SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2011. Vol. 770
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 90–111 P.

Fuzzified Algorithm for Game Tree Search with Statistical and Analytical Evaluation

Dmitrijs Rutko[1]
Faculty of Computing, University of Latvia
Raina blvd. 19, Riga, LV-1586, Latvia
dim_rut@inbox.lv

This paper presents a new game tree search algorithm which is based on the idea that the exact game tree evaluation is not required to find the best move. Therefore, pruning techniques may be applied earlier resulting in faster search and greater performance. The experiments show that applied to an abstract domain, the presented algorithm outperforms the existing ones such as PVS, Negascout, NegaC*, SSS*/ Dual* and MTD(f). This paper also provides improvements for algorithm such as statistical and analytical game tree evaluation.

Keywords: game tree search, alpha-beta pruning, fuzzified search algorithm, performance.

1 Introduction

Games are usually represented with the help of a game tree which starts at the initial position and contains all the possible moves from each position. Classical game tree search algorithms such as Minimax and Negamax operate using a complete scan of all the nodes of the game tree and are considered to be too inefficient. The most practical approaches are based on the Alpha-beta pruning technique, which seeks how to reduce the number of nodes to be evaluated in the search tree. It is designed to completely stop the evaluation of a move if at least one possibility is found, the one that proves the current move to be worse than the previously examined move. Such moves do not need to be evaluated further.

The examples of more advanced algorithms that are even faster while still being able to compute the exact minimax value, are PVS, Negascout and NegaC*. The other group of algorithms like SSS* / Dual* and MTD(f), use best-first strategy, which can potentially make them more time-efficient, however, typically at a heavy cost of space-efficiency.

Through analyzing and comparing these algorithms it can be seen that in many cases the decision about the best move can be made before the exact game tree minimax value is obtained. The author introduces a new approach which allows finding the best move faster while visiting less nodes.

The paper is organized as follows: the current situation in the game tree search is discussed; then the idea that allows performing game tree search in a manner

[1] This research is supported by the European Social Fund project No. 2009/0138/1DP/1.1.2.1.2/09/IPIA/VIAA/004.

based on the move that leads to the best result is proposed; the algorithm structure and implementation details are explained. Thereafter, improvements to the algorithm, such as statistical self-learning and analytical evaluation, are discussed. Then, the experimental setup and empirical results on the search performance obtained in abstract domain are shown. The paper is concluded with future research directions.

2 State of the Art

Classical game tree search algorithms are based on the Alpha-beta pruning technique. Alpha-beta is a search algorithm which tries to reduce the number of nodes to be evaluated in the search tree by the Minimax algorithm. It completely stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined one. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision [12].

The illustration of the Alpha-beta approach is given in *Fig. 1*.

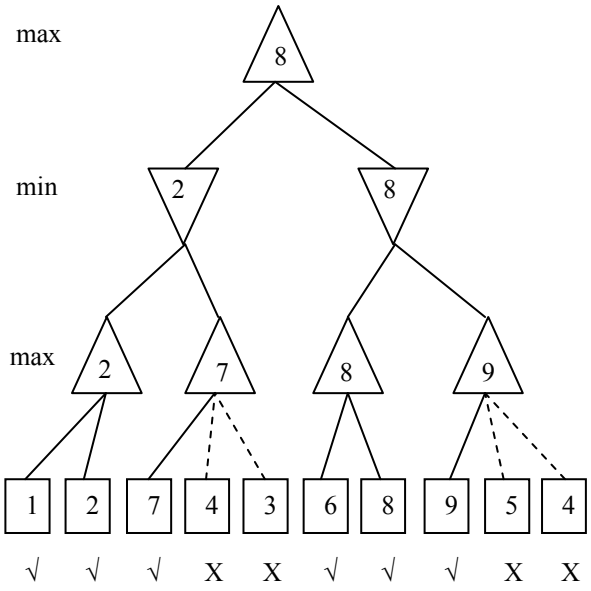


Fig. 1 Traditional Alpha-Beta approach

The game tree in *Fig. 1* has two branches with minimax values 2 and 8 for the left and right sub-trees respectively. In order to find the best move, the Alpha-beta algorithm is scanning all the sub-trees from the left to the right and is forced to evaluate almost each node. The possible cut-offs are depicted with a dashed line (at

each step, the previous evaluation is smaller than the value of currently checked node).

When all the nodes are checked, the algorithm compares the top-level sub-trees. The evaluation of the left and the right branches are 2 and 8 respectively; the highest outcome is chosen, and the best move goes to the right sub-tree.

The benefit of alpha-beta pruning lies in the fact that branches of the search tree can be eliminated. The search time can in this way be limited to the 'more promising' subtree, and a deeper search can be performed in the same time.

Since the minimax algorithm and its variants are inherently depth-first, a strategy such as iterative deepening is usually used in conjunction with alpha-beta so that a reasonably good move can be returned even if the algorithm is interrupted before it has finished execution. Another advantage of using iterative deepening is that searches at shallower depths give move-ordering hints that can help produce cutoffs for higher-depth searches much earlier than would otherwise be possible [11].

More advanced algorithms are the following:

- PVS (Principal Variation Search) is an enhancement to Alpha-Beta based on null or zero window searches of none PV-nodes to prove whether a move is worse or not than the already safe score from the principal variation [1][10].
- NegaScout, which is an Alpha-Beta enhancement. The improvements rely on a Negamax framework and some fail-soft issues concerning the two last plies which did not require any re-searches [3] [4].
- NegaC* – an idea to turn a Depth-First to a Best-First search like MTD(f) to utilize null window searches of a fail-soft Alpha-Beta routine and to use the bounds that are returned in a bisection scheme [5].
- SSS* and its counterpart Dual* are search algorithms which conduct a state space search traversing a game tree in a best-first fashion similar to that of the A* search algorithm and retain global information about the search space. They search fewer nodes than Alpha-Beta in fixed-depth minimax tree search [2].
- MTD(f), the short name for MTD(n, f), which stands for Memory-enhanced Test Driver with node n and value f. MTD is the name of a group of driver-algorithms that search minimax trees using null window alpha-beta with transposition table calls. In order to work, MTD(f) needs a first guess as to where the minimax value will turn out to be. The better than first guess is, the more efficient the algorithm will be, on average, since the better it is, the less passes the repeat-until loop will have to do to converge on the minimax value [6] [7] [8] [9].

Transposition tables are another technique which is used to speed up the search of the game tree in computer chess and other computer games. In many games, it is possible to reach a given position, which is called transposition, in more than one way. In general, after two moves there are 4 possible transpositions since either player may swap their move order. So it is still likely that the program will end up analyzing the same position several times. To avoid this problem transposition tables store previously analyzed positions of the game [11].

3 Fuzzy Approach

The author proposes a new approach, which is based on the attempt to implement a human way of thinking adapted to logical games. A human player rarely or almost never evaluates a given position precisely. In many cases, the selection process is limited to rejecting less promising nodes and making certain that the selected option is better than others. The important moment is that we are not interested in the exact position evaluation but in the node which guarantees the highest outcome.

Let the given problem be explained in details.

We could look at our game tree from a relative perspective like "is this move better or worse than some value X" (*Fig. 2*). At each level, we identify if a sub-tree satisfies "greater or equal" criteria. So passing search algorithm, for instance, with argument 5, we can obtain the information that the left branch has value less than 5 and the right branch has value greater or equal than 5. We do not know exact sub-tree evaluation, but we have found the move, which leads to the best result.

In this case, different cut-offs are possible:

- at max level, if the evaluation is greater (or equal) than the search value;
- at min level, if the evaluation is less than the search value.

In the given example, reduced nodes are shown with dashed line. Comparing to *Fig. 1* it can be seen that not only more cut-offs are possible, but also pruning occurs at higher level which results in better performance.

