International Journal of Software & Hardware Research in Engineering
ISSN No: 2347-4890 Volume 2 Issue 1/ January 2014

Review of Contemporary Research Investigations in Selected Brute-Force and Heuristic Tree Search Algorithms

[1]Parag Bhalchandra ,[2]Dr.S.D.Khamitkar, [3]Dr.N.K.Deshmukh, [4]Pawan Wasnik , [5]D.R.Patil

[1,2,3,4] School of Computational Sciences,

SRTM University, Nanded, MS, India

[5] Kusumtai Patil Mahila Mahavidyalaya ,

Islampur,Sangli, MS,India

ABSTRACT

This paper focuses to analyze why in some situations, the complexity of searching rises to worst case? Such analysis can be understood by reviewing basic features of the considered tree searching algorithms. In this work, we have stick up to the selected five algorithms DFS, BFS, A,IDA*, Hill climbing from the categories of Brute Force Searches and Heuristic Searches .While making rigorous analysis of them, we take into considerations, all the research inventions and contemporary changes made into them. We feel that this paper can be a lucid note for elementary researchers in computational complexities.*

General Terms

Tree Search Complexity

Keywords

Tree Search, Brute Force, Heuristic Search, Complexity

1. INTRODUCTION

In complexity analysis, numerical methods are preferred over other methods because of their efficiency and repeatability [2,19].Within numerical methods, those based on optimality criteria are to be preferred because they allow for hypothesis testing and tree comparisons based on objective measures. However, methods based on optimality criteria are more time consuming [19]. The reason for this is simple, in order to choose an optimal solution, multiple solutions need to be compared. The main optimality criteria are maximum likelihood. Brute force can not give us a clue for maximum likelihood where as heuristics does .While their limits on efficient searches differ due to the computation requirements by each method, the issues discussed in this chapter apply, at least in principle, to both search methodologies. Finding the optimal tree for a given problem space is called tree search and is a NP complete problem [15], a problem that is unlikely to have a solution in polynomial time. Tree searches are difficult to understand due to the exponential growth of possibilities when increasing the number of nodes or levels in the tree. Usually it is understood when we traverse down in the tree, traverse down to greater depth, in the search of a solution or a goal. If a search method were to compare against all possible trees using an explicit enumeration technique, an optimality value i.e., score for maximum likelihood would be assigned to each type of tree and those that optimize the selected criterion would be chosen[23]. However, this does not happen in reality as explicit enumeration is not a very efficient method and there are many algorithmic speedups that will find the optimal solution without the burden of evaluating all possible trees [19]. An alternative solution to explicit enumeration is the use of shortcuts that guarantee finding all optimal trees. The most common shortcut is the heuristics algorithm [18]. Indeed most investigators used different types of heuristics to attempt achieving an optimal solution.

It was a common question before all researchers where they often wonder what heuristic algorithms will yield the best and fastest result [25,27]. The answer to this question is not trivial since tree search depends not only on the number of nodes in it, but also on the structure of the analyzed data tree. The only good recipe one can receive for performing tree searches would be dataset specific, and therefore only by understanding the algorithms involved. Then only, the researcher will be able to design a proper search strategy. Nonetheless, certain techniques tend to work better than others for most data sets, and this knowledge can be used as a starting point for experimenting and fine-tuning the tree search algorithms. The intention of this chapter 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 search strategies. The algorithms discussed here apply in general to both brute force and heuristic searches. We start with most general search algorithms, brute-force searches. Since they do not require any domain specific knowledge, all that is required for them is a state description, a set of legal operators, an initial state and a description of the goal state

