Emergence of Complex Strategies in the Evolution of Chess Endgame Players

AMI HAUPTMAN

Department of Computer Science
Ben-Gurion University, Israel
www.cs.bgu.ac.il/∼amiha

MOSHE SIPPER

Department of Computer Science
Ben-Gurion University, Israel
www.moshesipper.com

Received (received date)
Revised (revised date)

We examine a strong chess-endgame player previously developed by us through genetic programming, focusing on the player's emergent capabilities and tactics in the context of a chess match. First, we provide a detailed description of the evolutionary approach by which our player was developed. Then, using a number of methods we analyze the evolved player's building blocks and their effect on play level. We conclude that evolution has found combinations of building blocks that are far from trivial and cannot be explained through simple combination—thereby indicating the possible emergence of complex strategies.

Keywords: Evolutionary algorithms; Genetic programming; Chess.

Draft

1. Introduction

Genetic programming (GP) has been shown to successfully produce solutions to hard problems from numerous domains, and yet an understanding of the evolved "spaghetti code" is usually lacking. Indeed, it seems a GPer must wear two hats: that of an evolutionary designer, and that of a molecular "biologist" [28].

We wore the first hat in [12] and presented successful chess endgame players, evolved via GP. Using numerous elements of relatively simple chess knowledge embodied as tree nodes and simple means of combining them logically, our evolved players were able to perform on four types of endgames at a level nearly equal to that of CRAFTY—a world-class chess program.

In this paper we wish to wear the second hat—that of the molecular "biologist"—in an attempt to understand the resultant complex intelligence, hidden within the innards of our evolved programs. We examine the complexity of our evolved programs by means of analyzing the performance of a strong player developed by evolution, across various positions taken from games played by it. We argue that

some of the strategic and tactical capabilities of our players are emergent. As far as we are aware no strong chess program has been written by relying on elements of chess knowledge—as we have done; programs to date rely mostly on deep search (for example, see [6]).

This paper is organized as follows: In the next two sections we provide background on chess programs and summarize previous work on evolving chess strategies, including the experiment in which our players were evolved. In Section 4 we display a sample endgame, played by a strong player we developed, along with a tactical analysis of its performance, comparing its move choices to those of a strong chess engine. Section 5 looks further into the emergent aspects of our player's skill by testing its building blocks, both separately and constructively. Section 6 ends with concluding remarks and future work.

2. Background and Previous Work

2.1. *Machine chess*

For more than 50 years the game of chess has served as a testing ground for research in the field of artificial intelligence. During these five decades the progress of chess programs in terms of their measured performance ratings has been steady. This progress has come most directly from the increase in the speed of computer hardware, and also straightforward software optimization. Deep Blue, the famous computer program and hardware (32 computing nodes, each with eight integrated processors) that defeated Kasparov in 1997, evaluated 200 million alternative positions per second [4]. In contrast, the first computer that executed Belle, the first program to earn the title of U.S. master in 1983, was more than 100,000 times slower. Faster computing and optimized programming allows a chess program to evaluate chessboard positions further into the game tree, and thus achieve stronger levels of play.

It is important to note, however, that state-of-the-art chess programs, such as Fritz and Junior, do not rely on generating a great number of positions alone. Such brute-force machines are now on the brink of extinction (and are referred to as "dinosaurs" by chess engine programmers). Nowadays, most programs participating in world championships run on standard PCs. Advanced methods, such as the NegaScout algorithm [20, 24], transposition tables [9] and the history heuristic [27] are commonly used, together with sophisticated evaluation functions.

Since our research employs genetic algorithms to develop chess players, we start with a description of the method.

2.2. *Genetic Algorithms*

Numerous fields of scientific research often had to deal with the classical problem of optimization. While purely analytical methods have widely proved their efficiency, they nevertheless suffer from an insurmountable weakness: reality rarely obeys the differentiable functions so common in scientific models.

Nature, on the other hand, has adopted a somewhat different approach. The problem of survival in a harsh, competitive environment was not straightforwardly solved by an accurate model. If we view nature as a problem solver, attempting to optimize survival of living beings, one of the first observations is that the very first solutions (creatures) were not optimally suited for this task. However, during millions of years, solutions kept improving. Gradually, more complex, adapted species developed, while the less fit perished. Only the fit survived the test of time.

The attempt to implement nature's way in the field of computer science is known as Genetic Algorithms (GAs). John Holland, from the University of Michigan, began his work on genetic algorithms at the beginning of the 60s. A first achievement was the publication of Adaptation in Natural and Artificial System in 1975 [13]. Holland had a double aim: to improve the understanding of the natural adaptation process, and to design artificial systems having properties similar to natural systems.

A GA is an iterative procedure, that consists of a constant-size population of individuals, each one represented by a finite string of symbols, encoding a possible solution in some problem space [26].

The generic GA (as described, e.g., by Mitchell [21]) works as follows: First, an initial population of chromosomes, called *individuals*, is generated. Every evolutionary step, or *generation*, the individuals in the current population are evaluated by applying a *fitness function*, and each member is assigned a *fitness* value. This value indicates how well the individual solves the problem, and affects the probability of its being selected for reproduction.

New individuals are created by stochastically applying the genetic operators to selected individuals. *Selection* is biased towards elements of the current generation which have better fitness, though it is usually not so biased that poorer elements have no chance to participate, in order to prevent the solution set from converging too early to a sub-optimal or local solution. There are several well-defined selection methods; roulette-wheel selection and tournament selection are popular methods.

Following selection, genetic operators are applied to the selected individuals. The most widely used ones are *crossover*, *reproduction*, and *mutation*. The *crossover* (or recombination) operation is reminiscent of natural gene transfer from parents to offspring. A simple form of crossover is the exchange of substrings after a randomly selected crossover point. The result is two offspring, containing data from both parents. *Reproduction* is simply passing individuals from the current generation to the next one. *Mutation* is a unary operator, usually taking place after crossover. It is used mainly to avoid premature convergence to a local minimum. Typically, one or more bits are selected and flipped at random with some (small) probability.

These processes ultimately result in a new generation, differing from the current one. The process then continues iteratively, until a termination condition is met. Optimally, the algorithm terminates when the performance of the best individual of the current generation abides to some predefined criteria. However, since GAs are stochastic by nature, there is no guarantee of success. To avoid endless iterations, execution is halted also when convergence is observed, or after a time limit is met.

```
Produce an initial population of individuals
Evaluate the fitness of all individuals
While termination condition not met do
   Select fitter individuals for reproduction
   Recombine individuals
   Mutate some individuals
   Evaluate the fitness of the new individuals
End while
```

Fig. 1. Pseudocode of a generic genetic algorithm.

The generic pseudocode of a GA is given in Figure 1

Genetic algorithms have been successfully applied to numerous problems from different domains (see [21] for several examples).

2.3. *Genetic Programming*

Representation is a key issue in problem solving, and genetic algorithms are no exception. Well-formed representations not only capture the essence of the problem, but do so in a compact way, minimizing noise overhead, and allowing to focus on what is truly relevant.

When solving real-world problems very few GA implementations use the generic representation we described above. One of the main problems is that string-based representation schemes are difficult and unnatural for many problems. The need for more powerful representations has been recognized for some time [16]. Moreover, using fixed-length strings forces predetermination of the size and shape of solutions. This has been the bane of machine learning systems for many years [25].