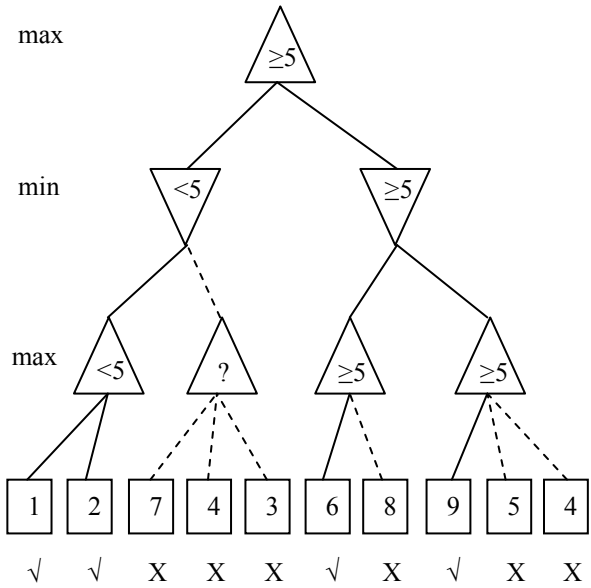


Fig. 2 Fuzzy best node approach

In this approach, the best/worst cases are the same as for alpha-beta pruning: $O(w^{d/2})$ for the best case as only one branch should be checked at cut-off level, and $O(w^{d})$ for worst case as all nodes should be checked (w is width, d is depth of the tree). But in the presented approach, cut-offs are more often possible in general.

If we use geometric interpretation and put our sub-tree minimax values on coordinate axis, then our task is to separate/divide branches so that only one branch would have higher value than the test value. *Fig. 3* illustrates our previous example. Alpha-beta window is initially set to leaf node range $\alpha = 0$, $\beta = 10$; then the following test values are used X_1, X_2, X_3. If value X_2 is chosen, then the successful separation is obtained after the first iteration – we know that the second sub-tree has higher estimation. If values X_1 or X_3 are chosen, then no separation is possible at this point – both values are on the same side of the test value. In this case, the algorithm continues with reduced alpha-beta search window: 1) $\alpha = X_1$ in the first case; or 2) $\beta = X_3$ in the second.

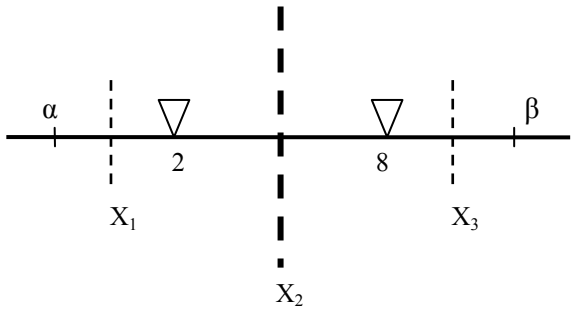


Fig. 3 Geometric interpretation of separation in the fuzzified game tree search

In a game tree with three or more sub-trees, the algorithm workflow remains the same. Our task is to separate sub-trees in a way that only one branch has higher estimation than the test value. However, more cases are possible – 0, 1, 2, 3 branches fall in on one side of separation line. In this case, alpha-beta window is reduced correspondingly and the algorithm proceeds with the next iteration.

Comparing to existing algorithms such as MTD(f) in order to work, it needs the first guess as to where the minimax value will turn out to be. If you feed MTD(f) the minimax value to start with, it will only do two passes, the bare minimum: one to find an upper bound of value x, and one to find a lower bound of the same value.

In the presented algorithm, it is possible to find the best move with a single iteration and we are not limited to the accurate first guess. For the presented example, any value from interval 3..7 (inclusive) would apply.

4 Fuzzified Search Algorithm

Best Node Search (BNS) is a new game tree search algorithm based on the idea described in the previous section. The main difference between the classical approach and the proposed algorithm is that BNS does not require the knowledge of the exact game tree minimax value to select a move. We only need to know which sub-tree has higher estimation. By iteratively performing search attempts the algorithm can obtain information about which branch has higher estimation without knowing the exact value. So less information is required and, as a result, the best move can be found faster – total number of searched nodes is smaller and total

algorithm execution time is reduced comparing to the algorithms based on the exact game tree evaluation.

The presented algorithm uses a standard call of Alpha-Beta search with 'zero window'. The proposed implementation relies on the transposition tables but variation without memory (transposition tables) usage is also possible. While scanning a game tree, algorithm checks all sub-trees and returns node which leads to the best result. In general, BNS is expected to be more efficient comparing to the classical algorithms in terms of number of nodes checked as it does not obtain additional information which is not required in many cases – the exact game tree minimax value.

BNS algorithm is given in *Fig. 4* which makes use of the following functions:

1. NextGuess() – returns next separation value tested by algorithm;
2. AlphaBeta() – alpha-beta search with Zero Window (Null Window) performs a boolean test whether a move produces a worse or better score than the passed value.

All sub-trees are tested with separation values (this information is stored in the transposition tables and reused in subsequent iterations). If exactly one branch exceeds test value, then the best node is found. If all branches have smaller estimation, then the number of sub-trees that exceeds separation test value remains the same, beta value is reduced. If several nodes exceed test value, then `subtreeCount` is updated correspondingly, and alpha value is updated to test value, and algorithm continues with the next iteration. If a single sub-tree that exceeds test value cannot be found and alpha-beta range is reduced to 1, it means that several sub-trees have the same estimation and we can choose any of them.

```
function BNS(node, α, β)
    subtreeCount := number of children of node
    do
        test := NextGuess(α, β, subtreeCount)
        betterCount := 0
        foreach child of node
            bestVal := -AlphaBeta(child, -test, -(test - 1))
            if bestVal ≥ test
                betterCount := betterCount + 1
                bestNode := child
        update number of sub-trees that exceeds separation
  test value
        update alpha-beta range
    while not((β - α < 2) or (betterCount = 1))
    return bestNode
```

Fig. 4 The BNS algorithm

One of the main parts of this algorithm is the method `NextGuess(α, β, subtreeCount)` which returns the next value to be checked by the algorithm. In the simplest case, it could be a formula based on linear distribution – alpha-beta range is proportionally divided into sections according to the sub-tree count:

```
NextGuess = α + (β - α) * (subtreeCount - 1) / subtreeCount;
```

where alpha and beta are the lower and the upper bounds of the search window respectively; `subtreeCount` is the number of sub-trees which satisfies the previous test call (the branches that have higher estimation than the test value). However, the best algorithm performance is achieved after its statistical training or analytical game tree evaluation resulting in non-linear distributions. These methods are described in the following chapters.

5 BNS Enhancement through Statistical Training

Some algorithms, such as MTD(f) benefit from accurate "first guess" – as to where the minimax value will turn out to be. The better than first guess is, the more efficient the algorithm will be, on average.

The BNS algorithm can greatly benefit from good separation value as well. The better separation value is, the faster the best node will be found, on average. So self-training becomes an important part of the BNS algorithm as it helps us to tune separation test values used by algorithm during consecutive search attempts and results in reduced search space and improved performance [12].

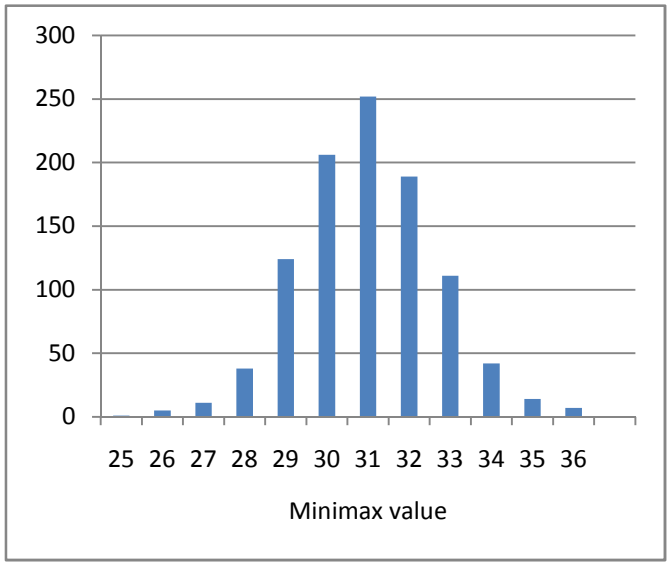
In this section, the author proposes a new multi-dimensional statistics approach which is developed to work in conjunction with BNS algorithm.

