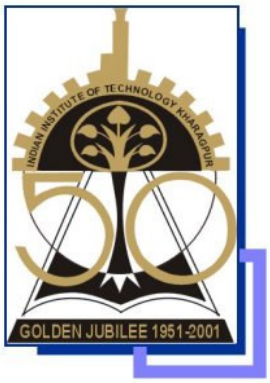
AI - Techniques

By
Kaushik Malakar

A project report submitted in partial fulfillment of the Requirements for the degree of

B.SC. IN MATHEMATICS & COMPUTING

UNDER THE GUIDANCE OF
DR. P. KUMAR



Department of Mathematics
INDIAN INSTITUTE OF TECHNOLOGY
Kharagpur -721302

December 2001

Indian Institute of Technology
Kharagpur - 721302

CERTIFICATE

This is to certify that the project entitled, **"AI Techniques"** submitted by Kaushik Malakar, (Roll number 9925219), to the Indian Institute of Technology, Kharagpur, is a bonafide record of the work carried out by my supervision and guidance.

DR. Pawan Kumar
Professor, Department of Mathematics
Indian Institute of Technology
Kharagpur - 721302

To my parents, Mr. Madhusudan Malakar and Swapna Malakar, who brought me up despite the stress and complexities of their lives and devoted themselves to my education;

To my teachers, who taught me the art of reacting to a changing environment; and

To millions of the poor and downtrodden people of my country and the world, whose sacrifice and tolerance paved the royal road of my education, and whose love and emotion, smile and tears inspired me to speak their thoughts in my words.

Kaushik Malakar

ACKNOWLEDGEMENTS

Acknowledgement is not a mere formality but a genuine opportunity to express the sincere thanks to all those without whose active support and encouragement this project wouldn't have been successful .

I express my deep sense of regards and indebtedness to my guide Dr . P.Kumar for his valuable guidance, continuous encouragement and wholehearted support , which were of immense help to me in completing the project . In spite of his hectic schedule he has always extended his help and invaluable suggestions.

I express my thanks to Dr .S. Nanda , HOD, Department of Mathematics and to all the faculty members and the staff of the Department of Mathematics for their continuous help .

Words are not enough to express my indebtedness and gratitude towards my parents to whom I owe every success and achievements of my life. Their constant support and encouragement under all odds that has brought me where I stand today.

Finally I thank all my friends specially Sudipta Kundu and S.Shreenath for their love , cheerful encouragements and valuable criticism .

CONTENTS

TITLE

1.1 Preamble

The phrase "Artificial Intelligence" can be defined as the simulation of human intelligence on a machine, so as to make the machine efficient to identify and use the right piece of "Knowledge" at a given step to solve a problem. A system capable of planning and executing the right task at the right time is generally called rational. Thus, artificial Intelligence alternatively may be stated as a subject dealing with computational models that can think and act rationally. A little thinking, reveals that a system that can reason well must be a successful planner, as planning in many circumstances is a part of a reasoning problem. Further, a system can act rationally only after acquiring adequate knowledge from the real world. So, perception that stands for building up of knowledge from real world information is a prerequisite feature for rational action. The rational acting of an agent, thus calls for possession of all the elementary characteristics of intelligence. Relating Artificial Intelligence with computational models capable of thinking and acting rationally, therefore, has a pragmatic significance.

The subject of Artificial Intelligence spans a wide horizon. It deals with the various kinds of knowledge representation schemes, different techniques of intelligent search, various methods for resolving uncertainty of data and knowledge, different schemes for automated machine learning and many others. Among the application areas of Artificial Intelligence, we have Expert system, Game-Playing and Theorem – proving, Natural language processing, Image recognition, Robotics and many others. The subject of Artificial Intelligence has been enriched with a wide discipline of knowledge from Philosophy, Psychology, Cognitive Science, Computer Science, Mathematics and Engineering. Thus in the following figure they have been referred to as the parent disciplines of Artificial Intelligence. An at a glance look of the figure also reveals the subject area of Artificial Intelligence and its application areas.

The subject of Artificial Intelligence was originated with game-playing and theorem-proving programs and gradually enriched with theories from a number of parent disciplines. As a young discipline of science, the significance of the topics covered under the subject changes considerably with time. At present, the topics which we find significantly and worthwhile to understand the subjects are: -

- **Learning System**
- **Knowledge Representation and Reasoning**
- **Knowledge Base:**
    - **Planning**
    - **Knowledge Acquisition**
    - **Intelligent search**
- **Logic computing**
- **Soft Computing**
    - **Fuzzy Logic**
    - **Artificial Neural Nets**
- **Genetic Algorithms**

1.2 Motivation

At the beginning of the Stone Age, when people started taking shelters in caves, they made attempts to immortalize themselves by painting their images on rocks. With the gradual progress in civilization, they felt interested to see themselves in different forms. So, they started constructing models of human being with sand, clay and stones. The size, shape, constituents and style of the model humans continued evolving but the man was not happy with the models that only looked like him. He had a strong desire to make the model 'intelligent', so that it could act and think as he did.

1.3 Objective

To understand what exactly Artificial Intelligence is, we illustrate some common problems. Problems dealt with in Artificial Intelligence generally use a common term called '**state**'. A state represents a status of the solution at a given step of the problem solving procedure. The solution of a problem thus is a collection of the problem states. The problem solving procedure applies an operator to a state to get the next stage. Then it applies another operator to a state and its subsequent transition to the next state, thus is continued until the goal (desired) state is derived. Thus our main objective is to understand the basics of various concepts of AI – Techniques in the simplest possible form.

1.4 Review of Extracts

Some of these well-known search algorithms are:

a) Generate and test
b) Hill Climbing
c) Heuristic-Search
d) Means and Ends analysis

(a) Generate and Test approach: This approach concerns the generation of the state-space from a known staring state (root) of the problem and continues expanding the reasoning space until the goal node or the terminal state is reached. In fact after generation of each and every state, the generated node is compared with the known goal state. When the goal is found, the algorithm terminates. In case there exist multiple paths leading to the goal, then the path having the smallest distance from the root is preferred. The basic strategy used in this search is only generation of states and their testing for goals but it does not allow filtering of states.

(b) Hill Climbing Approach: Under this approach, one has to first generate a starting state and measure the total cost for reaching the goal from the given starting state. Let this cost be f. while $f \leq$ a predefined utility value and the goal is not reached, new nodes are generated as children of the current node. However, in case all the neighborhood nodes (states) yield an identical value of f and the goal is not included in the set of these nodes, the search algorithm is trapped at a hillock or local extrema. One way to overcome this problem is to select randomly a new starting state and then continue the above search process. While proving trigonometric identities, we often use Hill climbing, perhaps unknowingly.