2. REVIEWING PRESENT STATUS

2.1 BREADTH FIRST SEARCH

Breadth-first search expands nodes in order of their distance from the root, generating one level of the tree at a time until a solution is found. Since it always expands all nodes at a given depth before expanding any nodes at a greater depth, the first solution path found by breadth-first search will be one of shortest length It is most easily implemented by maintaining a queue of nodes, initially containing just the root, and always removing the node at the head of the queue, expanding it, and adding its children to the tail of the queue. Since it never generates a node in the tree until all the nodes at shallower levels have been generated, breadth-first search always finds a shortest path to a goal. The amount of time used by breadth-first search is proportional to the number of nodes generated, since each node can be generated in constant time, and is a function of the branching factor and the solution depth [20, 24]. Since each level of the tree must be saved in order to generate the next level, and the amount of memory is proportional to the number of nodes stored, the space complexity of breadth-first search is also proportional to the number of nodes explored. As a result, breadth-first search is severely space-bound in practice. As a practical matter, a breadth-first search of most problem spaces will exhaust the available memory long before an appreciable amount of time is used. The reason for this is that the typical ratio of memory to speed in modern computers is a million words of memory for each million instructions per second (MIPS) of processor speed. For example, if we can generate a million states per minute and require a word to store each state, memory will be exhausted in one minute [15,25, 29].

2.2 DEPTH FIRST SEARCH

Depth-first search avoids this memory limitation. It works by always generating descendant of the most recently expanded node, until some depth cutoff is reached, and then backtracking to the next most recently expanded node and then continue with same logic by generating one of its descendants. Therefore, only the path of nodes from the initial node to the current node must be stored in order to execute the algorithm Depth-first search remedies the space limitation of breadth-first search by always generating next a child of the deepest unexpanded node. It works on LIFO mechanism as it uses a stack to store explored node. More commonly, depth-first search is implemented recursively, with the recursion stack taking the place depth. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node. The time complexity of a depth-first search is proportionate to the depth of the tree as DFS generates the same set of nodes as breadth-first search, but simply in a different order. Thus, as a practical matter, depth-first search is time-limited [9,20,24]. Since the space used by depth-first search grows only as the log of the time required, the algorithm is time-bound rather than space-bound in practice. The main drawback, however, to depth-first search is the requirement for an arbitrary cutoff depth. If branches are not cut off and duplicates are not checked for, the algorithm may not terminate. The another disadvantage of depth-first search is that it may not terminate on an infinite tree, but simply go down the left-most path forever. Although the ideal cutoff is the solution depth d, this value is rarely known in advance of actually solving the problem. If the chosen cutoff depth is less than d, the algorithm will fail to find a solution, whereas if the cutoff depth is greater than d, a large price is paid in execution time, and the first solution found may not be an optimal one A variation of DFS called, Depth-first iterative-deepening (DFID) combines the best features of breadth-first and depth-first search [15, 18]. Some times, breadth-first search may be much more efficient than any depth-first search. Its because , the complexity of breadth-first search grows only as the number of nodes at a given depth, while the complexity of depth-first search depends on the number of paths of a given length[25].

The problem with all BFS and DFS based search algorithms is that their time complexities grow exponentially with the problem size .This is called as combinatorial explosion and as a result , the size of problems that can be solved with these techniques is quite limited . For example, while the eight puzzle with about 10 3 states, it is easily solved by BFS; the fifteen puzzle contains 1013 states and hence can not be solved with BFS on current machines. To overcome these sort of difficulties, we have to either enhance the hardware constraints or refine the search strategies such that solutions can be intelligently searched. Heuristic search is one such innovative approach for intelligent searching. In order to solve larger problems, heuristics algorithms demands domain-specific knowledge must be added to improve search efficiency [25,29].

The DFS and its variants are called backtracking search and use relatively less memory as only one successor is generated at a time rather than all successors. Each partially expanded node remembers which successor to generate next. These are the most common algorithm for solving constraint satisfaction problem (CSP). A constraint satisfaction problem is a special kind of search problem in which states are

defined by the values of a set of variables and the goal test specifies a set of constraints that the value must obey. Constraint may be higher order, Unary or binary. When search is applied to a constraint problem, it is called as constraint search. A constraint search does not refer to any specific search algorithm but to a layer of complexity added to existing algorithms that limit the possible solution set [21]. Heuristic and acquired knowledge can be combined to produce the desired results [28].