Since using computer programs as a model for solving problems is one of the most fundamental aspects of computer science, it is clear that programs could also be used as the representation of solutions formed and manipulated by GAs. This notion is implemented in Genetic Programming.

Genetic Programming (GP) is a sub-class of evolutionary algorithms, introduced by Koza [16]. In GP, individuals are not represented by fixed-length strings, but by *hierarchical* computer programs of variable size. The language chosen by Koza was LISP (LISt Processing, in particular, the Common LISP dialect [29] is typically used). The reasons for choosing the LISP programming language are multiple. The main reason is that since LISP expressions are of nested form, the group of legal LISP expressions is closed under most genetic operators, while the group of legal C programs, for example, is not. This point is further discussed below.

GP Trees are constructed from *functions* and *terminals*. The functions are usually arithmetic and logic operators (including IF expressions) located at the inner nodes of the tree; the terminals are zero-argument functions, located at the leaves, which serve both as constants and as sensors. Sensors are a special type of function

that query the domain environment.

During the course of artificial evolution, GP trees are continually executed as programs, and changes are continually applied to them. This is when the advantages of LISP come to the fore. As stated above, GP individuals are composed of LISP expressions—nested lists of symbols (S-expressions). Due to the virtual synonymity of S-expressions and their parse trees, it is both possible and convenient to treat a computer program in the genetic population as data so that it can first be genetically manipulated. Then, with the same ease, the result of the manipulation can be executed as a program.

Most compiled programming languages internally convert, at the time of compilation, a given program into a parse tree representing the underlying composition of functions and terminals of that program. More often than not, this tree is unavailable to the programmer. If we were to write a genetic program in C or C++, for example, we would probably not have direct access to the parse tree, and would need to apply genetic operators to the program's code itself. This is far more difficult, due to the fact that the code's syntactic structure is not readily apparent in the text itself without applying grammatical rules; without structural knowledge replacing tokens this will probably lead to syntactically illegal programs. While GAs can deal with low-fitness programs, illegal ones pose a serious problem and must be avoided.

On the other hand, since S-expressions are, in effect, their own parse trees, this problem is avoided altogether–as long as we replace an *entire* sub expression (or sub tree) with another, the program will retain its syntactic legality. We are ready to apply the genetic operators.

The genetic operators used in GP are essentially the same as those described for the generic GA. However, since we are dealing with trees, some genetic operators need to be redefined:

- The GP binary crossover operator randomly selects an internal node in each of the two individuals and then swaps the sub-trees rooted at these nodes. An example is shown in Figrue 2. Note that tree depth may vary due to crossover.
- The unary mutation operator randomly selects one node from the tree, deletes the subtree rooted at that node, and then grows a new sub-tree instead. An example is shown in Figure 3.

Another widely used operator in GP is the reproduction operator, described in Section 2.2, but there is nothing different in applying it on programs.

Very similarly to the generic GA, at each generation individuals are selected for reproduction based on their fitness. A common fitness method is *competitive fitness*, in which an individual's score depends not on some absolute measure, but on success against its peers. This way, not only are we less dependant on finding an absolute measure for fitness (which is exceedingly difficult for complex problems), but it is also more likely that the population will learn to perform better in the

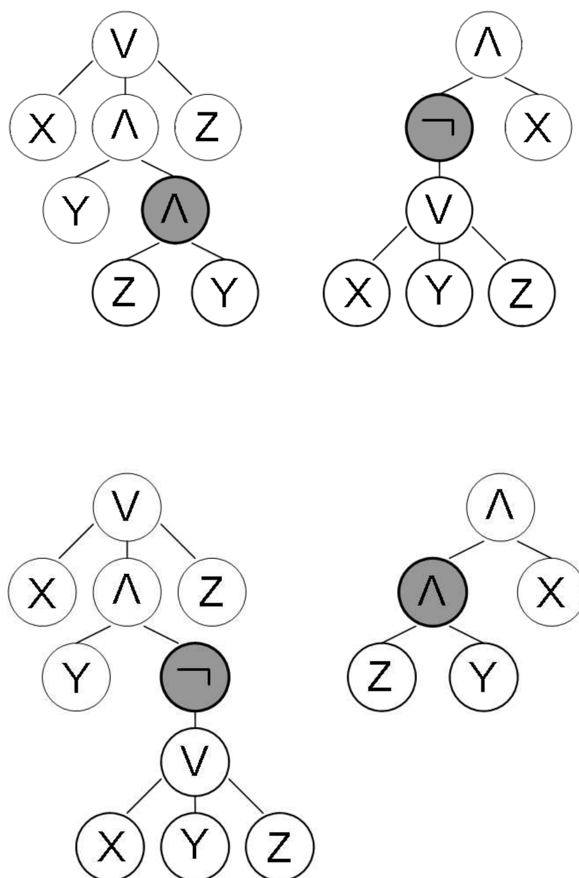

Fig. 2. Crossover operator in GP. Top: parent trees, with crossover points shaded. Bottom: Offspring resulting from crossover at shaded points.

broader sense, without maximizing a single fitness function [1].

2.4. *Evolutionary algorithms and chess*

Fine-tuning the evaluation functions lies at the center of efforts in the field of applying evolutionary algorithms to developing chess players. As GP has recently been argued to deliver "high-return, human-competitive machine intelligence" [18], this method (along with other forms of evolutionary algorithms) has been applied repeatedly to this domain. While numerous AI methods have been applied to the field of chess, we only review those related to genetic algorithms.

Kendall and Whitwell [15] used evolutionary algorithms to tune evaluation-function parameters. They focused mainly on the weights of the remaining pieces,

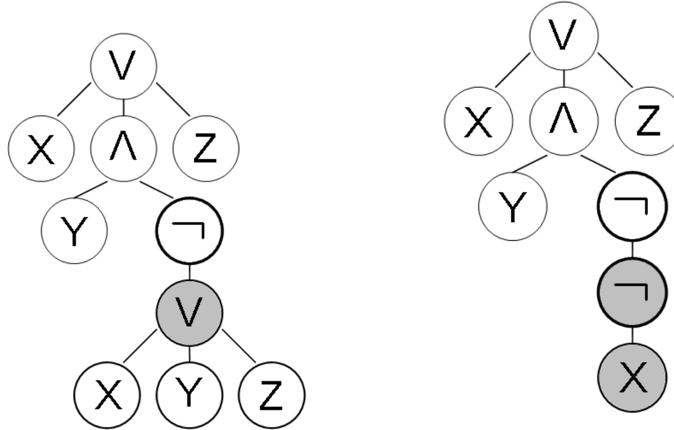

Fig. 3. Mutation operator. A sub-tree (root marked in gray) is selected from the parent individual (left) and removed. A new sub-tree (right) is grown instead (marked in gray).

and neglected more sophisticated board-evaluation functions. Typical functions included: material values for the different pieces, penalty for bishops in initial positions, bonus for pawns in center of chessboard, penalty for doubled pawns and for backward pawns, castling bonus if this move was taken and penalty if it was not, and rook bonus for an open line or on the same line of a passed pawn. The resulting individuals were successfully matched against commercial chess programs, but only when the lookahead for the commercial program was strictly limited.