(c) Heuristic Search: -Classically heuristics means rule of thumb. In heuristic search, we generally use one or more heuristic functions to determine the better candidate states among a set of legal states that could be generated from a known state. The heuristic function, in other words, measures the fitness of the candidate states, the better the selection of the states, the fewer will be the number of intermediate states for reaching the goal. However, the most difficult task in heuristic search problems is the selection of the heuristic functions. One has to select them intuitively, so that in most cases hopefully it would be able to prune the search space correctly.

(d) Means and Ends Analysis: - This method of search attempts to reduce the gap between the current state and the goal state. One simple way to explore this method is to measure the distance between the current state and the goal, and then apply an operator to the current state, so that the distance between the resulting state and the goal is reduced. In many mathematical theorem-proving processes, we use Means and End Analysis.

1.5 Outline of Work

Problem solving requires two prime considerations: first representation of the problem by an appropriately organized state space and then testing the existence of a well-defined goal state in that space. Identification of the goal state and determination of the optimal path, leading to the goal through one or more transitions form a given starting state, will be addressed here in sufficient details, we, thus, starts with some well-known search algorithms, such as the depth first and the breadth first search, with special emphasis on their results of time and space complexity. It then gradually explores the "heuristics search" algorithms, where the order of visiting the states in a search space is supported by thumb rules, called heuristics, and demonstrates their applications in complex problem solving. We also discuss some intelligent search algorithms for game playing.

2.1 Introduction - *Tic Tac Toe*

One of the major applications of tree is to game playing by computer. Our main objective here is to determine the "best" move in tic-tac-toe from a given board position. Tic-tac-toe is a two user simple game. Given a board position and the player, our main objective is to find a value that represents how "good" the position seems to be for that player (the larger the value the better is the position). Of course, a winning position yields the largest value, and a loosing position yields the smallest. An example of such an evaluation function for tic-tac-toe is the number of rows, columns, and diagonals remaining open for one player minus the number the number remaining open for his or her opponent (except that the value of 9 would be returned for a position that wins and –9 for a position that lose). Here we do not "look ahead" to consider any possible board position that might result from the current position. We merely evaluate the static board position.

Given a board position, the best next move could be determined by considering all possible moves and resulting positions. The move selected should be the one that results in the board position with the highest evaluation. Such an analysis, however, does not necessarily yield the best move. The following figure we illustrate a position and the five possible move that X can make from that position. Applying the evaluation function just describe to the five resulting positions yields the values shown.

Four moves yield the same maximum evaluation, although three of them are distinctly inferior to the fourth (The fourth position yields a certain victory for X, whereas the other three can be drawn by O.) In fact, the move that yields the smallest evaluation is as good or better than the moves that yield a higher evaluation. The static evaluation function, therefore, is not good enough to predict the outcome of the game. A better evaluation function could easily be produced for the game of tic-tac-toe (even if it were by the brute-force method of listing all positions and the appropriate response), but most games are too complex for static evaluators to determine the best response.

Suppose that it were possible to look ahead several moves. Then the choice of a move could be improved considerably. Define the **'looks ahead level'** as the number of future moves to be considered. Starting at any position, it is possible to construct a tree of the possible board positions that may result from each move. Such a tree is called a '**game tree'.** The game tree for the opening tic-tac-toe position with a look-ahead level of 2 is illustrated in the figure next page. (Actually other positions do exist, but because of symmetry considerations these are effectively the same as the positions shown.) Note that the maximum level (called the depth) of the nodes in such a tree is equal to the look-ahead level.

Let us designate the player who must move at the root's game position as **plus** and his or her opponent as **minus**. We attempt to find the best move for plus from the root's game position. The remaining nodes of the tree may be designated as **plus nodes** or **minus nodes**, depending upon player must move from that node's position. Each node in the following figure is marked as a plus or a minus node.

Suppose that the game positions of all the sons of a plus node have been evaluated for player plus. Then clearly, plus should choose the move that yields the maximum evaluation. Thus, the value of a plus node to player plus is the maximum of the values of its sons. On the other hand, once plus has moved, minus will select the move that yields the minimum evaluation for player plus. Thus the value of a minus node to player plus is the minimum of the values of its sons.

Therefore to decide the best move for player plus from the root, the position in the leaves must be evaluated for player plus using a static evaluation function. These values are then moved up the game tree by assigning to each plus node the maximum of its sons' values and to each minus node the minimum of its sons' values, on the assumption that minus will select the move that is worse for plus. The value assigned to each node in the following figure by this process is indicated in the figure immediately below the node.

The move that plus should select, given the board position in the root node, is the one that maximizes is value. Thus the opening move for X should be the middle square, as illustrated in the previous figure. The following figure illustrates the determination of O's best reply. Note that the designation of "plus" and 'minus" depends on whose move is being calculated. Thus, in the previous figure X is designated as plus, whereas in the following figure O is designated as plus. In applying the static evaluation function to a board position, the value of the position whichever player is designated as plus is computed. This method is called the **minimax method,** since as the tree is climbed the maximum and minimum functions are applied alternately.

2.2 Implementation

Here we have used the method of **minimax look ahead method** (two-level thinking) for finding the best move for a player. We have used the C-language for programming a simple implementation of tic-tac-toe in Turbo-C^{++}. In order to make the program work more efficiently the program may be modified so that if the evaluation of a minus node is greater than the minimum of the values of its father's old brother, the program does not bother expanding that minus node's younger brothers, and if the evaluation of a plus node is less than the maximum of the values of its father's older brother, the program does not bother expanding that plus node's younger brother. This method is called the **alpha-beta minimax** method.

2.2.1 Algorithm

The minimax algorithm is formally presented below.

Procedure MINIMAX

Begin

1. Expand the entire state-space below the starting node;

2. Assign values to the terminals of the state-space from {-1,0, +1}, depending on the success of the MINIMIZER, draw, or the success of the MAXIMIZER respectively;

3. **For** each node whose all children possess values, **do**

If it is a MAXIMIZER node, **then** its value will be maximum of its **BEGIN** children's value;

If it is a MINIMIZER node, **then** its value will be the minimum of its children;

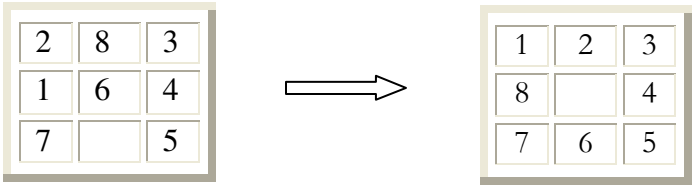
End For;

End

3.1 Introduction - *8 - Puzzle*

Many Artificial Intelligence applications involve composing a sequence of operations. Controlling the action of a robot and automatic programming are two examples. A simple and perhaps familiar problem of this sort, useful for understanding basic ideas is the 8-puzzle.

