{happy}(Z) \cup \text{exciting}(Z)$

$\neg \text{exciting}(W)$
The resolution proof for this is given below.

Finally we get the empty clause. This proves that some one can be found with an exciting life.
Forward and backward chaining

The completeness of resolution makes it a very important inference method. In many practical situations, however, the full power of resolution is not needed. Real world knowledge bases often contain only clauses of a restricted kind called Horn clause. A Horn clause is a disjunction of literals of which at most one is positive. Inference with Horn clauses can be done through the forward and backward chaining algorithms. In both these techniques, inference steps are obvious and easy to follow for humans.

Forward chaining

The forward chaining method is very simple:

Start with the atomic sentences in the knowledge base and apply modus Ponens in the forward direction, adding new atomic sentences, until no further inferences can be made.

In many cases, the reasoning with forward chaining can be much more efficient than resolution theorem proving.

Consider the following problem:

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.

We will prove that West is a criminal.
First we will represent these facts as first order definite clauses.

“…it is a crime for an American to sell weapons to hostile nations”:

\[
\text{American (x) \cap Weapon (y) \cap Sells (x, y, z) \cap Hostile (z) \rightarrow Criminal (x)} \quad \rightarrow \text{rule 1}
\]

“Nono… has some missiles.”

\[
\exists x \text{ Owns (Nono, x) \cap Missile (x)} \rightarrow \text{Owns (Nono, M1)} \quad \rightarrow \text{rule 2}
\]

\[
\text{Missile (M1)} \quad \rightarrow \text{rule 3}
\]

“All of its missiles were sold to it by Colonel West”

\[
\text{Missile (x) \cap Owns (Nono, x) \rightarrow Sells (West, x, Nono)} \quad \rightarrow \text{rule 4}
\]

Here we must provide the rule that missiles are weapons.

\[
\text{Missile (x) \rightarrow Weapon (x)} \quad \rightarrow \text{rule 5}
\]

Here we must provide the fact that enemy of America counts as hostile.

\[
\text{Enemy (x, America) \rightarrow Hostile (x)} \quad \rightarrow \text{rule 6}
\]

“West, who is American…”

\[
\text{American (West)} \quad \rightarrow \text{rule 7}
\]
“The country Nono, an enemy of America…”

Enemy (Nono, America) \(\rightarrow\) rule 8

Now we have the all the predicates.

Starting from the known facts, the forward chaining method triggers all the rules whose premises are satisfied, adding their conclusions to the known facts. This process repeats until the query is answered or no new facts are added.

On the first phase, rule 1 has unsatisfied premises.

Rule 2 \(\rightarrow\) Owns (Nono, M1)
Rule 3 \(\rightarrow\) Missile (M1)
Rule 4 \(\rightarrow\) Missile (x) \(\cap\) Owns (Nono, x) \(\rightarrow\) Sells (West, x, Nono)
Rule 4 is satisfied with \{x/M1\}, and
Sells (West, M1, Nono) is added.

Rule 3 \(\rightarrow\) Missile (M1)
Rule 5 \(\rightarrow\) Missile (x) \(\rightarrow\) Weapon (x)
Rule 5 is satisfied with \{x/M1\}, and
Weapon(M1) is added.

Rule 8 \(\rightarrow\) Enemy (Nono, America)
Rule 6 \(\rightarrow\) Enemy (x, America) \(\rightarrow\) Hostile (x)
Rule 6 is satisfied with \{x / Nono\},
Hostile (Nono) is added.

Rule 7 \(\rightarrow\) American (West)
Rule 9 $\rightarrow$ Sells (West, M1, Nono)
Rule 10 $\rightarrow$ Weapon(M1)
Rule 11 $\rightarrow$ Hostile (Nono)

Rule 1 $\rightarrow$ American (x) $\cap$ Weapon (y) $\cap$ Sells (x, y, z) $\cap$ Hostile (z) $\rightarrow$ Criminal (x)

In the second phase, rule 1 is satisfied with \( x / \text{West}, \ y / \text{M1}, \ z / \text{Nono} \),

\[
\text{Criminal (West)}
\]

The following figure shows the proof tree generated.

