1. What is Artificial Intelligence?

Data: Raw facts, unformatted information.

Information: It is the result of processing, manipulating and organizing data in response to a specific need. Information relates to the understanding of the problem domain.

Knowledge: It relates to the understanding of the solution domain – what to do?

Intelligence: It is the knowledge in operation towards the solution – how to do? How to apply the solution?

Artificial Intelligence: Artificial intelligence is the study of how make computers to do things which people do better at the moment. It refers to the intelligence controlled by a computer machine.

One View of AI is

- About designing systems that are as intelligent as humans
- Computers can be acquired with abilities nearly equal to human intelligence
- How system arrives at a conclusion or reasoning behind selection of actions
- How system acts and performs not so much on reasoning process.

Why Artificial Intelligence?

- ❑ Making mistakes on real-time can be costly and dangerous.
- ❑ Time-constraints may limit the extent of learning in real world.

The AI Problem

There are some of the problems contained within AI.

1. ***Game Playing and theorem proving*** share the property that people who do them well are considered to be displaying intelligence.
2. Another important foray into AI is focused on ***Commonsense Reasoning***. It includes reasoning about physical objects and their relationships to each other, as well as reasoning about actions and other consequences.
3. To investigate this sort of reasoning Nowell Shaw and Simon built the ***General Problem Solver (GPS)*** which they applied to several common sense tasks as well as the problem of performing symbolic manipulations of logical expressions. But no attempt was made to create a program with a large amount of knowledge about a particular problem domain. Only quite simple tasks were selected.
4. The following are the figures showing some of the tasks that are the targets of work in AI:

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

Perception of the world around us is crucial to our survival. Animals with much less intelligence than people are capable of more sophisticated visual perception. Perception tasks are difficult because they involve analog signals. A person who knows how to perform tasks from several of the categories shown in figure learns the necessary skills in standard order.

First perceptual, linguistic and commonsense skills are learned. Later expert skills such as engineering, medicine or finance are acquired.

Physical Symbol System Hypothesis

At the heart of research in artificial intelligence, the underlying assumptions about intelligence lie in what Newell and Simon (1976) call the physical symbol system hypothesis. They define a physical symbol system as follows:

1. Symbols
2. Expressions
3. Symbol Structure
4. System

A physical symbol system consists of a set of entities called **symbols**, which are physically patters that can occur as components of another type of entity called an **expression** (or symbol structure). A **symbol structure** is composed of a number of instances (or tokens) of symbols related in some physical way. At any instance of the time the **system** will contain a collection of these symbol structures. The system also contains a collection of **processes** that operate on expressions to produce other expressions: processes of creation, modification, reproduction and destruction.

They state hypothesis as:

> ***"A physical symbol system has the necessary and sufficient means for general 'intelligent actions'."***

This hypothesis is only a hypothesis there appears to be no way to prove or disprove it on logical ground so, it must be subjected to empirical validation we find that it is false. We may find the bulk of the evidence says that it is true but only way to determine its truth is by experimentation"

Computers provide the perfect medium for this experimentation since they can be programmed to simulate physical symbol system we like. The importance of the physical symbol system hypothesis is twofold. It is a significant theory of the nature of human intelligence and so is of great interest to psychologists.

What is an AI Technique?

Artificial Intelligence problems span a very broad spectrum. They appear to have very little in common except that they are hard. There are techniques that are appropriate for the solution of a variety of these problems. The results of AI research tells that

> ***Intelligence requires Knowledge***. Knowledge possesses some less desirable properties including:
>
> - *It is voluminous*

- *It is hard to characterize accurately*
- *It is constantly changing*
- *It differs from data by being organized in a way that corresponds to the ways it will be used.*

AI technique is a method that exploits knowledge that should be represented in such a way that:

- The knowledge captures generalizations. In other words, it is not necessary to represent each individual situation. Instead situations that share important properties are grouped together.
- It can be understood by people who must provide it. Most of the knowledge a program has must ultimately be provided by people in terms they understand.
- It can be easily be modified to correct errors and to reflect changes in the world and in our world view.
- It can be used in a great many situations even if it is not totally accurate or complete.
- It can be used to help overcome its own sheer bulk by helping to narrow the range of possibilities that must usually be considered.

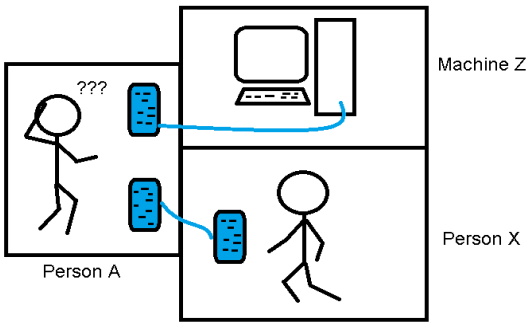
It is possible to solve AI problems without using AI techniques. It is possible to apply AI techniques to solutions of non-AI problems.

Important AI Techniques:

- **Search:** Provides a way of solving problems for which no more direct approach is available as well as a framework into which any direct techniques that are available can be embedded.
- **Use of Knowledge:** Provides a way of solving complex problems by exploiting the structures of the objects that are involved.
- **Abstraction:** Provides a way of separating important features and variations from the many unimportant ones that would otherwise overwhelm any process.

Criteria for Success (Turing Test)

In 1950, **Alan Turing** proposed the method for determining whether a machine can think. His method has since become known as the "**Turing Test**". To conduct this test, we need two people and the machine to be evaluated. Turing Test provides a definition of intelligence in a machine and compares the intelligent behavior of human being with that of a computer.



One person A plays the role of the interrogator, who is in a separate room from the computer and the other person. The interrogator can ask set of questions to both the computer Z and person X by

typing questions and receiving typed responses. The interrogator knows them only as Z and X and aims to determine who the person is and who the machine is.

The goal of machine is to fool the interrogator into believing that it is the person. If the machine succeeds we conclude that the machine can think. The machine is allowed to do whatever it can do to fool the interrogator.

> *For example, if asked the question "How much is 12,324 times 73,981?" The machine could wait several minutes and then respond with wrong answer.*

The interrogator receives two sets of responses, but does not know which set comes from human and which from computer. After careful examination of responses, if interrogator cannot definitely tell which set has come from the computer and which from human, then *the computer has passed the Turing Test*. The more serious issue is the **amount of knowledge** that a machine would need to pass the Turing test.

Overview of Artificial Intelligence

It was the ability of electronic machines to store large amounts of information and process it at very high speeds that gave researchers the vision of building systems which could emulate (imitate) some human abilities.

We will see the introduction of the systems which equal or exceed human abilities and see them because an important part of most business and government operations as well as our daily activities.

Definition of AI: Artificial Intelligence is a branch of computer science concerned with the study and creation of computer systems that exhibit some form of intelligence such as systems that learn new concepts and tasks, systems that can understand a natural language or perceive and comprehend a visual scene, or systems that perform other types of feats that require human types of intelligence.

To understand AI, we should understand

- Intelligence
- Knowledge
- Reasoning
- Thought
- Cognition: gaining knowledge by thought or perception learning

The definitions of AI vary along two main dimensions: thought process and reasoning and behavior.

AI is not the study and creation of conventional computer systems. The study of the mind, the body, and the languages as customarily found in the fields of psychology, physiology, cognitive science, or linguistics.

In AI, the goal is to develop working computer systems that are truly capable of performing tasks that require high levels of intelligence.

2. Problems, Problem Spaces and Search

Problem:

A problem, which can be caused for different reasons, and, if solvable, can usually be solved in a number of different ways, is defined in a number of different ways.

To **build a system or to solve a particular problem** we need to do four things.

1. Define the problem precisely. This definition must include precise specification of what the initial situation will be as well as what final situations constitute acceptable solutions to the problem
2. Analyze the problem
3. Isolate and represent the task knowledge that is necessary to solve the problem
4. Choose the best solving technique and apply it to the particular problem.

Defining the Problem as a State Space Search

Problem solving = Searching for a goal state

It is a structured method for solving an unstructured problem. This approach consists of number of states. The starting of the problem is "Initial State" of the problem. The last point in the problem is called a "Goal State" or "Final State" of the problem.

State space is a set of legal positions, starting at the initial state, using the set of rules to move from one state to another and attempting to end up in a goal state.

Methodology of State Space Approach

1. To represent a problem in structured form using different states
2. Identify the initial state
3. Identify the goal state
4. Determine the operator for the changing state
5. Represent the knowledge present in the problem in a convenient form
6. Start from the initial state and search a path to goal state

To build a program that could "**Play Chess**"

- we have to first specify the starting position of the chess board
  - Each position can be described by an 8-by-8 array.
  - Initial position is the game opening position.
- rules that define the legal moves
  - Legal moves can be described by a set of rules:
    - Left sides are matched against the current state.
    - Right sides describe the new resulting state.
- board positions that represent a win for one side or the other
  - Goal position is any position in which the opponent does not have a legal move and his or her king is under attack.

- We must make explicit the preciously implicit goal of not only playing a legal game of chess but also winning the game, if possible.

Production System

The entire procedure for getting a solution for AI problem can be viewed as "**Production System**". It provides the desired goal. It is a basic building block which describes the AI problem and also describes the method of searching the goal. Its main components are:

- A **Set of Rules**, each consisting of a left side (a pattern) that determines the applicability of the rule and right side that describes the operation to be performed if the rule is applied.
- **Knowledge Base** – It contains whatever information is appropriate for a particular task. Some parts of the database may be permanent, while the parts of it may pertain only to the solution of the current problem.
- **Control Strategy** – It specifies the order in which the rules will be compared to the database and the way of resolving the conflicts that arise when several rules match at one.
  - The first requirement of a goal control strategy is that it is cause motion; a control strategy that does not cause motion will never lead to a solution.
  - The second requirement of a good control strategy is that it should be systematic.
- **A rule applier:** Production rule is like below

```
if(condition) then
        consequence or action
```

Algorithm for Production System:

1. Represent the initial state of the problem
2. If the present state is the goal state then go to step 5 else go to step 3
3. Choose one of the rules that satisfy the present state, apply it and change the state to new state.
4. Go to Step 2
5. Print "Goal is reached " and indicate the search path from initial state to goal state
6. Stop

Classification of Production System:

Based on the direction they can be

1. Forward Production System
   - Moving from Initial State to Goal State
   - When there are number of goal states and only one initial state, it is advantage to use forward production system.
2. Backward Production System
   - Moving from Goal State to Initial State
   - If there is only one goal state and many initial states, it is advantage to use backward production system.

