Appl Intell (2017) 47:752–768
DOI 10.1007/s10489-017-0918-z

A GA based method for search-space reduction of chess game-tree

Hootan Dehghani[1] · **Seyed Morteza Babamir**[1]

Published online: 21 April 2017

Abstract In this study, a GA (Genetic Algorithm) basesented to reduce the chess game tree space. GA is exploited in some studies and by chess engines in order to: 1) tune the weights of the chess evaluation function or 2) to solve particular problems in chess like finding mate in number of moves. Applying GA for reducing the search space of the chess game tree is a new idea being proposed in this study. A GA-based chess engine is designed and implemented where only the branches of the game tree produced by GA are traversed. Improvements in the basic GA to reduce the problem of GA tactic are evident here. To evaluate the efficiency of this new proposed chess engine, it is matched against an engine where the Alpha-Beta pruning and Min-Max algorithm are applied.

Keywords Chess game tree · Genetic algorithm · Alpha-Beta pruning · Min-Max algorithm

1 Introduction

Chess game tree consists of finite space, move choices, and limitation rules. For instance, the chessboard is made of 64 squares where each player has 20 choices to move at the beginning of the game, and the rule of 3 repetitions of the same move leads to termination of the game; consequently, allowing the search space of chess game to be presented as a tree where each one of the branches from root to leaf indicates a possible play, thus the name chess game tree.

In this context, since the state space and chess game tree size are very large, for our purposes, the entire tree for a solution may not be searched until an available deadline. There exists a great number of possible moves and plays (i.e. $O(10^{44})$) [1]. Given that D, the depth of tree, is a level of a game tree and B, the branching factor, is the number of children of each node, $N = B^D$ will become the number of nodes at level D [2]. In a chess game tree if we consider a node as a legal position, the average B is about 35 [2]; this means that by increasing the levels of the tree, the searchable states will be increased. However, D, usually does not increase over 40 because a typical chess game (average length of chess game) lasts for about 40 move pairs until a player wins or there is a tie [2]. This fact indicates that, there exist $80 = 40+40$ levels for a typical chess game where every node, known as the sink of a move, consists of 35 children on average, that is, $35^{80} \# 3.35 \times 10^{123}$ nodes (possible moves) in a typical chess game tree [2].

If a processor could visit 2 million nodes per second (i.e. speed of 2GH), it would take 5.3×10^{109} years to visit all nodes in a typical chess game tree. Even if a processor could visit 2 billion nodes per second, no considerable progress would be yield in search. Therefore, the best choice for any move in a chess game is not definite. A method to overcome such a great space is traversing the game tree to a limited depth. Based on the available sources, game tree pruning like the Alpha-Beta pruning [3]) and Min-Max search [3] are applied for this purpose. In order to search for an optimal node in order to choose a proper move, an *evaluation function* is applied in assigning a fitness value to every node instead of assigning the labels win, loss, or tie. The focus

✉ Seyed Morteza Babamir
babamir@kashanu.ac.ir

[1] Department of Computer, University of Kashan, Kashan, Iran

of most studies is on tuning coefficients in terms of the evaluation function; while in this study, a GA is proposed to shorten search branches in the chess game tree.

The early attempts in producing chess engines date back to 1940s [4]. In 1950, Shannon [1] proposed the two methods named type-A and type-B for developing chess engines where type-A traverses all branches in the game tree to a limited depth and type-B does some branches. The type-B could traverse appropriate branches to a deeper depth than type-A.

In 1950, chess engines were applied at elementary levels. This followed by applying game tree pruning techniques and more powerful hardware leading to development of a special purpose computer named Deep Blue in 1997. The Deep Blue was capable of evaluating 200 million positions per second and beat the chess champion of the time, Garry Kasparov [5].

The reason here in applying GA for the chess game tree search is its effectiveness over the Min-Max algorithm, applied in many studies conducted on chess game tree search. Promising results are obtained by applying GA in chess game tree search [6].

Authors in [7] compared the effectiveness of Min-Max with GA plus Min-Max through playing a non-chess two-player game (Fig. 1). The numbers on the horizontal axis in Fig. 1 indicate the levels of the game tree. Level 7, for instance, represents a game tree with a depth of 7. The vertical axis indicates the time needed for traversing the game tree at different levels, where 160,000, for instance, represents the time needed for traversing the tree as a whole.

When the time is limited for a game, the GA can present better solutions. Consider the distance between the best actual solutions and the solutions obtained through GA and Min-Max [7]. In this comparison, GA and Min-Max search a tree with the depth of 7 (see Fig. 2). The solutions obtained by GA have less distance from the best ones than Min-Max. Note that for a game tree with depth of 7, Max-Min cannot traverse and present the best solution during a bounded time.

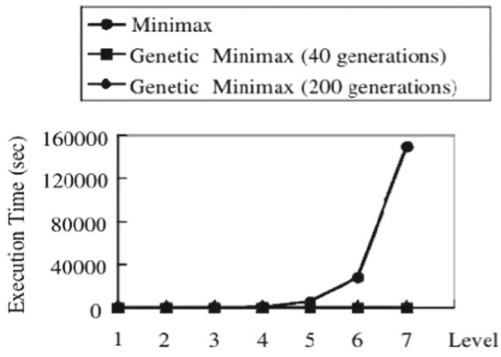

Fig. 1 Execution time of a non-chess two-player game through GA & Min-Max

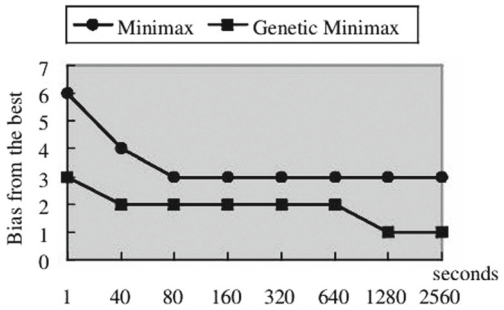

Fig. 2 Distance between the best actual solutions and the solutions obtained through GA and Min-Max and Min-Max

The GA has been and is being applied in tuning the terms coefficient of the chess evaluation function [8, 9] while its efficiency in searching chess game tree is not assessed.

The efficiency of GA in searching chess game tree is sought in this article. A GA-based algorithm is introduced to search the two-player game tree by [7], with a better performance in terms of accuracy and speed compared to Min-Max algorithm. This algorithm is not without its limitations: 1) the number of possible moves is the same in each node and 2) despite the height of the chess game tree, the length of each branch in the non-chess two-player game tree is the same. How to tune and apply the proposed fitness function in [7] for chess game tree is discussed in Section 4.2 of this article. If the modified algorithm is applied for producing a chess engine, the engine will be of type-B. In addition to removing the limitations of the former work in [7], attempt is made in their study to decrease the tactical problems developed due to lack of traversing all the branches.

The literature review is covered in Section 2; the preliminaries and basic concepts are explained in Section 3; the proposed algorithm is explained in Section 4; our chess engine is discussed in Section 5; the efficiency of our engine based on the proposed algorithm is analyzed in Section 6 and the article is concluded in Section 7.

2 Literature review

The only article where GA is applied to prune search game tree in non-chess two-player game tree is [7]. Authors in [7] applied the mutation operator in each population generation to improve the *material* features (i.e., value of the pawn, knight, bishop, rook, and queen pieces. (See Section 3.4 for the material feature).

In [10], 2 cases are considered for the chess game: *a material feature* and *two positional features* consisting of: a) *piece mobility* (the possibility of more moves for pieces of a player, resulting in more scores for the player, see Section 3.4 for the positional feature) and b) *individual*

pawns (the adjacent pawns that have non-same color; this is a disadvantage) and (c) *dynamic boundary strategy* for change of weights of these features.

By applying the DE algorithm [11–13], authors in [14] are concerned with tuning weights of the material features and the mobility of pieces. Among the latest studies, in addition to improvements made on coefficients of the material features in the evaluation function, many positional features of the evaluation function are considered as well [8, 9]. The parameters involved in search by adopting Alpha-Beta method and their improvements are explained in [10]. In all studies, for search in game tree, the Alpha-Beta algorithm and its improvements (like in main branch, pruning null moves, and iterative deepening) are applied while in this study the GA is applied for the same purpose.

In [15], authors applied Genetic Programming to produce a chess engine that plays a part of the chess game named Chess End Game. This engine, playing the chess End Games including king, rook, and queen, tied with the Crafty chess engine in special cases (http://www.craftychess.com/).

In [16], authors applied Genetic programming to solve a problem named mate in n moves. The results indicate that this algorithm solved the problems of mate in 4 moves with developing fewer nodes than the Crafty Engine. This mate in n moves was capable of solving most problems of the mate in 5 moves as well.