The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second phase at the top level.

**Backward chaining**
Many inference techniques use backward chaining approach. These algorithms work backward from the goal, chaining through rules to find known facts that support the proof. The list of goals can be thought of as a stack waiting to be worked on: if all of them can be satisfied, then the current branch of the proof succeeds. The backward chaining algorithm takes the first goal in the list and finds every clause in the knowledge base whose positive literal or head, unifies with the goal. Each such clause creates a new recursive call in which the premise, or body, of the clause is added to the goal stack.

The following figure shows the proof tree for deriving Criminal (West) from the above set of facts.

```
Criminal (West)
  /    /
American (West) Weapon (y) Sells (West, M1, z) Hostile
   |
   |  |
   | {z/Nono} |
   |
   | /
   | |
   | Missile (y) Missile (M1) Owns (Nono, M1) Enemy
   |
   | {y/M1} |
   |
   | |
   | |
   | America) |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
   |
**Introduction**

Prolog is a logic programming language. It has the following features.

They are:
- Declarative semantics, a means of directly expressing problem relationships in AI,
- Built in unification,
- High powered techniques for pattern matching and search.

**Syntax**

**Representing facts and rules**

<table>
<thead>
<tr>
<th>English</th>
<th>Predicate calculus</th>
<th>PROLOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>$\cap$</td>
<td>;</td>
</tr>
<tr>
<td>or</td>
<td>$\cup$</td>
<td></td>
</tr>
<tr>
<td>only if</td>
<td>←</td>
<td>:-</td>
</tr>
<tr>
<td>not</td>
<td>¬</td>
<td>not</td>
</tr>
</tbody>
</table>

Predicate names are expressed as a sequence of alphanumeric characters.

Variables are represented as a string of characters beginning with an uppercase alphabet.

The fact ‘Everyone likes Susie’ is represented in Prolog as

$$\text{likes (X, susie)}$$

The fact ‘George and Susie like some set of people’ is represented in Prolog as

$$\text{likes (George, Y), likes (Susie, Y)}$$

‘George likes Kate and George likes Susie’ is represented as
likes (george, kate), likes (george, susie)

‘George likes Kate or George likes Susie’ is represented as

likes (george, kate) ; likes (george, susie)

‘If George does not like Kate, then George likes Susie’ is represented as

Likes(george, susie) :- not ( likes (george, kate))

These examples show how the predicate calculus connectives ∩, U, − and ← are expressed in Prolog.

Prolog database

A Prolog program is a set of specifications in the first order predicate calculus describing the objects and relations in the problem domain. The set of specifications is referred to as the database for the problem.

Suppose we wish to describe a world consisting of George’s, Kate’s and Susie’s likes and dislikes. The database might contain the following set of predicates.

| likes (george, kate) |
| likes (george, susie) |
| likes (george, wine) |
| likes (susie, wine) |
| likes (kate, gin) |
| likes (kate, susie) |

We can ask questions to the Prolog interpreter as follows:

?- likes (george, kate)
yes

?- likes (kate, susie)
yes
?- likes (george, X)
X = kate ;
X = susie ;
X = wine ;
no

?- likes (george, beer)
no

In the request, likes (george, X), successive user prompts (;) cause the interpreter to return all the terms in the database specification that may be substituted for the X in the query. They are returned in the order they are found in the database.

Suppose we add the following predicate to the database

friends (X, Y) :- likes (X, Z), likes (Y, Z)

This means ‘X and Y are friends; if there exists a Z such that X likes Z and Y likes Z’.

Suppose we ask the following question to the interpreter

?- friends (george, susie)
yes

To solve the question, Prolog searches the database. The query ‘friends(george, susie)’ is matched with the conclusion of the rule ‘friends (X, Y):- likes (X, Z) , likes (Y, Z)’, with X as ‘george’ and Y as ‘susie’. The interpreter looks for a Z such that ‘likes (george, Z)’ is true.
The interpreter then tries to determine whether likes (susie, kate) is true. When it is found to be false, the value kate for Z is rejected. The interpreter then backtracks to find a second value for Z in ‘likes (george, Z)’.