Production System Characteristics

Production system is a good way to describe the operations that can be performed in a search for solution of the problem.

Two questions we might reasonably ask at this point are:

- *Can production systems, like problems, be described by a set of characteristics that shed some light on how they can easily be implemented?*
- *If so, what relationships are there between problem types and the types of production systems best suited to solving the problems?*

The answer for the first question can be considered with the following ***definitions of classes of production systems***:

A **monotonic production system** is a production system in which the applications of a rule never prevents the later application of another rule that could also have been applied at the time the first rule was selected.

A **non-monotonic production system** is one which this is not true.

A **partially commutative production system** is a production system with the property that if the application of a particular sequence of rules transforms state X into state Y, then any permutation of those rules that is allowable also transforms state X into state Y.

A **commutative production system** is a production system that is both monotonic and partially commutative.

In a formal sense, there is no relationship between kinds of problems and kinds of production of systems, since all problems can be solved by all kinds of systems. But in practical sense, there definitely is such a relationship between kinds of problems and the kinds of systems that led themselves naturally to describing those problems.

The following figure shows the four categories of production systems produced by the two dichotomies, monotonic versus non-monotonic and partially commutative versus non-partially commutative along with some problems that can be naturally be solved by each type of system.

	Monotonic	**Non-monotonic**
Partially commutative	Theorem proving	Robot Navigation
Not Partially commutative	Chemical Synthesis	Bridge

The four categories of Production Systems

- Partially commutative, monotonic production systems are useful for solving ignorable problems that involves creating new things rather than changing old ones generally ignorable. Theorem proving is one example of such a creative process partially commutative, monotonic production system are important for a implementation stand point because they can be implemented without the ability to backtrack to previous states when it is discovered that an incorrect path has been followed.
- Non-monotonic, partially commutative production systems are useful for problems in which changes occur but can be reversed and in which order of operations is not critical.

This is usually the case in physical manipulation problems such as "Robot navigation on a flat plane". The 8-puzzle and blocks world problem can be considered partially commutative production systems are significant from an implementation point of view because they tend to read too much duplication of individual states during the search process.

- Production systems that are not partially commutative are useful for many problems in which changes occur. For example "Chemical Synthesis"
- Non-partially commutative production system less likely to produce the same node many times in the search process.

Problem Characteristics

In order to choose the most appropriate method (or a combination of methods) for a particular problem, it is necessary to analyze the problem along several key dimensions:

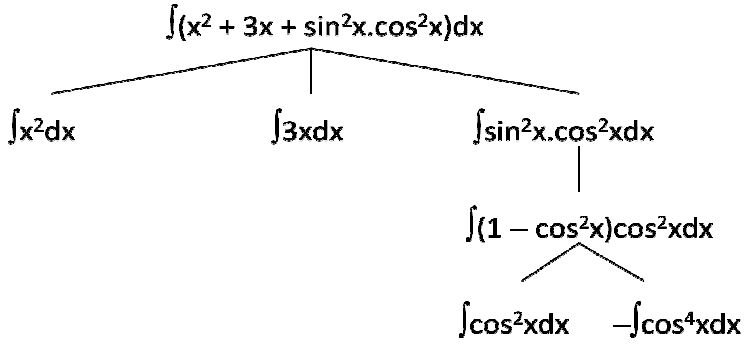
- Is the problem decomposable?
- Can solution steps be ignored or undone?
- Is the universe predictable?
- Is a good solution absolute or relative?
- Is the solution a state or a path?
- What is the role of knowledge?
- Does the task require human-interaction?
- Problem Classification

Is the problem decomposable?

Decomposable problem can be solved easily. Suppose we want to solve the problem of computing the expression.

$$\int (x^2 + 3x + \ sin^2 \ x.cos^2 x)\ dx$$

We can solve this problem by breaking it down into these smaller problems, each of which we can then solve by using a small collection of specific rules the following figure shows problem tree that as it can be exploited by a simple recursive integration program that works as follows.



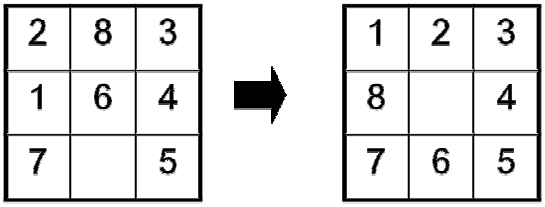
At each step it checks to see whether the problem it is working on is immediately solvable. If so, then the answer is returned directly. If the problem is not easily solvable, the integrator checks to

see whether it can decompose the problem into smaller problems. It can create those problems and calls itself recursively on using this technique of problem decomposition we can often solve very large problem easily.

Can solution steps be ignored or undone?

Suppose we are trying to prove a mathematical theorem. We proceed by first proving a lemma that we think will be useful. A lemma that has been proved can be ignored for next steps as eventually we realize the lemma is no help at all.

Now consider the 8-puzzle game. A sample game using the 8-puzzle is shown below:



In attempting to solve the 8 puzzle, we might make a stupid move for example; we slide the tile 5 into an empty space. We actually want to slide the tile 6 into empty space but we can back track and undo the first move, sliding tile 5 back to where it was then we can know tile 6 so mistake and still recovered from but not quit as easy as in the theorem moving problem. An additional step must be performed to undo each incorrect step.

Now consider the problem of playing chess. Suppose a chess playing problem makes a stupid move and realize a couple of moves later. But here solutions steps cannot be undone.

The above three problems illustrate difference between three important classes of problems:

1) **Ignorable:** in which solution steps can be ignored.
   Example: Theorem Proving
2) **Recoverable:** in which solution steps can be undone.
   Example: 8-Puzzle
3) **Irrecoverable:** in which solution steps cannot be undone.
   Example: Chess

The recoverability of a problem plays an important role in determining the complexity of the control structure necessary for problem solution.

Ignorable problems can be solved using a simple control structure that never backtracks. ***Recoverable problems*** can be solved by slightly complicated control strategy that does sometimes make mistakes using backtracking. ***Irrecoverable problems*** can be solved by recoverable style methods via planning that expands a great deal of effort making each decision since the decision is final.

Is the universe predictable?

There are certain outcomes every time we make a move we will know what exactly happen. This means it is possible to plan entire sequence of moves and be confident that we know what the resulting state will be. Example is 8-Puzzle.

In the uncertain problems, this planning process may not be possible. Example: Bridge Game – Playing Bridge. We cannot know exactly where all the cards are or what the other players will do on their turns.

We can do fairly well since we have available accurate estimates of a probabilities of each of the possible outcomes. A few examples of such problems are

- Controlling a robot arm: The outcome is uncertain for a variety of reasons. Someone might move something into the path of the arm. The gears of the arm might stick.
- Helping a lawyer decide how to defend his client against a murder charge. Here we probably cannot even list all the possible outcomes, which leads outcome to be uncertain.

- *For certain-outcome problems, planning can used to generate a sequence of operators that is guaranteed to lead to a solution.*
- *For uncertain-outcome problems, a sequence of generated operators can only have a good probability of leading to a solution.*
- *Plan revision is made as the plan is carried out and the necessary feedback is provided.*

Is a Good Solution Absolute or Relative?

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 Pompeian.
3) Marcus was born in 40 A.D.
4) All men are mortal.
5) All Pompeian's 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 a 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.

	Justification
1. Marcus was a man.	axiom 1
4. All men are mortal.	axiom 4
8. Marcus is mortal.	1, 4
3. Marcus was born in 40 A.D.	axiom 3
7. It is now 1991 A.D.	axiom 7
9. Marcus' age is 1951 years.	3, 7
6. No mortal lives longer than 150 years.	axiom 6
10. Marcus is dead.	8, 6, 9
OR	
7. It is now 1991 A.D.	axiom 7
5. All Pompeians died in 79 A.D.	axiom 5
11. All Pompeians are dead now.	7, 5
2. Marcus was a Pompeian.	axiom 2
12. Marcus is dead.	11, 2

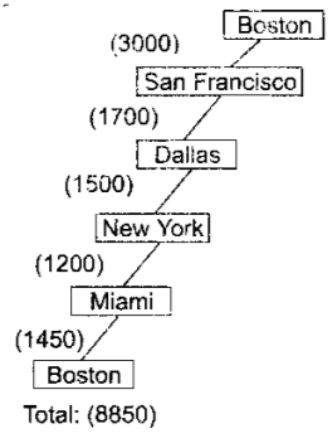
Two Ways of Deciding That Marcus Is Dead

Since we are interested in the answer to the question, it does not matter which path we follow. If we do follow one path successfully to the answer, there is no reason to go back and see if some other path might also lead to a solution. These types of problems are called as "Any path Problems".

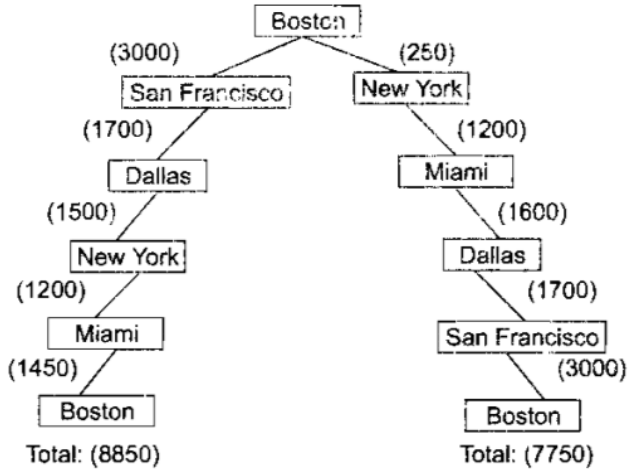
Now consider the Travelling Salesman Problem. Our goal is to find the shortest path route that visits each city exactly once.

	Boston	New York	Miami	Dallas	S.F.
Boston		250	1450	1700	3000
New York	250		1200	1500	2900
Miami	1450	1200		1600	3300
Dallas	1700	1500	1600		1700
S.F.	3000	2900	3300	1700	

An Instance of the Traveling Salesman Problem



One Path among the Cities



Two Paths Among the Cities

Suppose we find a path it may not be a solution to the problem. We also try all other paths. The shortest path (best path) is called as a solution to the problem. These types of problems are known as "Best path" problems. But path problems are computationally harder than any path problems.

Is the solution a state or a path?

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 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 financed institution or to a side of river. But only one of these may have a President.
- The word "dish" is the object of the word "eat". It is possible that a dish was eaten.
- But it is more likely that the pasta salad in the dish was eaten.

