Politechnika Wrocławska

Wydział Elektroniki

Metaheuristic Chess Artificial Intelligence

Maciej Borkowski
195968@student.pwr.edu.pl

Paweł Pałus
197004@student.pwr.edu.pl

Mariusz Waszczyński
192660@student.pwr.edu.pl

Abstract

This paper reflects on metaheuristics algorithms (genetic algorithm and simulated annealing) implemented for playing chess for standard and arbitrary initial arrangement of the chessboard. It is a new idea especially the possibility of starting the game from arbitrary initial arrangement of the chessboard which allows to consider certain close to the end of the game situations. external artificial intelligence is used for lerning of implemented algorithms. New software experimentation system is developed and described with explaining of its features. Finally the results are presented and summarised with conclusions and new ideas for research extension. The implemented algorithms seem to achieve not satisfactory results.

Index Terms: *chess; metaheuristics; artificial intelligence; genetic; simulated annealing*

1 Introduction

In nowadays metaheuristics are increasingly used in different aspects. They base on nature. Many problems are tried solve by those procedures which unfortunately can not give the optimal global solution but can find the good one. That is the cost, hence computing is not highly important. The memory should be considered. The "human DNA contains about 3 billion nucleotides constituting genes, regulatory sequences and other non-coding regions all residing in a one-dimensional sequence that is organized in 3-dimensional space" [(http://www.biomedcentral.com/content/pdf/1742-4682-7-3.pdf)]. This is the huge memory in which can be stored any informations.

Main goal is to use yet another Artificial Intelligence. There are significant number of IT systems try to appear intelligent one way to another. For example decision trees, neural networks, fuzzy logic. Each of them has different benefits. Therefore the aim is to extend the database of known solutions to have more potentials. There are many fields where metaheuristics can be applied. One of them where intelligence has major factor are multiplayer games. Today appeard e-sport, which many players contend together. Learning with artifical intelligence which imitates a real player could be very helpful.

In the paper game of chess is considered. The game is popular and its rules are known for many people. There are complex and therefore can be applied many strategies. The competence of predicting next few moves are also here used. The humans are using several methods. They are basing on their experience, strategies and predicting skills. The most important here is training. Each chess figure has well-defined moves it is able to take, which results in an astoundingly big number of possible scenarios. A new player quickly finds out that some pieces are more important than others and subconsciously assign to them different weights. It is an important property, which is generally taken into consideration when designing an artificial chess player. [1]

From the perspective of one of the players a real chess game is not deterministic. Nevertheless, a metaheuristic could be used in this situation to learn from its own miastakes when playing a game of chess, especially when the rival always uses the same strategy (which is the case when playing with a normal chess AI). The main goal of this research is to check how would genetic, ant colony search and simulated annealing algorithms behave when used as an artificial intelligence in chess game.

The paper is organised as follows: Section 2 contains the description of considered problem and its mathematical model. In Section 3 the experimentation system being GUI is introduced and described. Section 4 covers description and results for Genetic Algorithm and Simulated Annealing and also ideas for their enhancements, separately for each algorithm. Section 5 contains ideas for improvements in application. Section 6 presents conclusions and possibilities for further work

2 Optimization Problem

The problem describes a standard game of chess, with a square board of 64 fields. Two players have to consecutively move a piece the board onto another field according to complex, well-defined rules. Our task is to find the series of movements in a game of chess that gives the best chance of winning the game in the end. The starting position of pieces can be arbitrary.

2.1 Mathematical model

indices

- $t \in N$ - initial game turn

- $n \in N$ - maximum number of game turns intended for playing
- $i, j \in \{1, 2, 3, 4, 5, 6, 7, 8\}$ - number of rows and columns

variables

- $B_{8,8,t}$ - fields on the board taking values $v_{i,j,t} \in R$ describing the field on board (e.g. the chessmen standing on field) during turn t
- f$(B_{8,8,t}, v_{i,j,t})$ - takes negatives values for bad situations on the field $v_{i,j,t}$ (e.g. an opponent's piece) positive for good situations (e.g. player's piece)

objective

Find a series of movements starting from $B_{8,8,t}$ and ending in $B_{8,8,t+n}$ according to rules of chess

$$\sum_{i,j} f(B_{8,8,t+n}, v_{i,j,t+n}) \quad is \quad maximised \tag{1}$$

3 Experimentation system

3.1 Chess engine

For learning purposes a real, competitive artificial intelligence was required. A choice has been made to use an external chess engine, which provides required functionality. Such engines commonly implement the Universal Chess Interface (UCI) [5] for two-way communication between outside application (like the one made for this research) and the engine implementing an artificial intelligence. In the end we have chosen to use Firenzina Engine [6] because of its good implementation of UCI standard and being able to make good moves in a very short amount of time (one milisecond).

From the side of our application the UCI interface had to be implemented, as well as the whole set of chess rules so that human player can know which moves are valid (through graphical cues). Metaheuristics also require this knowledge, because they have to start from making random moves chosen from the set of valid ones. A Graphical User Interface (GUI) was also fully implemented for user's convinience.

