PATHOLOGY OF TREES, TREE SEARCHES & THEIR COMPUTATIONAL COMPLEXITIES

[1] PARAG BHALCHANDRA, [2] DR.S.D.KHAMITKAR, [3] S.N.LOKHANDE

School of Computational Sciences, S.R.T.M.University, Nanded, MS, India, 431606.

Srtmun.parag@gmail.com,s.khamitkar@gmail.com,sana_lokhande@rediffmail.com

ABSTRACT *: Searching trees is a most interesting application of Artificial Intelligence. Over the period of time, many innovative methods have been evolved to better search trees with respect to computational complexities. In this paper, we have tried to sketch fundamental aspects, pathological point of view to trees as well as their searching processes. We are hopeful that our contribution will be helpful for the researches working on tree searches.*

Keywords : Trees , Search, Asymptotic Complexity, Brute Force, Heuristics

1. INTRODUCTION

Representation of values in any computational domain is fundamentally important. This representation is mainly carried out by using data structures. For the domain of geometric computation, a few predominate representations have emerged. These include discrete space representations like arrays of lattice points, topological representations like boundary-representations, parametric surfaces, simplified decompositions and trees [1]. In computational paradigms such as algorithm analysis and complexity theories, trees are widely used data structures that simulate hierarchical structures with a set of linked nodes [2]. Trees find occurrence in Discrete Mathematics, Population Studies, Graph theories, Network Design Theories, Compilation & Parsing approaches, Optimization Theories, etc [1,6,7,12]. Beside these, trees are widely used for partitioning as every node/ element in them work as a decision making point. Trees are powerful tool for organizing data objects based on keys. They are equally useful for organizing multiple data objects in terms of hierarchical relationships. A tree can be defined recursively as a collection of nodes starting at a root node, where each node is a data structure consisting of a value, together with a list of nodes, called children nodes, with the constraints that no node is duplicated. When we examine a non-leaf node, we see that the node has trees growing underneath it, and we say that the node has children sub trees. A tree has many levels depicting sub trees, we can think of a family tree where the children are grouped under their parents in the tree.

Tree structures have made an excellent alternative to arrays, especially when the data stored within them is keyed or has internal structure that allows one element to be related to, or saved within another[9,11].This highlights their usefulness for complexity reduction and optimization processes [10]. Trees can hold objects that are sorted by their keys. For example, the nodes are ordered so that all keys in a node's left subtree are less than the key of the object at the node, and all keys in a node's right subtree are greater than the key of the object at the node. Such arrangement drastically reduces searching time and related time complexity. Trees can also hold objects that are located by keys that are sequences [7]. Trees can represent a structured object, such as a house that must be explored by a robot or a human player in an adventure game, along a specific path from the source. Trees are also useful to represent phrase structure of sentences, which is crucial to language processing programs / parse trees.

A tree can be defined abstractly as an ordered tree, with a value assigned to each node. If in a tree, an ordering of the nodes at each level is prescribed then such tree is called as an ordered tree. If we delete the root and its edges connecting the node at level 1, we obtain a set of disjoint trees. A set of disjoint trees is called a forest. Any node of a tree is the root of some subtree. Therefore, sub trees are seen immediately below a node form a forest. The tree can be also defined in a recursive fashion. According to this definition, a tree contains one or more nodes such that one of the nodes is called a root while all the other nodes are partitioned into a finite number of trees called sub trees. It can be defined as a restricted graph. This restriction imposed upon the graph yields a general tree. Each general tree can be represented by an equivalent binary tree. A tree can be analyzed mathematically as a whole. If represented as a data structure, it is usually represented and worked with separately by node rather than as a list of nodes and adjacency list of edges between nodes. Indeed, given a list of nodes, and for each node a list of its children, one cannot tell if this structure is a tree or not without analyzing its global structure and checking that it is in fact topologically a tree . In terms of references, a tree is a special kind of directed graph, with a global constraint on its topology namely no loops as an undirected graph. Trees have built in recursive aspects .Recursively, a tree is defined as a node (the root), which itself consists of a value (of some data type, possibly empty), together with a list of nodes