Gross *et al.* [11] introduced a system that integrates GP and Evolution Strategies to learn to play chess. This system did not learn from scratch, but instead a "scaffolding" algorithm that could perform the task already was improved by means of evolutionary techniques. This was accomplished by fine-tuning an alpha-beta search to traverse less nodes, using three separately evolving modules within the search: a depth module, determining the remaining search depth for the given node; a move-ordering module, changing the ordering of all possible moves; and a position module, returning a value for a given chess position. The main result was that evolution improved the search algorithm so that it wins by only using 50% of the resources needed by the f-negascout algorithm [20, 24], and only 6% of the resources used by a simple alpha-beta algorithm [19].

A recent important work was done by Fogel *et al.* [8]. A genetic algorithm was employed to improve tuning of parameters that governed a set of features regarding board evaluation. Evaluation functions were structured as a linear combination of: 1) the sum of the material values attributed to each player; 2) values derived from tables indicating the worth of having certain pieces at certain values—"positional value tables"(PVTs); and 3) three neural networks: one for each player's front two rows, and one for the central 16 squares. Games were played using an alpha-beta

search, the depth of which was four ply, except for certain advantageous positions, where the search depth was extended to six ply.

The best evolved neural network achieved an above-Master level of performance, estimated at 2437 [a]. Although this work proved that evolution can be successful at developing good artificial chess players, it is important to note that the nonevolved programs used, without the inclusion of the neural networks, still performed at an Expert level (estimated at 2066). Thus, the evolutionary procedure did not account for the entire level of performance, but only for the (non-trivial) transition from Expert to Master.

Another important aspect of all works described above is that no apparent effort was made to match human modes of thinking, relying more on knowledge and less on search. Such attempts are typically restricted to the field of cognitive psychology.

3. GP-EndChess

We previously developed a chess endgame player using GP [12]. In our work, each individual—a LISP-like tree expression—represented a strategy, the purpose of which was to evaluate a given board configuration and generate a real-valued score. The tree's internal nodes are called *functions*, and the leaves—*terminals*. We used simple Boolean functions (AND, OR, NOT), and IF functions; terminals were used to analyze certain features of the game position. We included a large number of terminals, varying from simple ones (such as the number of moves for the player's king), to more complex features (for example, the number of pieces attacking a given piece). A more complete description of functions and terminals used is given below.

In order to better control the structure of our programs we used *Strongly Typed Genetic Programming* (STGP) [22], a method in which types are assigned to all GP-tree edges. This way, it is possible to impose structural constraints on the tree (for example, deciding that a given terminal may or may not be the value returned from the entire tree).

3.1. *Board evaluation*

Our aim was to develop evaluation strategies that bear similarity to human board analysis (for example, see [5] and [6]). Thus, instead of looking deep into the game tree we traverse less nodes—but consider each node more thoroughly. As such, our strategies use only limited lookahead–typically 1.

[a]Chess players may obtain a nationally (or internationally) recognized numerical rating, using a scoring system developed by the mathematician Arpad Elo (therefore referred to as an "ELO" score). Beginners start at 1300 points, and every win raises a player's rating using a formula that takes the difference from the opponent's rating into account (i.e., more points for defeating a stronger opponent). Master level is typically attained at 2200 points, and Grandmaster at 2400 (although these titles are only earned at special official tournaments).

The machine player received as input all possible board configurations reachable from the current position by making one legal move (this is quite easy to compute). After these boards are evaluated, the one that received the highest score is selected, and that move is made. Thus, an artificial player is generated by combining an (evolved) board evaluator with a program that generates all possible next moves.

Although this approach has been successfully used in a number of game-strategy evolution scenarios (see [7]), it was, as far as we know, the first time it was applied to chess endgames.

3.2. *Tree topology*

Our programs played chess endgames consisting of kings, queens, and rooks (in the future we shall also consider bishops and knights). Each game started from a different (random) legal position, in which no piece is attacked, e.g., two kings, two rooks, and two queens in a KQRKQR endgame. Although at first each program was evolved to play a different type of endgame (KRKR, KRRKRR, KQKQ, KQRKQR, etc.), which implies using different game strategies, the same set of terminals and functions was used for all types. Moreover, this set was also used for our more complex runs, in which GP chess players were evolved to play several types of endgames. Our ultimate aim was the evolution of general-purpose strategies.

Still, as most chess players would agree, playing a winning position (e.g., with material advantage) is very different than playing a losing position, or an even one. For this reason, each individual contained three trees: an advantage tree, an even tree, and a disadvantage tree. These trees were used according to the current status of the board. The disadvantage tree is smaller, since achieving a stalemate and avoiding exchanges requires less complicated reasoning.

3.3. *Tree nodes*

While evaluating a position, an expert chess player considers various aspects of the board. Some are simple, while others require a deep understanding of the game. Chase and Simon found that experts recalled meaningful chess formations better than novices [6]. This lead them to hypothesize that chess skill depends on a large knowledge base, indexed through thousands of familiar chess patterns.

We assumed that complex aspects of the game board are comprised of simpler units, which require less game knowledge, and are to be combined in some way. Our chess programs use terminals, which represent those relatively simple aspects, and functions, which incorporate no game knowledge, but supply methods of combining those aspects. As we used STGP, all functions and terminals were assigned one or more of two data types: *Float* and *Boolean*. We also included a third data type, named *Query*, which could be used as any of the former two.

The function set used included the If function, and simple Boolean functions. Although our tree returns a real number, we omitted arithmetic functions, for several

Table 1. Function set of GP individual. B: Boolean, F: Float.

F=If3(B_1, F_1, F_2)	If B_1 is non-zero, return F_1, else return F_2
B=Or2(B_1, B_2)	Return 1 if at least one of B_1, B_2 is non-zero, 0 otherwise
B=Or3(B_1, B_2, B_3)	Return 1 if at least one of B_1, B_2, B_3 is non-zero, 0 otherwise
B=And2(B_1, B_2)	Return 1 only if B_1 and B_2 are non-zero, 0 otherwise
B=And3(B_1, B_2, B_3)	Return 1 only if B_1, B_2, and B_3 are non-zero, 0 otherwise
B=Smaller(B_1, B_2)	Return 1 if B_1 is smaller than B_2, 0 otherwise
B=Not(B_1)	Return 0 if B_1 is non-zero, 1 otherwise

reasons. First, a large part of contemporary research in the field of machine learning and game theory (in particular for perfect-information games) revolves around inducing logic rules for learning games (for example, see [10], [3] and [2]). Second, according to the players we consulted, while evaluating positions involves considering various aspects of the board, some more important than others, performing logic operations on these aspects seems natural, while mathematical operations does not. Third, we observed that numeric functions sometimes returned extremely large values, which interfered with subtle calculations. Therefore the scheme we used was a (carefully ordered) series of Boolean queries, each returning a fixed value (either an ERC or a numeric terminal, see below). See Table 1 for the complete list of functions.

We developed most of our terminals by consulting several high-ranking chess players (the highest-ranking player we consulted was Boris Gutkin, ELO 2400, International Master, and fully qualified chess teacher). The terminal set examines various aspects of the chessboard, and is be divided into 3 groups:

1. Float values, created using the ERC (*Ephemeral Random Constants*) mechanism (see [17] for details). An ERC is chosen at random to be one of the following six values $\pm 1 \cdot \{\frac{1}{2}, \frac{1}{3}, \frac{1}{4}\} \cdot MAX$ (MAX was empirically set to 1000), and the inverses of these numbers. This guarantees that when a value is returned after some group of features has been identified, it will be distinct enough to engender the outcome.

