The Application of Minimax Decision Rule in Games

Milena Karova[1], Lyubomir Genchev[2], Lyubomir Vasilev[3], Ivaylo Penev[4]

Abstract – **This paper demonstrates three different applications: Minimax algorithm, Alpha Beta pruning algorithm and Genetic Algorithm in games. They are used to evolve a Tic Tac Toe and Chess games. The size of strategies space is defined by the number of all possible game situations, which follows from the question of how many distinct matches can be played. The using of GA implementation improves the optimal paths and decreases the playing time.**

Keywords – **Minimax strategy, Alpha Beta Pruning, Genetic Algorithm, Fitness Function, Game Tree Decision.**

I. Introduction

The Minimax decision rule is applied as a solution to two-player zero-sum games and in those cases is equal to the Nash equilibrium. Since in these types of games both players work towards the same mutual goal and one player's moves towards winning directly affect the chances of winning for the other player in a negative manner.The Minimax theory is based on maximizing the potential gain for one player while minimizing it for his opponent (and vice versa).An algorithm exists in computer science which implements the Minimax decision rule and is normally used for simple two-player zero-sum games (e.g. Tic-tac-toe). It can also be applied to more complex games such as Chess and Go but without additional optimization it is highly inappropriate.

II. Minimax in General

`The Minimax algorithm [2] works by scanning the nodes (and all of its children) of a game tree from a given configuration and evaluates them based on the Minimax theory. The algorithm is in fact a form of depth-first search and on a programme level is normally implemented as a recursive algorithm. Two basic strategies exist for the Minimax algorithm: the first one consists of searching every children of every node of the whole tree and the second limits the depth of the search in order to save computational time. Both strategies have their pros and cons. The first provides a highly unlikely chance for a mistake but demands more time and is virtually impossible to implement in complex games. The second can greatly limit the computational time but does by creating the so-called "horizon effect", i.e. limiting the number of children nodes the computer can search ahead, this can lead to the choice of a move that might later prove to be bad (but the computer could not have predicted it). However, there are some methods that optimize the Minimax algorithm, the most popular of which is the alpha-beta pruning, which minimizes the time necessary for the machine to complete the task while at the same time allowing more depth for the search.

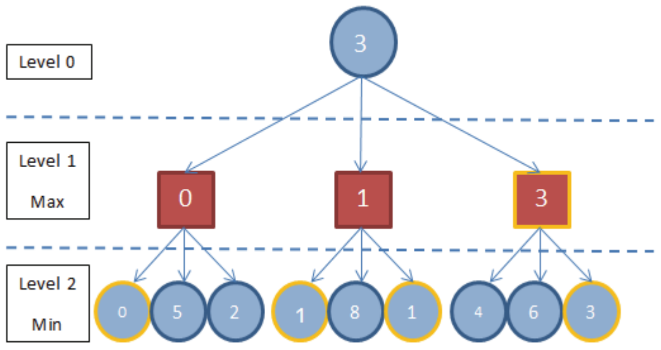

Fig. 1. The Basic Principal of Minimax Algorithm

Fig. 1 demonstrates the basic principal of the Minimax algorithm. Following the Minimax theory, the program will attempt to maximize one's player score while minimizing the others. The best move is the one that brings the most benefit to the maximizing player and the least to his opponent. Since two-player zero-sum games have a shifting nature, meaning that as the first player tries to maximize his own score in the first move, the second will try to minimize the first player's score in his own turn, the algorithm changes its action with every move. As in the figure, in level 1 the program is maximizing, while in level 2 – minimizing, hence the name "Minimax". In games, the algorithm works by analysing a given board configuration (in this case, that would be the node of level 0). The computer needs to choose the best move so it analyses all possible moves (the nodes of the game tree) to distinguish the one that brings the most benefit. Because the algorithm is maximizing in level 1, the most beneficial move would be the highest rated one. In order to rate the nodes (and, consequently, the moves themselves), the computer must search through all the children of the nodes. In a situation of a board game, for example Tic-tac-toe, this means the machine needs to recreate all the possible outcomes of a given configuration, i.e. play until a finished game, so that the said configuration can be evaluated.