Because of the interaction among the interpretations of the constituents of the sentence some search may be required to find a complete interpreter 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. But with the "water-jug" problem it is not sufficient to report the final state we have to show the "path" also.

So the solution of natural language understanding problem is a state of the world. And the solution of "Water jug" problem is a path to a state.

What is the role of knowledge?

Consider the problem of playing chess. The knowledge required for this problem is the rules for determining legal move and some simple control mechanism that implements an appropriate search procedure.
Now consider the problem of scanning daily newspapers to decide which are supporting 'n' party and which are supporting 'y' party. For this problems are required lot of knowledge.

The above two problems illustrate the difference between the problems for which a lot of knowledge is important only to constrain the search for a solution and those for which a lot of knowledge is required even to be able to recognize a solution.

Does a task require interaction with the person?

Suppose that we are trying to prove some new very difficult theorem. We might demand a prove that follows traditional patterns so that mathematician each read the prove and check to make sure it is correct. Alternatively, finding a proof of the theorem might be sufficiently difficult that the program does not know where to start. At the moment people are still better at doing the highest level strategies required for a proof. So that the computer might like to be able to ask for advice.
For Example:

- Solitary problem, in which there is no intermediate communication and no demand for an explanation of the reasoning process.
- Conversational problem, in which intermediate communication is to provide either additional assistance to the computer or additional information to the user.

Problem Classification

When actual problems are examined from the point of view all of these questions it becomes apparent that there are several broad classes into which the problem fall. The classes can be each associated with a generic control strategy that is approached for solving the problem. There is a variety of problem-solving methods, but there is no one single way of solving all problems. Not all new problems should be considered as totally new. Solutions of similar problems can be exploited.

PROBLEMS

Water-Jug Problem

Problem is "You are given two jugs, a 4-litre one and a 3-litre one. One neither has any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 litres of water into 4-litre jug?"

Solution:

The state space for the problem can be described as a set of states, where each state represents the number of gallons in each state. The game start with the initial state described as a set of ordered pairs of integers:

- State: (x, y)
  - x = number of lts in 4 lts jug
  - y = number of lts in 3 lts jug

  x = 0, 1, 2, 3, or 4 y = 0, 1, 2, 3
- Start state: (0, 0) i.e., 4-litre and 3-litre jugs is empty initially.
- Goal state: (2, n) for any n that is 4-litre jug has 2 litres of water and 3-litre jug has any value from 0-3 since it is not specified.
- Attempting to end up in a goal state.

Production Rules: These rules are used as operators to solve the problem. They are represented as rules whose left sides are used to describe new state that result from approaching the rule.

1	(x, y) if $x < 4$	→	$(4, y)$	Fill the 4-gallon jug
2	(x, y) if $y < 3$	→	$(x, 3)$	Fill the 3-gallon jug
3	(x, y) if $x > 0$	→	$(x - d, y)$	Pour some water out of the 4-gallon jug
4	(x, y) if $y > 0$	→	$(x, y - d)$	Pour some water out of the 3-gallon jug
5	$(x, y]$ if $x > 0$	→	$(0, y)$	Empty the 4-gallon jug on the ground
6	(x, y) if $y > 0$	→	$(x, 0)$	Empty the 3-gallon jug on the ground
7	(x, y) if $x + y \geq 4$ and $y > 0$	→	$(4, y - (4 - x))$	Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full
8	(x, y) if $x + y \geq 3$ and $x > 0$	→	$(x - (3 - y), 3)$	Pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full
9	(x, y) if $x + y \leq 4$ and $y > 0$	→	$(x + y, 0)$	Pour all the water from the 3-gallon jug into the 4-gallon jug
10	(x, y) if $x + y \leq 3$ and $x > 0$	→	$(0, x + y)$	Pour all the water from the 4-gallon jug into the 3-gallon jug
11	$(0, 2)$	→	$(2, 0)$	Pour the 2 gallons from the 3-gallon jug into the 4-gallon jug
12	$(2, y)$	→	$(0, y)$	Empty the 2 gallons in the 4-gallon jug on the ground

Fig. 2.3 *Production Rules for the Water Jug Problem*

The solution to the water-jug problem is:

Gallons in the 4-Gallon Jug	Gallons in the 3-Gallon Jug	Rule Applied
0	0	
		2
0	3	
		9
3	0	
		2
3	3	
		7
4	2	
		5 or 12
0	2	
		9 or 11
2	0	

One Solution to the Water Jug Problem

Chess Problem

Problem of playing chess can be defined as a problem of moving around in a state space where each state represents a legal position of the chess board.

The game start with an initial state described as an 8x8 of each position contains symbol standing for the appropriate place in the official chess opening position. A set of rules is used to move from one state to another and attempting to end up on one of a set of final states which is described as any board position in which the opponent does not have a legal move as his/her king is under attacks.

The state space representation is natural for chess. Since each state corresponds to a board position i.e. artificial well organized.

Initial State: Legal chess opening position
Goal State: Opponent does not have any legal move/king under attack

Production Rules:

These rules are used to move around the state space. They can be described easily as a set of rules consisting of two parts:

1. Left side serves as a pattern to be matching against the current board position.
2. Right side that serves decides the chess to be made to the board position to reflect the move.

To describe these rules it is convenient to introduce a notation for pattern and substitutions

E.g.:

1. `White pawn at square (file1,rank2)`
   `Move pawn from square (file i, rank2) AND square (file i, rank2)`
   `AND`

```
      Square (file i,rank3) is empty → To square (file i,rank4)
  AND
      Square (file i,rank4) is empty

2. White knight at square (file i,rank1)
      move Square(1,1) to → Square(i-1,3)
   AND
      Empty Square(i-1,3)
3. White knight at square (1,1)
      move Square(1,1) to → Square(i-1,3)
   AND
      Empty Square(i-1,3)
```

8-Puzzle Problem

The **Problem** is 8-Puzzle is a square tray in which 8 square tiles are placed. The remaining 9th square is uncovered. Each tile has a number on it. A file that is adjacent to the blank space can be slide into that space. The goal is to transform the starting position into the goal position by sliding the tiles around.

Solution:

State Space: The state space for the problem can be written as a set of states where each state is position of the tiles on the tray.

Initial State: Square tray having 3x3 cells and 8 tiles number on it that are shuffled

2	8	3
1	6	4
7		5

Goal State

1	2	3
8		4
7	6	5

Production Rules: These rules are used to move from initial state to goal state. These are also defined as two parts left side pattern should match with current position and left side will be resulting position after applying the rule.

1. Tile in square (1,1)
   AND — *Move tile from square (1,1) to (2,1)*
   Empty square (2,1)

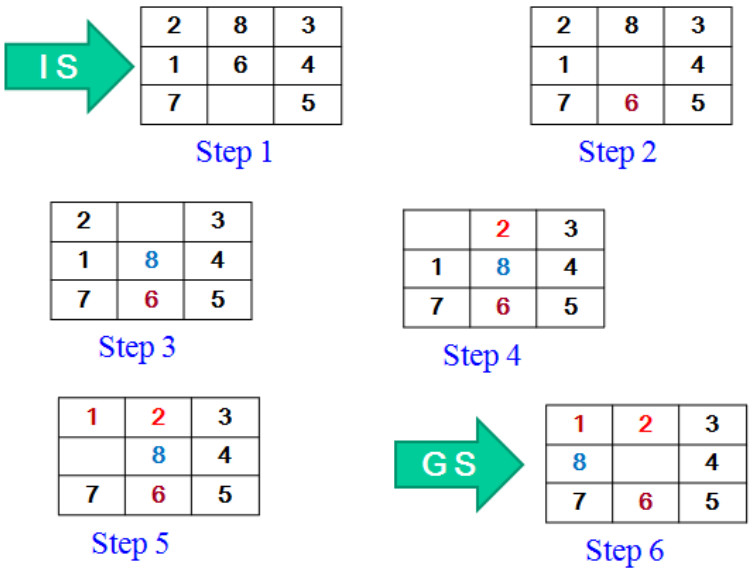
2. Tile in square (1,1)
   AND — *Move tile from square (1,1) to (1,2)*
   Empty square (1,2)

3. Tile in square (2,1)
   AND — *Move tile from square (2,1) to (1,1)*
   Empty square (1,1)

1,1 2	1,2 3	1,3 2
2,1 3	2,2 4	2,3 3
3,1 2	3,2 3	3,3 2

No. of Production Rules: *2 + 3 + 2 + 3 + 4 + 3 + 2 + 3 + 2 = 24*

Solution:



Travelling Salesman Problem

The **Problem** is the salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list. **Find the route the salesman should follow for the shortest possible round trip that both states and finishes at any one of the cities.**

Solution:

State Space: The state space for this problem represents states in which the cities traversed by salesman and state described as salesman starting at any city in the given list of cities. A set of rules is applied such that the salesman will not traverse a city traversed once. These rules are

resulted to be states with the salesman will complex the round trip and return to his starting position.

Initial State

- Salesman starting at any arbitrary city in the given list of cities

Goal State

- Visiting all cities once and only and reaching his starting state

Production rules:

These rules are used as operators to move from one state to another. Since there is a path between any pair of cities in the city list, we write the production rules for this problem as

- `Visited(city[i]) AND Not Visited(city[j])`
  - `Traverse(city[i],city[j])`
- `Visited(city[i],city[j]) AND Not Visited(city[k])`
  - `Traverse(city[j],city[k])`
- `Visited(city[j],city[i]) AND Not Visited(city[k])`
  - `Traverse(city[i],city[k])`
- `Visited(city[i],city[j],city[k]) AND Not Visited(Nil)`
  - `Traverse(city[k],city[i])`

Towers of Hanoi Problem

Problem is the state space for the problem can be described as each state representing position of the disk on each pole the position can be treated as a stack the length of the stack will be equal to maximum number of disks each post can handle. The initial state of the problem will be any one of the posts will the certain the number of disks and the other two will be empty.