‘likes (george, Z)’ then matches the second clause in the database, with Z bound to ‘susie’. The interpreter then tries to match ‘likes (susie, susie)’. When this also fails, the interpreter goes back to the database for yet another value for Z. this time wine is found in the third predicate, and the interpreter goes on to show that ‘likes (susie, wine)’ is true. In this case ‘wine’ is the binding that ties ‘george’ and ‘susie’. Prolog tries to match goals with patterns in the order in which the patterns are entered in the database.

Creating and changing the Prolog environment

In creating a Prolog program, the database of specifications is created first. The predicate ‘assert’ adds new predicates to Prolog database.

?- assert ( likes (david, susie) )

adds this predicate to the Prolog database.

Now the query

?- likes (david, susie)
X = susie

is returned.

‘asserta (P)’ adds the predicate P at the beginning of all the predicates P, and ‘assertz (P)’ adds the predicate P at the end of all the predicates named P.

There are also other predicates such as consult, read, write, see, tell, seen, told, listing, trace, spy for monitoring the Prolog environment.

Lists in Prolog
The list is a structure consisting of a set of elements. Examples of Prolog lists are

\[ [1, 2, 3, 4] \]
\[ [\text{tom}, \text{dick}, \text{harry}, \text{fred}] \]
\[ [\text{[george, kate]}, [\text{allen, amy}], [\text{don, pat}]] \]
\[ [\ ] \]

The first elements of a list may be separated from the list by the ‘|’ operator.

For instance, when the list is
\[ [\text{tom}, \text{dick}, \text{harry}, \text{fred}] \]
The first element of the list is ‘tom’ and
the tail of the list is ‘[dick, harry, fred]’.

Using vertical bar operator and unification, we can break a list in to its components.

- If \([\text{tom}, \text{dick}, \text{harry}, \text{fred}]\) is matched to \([X | Y]\), then \(X = \text{tom}\) and \(Y = [\text{dick, harry, fred}]\).
- If \([\text{tom}, \text{dick}, \text{harry}, \text{fred}]\) is matched to \([X, Y | Z]\), then \(X = \text{tom}\), \(Y = \text{dick}\), and \(Z = [\text{harry, fred}]\).
- If \([\text{tom}, \text{dick}, \text{harry}, \text{fred}]\) is matched to \([X, Y, Z | W]\), then \(X = \text{tom}\), \(Y = \text{dick}\), and \(Z = \text{harry}\) and \(W = [\text{fred}]\).
- If \([\text{tom}, \text{dick}, \text{harry}, \text{fred}]\) is matched to \([W, X, Y, Z | V]\), then \(W = \text{tom}\), \(X = \text{dick}\), \(Y = \text{harry}\), \(Z = [\text{fred}]\), and \(V = [\ ]\).

\([\text{tom}, \text{dick}, \text{harry}, \text{fred}]\) will not match \([V, W, X, Y, Z | U]\).

\([\text{tom}, \text{dick}, \text{harry}, \text{fred}]\) will match \([\text{tom}, X | [\text{harry, fred}]\], to give \(X = \text{dick}\).

**Recursion in Prolog**

Member check
The predicate member is used to check whether an item is present in a list. This predicate ‘member’ takes 2 arguments, an element and a list, and returns true if the element is a member of the list.

?- member ( a, [a, b, c, d] )
yes

?- member ( a, [1, 2, 3, 4] )
no

?- member ( X, [a, b, c] )
X = a
;  
X = b
;  
X = c
;  
no

See how this predicate works.

To define member recursively, we first test if X is the first item in the list.

member ( X, [ X | T])

This tests whether X and the first element of the list are identical. If they are not, then check whether X is an element of the rest of the list. This is defined by:

member ( X, [ Y | T]) :- member (X, T)

Thus the 2 lines of Prolog for checking list membership are then

member ( X, [ X | T])
member (X, [Y | T]) :- member(X, T)

We now trace member (c, [a, b, c]) as follows

1. member (X, [X | T])
2. member (X, [Y | T]) :- member(X, T)

?- member (c, [a, b, c])
   call 1. fail, since c ≠ a
   call 2. X = c, Y = a, T = [b, c], member (c, [b, c]) ?
      call 1. fail, since c ≠ b
      call 2. X = c, Y = b, T = [c], member (c, [c]) ?
         call 1. success, c = c
   yes (to second call 2)
   yes (to first call 2)
   yes

The use of cut to control search in Prolog

The cut is denoted by an exclamation symbol, !. The syntax for cut is that of a goal with no arguments. For a simple example of the effect of the cut, see the following.

Suppose we have the following predicates in the Prolog database.

