ISTANBUL UNIVERSITY –
JOURNAL OF ELECTRICAL & ELECTRONICS ENGINEERING

YEAR : 2003 (793-800)
VOLUME : 3
NUMBER : 1

A NEW RULE-BASED APPROACH FOR COMPUTER CHESS PROGRAMMING USING GP-ARTIFICIAL TECHNIQUES : PECP

Y.Güney HANEDAN [1] **Ahmet SERTBAŞ[2]**

[1,2] Istanbul University, Engineering Faculty, Computer Engineering Department
34850, Avcılar, İstanbul, TURKEY

[1]E-mail: ghanedan@yahoo.com [2]E-mail: asertbas@istanbul.edu.tr

ABSTRACT

In this paper, we use a brand new chess engine programming technique which we name PECP (Positional Evolutionary Chess Programming), that brings the Artificial Intelligence and Genetic Programming approaches together, to construct a chess endgame analyzing engine. Throughout the paper, the technique and the algorithm are discussed in detail. Also, using PECP, an example program (RETI V1.0) aimed to prove the correctness and performance of the rule-based theory and algorithm is written in PROLOG language.

Keywords: *Genetic Programming, Chess Play, Endgame Engine , Positional Rule-based*

I. INTRODUCTION

Evolutionary Computing (EC) is a rapid devoloping area in Computer Science. It has been used for a wide range of applications from optimization, modeling and simulation to entertaintment.

The one of its important application areas is the chess programming The complexity of the search space in chess is far beyond the imagination. To be search exhaustively, for interesting games, trees of possible continuations are very complex. Such complexity makes it practically impossible to evaluate all possible next moves that would occur from an initial position with any recent computer available. Thus, intelligent methods for tracing the search tree are necessary.

The most popular among them is the *minimax principle*[2], was introduced by Shannon (1950). In minimax principle the search tree is only traced down to a few levels and a general idea of the forthcoming moves is predicted. For the nodes in the search tree represent the board positions, a minimax run can be schematized just as the fig.1.

The minimax algorithm has to visit all of the nodes in the search tree before selecting the optimum move line indicated by bold arrows in fig.1. For this reason, a more economical approach inspired by the minimax principle called *alpha-beta pruning,* uses an advanced version of the minimax algorithm. Here, the algorithm keeps the minumum weight value estimated so far (alpha) along with the maximum weight value (beta).

Received Date : 09.09.2002
Accepted Date: 22.12.2002

While evaluating some other node, if beta is exceeded, the process stops tracing on that branch and advances the search on another one[1].

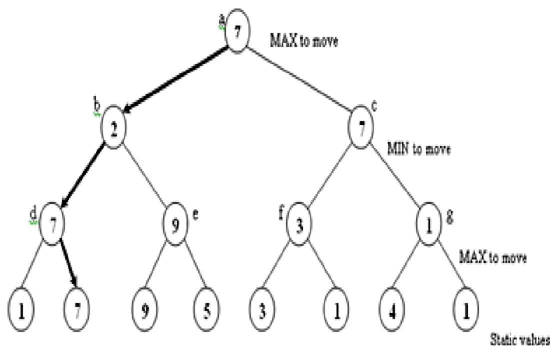


Figure 1. Minimax optimum play.

[Root node indicates the initial position and its corresponding weight. The player with side to move is denoted as MAX and the opponent as MIN.MAX to play selects the highest weighted move and MIN to play selects the lowest].
The basic alpha-beta algorithm is as follows;

I. Start with position *a*.
II. Move down to *b*.
III. Move down to *d*.
IV. Take the maximum weight value among *d*'s children, (*alpha*).
V. Backtrack to b and move down to *e*.
VI. Search for a weight value among *e*'s children, w(max $_e$), which is greater than *alpha*. When you encounter such a value, stop the search immediately. Because this is enough for MIN to realize that there is a better path for MAX after selecting move *e* than the successors of move *d.* This means MIN can decide position *e* is inferior for itself than position *d*, even without investigating all children nodes connected to node *e* [2].