[1]Milena Karova is with the Department of Computer Science and Technologie, Technical UniversityVarna, Bulgaria, E-mail: mkarova@ieee.bg.
[2]Lyubomir Gencheva student with the High School of Mathematics,Varna,Bulgaria,E-mail:lubo1993@gmail.com
[3]Lyubomir Vasilev a student with the Fourth Language School, Varna, Bulgaria, E-mail: lubodjwow@gmail.com
[4]Ivaylo Penev is with the Department of Computer Science and Technologie, Technical University Varna, Bulgaria, E-mail: ivailopenev@yahoo.com.

III. Implementation in Games (Tic Tac Toe and Chess)

A. Minimax Algorithm

Tic-tac-toe is a classic example of game which can be played by a computer using the Minimax algorithm. In this case, Minimax is an ideal solution because the branching factor of the game is only 9, as opposed to more complex games such as Chess, which has a branching factor of 35. The algorithm will work in absolutely the same way as in its general form [3].

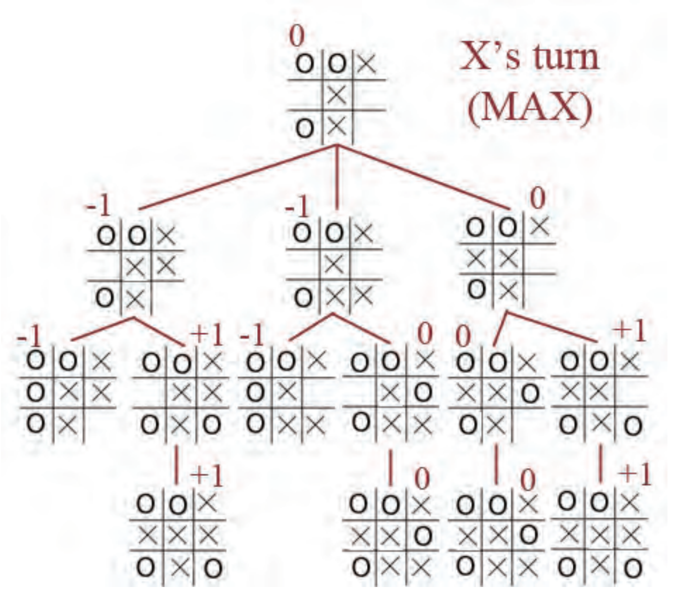

Fig. 2. Sample Tic Tac Toe Game Tree

Fig. 2 represents a sample Tic-tac-toe game tree. It is very similar to the tree in Fig. 1. This demonstrates that the Minimax theory and algorithm remain largely unchanged in their different uses.

```
Minimax_algorithm(player,board)
    if(game over in current board position)
        return player(the winner)
    child nodes = all legal moves for player from this board
    if(max's turn)
        return maximal score of calling Minimax on all the children
    else (min's turn)
        return minimal score of calling Minimax on all the children
```

Fig. 3. Pseudo code of the Minimax algorithm

Figure 3 shows the pseudo code implementation of the Minimax algorithm in Tic-tac-toe.

In the case of a game, what the algorithm does is recreate all the possible plies stemming from a configuration. The programme takes into consideration the opponent's best moves and the first player's best counter-attacks. This logic explains the Minimax algorithm and leads to the best moves for each player's turns. The most negative feature of this algorithm is that it requires great computational time in more complex games if it runs in full depth. Should the depth be limited, this will result in possible mistakes. This is the reason why an optimization is needed for a more efficient operation of the Minimax algorithm.