3.2 GUI

Figure 1 shows run panel that is displayed after starting the application. On the right side of the application the user can see options for white and black player. Checking proper radio button one can choose what type of player will play a game as white player and also as a black player. The choices are: AI (Firenzina engine), human, or one of 3 metaheuristics. There are also File buttons that are used to input knowledge base for ant search algorithm and genetic algorithm (proper file button for proper white or black player), and a Next button that is used to make movements one after another from a textbox that is located in annealing tab. It will be further discussed in a later section. The Next button is not displayed in another figures.

Figure 2 shows the run panel after pressing the run button. Initial locations of chessmen are set according to chess rules. In the bottom right corner of the board there is a square coloured black or white depending on which player is expected to make the next move.

The application allows the possibility to start a game from any position other than the standard initial locations

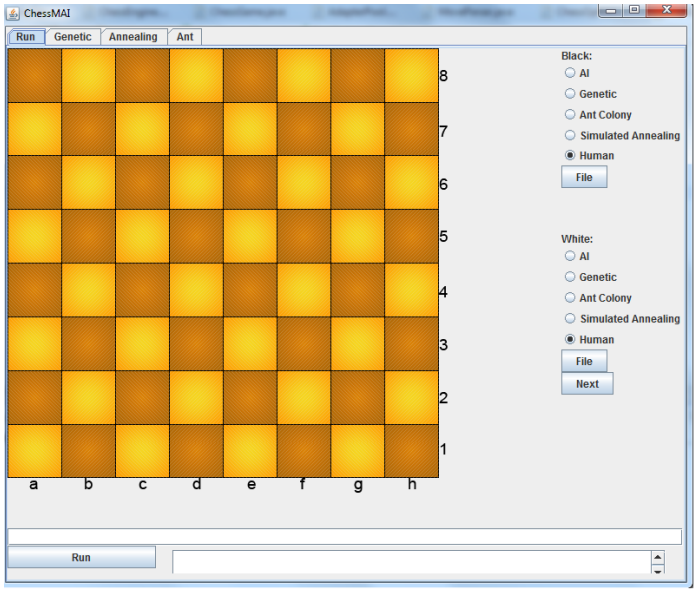

Figure 1: Initial, empty run panel enables us to choose the types of players and starting game situation

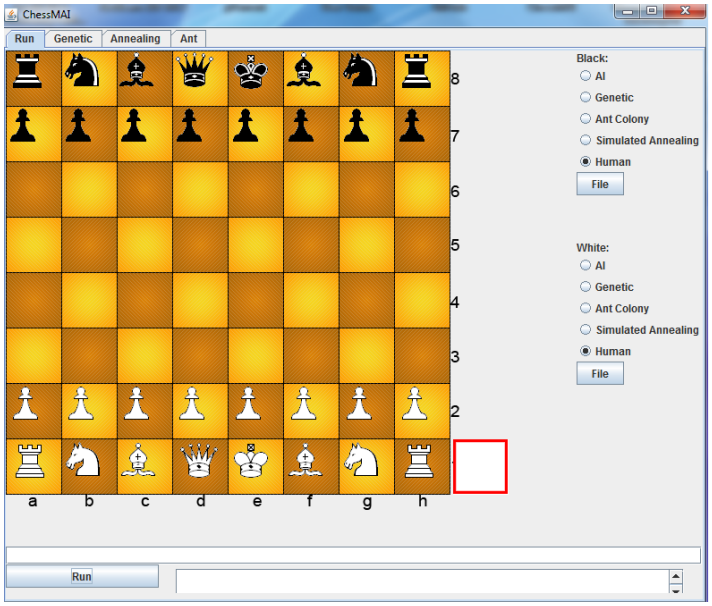

Figure 2: Initial Run panel after pressing run button to start the game

of chessmen what is depicted in figure 3. For this purpose sequence of valid movements should be typed (or pasted) to the textbox located above the run button.

On the right side from the run button one can see the textbox responsible for showing the history of the movements as depicted in figure 4. This is the full history of movements including movements that was made using the textbox above the run button and its primary use is to copy it into a move history field to always start the game from this position.

At the top of the application are tabs for metaheuristics, where the learning phase can be started, which will be described in according sections later.

Figure 3: Initial board after started with a certain move history

Figure 4: History of movements

4 Algorithms

4.1 Ant colony

4.1.1 Idea

For ant colony search algorithm a mapping is created between a placement of chess pieces on a chessboard and a list of possible moves the current player is able to do, when provided such board. Each move on this list is additionally annotated with a real value, which describes the fitness of the move. Moves with higher fitness ought to yield us better results. Such mapping is called a pheromone and a set of them pheromones (Figure 5).

Ant is defined here as a chess player, that uses pheromones to choose a move when it is metaheuristic's time to make a choice of movement. Ant can be a part of a colony, in which case the colony provides the pheromones or it can be independent (used for Greedy Mode). Pheromones can be stored to and loaded from a file. When an ant finds itself on a board that has not yet been added to pheromones a new pheromone is created (possible moves for the board are computed and assigned equal real values).