In 1975, it was developed a more compact formulation of the above algorithm by Knuth and Moore using the neg-max principle instead of minimax. In 1980 and 1982-85, Advice Language approach using pattern knowledge (positional approach) was introduced by Michie and developed by Bratko respectively.

On the other hand, as opposed to search intensive approach, the different method to chess programming, knowledge-intensive (tactical) approach was introduced by Berliner and Pitrat in 1977.

But, the increasing power of computer hardwire and the implementing of special purpose chess hardwire have effected the search speed of millions of positions per second. So, search intensive approach has gained the superiority on the knowledge intensive approach.

Nearly all commercial chess engines use alpha-beta pruning or advanced versions of it. For instance one of the most popular commercial chess engines Deep Junior, uses a method called Brute-Force approach culminated in 1997 which allows the program to ignore moves that threaten nothing in addition to alpha-beta pruning. This enables the tracing to dive deeper in the search tree but also creates a possibility to underestimate some good lines. However, it has the shortcoming of the program.

In this study, using genetic and artificial intelligent techniques, a new algorithm depends on the positioanal rule-based approach is presented on the chess endgame application.

2. THE APPROACH METHOD

In high level chess, there are two main styles of play; *tactical* and *positional*.

As it is well known, tactical play depends on calculating N moves further and discovering some piece capture there. From tactical point of view, any recent computer chess program that chiefly uses alpha-beta pruning can be really strong, but, as general, it is not strong from positional aspects.

Opposed to tactical play, positional play requires strict book knowledge on positional themes such as centralization, open files, over-protection or maneuvering against weaknesses, etc [7]. Positional play grants us the ability to choose the best move 'without' calculating N moves further. A positional player just examines the position; the relative places of the pieces, strong and weak points of both the enemy and his/her own, again he/she investigates the board from the view of

positional opportunities which is formally called 'small advantages' that can only be obtained by the accompaniment of the 'sense' that leads to a 'plan'. Positional play depends on rules which are accepted as guidelines for specific chess phases.

Although positional rules are the fundamental parts of chess, it's interesting that there are few studies for rule-based computer chess programming in the literature and they are at most designed as natural language advice engines for intermediate players [5]. The main reason for rule-based programming being not so admired relies on the logical claim that one cannot generate every general rule, or a series of general rules which would cover all types of positions that may arise in chess. Although this may seem quite reasonable, such a claim just refutes itself because there is no need to find such general rules that can be applied to any probable random chess position while programming. It's enough to express sufficient number of such rules. In chess, sometimes you have to give up a piece of your own in order to capture a more valued piece after N number of moves which we name 'piece sacrifice'. Another claim against rule-based programming is said to be the impossibility of making piece sacrifices with such a chess engine. The claim depends on this theme; if you don't calculate N moves further you will not be able to see such an advantaged line of play because giving up a piece always seems disadvantageous within the scope of one move.

In the next chapter, using the positional rule-based approach, the PECP algorithm is given in detail, on an endgame application.

3. PECP ALGORITHM

PECP algorithm was intended to be a Genetic Algorithm (GA), so the terminology used here belongs mainly to Genetic Programming (GP). For GP being a relatively new programming technique, it's important to have some biological background to understand the terminology here, references [3] and [6] would provide some useful help. In this algorithm, a specific purpose ('king & pawn vs. king' endgame) is targeted. So, standart GA rules are not used, all of time.

The *algorithm* is as follows;

- *Preparation Phase;*

1. Define the chess board.
2. Define adjacent squares.
3. Define a function that finds the shortest distance between two given squares. For future use in larger functions, define the vertical, horizontal and diagonal distances separated from irregular distances.
4. Define a list structure that will represent the board positions.
5. Define list operation functions that will be used in displaying and modifying lists. Some of those must perform special element insertions or deletions according to the board representation used. (For example deleting the 2nd element *e* from some list *L* or enumerating list L by inserting corresponding indexes as the nth element).
6. Define a function that is able to find all possible legal board positions. This function will also check if a given initial position by the user is a legal board position or not.
7. Construct a user interface which is capable of getting the initial position of the pieces and the side to move from user.
8. Define a function that transforms the information taken from user into a board representation. This same function should also be able to insert the initial board position to the database.