Inspired by the GA algorithm presented for the non-chess two-player games in [7], attempt is made here to search the chess game tree in playing a game in a complete set. Here, the algorithm proposed in [7] may not be applied directly, because there it is assumed that the number of moves of a play is fixed and predetermined while chess game tree contains a variable number of moves that are not predetermined. This means that branches (chromosomes) are of the same length as in [7] but vary in the chess game tree due to applying *quiescence* search (see Section 4.9) in leaf nodes. *Quiescence* search is an algorithm typically applied in evaluating Min-Max game trees in game-playing.

3 Preliminaries

Since the GA concepts, (i.e. chromosome, population and fitness) should be specified in applying GA to search the chess game tree they are defined in Table 1.

3.1 Reservation tree

The branches or chromosomes produced in each generation are stored in a Min-Max tree called the *reservation* tree. This tree is explained in Section 4.2 in detail. The first node of each branch is the root node and its last node is one which ends with a value (expressed by a circle). This value is calculated by the chess evaluation function.

3.2 Position

A position indicating an arrangement of pieces on the chess board (Fig. 3) is considered as a node in the chess game tree. Columns and rows are called *files* and *ranks* expressed by a to h and 1 to 8, respectively. Accordingly, a square is shown by notation Sn where, S$\in$ [a..h] and $n = 1..8$. For instance, notation b4 denotes the square in column 2 and row 4 of the board.

A position is named *quiet* (Fig. 4) if it does not consist of a move that leads to capture of a piece or a pawn *promotion*; this is when pawn reaches the last rank. Figure 4 shows a sample position indicating the white's turn but he/she cannot capture any piece or promote a pawn. Note the evaluation function is applied just to the *quiet* positions.

3.3 Castling

In chess game, castling is a move made by a king and a rook at once, and the only move where a piece can jump over another.

The abbreviations and values considered for pieces are tabulated in Table 2. The values obtained through genetic evaluation function are assigned to pieces [6].

Table 1 Definitions of chess game tree elements through GA elements

GA element	Chess game tree element
Chromosome	A branch of the game tree beginning with the root node and ending with a leaf node consisting of a value. The leaf node value is obtained by the chess evaluation function. Since the number of branches in chess game tree varies, the length of chromosomes is not fixed. This is because of the number and types of node moves being different in chess
Population	Set of game tree branches stored in the reservation tree
Fitness function	Fitness value of each branch is obtained through H-K+1 [7]. (See Section 4.2). Notations H and K are the height and level of the reservation tree, respectively
Crossover	Composition of 2 branches in producing a new branch
Mutation	A node change in a branch for producing a new branch

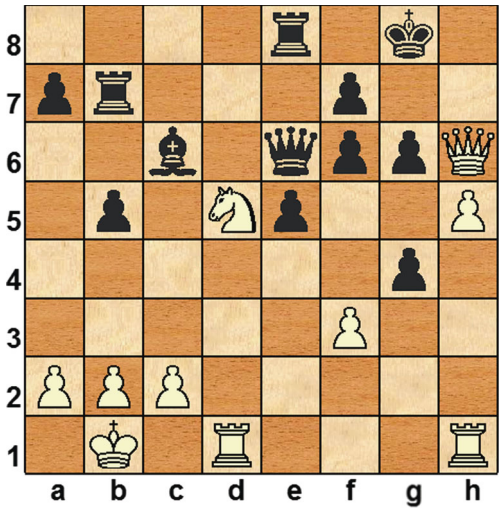

Fig. 3 A chess position

The formal set up for movements where a move represents a gene of a chromosome is tabulated in Table 2. For instance, if white moves his/her pawn from e2 to e4, it is shown as e4 and if black moves his/her knight from g8 to f6, it is shown as Nf6 (see Fig. 5). A move is shown as a string whose: (1) first character is the first letter of the piece name (except for pawn, which is denoted by "e") and (2) next characters indicate the position to which the piece should be moved. Character N in string Nf6, for instance, denotes move of the k***n***ight piece to position f6. Taking a piece from white or black is shown by notation ×.

Each gene of a chromosome (i.e. a move) is shown in 2 to 5 characters (see Table 3) according to the formal notation named Standard Algebraic Notation (SAN) [17]. As observed in the "Sample" column of Table 3, each piece type except pawn is shown by an uppercase letter (see

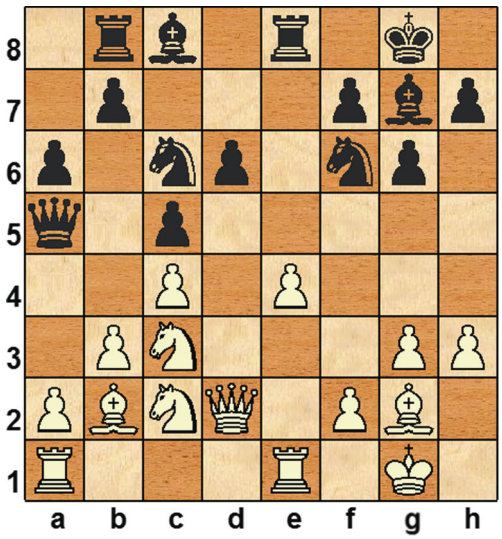

Fig. 4 A quiet position

Table 2 Piece specifications

L.	PN.	S.	V.
P	Pawn	♟	500
R	Rook	♜	500
N	Knight	♞	300
B	Bishop	♝	325
Q	Queen	♛	900
K	King	♚	–

L.:Letter, PN.:piece name S.: symbol, V.: value

Table 2). Since there exists three pairs of identical pieces (i.e. bishop, knight, and rook), a move is clarified by the letter of the moving piece, followed by source *file* (column) and the destination square. If knights are on g1 and d2 and move to f3, for instance, it is specified as Ngf3 or Ndf3.

3.4 Chess engine

The two main parts of each chess engine consist of: 1) evaluation function and 2) search algorithm. The *evaluation function* receives a position as input and returns an integer indicating the desirableness of the position. To evaluate a position, this function analyzes the *material* and *positional* features [10, 14]. The *material* feature represents the number and value of the pieces in a position and *positional* feature does advantages and disadvantages of a position. For instance, if rook is in an open *file* (current column), it is considered as a positional advantage. Note the evaluation function is applied in computing the evaluation of only the *quiet* positions.

While non-chess game trees apply the Transposition Table [18, 19], Iterative Deepening [19], Null Move Pruning [19], Principal Variation Search [19], and History Heuristic [19] methods for chess engine pruning, the Alpha-Beta

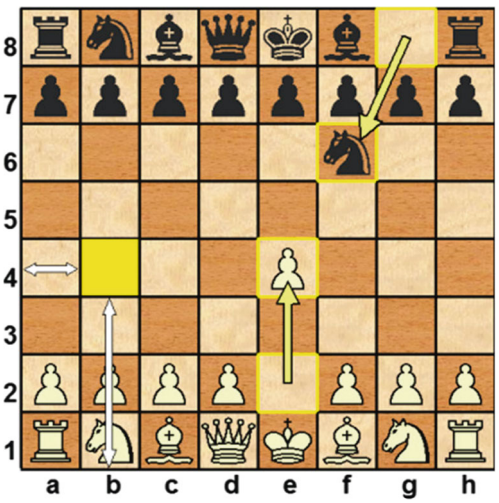

Fig. 5 White moves pawn to e4 and black does knight to f6

Table 3 Formal format of a chromosome gene (move)

	Format	Format explanation	Sample	Explanation
1	MxSn	M∈piece={P, R, N, B, Q, K}, x: captures, S∈places={a..h}, n:1..8	Bxb6	Bishop captures the piece on the b6 square.
2	exSn	S∈places={a..h}, n:1..8	exd5	Pawn captures the piece on the d5 square
2	MSn	M∈piece={P, R, N, B, Q, K}, S∈places={a..h}, n:1..8	Bd8	Moving bishop to the d8 square.
3	Sn	S∈places={a..h}, n:1..8	b5	Moving pawn to the b5 square.
4	S8=Q	S∈places={a..h}	c8 = Q	Pawn moves to the last rank of the 'c' column and is promoted
5	MmSn	M∈piece={R, N, B}, m∈{a,b},S∈places={a. h}, n:1..8,	Rad1	Moving the rook from column 'a' to the d1 square
6	O-O or O-O-O	O-O: kingside castling O-O-O: queenside castling	O-O-O	Castling to queenside

algorithm is applied in pruning the branches that do not need traversing. This is because the chess search space is the space of the Min-Max tree. To prune the branches and consequently decrease the search space in this proposed method, the GA is applied.