```
path2(X, Y) :- move(X, Z), move(Z, Y)
move(1, 6)
move(1, 8)
move(6, 7)
move(6, 1)
move(8, 3)
move(8, 1)
```
Suppose the prolog interpreter is asked to find all the two move paths from 1; there are 4 answers as shown below.

?- path2 (1, W)

W = 7
;

W = 1
;

W = 3
;

W = 1
;

no

When path2 is altered with cut, only 2 answers result.

path2 (X, Y) :- move (X, Z), !, move (Z, Y)
moves (1, 6)
moves (1, 8)
moves (6, 7)
moves (6, 1)
moves (8, 3)
moves (8, 1)
This happens because variable Z takes on only one value namely 6. Once the first sub goal succeeds, Z is bound to 6 and the cut is encountered. This prohibits further backtracking to the first sub goal and no further bindings for Z.

Thus cut has several side effects.

First, when originally encountered it always succeeds, and

Second, if it is failed back to in backtracking, it causes the entire goal in which it is contained to fail.

There are several uses for the cut in programming.

First, as shown in this example, it allows the programmer to control the shape of the search tree. When further search is not required, the tree can be explicitly pruned at that point.

Second, cut can be used to control recursion.

**Abstract data types in Prolog**

We will build the following data structures in Prolog.

Stack,
Queue and
Priority queue.

**Stack**
A stack is a Last in First out data structure. All elements are pushed on to the top of the stack. Elements are popped from the top of the stack. The operators that we define for a stack are

1. Test check whether the stack is empty.
2. Push inserts an element on to the stack.
3. Pop removes the top element from the stack.
4. Member_stack which checks whether an element is in the stack.
5. Add_list which adds a list of elements to the stack.

We now build these operators in prolog.

1. empty_stack ( [ ] )
   This predicate can be used to test a stack to see whether it is empty or to generate a new empty stack.

2. stack (Top, Stack, [ top | Stack ])
   This predicate performs the push and pop operations depending on the variable bindings of its arguments.
   Push produces a new stack as the 3rd argument when the first 2 arguments are bound.
   Pop produces the top element of the stack when the 3rd argument is bound to the stack. The 2nd argument will then be bound to the new stack, once the top element is popped.

3. member_stack( Element, Stack) :- member (Element, Stack)
   This allows us to determine whether an element is a member of the stack

4. add_list_to_stack (List, Stack, Result) :- append (List, Stack, Result)
   List is added to Stack to produce Result, a new stack. (The predicate append adds Stack to the end of List and produces the new list result.)

Queue
A queue is a first in first out data structure. Here elements are inserted at the rear end and removed from the front end. The operations that we define for a queue are

1. **empty_queue** ([ ])
   This predicate either tests whether a queue is empty or initializes a new empty queue.

2. **enqueue** (E, [ ], [E])
   
   enqueue (E, [H | T], [H | Tnew]) :- enqueue (E, T, Tnew)
   
   This predicate adds the element E to a queue, the second argument. The new augmented queue is the third argument.

3. **dequeue** (E, [E | T], T)
   This predicate produces a new queue, the third argument that is the result of taking the next element, the first argument, off the original queue, the second argument.

4. **dequeue** (E, [E | T], _)
   This predicate lets us peek at the next element, E, of the queue.

5. **member_queue** (Element, Queue) :- member (Element, Queue)
   This tests whether Element is a member of Queue.

6. **add_list_to_queue** (List, Queue, Newqueue) :- append (Queue, List, Newqueue)
   This adds the list List to the end of the queue Queue.

Priority queue
A priority queue orders the elements of a queue so that each new item to the priority queue is placed in its sorted order. The dequeue operator removes the best element from the priority queue. The operations that we define for a priority queue are

1. **empty_queue ([ ] )**
   
   This predicate either tests whether a queue is empty or initializes a new empty queue.

2. **dequeue ( E, [E | T], T )**
   
   This predicate produces a new queue, the third argument that is the result of taking the next element, the first argument, off the original queue, the second argument.

3. **dequeue (E, [E | T], _ )**
   
   This predicate lets us peek at the next element, E, of the queue.

4. **member_queue ( Element, Queue ) :- member (Element, Queue)**
   
   This tests whether Element is a member of Queue.

5. **insert_pq ( State, [ ], [State] ) :- !**
   
   insert_pq ( State, [H | Tail], [State, H | tail] ) :- precedes (State, H)
   
   insert_pq ( State, [H | T], [H | Tnew] ) :- insert_pq (State, T, Tnew)
   
   precedes (X, Y) :- X < Y

   insert_pq operation inserts an element to a priority queue in sorted order.
Searching strategies

Here we will learn how depth first search, Breadth first search and Best first search are implemented in Prolog.

Depth first search in Prolog

The following was the algorithm we learned for Depth first search.