Heuristic is a problem specific knowledge that decreases expected search efforts. It is a technique which some times work but not always. Heuristic search algorithms use information about the problem to help directing the path through the tree search space. These searches use some functions that estimate the cost from the current state to the goal presuming that such function is efficient. Generally heuristic incorporates domain knowledge to improve efficiency over blind search. In AI, heuristic search has a general meaning, and a more specialized technical meaning. In a general sense, the term heuristic is used for any advice that is often effective, but isn't guaranteed to work in every case[22]. Within the heuristic search literature, however, the term heuristic usually refers to the special case of a heuristic evaluation function. In a single-agent path-finding problem, a heuristic evaluation function estimates the cost of an optimal path between a pair of states. For example, Euclidean or Airline distance is an estimate of the highway distance between a pair of locations. A common heuristic function for the sliding-tile puzzles is called Manhattan distance. It is computed by counting the number of moves along the grid that each tile is displaced from its goal position, and summing these values over all tiles[29].

For a fixed goal state, a heuristic evaluation is a function of a node, h(n), that estimates the distance from node n to the given goal state. The key properties of a heuristic evaluation function are that it estimates actual cost, and that it be inexpensive to compute. In addition, most naturally occurring heuristic functions are lower bounds on actual cost, and this property is referred to as admissibility. A number of algorithms make use of heuristic functions, including A* algorithm, Iterative-Deepening-A*, Hill climbing , etc

To understand these algorithms , we need to understand the basic pure heuristic search algorithm. The pure heuristic search, expands nodes in order of their heuristic values h(n) [3]. It maintains a Closed list of those nodes that have already been expanded, and an Open list of those nodes that have been generated but not yet expanded. The algorithm begins with just the initial state on the Open list. At each cycle, a node on the Open list with the minimum h (n) value is expanded, generating all of its children, and is placed on the Closed list. The heuristic function is applied to the children, and they are placed on the Open list in order of their heuristic values. The algorithm continues until a goal state is chosen for expansion. The main drawback of pure heuristic search is that since it ignores the cost of the path so far to node n, it does not find optimal solutions [27]

2.3 A* ALGORITHM

A* is a cornerstone name of many AI systems and has been used since it was developed in 1968[20] by Peter Hart, Nils Nilsson and Betram Rapahel. It is combination of Dijksra's algorithm and best first search. It can be used to solve many kinds of problems. A* search finds the shortest path through a search space to goal state using heuristic function. This technique finds minimal cost solutions and is also directed to a goal state called A* search. The A* algorithm also finds the lowest cost path between the start and goal state, where changing from one state to another requires some cost. A*requires a heuristic function to evaluate the cost path that passes through the particular state [2].It is very good search method but with complexity problems. This algorithm is complete if the branching factor is finite and every action has fixed cost. A* requires heuristic function to evaluate the cost of path that passes through the particular state. The A* algorithm [13] combines features of uniform-cost search and pure heuristic search to efficiently compute optimal solutions. A* is a best-first search[29] in which the cost associated with a node is f(n) = g(n)+h(n), where g(n) is the cost of the path from the initial state to node n, and h (n) is the heuristic estimate of the cost of a path from node n to a goal. Thus, f(n) estimates the lowest total cost of any solution path going through node n. At each point a node with lowest f value is chosen for expansion. Ties among nodes of equal f value should be broken in favor of nodes with lower h values. The algorithm terminates when a goal node is chosen for expansion. The A* finds an optimal path to a goal if the heuristic function h(n) is admissible, meaning it never overestimates actual cost[13]. In addition, A* makes the most efficient use of a given heuristic function in the following sense: among all shortest-path algorithms using a given heuristic function h(n), A* expands the fewest number of nodes[27]. The main drawback of A*, and indeed of any best-first search, is its memory requirement. Since at least the entire Open list must be saved, A* is severely space-limited in practice, and is no more practical than breadth-first search on current machines. For example, while it can be run successfully on the Eight Puzzle, it exhausts available memory in a matter of minutes on the Fifteen Puzzle[21].