3.5 Communication protocol

UCI [20] and Winboard [21] are the 2 common protocols adopted as user's interface to communicate with chess engines. A communication protocol is responsible for sending and returning the moves from user to chess engine and vice versa, respectively. The Winboard protocol is applied in our proposed chess engine.

4 Proposed algorithm

To find an appropriate move against the adversary using our proposed GA, the following steps are taken:

1) Receiving a position from the chessboard as the input position through the Winboard user interface protocol,
2) Creating initial population of chromosomes (tree branches) and adding them to the reservation tree,
3) Computing fitness value of chromosomes in the reservation tree,
4) If time is running out, find and return the best chromosome of the reservation tree according to its fitness value; this chromosome denotes the first move,
5) If time is not running out, create new chromosomes (branches) by applying mutation and crossover operators and add them to the reservation tree (see Subsections 3 & 6 in this section) and go to step 3.

Note that this procedure is repeated per move. The algorithm will be executed 40 times.

4.1 Initial population

The first stage of our proposed GA is creation of an initial population of chromosomes. In the Min-Max tree, a branch is considered as a chromosome whose first node is the root of the game tree and last node is a value obtained through the evaluation function. The initial population of chromosomes is created as follows and is added to the reservation tree:

White starts the game, that is, the first level of the game tree where there exist 20 possible moves. These moves (chromosomes) are considered as the initial population whose fitness values are determined through the evaluation function and are added to the reservation tree.

For instance, if the white pawn is moved to e4 we have a *quiet* position (see Fig. 6); the corresponding chromosome for this move is shown in Fig. 7 and the

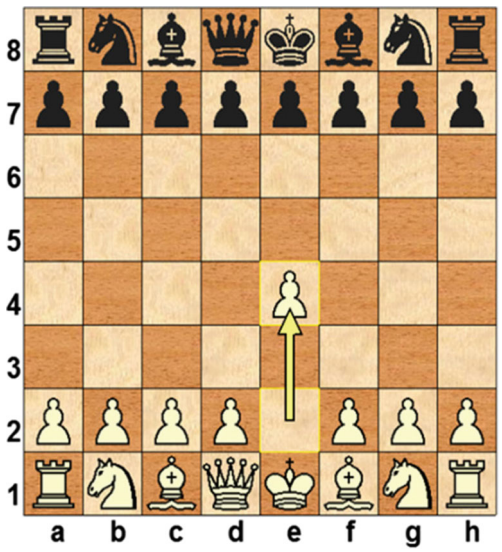

Fig. 6 Chess board position after move e4

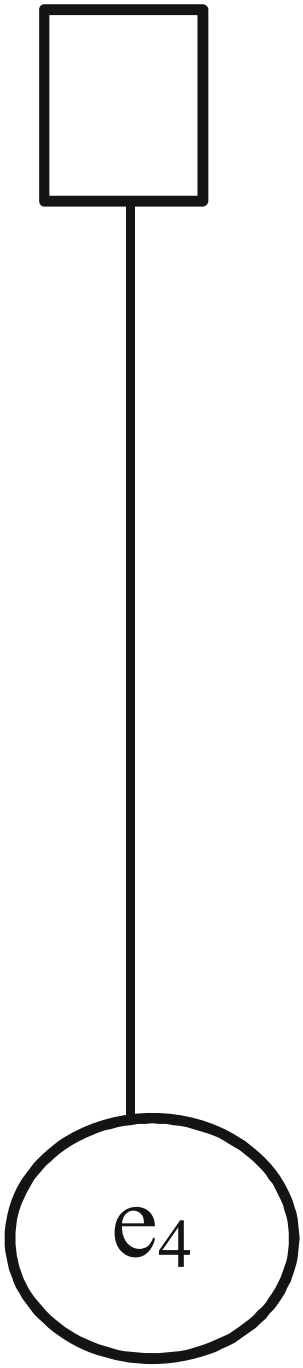

Fig. 7 The chromosome created from move e4

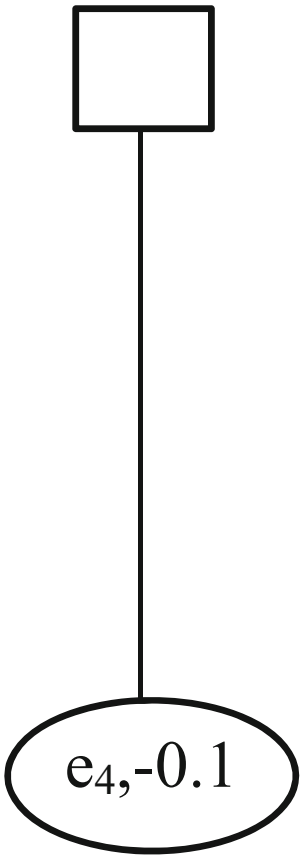

Fig. 8 The node evaluation value by evaluation function

updated chromosome with the *evaluation* value of the leaf node of the chromosome is shown in Fig. 8; then, this chromosome (branch) is added to the reservation tree.

4.2 Computing chromosome fitness through the reservation tree

After generating the initial population, mutation and crossover operations are run for generation of the next population from the fittest chromosomes; to be added to the reservation tree.

While the *fitness* function is applied to compute the fitness value of chromosomes (branch), the *evaluation* function is applied to compute the value of the nodes indicating a *quiet* position (see Section 3.2 for quiet positions).

As expressed in Table 1, while chromosomes (branches) are of the same length in Min-Max tree for non-chess two-player games like the one proposed in [9], they may not be same length in the chess game tree; this is due to the quiescence search in the leaves. Some of the produced branches that are to be added to the reservation tree are ignored because of their fitness values.

How the chromosomes are constructed from pieces' moves and how they are attached to the reservation tree is explained through the following example:

Consider chess in the initial position in Fig. 9. The chess position after the moves e4 and d5 are shown in Fig. 10. The corresponding chromosomes created from moves e4 and d5, the left and right chromosomes created from: (1) move e4 and (2) move e4 and then move d5, respectively (see Fig. 11). The right chromosome overlaps the left one; therefore, the reservation tree is represented as Fig. 12.

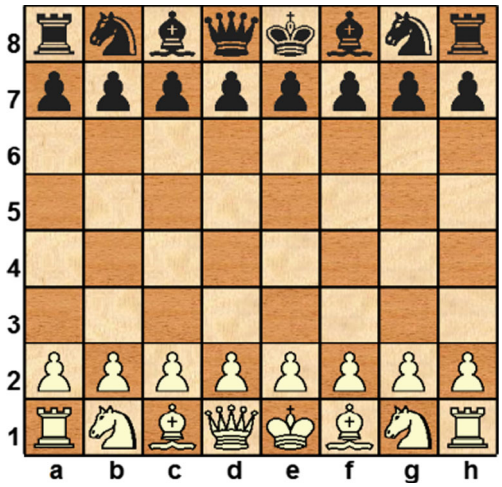

Fig. 9 Initial chess position

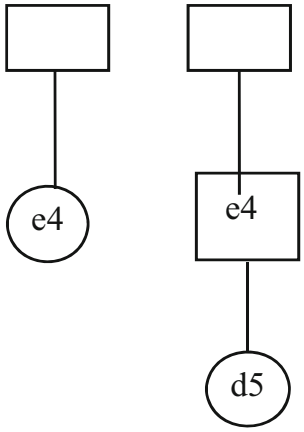

Fig. 11 Reservation tree after adding two chromosomes with overlap

Now, we explain the details of the reservation tree constructed through the proposed algorithm: given that an initial population has 5 chromosomes with fitness values; Fig. 13 shows the reservation tree for this population of chromosomes where the origin gene of each chromosome is the root node and the final gene of chromosomes are the nodes with values of −5, 35, 80, 90, and 8; as Fig. 13 shows, the chromosomes ended with values 35 and -5 with an overlap edge. The inner nodes of the Min-Max reservation tree are shown in Fig. 14.

The reservation tree in Fig. 13 is a Min-Max tree where nodes and edges are chess game positions and the possible moves in these positions, respectively. The input position to the engine (i.e. the root node) is a position where the best move (it may be any arbitrary position) is sought. Each bubble node shows the final node of a chromosome whose value is computed through the chess evaluation function. The final node of each chromosome is a bubble leaf node with a *value*.