```c
void depth_first_search ()
{
    open = [start];
    closed = [ ];
    while (open not empty )
    {
        remove leftmost state from open, call it X;
        if X is a goal state
            then return SUCCESS;
        else
            {
                generate children of X;
                put X on closed;
                discard children of X, if already on open or closed;
                put remaining children on left end of open;
            }
    }
    return FAIL;
}
```
The following is the Prolog program corresponding to this for depth first search.

go ( Start, Goal):-
    empty_stack (Empty_open),
    stack ( [Start, nil], Empty_open, Open_stack),
    empty_set (Closed_set),
    path (Open_stack, Closed_set, Goal).

path (Open_stack, _, _) :-
    empty_stack (Open_stack),
    write ('No solution found with these rules').

path (Open_stack, Closed_set, Goal) :-
    stack ( [State, Parent], _, Open_stack), State = Goal,
    write (' A solution is found'), nl,
    printsolution ( [State, Parent], Closed_set).

path (Open_stack, Closed_set, Goal) :-
    stack ( [State, Parent], Rest_open_stack, Open_stack),
    get_children ( State, Rest_open_stack, Closed_set, Children)
    add_list_to_stack ( Children, Rest_open_stack, New_open_stack),
    union ( [ [ state, Parent ] ], Closed_set, New_closed_set ),
    path ( New_open_stack, New_closed_set, Goal ), !.

get_children ( state, Rest_open_stack, closed_set, Children) :-
    bagof ( Child,  moves ( State, Rest_open_stack, Closed_set, Child ), Children).

moves ( State, Rest_open_stack, closed_set, [Next, state] ) :-
    move ( State, Next),
    not ( unsafe ( Next),
    not ( member_stack ( [Next, _], Rest_open_stack ) ),
    not ( member_set ( [Next, _], Closed_set ) ).
‘Closed_set’ holds all states on the current path plus the states that were rejected when the algorithm backtracked out of them. To find the path from the start state to the current state, we create the ordered pair [State, Parent] to keep track of each state and its parent; the start state is represented by [Start, nil].

Search starts by a go predicate that initializes the path call. Initially, [start, nil] is in the ‘Open_stack’. ‘Closed_set’ is empty.

The first path call terminates search when the ‘Open_stack’ is empty.

‘Printsolution‘ will go to the ‘Closed_set’ and recursively rebuild the solution path. Note that the solution is printed from start to goal.

The 3rd ‘path’ call uses ‘bagof’, a Prolog predicate standard to most interpreters. ‘bagof’ lets us gather all the unifications of a pattern into a single list. The 2nd parameter to ‘bagof’ is the pattern predicate to be matched in the database. The 1st parameter specifies the components of the 2nd parameter that we wish to collect.

‘bagof’ collects the states reached by firing all of the enabled production rules. This is necessary to gather all descendents of a particular state so that we can add them, in proper order, to ‘open’. The 2nd argument of ‘bagof’, a new predicate named ‘moves’, calls the ‘move’ predicates to generate all the states that may be reached using the production rules. The arguments to ‘moves’ are the present state, the open list, the closed set, and a variable that is the state reached by a good move. Before returning this state, ‘moves’ checks that the new state, ‘Next’, is not a member of either ‘rest_open_stack’, ‘open’ once the present state is removed, or ‘closed_set’. ‘bagof’ calls ‘moves’ and collects all the states that meets these conditions. The 3rd argument of ‘bagof’ thus represents the new states that are to be placed on the ‘Open_stack’. 
Breadth first search in Prolog

The following was the algorithm we learned for breadth first search.