2.4 IDA* ALGORITHM

The iterative- deepening-A* (IDA*) eliminates the memory constraint of A*, without sacrificing solution optimality[18]. Each iteration of the algorithm is a complete depth-first search that keeps track of the cost, f (n) = g(n) + h(n), of each node generated. As soon as a node is generated whose cost exceeds a threshold for that iteration, its path is cut off, and the search backtracks before continuing. The cost threshold is initialized to the heuristic estimate of the

initial state, and in every successive iteration, it is increased to the total cost of the lowest-cost node that was pruned during the previous iteration. The algorithm terminates when a goal state is reached whose total cost does not exceed the current threshold. Since IDA* performs a series of depth-first searches, its memory requirement is linear in the maximum search depth. In addition, if the heuristic function is admissible, the first solution found by IDA* is an optimal one. It is observed that the IDA* expands the same number of nodes, asymptotically, as A* on a tree, provided that the number of nodes grows exponentially with solution cost. These facts, together with the optimality of A*, imply that IDA* is asymptotically optimal in time and space over all heuristic search algorithms that find optimal solutions on a tree. Additional benefits of IDA* are that it is much easier to implement, and often runs faster than A*, since it does not incur the overhead of managing the Open and Closed lists [29].

2.5 HILL CLIMBING ALGORITHM

Hill climbing search algorithm is simply a loop that continuously moves in the direction of increasing value, which is uphill. It stops when it reaches a "peak" where no neighbor has a higher value. The hill climbing comes from that idea that if you trying to find the top of the hill and you go up direction from where ever you are. The question that remains is whether this hill is indeed the highest hill possible. Unfortunately, without further extensive exploration, that question cannot be answered [20].This techniques works but as it uses local information that's why it can be fooled .The algorithm does not maintain a search tree, so the current node data structure need only record the state and its objective function value [13]. In this algorithm only a local state is considered when making a decision of which node is to expand next? When a node is entered all of its successors nodes have a heuristic function applied to them. The successor node with the most desirable result is chosen for traversal [24].Hill climbing sometimes called greedy local search because it catches a good neighbor state without thinking ahead about where to go next. Hill climbing often makes very rapid progress towards a solution because it is usually quite easy to improve a bad state. Hill climbing is best suited to the problems where the heuristic gradually improve the closer it gets to the solution it works badly where there are sharp drop-offs. It assumes that local improvement will lead to global improvement. There are some reasons by which hill climbing often gets suck which are stated below[27],

Local Maxima: A local maximum is the peak that is higher than each of its neighboring states, but lower than the global maximum. Hill climbing algorithms that reach the vicinity of local maximum will be drawn upwards towards the peak, but then will be stuck with nowhere else to go.

Ridges : Steps of east, north south and west may go down but the step to north west may go up. Ridges result in a sequence of local maxima that is very difficult for greedy algorithm to navigate .

Plateaus : The space has a broad flat area that gives the search algorithm no direction (random walk).Many variants of hill climbing have also been invented which are described below. This variant chooses at random from among the uphill moves and the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than steepest ascent but in some state landscapes it finds better solution.

There are many versions of Hill climbing algorithm. The First choice Hill climbing variant implements stochastic hill climbing by generating successors randomly until one is generated that is better than current state. This is a good strategy when a state has thousands of successors [27]. The Random Hill Climbing variant adopts the well known adage, if at first you don't succeed try again and again. It conducts a series of hill climbing searches from randomly generated initial states, stopping when a goal is found.