- *Defining Artificial Intelligence sub-functions;*

9. Define a function for the attacking conditions of pieces. The function should be able to find all squares to which a piece attacks, from a given piece position.
10. Define a function that manages the movement of pieces. The function should be capable of finding all squares to which a piece can move, from a given piece position.
11. Define a function for the capturing conditions of the pieces.
12. Define positional functions such as; opposition for white & black, distant and diagonal opposition, the endgame

rule for black king that decides the position of it to reach the white pawn (staying in the magic square) etc. Those functions should use the whole board representations as input. Define as many logical and theoretical functions as possible (We used basically references [8] and [9] to find out some important endgame rules).

13. Define a function that takes a board position as input, mutates it (makes a move), and gives the resulting board position as output. The resulting position should have generation number N + 1 and thus, must be a legal next position. This function should be able to make every possible legal move within every possible position. You can use the function defined in step 10 as a sub function to build this one.
14. Define *absolute draw* and *absolute win* positions as separate functions. In the program, an absolute win is a position in which the white pawn had reached to 8th rank (queened) and black has a legal next move. Absolute drawn board positions should just contain legal drawn positions defined by chess rules.

- *Defining Weights as AI functions;*

15. Define weight values for board positions. In the 'king & pawn vs. king' endgame stage there are only two possible outcomes for a problem; *win* or *draw* (a *loss* for black is accepted as a *win* for white). An *absolute win* position should have the weight value 1 and an *absolute draw* position should have the weight value 0.Between 0 and 1, define as much weight functions as possible. These vice-functions should check up to which of the positional rules, or a series of rules, a given position obeys, that were defined in step 12.The vice-functions whose weights are extremely close to a win or draw (including the *win* and *draw*) should be deterministic. Other vice-functions should be non-deterministic.For instance if the white pawn is on the 7th rank and if the black king is unable to prevent it from queening, then such a position (this is not an absolute win yet, but is likely to become an absolute win in one move), must have a deterministic weight value. If not, the position would also obey rules that were applied to less advantaged positions for white. For example, although the discussed position's weight is 0.9 in our program, if the extreme weight function was designed to act non-deterministic, then the same position would also obey rules whose weights are, let's say, 0.8 and 0.75.This would cause the level-1 selection algorithm to underestimate the move.This is one of the most important and brand new properties of PECP technique. Our rules are not position specific. Rather, the rules defined in the program are generalized. Usually a position can obey to several rules and this adds great dynamics to our level-1 selection algorithm.

- *The GA phase;*

16. Get the places of the pieces and whose side it is to move from user.
17. Transform the user position into a board position represented as a function by assigning a generation number $N = 0$ to the initial position. Insert *generation N* to the database.
18. While $N \leq 50$ do ;
    a. Create a list representation *L* of generation *N*.
    b. Create a population *EL 'Enumerated List'* , from *L*, which consists of all possible board positions represented as list structures with their corresponding individual numbers *I* and generation numbers *N*, which may arise from the next move.This process is equivalent to mutated reproduction of individual *L*. The mutation probability is %100 and the initialization method is *full*. The population size is the number of moves that the player with side to move can make and is not a fixed value. For a 'king & pawn vs. king' endgame, maximum population size is 10 and minimum size is 0.
    c. Apply weight functions to all of the individuals in *EL* and create *WEL (weighted enumerated list)* which is a list containing the same individuals with

EL but this time with their corresponding weight values W_i with *i* being the individual number. While applying weight functions, if a position obeys a deterministic rule it's clear that it must only have one weight value, if not, find all possible weight values for that individual and take their arithmetic average and assign this value as W_i. We call step-c and d as 'level-1 selection algorithm'.