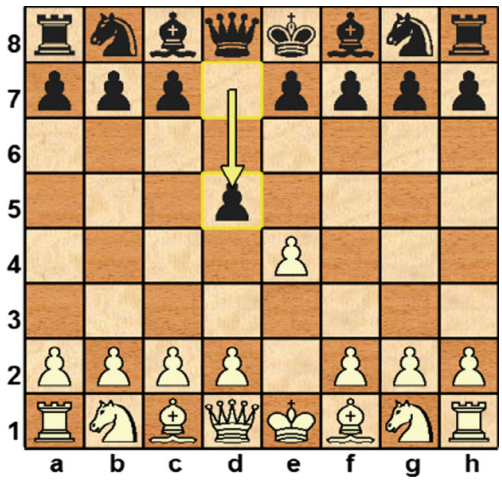

Fig. 10 Chess position after moves e4 and d5

Fig. 12 Position after moves e4 and d5

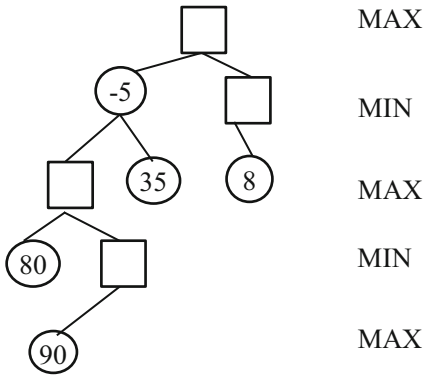

Fig. 13 Reservation tree with 5 branches

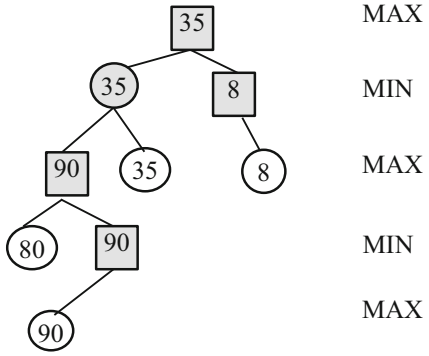

Fig. 15 Reservation tree (after applying regular Min-Max procedure) for Fig. 10

In order to add a branch (chromosome) to the reservation tree, a *quiet* position should be reached so that the evaluation function can compute the leaf node value. Positive and negative values in the reservation tree indicate dominancy of the white and black, respectively, which contain more absolute values.

In order to determine the fitness of 5 branches in Fig. 13, the values of all the tree nodes based on the Min-Max procedure should be obtained. Applying formula (H-K+1) [7] a value is assigned to each branch as fitness, where H denotes the height of the reservation tree and K denotes a level of the tree from where the value of a leaf node moves upwards. For example in Fig. 16, fitness value of the branch whose leaf node is 8 has value $5-0+1=6$, where $H=5$ denotes the height of the tree and $K=0$ does the level that leaf node 8 has reached.

If the reservation tree in Fig. 13 is traversed based on the regular Min-Max procedure, the 1st chromosome (the branch with value -5 as the last node) will be removed (see Fig. 15). The fitness value of branches (chromosomes) in this study is computed as follows:

In traversing the game tree through Min-Max algorithm, when a Max/Min node is reached, the maximum/minimum value of the node and its children as a new value is assigned to the node. For instance in traversing the tree in Fig. 14 using the Min-Max algorithm, when the node with value of -5 is reached, the minimum of values -5, 35 and 90, i.e. value -5 is assigned to the node (see Fig. 16). This assignment changes the result, that is, the root value in Fig. 16 is changed from 35 to 8. However, in the previous situation (Fig. 15) the root value was 35 and the node with value of -5 was removed.

4.3 Cross over

In every generation of population in GA, a number of chromosomes is added to the initial population (initial chromosomes) through the crossover and mutation operators. Figure 17 shows crossover where the breaking point of the chromosomes A & B should be determined. For instance, in Fig. 17 the third node in chromosomes A & B is chosen as the breaking point. The third node in chromosome A is created by the move e5 and in chromosome B a node is created from c5. Then, putting the sequence of moves Nc3-g6 of chromosome B at the end of the third node of chromosome A, a new chromosome named A' is obtained. Moreover, by putting the sequence of moves Nf3-Nc6 of chromosome A at the end of the third node of chromosome

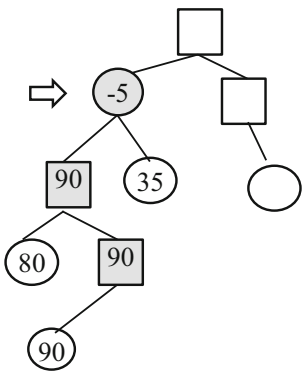

Fig. 14 Inner nodes in Min-Max tree

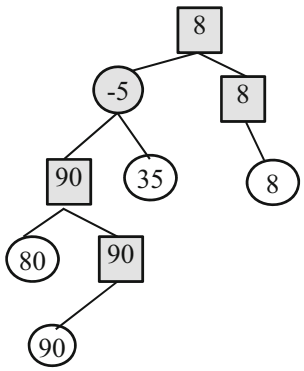

Fig. 16 Reservation tree (after applying the changed Min-Max procedure) for Fig. 10

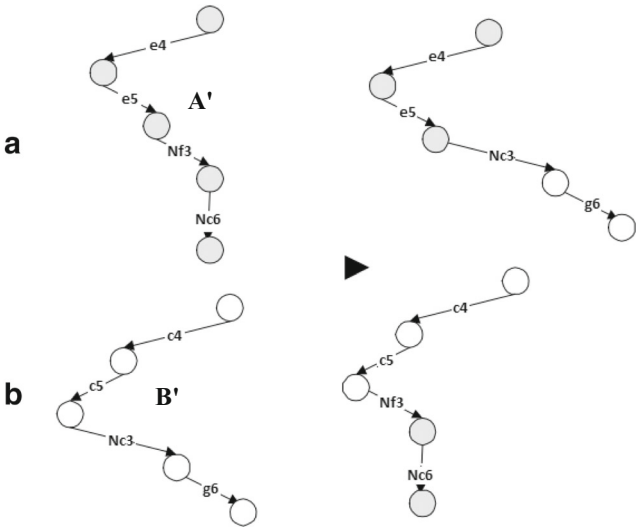

Fig. 17 Generation of branches/chromosomes A′ and B′ applying crossover on branches/chromosomes A and B

B, a new chromosome named B' is obtained. Such crossover may produce an invalid chromosome.

4.4 Move validation

It is not possible to have 2 consecutive nodes in a branch (chromosome) which belong to one player. For example, in chromosome A in Fig. 17, if the 3rd node is considered as the move made by white, the 4th node will represent the move made by black. This point should be observed in production of chromosomes during the crossover procedure; otherwise, it will lead to production of invalid chromosomes or branches. This process is controlled by a Boolean variable in each node of the branches.

4.5 Crossover validation

By combining two chromosomes (branches), the crossover operator creates two new chromosomes. In [12], at every position or node, there exist only a limited and predetermined number of moves or actions. However, in chess, due to different positions, moves will be different i.e., if a move is in a position, it may not be in another one. For instance, if move e4 is made by white at the beginning of the game (see Fig. 6), and black responds by move e5, in the rest of the game, white would not be able to perform e4 because the white's pawn has occupied this square in its previous move. This may in turn yield an invalid branch through the crossover operator during production of chromosomes. Consider the 3rd node of chromosome *A* in Fig. 17; if move Nc3 is not possible, an invalid chromosome is obtained. Validity analysis of the chromosomes in the proposed algorithm is run by adding a procedure; the produced

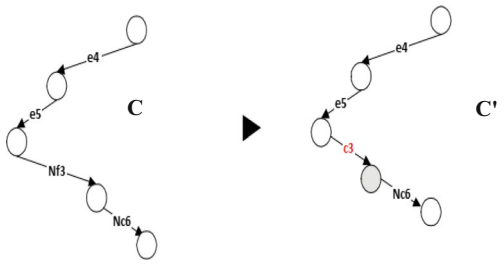

Fig. 18 Mutation through change of a node

chromosome of the crossover action is received as an input of the procedure which in turn would be put out of cycle and not be added to the reservation tree if one of its present moves is not possible.

4.6 Mutation

By changing a node in a branch of the tree through the mutation operator, a new branch is produced (Fig. 18). Branch C is a chromosome selected for production of the next generation. This branch is made of 4 moves where white makes move e4; in the second level, black does move e5; in the third level the white does move Nf3 and finally, black makes move Nc6 (Fig. 19). Branch C' is produced by changing move Nf3 to move c3 (Fig. 20). Finding this move is explained in Subsection 4.7.