Ant can work in one of two modes:

- Adventurous Mode
  Used for learning. In this case the ant works for the betterment of its colony. It chooses moves randomly, according to the values of pheromones. This strategy improves the pheromones, by visiting a wide range of possible boards, which results in frequent updates and addition of new pheromones. The probability of choosing move M (with real value v):

$$P(M) = \frac{v_m + |min(V)|}{\sum V + n|(min(V))|} \tag{2}$$

  where V are all values in a given pheromone, v_m the value for move and n is the length of V.

- Greedy Mode
  Used for testing and real games. In this case the ant plays for the best end result in its game. Ant chooses a move from the pheromone with the highest value to choose the best move in each turn.

The process of learning consists of many iterations of ants in Adventurous Mode working as a colony. Each iteration amounts to a few phases:

1. Start new games and wait for them to end
   Each ant plays one game of chess and remembers all boards it has run across, all moves it has chosen to do and the the cost function of the series of movements (value of cost function for last board).

2. Update pheromones
   For each ant the pheromones connected to visited boards are updated by a fraction of the value of cost function of the whole series of movements.

$$v_{new} = v_{old} + \frac{i}{m} cost \tag{3}$$

   where v_{new} is the new value, v_{old} is the old value, i is the index of the movement in this series of movements, m is the length of the series of movements and *cost* ist the value of cost function.

3. Dissipate pheromones
   Pheromone for each of the boards that has been visited at least once by any ant in this game is decreased by multiplying the value by a parameter.

$$v_{new} = v_{old} * (1 - dissipation) \tag{4}$$

   where v_{new} is the new value, v_{old} is the old value and *dissipation* is a parameter.

Pheromones can be stored to a file. The file consists of a list of pheromones, which can be used when playing in the Run panel. Each is described with two lines:

1. String representation of a board, left to right, bottom to up, where # means no chessman, upper case letters mean white chessmen and lower case letters mean black chessmen

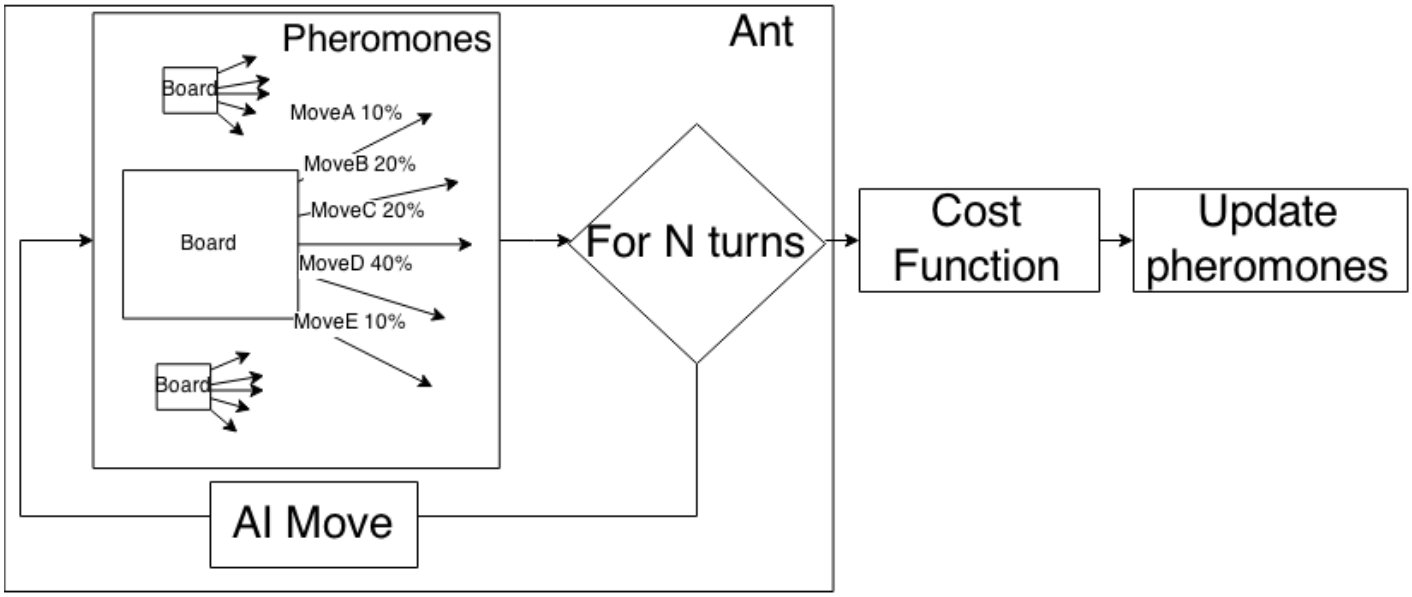

Figure 5: Diagram of the implemented ant colony search algorithm with visualisation of pheromones for one iteration with one ant

2. A list of moves. Each move is described by five integer values. First two are the coordinates of chessman to move, third and fourth where to move the chessman to, the fifth is a special value used for promotion (when a pawn becomes another chess piece) and the sixth a real value of pheromone describing its effectiveness.