d. Select the best 3 individuals from *WEL* and insert them into the population deleting the rest of the moves. If it's white to move, select the highest weighted moves; else select the lowest weighted moves. New population is called *Best3ofPop.*(Of course, if there are only two possible moves, then the *Best3ofPop* will contain both of them, or if there is one then it will contain just that position.).
e. Delete all of the weight values from *Best3ofPop* and arrange a *tournament selection* ;
   1. Select two individuals, i_1 and i_2 from *Best3ofPop* and apply level-2 selection algorithm to them. Winner of the two is denoted as *SW (Sub winner).*Since we are concerning about the endgame stage, the program again uses positional rules for level-2 selection criteria but this time with a higher understanding of the position. In this step it's also possible to use alpha-beta pruning or some other selection criteria as well. In our program level-2 selection is completely deterministic and is independent of weight values.
   2. Apply level-2 selection algorithm to the winner (*SW*) of step e-I and the remaining individual i_3.The winner of the two is denoted as *Winner*.
f. Insert *generation N* as *move N* to the database.
g. Delete *generation N* from the database.
h. If the Winner is an *absolute draw* or an *absolute win* (termination criteria) pass to step 19, else ;
i. Transform *Winner* into a *generation* function; *generation (Winner)* which already has generation number N + 1.
j. NewN ← N + 1.
k. Repeat step 18 for *generation NewN*

19. Display $\sum$ *(i = 0 , i = NewN)* move(i).

4. REALIZATION

In this application, the chess board is represented by defining all of the squares as follow:

[Square(1/1), Square(1/2), Square(1/3),..,Square(8/8)],

where, Square(1/1) stands for a1 and Square (2/3) stands for b3 in chess notation. Also, the board positions are represented as;

[N,wk (WX / WY), wp (PX/PY), bk (BX/BY), Stm],

where N is the generation number, wk (WX / WY), wp (PX / PY), bk (BX / BY) are the piece positions, with WX, PX, BY being the pieces' X coordinates and WY, PY, BY being the Y coordinates on board (wk = white king, wp = white pawn, bk = black king) and Stm being the side to move (black or white).

One of the most important functions used frequently in larger procedures is visually Closer/3 which takes 3 squares S1, S2, S3 as input and generates a true value if S1 is closer to S2 than S3. Which means the shortest distance between S1 and S2 is less than the shortest distance between S2 and S3.

We use many weight functions to determine the weight value of a position. The function can be a theoretical position or it can be a logical procedure. For example:

draw ([_, _, wk (3/8), wp (2/6), bk (1/8), black]): - !. (1)

(1) is a theoretically drawn position. Also,

WhiteCanDrive
([I,N,wk(WX/WY),wp(PX/PY),bk(BX/BY), white]):-
PY2 is PY + 2, PX =\= 8, PX =\= 1,
VisuallyCloser
(Square(WX/WY),Square(PX/PY2),Square(BX/BY)),
not(cancapture

([I,N,wk(WX/PY2),wp(PX/PY),bk(BX/BY), white]))

(2)

is a logical procedure defining a position in which white king 'seems' to be able to drive black king from the queening square. If the reader takes a closer look at these procedures he/she will notice that although (1) is deterministic (2) is not. This is because;

weight(P , 0):- draw
and;
weight(P , 0.58):- whiteCanDrive(P).

That is to say; if a position is a definite drawn position which we represent as draw(P) then there is no need to look for if the same position is also draw, according to some other absolute draw rule, and again, there is no need to find alternative weight values for this position because the weight value for a draw position is an extreme weight value (0), and must strictly be deterministic to avoid it from being under or overestimated.

In contrast with the previous rule, if a position doesn't obey to an extreme weight function then find all other rules by which this position is also weighted with. For instance, for a whiteCanDrive position, try all other whiteCanDrive rules to which this position obeys, and find all other rules (such as opposition(P), instinct(P), intelligence(P) etc.) to which the position also obeys. This means, find all possible weight values for position P whose weight is not an extreme weight value, even if there are multiple same weight values for P. Thus, non-extreme weighted positions' weight functions must strictly be non-deterministic to provide sensitivity for level-1 selection.