```c
void breadth_first_search ( )
{
    open = [ start ];
    closed = [ ];
    while ( open not empty )
    {
        Remove the leftmost state from open, call it X;
        if X is a goal,
            then return SUCCESS;
        else
        {
            Generate children of X;
            Put X on closed;
            Discard children of x, if already on open or closed;
            Put remaining children on right end of open;
        }
    }
    return FAIL;
}
```
The following is the Prolog program corresponding to this for breadth first search.

```prolog
go ( Start, Goal):-
    empty_queue (Empty_open_queue),
    enqueue ([Start, nil], Empty_open_queue, Open_queue),
    empty_set (Closed_set),
    path (Open_queue, Closed_set, Goal).

path (Open_queue, _, _ ) :-
    empty_queue (Open_queue),
    write ('No solution found with these rules').

path (Open_queue, Closed_set, Goal) :-
    dequeue ( [State, Parent], Open_queue, _ ), State = Goal,
    write (' A solution is found') , nl,
    printsolution ( [State, Parent], Closed_set).

path (Open_queue, Closed_set, Goal) :-
    dequeue ( [State, Parent], Open_queue, Rest_open_queue ),
    get_children( State, Rest_open_queue, Closed_set, Children )
    add_list_to_queue ( Children, Rest_open_queue, New_open_queue ),
    union ( [[ state, Parent ]], Closed_set, New_closed_set ),
    path ( New_open_queue, New_closed_set, Goal ), !.

get_children ( state, Rest_open_queue, closed_set, Children) :-
    bagof ( Child, moves ( State, Rest_open_queue, Closed_set, Child ), Children).
```

www.lectnote.blogspot.com
Best first search in Prolog

The following was the algorithm we learned for best first search.