(possibly empty). Mathematically, tree is viewed as a whole, a tree data structure is an ordered tree generally with values attached to each node [2,3,9]. Concretely, it is: A rooted with the "away from root" direction meaning:

1. A directed graph , whose underlying undirected graph is a tree (any two vertices are connected by exactly one simple path),
2. With a distinguished root (one vertex is designated as the root), which determines the direction on the edges (arrows point away from the root; given an edge, the node that the edge points from is called the parent and the node that the edge points to is called the child), together with an ordering on the child nodes of a given node, and a value (of some data type) at each node.
3. There is exactly one path connecting any two nodes in a tree
4. A tree with n nodes has n-1 edges.
5. A full binary tree with n internal nodes has n+ 1 external node.
6. The height of a complete binary tree with n internal nodes is about log_2n

Above discussion implies that, we can organize the data so that items of information are related by the branches. Thus a tree is a finite set of one or more nodes such that:

1. There is a specially designated node called the root.
2. The remaining nodes are partitioned into n > 0 disjoint sets $T_1,\ldots, T_n$, where each of these sets is a tree. We call $T_1,\ldots, T_n$ the sub trees of the root.
3. Often trees have a fixed , bounded branching factor particularly always having number of child nodes that determines whether tree is binary (two children), ternary (three children) or a m-ary tree[8] (possibly many non-empty child nodes)

The parent-child relationship can be extended naturally to ancestors and descendants. Informally, the ancestors of a node are found by following the unique path from the node to its parent, to its parent's parent, and so on. Strictly speaking, a node is also its own ancestor. The descendant relationship is the inverse of the ancestor relationship, just as the parent and child relationships are inverses of each other. The length of the path is k –1, one less than the number of nodes on the path. Note that a path may consist of a single node (if k = 1), in which case the length of the path is 0.

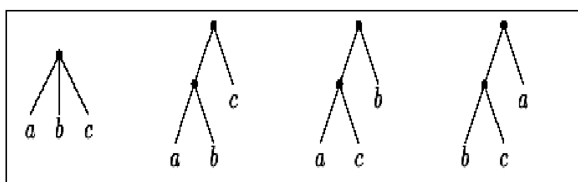

Figure 1 : General Trees with root node and other three nodes a,b,c

Above figure shows different forms of a tree with a root node and other three nodes a,b,c . From figure, we can say that a tree is associated with following terminology,

1. Root of the tree: The top node of the tree that is not a subtree to other node, and has two children of subtrees.
2. Node: It is stands for the item of information and the branches to other nodes.
3. The degree of a node: It is the number of subtrees of the node.
4. The degree of a tree: It is the maximum degree of the nodes in the tree
5. The parent node: a node that has subtrees is the parent of the roots of the subtrees
6. The child node: a node that is the roots of the subtrees are the children of the node
7. The Level of the tree: We define the level of a node by initially letting the root be at level one
8. The depth of a tree**:** It also called height of a tree. It is the maximum level of any node in the tree

2. DISCUSSIONS AND ANALYSIS OF TREE SEARCHING METHODS

Searching is frequently associated with trees. Such search is called Tree Search, since long has been a topic of interest in Artificial Intelligence [9,11,13]. There are two different strategies for searching a solution space. One of these strategies is data-driven. A data-driven strategy starts at the root and traverses the branches of the tree until it finds the solution. The other is a goal-driven strategy that starts at the solution and then tries to find a path back to the starting point. The goal-driven strategy is also commonly called backtracking. The basic tree search algorithms found in the literature are either data-driven strategies or goal driven strategies. Irrespective of the strategy the algorithm uses, it is witnessed that the traversing operation is carried out on trees to find the solution or goal. A basic requirement for tree searching is invoking a traversal process that visits every node to cross check required goal. Compared to linear data structures like linked lists or one dimensional arrays, which have a canonical method of traversal namely in linear order, tree structures can be traversed in many different ways. For a binary tree, starting at the root, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps are: performing an action on the current node, traversing to the left / right child node, and then traversing to the right/ left child node. Traversing a tree involves iterating / looping over all nodes in some manner[9]. Because from a given node there is more than one possible next node as it is not a linear data structure, then, assuming sequential computation, some nodes must be deferred – stored in some way for later visiting. This is often done via a stack (LIFO) or queue (FIFO). As a tree is a self-referential and recursively defined data structure,