The 8-puzzle is a small board game for a single player, it consists of 8 square tiles numbered 1 through 8 and one blank space on a 3 x 3 board. (A 15-puzzle, using a 4 x 4 board, is commonly sold as a child's puzzle. We will use an 8-puzzle to keep the search space reasonable.) Moves of the puzzle are made by sliding an adjacent tile into the position occupied by the blank space, which has the effect of exchanging the positions of the tile and blank space. Only tiles that are horizontally or vertically adjacent (not diagonally adjacent) may be moved into the blank space. Such a puzzle is illustrated in the following figure. Two configurations of tiles are given. Consider the problem of changing the initial configuration into the goal configuration. A solution to the problem is an appropriate sequence of moves, such as "move tile 6 down, move tile 8 down, . . . , etc."



Initial Goal

3.2 Production Systems

Various generalization of the computational formalism known as a production system involve a clean separation of these computational components and thus seem to capture the essence of operation of many Artificial Intelligence systems. The major elements of an Artificial Intelligence production system are a global database, a set of production rules, and a control system. The production rules operate on the global database. Each rule has a precondition. If the precondition is satisfied, the rule can be applied.

To solve a problem using a production system, we must specify the global database, the rules, and the control strategy. Transforming a problem statement into these three components of a production system is often called the representation problem in Al. Usually there are several ways to so represent a problem. Selecting a good representation is one of the important arts involved in applying Al techniques to practical problems.

For the 8- puzzle and certain other problems, we can easily identify elements of the problem that correspond to these three components. These elements are the problem states, moves, and goal. In the 8-puzzle each tile configuration is a problem state. The set of all possible configurations is the space of problem states or the problem space. Many of the problem in which we are interested have large problem spaces. The 8- puzzle has a relatively small space; there are only 362,880 (that is, 9!) different configuration of the 8 tiles and the

blank space. (This space happens to be partitioned into two disjoint subspaces of 181,440 states each).

Once the problem states have been conceptually identified, we must construct a computer representation, or description, of them. This description is then used as the global database of a production system. For the 8 - puzzle, a straightforward description is a 3 x 3 array of matrix of numbers. The initial global database is this description of the initial problem state. Virtually any kind of data structure can be used to describe states. These include symbol strings, vectors, sets, arrays, trees, and lists. Sometimes, as in the 8 - puzzle, the form of the data structure bears a close resemblance to some physical property of the problem being solved.

A move transforms one problem state into another state. The 8- puzzle is conveniently interpreted as having the following four moves: Move empty space (blank) to the left, move blank up, move blank to the right, and move blank down. These moves are modeled by production rules that operate on the state descriptions in the appropriate manner. The rules each have preconditions that must be satisfied by a state description in order for them to be applicable to that state description. Thus, the precondition for the rule associate with "move blank up" is derived from the requirement that the blank space must not already be in the top row.

In the 8- puzzle, we are asked to produce a particular problem state, namely, the goal state shown in the previous figure. We can also deal with problems for which the goal is to achieve any one of an explicit list of problem states. A further generalization is to specify some true / false condition on state to serve as a goal condition. Then the goal would be to achieve any state satisfying this condition. Such a condition implicitly defines some set of goal states. For example, in the 8- puzzle, we might want to achieve any tile configuration for which the sum of the numbers labeling the tile in the first row is 6. In our language of states, moves, and goals, a solution to a problem is a sequence of moves that transforms an initial state to a goal state.

The problem goal condition forms the basis for the termination condition of the production system. The control strategy repeatedly applies rules to state descriptions until a description of a goal state is produced. It also keeps track of the rules that have been applied so that it can compose them into the sequences representing the problem solution.

In certain problems, we want the solution to be subject to certain additional constraints. For example, we may want the solution to our 8 - puzzle problem having the smallest number of moves. In general we ascribe a cost to each move and then attempt to find a solution having minimal cost. These elaborations can easily be handled by methods we describe later on.

CONTROL

The above procedure is non-deterministic we have not yet specified precisely how we are going to select an applicable rule. Selecting rules and keeping track of those sequences of rules already tried and the databases they produced constitute what we call the control strategy for production system. In most AI application, the information available to the control strategy is not sufficient to permit selection of the most appropriate rule on every

step. The operation of AI production system can thus be characterized as a search process in which rules are tried until some sequence of them is found that produces a database satisfying the termination condition. Efficient control strategies require enough knowledge about the problem being solved so that the rule selected in step has a good chance of being the most appropriate one.

We distinguish two major kinds of control strategies: **irrevocable and tentative.** In an irrevocable control regime, an applicable rule is selected and applied irrevocable without provision for reconsideration later. In a tentative control regime, an applicable rule is selected (either arbitrarily or perhaps with some good reason), the rule is applied, but provision is made to return later to this point in the computation to apply some other rule.

We further distinguish two different types of tentative control regimes. In one which we call backtracking, a backtracking point is established when a rule is selected. Should subsequent computation encounter difficulty in producing a solution, the state of the computation reverts to the previous backtracking point, where another rule is applied instead and the process continues.

In the second type of tentative control regime, which we call graph - search control, provision is made for keeping track of the effects of several sequences of rules simultaneously. Various kinds of graph structures and graph searching procedures are used in this type of control.

Examples of Control (HILL CLIMBING)