B. Alpha Beta pruning algorithm

The desired improvement turned out to be the Alpha-beta pruning algorithm. Alpha-beta pruning is a search algorithm that relies on the Minimax theorem but brings new light to the implementation of the theorem by minimizing the game tree [2]. The algorithm returns the same result as pure Minimax but in the best case it does it twice as fast. It literally prunes or cuts off some nodes that cannot lead to a better overall result (they are suboptimal). While simple Minimax will explore all possible nodes alpha-beta explores only those which seem to be better than the best move till now. This is a huge advantage when it comes to exploring game with big branching factors such as Chess. Chess has a big branching factor ≈ 35 compared to 9 in simple games like Tic-tac-toe. Using brute-force-like algorithms such as simple Minimax lead to search explosion in such conditions. This makes the application of simple Minimax rule in games like Chess not impossible but unprofitable as it requires immense computational power.

Table I
Worst and Best Scenario

Depth	*Worst scenario*	*Best scenario*
n	**b^n**	**$b^{n/2}+b^{n/2}-1$**
1	20	20
2	400	39
3	8 000	178
4	160 000	399
5	3 200 000	3576
6	64 000 000	15 999
7	1 280 000 000	71 553
8	25 600 000 000	319 999
9	512 000 000 000	715 540
10	10 240 000 000 000	6 399 999

TableI shows how much we can benefit from alpha-beta pruning in cases of big branching factor and good move-ordering. It shows the number of child nodes with depth **n** and branching factor **b**=20. Table1 is divided in two categories: worst scenario and best scenario. Here it should be mentioned that the alpha-beta algorithm highly depends on the move-ordering. Hence, for the minimizing player, sorting succesor's utility in an increasingorder is better. For the maximizing player, sorting successor's utility in a decreasing order is better. The maximal number of leaves is $\mathbf{b^n}$. In this worst case the program has to explore all the nodes in order to find the best one. After the best move has been found there are now nodes left to be pruned. This means that no pruning will be made, which proves to be the same as pure Minimax searching. However, if the move-ordering is good (best case) the number of leaves plummets as TableI shows. Slagle and Dixon first showed that the number of leaves visited by the alpha-beta search in this "best case" must be at least: $\mathbf{b^{n/2}+b^{n/2}-1}$. Since the best move has been found first there is no point in exploring all the remaining nodes. D. McIllro then proved that alpha-beta search for a random-generated game will be 33% faster than pure Minimax. That is why move-ordering is the focus of a lot of effort when writing an efficient program.

The algorithm calculates and keeps track of two variables: alpha and beta, one for each player. Alpha represents the value of the best possible move the current player has made so far. Beta, on the contrary, represents the value of the best possible move the opponent has made so far.

Figure 5 gives a clearer picture of how the algorithm actually operates:

№	Line
1.	AlphaBeta(player, α, β,depth)
2.	If(depth==0) return heuristic_evaluation();
3.	If(Maximizing player)
4.	{
5.	For each following child node
6.	Node_score= AlphaBeta(minplayer, α, β,child);
7.	If (node_score> α) α =node_score ; // *A better move has*
8.	*been found*
9.	If(α>= β)
10.	return alpha;//*Cut off*
11.	}
12.	Else If(Minimizing player)
13.	{
14.	For each following child node
15.	Node_score= AlphaBeta(max player, α, β,child);
16.	If(node_score<beta) β =node_score; // *A better move*
17.	*has been found*
18.	If(α>= β)
19.	return beta;// *Cut off*
20.	}

Fig. 5. Pseudo code Alpha-Beta pruning algorithm

C. A Genetic Minimax Algorithm

A Genetic Algorithm (GA) is most effective in situations, for which a well defined problem offers a compact encoding of all necessary solution parameters [1]. If this encoding grows two large and complex, the algorithm faces similar limitations of other local search methods and cannot be expected to find a global optimum. In considering the Tic Tac Toe strategy problem it is at first important to find a suitable representation and to ensure that a GA can be effectively applied.

The encoding of chromosome depends on game problem. The Fig. 4 presents Tic Tac Toe game tree encoding. Each gene is defined by the correspond move to be taken. The chromosome is a table with 827 genes to represent each game situation.