traversal can naturally be described by recursion where the deferred nodes are stored implicitly .While traversing, we need no memory other than the current and the previous node/state. Such traversal algorithms are very simple:

1. If the previous node is this node's parent node, descend to the left child node.
2. If the previous node is this node's left child node, descend to the right child node.
3. If the previous node is this node's right child node, ascend to the parent node.

Traversal is complete when an attempt to ascend to the parent node fails because there is no parent. Literature shows that the above three traversals are named as Pre order, Post order and In order traversals[9]. The name given to above particular style of traversal comes from the order in which nodes are visited. Technically, these three belongs to one of the simple way to intervene the tree space. Thus we need to study traversal techniques if we want to search something from trees. Further searching in a tree is far quick than searching in a linear structure. This discussion highlights that the search perform better when data is structured as a tree. Trees, and particularly binary trees, appear frequently in the classification literature [11]. Many applications, such those found in combinatorial optimization, graph searching, Network structure Analysis, can be addressed by searching through a large tree of possible solutions. Since search problems reflects fundamental task that must be tackled quite frequently, researchers have attempted to develop algorithms efficient in terms of Time or Space Complexities [8,9]. For example, in Artificial Intelligence there has been extensive research into search algorithms for problems such as propositional satisfability and constraint satisfaction problems. Tree search methods are categorized as Informed ,Uninformed, Exhaustive, Brute force, Uniform, Complete ,Optimal , Local, Backtracking depending upon the style of searching , amount of time spent in searching, does or does not take into account the specific nature of the problem, guarantee to find the goal, etc[2,4] .

The most common categories are Brute Force Search methods and Heuristic Search methods [2,11]. These methods are associated with specific data structures and follow a similar searching pattern. The search begins by expanding the initial node i.e., by generating successors. At each later step, one of the previously generated nodes is expanded and the process continues until a goal node is found**.** Both these categories differ in the way "How the generated node is chosen for expansion?" The Heuristic Search Algorithms take advantage of the property of maintaining current minimum and eliminating from the search process, the sub trees whose nodes exceed in cost and value of current minimum. They need additional domain specific knowledge to operate. A*. AO*, Best-First Search, etc are the few examples of Heuristic Search methods. On other hand, Brute Force Search Algorithms are useful when no additional information is available. These are blind in the sense that they go through all possible answers to the problem and chose the best one. Hence they are weak and less efficient. Depth First Search (DFS) and Breadth First Search (BFS) are the two most widely used Brute Force Search methods [2]. No doubt, there exits many ways of searching, technically, there are only two ways to penetrate into trees does one go down first (depth-first: first child, then grandchild before second child) or across first (breadth-first: first child, then second child before grandchildren). Depth-first traversal is further classified by position of the root element with regard to the left and right nodes. Imagine that the left and right nodes are constant in space, then the root node could be placed to the left of the left node (pre-order), between the left and right node (in-order), or to the right of the right node (post-order). There is no equivalent variation in breadth-first traversal – given an ordering of children; "breadth-first" is unambiguous. While traversal is usually done for trees with a finite number of nodes and hence finite depth and finite branching factor, it can also be done for infinite trees. This is of particular interest in functional programming, as infinite data structures can often be easily defined and worked with, though they are not strictly evaluated, as this would take infinite time. Some finite trees are too large to represent explicitly, such as the game tree for chess and so it is useful to analyze them as if they were infinite. For infinite trees, simple algorithms often fail. For example, given a binary tree of infinite depth, a depth-first traversal will go down one side (by convention the left side) of the tree, never visiting the rest, and indeed if in-order or post-order will never visit *any* nodes, as it has not reached a leaf (and in fact never will). By contrast, a breadth-first (level-order) traversal will traverse a binary tree of infinite depth without problem, and indeed will traverse any tree with bounded branching factor [5]. On the other hand, given a tree of depth 2, where the root node has infinitely many children, and each of these children has two children, a depth-first traversal will visit all nodes, as once it exhausts the grandchildren (children of children of one node), it will move on to the next (assuming it is not post-order, in which case it never reaches the root). By contrast, a breadth-first traversal will never reach the grandchildren, as it seeks to exhaust the children first. A more sophisticated analysis of running time can be given via infinite ordinal numbers. Thus, simple depth-first or breadth-first searches do not traverse every infinite tree, and are not efficient on very large trees. However, hybrid methods can traverse any (countable) infinite tree, essentially via a diagonal argument ("diagonal" – a combination of vertical and