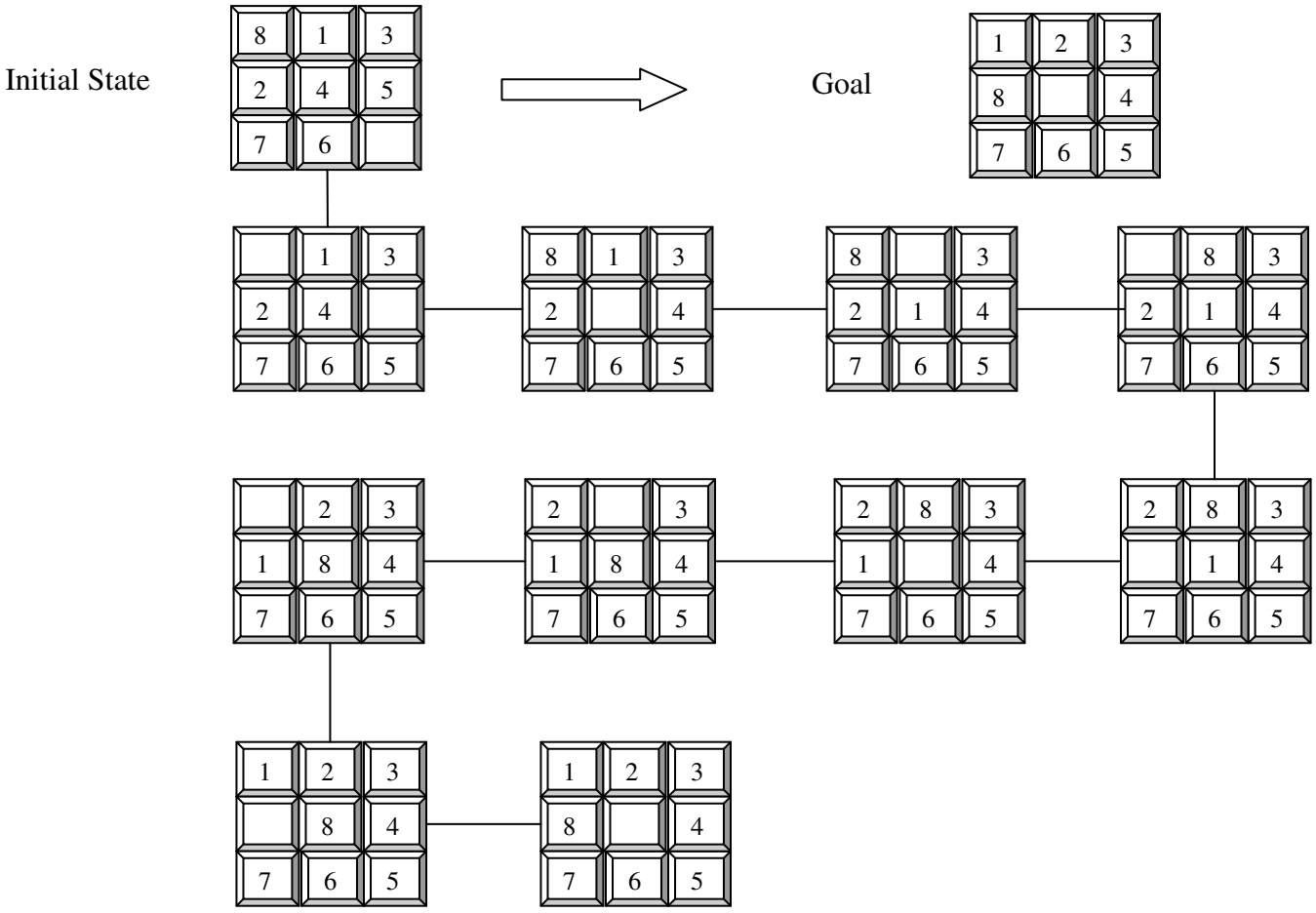
Irrevocable. At first thought, it might seem that an irrevocable control regime would never be appropriate for production systems expected to solve problems requiring search. Trials - and - error methods seem to be inherent in solving puzzles for example. One might argue that if a control strategy of a production system possessed sufficient knowledge about puzzle to select irrevocable an appropriate rule to apply to each state description, then it would have the puzzle's solutions built into it and, if so can hardly be said to have " solved " the puzzle, for it already knew the solutions. Such an argument fails to acknowledge the distinction between the explicit local knowledge about how to proceed toward a goal from any state, and the implicit global knowledge, of the complete solution. When infallible local knowledge is available, an irrevocable production system can use it to construct the explicit global knowledge of a solution (without having the explicit global knowledge originally).

Outside of AI, One of the most common examples of the use of local knowledge to construct global solution is in the "hill - climbing " process of finding the maximum of a function. At any point, we proceed in the direction of the steepest gradient (the local knowledge) to find eventually a maximum of the function (the global knowledge). For certain kinds of functions (those with a single maximum and certain other properties), knowledge of the direction of the steepest gradient is sufficient to find a SOLUTION.

We can use the hill - climbing process directly in an irrevocable production system. We need only some real - valued function on the global database. The control strategy uses this function to select a rule. It select (irrevocable) the applicable rule that produces a database giving the largest increase in the value of the function. Our hill - climbing function

must be such that it attains its highest value for a database satisfying the termination condition.

Applying hill - climbing to the 8 - puzzle we might use, as a function of the state description, the negative of the number of tiles " Out of place", as compared to the goal state description. For example the value of this function for the initial state in the following figure is - 4, and the value for the goal state is 0. We can easily compute the value of this function for any state description.



From the initial state, we achieve maximum increase in the value of this function by moving the blank up, so, our production system selects the corresponding. In the above figure we show the sequence of states traversed by such a production in solving this puzzle. The value of our hill - climbing function for each state description is circled. The above figure shows that one of the rule applications along the path did not increase the value of our function. If none of the applicable rules permits an increase in the value of our function, a rule is selected (arbitrarily) that does not diminish the value. If there are no such rules, the process halts.

Any applicable rule applied to the initial state description lowers the value of our hill - climbing function. In this case the initial state description is at a local (but not a global) maximum of the function.

Other types of hill - climbing frustrations also occur: The process may get stuck on " plateaus " and "ridges" Of course, these difficulties could be solved if we could devise a better behaved hill climbing function - one that had just one global maximum and no plateaus, for example. Easily computable functions for problems of interest in AI typically have some of the difficulties we have mentioned. Thus, the use of hill climbing methods to guide rule selection in irrevocable production systems is quite limited.

Even though the control strategy cannot always select the best rule to apply at any stage there are times where an irrevocable regime is appropriate. For example, if the application of what might turn out to be an inappropriate rule does not foreclose a subsequent application of an appropriate rule, nothing (other than making superfluous rule applications) is risked by applying rules irrevocable.

USE OF EVALUATION FUNCTIONS (HEURISTIC SEARCH)

Heuristic information can be used to order the nodes GRAPHSEARCH so that search expands along those sectors of the frontier though to be most promising. In order to apply such an ordering procedure, we need a method for computing the "promise" of a node One important method uses a real - valued function over the nodes called an evaluation function. Evaluation functions have been based on a variety of ideas: Attempts have been made to define the probability that a node is on the best path; distance or difference metrics between an arbitrary node and the goal set have been suggested; or in board games or puzzles, a configuration is often scored points on the basis of those features that it possesses that are thought to be related to its promise as a step towards the goal

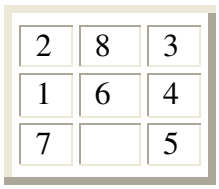
Suppose we denote the evaluation function by the symbol f then f (n) gives the value of the function at node n. For the moment we let f be any arbitrary function; later, we propose that it be an estimate of the cost of a minimal cost path from the start node to a goal node constrained to go through node n.

We use the function f to order the nodes of GRAPHSEARCH. By convention, the nodes are ordered in increasing order of their f values. Ties among f values are ordered arbitrarily, but always in favor of goal nodes. Supposedly, a node having a low evaluation is more likely to be on an optimal path.

The way in which GRAPHSEARCH uses an evaluation function to order nodes can be illustrate by considering again our 8 - puzzle example. WE use the simple evaluation function:

$f(n) = d(n) + W(n)$

Where d (n) is the depth of node n in the search tree and w (n) counts the number of misplaced tiles in that database associated with node n. Thus the start node configuration



has an *f* value equal to 0 +4 = 4

The results of applying GRAPGSEARCH to the 8 - puzzle using this evaluation function are summarized in the following figure. The value of *f* for each node is circled; the non-circled numbers show the order in which nodes are expanded. We see that the same solution path is found here as was found by the other search methods, although the use of the evaluation function has resulted in substantially fewer nodes being expanded. (If we simply use the evaluation function f (n) = d (n), we get the breadth first search process.