2. Simple terminals, which analyze relatively simple aspects of the board, such as the number of possible moves for each king, and the number of attacked pieces for each player. These terminals were derived by breaking relatively complex aspects of the board into simpler notions. More complex terminals belong to the next group (see below). For example, a player should capture his opponent's piece if it is not sufficiently protected, meaning that the number of attacking pieces the player controls is greater than the number of pieces protecting the opponent's piece, and

the material value of the defending pieces is equal to or greater than the player's. Adjudicating these considerations is not simple, and therefore a terminal that performs this entire computational feat by itself belongs to the next group of complex terminals.

The simple terminals comprising this second group are derived by refining the logical resolution of the previous paragraphs' reasoning: Is an opponent's piece attacked? How many of the player's pieces are attacking that piece? How many pieces are protecting a given opponent's piece? What is the material value of pieces attacking and defending a given opponent's piece? All these questions are embodied as terminals within the second group. The ability to easily embody such reasoning within the GP setup, as functions and terminals, is a major asset of GP.

Other terminals were also derived in a similar manner. See Table 2 for a complete list of simple terminals. Note that some of the terminals are inverted—we would like terminals to always return positive (or true) values, since these values represent a favorable position. This is why we used, for example, a terminal evaluating the player's king's *distance* from the edges of the board (generally a favorable feature for endgames), while using a terminal evaluating the *proximity* of the opponent's king to the edges (again, a positive feature).

3. Complex terminals. These are terminals that check the same aspects of the board a human player would. Some prominent examples include: the terminal OppPieceCanBeCaptured considering the capture of a piece; checking if the current position is a draw, a mate, or a stalemate (especially important for non-even boards); checking if there is a mate in one or two moves (this is the most complex terminal); the material value of the position; comparing the material value of the position to the original board—this is important since it is easier to consider change than to evaluate the board in an absolute manner. See Table 3 for a full list of complex terminals.

3.4. *Fitness evaluation*

As we used a competitive evaluation scheme, the fitness of an individual was determined by its success against its peers. We used the random-2-ways method, in which each individual plays against a fixed number of randomly selected peers (see [23] for full details). Each of these encounters entails a fixed number of games, each starting from a randomly generated position.

The score for each game is derived from the outcome of the game. Players that manage to mate their opponents receive more points than those that achieve only a material advantage. Draws are rewarded by a score of low value and losses entail no points at all.

The final fitness for each player is the sum of all points earned in the entire tournament for that generation. We used the standard reproduction, crossover, and mutation operators, as in [17]. The major parameters were: population size – 80,

Table 2. Simple terminals. Opp: opponent, My: player.

B=NotMyKingInCheck()	Is the player's king not being checked?
B=IsOppKingInCheck()	Is the opponent's king being checked?
F=MyKingDistEdges()	The player's king's distance form the edges of the board
F=OppKingProximityToEdges()	The player's king's proximity to the edges of the board
F=NumMyPiecesNotAttacked()	The number of the player's pieces that are not attacked
F=NumOppPiecesAttacked()	The number of the opponent's attacked pieces
F=ValueMyPiecesAttacking()	The material value of the player's pieces which are attacking
F=ValueOppPiecesAttacking()	The material value of the opponent's pieces which are attacking
B=IsMyQueenNotAttacked()	Is the player's queen not attacked?
B=IsOppQueenAttacked()	Is the opponent's queen attacked?
B=IsMyFork()	Is the player creating a fork?
B=IsOppNotFork()	Is the opponent not creating a fork?
F=NumMovesMyKing()	The number of legal moves for the player's king
F=NumNotMovesOppKing()	The number of illegal moves for the opponent's king
F=MyKingProxRook()	Proximity of my king and rook(s)
F=OppKingDistRook()	Distance between opponent's king and rook(s)
B=MyPiecesSameLine()	Are two or more of the player's pieces protecting each other?
B=OppPiecesNotSameLine()	Are two or more of the opponent's pieces protecting each other?
B=IsOppKingProtectingPiece()	Is the opponent's king protecting one of his pieces?
B=IsMyKingProtectingPiece()	Is the player's king protecting one of his pieces?

generation count – between 150 and 250, reproduction probability – 0.35, crossover probability – 0.5, and mutation probability – 0.15 (including ERC).

3.5. *Results*

Our evolved players were capable of drawing (and winning once in a while) against the CRAFTY engine (version 19.01) by Hyatt (CRAFTY's source code is avail-

Table 3. Complex terminals. Opp: opponent, My: player. Some of these terminals perform lookahead, while others compare with the original board.

F=EvaluateMaterial()	The material value of the board
B=IsMaterialIncrease()	Did the player capture a piece?
B=IsMate()	Is this a mate position?
B=IsMateInOne()	Can the opponent mate the player after this move?
B=OppPieceCanBeCaptured()	Is it possible to capture one of the opponent's pieces without retaliation?
B=MyPieceCannotBeCaptured()	Is it not possible to capture one of the player's pieces without retaliation?
B=IsOppKingStuck()	Do all legal moves for the opponent's king advance it closer to the edges?
B=IsMyKingNotStuck()	Is there a legal move for the player's king that advances it away from the edges?
B=IsOppKingBehindPiece()	Is the opponent's king two or more squares behind one of his pieces?
B=IsMyKingNotBehindPiece()	Is the player's king not two or more squares behind one of my pieces?
B=IsOppPiecePinned()	Is one or more of the opponent's pieces pinned?
B=IsMyPieceNotPinned()	Are all the player's pieces not pinned?

able at `ftp://ftp.cis.uab.edu/pub/hyatt`). CRAFTY is a state-of-the-art chess engine, which uses a typical brute-force approach, with a fast evaluation function (NegaScout search) and all the standard enhancements [14]. CRAFTY finished *second* at the 12th World Computer Speed Chess Championship, held in Bar-Ilan University in July 2004. According to `www.chessbase.com`, CRAFTY has a rating of 2614 points, which places it at the human Grandmaster level. CRAFTY is thus, undoubtedly, a worthy opponent.

GP individuals were also pitted against MASTER: A strategy we developed by consulting several highly skilled chess players, including an International Chess Master (see Section 3.3). We implemented the ideas gleaned—both as terminals, and as more sophisticated terminal combinations, reflecting deep considerations while evaluating a position—to form the strongest man-made evaluation function we could construct: MASTER. Our evolved program, GPEC, turned out to be notably better than MASTER, the best program *we* could come up with.

We challenged both CRAFTY and MASTER in fast-paced games (known as blitz games), playing 4 types of endgames: KRKR (i.e., King and Rook vs. King

Table 4. Percent of wins, advantages, and draws for best GP-EndChess player in tournament against two top competitors.

	%Wins	%Advs	%Draws
MASTER	6.00	4.00	80.00
CRAFTY	2.00	2.00	77.00

and Rook), KRRKRR, KQKQ, and KQRKQR. Strategies were first evolved to play one type of endgame, and then to play multiple endgames. The former means that the same pieces (one endgame type) were used as starting board, with their positions changing randomly, while the latter means that several combinations of pieces (several endgame types) were used, their placement also being random. Since random starting positions can sometimes be uneven (for example, allowing the starting player to attain a capture position), every starting position was played twice, each player playing both Black and White. This way a better starting position could benefit both players and the tournament was less biased (this stratagem was adopted for both fitness evaluation and post-evolutionary benchmarking).