Initial State:

- Full(T1) | Empty(T2) | Empty(T3)

Goal State:

- Empty(T1) | Full(T2) | Empty (T3)

Production Rules:

These are rules used to reach the Goal State. These rules use the following operations:

- POP(x) → Remove top element x from the stack and update top
- PUSH(x,y) → Push an element x into the stack and update top. [Push an element x on to the y]

Now to solve the problem the production rules can be described as follows:

1. Top(T1)<Top(T2) → PUSH(POP(T1),T2)
2. Top(T2)<Top(T1) → PUSH(POP(T2),T1)
3. Top(T1)<Top(T3) → PUSH(POP(T1),T3)
4. Top(T3)<Top(T1) → PUSH(POP(T3),T1)
5. Top(T2)<Top(T3) → PUSH(POP(T2),T3)
6. Top(T3)<Top(T2) → PUSH(POP(T3),T2)
7. Empty(T1) → PUSH(POP(T2),T1)

8. Empty(T1) → PUSH(POP(T3),T1)
9. Empty(T2) → PUSH(POP(T1),T3)
10. Empty(T3) → PUSH(POP(T1),T3)
11. Empty(T2) → PUSH(POP(T3),T2)
12. Empty(T3) → PUSH(POP(T2),T3)

Solution: Example: 3 Disks, 3 Towers

1) T1 → T2
2) T1 → T3
3) T2 → T3
4) T1 → T2
5) T3 → T1
6) T3 → T2
7) T1 → T2

Monkey and Bananas Problem

Problem: A hungry monkey finds himself in a room in which a branch of bananas is hanging from the ceiling. The monkey unfortunately cannot reach the bananas however in the room there are also a chair and a stick. The ceiling is just right high so that a monkey standing on a chair could knock the bananas down with the stick. The monkey knows how to move round, carry other things around reach for the bananas and wave the stick in the air. **What is the best sequence of actions for the monkey to acquire lunch?**

Solution: The state space for this problem is a set of states representing the position of the monkey, position of chair, position of the stick and two flags whether monkey on the chair & whether monkey holds the stick so there is a 5-tuple representation.

(M, C, S, F1, F2)

- M: position of the monkey
- C: position of the chair
- S: position of the stick
- F1: 0 or 1 depends on the monkey on the chair or not
- F2: 0 or 1 depends on the monkey holding the stick or not

Initial State (M, C, S, 0, 0)

- The objects are at different places and obviously monkey is not on the chair and not holding the stick

Goal State (G, G, G, 1, 1)

- G is the position under bananas and all objects are under it, monkey is on the chair and holding stick

Production Rules:

These are the rules which have a path for searching the goal state here we assume that when monkey hold a stick then it will swing it this assumption is necessary to simplify the representation.

Some of the production rules are:

1) (M,C,S,0,0) → (A,C,S,0,0) {An arbitrary position A}
2) (M,C,S,0,0) → (C,C,S,0,0) {monkey moves to chair position}
3) (M,C,S,0,0) → (S,S,S,0,0) {monkey brings chair to stick position}
4) (C,C,S,0,0) → (A,A,S,0,0) {push the chair to arbitrary position A}
5) (S,C,S,0,0) → (A,C,A,0,1) {Taking the stick to arbitrary position}
6) (S,C,S,0,0) → (C,C,S,0,0) {monkey moves from stick position to chair position}
7) (C,C,C,0,1) → (C,C,C,1,1)
    - {monkey and stick at the chair position, monkey on the chair and holding stick}
8) (S,C,S,0,1) → (C,C,C,0,1)

Solution:

1) (M,C,S,0,0)
2) (C,C,S,0,0)
3) (G,G,S,0,0)
4) (S,G,S,0,0)
5) (G,G,G,0,0)
6) (G,G,G,0,1)
7) (G,G,G,1,1)

Missionaries and Cannibals Problem

Problem is 3 missionaries and 3 cannibals find themselves one side of the river. They have agreed that they would like to get the other side. But the missionaries are not sure what else the cannibals have agreed to. So the missionaries want to manage the trip across the river on either side of the river is never less than the number of cannibals who are on the same side. The only boat available holds only two people at a time. How can everyone get across without missionaries risking hang eager?

Solution:

The state space for the problem contains a set of states which represent the present number of cannibals and missionaries on the either side of the bank of the river.
(C,M,C1,M1,B)

- C and M are number of cannibals and missionaries on the starting bank
- C1 and M1 are number of cannibals and missionaries on the destination bank
- B is the position of the boat wither left bank (L) or right bank (R)

Initial State → C=3,M=3,B=L so (3,3,0,0,L)
Goal State → C1=3, M1=3, B=R so (0,0,3,3,R)

Production System: These are the operations used to move from one state to other state. Since at any bank the number of cannibals must less than or equal to missionaries we can write two production rules for this problem as follows:

- (C,M,C1,M1,L / C=3, M=3) → (C-2,M,C1+2,M1,R)
- (C,M,C1,M1,L / C=3, M=3) → (C-1,M-1,C1+1,M1+1,R)
- (C,M,C1,M1,L / C=3, M=3) → (C-1,M,C1+1,M1,R)
- (C,M,C1,M1,R / C=1, M=3) → (C+1,M,C1-1,M1,L)
- (C,M,C1,M1,R / C=0, M=3,C1=3,M1=0) → (C+1,M,C1-1,M1,L)

The solution path is

LEFT BANK			RIGHT BANK	
C	M	BOAT POSITION	C1	M1
3	3		0	0
1	3	→	2	0
2	3	←	1	0
0	3	→	3	0
1	3	←	2	0
1	1	→	2	2
2	2	←	1	1
2	0	→	1	3
3	0	←	0	3
1	0	→	2	3
2	0	←	1	3
0	0	→	3	3

3. Heuristic Search Techniques

Control Strategy

The question arises

"***How to decide which rule to apply next during the process of searching for a solution to a problem?***"

Requirements of a good search strategy:

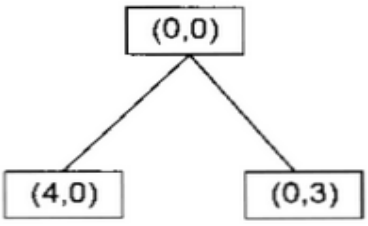
1. It causes motion. It must reduce the difference between current state and goal state. Otherwise, it will never lead to a solution.
2. It is systematic. Otherwise, it may use more steps than necessary.
3. It is efficient. Find a good, but not necessarily the best, answer.

Breadth First Search

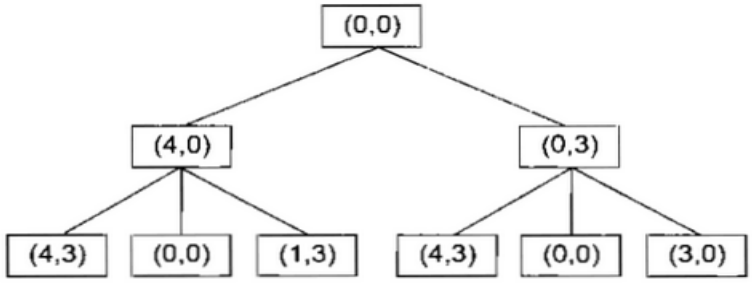
To solve the water jug problem systemically construct a tree with limited states as its root. Generate all the offspring and their successors from the root according to the rules until some rule produces a goal state. This process is called **Breadth-First Search.**

Algorithm:

1) Create a variable called NODE_LIST and set it to the initial state.
2) Until a goal state is found or NODE_LIST is empty do:
    a. Remove the first element from NODE_LIST and call it E. If NODE_LIST was empty quit.
    b. For each way that each rule can match the state described in E do:
        i. Apply the rule to generate a new state
        ii. If the new state is goal state, quit and return this state
        iii. Otherwise add the new state to the end of NODE_LIST



One Level of a Breadth-First Search Tree



Two Levels of a Breadth-First Search Tree

The data structure used in this algorithm is QUEUE.

Explanation of Algorithm:

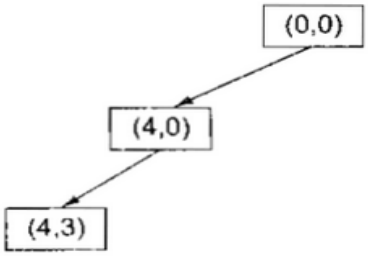
- Initially put **(0,0)** state in the queue
- Apply the production rules and generate new state
- If the new states are not the goal state, (not generated before and not expanded) then only add these states to queue.

Depth First Search

There is another way of dealing the Water Jug Problem. One should construct a single branched tree utility yields a solution or until a decision terminate when the path is reaching a dead end to the previous state. If the branch is larger than the pre-specified unit then backtracking occurs to the previous state so as to create another path. This is called **Chronological Backtracking** because the order in which steps are undone depends only on the temporal sequence in which the steps were originally made. This procedure is called **Depth-First Search.**

Algorithm:

1) If the initial state is the goal state, quit return success.
2) Otherwise, do the following until success or failure is signaled
    a. Generate a successor E of the initial state, if there are no more successors, signal failure
    b. Call Depth-First Search with E as the initial state
    c. If success is returned, signal success. Otherwise continue in this loop.



A Depth-First Search Tree

The data structure used in this algorithm is STACK.
Explanation of Algorithm:

- Initially put the **(0,0)** state in the stack.
- Apply production rules and generate the new state.
- If the new states are not a goal state, (not generated before and no expanded) then only add the state to top of the Stack.
- If already generated state is encountered then POP the top of stack elements and search in another direction.

Advantages of Breadth-First Search

- BFS will not get trapped exploring a **blind alley.**
    - In case of DFS, it may follow a single path for a very long time until it has no successor.
- If there is a solution for particular problem, the BFS is generated to find it. We can find **minimal path** if there are multiple solutions for the problem.

Advantages of Depth –First Search

- DFS requires less memory since only the nodes on the current path are stored.
- Sometimes we may find the solution without examining much.

Example: Travelling Salesman Problem

To solve the TSM problem we should construct a tree which is simple, motion causing and systematic. It would explore all possible paths in the tree and return the one with the shortest length. If there are N cities, then the number of different paths among them is 1.2...(N-1) or (N-1)!

The time to examine a single path is proportional to N. So the total time required to perform this search is proportional to N!

Another strategy is, begin generating complete paths, keeping track of the shorter path so far and neglecting the paths where partial length is greater than the shortest found. This method is better than the first but it is inadequate.

To solve this efficiently we have a search called HEURISTIC SEARCH.

HEURISTIC SEARCH

Heuristic:

- It is a "rule of thumb" used to help guide search
- It is a technique that improves the efficiency of search process, possibly by sacrificing claims of completeness.
- It is involving or serving as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods.

Heuristic Function:

- It is a function applied to a state in a search space to indicate a likelihood of success if that state is selected
- It is a function that maps from problem state descriptions to measures of desirability usually represented by numbers
- Heuristic function is problem specific.

The purpose of 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 (best promising way).