4.1.2 Experiments

Firstly, it has to be accentuated that chess is a complex game, the number of possible boards that have to be remembered in pheromones is huge and for each ant move another move has to be done by external Artificial Intelligence. Because of these reasons the learning phase of this metaheuristic takes a very long time. To get results appearing significantly different from completely random ones, hours have to be spent on learning. This makes it very difficult and time-consuming to experiment with parameters properly.

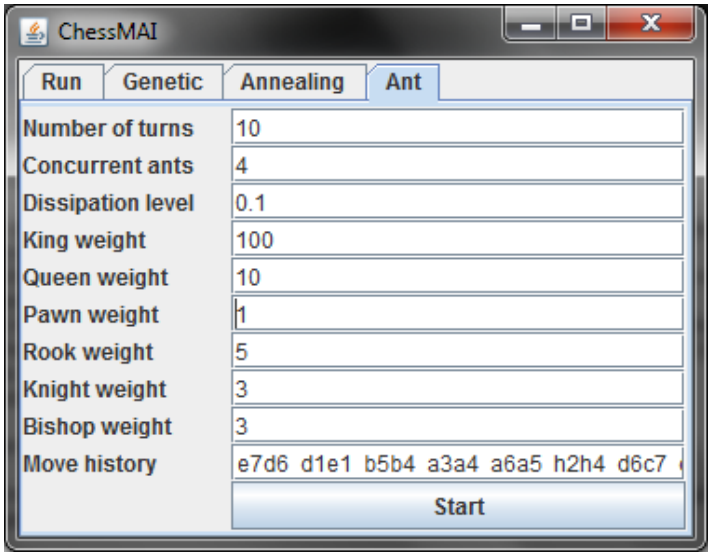

Figure 6: Ant colony dialog boxes

For the experiments the parametrization of many ant search specific variables has been put into dialog boxes in the application(Figure 6). This way user can change these values easily and create his own colonies. The user-available parameters are as follows:

- Number of turns
  The number of turns for each iteration. Each iteration consists of an all concurrent ants playing one game of chance up to a win, lose, draw or the artificial end of game, when it takes too long. This parameter should be kept low if we want the learning phase to take less time and if we want ants to have more broad "knowledge" of possible moves in the beginning of the game.
- Concurrent ants
  The number of ants playing a game in each iteration. The more of them the longer each iteration takes.
- Dissipation level
  The dissipation parameter of ant search algorithm dissipation equation (Equation 4).
- Piece weight
  Each piece has its own weight used in cost function (Equation 1)
- Move history
  This parameter sets the starting position of the game from any point, provided a valid chess move history. Each ant in each iteration starts its game from this point and plays up until the end of the game (including the end of maximum number of turns defined as another parameter)

The more obvious rules had to be applied to get to the point of metaheuristic being better than a random algorithm and able to win one game out of hundreds when playing against the artificial intelligence. Most importantly the weight of king should far bigger than other figures, the number of turns small, and a move history provided that gives a possibility of winning in a few turns. Increasing the number

of concurrent ants and, at the same time, the dissipation level makes the results possibly even better but by a very small margin at the cost of a longer learning phase (making it difficult to test).

4.1.3 Result

Meddling in all of these parameters proved to be insufficient in obtaining better results. After careful debugging of the application, the conclusion has been made, that the possible reason for improvement could be to increase the probability of choosing good moves instead of dwell on the bad moves. Even though the move choosing equation (Equation 2) gives more probability to good moves, it is not very significant to the sum of all the probabilities, because there are generally a lot of bad moves. One of the ideas was to sum up the logarithms of values in equation, which is often a way of dealing with those kind of issues. This however only made the values of pheromones more close to each other so the probability of choosing a good move didn't really change that much. The really important change was observed when the Equation 2 was modified to one with a "tolerance" factor multiplied by each value except the best one. This way the probability of choosing the best move in training is big, enabling the colony to thoroughly establish how good of a situation on the board it results in. This can quite easily culminate in a fall into a local maximum of cost function, but it is not a bad situation to be in chess - at least we end up in a better situation than before. Additionally to really fall into a local maximum the moves have to contain a lot of weight gain, so it would be most probably a checkmate anyway, which is, for all intents and purposes, a global maximum. Indeed, if a winning sequence of moves has been established in this version the colony started winning very often. At this point a file containing the pheromones could be exported and played in Greedy Mode to win games.

4.2 Genetic algorithm

4.2.1 Idea

Our second proposal of a metaheuristic algortihm is the genetic algortihm. It is purely related to how living beings nature evolves through the ages. Looking at it from the perspective of one life beaing equal to one chess game, it may give interesting results. The population consist of many individuals living through their lifes, which in this case are end up being games. The individual is a set of gens which here are called moves. From a programming point of view a mapping is created between a placement of chess pieces on a chessboard and a move for the current player, when provided such board. Each individual is additionally annotated with a real value, which describes the fitness of the final move, calculated with help of cost function (Equation 1).