3.3 Implementation

We have used the C-language for programming a simple implementation of 8-puzzle in Turbo-C^{++}.

3.3.1 Algorithm

Here we have used the algorithm for heuristic search, which is far better than the hill climbing searching procedure.

Procedure Heuristic Search: -

Begin

1. Identify possible starting states and measure the distance (f) of their closeness with the goal; Put them in a list L;

2. **While** L is not empty do
Begin

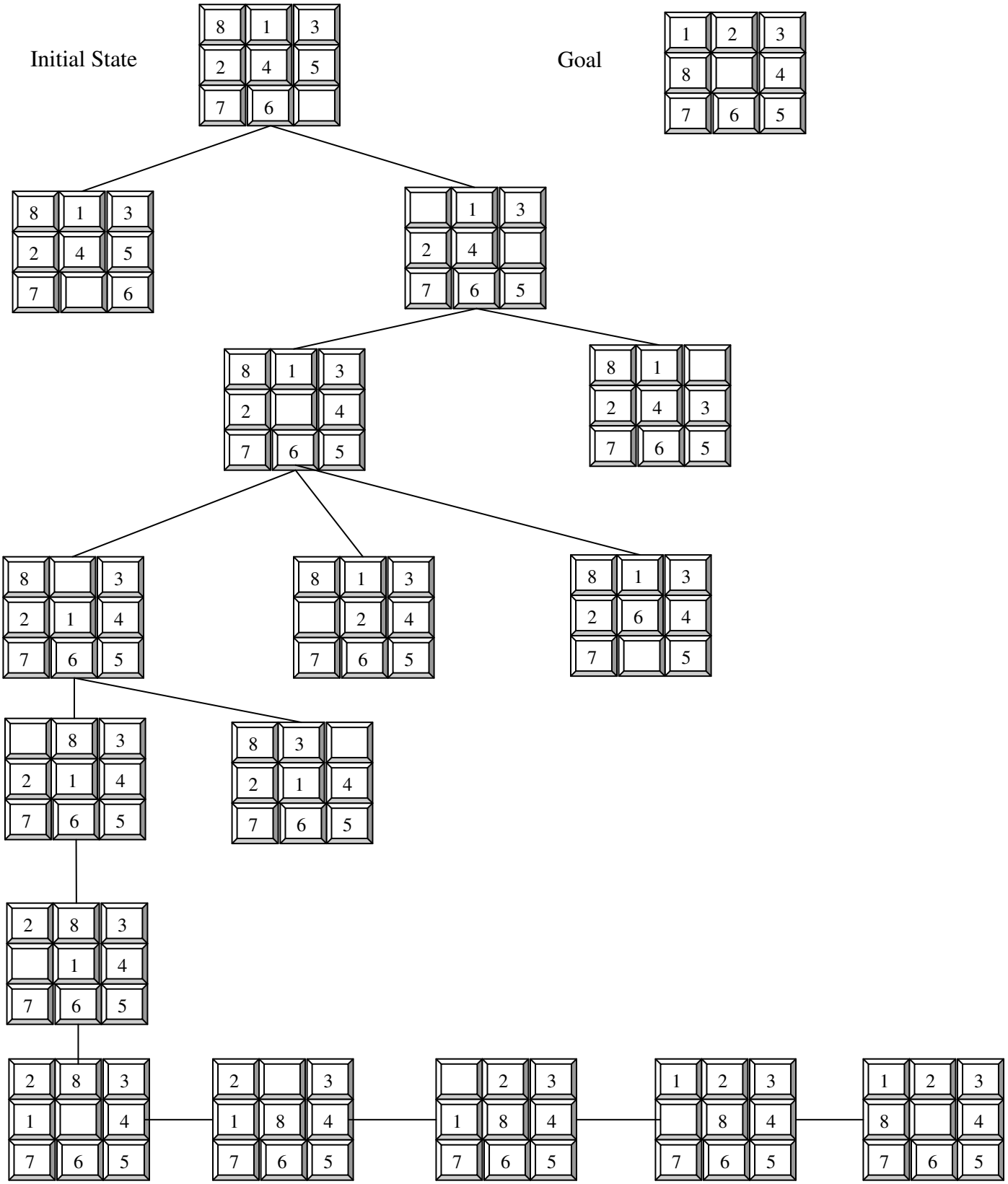
a) Identify the node n from that has the minimum f; If there exist more than one node with minimum f, select any one of them (say, n) arbitrary;

b) **If** n is the goal
Then return n along with the path from the starting node and exist;
Else remove n from L and add all the children of n to the list L with their labeled path from the starting node;

End While;

End

The choice of evaluation function critically determines search results The use of an evaluation function that fails to recognize the true promise of some nodes can result in non-minimal cost paths; Whereas, the use of an evaluation function that overestimates the promise of all nodes (such as the evaluation function yielding breadth - first search) results in expansion of too many nodes.


Initial State
Goal

4.1 Introduction - *Water Measurement*

Suppose it is given to measure 4 liters of water using a 3-liter and a 5-liter bucket then, any human beings can say how to do it. But, for a machine it might not seem so easy to do it in the minimum number of steps. How to decide what to do next in a given state

4.2 Implementation

Here our main objective is to measure a given amount of water using two buckets of fixed given capacity. We have used the C-language for programming a simple implementation of **'water measuring problem'** in Turbo-C^{++}.

If we are supplied with 5 liter and 3 liter buckets and asked to measure 4 liters using these buckets, the, we can following the following operations at an instant (state):

1. **Fill bucket one**
2. **Fill bucket two**
3. **Throw the water of bucket one**
4. **Throw the water of bucket two**
5. **Transfer the water from bucket one to bucket two**
6. **Transfer the water from bucket two to bucket one**

We discard the state obtained by the above operations if that state has been already fetched. In the following table we have used the following notation to describe the operations implemented during solving the problem:

States
Amount of water in bucket one Amount of water in bucket two
State on which operation is done **Operation implemented**

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
00	**5**0	**0**3	**5**3	**2**3	**3**0	**2**0	**3**3	**0**2	**5**1	**5**2	**0**1	**4**3	**1**0	**4**0
-	0**1**	0**2**	1**2**	1**5**	2**6**	4**4**	5**2**	6**5**	7**6**	8**1**	9**3**	10**5**	11**6**	12**4**

4.2.1 Algorithm

Here we have used the technique of **'Breadth first Search'.** The breadth first search algorithm is formally presented below:

Procedure Breadth-first-search

Begin

i) Place the starting node in a queue;

ii) **Repeat**

Delete queue to get the front element;
If the front element of the queue = goal,
Return success and stop;
Else do
Begin

Insert the children of the front element,
If exist, in any order at the rear end of the queue;

End

Until the queue is empty;

End

5.1 Introduction - *Robot Movement*

The word '**planning**' informally refers to the generation of the sequence of actions to solve a complex problem. For instance, consider the problem of placing the furniture in our new-built house, so that

i) we can fully utilize the available free space for common use
ii) the room looks beautiful

An analysis of the problem reveals that there exist many possible alternative solutions to the problem. But finding even a single solution to the problem is not so easy. Naturally, the question arises: why? Well, to understand this, we explore the problem a little more.