horizontal – corresponds to a combination of depth and breadth).

The Brute force methods are simplest one and perform an exhaustive examination of all possible sequences of moves until goal states are reached [11]. The search through the state space systematically checks if the current state is a goal state. If a non goal state is discovered then the current state is expanded by applying a successor function, generating a new set of states. The choice of which state to expand is determined by a search strategy. In a great deal of occasions, an artificial intelligence application does not possess an adequate level of knowledge enabling the choice of the most promising state. Strategies that can only distinguish between goal states and non-goal states, without being able to determine if one state is more promising than another, are referred to as uninformed search strategies. All brute force algorithms are the examples of uninformed search strategies.

Uninformed strategies are only successful for small problem instances. Typically, most problems search space is characterized by an exponential growth due to the mammoth dimensions of the search space it becomes impractical, both time- and space-wise, to perform an exhaustive examination. Alternatively, it is possible to employ additional insights that arise beyond the definition of the problem. The use of this information, thus the term informed search strategies, allows for solutions to be found more efficiently. Typically, all heuristics algorithms are informed search strategies and often employ an evaluation function f(n) which considers a cost function g(n) alongside a heuristic function h(n). Function g(n) can be interpreted as representing the cost to reach node n whilst h(n) represents an estimate on the cost to reach a leaf node from node n. Traditionally, the node with the lowest evaluation value is selected for expansion[11,13,14].

Above discussion has put us in to a concise understanding of basic aesthetics of tree searches. It is observed that a tree search is basically a problem-solving system, which has a knowledge base containing information about the current state of the problem in hand, a set of operators for transforming the knowledge base in some way, a control strategy or means of deciding which operator to use at any particular point during the solution of the problem, and a way of deciding when the problem is solved or unsolvable [15,16, 18]. To understand this better, let us consider a program to solve the 8-puzzle. This is a puzzle where there are 8 numbered tiles that can slide either sideways or vertically in a 3 by 3 square framework containing 9 spaces. Only one tile can be moved at a time into the single unoccupied space in the framework. In most versions of the puzzle the tiles are set in some disorganized order at the start and the problem is to rearrange them into order by sliding one tile at a time (and not taking them out of the framework). In a program to solve the 8-puzzle the knowledge base would contain a representation of the tiles in the frame and the operators would be the moves available at the particular point in play. Here a solution would be a completed sequence of tile moves that brought the puzzle to the required state. The control strategy would make sure that in general it was the tiles which were out of place that were moved (though of course it might be necessary to temporarily move tiles that were already in the right place), because it would be these misplaced tiles that would necessarily have to be moved as part of any solution. The amount of search involved in a problem can be reduced if there is a method of estimating how effective an operator will be in moving the initial problem state towards a solution, e.g. a method for choosing "promising" moves in chess or choosing which tile to slide in the 8-puzzle. A great deal of attention has been given to methods of making such estimates and to the repercussions of such estimates on control strategy [18]. Occasionally the amount of search can be reduced very dramatically by representing the problem in a new way, that is by looking at it from a different viewpoint .Planning programs, for example, can usually reduce the amount of search by first considering only the most important factors of a problem before going on to consider the details once the main issues have been sorted out. In general, the representation of the problem by the program is determined by the programmer and even these "hierarchical" planners have their notion of importance built in. Getting a program to decide for itself, what counts as important or determine the best way to view a problem is extremely difficult. Getting a program to automatically re-orientate its view of a problem in the way suggested is, alas, beyond the state of the art [18]. Many search problems fall into one or other of two classes depending on the action of the operators involved. In some problems each operator takes the problem state or situation and transforms it into a single new problem state. For example, in the 8-puzzle each move transforms the arrangement of the tiles to give a new arrangement. Here the problem is to find that sequence of operators (moves) which in total transforms the initial state of the frame of tiles to a concluding state. This is called State Space Search. By contrast Problem Reduction involves the use of operators which break down a complex problem possibly into several simpler, possibly independent, sub-problems each of which must be solved separately. In both State Space approaches and Problem Reduction there may be a variety of operators which in principle can apply at any particular point. Thus in the 8-puzzle (a State Space example) there is always a choice of at least two tiles that could be moved at each stage; in undertaking a mathematical integration (a typical Problem Reduction example) there typically will be a number of methods available, each of which will sub-divide