In the application each metaheuristic algorithm works as a set of players. Even Artificial Inteligence is marked as player. Hence different players can play against each other. Genetic player is divided into two modes (similar to the ant colony algorithm's):

- Adventurous Mode
  Used for learning. In this case the individual is created. The moves are chosen randomly.
- Greedy Mode
  Used for testing and real games. Based on global individual choosing appropriate moves. If the individual does not recognize current placement of chess pieces on a chessboard, the move is picked randomly.

Entire proccess is divided into two phases: generating a population and learning. Genereting population phase creates a new thread for playing individuals, waits for them to end and updates their fitness. When the first phase has been finished the learning process which consist of many interations is started.

The iteration has follwing steps:

1. Selection
   The general assume is 80 percent of population have wrong gens and are going to remove (Figure 8).

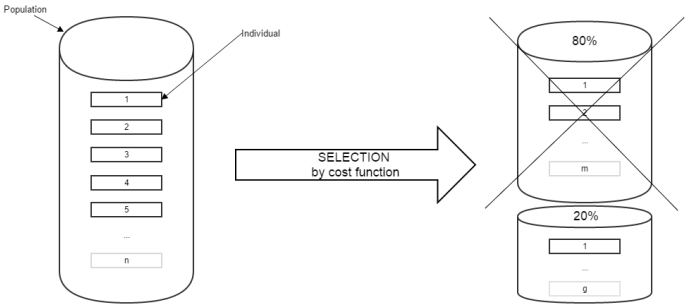

Figure 8: Genetic - selection phase

2. Choosing the global individual
   From the remaining population the individual which has the best cost function is compared with global individual. If it is higher, then global individual is overwritten (Figure 9).

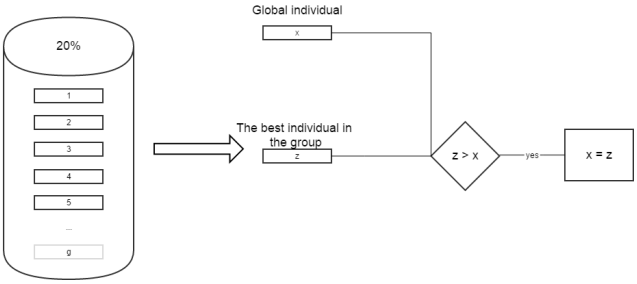

Figure 9: Genetic - comparing with the global individual

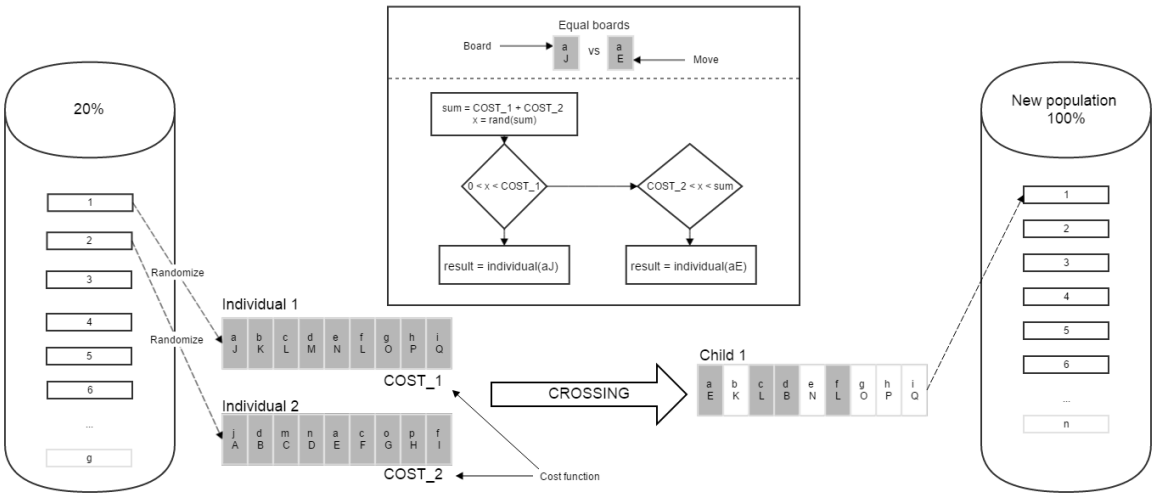

Figure 7: Genetic - crossing phase

3. Crossing phase
   Two individuals are chosen randomly. Every board[1] from first individual is comparing with every board from second individual. If the boards are equal the proper one is chosen and added to the descendant. Cost function is used to provide probabilities, by which a good candidate is chosen. Cost function from first and second individual is summed. The sum is the boundary for a random number. If the random value is from zero to the value of first individual cost function then first individual is chosen, otherwise second individual. Nevertheless if the board has not found an equivalent, it is added to the descendant. Process is showed on Figure 7. This step is repeating up to fill population, in order to avoid a decrease in population size.

When user decides to stop learning, global individual is stored to file which can be used in normal game - greedy mode. Optimal time for 5 moves for learning is couple hours.