```prolog
function best_first_search() {
    open = [start];
    closed = []; 
    while (open not empty) {
        remove the left most state from open, call it X;
        If X = goal then return the path from start to X;
        Else {
            generate children of X;
            for each child of X do {
                case 
                the child is not in open or closed : {
```
assign the child a heuristic value;
add the child to open;
}
case
the child is already on open:
{
    if the child was reached by a shorter path
        then give the state on open the shorter path;
}
case
the child is already on closed:
{
    if the child was reached by a shorter path
        then
            remove the state from closed;
            add the child to open;
    }  
}
/*end of for */

put X on closed;
reorder states on open by heuristic merit;
}" end of else */

return FAIL;
}

Our algorithm for best first search is a modification of the breadth first search algorithm in which the open queue is replaced by a priority queue, ordered by heuristic merit.

To keep track of all required search information, each state is represented as a list of 5 elements: the state description, the parent of the state, an integer giving the depth of the graph of
its discovery, an integer giving the heuristic measure of the state, and the integer sum of the 3\textsuperscript{rd} and 4\textsuperscript{th} elements.

The 1\textsuperscript{st} and 2\textsuperscript{nd} elements are found in the usual way; the 3\textsuperscript{rd} is determined by adding one to the depth of its parent; the 4\textsuperscript{th} is determined by the heuristic measure of the particular problem. The 5\textsuperscript{th} element, used for ordering the states on the open\_pq, is $f(n) = g(n) + h(n)$.

\begin{verbatim}
go ( Start, Goal):-
    empty_set ( Closed_set),
    empty_pq ( Open),
    heuristic ( start, Goal, H),
    insert_pq ( [Start, nil, 0, H, H], Open, Open_pq ),
    path (Open_pq, Closed_set, Goal).

path (Open_pq, _, _) :-
    empty_pq (Open_pq),
    write ("No solution found with these rules"),
path (Open_pq, Closed_set, Goal) :-
    dequeue_pq ( [ State, Parent, _, _, _ ], Open_pq, _ ),
    State = Goal,
    write ("A solution is found"), nl,
    printsolution ( [State, Parent, _, _, _ ], Closed_set )..

path (Open_pq, Closed_set, Goal) :-
    dequeue_pq ( [ State, Parent, D, H, S ], Open_pq, Rest_open_pq ),
    get_children ( [ State, Parent, D, H, S ], Rest_open_pq, Closed_set, Children, Goal ),
    insert_list_pq ( Children, Rest_open_pq, New_open_pq ),
    union ( [ [ state, Parent, D, H, S ] ], Closed_set, New_closed_set ),
    path ( New_open_queue, New_closed_set, Goal ), !.

get_children is a predicate that generates all the children of state. It uses bagof and moves predicates as in the previous searches.
\end{verbatim}
get_children ([ state, _, D, _, _ ], Rest_open_pq, Closed_set, Children, Goal) :-
    bagof ( Child, moves ([ state, _, D, _, _ ], Rest_open_pq, Closed_set, Child, Goal), Children).

moves ([ state, _, Depth, _, _ ], Rest_open_pq, closed_set,
    [ Next, state, New_D, H, S ], Goal ) :-
    move ( State, Next ),
    not ( unsafe ( Next ) ),
    not ( member_pq ( [ Next, _ , _ , _ ], Rest_open_pq ) ),
    not ( member_set ( [ Next, _ , _ , _ ], Closed_set ) ),
    New_D is Depth+1,
    Heuristic ( Next, Goal, H ),
    S is New_D + H.

Meta predicates

These predicates are designed to match, query and manipulate other predicates that make up the specifications of the problem domain.

We use meta predicates

1. to determine the type of an expression.
2. to add type constraints to logic programming applications.
3. to build, take apart and evaluate Prolog structures.
4. to compare values of expressions.
5. to convert predicates passed as data to executable code.

assert

this predicate adds a clause to the current set of clauses.

assert ( C ) adds the clause C to the current set of clauses.
var
    var (X) succeeds only when X is an unbound variable.
nonvar
    nonvar (X) succeeds only when X is bound to a non variable term.
=..
    =.. creates a list from a predicate term.
    For example,
    foo (a, b, c) = ..Y
    unifies Y with [ foo, a, b, c ]. The head of the list Y is the function name
    and its tail is the function’s arguments.
functor
    functor ( A, B, C) succeeds with A a term whose principal factor has name B and arity C.
    for example, functor ( foo (a, b), X, Y )
    will succeed with variables X = foo and Y = 2.
clause
