SYNTHESIS OF SELECTED TREE SEARCHING ALGORITHMS

[1]PARAG BHALCHANDRA, [2]SANTOSH KHAMITKAR, [3]PAWAN WASNIK

[1,2]Assistant Professor at School of Computational Sciences, S.R.T.M. University, Nanded, MS, 431606, India
[3]Research Scholar and purusing his M.Phil course in Computer Science from School of Computational Sciences, S.R.T.M. University, Nanded, MS, 431606, India
E-mail: srtmun.parag@gmail.com, s.khamitkar@gmail.com, srtmun.parag@gmail.com

Abstract- The intention of this paper is two folds, one to review selected tree search algorithms / search strategies that can be applied to a problem space and the second objective is to stimulate to implement recent developments in the complexity behavior of search strategies. The algorithms discussed here apply in general to both brute force and heuristic searches over tree spaces.

Keywords- Trees Search, Asymptotic Complexity, Brute Force, Heuristics Algorithms

I. INTRODUCTION

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 [1,2]. Trees find occurrence in Discrete Mathematics, Population Studies, Graph theories, Network Design Theories, Compilation & Parsing approaches, Optimization Theories, etc [1,6,7,12]. 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. 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. Such arrangement drastically reduces searching time and related time complexity. 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]. 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. 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 – 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].

II. SYNTHESIS OF TREE SEARCH ALGORITHMS

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.

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].

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 . 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] 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]

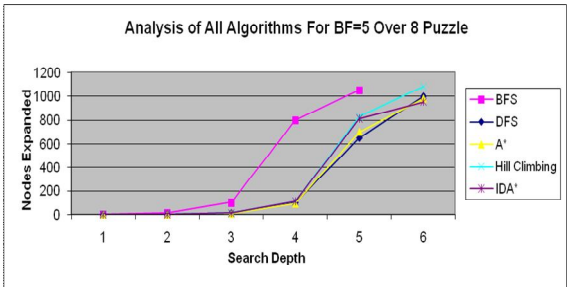


Figure 1: Analysis of all Algorithms for Branching Factor = 5 over 8 Puzzle

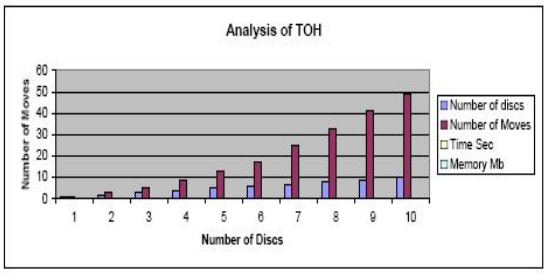


Figure 2: Analysis of TOH using DFS and BFS

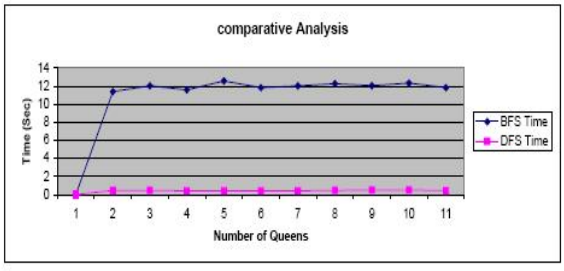


Figure 3: Comparative Analysis of N-Queens using DFS and BFS

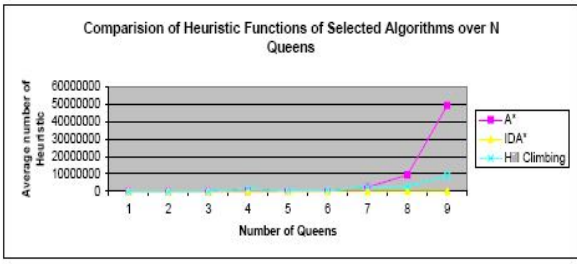


Figure 4: Comparative Analysis of Heuristic Functions in N-Queens using A*, IDA* and Hill Climbing

CONCLUSION

Previous research on the tree search has been fairly limited due to the widespread acceptance of the standards. It was more theoretical as no sophisticated hardware platforms were available then times. Thus a reformulation of tree search methods is extreme need of time. Since theoretical analysis of Tree searching algorithms does not give much of an idea of how well a given algorithm will perform in a specific situation, we need experimentations also. Therefore we have to run experiments mostly in order to achieve analytical characterization. Hence rigorous empirical analysis would help here. During the course of the research, it's observed that the real time analysis of tree searching methods is very interesting and vague. We can not come with concrete ideas Our paper in this context highlights that there is a tremendous scope for research between the investigation of general purpose methods that can be applied across domains and the discovery and exploitation of the special knowledge, heuristics, tricks and shortcuts that can be applied in particular domains to improve performance. This will give a new perspective for tree searching methods.

ACKNOWLEDGMENT

Prof. Richard Korf , for all his generous support

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

[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 ,

[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

[19] Pathology of Trees, Tree Searches and their Computational Complexities, P.U.Bhalchandra, Dr.S.D.Khamitkar , S.N.Lokhande, JIKRCSA International Journal , Vol-2, Issue-02, 2014 .

[20] A Review of Contemporary Research Investigations in selected Brute Force and Heuristics Tree Search Algorithms , P.U.Bhalchandra, Dr.S.D.Khamitkar , et al , IJSHRE International Journal , Vol 2, Issue -1,

[21] Experimental Analysis of selected constraint satisfaction problems using tree searching algorithms, P.U.Bhalchandra et al, Proceedings of ACM Sponsored ICCI-2014, Mumbai.



Parag Bhalchandra is an Assistant Professor at School of Computational Sciences, SRTM University, Nanded, MS, India. He has 10+ research papers in international journals and conferences. He has participated in more than 20 conferences and workshops. His research interest includes Algorithms, Data mining



Dr.Santosh Khamitkar is an Associate Professor at School of Computational Sciences, SRTM University, Nanded, MS, India. He has 30+ research papers in international journals and conferences. He has participated in more than 10 conferences and workshops. His research interest includes Network Security, Algorithms and ICT. He is a research supervisor and has guided 5 PhD students



Mr.Pawan Wasnik is a Research Scholar at School of Computational Sciences, SRTM University, Nanded, MS, India. He has 3+ research papers in international journals and conferences. He has participated in more than 03 conferences and workshops. His research interest includes Data Mining and Algorithms.

★★★