4.2.2 Experiment

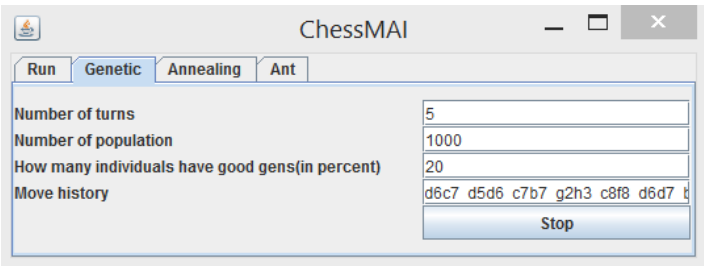

Figure 10: Genetic tab in application

In order to play with genetic algorithm, firstly the optimal global individual should be finded. The genetic tab is designed for learning purposes. It contains four text fields.

Number of turns
Determine how many moves the individual have to do. Low value should be used to confine time for learning.

Number of population
Determine size of population.

How many individuals have good gens
Needed for selection phase. Defined 20 percent by default.

Move history
Learning proccess can start from specified board.

Button "start" implies generating population and turning learning proccess on. Optimal time for learning depends on parameters provided by user. Significant parameters are "Number of turns" and "How many individuals have good gens". First one can minimalize time of learning to achive certain point of knowledge. Second one has direct impact of removing individuals and is associated with "Number of population" parameter.

4.2.3 Result

Unfortunetelly checkmate was not achived. For almost 2 thousand tries Artificial Inteligence was winning. The importance of initial board should be emphasized. If cost function is very high at the beginning then is the most probability that genetic algorithm will win. Of course there is a significant impact of the way how cost function is calculated. What factors are takien into consideration, what weight are associated with pieces.

4.3 Simulated Annealing

4.3.1 Idea

Simulated annealing as any other metaheuristics algorithm is used when search space is too large to use exact methods. The algorithm seeks to find an optimal solution (but not necessarily finds it) unlike many metaheuristics algorithms it allows with certain probability for the adoption of worse solution what makes the algorithm resistant to getting cought in local minimum solutions

The algorithm consideres typically optimization task which is every game it plays, it is not intended to learn and then play with other player. The algorithm plays his first game choosing random moves. At the end of the first game all