It is possible to collect this statistics before the game starts analyzing multiple test data or on-line during the game process reusing previous estimations.

Table 1

Game tree minimax value distribution over 1000 trees

Minimax value	Tree count
25	1
26	5
27	11
28	38
29	124
30	206
31	252
32	189
33	111
34	42
35	14
36	7
	1000



The statistical approach for finding initial value (first guess) can be demonstrated in the following example. 1000 game trees were generated with fixed

structure and randomly assigned values for leaf nodes in a specific range (for given example, the following values were used – width 2, depth 14, leaf node values are in interval [0; 80]). For these game trees, statistics was collected and results are shown in *Table 1*

It can be seen that there are 252 trees with minimax value 31 and there is only one tree out of one thousand with minimax value 25. These statistics results are used, for example, to determine the first guess in MTD(f) algorithm, and in all tests it was called with argument f = 31 showing its best results.

However, this information does not provide additional benefits. So new approach is proposed – single-dimension statistics is extended into two-dimension statistics meaning collecting all possible pair info – for each sub-tree in our binary tree. As a result, we have a matrix showing a number of trees having respectively one sub-tree value (columns) and other sub-tree value (rows) – *Table 2* Due to symmetry reasons (according to the main diagonal) one half is shown. Tree count column has summed up matrix values in the row resulting in the previous single-dimension statistics (*Table 1*).

It can be seen that there are 78 trees which have sub-trees correspondingly with branch values 31 and 29 (in this case, tree minimax value is 31).

Table 2

Two dimensional game sub-tree distribution over 1000 trees

	23	24	25	26	27	28	29	30	31	32	33	34	35	36	Tree count
23	0														0
24	0	0													0
25	0	1	0												1
26	0	0	2	3											5
27	0	0	5	3	3										11
28	0	1	0	12	12	13									38
29	0	0	2	10	35	43	34								124
30	1	2	6	9	26	58	71	33							206
31	0	0	6	10	27	41	78	57	33						252
32	0	1	3	13	17	30	32	41	38	14					189
33	0	0	1	2	8	12	26	28	21	11	2				111
34	0	0	0	1	3	5	13	8	6	2	2	2			42
35	0	0	0	0	0	2	4	3	2	3	0	0	0		14
36	0	0	0	0	0	0	1	2	2	1	1	0	0	0	7
															1000

BNS algorithm divides search interval into parts and verifies if sub-tree values stay in different parts or not. If one branch value is less than separation value and another branch value is higher, then algorithm immediately returns better move and stops its work. If the branch values lay in the same part, then the interval is reduced and the algorithm continues with an updated alpha-beta window. So, the algorithm

becomes more efficient with the accurate first guess when the most of the game trees get separated into parts after the first iteration.

The grayed-out rectangle in *Table 2* gives us separation distribution for X = 30. All marked cells represent trees with one branch greater or equal than 30 (by row) and the other branch less than 30 (by column). It means that all these trees will be separated into parts after the first iteration. To calculate the number of trees for separation value X = 30, we need to sum up all the marked cells. For the given example, 509 trees will get separated.

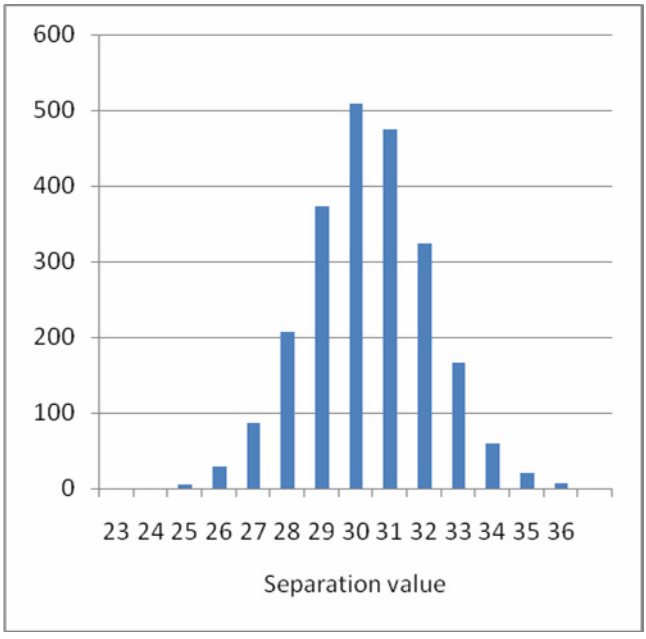
So, to find such value X when the highest number of trees will be divided, we need to build remaining rectangles along the main diagonal for each X value and sum up the cells bounded by X along axis as it is done in the previous example. The resulting table is shown in *Table 3*

As it can be seen from *Table 3*, the best results are given with X = 30, meaning that if we call BNS algorithm with argument 30, then 509 game trees will be divided into two parts and the best node will already be found after the first iteration. So trained BNS is more efficient and if we continue this idea we can find the best X value for the second, third etc. iteration, until the best node is found.

Table 3

Statistical sub-tree separation over 1000 trees

Separation value	Tree count
23	0
24	1
25	6
26	30
27	88
28	208
29	374
30	509
31	475
32	325
33	167
34	61
35	21
36	7
	2272



Note: the total count of the game trees is higher than 1000 as many values are overlapping – the same X value could divide different trees and the same tree could be divided by different X values.

If we take a look at the tree with branching factor 3, we can apply similar techniques for finding the best separation value. In this case, we have triplets [x, y, z] defining minimax value of each sub-trees, so we can build corresponding 3D matrix displaying the total number of the game trees with the given triplet.

While searching this matrix, we look for such separation value X, so one sub-tree would be greater or equal with X, and two other sub-trees would have smaller estimation. And, therefore, we maximize the number of trees which would be separated after the first method call, so the best move is found after the first iteration.

6 Game Tree Analytical Evaluation

In the previous chapter (BNS enhancement through self-training), statistical analysis, which can improve BNS algorithm performance by calculating and applying "good" separation values, was discussed. Therefore, in the development of this idea, the author offers a new approach which is based on fully analytical determination of best successful separation value generally for any type of tree with various structures (alpha-beta range, tree width, depth, etc).

As it was stated before, we use abstract domain search in our experiments – meaning tree generation with fixed structure (width / depth) and randomly assigning leaf values based on uniform distribution within a given range.

In *Fig. 5*, leaf nodes are noted as probabilistic function F_X. Here, our task is to calculate resulting function starting from the lowest level (leaf nodes) up to the top level (root node).

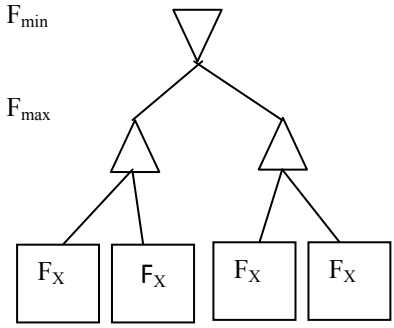


Fig. 5 Application of probabilistic function to maximum and minimum levels

In this case, the following functions demonstrate the behavior of leaf nodes:

- *Probability density function* describes the relative likelihood for this random variable to occur at a given point. For our example (leaf node values are in interval [0; 80]), this likelihood is given in Fig. 6;
- *Cumulative distribution function* describes the probability that a real-valued random variable X with a given probability distribution will be

found at a value less than or equal to X. For our example, it is given in Fig. 7.