4.7 Finding proper move for mutation

The first step in production of a chromosome through mutation operator is to find a node for mutation. This node could be selected in a random manner. Mutant C' (Fig. 18) is shown in Fig. 20. Moves in the chess game tree cannot be determined in advance; therefore, there is no solution except traversing a chromosome to reach the target node where the

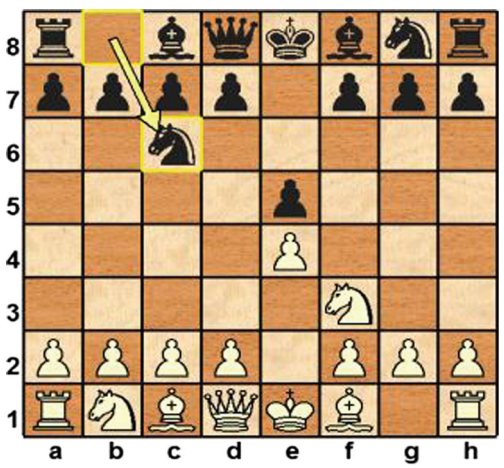

Fig. 19 Moves of e4 e5 Nf3 Nc6

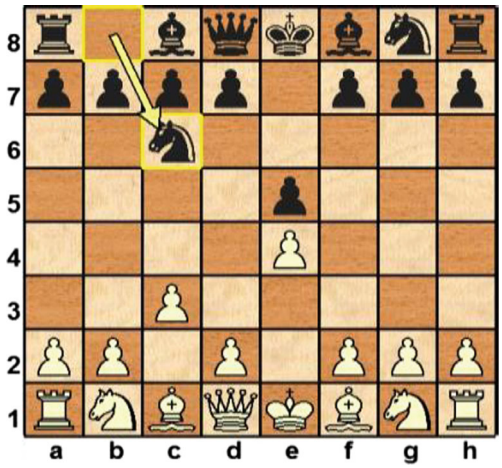

Fig. 20 Moves of e4 e5 c3 Nc6

change occurrence is sought. This means that if changing the node in level 4 of the chromosome indicated by Fig. 18 (the node resulted from the move Nf3) is sought, the saved position in the memory should be changed to the position of its previous node, i.e. the node in level 3 (the node created by move e5). This leads to identification of the moves which may be made in this position. Changing the saved position in the memory is carried out through the makeMove function [22]. The input of this function is the present move (the last position where change is sought). This function updates all the variables that save the position of the pieces in memory. If production of a chromosome like the one presented in Fig. 18 is in mind, this function must be applied twice to allow change of the present position (i.e., the node in level 3) once for move e4 and once for move e5 (Fig. 21).

4.8 Mutation validation

To apply mutation on the chromosome presented in Fig. 18, the branch is traversed up to reach the target node which

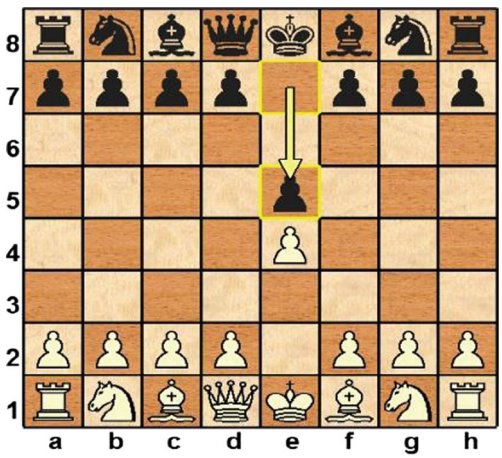

Fig. 21 Moves e4 and e5

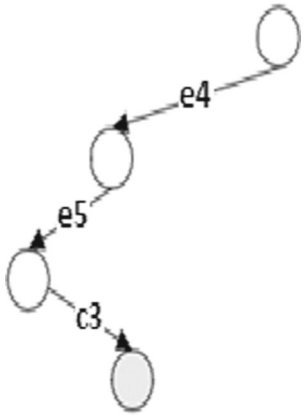

Fig. 22 The branch (chromosome) validation

should be changed. Here, the saved position is updated through the makeMove function. When the target node (level 4) is reached it is replaced by one of the available moves. To this point it is assured that the obtained branch is *valid*, because: 1) the only change made in the previous branch was in its last node and 2) in the last node, an available move was chosen and replaced by the previous move. Now, if this branch has other nodes, (i.e. the node in level 5) it should be checked whether they can be placed in the branch after the change. For instance, if move Nc6 in C' (Fig. 18) is not possible anymore, the branch shown in Fig. 22 is added to the reservation tree.

4.9 Quiescence search on the created branches

The advantage of the GA over Brute-force algorithms (Like Min-Max and Alpha-Beta) is the ability to search some of

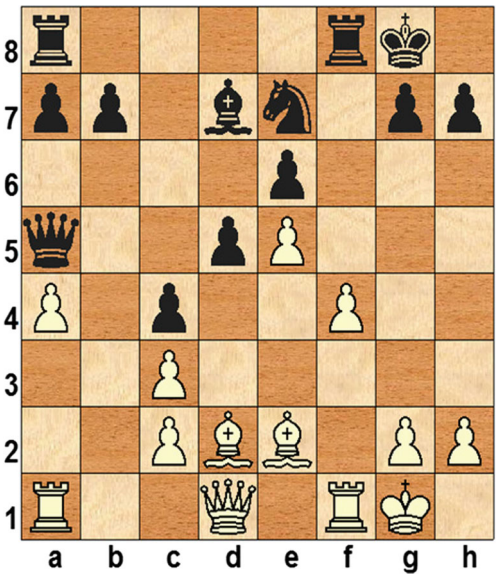

Fig. 23 Input position to the proposed algorithm

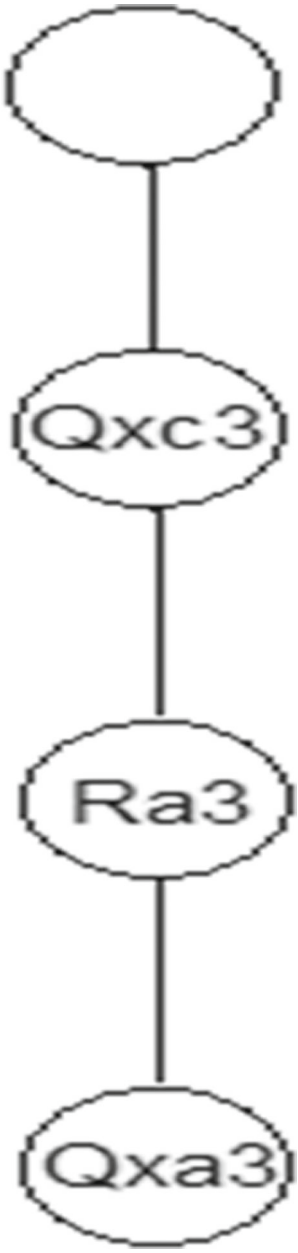

Fig. 24 A produced branch

the branches instead of all. In fact, chromosomes act as branches in the Min-Max algorithm and by a decrease in the game tree search space, GA seeks to find the best move. The selective search of branches, in comparison to searching all branches in the game tree, decreases the search time. It is notable that not searching all branches may cause a blunder (bad move). Applying quiescence search on the nodes of the produced chromosomes would prevent this newly proposed algorithm from introducing inappropriate chromosomes (those cause blunders) as the output to a great extent.

Consider the position in Fig. 23, the black turn. If this position is considered as an input of the proposed GA: 1) initial chromosomes are generated, 2) mutation and crossover operations are applied to the population members, 3) the generated chromosomes are added to the reservation tree, and 4) the fitted chromosome is selected through the reservation tree during a deadline. Now that the chromosome is found, the algorithm selects its first move. Consider the position shown in Fig. 23 where a chromosome like the one shown in Fig. 24 is produced and added to the reservation tree. In this figure, a branch is made of three moves: 1) it is the black's turn, which makes move Qxc3 (i.e. pawn in square c3 is captured by his/her Queen), 2) white makes move Ra3 (moves his rook from square a1 to a3) and 3) black captures the white rook through move Qxa3. Figure 25 shows the position after sequence of the moves in Fig. 24.

Move Qxc3 by black in the position shown in Fig. 23, will lead to losing his/her Queen because white can capture the black's Queen at the next move by the bishop in square d2 (here, white achieves a material advantage). In order to prevent the algorithm from selecting the branch shown in Fig. 24 as the final result, the fitness value assigned to this chromosome should be decreased and this is accomplished by adding another branch to the reservation tree. Another branch that brings out the fitness reduction is shown in Fig. 26. This branch is the outcome of two moves: 1) black makes move Qxc3 (a bad move, loss of his/her queen), and 2) white makes move Bxc3 and captures the black's queen by his/her bishop in square d2. The generation of this branch is finished with this node because the obtained position after move Bxc3 is a quiet position (Fig. 27). The last move by white (Bxc3) is presented by an arrow in Fig. 25.