Suppose, we started planning about the placements of the following furniture in our drawing room:

a) one computer table
b) one TV trolley
c) one book case
d) one corner table
e) two sofa sets and
f) one divan

We also assume that we know the dimensions of our room and the furniture. We obviously will not place the furniture haphazardly in the room as it will look unimpressive and it will not provide us with much space for utilization. But where is the real difficulty in such planning?

To answer this, let us try to place the corner table first. Since only two corners B and D are free, we have to place it at either of the two locations. So if we do not place it first, and fill both the corners with other furniture, we will have to revise your plan. Fixing the position of your corner table at the beginning does not solve the entire problem. For example if we fix the position of the corner table at B then the place left along AB allows us to place the bookcase or one sofa set or the TV trolley or the computer table. But as the switchboard (SB) is on the wall AC, we will prefer to keep our computer table and TV trolley in front of it. Further, we like to keep the sofa set opposite to the TV. So they occupy the position as shown in the figure. The bookcase thus is the only choice that could be placed along the wall CD. The following steps thus represent the scheduling of our actions:

1. Place the corner table at B.
2. Place the TV trolley and the computer table along the wall AC.
3. Place the two sofa sets along the wall BD.
4. Place the bookcase along the wall AB.
5. Place the divan along the wall CD.

What do we learn from the above plan? The first and foremost, with which all of us should agree, is minimizing the scope of options. This helps in reducing the possible alternatives at the subsequent steps of solving the problem. In this example, we realize it by placing the TV set and the computer close to the switchboard. Another important point to note is the 'additional constraints imposed to subsequent steps by the action in the current

step'. For example, when we fix the position of the TV set, it acts as a constraint to the placement of the sofa sets. There are, however, instances, when the new constraints generated may require revising the previous schedule of actions.

5.2 Planning with If-Add-Delete Operator

We consider the problem of block world, where a number of blocks are to be stacked to a desired order from a given initial order. The initial and the goal state of the problem is given similar to the following figure. To solve this problem, we have to define a few operators using the if-add-delete structures.

The database corresponding to the initial and the goal state can be represented as follows:

The initial state:

On (A, B)
On (B, Table)
On (C, Table)
Clear (A)
Clear (C)

The goal state:

On (B, A)
On (C, B)
On (A, Table)
Clear (C)

Where On (X, Y) means the object X is on object Y and Clear (X) means there is nothing on top of object X. The operators in the present context are given by the following if-add-delete rules:

We can try to solve the above problem by following sequencing of operators. Rule 2 is applied to the initial problem state with an instantiation of X=A and Y=B to generate state S1 as in the previous figure. Then we apply Rule 3 with an instantiation of X=b and Z=A to generate state S2. Next Rule 3 is applied once again to state S2 with an instantiation of X=C and Z=B to yield the goal state. Generating the goat from the given initial state by application of a sequence of operators causes expansion of many intermediate states. So using the method of **'backward reasoning'** may prove to be more efficient.

5.3 Implementation

Here we have implemented the problem of block world, where a number of blocks are to be stacked to a desired order from a given initial order. Here we have considered the number of blocks to be 4.

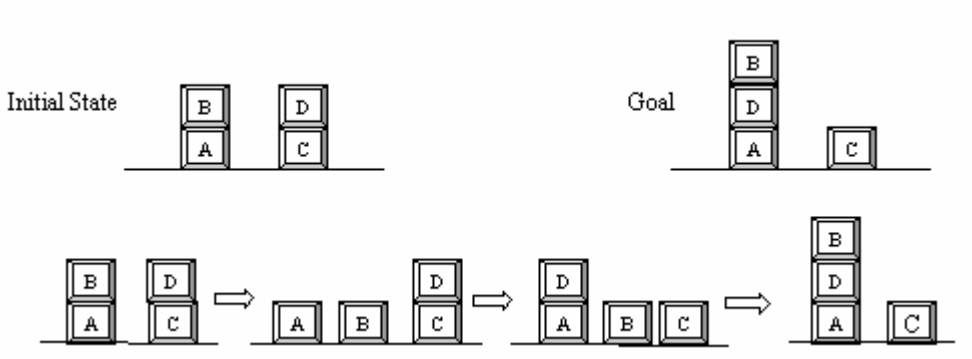
Box are named as 0 1 2 3 (4 nos. of box)
Box having nothing at the top is represented by -1

Example: - If situation is 1 above 0
2 above 1
3 above 2
Input given is: - 1 2 3 -1

i.e.