the larger integration problem into smaller ones in a different way.
The totality of all possible nodes in a problem is known as its search space. It is important to distinguish the complete search space for a problem (i.e. all possible nodes and arcs) from that part of the space that any particular system explores in its attempt to find a solution (i.e. those nodes and arcs actually considered). Usually only a portion of this space is or can be explored. Some control strategies for exploring the search space work forward from an initial state towards a solution. Such methods are also sometimes called data-directed. An alternative strategy is to work backward from a goal or final state towards either soluble sub-problems or the initial state, respectively. Such a control strategy is sometimes called goal directed. Problem reduction systems often work backward in this way. Sometimes a mixture of both forward and backward strategies is employed, in the hope that working forward and backward will successfully meet in the middle. If the search space is small, systematic (but blind) methods can be used to explore the whole search space [5,9]. These include depth-first search where, for each new node encountered, one of the arcs from it is explored. Only if a dead end is reached does the system return to the most recent choice-point and try a different arc. This method is easy to implement but can be dangerous in that the system may spend a long (or infinite!) time fruitlessly exploring a hopeless path. A variation on this method, called bounded depth first search, sets a limit on the depth to which exploration is allowed. Breadth-first search is another systematic but blind method. Here all the arcs from a node are explored before moving on to explore any arcs from the new nodes encountered. The advantage of this method is that it is guaranteed to find a solution consisting of the shortest path (if one exists) but can be computationally expensive on memory especially if each node is bushy, i.e. has many arcs coming out of it. These systematic but blind methods can be applied to either state space representations or problem reduction. Problems differ in the form that their goal states take. Sometimes the goal state is known explicitly, such as in the 8- puzzle when all the tiles are to be in correct numerical order. In these cases it may be possible to make a comparison between the current state and the goal state as part of the process of monitoring progress [9,18].
In other problems, the goal state is not known at the start, but there is a procedure which will decide whether or not a given state conforms to the criterion of being a goal state. An example here might be successfully solving an equation. Problems also differ in whether the value of the goal state is the real goal of the problem or whether it is the path to that goal which is more important. For instance, route-finding emphasizes the path where equation-solving emphasizes the goal itself. Most systems are limited by either time or space constraints to explore only a portion of the search space, choosing only certain of the alternatives available. Such systems depend on knowledge of the problem domain to decide what might be promising lines of development. They will have some measure of the relative merits either of the different nodes or of the available operators to guide them. Search which is so guided is called heuristic search. The methods used in such search are called heuristics. These terms often carry the connotation of inexactness and fallibility and are contrasted with algorithms which are bound to work. Again both state-space search and problem-reduction search can be conducted using heuristics [9].
Above discussion, up to some extent, may give the impression that the search tree of nodes and arcs is a pre-existing artifact which the program explores. Occasionally this is true, but in most cases the problem-solving program explores the space by "growing" a tree as it goes. Thus it is often reasonable to picture a problem-solving program as if it were a driver, without a map but with a notepad, driving around searching for a particular village in a maze of country roads. As the search proceeds the traveler keeps a record of which roads and villages have already been encountered but can have no foreknowledge of what lies ahead. By choosing one node in the search space as a starting point (whether working forward or backward) and by applying the available operators, the program grows a tree rather than a graph [17,18]. That is, each node explored points back to one node only, the one from which it was produced. The production of a tree rather than a graph makes it straightforward to extract a solution path, traced from the node representing the solution back to the starting node. Depending on the particular problem and on the way the program is implemented this tree may contain the same node at different points or may not. Re-encountering a node on a particular solution path during search indicates that the program has found a loop in the search space. For example, if this happens while solving the 8-puzzle, it means that the program has found a set of moves which takes it to exactly the same position as has been met before.