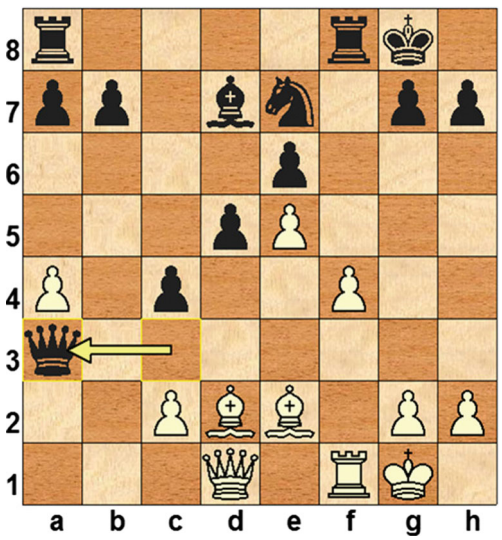

Fig. 25 Chess position after moves the chromosome shown in Fig. 23

Fig. 26 The branch should be added to the reservation tree

The branch shown in Fig. 26 can be obtained by applying the quiet search procedure to the node or the position which is created after move Qxc3 in the branch shown in Fig. 24 and then added to the reservation tree. The reservation tree created after branches in Figs. 24 and 26 are shown in Fig. 28.

Values −620 and 799 in Fig. 28 are obtained through the evaluation function for nodes Qxa3 and Bxc3, respectively. Before adding branches shown in Figs. 24 and 26 to the reservation tree, if the last node of each branch is in a quiet position, it is assigned a value (indicating a good or bad input position) using the evaluation function. The evaluation function should be applied to the quiet positions since it provides the best position estimation.

The evaluation function shows the advantages of black and white by negative and positive values, respectively, where the magnitude of a value indicates the amount of the advantage. Consider Fig. 28; the value 799 instead of value −629 is moved to the root if the branch shown by Fig. 26 is put beside the branch made by Fig. 24. The fitness value of these 2 branches is calculated through the formula H−k+1 [7] where H is the depth of the reservation tree and K is the rate of upward movement of a leaf. The fitness value of branches shown by Figs. 24 and 26 are $3-2+1=2$ and $3-0+1=4$, respectively; indicating that the assigned fitness values to branches become more real.

5 The engine implementation

To design a chess engine, four components are of concern: 1) two data structures to keep moves and piece positions, 2) a mechanism to move generator, 3) an evaluation function and 4) a search algorithm.

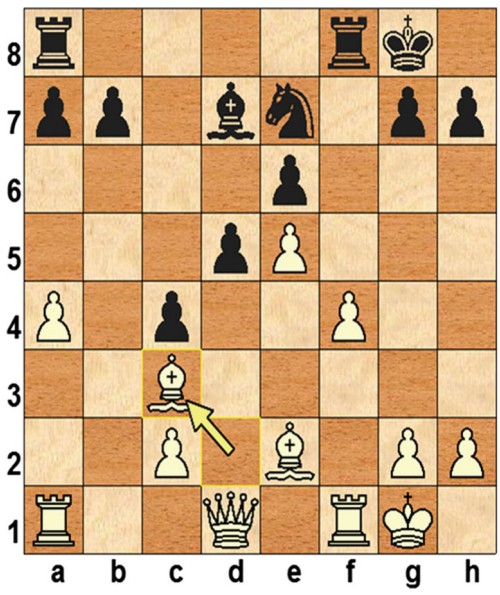

Fig. 27 Position after move Bxc3

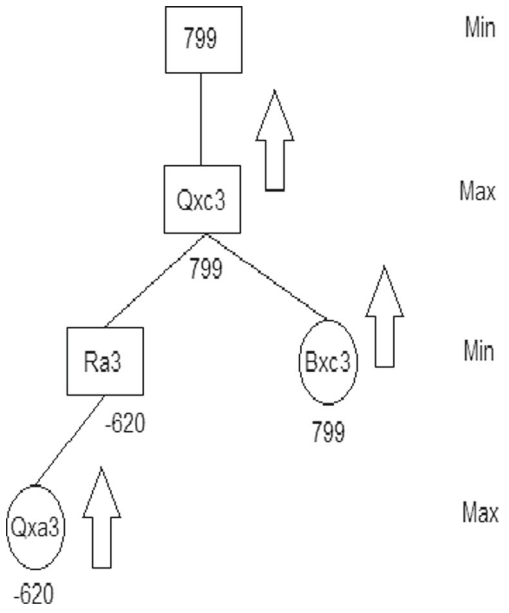

Fig. 28 Reservation tree after adding branches in Figs. 24 and 26

Component 2 should include 5 types of data: two values for source and destination of the move and 3 values for the moved, captured, and promoted pieces. To present these data, a 4-byte variable (32 bits) is used as follows: 1) six bits for the source and 6 bits for destination of a move because the chessboard has 64 squares, and 4 bits for each of the moved, captured and promoted pieces because black and withe have 16 pieces each, indicating that by using 4 bits, 16 black or 16 white pieces are addressed.

To keep a position, a bitboard data structure is applied as follows: black or white consists of 5 types of pieces (except King) each, totaling 10 types of pieces. Since chessboard has 64 squares, 10 64-bit variables are used to represent a square status. A position where a piece lies in *a4* is shown on the left side in Fig. 29. Assume this piece is a knight; the right side of Fig. 29 shows positions the knight can lay in.

The proposed GA and the data structure mentioned above are applied to set up a mechanism to generate moves. Another component for evaluation of a move in the chess game is an evaluation function. Since the chess problem has not already resolved completely, traversing the whole chess game tree and selecting the best move is a big challenge, indicating that there exists no known evaluation function to compute the best fitness value for the moves. If we assume black as the adversary in the chess game, the positive/negative values returned by the evaluation function indicate the dominancy of white or black.

The *material* and *positional* features are considered for computation of the evaluation function. The value of Queen, for instance, is more than Rook (see Table 2). A *Material* value is computed using relation $\mathrm{MB} = \Sigma(N_{wi} - N_{bi})V_i$ where i is the type of piece (Pawn, Knight, Bishop, Rook, and Queen), N_{wi} and N_{bi} are the number of white and black pieces of type i, respectively and V_i is the value of the piece with type i. Considering the assignment of the values stated in Table 2 to pieces, the MB value is computed and interpreted as follows:

MB > 0, white is dominant in both the number and values of pieces,
MB < 0, black is dominant, in both the number and values of pieces

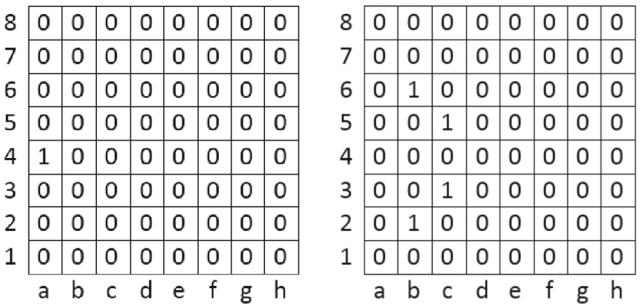

Fig. 29 Data structure for storing move of pieces

MB $= 0$, black and white are in the same situation in both the number and values of pieces.

For instance, if white has a queen more than black, then MB $= (2 - 1) \times 900 = 900$.

As stated in Section 3.3, the positional feature has advantages and weaknesses. To compute a positional value, 5 factors are involved: 1) the number of moves that white or black already made, 2) the positions of the pieces on the chessboard, 3) the possible moves to be made, 4) safe king (the player that lies in Castle will have more scores) and 5) arrangement of pawns (a player has a disadvantage if its pawns are in consecutive squares). According to [6] and the factors stated above, we exploited the evaluation function applied in engine Winglet [23] to compute the material and positional features.