Fitness function is important to create an efficient GA and it is formed as way:

f(n)=possible win configurations for current player – possible win configurations for opponent player.

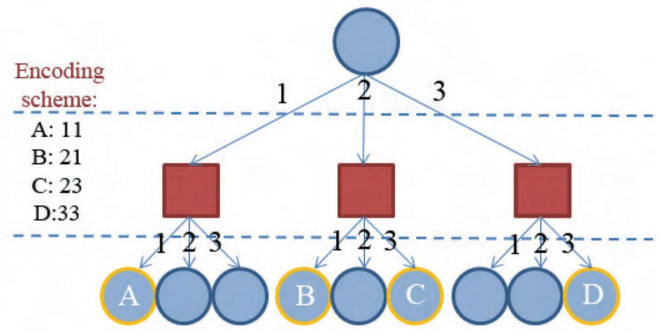

Fig. 4. Game tree encoding scheme

Based on its performance, each individual is assigned a fitness measure. The higher its measure, the more likely it is that an chromosome will take part in crossover and will be passed to the next generation. The parents can be choosed applying the genetic operator selection.

Once the parents are determined, the offspring is created by one-point crossover [Fig. 6]. The genes are copied from first parent at point of node 3 and the genes continue to be copied from the other parent. The crossover probability p_c is around 0,90.

The mutation [Fig. 7.] can occur at each gene of chromosome with probability p_m by a new random validate value is chosen to replace the current one. The p_m is very low value (less than 0,009) since it is evaluated for each gene independently. The essential goal of genetic operators is to ensure general variety in the reproductive process over time.

The size of population is 50 and the number of generations is 50.

The code for the implementation of Tic Tac Toe GA is a C++-based programming framework for GAs. Some modifications are made to accommodate the chromosome and the fitness function for the specific encoding and game tree solution.

IV. Experiments and Results

Each of those algorithms has experimented and Fig.8and Fig. 9give the results.