Although individuals developed in multiple-endgame runs achieved slightly lower scores against our two opponents, scores were still close to draw, including some wins as well. In addition, GP individuals learned more generalized patterns, allowing them to compete successfully in several types of games. This suggests stronger learning has taken place. Table 4 summarizes the results attained [12].

4. Analysis of Moves

The previous section presented results pertaining to the ensemble of evolutionary experiments performed. In this section we "zoom in" on evolved capabilities of one strong player. We wish to demonstrate that some of the considerations made by our player (embodied by different scores assigned to moves in each position)—the *output*—cannot be trivially explained by the elements supplied as building blocks for evolution—the *input*.

We describe a sample game played by a strong GP-Endchess individual, obtained at generation 190, dubbed GPEC190, or GPEC for short. scored 0.485 points against CRAFTY and MASTER on average (0.5 being a draw) in multiple-endgame runs. Table 5 summarizes the terminology used below. Here is the code for GPEC's advantage tree:

```
(If3 (Or2 (And2 OppKingStuck OppKingInCheck)
    (And3 MyFork NotMyPieceAttUnprotected (And3
        MyFork NotMyPieceAttUnprotected OppKingInCheckPieceBehind)))
    (If3 (Or2 (And2 OppKingStuck OppKingInCheck)
        (And3 MyFork NotMyPieceAttUnprotected (And3
            MyFork NotMyPieceAttUnprotected OppKingInCheckPieceBehind)))
        (If3 (And3 MyFork NotMyPieceAttUnprotected
            (And3 MyFork NotMyPieceAttUnprotected OppKingInCheckPieceBehind))
            (If3 OppKingStuck MyKingDistEdges MyKingDistEdges)
            (If3 OppKingStuck OppKingProxEdges OppKingInCheckPieceBehind))
```

Table 5. Chess-game terminology.

a..h	columns
1..8	rows
K	King
Q	Queen
R	Rook
QxR	Queen captures Rook ("x" is a capture)
Qe4+	Queen moves to e4 and CHECKS ("+" is a check)
Qe4#	Queen moves to e4 and MATES ("#" is a mate)
mate-in-n	Mate is unavoidable in n moves
Crft = 4.2	Score assigned by CRAFTY to given position
Positive scores	Favorable for White
Negative scores	Favorable for Black
9.0	(material) value of Queen
5.0	(material) value of Rook
1.5	If this is the score (or higher) the player is considered to be in a winning position

```
    (If3 (Not (Or3 OppKingInCheckPieceBehind
        (And3 -1000*MateInOne (Or2 (And2 OppKingStuck
            OppKingInCheck) (And3 MyFork NotMyPieceAttUnprotected
            (Or2 (And2 OppKingStuck OppKingInCheck) (And3
                MyFork NotMyPieceAttUnprotected (Or2 (And2
                OppKingStuck OppKingInCheck) (And3 MyFork
                NotMyPieceAttUnprotected (Or2 (And2 OppKingStuck
                OppKingInCheck) (And3 MyFork NotMyPieceAttUnprotected
                (And3 MyFork NotMyPieceAttUnprotected OppKingInCheckPieceBehind))))))))
            NotMyKingInCheck) (And2 MyFork OppPieceAttUnprotected)))
        (If3 OppKingStuck NumMyPiecesUNATT NumMyPiecesUNATT)
        (If3 (Or2 (And2 OppKingStuck OppKingInCheck)
            (And3 MyFork NotMyPieceAttUnprotected (And3
                MyFork NotMyPieceAttUnprotected OppKingInCheckPieceBehind)))
            (If3 (Or2 (And2 OppKingStuck OppKingInCheck)
                (And3 MyFork NotMyPieceAttUnprotected (And3
                    MyFork NotMyPieceAttUnprotected OppKingInCheckPieceBehind)))
                MyKingDistEdges (If3 (Not (Or3 OppKingInCheckPieceBehind
                (And3 -1000*MateInOne (Or2 (And2 OppKingStuck
                    OppKingInCheck) (And3 MyFork NotMyPieceAttUnprotected
                    (And3 MyFork NotMyPieceAttUnprotected (And3
                        MyFork NotMyPieceAttUnprotected OppKingInCheckPieceBehind))))
                    NotMyKingInCheck) (And2 OppKingInCheck OppKingInCheckPieceBehind)))
                (If3 OppKingStuck NumMyPiecesUNATT NumMyPiecesUNATT)
                (If3 (And2 MyFork OppPieceAttUnprotected)
                    (If3 OppKingStuck NumMyPiecesUNATT NumMyPiecesUNATT)
                    (If3 NotStallmateAdv 10*MaterialValue #NotMovesOppKing))))
            (If3 OppKingStuck NumMyPiecesUNATT NumMyPiecesUNATT))))
(If3 OppKingStuck NumMyPiecesUNATT NumMyPiecesUNATT))
```

In the games below GPEC plays against itself, with CRAFTY being used to analyze every move (i.e., CRAFTY also plays each board position of the game). Although GPEC played both Black and White (the original idea here was to see

Fig. 4. Sample Game: Opening position.

how GPEC fares both as Black and White) we focus only on White's moves, since in the following examples the number of possible moves for White is on average above 30, while for Black—only 3-4.

CRAFTY here serves as a superb yardstick, allowing us to compare the scores assigned by GPEC to "real" scores (CRAFTY's scores were obtained by deep analysis of each move, typically lasting 25-30 seconds, at the average speed of above 1500 KNodes/sec, so are therefore highly reliable). Since exact scores assigned by different chess engines to moves vary widely, calculating a correlation factor between CRAFTY's scores and GPEC's scores would be futile. However, if a win (mate-in-n) exists, or there is a smaller advantage to one of the sides, (near) optimal moves are more easily identified (as maintaining the advantage), and distinguished from "bad" ones (losing the advantage).

Since in this type of position (KQRKQR, with no piece attacked at the starting position) the player to move first (White in our experiments) has the strategic advantage, in order to verify correct play we should check that the player maintains the advantage, especially if a *mate-in-n* was identified by CRAFTY, making the optimal line of play well defined. The player should assign higher scores to moves assigned higher scores by CRAFTY, and lower scores to moves forfeiting the advantage.

Instead of going into the details of each move, we display scoring tables for the moves considered, and their assigned scores (both by CRAFTY and GPEC), and only discuss some of the moves.

We discuss a sample game, for which the moves appear in Table 6. The starting position is given in Figure 4.

As can be seen, the best moves according to CRAFTY were always included in GPEC's highest-rated moves (top scores). However, play was not always optimal (for example, see second move) since other, sub-optimal, good moves also received high scores. Since there are always 36 or more possible moves, it is highly unlikely that such a result would stem from mere chance.

We hereby discuss some of GPEC's tactics, and and relevant terminals that may effect them:

- Capturing pieces (also known as *material*) is an integral part of any chess-playing program's strategy. Indeed, one might even construct a strong chess program based solely on material considerations and deep lookahead. However, since blindly capturing pieces is far from being a perfect strategy, an important test to a program's playing strength is its ability to avoid capturing "poisoned" pieces (eventually leading to losing the game). Knowing that capturing a piece is wrong typically requires tapping more elaborate

Table 6. Top-scoring possible moves for White in sample game, along with scores assigned by CRAFTY and GPEC. Each column represents the best options for White for the given moves (according to GPEC). All moves not appearing in the table were assigned a score of 0.0 by GPEC. Moves for Black are not included since the number of possible moves is very small. Moves played by GPEC are shown in **bold**. The bottom lines show the total number of possible moves for this position, and Black's reply. This game was only six moves long.