Table 4 Verifying blunders of our proposed engine using Stockfish reference

#	Pos. (input)	GA move & value	Stockfish 5
1	1 B	Bxa4	Bxa4
2	21 W	Qd3(0.63)	Rdh1(1.02)
3	41 W	Qd3(0.67)	a4(0.73)
4	61 B	Bc5(0.13)	Rc7(-0.93)
5	70 B	O-O-O(0.84)	Qa6(0.7)
6	82 B	b4(-0.6)	e5(-1.04)
7	91 B	Qa6(-0.54)	Qd4(-1.38)
8	97 W	Qg3(0.97)	f3(0.98)
9	105 B	Rd8(0.84)	Nc7(0.08)
10	115 W	Rad1(0.76)	Qf4(2.22)
11	127 B	Rac8(0.08)	Qc7(-0.09)
12	137 W	Be7(-1.23)*	Bf2(0.74)
13	149 B	Kh8(-0.1)	Bxf4(-0.73)
14	155 W	Kg1(0)	Bxc6(0.77)
15	163 W	Qb5(0.3)	f4(0.98)
16	175 B	Rae8(0.20)	b5(-0.07)
17	193 W	Kg1(-0.15)	Ra1(-0.12)
18	209 W	Rfe1(0.25)	f3(0.56)
19	219 B	Qf6(0.32)	f5(0.1)
20	229 W	d4	d4(0.97)
21	247 W	Qe4(0.91)	Qf4(0.99)
22	267 W	Rc1(-10.51)*	Kh1(3.29)
23	287 W	Bxb7(-2.97)*	Bg4(2.95)
24	307 B	Bf5(2.01)*	Qd8(0)
25	315 W	Qd3(0.77)	Qb3(0.93)
26	325 B	Bf5(0.29)	Nd7(-0.09)
27	335 W	Bg2(0.68)	Bg2(0.68)
28	351 W	Rhe1(0.23)	e5(1.58)
29	367 B	Rd8(0.73)	Bc6(0.34)

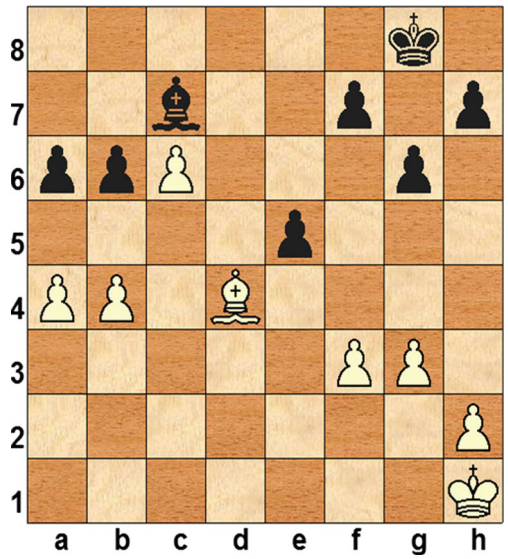

Fig. 30 The extracted position

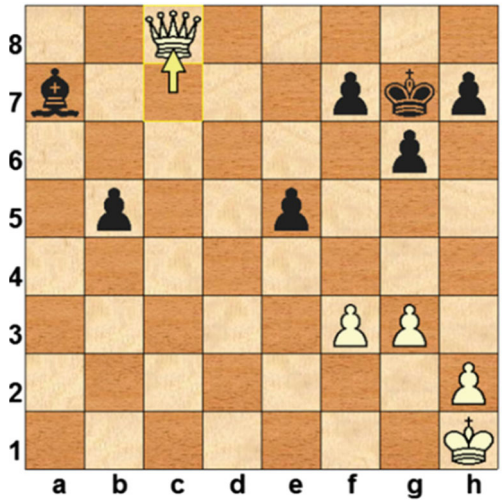

Fig. 31 The position after move Bxb6 by white, with a material advantage

The final component needed in designing a chess engine is a search algorithm, where GA was used as an evolutionary search algorithm in the current study.

6 Evaluation of results

6.1 Considering blunders and tactics

The Stockfish chess engine [24] is a professional engine applied in this study as a reference to detect blunders (bad moves) of our proposed engine. To this end, 29 positions are selected from [25] as the inputs in both the engines (Table 4). Letters *B* or *W* in positions represent the black or white pieces. The values in columns GA and Stockfish are obtained through Stockfish evaluation function. According to [26], if the difference between values obtained for a position from both the engines is more than 1.5, our proposed engine suffers a blunder. A move made by our proposed engine is a blunder if its value differs from the corresponding value of the move done by the Stockfish engine. According to Table 4, our proposed engine suffers 4 blunders in 29 moves (indicated by *). These blunders are made since the total of the branches are not created and added to the reservation tree.

A game position was extracted from the database of the tactical problems (BT2450) proposed by Bednorz and Toenneissen (http://www.schach-computer.info/wiki/index.php/BT-2450) (Fig. 30). In this position, the turn is white's; the best possible move is Bxb6 (white captures pawn b6 by his bishop) because after the sequence of the moves in Table 5, white will have a won position (white has obtained huge material advantage, see Fig. 31) and black has no way to escape from this branch.

Tactic Consider the sequence of 7 moves (Table 5), which is a branch with a length of 13 (7 moves by white and 6 moves by black, see Fig. 32). Here, the problem is that finding a good move (move Bxb6 by white in the position shown in Fig. 31) demands adding a particular branch

Table 5 Sequence of 7 moves

Move#	Move symbol	Explanation
1	Bxb6 Bxb6	White takes b6, black takes b6
2	a5 Bd8	White moves his pawn to a5, black moves his bishop to d8
3	b5 axb5	White moves his pawn to b5, black takes b5 pawn
4	a6 Bb6	White moves his pawn to a6, black moves his bishop to b6
5	a7 Bxa7	White moves his pawn to a7, black takes white's pawn on a7 with bishop
6	c7 Kg7	White moves his pawn to c7, black moves his king to g7
7	c8 = Q	Pawn moves to the last rank of the 'c' column and is promoted

1.Bxb6 Bxb6 2.a5 Bd8 3.b5 axb5 4.a6 Bb6 5.a7 Bxa7 6.c7 Kg7 7.c8 = Q

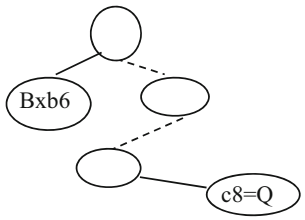

Fig. 32 A scheme of the branch containing move sequences indicated by Table 4

(branch move 13 in Table 4) to the reservation tree. Such problem is named tactics [20, 27]. In such situations, the player who carries out such a sequence of moves (moves in Table 5), will gain a significant advantage even if the other player makes the best moves (in the position shown in Fig. 32, black could not prevent promoting the white's pawn to queen).

6.2 Evaluation of the proposed Search algorithm with Min-Max

To evaluate the proposed GA in practice, we created 2 engines of: 1) our proposed GA and 2) the Min-Max algorithm and they are applied to make 20 game matches. Both engines applied the same evaluation function. The scores for 4 levels of game each indicating the game tree depth are shown in Fig. 33. According to Fig. 33, when the game tree is small with depth of not more than 3 levels, the Min-Max algorithm obtains more scores than its opponent (the GA); this is because the Min-Max covers all the branches, but in a game tree with depth of more than 4 levels, our proposed GA obtains more scores. When depth of a game tree increases, the Brute-Force algorithms undergo the high run-time because they traverse all nodes of the game tree, while under some circumstances, the proposed GA traverses the lengthy branches in less run-time.

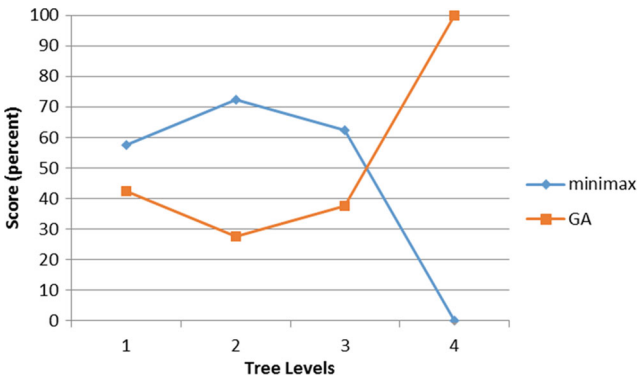

Fig. 33 Scores of the two chess engines applying: 1) GA and 2) Min-Max algorithm

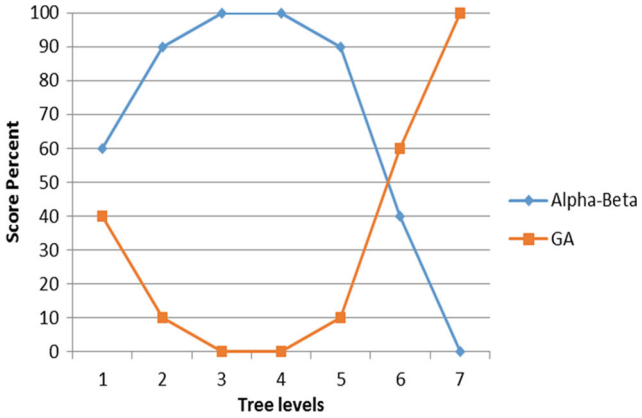

Fig. 34 Scores of chess engines of GA-based engine and Alpha-Beta based pruning engine

6.3 Evaluation of proposed search algorithm with Alpha-Beta

The GA-based engine played against the Alpha-Beta algorithm based engine with the *principal variation* search (PVS) and null move pruning. The PVS is a variant of the Min-Max search algorithm capable of achieving faster than Alpha-Beta pruning, where the zero-sum property of a two-player game and relation $\max(a, b) = -\min(-a, -b)$ is applied. (i.e., the value of a position to player 1 is the negation of the same to player 2).