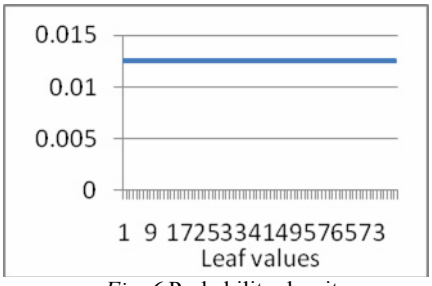


Fig. 6 Probability density

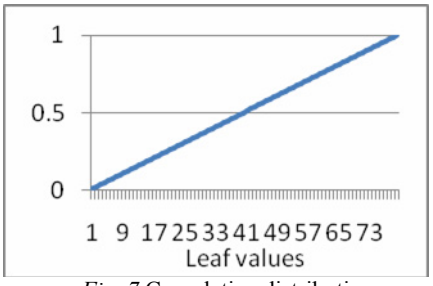


Fig. 7 Cumulative distribution

To calculate probabilistic values correspondingly at maximum and minimum levels, the author proposes the following formulas which are applicable for binary tree (square of probabilistic function) – for max level, it is probability that both sub-trees are less than our cumulative distribution function; for min level, that not both elements are greater than our cumulative distribution function:

$$F_{max} = (F_x)^2 \quad (1)$$

$$F_{min} = 1 - (1 - F_x)^2 \quad (2)$$

For trees with larger branching factor, the following general formula should be used, where *w* is the width of the tree:

$$F_{max} = (F_x)^w \quad (3)$$

$$F_{min} = 1 - (1 - F_x)^w \quad (4)$$

Correspondingly, if we apply this formula to our example with binary tree with leaf nodes in the given range [0..80], we receive the following equations:

$$F_{max} = \left(\frac{x}{80}\right)^2 \quad (5)$$

$$F_{min} = 1 - \left(1 - \frac{x}{80}\right)^2 \quad (6)$$

By using these formulas we can build up the following matrix (*Table 4.*) with iteration results and iteration values for each minimum and maximum level up to level of depth 14 (actually, we start from the lowest level with leaf nodes and go up to the highest level – the root node).

Table 4

Calculated cumulative distribution for binary tree with leaf node values from interval [0; 80] and depth 14

Leaf values		*Level*				
		1 – min	2 – max	3 – min	…	14 – max
x	F_x	$1-(1-F_x)^2$	$(F_x)^2$	$1-(1-F_x)^2$	…	$(F_x)^2$
1	1 / 80	0,02484375	0,00061721	0,00123404	…	0
2	2 / 80	0,049375	0,00243789	0,00486984	…	0
3	3 / 80	0,07359375	0,00541604	0,01080275	…	0
…	…	…	…	…	…	…
80	80 / 80	1	1	1	…	1

Fig. 8 demonstrates the progress of cumulative probability function bottom up changing its slope and coming nearer to vertical. Correspondingly, the transformed probability density function is displayed in *Fig. 9* with higher and higher peaks at each next level where the highest peak corresponds to level 14.

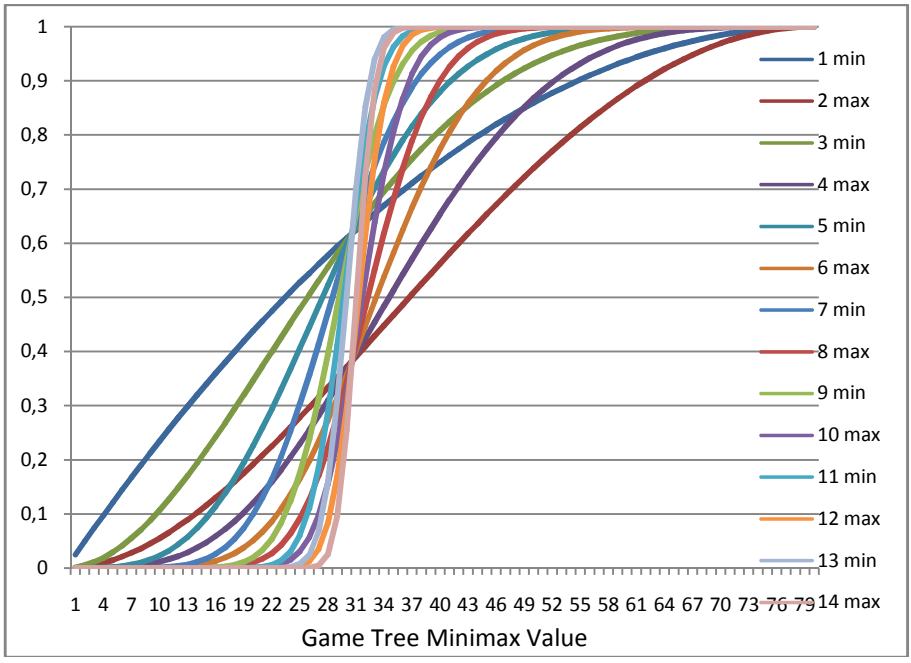


Fig. 8 Cumulative probability function by level for depth 14

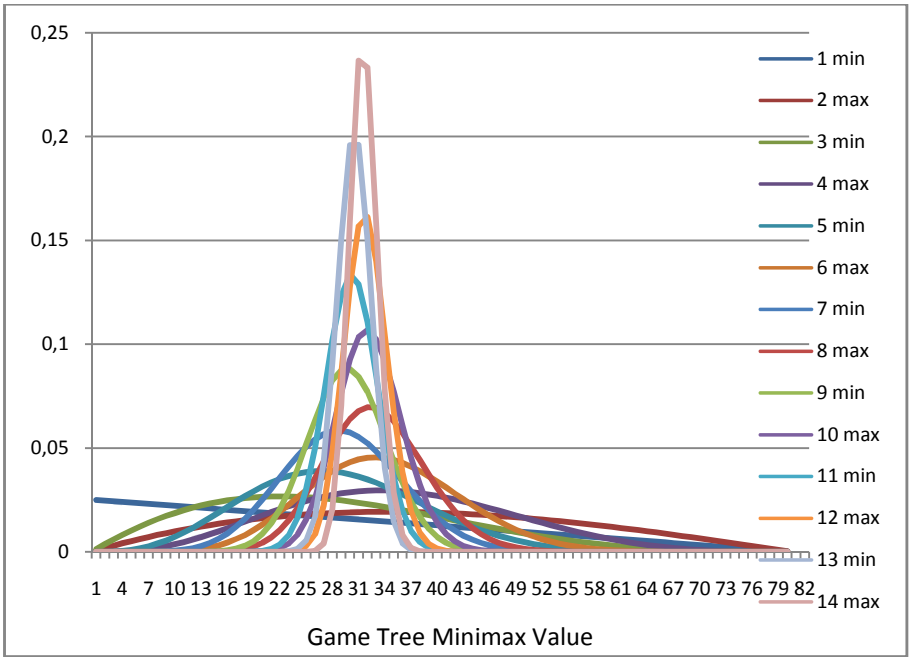


Fig. 9 Probability density function by level for depth 14

In the conducted experiments, statistical information is collected to prove the correctness of analytical game tree evaluation. The difference between analytically received data and statistical experiments is shown in *Fig. 10*. The error rate is relatively low meaning that analytical estimation is really close to experimentally received results.