On the context of discussions of all the above algorithms, we have observed that, by common sense, we can reduce the complexity of searching [23]. Instead calling a tree search algorithm blindly, why do not we make analysis of the problem space first and then determine the searching strategy. Further, it is observed that, for many problems, the search tree is finite that is depth of the tree is known in advance. For example, consider the travelling salesman problem (TSP) of visiting each of a set of cities and returning to the starting city in a tour of shortest total distance. The most natural problem space for this problem consists of a tree where the root node represents the starting city, the nodes at level one represent all the cities that could be visited first, the nodes at level two represent all the cites that could be visited second, etc . In this tree, the maximum depth is the number of cities, and all candidate solutions occur at this depth. In such a space, a simple depth-first search guarantees finding an optimal solution using space that is only linear in the number of cities. Instead, if we invoke a BFS or any Heuristic algorithm, complexity behavior could go entirely different and that too in worst case direction [25].

3. CONCLUSION

The standard breadth-first and depth-first algorithms are seen to be inferior to the heuristics algorithms. We have found that these algorithms are asymptotically optimal along all three dimensions for exponential tree searches. Since almost all heuristic tree searches have exponential complexity, this is a fairly general result. We have also found that the complexity of a problem can be expressed in terms of two parameters: the branching factor of the problem space, and the depth of solution of the problem. Further, none of the algorithm is best , rather the algorithms are situational better to another.

4. ACKNOWLEDGMENTS

Prof. Richard Korf , computer science at the University of California, Los Angeles.

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, Parag Himanshu Dave et al, Pearson Education, 1e,2008

[4] The continuum random tree I, I, IIII: An overview in Stochastic Analysis, D. Aldous. Cambridge Univ. Press, Cambridge 1991.

[5] Branching Processes, K.B. Athreya and P.E. Ney.Springer, Berlin 1972.

[6] Asymptotic normality in the generalized Polya Eggenbergerurn model with an application to computer data structures Bagchi and A.K. Pal. SIAMJ, Algebraic Discrete Methods 6 (3), 1985.

[7] Martingales and profile of binary search trees, Chauvin, T. Klein, J.-F. Marckert and A. Rouault. Electron. J. Probability. 10, 2005.

[8] m-ary search trees when m =27: a strong asymptotic for the space requirements, Chauvin and N. Pouyanne, Random Structural Algebra, 24 (2), 2004.

[9] Introduction to Algorithms, Cormen, T. H., Leiserson C. E., Rivest R. L. & Stein C, 2e, 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 , 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] Efficient tree searches with available algorithms ,Gonzalo Giribet, Lecture Notes, Harvard University , Cambridge , 2008

[20] Search, Artificial Intelligence , Berliner, H., Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1983.

[21] A quantitative study of search methods and the effect of constraint satisfaction, Berliner, H. and Goetsch, G., Tech. Rept. CMU-CS-84-147, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. 1984.

[22] Learning to Solve Problems by Searching for Macro-Operators , Korf, R.E., Pittman, London, 1985

[23] Human Problem Solving , Newell, A. and Simon, H.A., Prentice-Hall, Englewood Cliffs, NJ, 1972

[24] Search Methods for Artificial Intelligence, Bolc, L., and J. Cytowski, Academic Press, London, 1992.

[25] The advantages of using depth and breadth components in heuristic search, Davis, H.W, A. Bramanti-Gregor, and J. Wang, in Methodologies for Intelligent Systems 3, Z.W. Ras and L. Saitta (Eds.), North-Holland, Amsterdam, 1989, pp. 19-28.

[26] Performance measurement and analysis of certain search algorithms, Gaschnig, J. Ph.D. thesis, Department of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa, 1979.

[27] How to use limited memory in heuristic search, Kaindl, H., G. Kainz, A. Leeb, and H. Smetana, Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95), Montreal, Canada, Aug. 1995.

[28] Depth-first iterative-deepening: An optimal admissible tree search, Korf, R.E., Artificial Intelligence, Vol. 27, No. 1, 1985, pp. 97-109.

[29] Search techniques, Pearl, J., and R.E. Korf, in Annual Review of Computer Science, Vol. 2, Annual Reviews Inc., Palo Alto, Ca., 1987.