For example let's consider position P, if P is a highestW position (a rule that indicates the highest wining probability), then it will only have a weight value of 0.80 which is limit for extreme values and is also accepted as an extreme weight value. On the other hand let's consider P doesn't obey any of the extreme weight functions but obeys 4 non-deterministic functions like whiteDrives, whiteCanDrive, instinctW1 and intelligenceB. The corresponding Weight List for this position could be

[0.70, 0.58, 0.58, 0.47, 0.11, 0.11, 0.11]

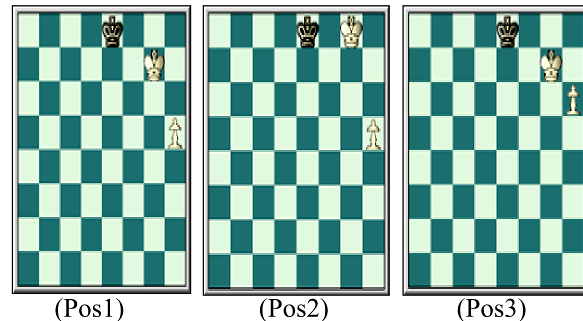
From the above Weight List we can conclude that P obeys 1 whiteDrives rule (weight = 0.70) , 2 different whiteCanDrive rules (w=0.58), 1 instinctW1 rule (w = 0.47) and 3 different intelligence B rules (w=0.11). The level-1 selection takes the arithmetic average of all these values ;

$W_p= (0.70+0.58+0.58+ 0.47+ 0.11+ 0.11+0.11)/7 = 0.38$

From which we can conclude that P is a slightly better position for black (limit value is 0.4). Although taking the arithmetic average was sufficient in our case, of course more complex formulations other than just taking the arithmetic average can be developed for more complex phases of the game.

After assigning all corresponding weight values to the individuals of a population, level-1 selection selects the best 3 individuals amongst those. As we stated earlier, after this stage, level-2 selection is applied to these best individuals. We need level-2 selection, because the weights assigned in level-1 selection were all singular values, by which we mean those weight values were determined by a judgment (rules) that were only applied to the singular properties of the position.

Let's consider the positions below;



(Pos1) (Pos2) (Pos3)

Fig.2- The positions selection based on a rule
Pos1: Initial Position (white),
Pos2: Probable next move (black) ,
Pos3: Probable next move (black).

From given initial position Pos1 (white to move), level-1 selection might have selected Pos2 (black to move), Pos3 (black to move) and some other next position as the best 3 individuals. Although

it's obvious that Pos3 is preferable to Pos2, while level-1 selection is assigning weights, it just judges Pos2 within Pos2, and Pos3 within Pos3. It hasn't got the ability to *compare* Pos2 and Pos3. Here, even the weight value of Pos2 could be slightly higher than Pos3, because in Pos2 white has the direct opposition. So it's level-2 selection who decides which move is the 'real' best among the best 3 selected by level-1 selection. As Pos2 and Pos3 are tested through double tournament selection, immediately the deterministic rule $PY_3 > PY_2$ (a rule being; white pawn is more advanced in Pos3 than in Pos2 and black is unable to reach it), will apply and Pos3 will be selected as the sub winner.
We can now pass to a test run the program with the initial position given in Fig.3;

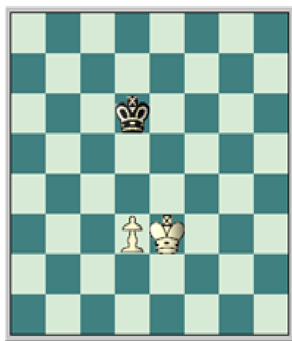


Fig.3 – Initial position to test the program.

This position is a win for white with white to move and a draw for black with black to move. Our initial position was with white to move and test results are given below. We must note that program had no specific knowledge on this position.