3. UNDERSTANDING COMPLEXITY BEHAVIOR OF TREE SEARCHES

It is evident from above discussion that, in case of any tree search deployed over a problem; we need to compute a solution. As there can be many ways to compute a solution for a given problem, we always concern about a good solution. The goodness always has both quantative as well as qualitative aspects. A good solution is economical in the use of computing and human resources. Here resources means execution time- CPU time and memory space- cache/main/file memory used [2,3]. An important factor in deciding the quantitative aspect of the goodness of a tree search program is the efficiency of its underlying algorithm. We are concerned with the relationship between the execution time and the

problem size of the input to an algorithm that is with the increase in execution time with a specified increase in the size of the input to the problem. Previous studies by Algorithimicians as well as computational scientists have categorized algorithms as tractable (reasonable) and non tractable (unreasonable).

If the complexity of an algorithm is expressed as O(pn) , where p(n) is some polynomial of n, then the algorithm is said to be polynomial time algorithm. It is generally accepted that the polynomial time algorithms are tractable. Any algorithm with a time complexity that can not be bounded by such bounds is called as a non polynomial time algorithm or non deterministic polynomial (NP) algorithm. Usually, for an input size of n, if the complexity is proportional to n, n^2,n^3,n^5, log (n), n log(n) then such algorithms are polynomial time algorithms. On the other hand, if the complexity is proportional to 2^n,3^n,n!, then its Non Polynomial algorithm[3,6,15].

In analyzing any given tree search algorithm, there are two measures of performance that are usually considered, the worst case behavior and the average case behavior. If for a given problem of size n , an algorithm corresponds to maximum complexity among all problems of size n, then its definitely the worst case behavior. It's usually determined by choosing a set of input conditions that forces the algorithm to make the least possible progress towards its final goal at each step. In many practical applications, it is important to have measures of the expected complexity of a given algorithm rather than the worst case behavior. The expected complexity gives a measure of the algorithm's working or behavior averaged over all possible problems of size n. However, while computing or solving a problem in a practical implementation, we have to choose between two or more algorithms. Generally we would opt for an algorithm that has the lower expected complexity. This is practically feasible if we set up a complex and sophisticated combinational analysis. In case of analysis of average case behavior, it is assumed that all possible points of termination are equally distributed (needs assumption of statistical distribution). Consider a list of n items. If a search algorithm finds goal at first position, then it is best case complexity behavior. If the algorithm needs to analyze all values in the list before termination, then its worst case behavior . If we do not have best case or worst case behavior, and it is likely that the probability that an item could be found at position 1 is 1/n, at position 2 is also 1/n and so on , then its average case complexity behavior. Here the average search cost is the sum of all possible search costs multiplied by their associated probability [3,15].