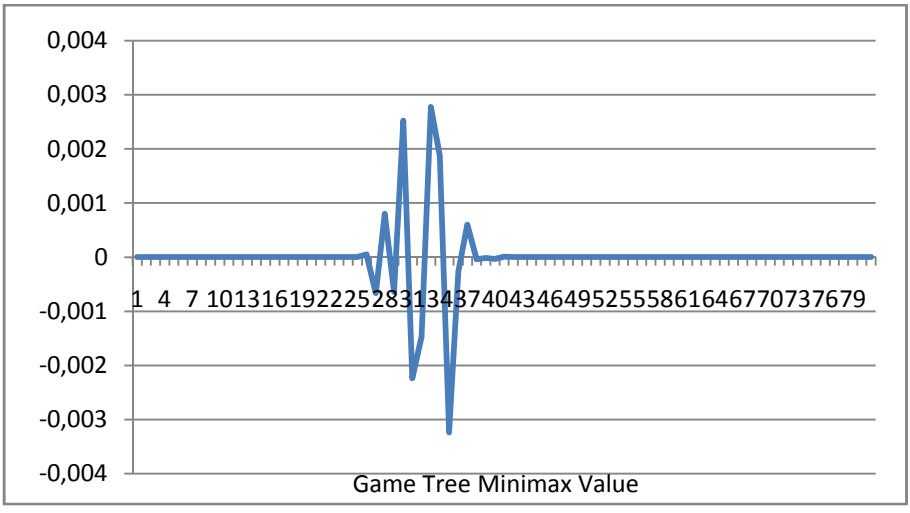


Fig. 10 Error function between analytical estimation and experimentally received results

Resulting probability density function is given in *Fig. 11*. These results correspond to the statistically received results in previous section.

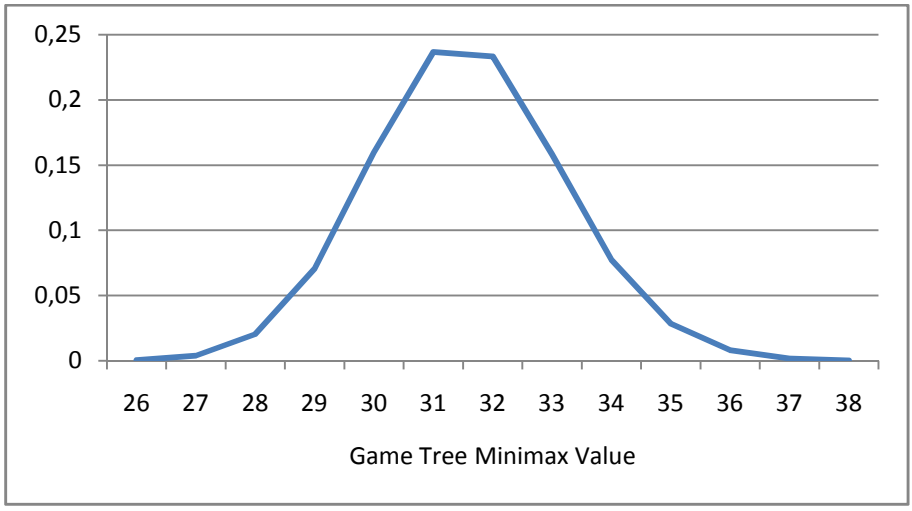


Fig. 11 Resulting zoomed-in function

Given the probability density function we can predict the most probabilistic outcome of the game tree. Thus, we can choose the best separation value for our BNS algorithm – such value X that the greatest number of trees will be separated / divided after the first iteration of the algorithm.

These are the same values as in statistical evaluation we have been used before, except that analytically we could improve precision and make calculations much faster without performing long-running experiments.

We are querying our tree with some separation value X. So, given density function, we can calculate probability, that the tree value is less than our test value, or the tree value is greater. So, our task is to maximize our chances to separate tree with the given value X.

The entropy, H, of a discrete random variable X is a measure of the amount of uncertainty associated with the value of X.

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_b p(x_i)$$

Having probabilistic outcome when tree is separated with probability P, and its counterpart outcome when tree is not separated with probability 1-P, results in Binary entropy function, H_b. The entropy is maximized at 1 bit per trial when the two possible outcomes are equally probable, as in an unbiased coin toss.

$$H_b(p) = -p \log_2 p - (1-p) \log_2 (1-p)$$

So, we should find such separation value that maximizes amount of information received after querying the tree. For the first iteration, we receive value 30. For the second iteration, we do the same way – if separation is not obtained after the first query, that means all sub-trees are either less (fall down) or greater (fall up). So, we chose the next separation value in the given range maximizing probability of successful separation. Correspondingly, the separation values for the second iteration are 29 and 31 respectively.

In *Fig. 12*, separation value X_1 is shown for the first iteration. If no successful separation is obtained after the first query, then we use the next group of separation values X_2 going to the left or to the right.

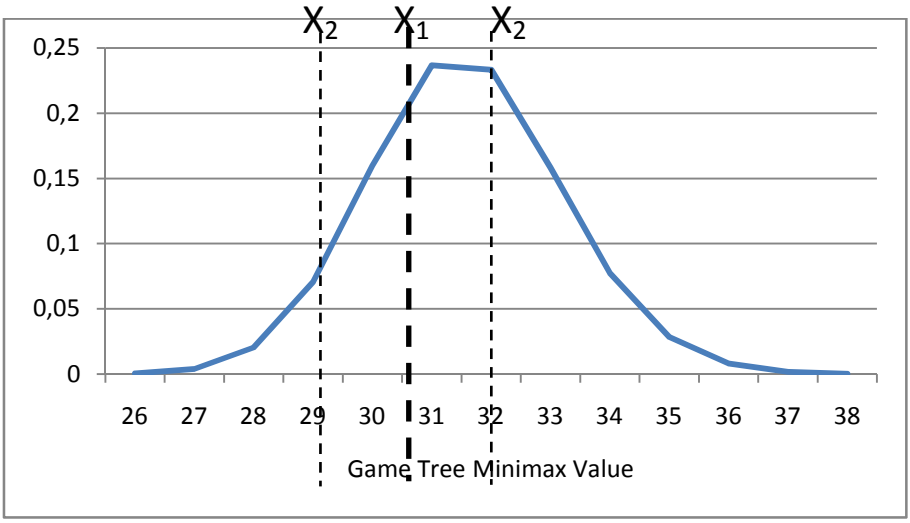


Fig. 12 Separation usage by resulting density function

Similarly, we seek for separation values for the third, the fourth etc. iterations until the best value is found. At each step, we reduce alpha-beta window. This process is similar to binary search, except for the selection separation coefficients where we use probability density function.

7 Experimental Results

More than 10 algorithms were implemented and over 40000 test runs were conducted during this research. Both versions with transposition tables and without them were used in our setup.

These algorithms were tested in an abstract domain – generating the game tree test set with fixed structure (width / depth) and randomly assigning leaf node values from the given range. Then, these experiments extended to trees with a different branching factor starting from 2 to 5 and full alpha-beta window (not limited range [-infinity, +infinity]).

All the algorithms were run on the same game tree test set (each consisting of 10000 generated samples) to compare algorithm efficiency under the same

conditions. For each algorithm, visited leaf nodes count (evaluation function call) and total visited node count was measured and average count per tree was calculated. In most cases, the first parameter is more important as in real games evaluation functions are usually complex enough and require some computing resources. The second parameter is usually less important but for some algorithms total node count increases dramatically and should be considered while comparing algorithm efficiency. In the algorithms with reiterative techniques based on transposition tables when node is visited multiple times, the total node count is increased and leaf node count remains the same as this info is stored in TT.

In the chart in *Fig. 13*, MTDF performance is taken as the base point (treated as 100%) and the performance of other algorithms is measured as a ratio to it, so a result greater than 100% means larger number of search iterations and respectively only BNS was able to show results less than 100%. It is a combined graph showing trends increasing width of the search tree – from binary tree to a tree with 5-width structure at each node. In this section, number of visited leaf nodes is counted.