Table I. *The positional test results*

Initial Position:
BestMove = [0, wk(5/3), wp(4/3), bk(4/6), white];
The 'only' move that leads to a win,
BestMove = [1, wk(4/4), wp(4/3), bk(4/6), black] ;
BestMove = [2, wk(4/4), wp(4/3), bk(4/7), white]
BestMove = [3, wk(4/5), wp(4/3), bk(4/7), black] ;
BestMove = [4, wk(4/5), wp(4/3), bk(4/8), white]
BestMove = [5, wk(4/6), wp(4/3), bk(4/8), black] ;
BestMove = [6, wk(4/6), wp(4/3), bk(3/8), white]
BestMove = [7, wk(5/7), wp(4/3), bk(3/8), black] ;
BestMove = [8, wk(5/7), wp(4/3), bk(3/7), white]
A hard choice but correct,
BestMove =[9, wk(5/7), wp(4/4), bk(3/7), black]
BestMove=[10,wk(5/7),wp(4/4),bk(3/6),white]
Again the only move,
BestMove=[11,wk(5/6),wp(4/4),bk(3/6),black]
Almost equal to c7,
BestMove=[12, wk(5/6), wp(4/4), bk(2/6), white]
BestMove = [13, wk(5/6), wp(4/5), bk(2/6), black]
BestMove = [14, wk(5/6), wp(4/5), bk(3/7), white]
BestMove = [15, wk(5/6), wp(4/6), bk(3/7), black]
A clever maneuver without any specific hint,
BestMove = [16, wk(5/6), wp(4/6), bk(3/8), white]
This was played according to a general rule,
BestMove = [17, wk(5/7), wp(4/6), bk(3/8), black]
BestMove=[18,wk(5/7),wp(4/6),bk(2/8),white]
BestMove =[19, wk(5/7), wp(4/7), bk(2/8), black]
And white queens.
BestMove =[20, wk(5/7), wp(4/7), bk(2/7), white]

5. CONCLUSIONS

The obtained results for the endgame analized in this study prove that PECP method will give better results in any kind of endgame situation, than the classical methods. Recent commercial engines use tablebases in endgame but that reduces the speed of the evaluation process. Without tablebases alpha-beta pruning is not sufficient by itself, because endgame stages are fully described in chess literature; there is no point in calculating N moves further in the endgame stage.

The program can also be used in creating tablebases, also in midgame too.

REFERENCES

[1] Leon Sterling and Ehud Shapiro, *The art of PROLOG*, pp. 400-407,second edition, The MIT press London, 1999.
[2] Ivan Bratko, *PROLOG programming for Artificial Intelligence*, pp. 581-592 , third edition, Addison-Wesley 2001.
[3] W.Banzhaf, P.Nordin, R.E.Keller, F. D. Francone, *Genetic Programming*, pp133-136, dpunkt.verlag & Morgan Kaufmann Publishers, Inc. 1998.
[4] Richard Reti, "A Turkish translation of Reti's 'Modern Ideas In Chess – 1923 and Masters of the Chess Board – 1933" combined into one volume, second edition, *Broy yayınları,* pp. 32-33, Ağustos 2000.
[5] W.Barth , "Combining Knowledge and Search to Yield Infallible Endgame Programs", A study of passed Pawns in the KPKP ndgame.6,

In: *ICCA Journal,* Vol. 18 (1995), No. 3, pp. 148-159.
[6] T.A. Brown, *Genomes*, pp. 331-336, John Wiley & Sons – 1999.
[7] A. Nimzowich, *My System*, pp.157-256, revised edition, David McKay Company – 1947.
[8] R. Fine, *Basic Chess Endings*, pp. 7-9, DavidMcKay Company – 1941.
[9] L.Polgar, *Chess Endgames*, pp. 23-2 Könemann Verlagsgesellschaft mbH –199

Güney Hanedan : He was born in ORDU in 1978. He received the B.S. degree in computer engineering from the Istanbul University in 2001. He has been a chess player as professional since 1998. His research interests include digital sound design and composition, genetic and chess programming.

Ahmet Sertbaş was born in İstanbul in 1965. He received the B.S. and M.Sc. degrees in electronic engineering from the Istanbul Technical University in 1990, the Ph.D. degree in electronic department from Istanbul University in 1997 respectively. He has worked as Research Assistant at I.T.U. during 1987-1990, research engineer at Grundıg firm during 1990-1992, an instructor at the Vocational School of Istanbul University during 1993-1999. He is currently an Assoc. Professor in the Department of Computer Engineering at the University of Istanbul. His research interests include computer arithmetic circuit design, computer architecture and computer-aided circuit design, circuit theory and applications.