We can find the TSM problem in less exponential items. On the average Heuristic improve the quality of the paths that are explored. Following procedure is to solve TRS problem

- Select a Arbitrary City as a starting city
- To select the next city, look at all cities not yet visited, and select one closest to the current city
- Repeat steps until all cities have been visited

Heuristic search methods which are the general purpose control strategies for controlling search is often known as "weak methods" because of their generality and because they do not apply a great deal of knowledge.

Weak Methods

a) Generate and Test
b) Hill Climbing
c) Best First Search
d) Problem Reduction
e) Constraint Satisfaction
f) Means-ends analysis

Generate and Test

The generate-and-test strategy is the simplest of all the approaches. It consists of the following steps:

Algorithm:

1. Generate a possible solution. For some problems, this means generating a particular point in the problem space. For others, it means generating a path from a start state.
2. Test to see if this is actually a solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states.
3. If a solution has been found, quit. Otherwise return to step 1.

If there exists a solution for one problem then this strategy definitely finds the solution. Because the complete solution must be generated before they can be tested. So, we can say that Generate-and-test algorithm is a Depth-First Search procedure. It will take if the problem space is very large. In the strategy we can operate by generating solution randomly instead of systematically. Then we cannot give the surety that we will set the solution.

To implement this generate and test usually, we will use depth-first tree. If there are cycles then we use graphs rather than a tree. This is not an efficient (mechanism) technique when the problem is much harder. It is acceptable for simple problems. When it is combined with the other techniques it will restrict the space.

For example, one of the most successful AI program is DENDRAL, which informs the structure of organ i.e. components using mass spectrum and nuclear magnetic resonance data. It uses the strategy called ***plan-generate-test***, in which a planning process that uses constraint satisfaction techniques, which creates lists of recommended structures. The generate-and-test procedure then uses those lists so that it can explain only a limited set of structures, which is proved highly effective.

Examples:

- Searching a ball in a bowl (Pick a green ball) - State
- Water Jug Problem – State and Path

Hill Climbing

A GENERATE and TEST procedure, if not only generates the alternative path but also the direction of the path in the alternatives which be near, than all the paths in Generate and Test procedures the heuristic function responds only yes or no but this heuristic function responds only yes will generate an estimate of **how close a given state is to a goal state**.

> Searching for a goal state = Climbing to the top of a hill
> Hill Climbing is Generate-and-test + direction to move.

Simplest Hill Climbing

- Apply only one particular rule at a time.

Algorithm:

1. Evaluate the initial state. If it is also goal state then return it, otherwise continue with the initial states as the current state.
2. Loop until the solution is found or until there are no new operators 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 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 as current state
        iii. If it is not better than the current state, then continue in loop.

The key difference between this algorithm and generate and test algorithm is the use of an evaluation function as a way to inject task-specific knowledge into the control process.

Steepest Hill Climbing

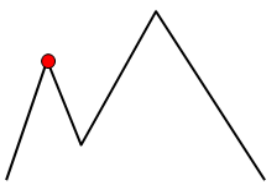
A useful variation on simple hill climbing considers all the moves from the current state and selects the best one as the next state. This method is called ***steepest-ascent hill climbing*** or ***gradient search***.

Algorithm:

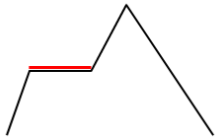
1. Evaluate the initial state. If it is also a goal state then return it and quit. Otherwise continue with the initial state as the current state.
2. Loop until a solution is found or until a complete iteration produces no change to current state:
    a. Let SUCC be a state such that any possible successor of the current state 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.

Bothe basic and steepest-ascent hill climbing may fail to find a solution. Either algorithm may terminate not by finding a goal state but by getting a state from which no better states can be generated. This will happen if the program has reached a local maximum, a plateau or a ridge.

A ***local maximum*** is a state that is better than all its neighbors but it not better than some other states farther away. At the local maximum, all moves appear to make things worse. Local maxima are particularly frustrating because they often occur almost within sight of a solution. In this case, they are called *foothills*.



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



A ***ridge*** is a special kind of 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, although these methods are by no means guaranteed:

- Backtrack to some earlier node and try going in a different direction. This is particularly reasonable if at that node there was another direction that looked as promising or almost as promising as the one that was chosen earlier. This is a fairly good way to deal 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 two or more rules before doing the test. This corresponds to moving in several directions at once. This is a good strategy for dealing with ridges.

Simulated Annealing:

A variation of hill climbing in which, at the beginning of the process, some downhill moves may be made.

In simulated annealing at the beginning of the process some hill moves may be made. The idea is to do enough exploration of the whole space early on. So that the final solution in relatively insensitive to the starting state. By doing so we can lower the chances of getting caught at local maximum, plateau or a ridge.

In this we attempt to minimize rather than maximize the value of the objective function. Thus this process is one of valley descending in which the object function is the energy level.

Physical Annealing

- Physical substances are melted and then gradually cooled until some solid state is reached.
- The goal is to produce a minimal-energy state.
- Annealing schedule: if the temperature is lowered sufficiently slowly, then the goal will be attained.
- Nevertheless, there is some probability for a transition to a higher energy state: $e^{-\Delta E/kT}$.

The probability that a transaction to a higher energy state will occur and so given by a function:

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

- ΔE is the +ve level in the energy level
- T is the temperature
- k is Boltzmann's constant

The rate at which the system is cooled is called annealing schedule in an analogous process. The units for both E and T are artificial. It makes sense to incorporate k into T.

Algorithm:

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

$$\Delta E = (value\ of\ current) - (value\ of\ new\ state)$$

        (i) If the new state is goal state then return it and quit
        (ii) 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.
        (iii) 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.

Note:

For each step we check the probability of the successor with the current state. If it is greater than the current state the move is accepted. Otherwise move is rejected and search in other direction.

$$Current\ State\ p = 0.45\ and\ New\ State\ p' = 0.36 \rightarrow (p > p') - Move\ is\ Rejected$$
$$Current\ State\ p = 0.45\ and\ New\ State\ p' = 0.66 \rightarrow (p < p') - Move\ is\ Accepted$$

Best-First Search

Best-First Search (BFS) is a way of combining the advantages of both depth-first search and breadth first search into a single method, i.e., is to follow a single path at a time but switch paths whenever completing path looks more promising than the current one does.

The process is to select the most promising of the new nodes we have generated so far. We then expand the chosen node by using the rules to generate its successors. If one of them is a solution, then we can quit, else repeat the process until we search goal.

In BFS, one move is selected, but others are kept around so that they can be revisited later if the selected path becomes less promising. This is not the case steepest ascent climbing.

OR Graphs

A graph is called OR graph, since each of its branches represents alternative problems solving path.

To implement such a graph procedure, we will need to use lists of nodes:

1) *OPEN*: nodes that have been generated and have had the heuristic function applied to them which have not yet been examined. It is a priority queue in which the elements with highest priority are those with the most promising value of the heuristic function.
2) *CLOSED*: nodes that have already been examined whenever a new node is generated we need to check whether it has been generated before.
3) A heuristic function f which will estimate the merits of each node we generate.

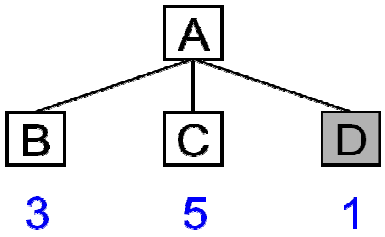
Algorithm:

1. Start with OPEN containing just the initial state
2. Until a goal is found or there are no nodes left on OPEN do:
    a. Pick the best node on OPEN
    b. Generate its successors
    c. For each successor do:
        i. If it is not been generated before, evaluate it, add it to OPEN and record its parent.
        ii. If it has been generated before, change the parent if this new path is better than the previous one. In that case update the cost of getting to this node and to any successors that this node may already have.

Step 1:

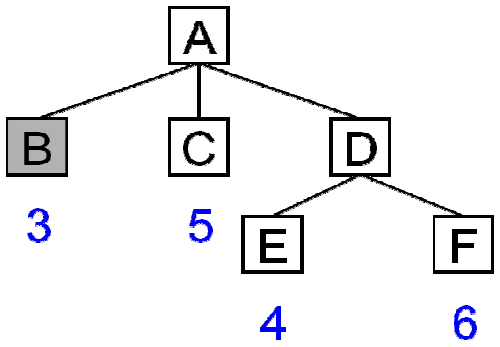
A → NIL

Step 2:



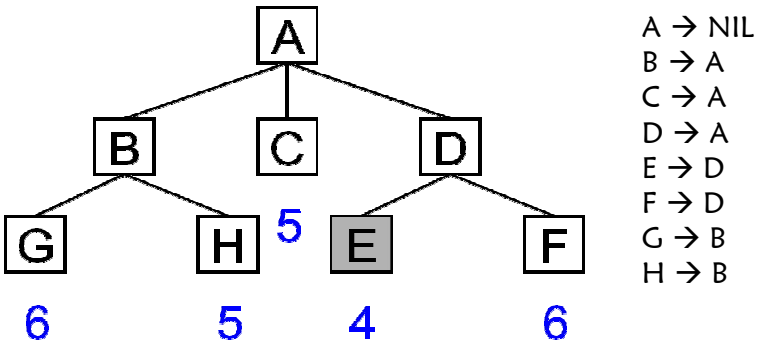
A → NIL
B → A
C → A
D → A

Step 3:

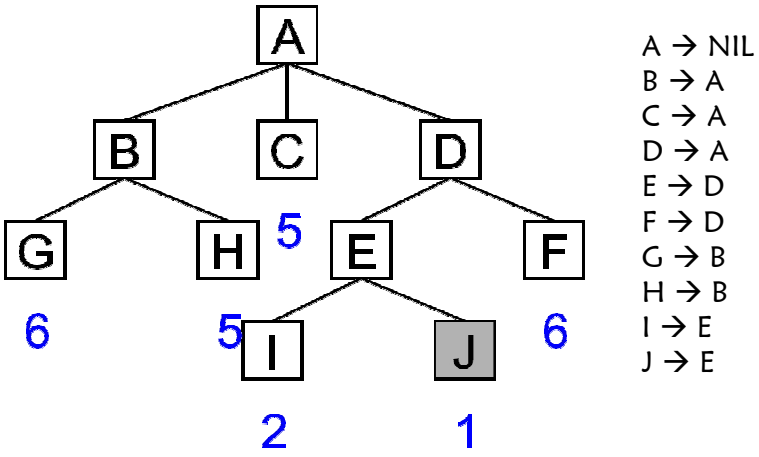


A → NIL
B → A
C → A
D → A
E → D
F → D

Step 4:



Step 5:



The Element with the low cost is the first element. The new states are added according to the cost value.