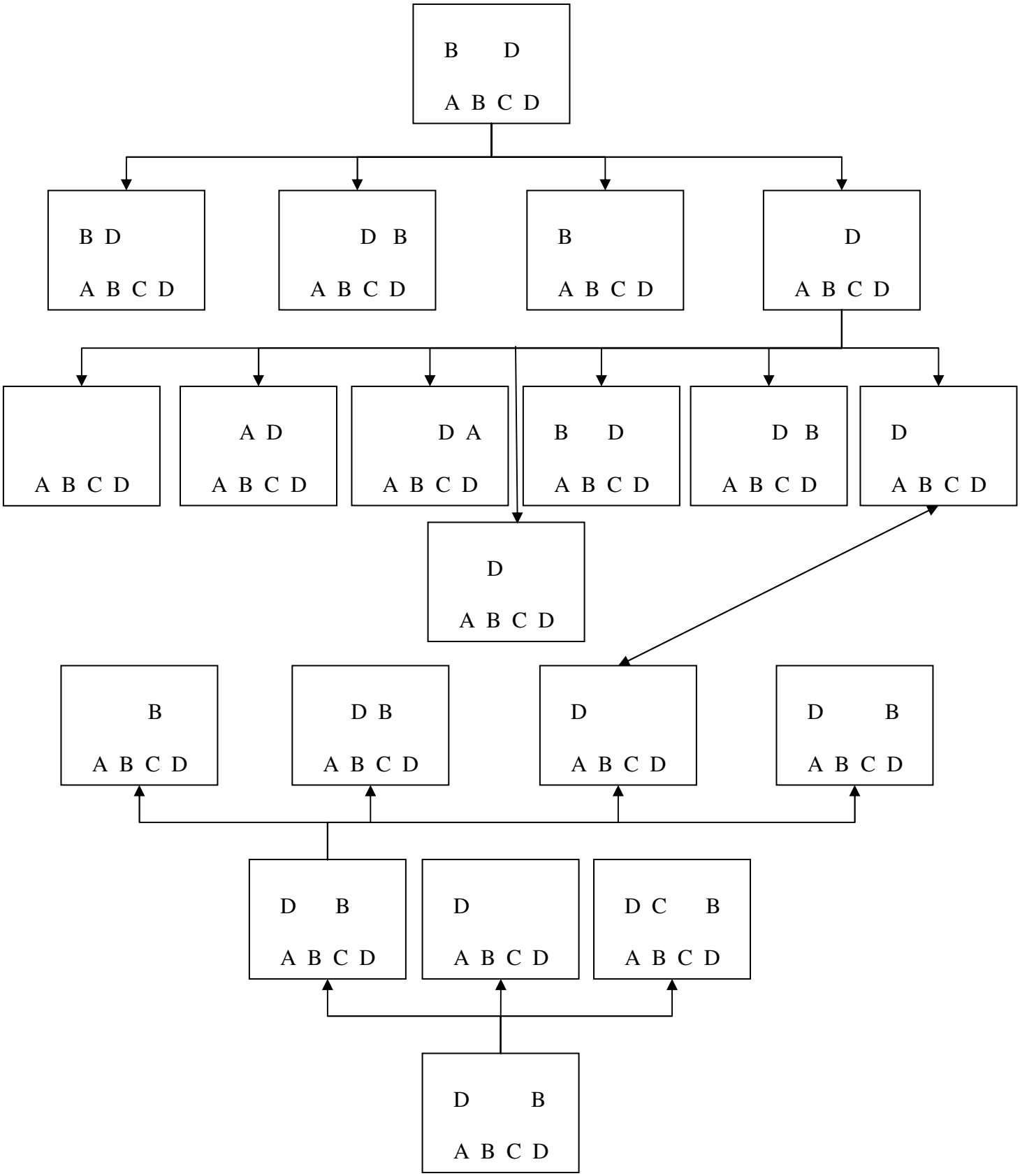
1 2 3 -1
0 1 2 3

Here we have used the If-Add-Delete Operator and have performed **'breadth first search'**. The function check () checks if the desired order have been reached or not, not exist () checks if the state have been achieved before and print () prints the output.



5.3.1 Algorithm

In the method of **'breadth first search'** we visit all the successors of a visited node before visiting any successor of those successors. This is in contradistinction to depth-first traversal, which visits the successors of a visited node before visiting any of its "brothers". Whereas depth-first traversal tends to create very long, narrow tree, breadth-first traversal tends to create very wide, short tree. Here we have used the both way traversal technique i.e. we start breadth first search from the desired order and also from the given initial order and stop when any one of the derived state in any one traversal get matched with any one of the previously derived state through the other traversal. We have used the C-language for programming a simple implementation in Turbo-C^{++}.


B D
A B C D
B D
A B C D
D B
A B C D
B
A B C D
D
A B C D
A B C D
A D
A B C D
D A
A B C D
B D
A B C D
D B
A B C D
D
A B C D
D
A B C D
B
A B C D
D B
A B C D
D
A B C D
D B
A B C D
D B
A B C D
D
A B C D
D C B
A B C D
D B
A B C D

6.1 Introduction - *Pattern Recognition*

Human beings can form perception about their 3-D world through many years of their experience. Building perception for machines to recognize not only their 3-D world but also any simple image is difficult. Here our main aim is to study and implement a simple system capable of identifying simple patterns.

6.2 Implementation

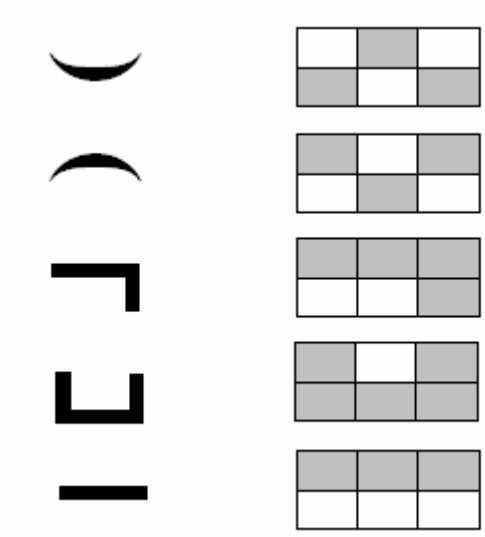
Here our main objective is to create a compact database of various patterns and thereafter use the same database to recognize a given pattern under noise.

6.2.1 Mathematical Formulation

An example pattern is inputted in the form of a grid structure. A pattern is represented in the form of a '**bipolar row vectors'** (V). The bipolar row vectors are generated from pixel grid with an assignment of +1 to the black pixels and –1 to white pixel. Then multiplying the bipolar vector with its transpose creates a matrix (M), which is, then added to the original matrix (database) (M_s).

For example: -

Example Pattern **Pixel Grid**



Bipolar vectors: -

$$\underset{\frown}{V} = (-1\ +1\ -1\ +1\ -1\ +1)$$

$$\underset{\frown}{V} = (+1\ -1\ +1\ -1\ +1\ -1)$$

$$V_{\neg} = (+1\ +1\ +1\ -1\ -1\ +1)$$

$$\mathbf{V_u = (+1\ -1\ +1\ +1\ +1\ +1)}$$
$$V\_ = (+1\ +1\ +1\ -1\ -1\ -1)$$

$$M_u = \begin{bmatrix} +1 \\ -1 \\ +1 \\ +1 \\ +1 \\ +1 \end{bmatrix} [\ +1\ \ -1\ \ +1\ \ +1\ \ +1\ \ +1\ ] = \begin{bmatrix} +1 & -1 & +1 & +1 & +1 & +1 \\ -1 & +1 & -1 & -1 & -1 & -1 \\ +1 & -1 & +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & +1 & +1 & +1 \end{bmatrix}$$

$$\begin{bmatrix} +4 & -2 & +4 & -2 & +2 & -2 \\ -2 & +4 & -2 & +0 & -4 & +0 \\ +4 & -2 & +4 & -2 & +2 & -2 \\ -2 & +0 & -2 & +4 & +0 & +4 \\ +2 & -4 & +2 & +0 & +4 & +0 \\ -2 & +0 & -2 & +4 & +0 & +4 \end{bmatrix} = M_s$$

A pattern is recognized by multiplying its bipolar vector by M_s. Let the bipolar vector of the pattern to be recognized be (+1 +1 +1 –1 –1 –1)

Hence the outputted pattern is:

$$\begin{bmatrix} [+1\ +1\ +1\ -1\ -1\ -1] \begin{bmatrix} +4 & -2 & +4 & -2 & +2 & -2 \\ -2 & +4 & -2 & +0 & -4 & +0 \\ +4 & -2 & +4 & -2 & +2 & -2 \\ -2 & +0 & -2 & +4 & +0 & +4 \\ +2 & -4 & +2 & +0 & +4 & +0 \\ -2 & +0 & -2 & +4 & +0 & +4 \end{bmatrix} \end{bmatrix}^T = [+8\ +4\ +8\ -12\ -4\ -12]$$