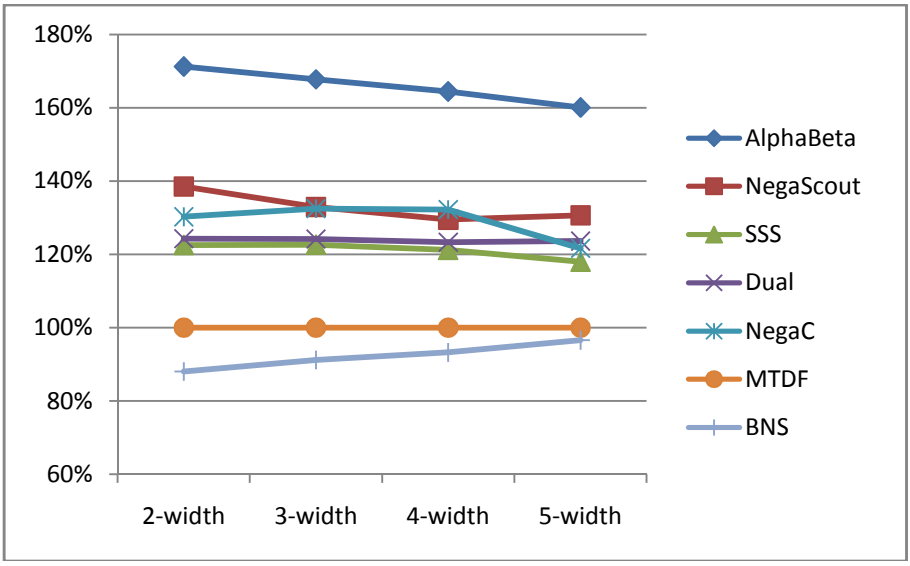


Fig. 13 Algorithm relative performance across different tree widths (leaf nodes visited)

BNS algorithm shows progress from 88% at 2-width stage to 96% at 5-width stage.

Fig. 14 demonstrates the same data slice, but here, the total number of visited nodes is measured. It can be seen that BNS performance still remains at the level of approximately 80% comparing to MTDF algorithm across all branching factors. Note: SSS and Dual algorithms show low results of 700% and 300% correspondingly and fall outside of diagram range.

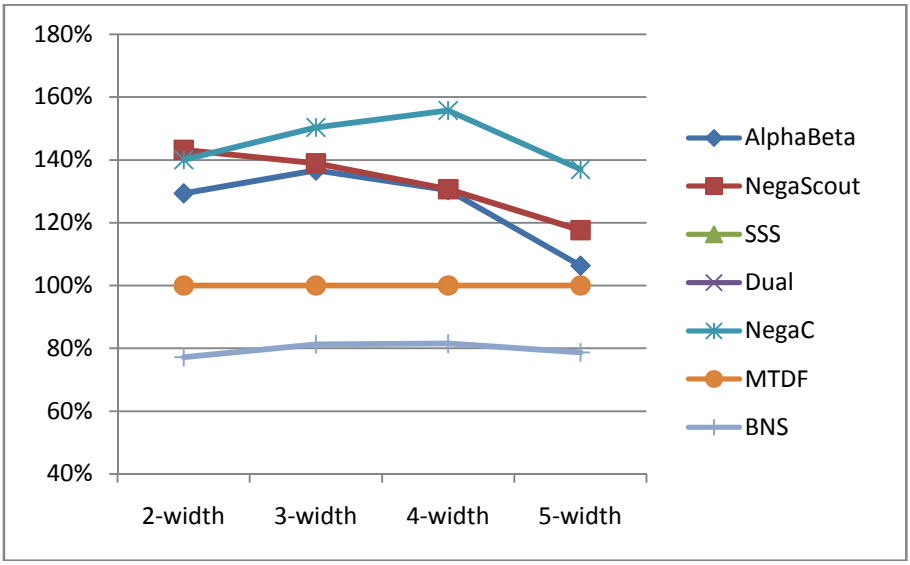


Fig. 14 Algorithm relative performance across different tree widths (total nodes visited)

8 Conclusions and Future Work

The main goal of this paper was to show that it is possible to find the best move without the exact tree minimax value. After self-training based on multi-dimension statistics, the proposed BNS algorithm was able to demonstrate better results than other existing algorithms. Game tree analytical evaluation gives additional improvement allowing us to use this algorithm as general purpose approach for different tree types.

Having analyzed the results we can conclude the following:

- Among algorithms without Transposition Tables (TT) BNS shows competitive results. Both leaf node count and total node count is less comparing to other algorithms;
- The algorithms based on TT approach show different performance in different conditions. Currently, MTD(f) is more preferable choice providing the highest performance. But in the current experiments, BNS demonstrated itself to be more efficient comparing both scanned leaf node count and total node count;
- Considering leaf nodes visited, BNS algorithm demonstrates an improvement in a range from 12% (for binary trees) to 4% (for 5-width trees) comparing to MTD(f);
- Regarding total nodes visited, BNS algorithm demonstrates a stable improvement up to 20% across different branching factors comparing to MTD(f);

The current results are based on experiments in abstract domain and additional research is needed to verify the behavior of the algorithm for wider trees (with

branch factor larger than 15-20). Interesting results may be obtained in testing non-regular trees with asymmetrical structure. Future experiments should also consider analyzing algorithm performance in real games, but it is believable that proposed approach could be successfully applied for real domain games as well.

9 Acknowledgments

The author would like to thank Nikolajs Nahimovs for valuable ideas in the field of game tree analytical evaluation.

References

1. T. A. Marsland, M. Campbell. Parallel Search of Strongly Ordered Game Trees. ACM Comput. Surv., 1982
2. Judea Pearl. The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. Communications of the ACM, 1982
3. Reinefeld, A. An Improvement to the Scout Tree-Search Algorithm. ICCA Journal, 1983, Vol. 6, No. 4, pp. 4-14
4. A. Reinefeld. Spielbaum-Suchverfahren. Informatik-Fachbericht 200, Springer-Verlag, 1989
5. Jean Christophe Weill. The NegaC* search. ICCA Journal, March 1992
6. Plaat, A., Schaeffer, J., Pijls, W., and Bruin, A. de. A New Paradigm for Minimax Search, Technical Report EUR-CS-95-03, 1994
7. Plaat, A., Schaeffer, J., Pijls, W., and Bruin, A. de. Best-First and Depth-First Minimax Search in Practice, Proceedings of Computing Science in the Netherlands, 1995
8. Plaat, A., Schaeffer, J., Pijls, W., and Bruin, A. de. An Algorithm Faster than NegaScout and SSS* in Practice, Computer Strategy Game Programming Workshop at the World Computer Chess Championship, 1995
9. Plaat, A., Schaeffer, J., Pijls, W., and Bruin, A. de. Best-First Fixed-Depth Minimax Algorithms, Artificial Intelligence, volume 87, 1996
10. Yngvi Björnsson. Selective Depth-First Game-Tree Search. Ph.D. thesis, University of Alberta, 2002
11. Russell, Stuart J.; Norvig, Peter, Artificial Intelligence: A Modern Approach (3rd ed.), Upper Saddle River, New Jersey: Pearson Education, Inc., 2010
12. Dmitrijs Rutko, Fuzzified Algorithm for Game Tree Search. Second Brazilian Workshop of the Game Theory Society, BWGT 2010

Appendix

The following section contains the performance results of algorithms implemented during the current research for different tree structures and leaf node ranges.