The analysis of any algorithm is done with respect to space and time. In space analysis, we are interested in analyzing how much storage space is used. While performing timing analysis, we thought of an estimated time rather than the exact time for execution . this is usually done by isolating a particular operation, called an active / dominated operation which is central to the algorithm and which is executed frequently. Once the active operation is isolated, the number of times it is executed is counted. As long as the active operation occurs at least as often as the others, the execution time will increase in some proportion to the number of times the active operation is executed. The asymptotic notations are useful in such situations where the complexity is expressed in terms of active operations. In our entire work, the complexity analysis is expressed in terms of asymptotic growth rates which are the measure of the time taken by an algorithm as the value of the input measure grows without bounds. In computing the asymptotic growth rate , we ignore multiplicative constant in the complexity function and focus on the rate it self [3,15] . For the purpose of comparing asymptotic growth rates the notations 'O'(for "order") that is "Big Oh", ' Θ' that is Big Theta notation and 'Ω ' Big Omega notations are used. Their definitions and properties are expressed in Appendix-1

While carrying out research , we felt that, there are situations where we have to choose between the CPU time the program uses and the primary / secondary memory it will occupy. The choice depends upon the implementation criteria. If storage space is available and otherwise unused, it is preferable to use algorithms that require more space and less time as compared to other algorithms solving the same problem. If space is not available, then time may have to be sacrificed. Further, it is observed that a non recursive algorithm will execute more efficiently in terms of space and time than a recursive one[3,4]. This is because the overhead involved in entering and exiting a block is avoided in the non recursive code .In a recursive algorithm, a number of local and temporary variables are to be tacked and unstacked on the stack which consumes both time and space. Beside space and time analysis of tree searching algorithms, we also stick to modularity, scalability, correctness, maintainability , simplicity and graceful degradation aspects of performance of algorithms. These factor, indeed were the criteria to judge whether one algorithm is better than second one or not? Some considerations to user friendliness, extensibility, concurrency behavior, distributedness, security , hardware- software compliantness were also given .

Complexity Picture of Selected Tree Search Methods				
Name	**Time Complexity**	**Space Complexity**	**Optimal?**	**Comment**
BFS	O (b^d)	O (b^d)	May be	Optimal only when the optimal path is the shortest
DFS	O (b^d)	O (d)	No	Blind Alley trapping
A*	---	---	Yes	Optimal
IDA*	---	---	Yes	Optimal
Hill Climbing	Width^Depth	Width^Depth	Yes	Optimal

Table 1 : **Complexity Picture of Selected Tree Search Methods**

We do both theoretical and empirical complexity analysis of tree searching algorithms. Theoretical analysis is helpful for understanding asymptotic behavior of an algorithm. The Empirical analysis is important in comparing two algorithms which may or may not have same order of complexity and then to decide when would one use one and not other. A theoretician, Baase [14,15] lists five aspects to consider in the process of analyzing an algorithm. These are also applicable to tree searching algorithms. These aspects are Correctness, Work done, Space used, Simplicity or clarity, Optimality. Another Algorithmician Sedgwick [9,14,15] devotes a chapter to the "Implementation of Algorithms". Here he makes the claim that "it is unfortunately all too often the case that mathematical analysis can shed very little light on how well a given algorithm can be expected to perform in a given situation". He stresses the importance of empirical analysis in this case. He also advocates the use of empirical analysis in comparing two algorithms to solve the same problem. Brunskill and Turner [9,15] give a list of some things that the execution time of a given program will depend on the CPU, the compiler, the programming language, the way the program is constructed, time for disk accesses and other IO ,whether the system is single or multitasking, etc .To make this real, this study has insisted on implementation of five selected algorithms and conducting rigorous analysis as pointed out in above analysis context.

This requires experimentally verifying the complexity behavior of selected tree searching algorithms and then using their experimental data to compare the algorithms. In order to do proper empirical analysis to verify and expand on the theoretical analysis of an algorithm we need to understand the theoretical analysis , decide on what should be measured, decide on appropriate hardware ,decide on an appropriate implementation language, decide on appropriate data structures, implement the algorithms ,implement some form of timing device, create the input data sets necessary to produce the measurements we need, measure the performance of the algorithm on the different input data sets created to meet our aim, interpret the results ,relate the results to the theoretical analysis.