Hence the matched image's bipolar vector is [+1 +1 +1 –1 –1 –1]

6.2.2 Algorithm

The matrix-multiplication algorithm is formally presented below.

```
Procedure MULTIPLY
Begin
    // Let the pattern's bipolar vector be stored in input, database's matrix be fmat and
    the output pattern's bipolar vector be output
    For (I =0;i< max; i++)
    Begin
        For (j=0; j< max; j++)
        Begin
            output [i]=output [i]+input[j]*fmat [j][i];
        End For;
```

```
        End For;
End
```

7.1 Applications of Artificial Intelligence

Almost every branch of science and engineering currently shares the tools and techniques available in the domain of Artificial Intelligence. A few typical applications where Artificial Intelligence plays a significant and decisive role in engineering automation are:

- ***Expert Systems***
- ***Image Understanding and Computer Vision***
- ***Navigation Planning for Mobile Robots***
- ***Speech and Natural Language Understanding***
- ***Scheduling***
- ***Intelligent Control***

REFERENCES

1. Artificial Intelligence and Soft Computing – Behavioral and Cognitive Modeling of the Human Brain - Amit Konar, CRC, Boca Raton London New York Washington, D.C.

2. Principles of Artificial Intelligence – Nils J.Nilsson, Springer-Verlag, Berlin.

3. Artificial Intelligence – Rich, E. and Knight, McGraw-Hill, New Yolk

4. Artificial Intelligence – Planning in a hierarchy of abstraction spaces – Sacerdoti, E. D.

5. Artificial Intelligence – Winston, P. H., Addison-Wesley, Reading, MA.

RESULTS

TIC TAC TOE

MY MOVE

	X	

COMPUTER'S MOVE

O		
	X	

O		
	X	
X		

O		**O**
	X	
X		

O	**X**	**O**
	X	
X		

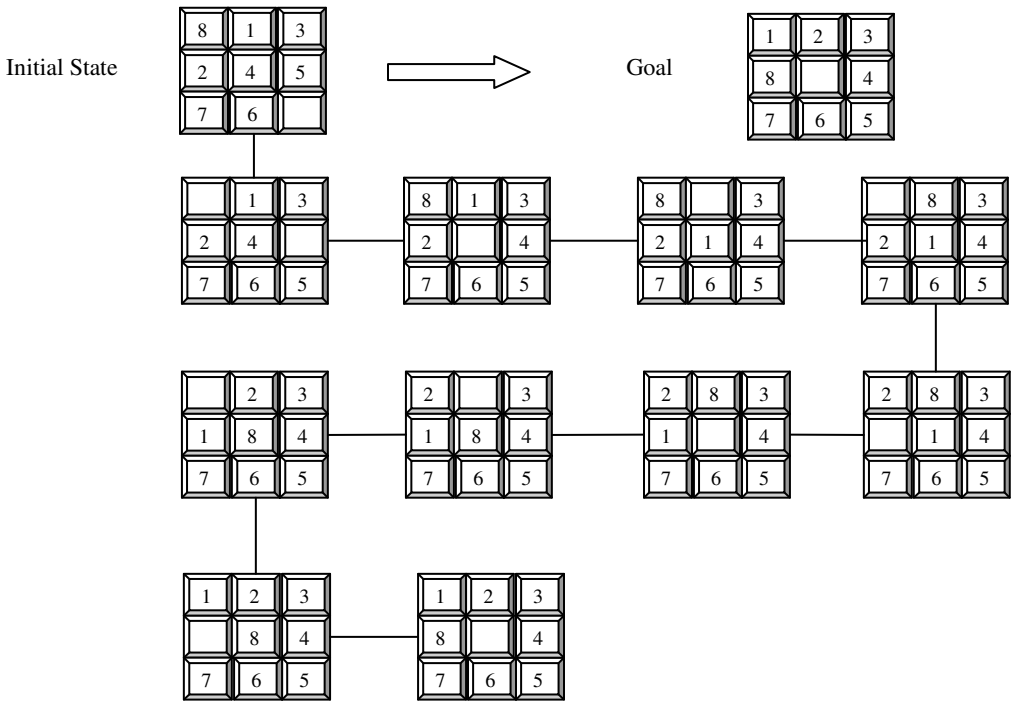
O	**X**	**O**
	X	
X	**O**	

O	**X**	**O**
X	**X**	
X	**O**	

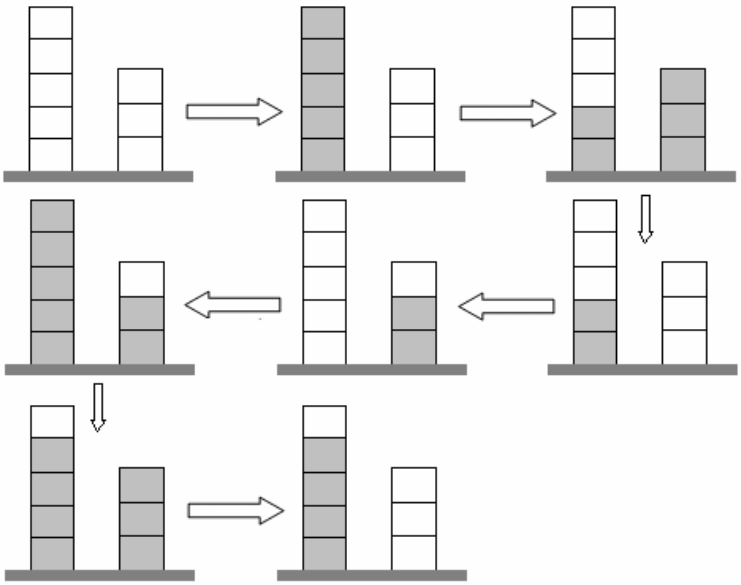
O	**X**	**O**
X	**X**	**O**
X	**O**	

Game is draw!
Do you want to continue (y/n)? :

8– PUZZLE



WATER MEASUREMENT



ROBOT MOVEMENT

ROBOT ARRANGING BOX

Box are named as 0 1 2 3 (4 nos. of buckets)
Box having nothing at the top is represented by -1

Ex :- If situation is 1 above 0 | 1 2 3 -1 |
2 above 1 i.e. | |
3 above 2 | 0 1 2 3 |
Input must be : 1 2 3 -1

Give the starting situation : 1 -1 3 -1

Give the final situation : 3 -1 -1 1

1 -1 3 -1
0 1 2 3

-1 -1 3 -1
0 1 2 3 Put 1 down

3 -1 -1 -1
0 1 2 3 Put 3 over 0

==========

3 -1 -1 -1
0 1 2 3 Put 1 down

3 -1 -1 1
0 1 2 3
Press any key to continue

PATTERN RECOGNITION

Images in the Database are :-