[1] a placement of chess pieces on a chessboard

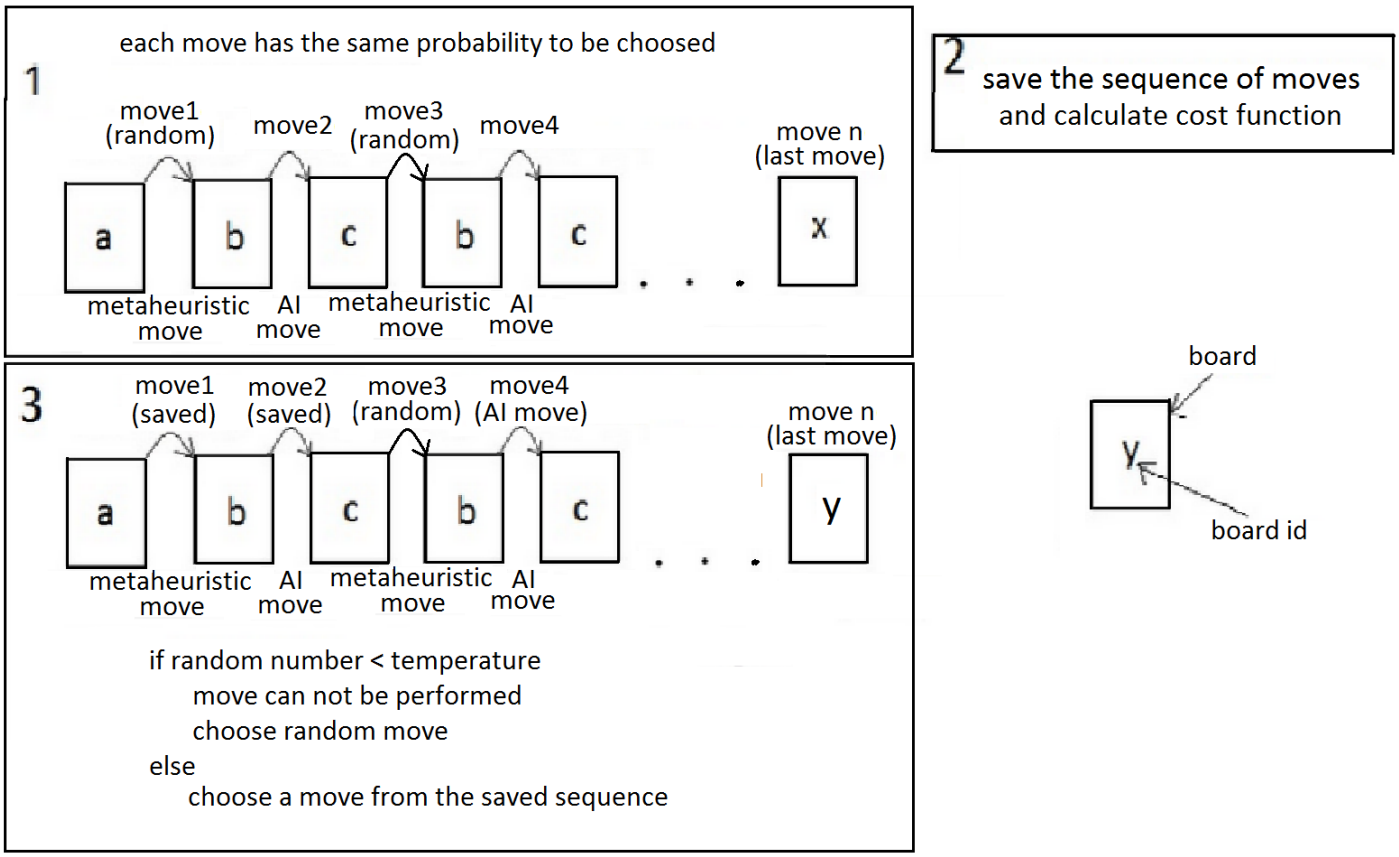

Figure 11: Diagram of the implemented simululated annealing algorithm

moves (including opponent's moves) are stored and cost function is calculated (at this point it is the best cost function). Then the algorithm simulate another game (iteration) in which depending on the temperature it chooses a stored move or a random move. If the algorithm chooses a stored move for its turn then a stored move is chosen for the opponent's turn, but if a random move is chosen then the opponent's move is performed by AI (and the stored move is omitted in further game). At the beginning the algorithm chooses random moves often but in any another iteration the probability of random move decreases. If there is a checkmate or maximum number of turns is achived the game ends. At the end of every simulated game (every iteration) cost funcion is calculated and if it is better than the best stored cost function then the old stored moves and the best cost function are exchanged for present ones. Sometimes a stored move cannot be performed becouse of a change in previous moves therefore in such situations random move is performed. There can occour a situation in which game ends before achieving maximum number of turns (for example if there is a checkmate) and its cost function is greater than the previous one. Then in the next simulated game the algorithm will have less stored moves for example 4 and when this number is exceeded random moves are performed in metaheuristic's turns and AI moves in opponent turns. The cost function is based only on weights of chess pieces. Weights for chess pieces are as follows:

pawn - 1
knight - 5
bishop - 5
rook - 7
queen - 10
king - 1000

Weights are positive for own pieces and negetive for opponent pieces. It is important to assign a weight to the king instead of giving weight (in absolute value) value for checkmate, because then it is possible to consider better and worse situations for games that end before achieving max number of turns. without that the algorithm would have all the time the same sequence of stored moves except very unlikly situations that metaheuristics wins, then the sequence would change only once.

4.3.2 GUI

GUI is depicted in figure 12).

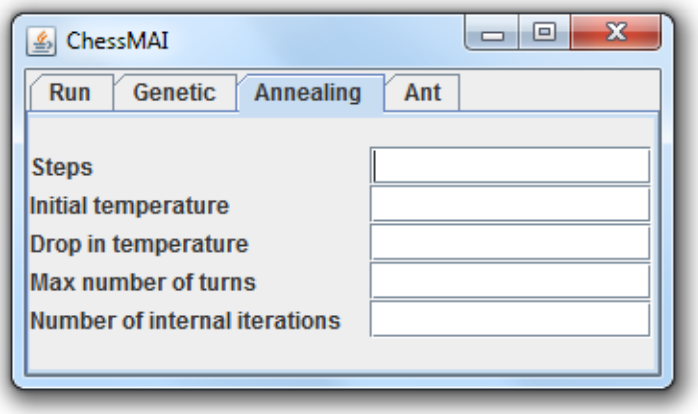

Figure 12: Simulated annealing - dialog boxes

- Steps
  Here is a place to input sequence of moves, after that

one can display all or a part of a game in a way that every another move is displayed after pressing a button

- Initial temperature
  A decimal number in a range from 0 to 1.0. The greater the temperature the greater the chance of choosing a random move. Temperature decreases after every iteration of algorithm.

- Drop in temperature
  A value by which the temperature drops after each iteration.

- Max number of turns
  An integer number that says how many turns at most can be performed in one game. A turn is defined as two moves, but to simplify number of turns increases after every black player move.

- number of internal iterations
  An integer number that says how many times the algorithm should repeat an action of searching better solution.

4.3.3 experiments

Due to big randomness of the algorithm work it is not simple to tell which parameters should be used in order to obtain the best results. Using trial and error method following parameters have been chosen:

- Initial temperature = 0.8

- Drop in temperature = $\frac{Initial\ temperature}{number\ of\ internal\ iterations} \cdot 0.9$

  in this case the temperature for the last iteration equals 10% of the initial temperature. that means 7.2%

Of course the greater number of the "number of internal iterations" parameter the greater the chance to achieve better solution. Experiments was made in a range from 10 to 20000 for "number of internal iterations" parameter. Experiments was made most often for value of 10 for "Max number of turns" parameter. There were considered 2 kinds of initial board, standard initial situation for chess and situation depicted in figure 5.