A* Algorithm:

A* algorithm is a best first graph search algorithm that finds a least cost path from a given initial node to one goal node. The simplification of Best First Search is called A* algorithm. This algorithm uses f', g *and* h' functions as well as the lists OPEN and CLOSED.

For many applications, it is convenient to define function as the sum of two components that we call g and h'.

$$f' = g + h'$$

- g :
  - Measures of the cost of getting from the initial state to the current node.
  - It is not the estimate; it is known to be exact sum of the costs.
- **h' :**
  - is an estimate of the additional cost of getting from current node to goal state.

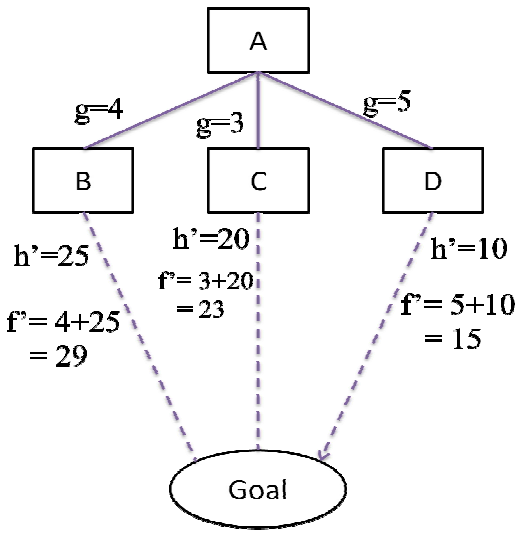
Algorithm:

1) Start with OPEN containing only the initial state (node) set that node g value 0 its h' value to whatever it is and its f' value $h'+0$ or h'. Set CLOSED to the empty list.

2) Until a goal node is found repeat the following procedure: If there are no nodes on OPEN, report failure. Otherwise pick the node on OPEN with lowest f' value. CALL it BESTNODE. Remove from OPEN. Place it on CLOSED. If BESTNODE is the goal node, exit and report a solution. Otherwise, generate the successors of BESTNODE. For each successor, do the following
   a) Set successors to point back to BESTNODE this backwards links will make possible to recover the path once a solution is found.
   b) Compute
   $$g(successor) = g(BESTNODE) + cost\ of\ getting\ from\ BESTNODE\ to\ successor$$
   c) If successor is already exist in OPEN call that node as OLD and we must decide whether OLD' s parent link should reset to point to BESTNODE (graphs exist in this case)

      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 on OPEN, see if it is on CLOSED. If so, call node on CLOSED OLD and add OLD to the list of BESTNODE successors. Calculate all the g, f' and h' values for successors of that node which is better then move that.

      So to propagate the new cost downward, do a depth first traversal of the tree starting at OLD, changing each nodes value (and thus also its f'value), terminating each branch when you reach either a node with no successor or a node which an equivalent or better path has already been found.
   e) If successor was not already on either OPEN or CLOSED, then put it on OPEN and add it to the list of BESTNODE successors. Compute
   $$f'(successor) = g(successor) + h'(successor)$$



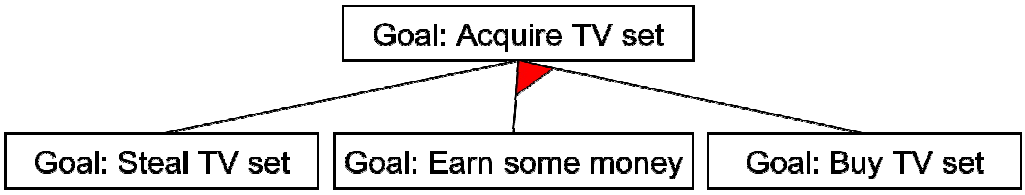
A* algorithm is often used to search for the lowest cost path from the start to the goal location in a graph of visibility/quad tree. The algorithm solves problems like 8-puzzle problem and missionaries & Cannibals problem.

Problem Reduction:

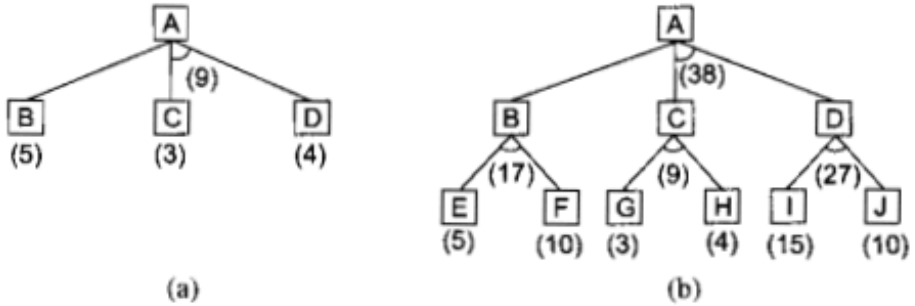
- Planning how best to solve a problem that can be recursively decomposed into sub-problems in multiple ways.
- There can be more than one decompositions of the same problem. We have to decide which is the best way to decompose the problem so that the total solution or cost of the solution is good.
- Examples:
  - Matrix Multiplication
  - Towers of Hanoi
  - Blocks World Problem
  - Theorem Proving
- Formulations: (AND/OR Graphs)
  - An OR node represents a choice between possible decompositions.
  - An AND node represents a given decomposition.

The AND-OR graph (or tree) is useful for representing the solution of problems that can be solved by decomposing them into a set of smaller problems, all of which must then be solved. This decomposition or reduction generate arcs that we call AND arcs.

One AND arc may point to any number of successors nodes all of which must be solved in order for the arc to point to a solution. Just as in OR graph, several arcs may emerge from a single node, indicating a variety of ways in which the original problem might be solved.



AND-OR Graphs



AND-OR Graphs

In order to find solutions in an AND-OR graph, we need an algorithm similar to best-first search but with the ability to handle the AND arcs appropriately.

To see why our Best-First search is not adequate for searching AND-OR graphs, consider **Fig (a)**.

- The top node A has been expanded, producing 2 arcs, one leading to B and one leading to C and D. The numbers at each node represent the value of f' at that node.
- We assume for simplicity that every operation has a uniform cost, so each arc with a single successor has a cost of 1 and each AND arc with multiple successors has a cost of 1 for each of its components.
- If we look just at the nodes and choose for expansion the one with the lowest f' value, we must select C. It would be better to explore the path going through B since to use C we must also use D, for a total cost of 9 (C+D+2) compared to the cost of 6 that we get through B.
- The choice of which node to expand next must depend not only on the f' value of that node but also on whether that node is part of the current best path from the initial node.

The tree shown in **Fig (b)**

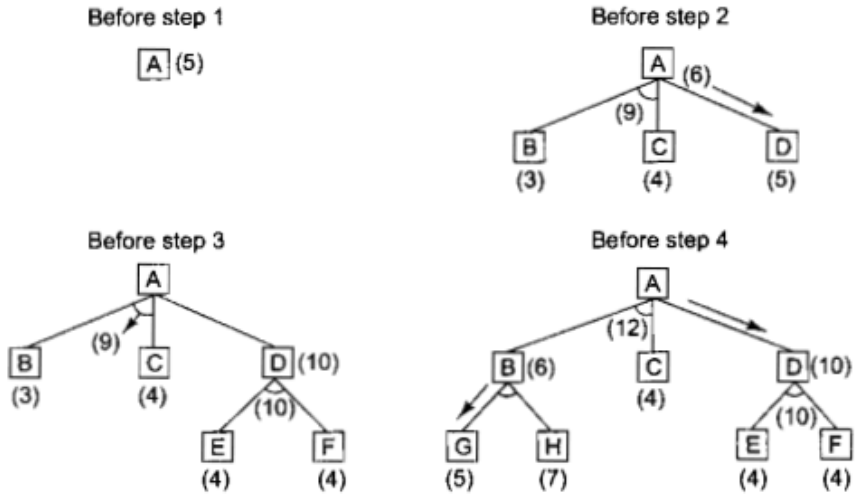
- The most promising single node is G with an f' value of 3. It is even part of the most promising arc G-H, with a total cost of 9. But that arc is not part of the current best path since to use it we must also use the arc I-J, with a cost of 27.
- The path from A, through B, to E and F is better, with a total cost of 18. So we should not expand G next; rather we should examine either E or F.

In order to describe an algorithm for searching an AND-OR graph we need to exploit a value that we call FUTILITY. If the estimated cost of a solution becomes greater than the value of FUTILITY, then we abandon the search. FUTILITY should be chosen to correspond to a threshold such any solution with a cost above it is too expensive to be practical even if it could ever be found.

Algorithm:

1. Initialize the graph to the starting node.
2. Loop until the starting node is labeled SOLVED or until its cost goes above FUTILITY:
    a. Traverse the graph, starting at the initial node following the current best path and accumulate the set of nodes that are on that path and have not yet been expanded or labeled solved.
    b. Pick up one of those unexpanded nodes and expand it. If there are no successors, assign FUTILITY as the value of this node. Otherwise add the successors to the graph and each of this compute f' (use only h' and ignore g). If f' of any node is "0", mark the node as SOLVED.
    c. Change the f' estimate of the newly expanded node to reflect the new information provided by its successors. Propagate this change backward through the graph. If any node contains a successor whose descendants are all solved, label the node itself as SOLVED. At each node that is visible while going up the graph, decide which of its successors arcs is the most promising and mark it as part of the

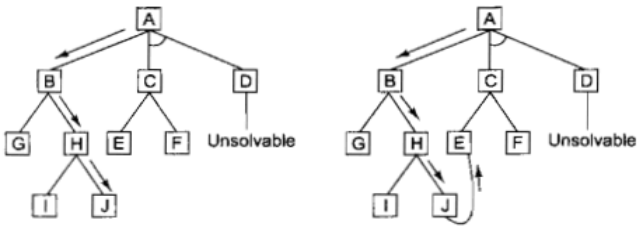
current best path. This may cause the current best path to change. The propagation of revised cost estimates backup the tree was not necessary in the best-first search algorithm because only unexpanded nodes were examined. But now expanded nodes must be reexamined so that the best current path can be selected. Thus it is important that their f' values be the best estimates available.



The Operation of Problem Reduction .