    clause ( A, B) unifies B with the body of a clause whose head unifies with A.
    suppose we have a predicate
    p(X) :- q(X) in the database,
    then clause ( p (a), Y) will succeed with Y = q (a).

Unification, the engine for predicate matching and evaluation
    In prolog, the interpreter behaves as a resolution based theorem prover. As a theorem
    prover, Prolog performs a series of resolutions on database entries, rather than evaluating
    statements and expressions.
    In prolog, variables are bound by unification and not by evaluation.
    Unification is a powerful technique for rule based and frame based expert systems. All
    production systems require a form of this matching. For those languages that do not provide it, it
    is necessary to write a unification algorithm.
    Unification performs syntactic matches. It does not evaluate expressions.
example
    suppose we have a predicate
    successor (X, Y) :- Y = X + 1
We might have formulated this clause for checking whether Y is a successor of X. but this will fail because the = operator does not evaluate its arguments, but only attempts to unify the expressions on either side. The call successor (3, 4) fails.

For evaluation, prolog provides an operator ‘is’. ‘is’ evaluates the expression on its RHS and attempts to unify the result with the object on its left. Thus

\[ X \text{ is } Y + Z \]

unifies X with the value of Y added to Z.

Using ‘is’, we may define successor as

\[
\text{successor ( X, Y) :- Y is } X +1 \\
?\text{- successor (3, X)} \\
\text{X = 4} \\
\text{yes}
\]

\[
?\text{- successor (3, 4)} \\
\text{yes}
\]

\[
?\text{- successor (4, 2)} \\
\text{no}
\]

Thus prolog does not evaluate expressions as a default as in traditional languages. The programmer must explicitly indicate evaluation using ‘is’.

**Semantic nets in Prolog**

Here we discuss the implementation of semantic nets in Prolog.

The following shows a semantic net.
Some of the prolog predicates describing the bird hierarchy in the above diagram is:

- isa (canary, bird)
- isa (ostrich, bird)
- isa (bird, animal)
- isa (opus, penguin)
- isa (tweety, canary)
hasprop (tweety, color, white) hasprop (robin, color, red)
hasprop (canary, color, yellow) hasprop (penguin, color, brown)
hasprop (bird, travel, fly) hasprop (fish, travel, swim)
hasprop (ostrich, travel, walk) hasprop (penguin, travel, walk)
hasprop (robin, sound, sing) hasprop (canary, sound, sing)

We write an algorithm to find whether an object in our semantic net has a particular property.

hasproperty (Object, Property, Value) :-
    hasprop (Object, Property, Value)

hasproperty (Object, Property, Value) :-
    isa (Object, Parent),
    hasproperty (Parent, Property, Value)

‘hasproperty’ searches the inheritance hierarchy in a depth first fashion.

Frames in Prolog

The following shows some of the frames from the previous semantic net.

| name    | : bird          |
| isa     | : animal       |
| properties | : flies        |
|         | : feathers     |
| default | :             |
The 1\textsuperscript{st} slot of each frame names the node, such as name (tweety) or name (vertebrate).

The 2\textsuperscript{nd} slot gives the inheritance links between the node and its parents.

The 3\textsuperscript{rd} slot in the node’s frame is a list of frames that describe that node.

The final slot in the frame is the list of exceptions and default values for the node.

We now represent the relationships in the frames given above using Prolog.

```prolog
frame ( name (bird),
    name : animal
    isa : animate
    properties : eats
        skin
    default :

    name : canary
    isa : bird
    properties : color (yellow)
        sound (sing)
    default : size (small)

    name : tweety
    isa : canary
    properties :
    default : color (white)

    name (vertebrate)
)```

```prolog
```
isa (animal),
[ travel (flies), feathers],
[ ]).

frame ( name (penguin),
isa (bird),
[ color (brown) ],
[ travel (walks) ]).

frame ( name (canary),
isa (bird),
[ color (yellow), call (sing) ],
[ size (small) ]).

frame ( name (tweety),
isa (canary),
[ ],
[ color (white) ]).

The following are the procedures to infer properties from their representation.

get ( Prop, Object) :-
frame ( name (Object), _, List_of_properties, _ ),
member ( Prop, List_of_properties ).

get (Prop, Object) :-
frame ( name (Object), _, _, List_of_defaults),
member ( Prop, List_of_defaults).

get ( Prop, Object) :-
frame ( name (Object), isa (Parent), _, _ ),
get ( Prop, Parent).