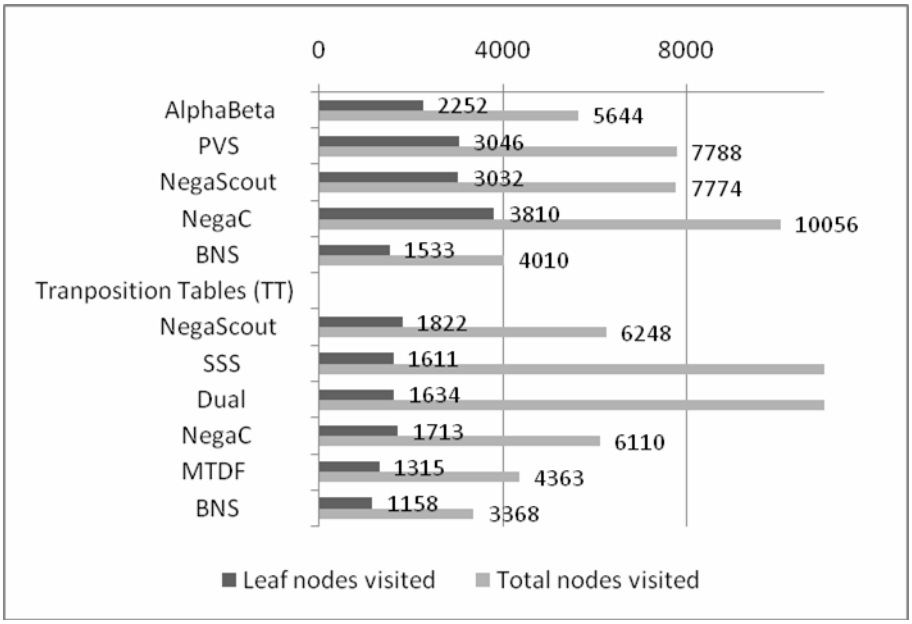


Fig. 15 Tree width – 2, depth – 14

Leaf node range 0..80; full alpha-beta window

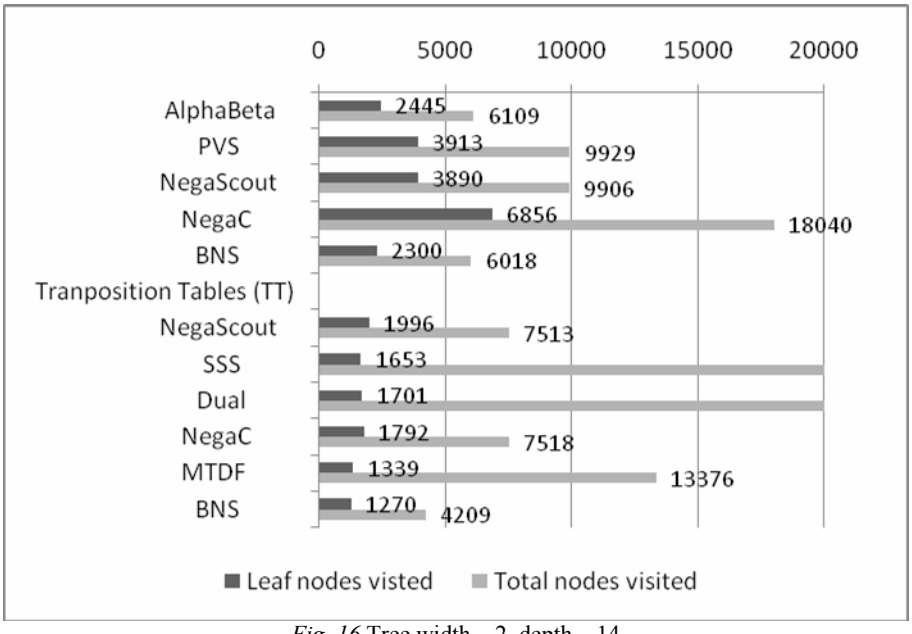


Fig. 16 Tree width – 2, depth – 14

Leaf node range 0..800; full alpha-beta window

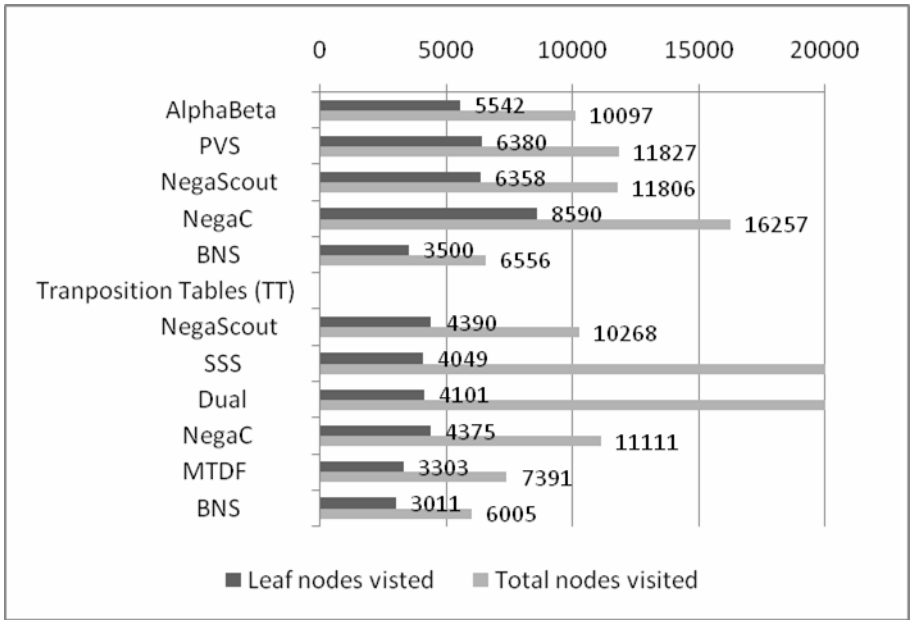


Fig. 17 Tree width – 3, depth – 10

Leaf node range 0..80; full alpha-beta window

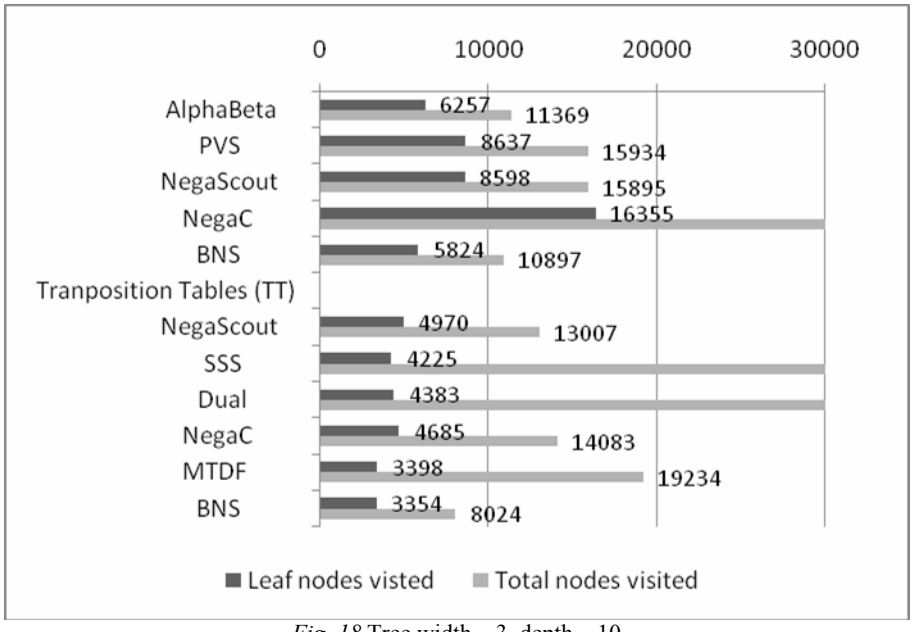


Fig. 18 Tree width – 3, depth – 10

Leaf node range 0..800; full alpha-beta window