Both engines applied the same evaluation function. When the game tree has a depth of more than 6 levels, GA-based engine outperforms Alpha-Beta and receives more scores (Fig. 34).

6.4 Evaluation of results through the Ufim engine

The Ufim engine is designed by Niyaz Khasanov [28]; the GA based engine and Ufim engine played in 8 levels (levels

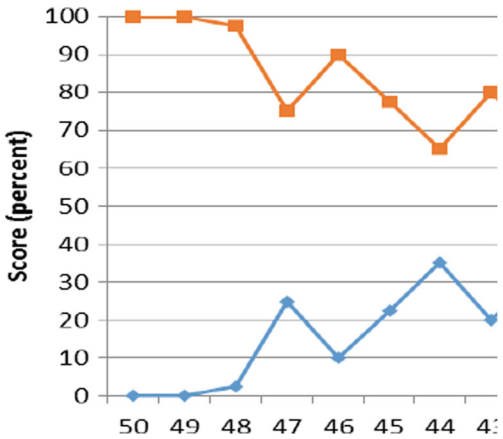

Fig. 35 Score percentages of the GA based chess engine and Ufim engine

Table 6 Scores of both improved GA-based engine (2level) and the basic GA-based engine (1level) proposed in this study

	Win	Loss	Tie	Score-percent	Score
GA-Level 1	4	13	3	27.5%	5.5/20
GA-Level 2	13	4	3	72.5%	14.5/20

43 to 50), a total of 100 min (20 games of 5 min each). In each level, the game had a different initial position and was played twice: 1) it was white and (2) it was black. The scores are assigned according to [17] with the winner as one and the loser as zero points. A score of half point is assigned when they are tied. The score percentages obtained by each engine are shown in Fig. 35. The red and blue lines show scores of our and Ufim's engine, respectively.

6.5 Improving the proposed algorithm

At the initialization phase of the proposed GA algorithm, all possible moves of the input position (in one level) were added to the reservation tree as the chromosomes. This does not cover a considerable search space for two levels and improves the search efficiency of the proposed algorithm. An example of a two-level branch, obtained from two moves, is exhibited in Fig. 26.

To improve the efficiency of the proposed algorithm, instead of considering all possible moves in one level, they were considered in two levels and the chess engine was modified. Both the initial and modified engines matched for 20 times with time limitation of 5 min per game; the results, where 72.5% of the scores belong to the proposed modified engine, are shown in Table 6.

7 Conclusions

In this study, a basic GA (one that has already been applied to non-chess game tree) tree is tailored and extended for the chess game. The GA-based engine played with 2 engines where Min-Max and Alpha-Beta algorithms are applied for searching and pruning. It is observed that when the search space becomes more than a specific amount, the GA-based engine outperforms the other 2 engines. This proposed GA was improved for generating initial chromosomes by adding branches to the reservation tree up to 2 levels.

When a game tree was small with depth of not more than 3 levels, the Min-Max algorithm obtains more scores than the GA-based; this is because the Min-Max covers all the branches, but when tree has depth of more than 4 levels, the GA-based obtained more scores. When depth of a game tree increases, the Brute-Force algorithms suffer the high run-time because they traverse all nodes of the tree; however, under some circumstances, the proposed GA traversed the lengthy branches in less run-time.

The blunder and tactic problems were considered in the proposed GA-based engine. Although problems are few here, further studies are necessary to overcome them. A professional chess engine called the Stockfish is applied in this study as a reference to detect blunders (bad moves) of our proposed engine. The GA-based engine suffers 4 blunders in 29 moves. The blunders are made since the total of the branches are not created and added to the reservation tree.

References

1. Shannon CE (1950) XXII. Programming a computer for playing chess. Philos Mag 41:256–275
2. Mandziuk J (2010) Knowledge-free and learning-based methods in intelligent game playing. Springer, Berlin
3. Tim Jones M (2008) Artificial intelligence: a systems approach. Jone & Bartlett Learning Publication, ISBN-13: 9780763773373
4. Bowden BV (1953) Faster than thought. In: A symposium on digital computing machines. Pitman Publishing
5. Hsu F-H (1999) IBM's deep blue chess grandmaster chips. IEEE Micro 19:70–81
6. Dehghani H, Babamir SM (2015) Effectiveness analysis of genetic algorithm for chess game tree search. In: The 8th international conference of iranian operations research society, pp 251–253
7. Hong T-P, Huang K-Y, Lin W-Y (2001) Adversarial search by evolutionary computation. Evol Comput 9:371–385
8. David O, van den Herik J, Koppel M, Netanyahu N (2014) Genetic algorithms for evolving computer chess programs. In: IEEE transactions on evolutionary computation
9. Vázquez-Fernández E, Coello CAC, Troncoso FDS (2013) An evolutionary algorithm with a history mechanism for tuning a chess evaluation function. Appl Soft Comput 13:3234–3247
10. Nasreddine H, Poh HS, Kendall G (2006) Using an evolutionary algorithm for the tuning of a chess evaluation function based on a dynamic boundary strategy. In: IEEE conference on cybernetics and intelligent systems, pp 1–6
11. Price K, Storn R (1997) Differential evolution–a simple evolution strategy for fast optimization. Dr. Dobb's J 22:18–24
12. Price K, Storn RM, Lampinen JA (2006) Differential evolution: a practical approach to global optimization: Springer Science & Business Media
13. Ronkkonen J, Kukkonen S, Price KV (2005) Real-parameter optimization with differential evolution. In: IEEE congress on evolutionary computation. Edinburgh, pp 506–513
14. Boskovic B, Greiner S, Brest J, Zumer V (2006) A differential evolution for the tuning of a chess evaluation function. In: IEEE congress on evolutionary computation. Vancouver, pp 1851–1856
15. Hauptman A (2005) GP-EndChess: using genetic programming to evolve chess endgame players. In: The 8th European conference on genetic programming. Switzerland, pp 120–131
16. Hauptman A, Sipper M (2007) Evolution of an efficient search algorithm for the mate-in-N problem in chess. In: The 10th European conference on genetic programming, pp 78–89
17. Laws of Chess. Available: http://www.fide.com/component/handbook/?id=124&view=article, Access Date: 29-7-2016
18. Veness J, Bair A (2007) Effective use of transposition table in stochastic game tree search. In: IEEE symposium on computational intelligence and games, pp 112–116

19. Millington I, Funge J (2009) Artificial intelligence for games, 2nd edn. Morgan Kaufmann, Elsevier
20. http://chessprogramming.wikispaces/UCI, Access Date: 20-2-2016
21. http://hgm.nubati.net, Access Date: 20-2-2016
22. Skiena SS (2009) The algorithm design manual. Springer, London
23. http://chessprogramming.wikispaces/Winglet, Access Date: 20-2-2016
24. https://stockfishchess.org, Access Date: 20-2-2015
25. Hellsten J (2010) Mastering chess strategy. Everyman Chess
26. Chabris CF, Hearst ES (2003) Visualization, pattern recognition, and forward search: effects of playing speed and sight of the position on grandmaster chess errors. Cogn Sci 27:637–648
27. Wilson F, Alberston B (1999) 303 tricky chess tactics. Cardoza Publishing
28. http://chessprogramming.wikispaces.com/Ufim, Access Date: 29-7-2016

Hootan Dehgani received BSc degree in Software Engineering from Islamic Azad University of Najafabad, Najafabad, Iran and MSc degree in Software Engineering from University of Kashan, Kashan, Iran. He is a professional chess player and a member of Iran Chees Federation. He authored 2 conference papers on evaluation of the GA effectiveness for chess game tree search and analysis of the mutation testing.

Seyed Morteza Babamir received BS degree in Software Engineering from Ferdowsi University of Meshhad and MS and PhD degrees in Software Engineering from Tarbiat Modares University in 2002 and 2007 respectively. He was a researcher at Iran Aircraft Industries, in Tehran, Iran, from 1987 to 1993, head of Computer Center in University of Kashan, Kashan, Iran, from 1997 to 1999 and haed of Computer Engineering Department in University of Kashan from 2002 to 2005. Since 2007, he has been an associate professor of Department of Computer Engineering in University of Kashan, Kashan, Iran. He authored one book in Software Testing, 4 book chapters, 20 journal papers and more than 50 international and internal conference papers (http://ce.kashanu.ac.ir/babamir/Publication.htm). He is managing director of Soft Computing Journal published by supporting University of Kashan, Kashan, Iran.