Move1	CRAFTY	GPEC
Qd3+	mate-in-9	6.0
Qf3+	mate-in-9	6.0
Qg4+	6.75	6.0
Qe1+	6.7	5.0
Qb1+	0.0	4.0
possible:	39 moves	
Black:	Ke5	

Move2	CRAFTY	GPEC
Ra5+	mate-in-8	6.0
Qf5+	mate-in-8	6.0
Qe3+	mate-in-13	6.0
Qxc4	6.7	4.0
Qg3+	0.0	3.0
Possible:	37 moves	
Black:	Kd5	

Move3	CRAFTY	GPEC
Rd3+	mate-in-12	6.0
Ra6+	6.05	5.0
Qb6+	6.8	4.0
Qd3+	0.0	2.0
Qg3+	0.0	2.0
Possible:	39 moves	
Black:	Kc6	

Move4	CRAFTY	GPEC
Qe6+	mate-in-11	4.0
Qe8+	mate-in-11	4.0
Qf3+	mate-in-15	0.0
Possible:	36 moves	
Black:	Kc5	

Move5	CRAFTY	GPEC
Qd5+	mate-in-7	5.0
Qf5+	6.5	5.0
Qe3+	mate-in-8	5.0
Qc8+	mate-in-13	4.0
Possible:	40 moves	
Black:	Kb6	

Move6	CRAFTY	GPEC
Rb3+	mate-in-4	6.0
Qxc4	6.05	5.0
Qd6+	6.8	4.0
Qe6+	0.0	2.0
Qd8+	0.0	2.0
Possible:	39 moves	
Black:	Ka6	

knowledge, or looking ahead deeper into the game tree. On the second move GPEC can capture the opponent's rook by Qxc4. This is a good move (scored 6.7 by CRAFTY), not a blunder, but still sub-optimal since there exist other moves leading to mate-in-8 and mate-in-13. GPEC awarded 6.0 points to the two optimal moves (leading to mate-in-8) and to Qe3+, which is slightly sub-optimal (mate-in-13) and stochastically selected Qe3+. The capture move received only 4.0 points. Preferring a move leading to mate-in-13 over a strong capturing move is by no means a trivial achievement for a program with a lookahead of 1! To sum up, GPEC has learned

through emergent evolution the value of material (manifested in not losing pieces), but knows when it is less important. Relevant terminals are: EvaluateMaterial, IsMaterialIncrease, IsMyQueenNotAttacked, IsMyFork, OppPieceAttackedUnprot.

- GPEC has learned to identify the need to repeatedly check the opponent (most moves in the table are checking moves), which is usually crucial to maintaining the advantage. Still, there is much difference between various checking moves (noted both by GPEC's and CRAFTY's varying scores to these moves). Related terminals: IsOppKingInCheck, IsOppKingBehindPiece, IsMate.
- Cornering the opponent's king was an important factor in GPEC's decision. As can be seen in the table, moves leaving less free squares to the opponent's king received higher scores (this is reflected in the NumNotMovesOpponentKing terminal's output). While this tactic is important when trying to deliver a mate, GPEC correctly avoids such moves that jeopardize the attacking piece. Moreover, GPEC still differentiated between cornering moves, assigning higher scores to moves leading to relatively closer mates (see moves 4, 5, 6). More relevant terminals: IsOppKingStuck, OppKingProximityToEdges, IsOppKingBehindPiece.
- GPEC preferred to attack with the queen, instead of the rook (though sometimes the rook is selected to attack). This was correct in various positions (as can be seen in CRAFTY's scores). The queen is also more capable of delivering forks. Relevant terminals: ValueMyPiecesAttacking, IsMyFork.

The list given here is only partial, but aids in grasping some of the complicated considerations involved in our player's decisions.

It should be mentioned that our evolved player played extremely well, though not perfectly. The main flaw was the lack of diversity in scoring—GPEC typically assigned a score of 0.0 to most non-winning moves. It is true that identifying the winning moves is usually sufficient to win, but when playing on the losing side it is still important to delay the loss as much as possible, as the opponent may make a mistake. As we know that GPEC can differentiate near-mates from further ones, this was not utilized while playing the losing side.

5. Strategic Emergence

In the previous section we witnessed a strong (though not perfect) level of play presented by the evolved individual. Now, we turn to examining the emergent aspects of this skill. First, we will try to break it down and show that it is more than the sum of its components (GP-tree terminals). Then, we will gradually construct the strongest players we can, using increasingly large groups of these "parts," and test their performance.

As several terminals were deemed relevant to our player's strategies, we first turn to examining them in isolation.

5.1. *Single terminals*

The "atoms" of our individual are single terminals. The basic method for calculating scores is the same as the one we used in [12]: 1 point is awarded per win, and 0.5 points per draw (as in chess tournaments)—averaged across 500 games. Thus, an overall value of 1 would be a perfect score, and 0.5 would mean playing at a level more or less equal to the opponent's. The score against CRAFTY and against MASTER is averaged to receive the overall final score. We assessed the playing strength of each terminal, using three measures:

(1) First, we examined the playing strength of individuals containing only the given terminal in their evaluation function (playing vs. CRAFTY and MASTER). [b] Each terminal was assigned a score reflecting its performance, marked S_{SINGLE}. Since 0.16 is the score for random evaluation functions in our experiments, terminals that score 0.16 (or less) presented zero (or even negative) playing strength.
(2) For the second measure, marked S_{DIS}, we "handicapped" several strong endgame players we developed (including GPEC190), by disabling the given terminal (altering its function to return either a random number or zero whenever it was called, instead of the regular output). The scores reflect the average *decrease* in performance when the given terminals were thus disabled.
(3) The third measure was the result of sets of experiments (containing 10 runs for each terminal), in which we evolved individuals containing all terminals listed *except for* the given terminal. Under this condition, the strongest individuals possible were evolved. We averaged their performance, and subtracted it from 0.485, which is GPEC's score (to reflect the fact that if stronger individuals were evolved without the terminal, it is probably less influential). This score is presented as the last measure [c], marked S_{NO}.

The *Strength* of each terminal was computed as the average of all three measures: $Strength = \frac{1}{3} \cdot (S_{SINGLE} + S_{DIS} + S_{NO})$.

Results are summarized in Table 7. As can be seen, the contribution of each "atom" to GPEC's overall success—even when measured in multiple ways—is relatively small. As noted above, S_{SINGLE} scores below 0.16 mean that the terminal is, in and of itself, worse than a random function (although a random function would score 0 on the 2nd and 3rd measures). As some terminals used by GPEC190 (for example, OppKingStuck) received zero or *negative* S_{SINGLE} scores, it is highly likely that using these terminals is non-trivial for evolution with the full terminal set.