- At **Step 1**, A is the only node, so it is at the end of the current best path. It is expanded, yielding nodes B, C and D. The arc to D is labeled as the most promising one emerging from A, since it costs 6 compared to B and C, which costs 9.
- In **Step 2**, node D is chosen for expansion. This process produces one new arc, the AND arc to E and F, with a combined cost estimate of 10. So we update the f' value of D to 10.
- We see that the AND arc B-C is better than the arc to D, so it is labeled as the current best path. At **Step 3**, we traverse that arc from A and discover the unexpanded nodes B and C. If we are going to find a solution along this path, we will have to expand both B and C eventually. SO explore B first.
- This generates two new arcs, the ones to G and to H. Propagating their f' values backward, we update f' to B to 6. This requires updating the cost of AND arc B-C to 12 (6+4+2). Now the arc to D is again the better path from A, so we record that as the current best path and either node E or F will be chosen for the expansion at **Step 4**.

This process continues until either a solution is found or all paths have led to dead ends, indicating that there is no solution.



Limitations

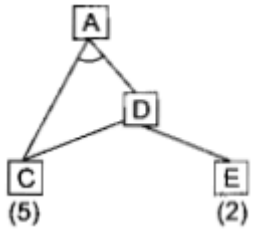
1. **A longer path may be better**

   In Fig (a), the nodes were generated. Now suppose that node J is expanded at the next step and that one of its successors is node E, producing the graph shown in Fig (b). The new path to E is longer than the previous path to E going through C. Since the path

through C will only lead to a solution if there is also a solution to D, which there is not. The path through J is better.

While solving any problem please don't try to travel the nodes which are already labeled as solved because while implementing it may be struck in loop.

2. Interactive Sub-goals



Another limitation of the algorithm fails to take into account any interaction between sub-goals. Assume in figure that both node C and node E ultimately lead to a solution; our algorithm will report a complete solution that includes both of them. The AND-OR graph states that for A to be solved, both C and D must be solved. But the algorithm considers the solution of D as a completely separate process from the solution of C.

While moving to the goal state, keep track of all the sub-goals we try to move which one is giving an optimal cost.

AO* Algorithm:

AO* Algorithm is a generalized algorithm, which will always find minimum cost solution. It is used for solving cyclic AND-OR graphs The AO* will use a single structure GRAPH representing the part of the search graph that has been explicitly generated so far. Each node in the graph will point both down to its immediate successors and up to immediate predecessors. The top down traversing of the best-known path which guarantees that only nodes that are on the best path will ever be considered for expansion. So h' will serve as the estimate of goodness of a node.

Algorithm (1):

1) Initialize: Set G* = {s}, f(s) = h(s).
   If $s \in T$, label s as SOLVED, where T is terminal node.
2) Terminate: If s is SOLVED then Terminate
3) Select: Select a non-terminal leaf node n from the marked sub tree
4) Expand: Make explicit the successors of n.
   For each new successor, m: Set f(m) = h(m)
   If m is Terminal, label m as SOLVED.
5) Cost Revision: Call cost-revise(n)
6) Loop: Goto Step 2.

Cost Revision

1. Create Z = { n }
2. Z = { } return
3. Otherwise: Select a node m from z such that m has no descendants in Z
4. If m is an AND node with successors
   r1, r2, …, rk

Set $f(\mathrm{m}) = \sum\ [\ f(ri)\ +\ c(m,ri)\ ]$

Mark the edge to each successor of m. If each successor is labeled SOLVED then label m as SOLVED.

5. If m is an OR node with successors
   r1, r2, …, rk
   Set $f(\mathrm{m}) = \min\{\ f(ri)\ +\ c(m,ri)\ \}$
   Mark the edge to each successor of m. If each successor is labeled SOLVED then label m as SOLVED.
6. If the cost or label of m has changed, then insert those parents of m into Z for which m is marked successor.

Algorithm (2):

1. Let *GRAPH* consist only of the node representing the initial state. (Call this node *INIT.*) Compute $h'(INIT)$
2. Until *INIT* is labeled *SOLVED* or until *INIT's* h' value becomes greater than *FUTILITY,* repeat the following procedure:
   (a) Trace the labeled arcs from *INIT* and select for expansion one of the as yet unexpanded nodes that occurs on this path. Call the selected node *NODE.*
   (b) Generate the successors of *NODE.* If there are none, then assign *FUTILITY* as the h' value of *NODE.* This is equivalent to saying that *NODE* is not solvable. If there are successors, then for each one (called *SUCCESSOR)* that is not also an ancestor of *NODE* do the following:
      (i) Add *SUCCESSOR* to *GRAPH.*
      (ii) If *SUCCESSOR* is a terminal node, label it *SOLVED* and assign it an h' value of 0.
      (iii) If *SUCCESSOR* is not a terminal node, compute its h' value.
   (c) Propagate the newly discovered information up the graph by doing the following: Let *S* be a set of nodes that have been labeled *SOLVED* or whose h' values have been changed and so need to have values propagated back to their parents. Initialize 5 to *NODE.* Until *S* is empty, repeat the, following procedure:
      (i) If possible, select from *S* a node none of whose descendants in *GRAPH* occurs in *S.* If there is no such node, select any node from *S.* Call this node *CURRENT,* and remove it from *S.*
      (ii) Compute the cost of each of the arcs emerging from *CURRENT.* The cost of each arc is equal to the sum of the h' values of each of the nodes at the end of the arc plus whatever the cost of the arc itself is. Assign as *CURRENT'S* new h' value the minimum of the costs just computed for the arcs emerging from it.
      (iii) Mark the best path out of *CURRENT* by marking the arc that had the minimum cost as computed in the previous step.
      (iv) Mark *CURRENT SOLVED* if all of the nodes connected to it through the new labeled arc have been labeled *SOLVED.*
      (v) If *CURRENT* has been labeled *SOLVED* or if the cost of *CURRENT* was just changed, then its new status must be propagated back up the graph. So add all of the ancestors of *CURRENT* to *S.*

Means-Ends Analysis:

One general-purpose technique used in AI is **means-end analysis**, a step-by-step, or incremental, reduction of the difference between the current state and the final goal. The program selects actions from a list of means–in the case of a simple robot this might consist of PICKUP, PUTDOWN, MOVEFORWARD, MOVEBACK, MOVELEFT, and MOVERIGHT–until the goal is reached. This means we could solve major parts of a problem first and then return to smaller problems when assembling the final solution.

Usually, we search strategies that can reason either forward or backward. Often, however a mixture of the two directions is appropriate. Such mixed strategy would make it possible to solve the major parts of problem first and solve the smaller problems arise when combining them together. Such a technique is called "**Means - Ends Analysis**".

This process centers on the detection of difference between the current state and goal state. After the difference had been found, we should find an operator which reduces the difference. But this operator cannot be applicable to the current state. Then we have to set up a sub-problem of getting to the state in which it can be applied if the operator does not produce the goal state which we want. Then we should set up a sub-program of getting from state it does produce the goal. If the chosen inference is correct, the operator is effective, then the two sub-problems should be easier to solve than the original problem.

The means-ends analysis process can be applied recursively to them. In order to focus system attention on the big problems first, the difference can be assigned priority levels, in which high priority can be considered before lower priority.
Like the other problems, it also relies on a set of rules rather than can transform one state to another these rules are not represented with complete state description. The rules are represented as a left side that describes the conditions that must be met for the rule applicable and right side which describe those aspects of the problem state that will be changed by the application of the rule.

Consider the simple HOLD ROBOT DOMAIN. The available operators are as follows:

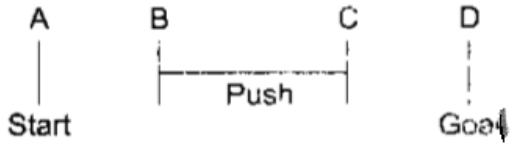
OPERATOR	PRECONDITIONS	RESULTS
PUSH(obj,loc)	At(robot,obj) ^ large(obj) ^ clear(obj) ^ armempty	At(obj,loc) ^ at(robot,loc)
CARRY(obj,loc)	At(robot,obj) ^ small(obj)	At(obj,loc) ^ at(robot,loc)
WALK(loc)	NONE	At(robot,loc)
PICKUP(obj)	At(robot,obj)	Holding(obj)
PUTDOWN(obj)	Holding(obj)	7 Holding(obj)
PLACE(obj1,obj2)	At(robot,obj2) ^ Holding (obj1)	On(obj1,obj2)

Difference Table

	Push	Carry	Walk	Pickup	Putdown	Place
Move object	*	*				
Move robot			*			
Clear object				*		
Get object on object						*
Get arm empty					*	*
Be holding object				*		

A Difference Table

The difference table describes where each of the operators is appropriate table:

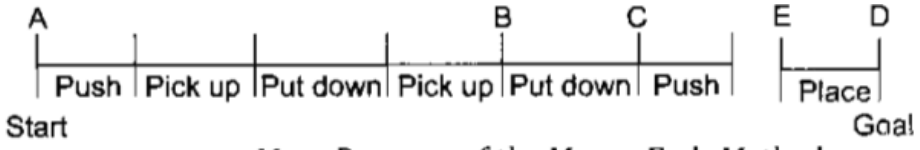


The Progress of the Means-Ends Analysis Method

Suppose that the robot were given the problems of moving desk with two things on it from one room to another room. The objects on top must also be moved the difference between start and goal is the location of the desk.

To reduce the difference either PUSH or CARRY can be chosen. If the CARRY is chosen first its precondition must be met. These results in two more differences that must be reduced; the location of the robot and the size of the desk. The location of the robot can be handled by applying WALK, but there are no operators that can change the size of the objects. So their path problem solve program will be shown above AND here also the thing does not get it quit to the goal state. So now the difference between A, B and between C, D must be reduced.

PUSH has 4-preconditions. Two of which produce difference between start and goal states since the desks is already large. One precondition creates no difference. The ROBOT can be brought to the location by using WALK, the surface can be cleared by two uses of pickup but after one pick-up the second results in another difference – the arm must be empty. PUTDOWN can be used to reduce the difference.



More Progress of the Means-Ends Method

One PUSH is performed; the problem state is close to the goal state, but not quite. The objects must be placed back on the desk. PLACE will put them there. But it cannot be applied immediately. Another difference must be eliminated, since the robot is holding the objects. Then we will find the progress as shown above. The final difference between C and E can be reduced by using WALK to get the ROBOT back to the objects followed by PICKUP and CARRY.

Algorithm:

1. Until the goal is reached or no more procedures are available:
   - Describe the current state, the goal state and the differences between the two.

- Use the difference the describe a procedure that will hopefully get nearer to goal.
- Use the procedure and update current state.