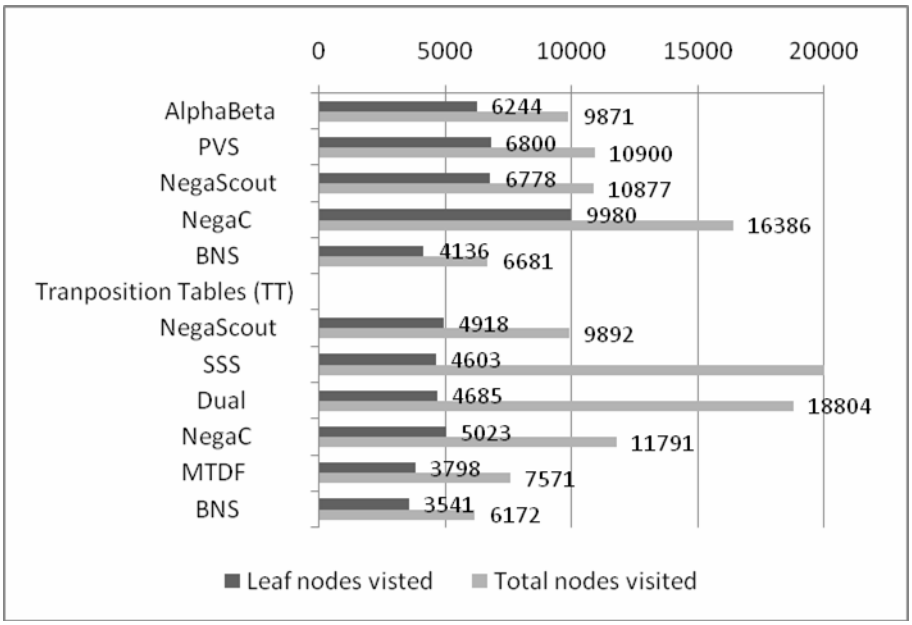


Fig. 19 Tree width – 4, depth – 8

Leaf node range 0..80; full alpha-beta window

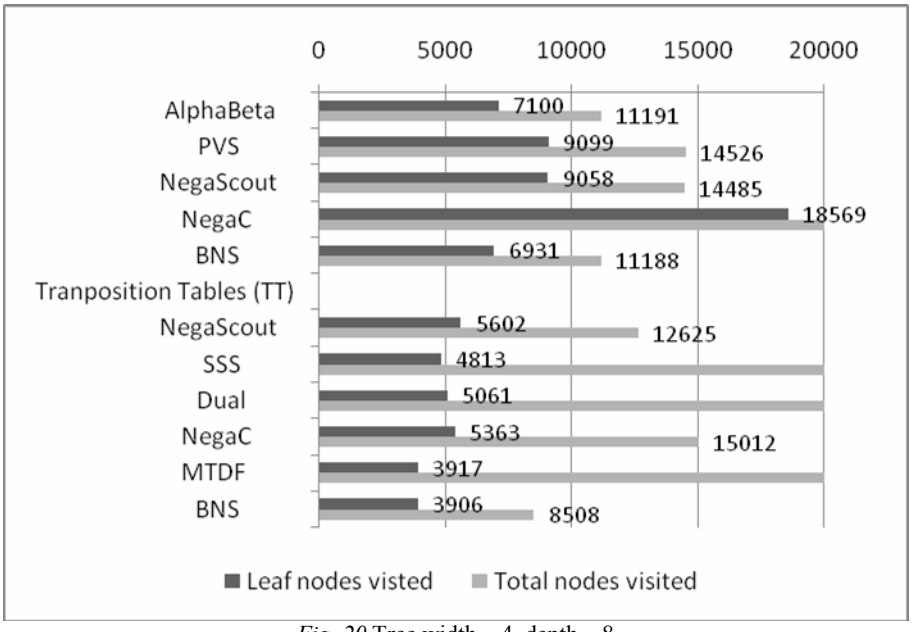


Fig. 20 Tree width – 4, depth – 8

Leaf node range 0..800; full alpha-beta window

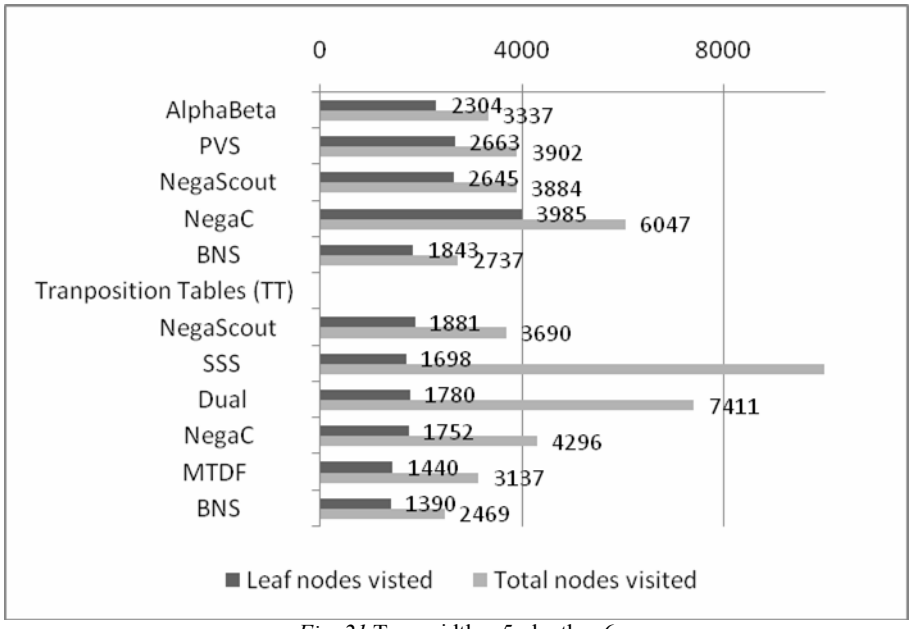


Fig. 21 Tree width – 5, depth – 6

Leaf node range 0..80; full alpha-beta window

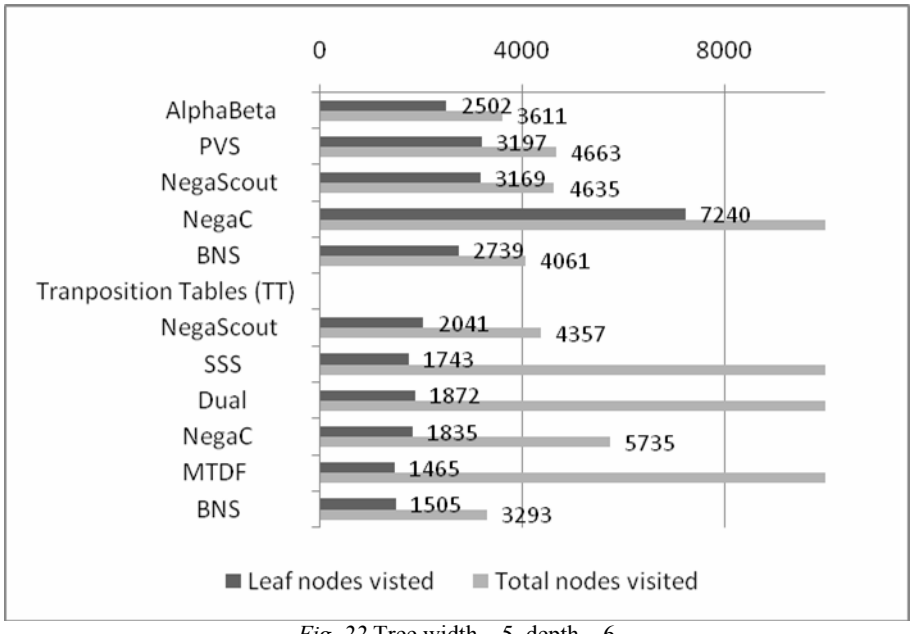


Fig. 22 Tree width – 5, depth – 6

Leaf node range 0..800; full alpha-beta window