For both boards metaheuristic player is white and it is actually its turn.

4.3.4 Result

Unfortunetelly checkmate was not achived. The best result for standard initial chess board was cost function: 21.0 for parameters:
- internal iteration number: 10000
- max turn number: 10
- initial temperature: 0.8
- drop in temperature: $\frac{0.8}{10000} \cdot 0.9 = 0.000072$

in time: 1485 seconds

And for initial situation depicted in figure 13 the best result was cost function equals 31.0 for parameters:
- internal iteration number: 20000
- max turn number: 10
- initial temperature: 0.8
- drop in temperature: $\frac{0.8}{20000} \cdot 0.9 = 0.000031$

in time: 3734 seconds

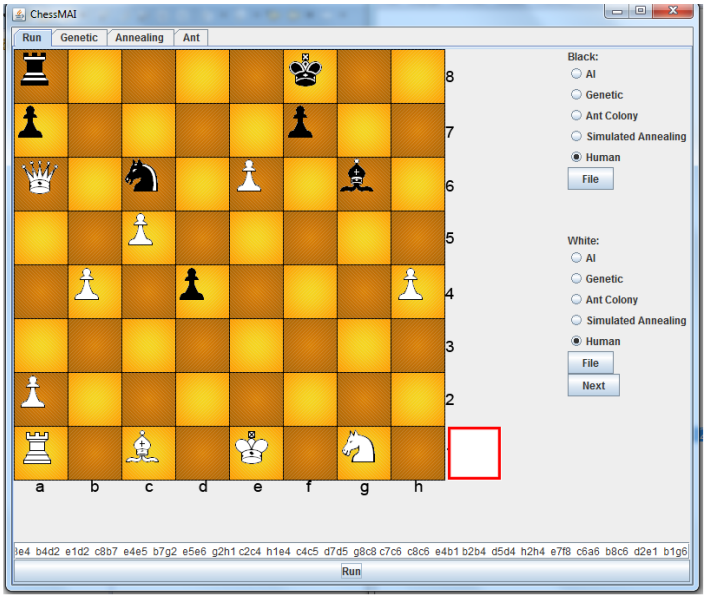

Figure 13: Initial board

It is difficult to achive a checkmate by the algorithm due to several precised moves has to be performed to obtain one-time increasing of the cost function. Different wariants of the algorithm could be considered, the algorithm could simultaneously play for example 5 different games instead of one and at the end of all games choose moves (that would exchange previous stored moves) from the game that achieved the greatest cost function (as far as this cost function would be greater than previous one), or the algorithm could save only its moves and every opponent move would be performed by AI.

5 Improvements

Some improvements in application could be considered for further development:

- Providing a possibility of entering all the modes and variables in GUI, so that one is able to change a wider range of parameters non-programically

- providing new engines implementing UCI (other than Firenzina). They could be used randomly as AI is choosing his move. Through that the metaheuristics algorithms would have much wider knowledge. Sometimes mistake moves can be made due to using only one artificial intelligence, but by using several different engines randomly, it would with high probability not be just a mistake of external AI, but a truly needed sacrifice.

- Providing previous data import for algorithms to make them able to continue learning from previous stored points.

6 Conclusion

In conclusion, our experiments proved that it is possible to make a kind of artificial intelligence for chess using metaheuristics that can play better than a completly random algorithm. The results can vary greatly between not only a chosen metaheuristic, but also the chosen parameters and

algorithm's details. However, it has to be emphasized that metaheuristics require a huge amount of learning time even when being close to the end of the game and as such would not be viable alternative for a standard type of artificial intelligence. The algorithm would work better if the cost function better expressed a situation on the board. Actually only a material situation is considered, there is a possibility to consider a positional situation, for example the weight of a pawn would increase after each move towards promotion field or the weight of a rook would increase with the number of possible moves that can be performed by the rook, it could be also considered if certain chess piece is protected or not. Implementation of prediction of one or several moves ahead could make the algorithms work better. Through that the algorithms could avoid simple tricks depending on temptation by the enemy to execute a favorable movement in this turn but obtaining much worse situation in next turn, for example, in this turn you can refute my bishop but in another turn I will refute your queen.

References

[1] Vecek, N. ; Crepinsek, M. ; Mernik, M. ; Hrncic, D., A comparison between different chess rating systems for ranking evolutionary algorithms

[2] Dorigo, M. ; Maniezzo, V. ; Colorni, A., Ant system: optimization by a colony of cooperating agents

[3] David, O.E. ; van den Herik, H.J. ; Koppel, M. ; Netanyahu, N.S. , Genetic Algorithms for Evolving Computer Chess Programs

[4] S. Kirkpatrick; C. D. Gelatt; M. P. Vecchi., Optimization by Simulated Annealing

[5] R. Huber, S. Meyer-Kahlen, Universal Chess Interface, http://www.shredderchess.com/chess-info/features/uci-universal-chess-interface.html, 2015/06/01

[6] Kranium (Norman Schmidt), Yuri Censor (Dmitri Gusev), Firenzina Engine, http://firenzina.wikispaces.com/, 2015/06/01