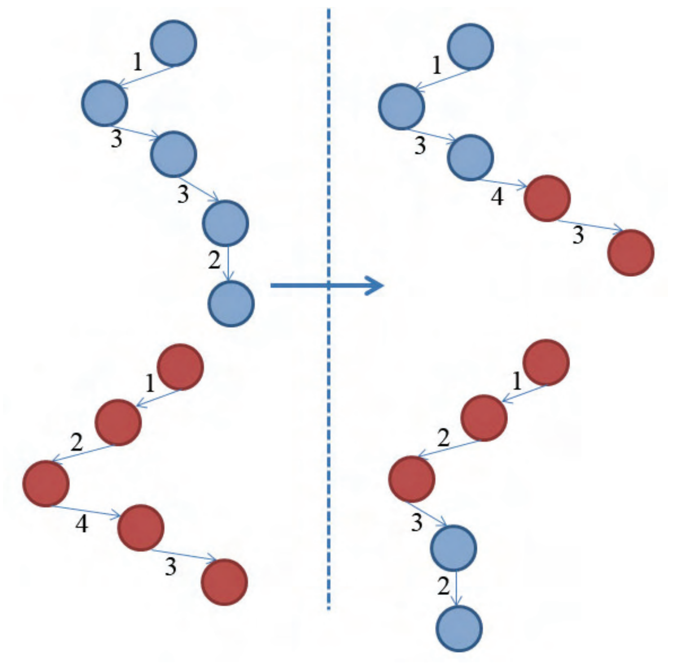

Fig. 6. Genetic operator One-point crossover

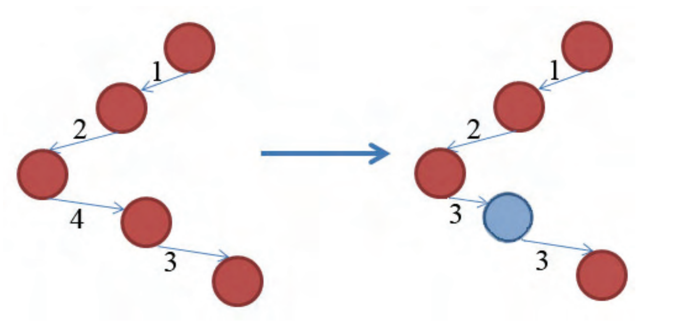

Fig. 7. Genetic operator mutation

To illustrate the efficiency of the different methods, the following charts are provided. The first chart [Fig. 8.] exhibits the relative time needed for the programme to calculate the best move for the Tic-tac-toe configuration, depicted in Fig. 9, using each of the three methods. The results show that there is little difference between the computational time for the three algorithms for this task. However, there is a sharp distinction in the second chart, which demonstrates the necessary time for calculating the best 11th move of an ongoing chess game. Our results show that the genetic algorithm is the best solution for two-player zero-sum games with a high branching factor.

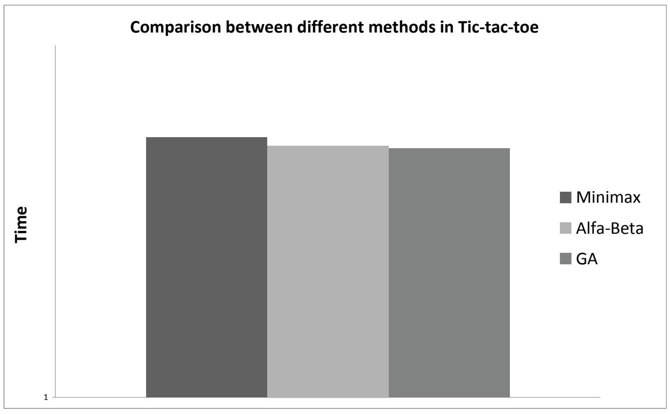

Fig. 8. Comparison between different methods in Tic Tac Toe

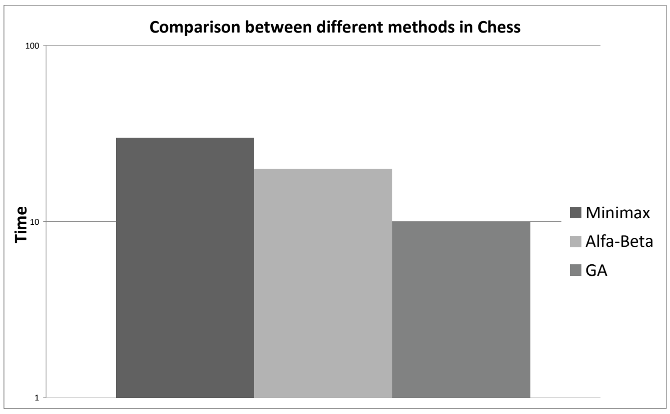

Fig. 9. Comparison between different methods in Chess

V. Conclusion

We design and implement a variety of techniques for solving Tic Tac Toe game and Chess. The GA could be applied successfully to evolve these game strategies.

The optimal result of genetic algorithm for different games is not guaranteed because it depends on the length of the chromosome and the depth of the tree decision.

Further testing would serve for running GA with different population parameters. In order to improve the fitness function, it can change the weights, given to the possible moves for each player. The number of paths or considering the next shortest path from a given position may make the fitness function more optimal heuristic function. It possible also to create hybrid algorithm Alpha Beta pruning with GA to find a good moves in two players games in a faster way.

References

[1] GA of Tic Tac Toe game: http://www.vclcomponents.com/s/0__/code_genetic_algorithm_for_tic_tac_toe/v.

[2] Russell S., Norvig P., “Artificial Intelligence A Modern Approach”, Third Edition, Prentice Hall, 2003, 2010.

[3] Schaefer S., “How Many games of Tic Tac Toe are there? http://www.mthrec.org/old/2002jan/solutions.html, 2002.