2. If goal is reached then **success** otherwise **fail**.

Algorithm:

Algorithm: Means-Ends Analysis (CURRENT, GOAL)

1. Compare *CURRENT* to *GOAL*. If there are no differences between them then return.
2. Otherwise, select the most important difference and reduce it by doing the following until success or failure is signaled:
   (a) Select an as yet untried operator *O* that is applicable to the current difference. If there are no such operators, then signal failure.
   (b) Attempt to apply *O* to *CURRENT*. Generate descriptions of two states: *O-START*, a state in which *O*'s preconditions are satisfied and *O-RESULT*, the state that would result if *O* were applied in *O-START*.
   (c) If
   (FIRST-PART ← MEA(CURRENT, O-START))
   and
   (LAST-PART ← MEMO-RESULT, GOAL))
   are successful, *then* signal success and return the result of concatenating *FIRST-PART*, *O*, and *LAST-PART*.

Constraint Satisfaction

- Search procedure operates in a space of constraint sets. Initial state contains the original constraints given in the problem description.
- A goal state is any state that has been constrained enough – *Cryptarithmetic:* "enough" means that each letter has been assigned a unique numeric value.
- Constraint satisfaction is a 2-step process:
  - Constraints are discovered and propagated as far as possible.
  - If there is still not a solution, then search begins. A guess about is made and added as a new constraint.
- To apply the constraint satisfaction in a particular problem domain requires the use of 2 kinds of rules:
  - Rules that define valid constraint propagation
  - Rules that suggest guesses when necessary

Problem:

```
  SEND
+ MORE
.....
 MONEY
```

Initial State:

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

Fig. 3.13 *A Cryptarithmetic Problem*

Algorithm: Constraint Satisfaction

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 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.

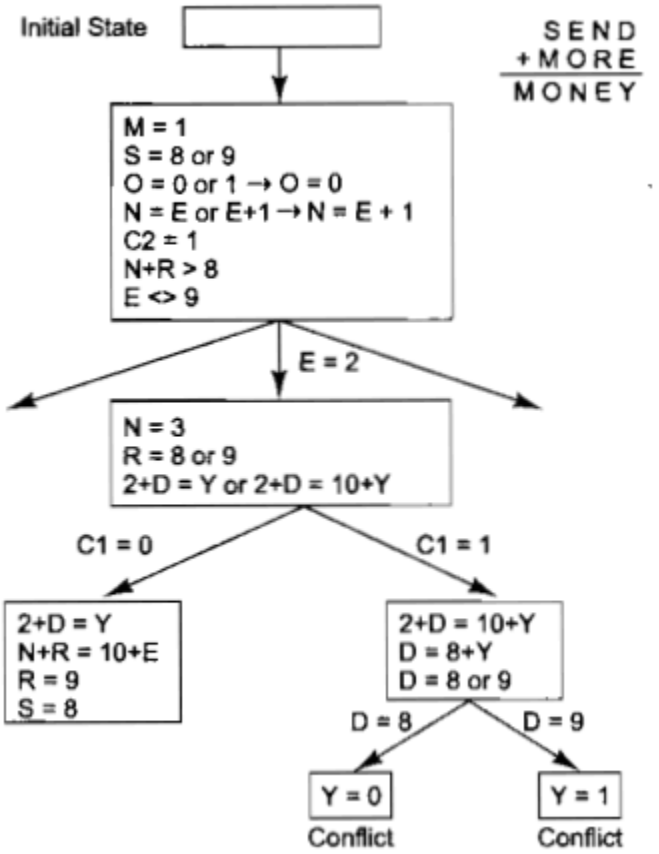


Fig. 3.14 *Solving a Cryptarithmetic Problem*

Goal State:

- We have to assign unique digit for the above specified alphabets.

→ 1. SEND
MORE
MONEY

0 1 2 3 4 5 6 7 8 9

Sol:-

C_3 C_2 C_1
S E N D
M O R E
M O N E Y

0 1 2 3 4 5 6 7 8 9
↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
O M Y E N O R S

1) Two single digit numbers sum is maximum 18 or 19.
(one is carry from previous addition)

So we conclude that m=1

2) $S + m + C_3 \geq 10$

$S + 1 + C_3 \geq 10$

The carry may be $C_3 = 0$ or 1.

$S + 1 + 0 \geq 10 \Rightarrow S \geq 9 \Rightarrow S = 9$.

$S + 1 + 1 \geq 10 \Rightarrow S \geq 8 \Rightarrow S = 8$ or 9

If $S = 9 \Rightarrow 9 + 1 + C_3 \Rightarrow C_3 = 0 \Rightarrow \frac{1}{m}\ \frac{0}{O}$ ✓

$C_3 = 1 \Rightarrow \frac{1}{m}\ \frac{1}{O}$

$S = 8 \Rightarrow 8 + 1 + C_3 \Rightarrow C_3 = 1 \Rightarrow \frac{1}{m}\ \frac{0}{O}$

3) let us consider $S = 9$ ✓

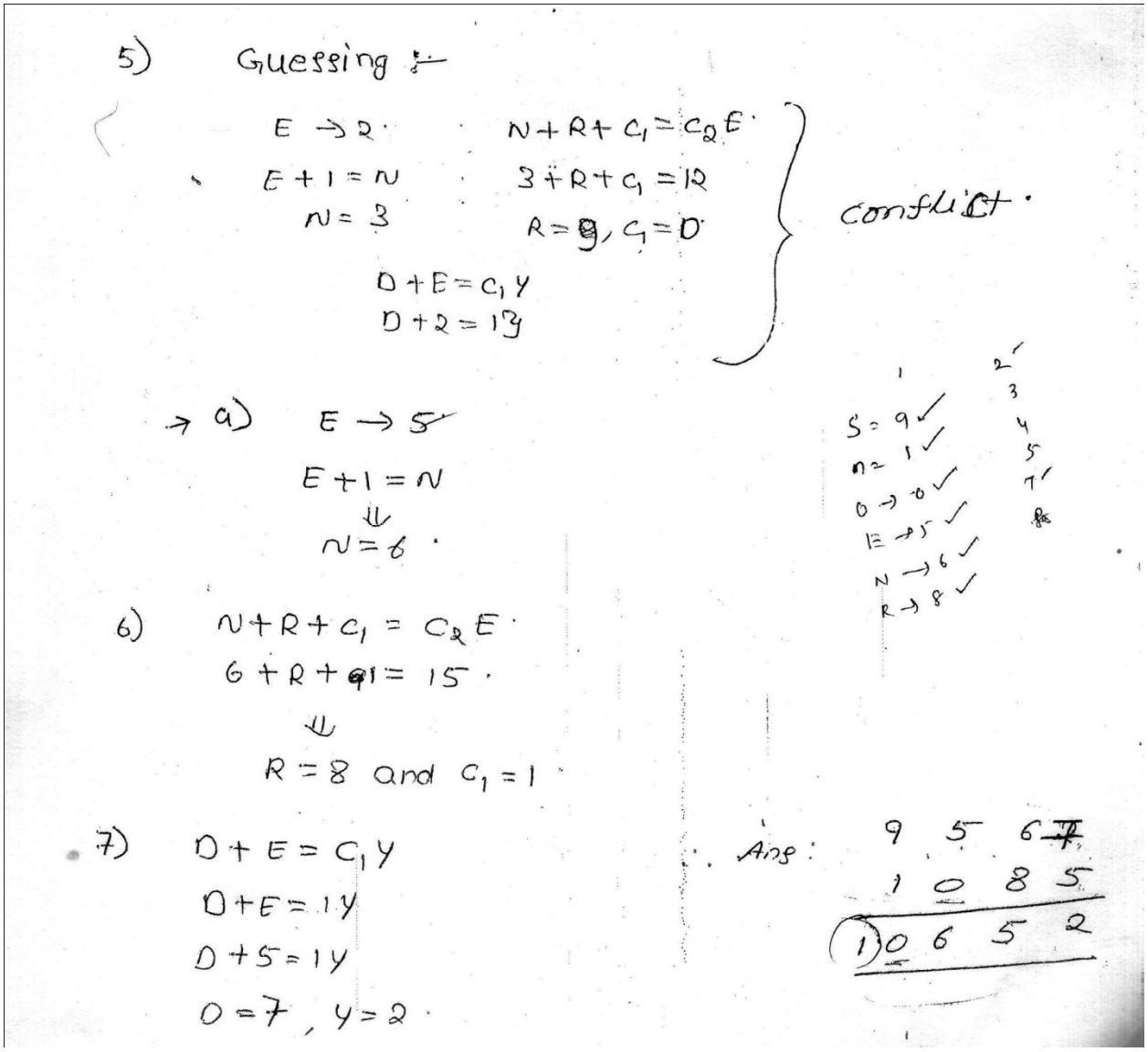
4) $E + O + C_2 = N$

$E + 0 + C_2 = N$

$E + 1 = N$

it implies C_2 must be 1

$C_2 = 1$


5) Guessing :-
E → 2
E + 1 = N
N = 3
N + R + C1 = C2 E
3 + R + C1 = 12
R = 9, C1 = 0
conflict.
D + E = C1 Y
D + 2 = 13
→ a) E → 5
E + 1 = N
N = 6
S = 9
n = 1
0 → 0
E → 5
N → 6
R → 8
6) N + R + C1 = C2 E
6 + R + 01 = 15
R = 8 and C1 = 1
7) D + E = C1 Y
D + E = 1Y
D + 5 = 1Y
D = 7, Y = 2
Ans:
9 5 6 7
1 0 8 5
1 0 6 5 2

FREQUENTLY ASKED QUESTIONS

1) Define Intelligence, Artificial Intelligence.
2) List four things to build a system to solve a problem.
3) What is Production System?
4) Explain water Jug problem as a state space search.
5) Explain production system characteristics.
6) Explain A* algorithm with example.
7) What is Means-Ends Analysis? Explain with an example.
8) What do you mean by heuristic?
9) Write a heuristic function for travelling salesman problem.
10) What is heuristic search?
11) Explain problem characteristics.
12) Write AO* algorithm and explain the steps in it.
13) What is constraint satisfaction problem? Explain it.
14) Explain annealing schedule.
15) Explain Breadth-first search and depth-first search. List down the advantages and disadvantages of both?
16) What do you mean by an AI technique?
17) Discuss the tic-tac-toe problem in detail and explain how it can be solved using AI techniques.
18) What are the advantages of Heuristic Search?
19) Explain Turing Test as Criteria for success.
20) Explain Hill Climbing and give its disadvantages.
21) Define Control Strategy and requirements for good search strategy.
22) Define State Space Search. Write algorithm for state space.