Few of these tasks were trivial. To deal with them adequately, knowledge and understanding of a number of theoretical concepts/areas are required – asymptotic notation; probability theory regarding tree searching, tree architecture, tree specific data structures and tree representation; and experimental statistics, etc[17] .By carrying out this research , our plan was to

1) better understand the trees
2) better understand tree searching algorithmic ideas in practice
3) check for accuracy or correctness in standardized case
4) access the quality of tree search algorithms , and
5) comparing the actual performance of competing algorithms for some tractable , constraint satisfaction problems, etc

4. CONCLUSION

As concluding remarks, we have observed that, for any problem that can be represented as a problem space, search techniques can be used to solve it. The price of this generality is exponential complexity, with the result that many problems of practical interest are solvable in principal with tree search. The limitations of computational capacity prevent them from being solved with expected complexities in practice. The increasing diversity in computing platforms motivates consideration of multi-processor environment. In such circumstances, we can also view the traditional tree search algorithms through a new perspective.

5. REFERENCES

1. Constructing Good Partitioning Trees , Bruce Naylor, AT&T Bell Laboratories, Murray Hill, NJ, Lecture notes,1990
2. Artificial Intelligence and its Teaching, Dr.Gergely Kovasznai & Dr.Gabor Kusper , Institute of mathematics and Informatics ,Lecture notes published by National Development Agency,Hungery,1990

3. Design and Analysis of Algorithms, Aarag Himanshu Dave et al, Pearson Education ,First edition,2008
4. D. Aldous. The continuum random tree I, I, IIII: An overview. In Stochastic Analysis ,. Cambridge Univ. Press, Cambridge 1991.
5. K.B. Athreya and P.E. Ney. Branching Processes. Springer, Berlin 1972.
6. A. Bagchi and A.K. Pal. Asymptotic normality in the generalized Polya Eggenbergerurn model with an application to computer data structures. SIAMJ, Algebraic Discrete Methods 6 (3), 1985.
7. B. Chauvin, T. Klein, J.-F. Marckert and A. Rouault. Martingales and profile of binary search trees. Electron. J. Probab. 10, 2005.
8. B. Chauvin and N. Pouyanne, m-ary search trees when m =27: a strong asymptotic for the space requirements. Random Structural Algebra, 24 (2), 2004.
9. Cormen, T. H., Leiserson C. E., Rivest R. L. & Stein C, Introduction to Algorithms (Second Edition), MIT Press/McGraw-Hill,2008
10. Genesis of DB Routing Algorithm in Unicasting Networks, S.Anuradha, G.Raghu Ram, et al ,ICGST-CNIR Journal, Volume 9, Issue 1, July 2009
11. Performance of linear-space search algorithms, R.E. Korf, Weixiong Zhang, Artificial Intelligence, Vol. 79, No. 2, Dec. 1995, pp. 241-292.
12. Random Generation of Trees: Alonso and R.Schott, Kluwer, Boston ,1995.
13. Data structures and Algorithms, A Aho, J. Hopcroft and J. Ullman, Addison Wesley, Reading, MD, 2005
14. Fundamentals of computer algorithms, Ellis Horowitz and Sartaj Sahani, Computer Science Press, Rockville, MD, 2008
15. The Art of Computer programming, Volume-1,2,3 , fundamentals Of Algorithm, D.E.Kunth ,Addison Wesley, Reading, MA, 2008
16. Algorithms, R. Sedgewick, Addison Wesley, Reading, MA,2003
17. Constructing evolutionary trees : Algorithms & Complexity, Anna Ostlin, Department of Computer Science, Lund University,Sweeden,2001
18. Artificial Intelligence, Strategies, Applications and Models through Search, Christopher Thornton, Benedict du Boulay,Amacon Publications,New York,2e,1998