[b] It was possible to test out terminals this way (the entire evaluation function being the terminal), since they are all implemented such that they return larger values for better properties of the board. For example, compare NotMyKingInCheck (returning 1 when the player's king is *not* attacked, and 0 otherwise) to OppKingInCheck (returning 1 when the opponent's king *is* attacked).
[c] For some terminals, this score is less significant since there are other terminals with highly similar functionality available to evolution. For example: MaterialValue and IsMaterialIncrease.

Table 7. We present the 12 most influential single terminals, sorted by overall *Strength* score (right column). This score was derived from 3 measures: S_{SINGLE}, the average score against CRAFTY and MASTER when using only the given terminal in the evaluation function, after subtracting 0.16 (random evaluation function score); S_{DIS}, the average decrease in performance of several strong players when the given terminal is still present—but disabled (returns a random value); S_{NO}, the average score for the best evolved individuals when evolving with all terminals *except* the given one, subtracted from 0.5, which is the playing strength required to draw with CRAFTY. The overall *Strength* of the terminal is the average of these 3 measures.

Terminal	S_{SINGLE}	S_{DIS}	S_{NO}	Strength
NotMyPieceAttackedUnprot	0.26	0.131	0.10	0.164
IsMateInOne	0.16	0.105	0.15	0.138
NotMyPieceAttacked	0.32	0.010	0.04	0.123
NumMyPiecesNotAttacked	0.30	0.008	0.06	0.123
MaterialValue	0.18	0.056	0.11	0.115
IsMate	0.16	0.08	0.10	0.113
IsMaterialIncrease	0.19	0.021	0.12	0.110
OppKingStuck	0.14	0.048	0.12	0.103
OppKingProximityToEdges	0.16	0.106	0.02	0.096
IsMyFork	0.16	0.024	0.05	0.078
NotMyKingStuck	0.16	0.027	0.03	0.073
OppKingInCheck	0.04	0.027	0.10	0.057

Another interesting thing to note is that the terminal NumNotMovesOppKing, which is clearly an integral part of GPEC190's strategy (due to the apparent closeness of GPEC's evaluation scores and those of this terminal) did not even make it to the top 12 (it is ranked only 15th). Also, IsMateInOne (ranked 2nd) is not used by GPEC190, and several other strong players.

We conclude that single terminals are weak and insufficient to explain the overall playing strength of a full-blown evolved strategy, even if some diversity can be seen in their Strength measures.

5.2. *Terminal pairs*

We turn to examining small "molecules" containing 2 atoms each: we selected pairs of strong terminals—the top-ranking ones from Table 7 (except that we avoided pairing similar terminals, such as NotMyPieceAttackedUnprot and NotMyPieceAttacked)—and attempted to reach the maximal level of play attainable with these pairs. This was accomplished by evolving GP individuals using only one pair of terminals (per experiment), and all functions from the function set (see Tables 1, 2 and 3). The depth of the GP-trees was bound by 4.

Results appear in Table 8, reflecting the best evolved individual's playing strength against CRAFTY and MASTER for each pair. As can be seen in this table, combining two elements does not necessarily yield a better result. Sometimes

Table 8. Scores for pairs of several top-ranking terminals. The top table lists the terminals used, along with S_{SINGLE} scores (for reference). Note that the terminals are sorted according to overall Strength, and not S_{SINGLE} scores. In the bottom table, column i shows the scores of the terminal at row i, paired with all those in the rows above it. For example, the value at column 5, row 3 (0.185) is the score when the 3rd and 5th terminals together—MaterialValue and OppKingProximityToEdges—comprise the evaluation function. Scores are actual points awarded for the strongest individuals in each run.

Index	Terminal	S_{SINGLE}
1	NotMyPieceAttackedUnprot	0.26
2	IsMateInOne	0.16
3	MaterialValue	0.18
4	OppKingStuck	0.14
5	OppKingProximityToEdges	0.16
6	IsMyFork	0.16
7	OppKingInCheck	0.04

Index	2	3	4	5	6	7
1	0.23	0.217	0.212	0.232	0.252	0.26
2		0.191	0.149	0.151	0.194	0.04
3			0.19	0.185	0.178	0.18
4				0.169	0.146	0.174
5					0.15	0.05
6						0.164

the scores for the pair were higher than each of the individual terminals comprising it (for example, MaterialValue and OppKingStuck combined, received a score of 0.19, which is higher than their separate scores), but mostly this did not occur. The score of the terminal NotMyPieceAttackedUnprot, which was 0.26, was not improved with any combination of any single terminal (and was often even hampered).

Thus, it may be observed that emergence did not occur here—the road to improving individual terminals' performance lies far beyond simply pairing them with other strong terminals, even when many combinations are tried by evolution.

5.3. *Variant terminal groups*

In light of the experiments described above, in order to better understand the connection between the number of "atoms" or "molecules" to the player's performance, and to partially answer the questions regarding the linearity of improvement, we conducted a final series of experiments. Here the size of individuals was allowed to grow up to full-scale "poly-molecules" ("cells"), comprising all terminals. This way, the increase in playing level (and emergent aspects) may be observed as we move from simple units to more complex ones.

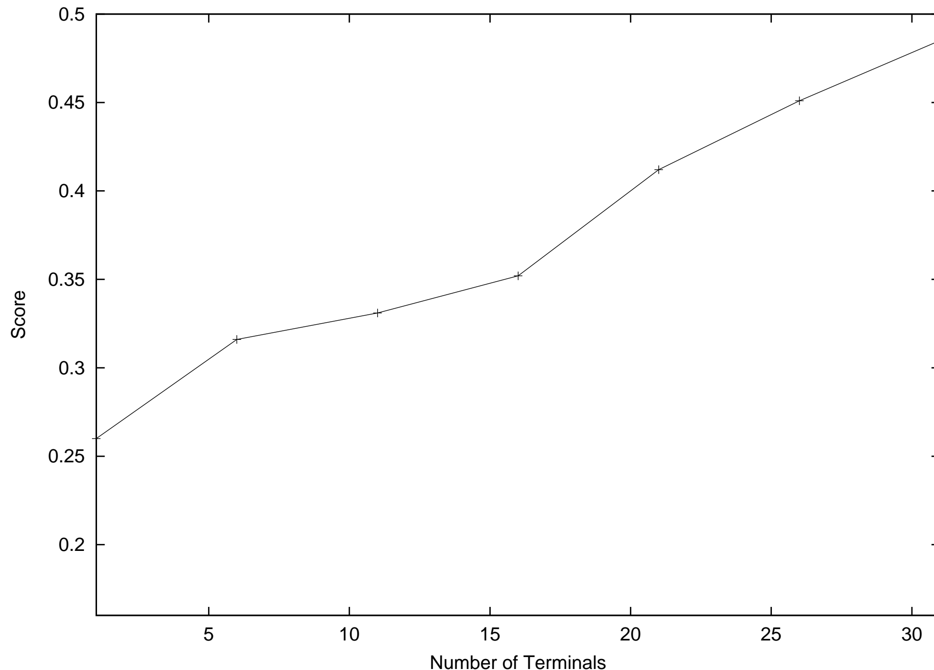

Fig. 5. Best results obtained by evolution using an increasing number of terminals.

This was accomplished by increasing the size, S, of the terminal group in each set of experiments. Starting from $S = 1$, for each S in $\{1, 6, 11, ..31\}$ (31 being the total number of different terminals), we manually select S terminals and conduct an experiment (evolutionary process) using only these terminals, with all functions available. Since weaker terminals may be randomly selected, and evolutionary runs, being partially stochastic, may sometimes fail to come up with good solutions, we repeated the process several times for each S.

After each run the strongest individual's score against both CRAFTY and MASTER was averaged for all runs (with the same S-value). Since improvement with larger groups of terminals had been more difficult, several small "cells" were constructed by hand (e.g., a few functions containing a good combination of two or more terminals were constructed), and added to the process. Results appear in Figure 5.

It is important to note that although the graph seems quasi-linear, improvement is strictly non-linear, since as the level of play increases, improvement becomes more difficult. Indeed, reaching the higher scores (in the experiments with more terminals), took considerably more time and computational effort.

6. Concluding Remarks and Future Work

After describing the experiments that gave rise to strong evolved chess endgame players we analyzed them in several empirical ways. Moreover, we examined the emergent capabilities of evolved individuals, primarily in the sense that their knowledge of the game (reflected in their scores) transcended the knowledge that was infused into them.

We started by breaking up a strategy into its comprising parts, and examining the parts' effect in several ways. As expected, simple functions did not perform well on their own. However, their effect was more pronounced when removed (the S_{NO} measure); for example, while the IsMateInOne terminal, on its own, scored 0.16 (the same score as the random function), when it was disabled in strong players, their scores decreased on average by $S_{DIS} = 0.105$, which is a strong effect on playing level (as noted in the previous section, when a player's level is high, competition becomes harsh, and every point counts).

Pairs of terminals did not prove to be much of an improvement. Although for now we only checked the pairs' scores (and did not conduct the more elaborate testing we did with single terminals), we were still surprised by the difficulty in joining strong terminals together correctly to use with the entire function set, even in such small groups.

As a result, considerably more computational effort was put into constructing the variant terminal groups. In addition, evolution was aided with several functions specifically tailored for this experiment. While this helped evolution converge faster, it may have diverted it towards local maxima. More time and effort is needed to ascertain whether evolution may find better solutions, comprised of smaller parts.

All in all, we gained insight into the emergent aspects of our evolving players' capabilities. An important conclusion is that the "leap" in performance occurs somewhere around 21 terminals, since the level of play presented by players with more terminals surpassed the capabilities of MASTER, which was constructed by hand, and represents our best non-evolutionary improvement.

The methodology used herein may also be applied to other domains. Since GP is widely used throughout the academic world, as well as in numerous industrial applications, emergent qualities of solutions crafted by GP may be examined by following the steps described herein: determining the strength (defined according to the task domain) of each member of the terminal and function sets, as well as combinations of these elements thereof, which in turn may shed light on the transcendence from simple atoms to fully functioning solutions.

In the future we intend to tackle more endgames. Also, we intend to introduce more game-tree search (deeper than the lookahead of 1 used in these experiments) to the process, and examine its effect on playing strengths of groups of terminals of varying sizes. As contemporary search engines present strong playing level using deep search and little knowledge, we feel that combining search with knowledge will yield even better insights into the emergent aspect of the intelligence presented by

artificial chess players.

References

[1] P. J. Angeline and J. B. Pollack. Competitive environments evolve better solutions for complex tasks. In *Proceedings of the 5th International Conference on Genetic Algorithms (GA-93)*, pages 264–270, 1993.

[2] M. Bain. *Learning Logical Exceptions in Chess.* PhD thesis, University of Strathclyde, Glasgow, Scotland, 1994.

[3] G. Bonanno. The logic of rational play in games of perfect information. Papers 347, California Davis - Institute of Governmental Affairs, 1989. available at http://ideas.repec.org/p/fth/caldav/347.html.

[4] M. Campbell, A. J. Hoane, Jr., and F.-H. Hsu. Deep blue. *Artificial Intelligence*, 134(1–2):57–83, 2002.

[5] C. F. Chabris and E. S. Hearst. Visualization, pattern recognition, and forward search: Effects of playing speed and sight of the position on grandmaster chess errors. *Cognitive Science*, 27:637–648, 2003.

[6] N. Charness. Expertise in chess: The balance between knowledge and search. In K. A. Ericsson and J. Smith, editors, *Toward a general theory of Expertise: Prospects and limits.* Cambridge University Press, Cambridge, 1991.

[7] G. J. Ferrer and W. N. Martin. Using genetic programming to evolve board evaluation functions for a board game. In *1995 IEEE Conference on Evolutionary Computation*, volume 2, pages 747–752, Perth, Australia, 1995. IEEE Press.

[8] D. Fogel, T. J. Hays, S. Hahn, and J. Quon. A self-learning evolutionary chess program. In *Proceedings of the IEEE*, volume 92:12, pages 1947–1954. IEEE Press, 2004.

[9] P. W. Frey. *Chess Skill in Man and Machine.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1979.

[10] J. Fürnkranz. Machine learning in computer chess: The next generation. *International Computer Chess Association Journal*, 19(3):147–161, 1996.

[11] R. Gross, K. Albrecht, W. Kantschik, and W. Banzhaf. Evolving chess playing programs. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 740–747, New York, 9-13 2002. Morgan Kaufmann Publishers.

[12] A. Hauptman and M. Sipper. GP-endchess: Using genetic programming to evolve chess endgame players. In M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 120–131, Lausanne, Switzerland, 2005. Springer.

[13] J. H. Holland. *Adaptation in natural artificial systems.* University of Michigan Press, Ann Arbor, 1975.

[14] A. X. Jiang and M. Buro. First experimental results of ProbCut applied to chess. In *Proceedings of 10th Advances in Computer Games Conference*, pages 19–32. Kluwer Academic Publishers, Norwell, MA, 2003.

[15] G. Kendall and G. Whitwell. An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 995–1002, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 2001. IEEE Press.

[16] J. R. Koza. *Genetic programming: On the Programming of Computers by Means of*

Natural Selection. MIT Press, Cambridge, MA, 1992.

[17] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, 1994.

[18] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, Norwell, MA, 2003.

[19] T. A. Marsland and M. S. Campbell. A survey of enhancements to the alpha-beta algorithm. In *Proceedings of the ACM National Conference*, pages 109–114, 1981.

[20] T. A. Marsland, A. Reinefeld, and J. Schaeffer. Research note: Low overhead alternatives to sss. *Artificial Intelligence*, 31:185–199, 1987.

[21] M. Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, Cambridge, MA, 1996.

[22] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[23] L. A. Panait and S. Luke. A comparison of two competitive fitness functions. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 503–511, New York, 2002. Morgan Kaufmann Publishers.

[24] A. Reinefeld. *Spielbaum-Suchverfahren*. Springer, Berlin, Heidelberg, 1989.

[25] A. L. Samuel. Machine learning. *Technology Review*, 62:42–45, 1959.

[26] E. Sanchez and M. Tomassini, editors. *Evolutionary Algorithms in Towards Evolvable Hardware, The Evolutionary Engineering Approach, Papers from an international workshop, Lausanne, Switzerland, October 2-3, 1995*, volume 1062 of *Lecture Notes in Computer Science*. Springer, 1996.

[27] J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Trans. Pattern Anal. Mach. Intell.*, 11(11):1203–1212, 1989.

[28] M. Sipper. *Machine Nature: The Coming Age of Bio-Inspired Computing*. McGraw-Hill, New York, 2002.

[29] G. L. Steele Jr. *CommonLisp the Language*. Digital Press, second edition, 1990.