Evolving Board-Game Players with Genetic Programming

Amit Benbassat
Dept. of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel
amitbenb@cs.bgu.ac.il

Moshe Sipper
Dept. of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel
sipper@cs.bgu.ac.il

ABSTRACT

We present the application of genetic programming (GP) to zero-sum, deterministic, full-knowledge board games. Our work expands previous results in evolving board-state evaluation functions for Lose Checkers to a 10x10 variant of Checkers, as well as Reversi. Our system implements strongly typed GP trees, explicitly defined introns, and a selective directional crossover method.

Categories and Subject Descriptors

I.2.6 [**Parameter learning**]: Knowledge acquisition; I.2.1 [**Applications and Expert Systems**]: Games; I.2.8 [**Problem Solving, Control Methods, and Search**]: Heuristic methods

General Terms

Design

Keywords

Genetic programming, games, board games, alpha-beta search

1. INTRODUCTION

Developing players for board games has been part of AI research for decades. Board games have precise, easily formalized rules that render them easy to model in a programming environment. In this work we will focus on full knowledge, deterministic, zero-sum board games, expanding on our previous results in Lose Checkers [1].

We apply tree-based GP to evolving players for a number of games. Our guide in developing our design, aside from previous research games and GP, is nature itself. Evolution by natural selection is first and foremost nature's algorithm, and as such will serve as a source for ideas. Though it is by no means assured that an idea that works in the natural world will work in our synthetic environment, it can be seen as evidence that it might. We are mindful of evolutionary theory, particularly as pertaining to the gene-centered view of evolution. This view, presented by Williams [21] and expanded upon by Dawkins [3], focuses on the gene as the unit of selection. It is from this point of view that we consider how to adapt the ideas borrowed from nature into our synthetic GP environment.

GECCO'11, July 12–16, 2011, Dublin, Ireland.

2. THE GAMES

The games we explore in this work—Lose Checkers, 10x10 Checkers, and Reversi—are zero-sum, deterministic, full-knowledge, two-player board games played on an nXn board for some given n.

We worked on two variants of Checkers. The first was the 8x8 Lose Checkers variant, which is similar to the popular American Checkers game but has a different objective—to lose all of one's pieces. The second variant was an expansion of the rule set of American Checkers to a 10x10 board, creating a significantly higher branching factor and higher game complexity (due, in addition to the increased branching factor, to the existence of 30 pieces instead of 24 on the opening board).

Reversi, also known as Othello, is a popular game with a rich research history [13, 7, 11]. Though a board game played on an 8x8 board, it differs widely from the Checkers variants in that it is a piece-placing game rather than a piece-moving game. In Reversi the number of pieces on the board increases during play, rather than decreasing as it does in Checkers. The number of moves (not counting the rare pass moves) in Reversi is limited by the board's size, making it a short game.

3. PREVIOUS WORK

In the years since Strachey [20] first designed a Checkers-playing algorithm, there has been much work on Checkers-playing computer programs. Notable progress was made by Samuel [15, 16], the first to use machine learning to create a competent Checkers-playing computer program. Samuel's program managed to beat a competent human player in 1964. In 1989 a team of researchers from the University of Alberta led by Jonathan Schaeffer began working on an American Checkers program called Chinook. By 1990 Chinook's level of play was comparable to that of the best human players. Chinook continued to grow in strength, establishing its dominance [18]. In 2007, Schaeffer et al. [17] solved Checkers and became the first to completely solve a major board game.

To date, there has been limited research interest in Lose Checkers, all of it quite recent [5, 19]. This work concentrates either on search [5] or on finding a good evaluation function [19]. The 8x8 variant of Reversi has received its fair share of research attention. Early landmark work by Rosenbloom [13] yielded IAGO, an expert level Reversi program. Subsequent work by Lee and Mahajan [7] greatly improved on IAGO's level of play by utilizing Bayesian learning to improve the player's evaluation function. The evolutionary approach has been applied to Reversi by several researchers. Moriarty and Miikkulainen [11] evolved artificial neural networks (ANNs) using the marker-based approach to evolve a

Table 1: Domain-specific terminal nodes. F: floating point, B: boolean.

Node name	Type	Return value
EnemyManCount()	F	The enemy's man count
FriendlyManCount()	F	The player's man count
ManCount()	F	FriendlyManCount() – EnemyManCount()
Mobility()	F	The number of plies available to the player
IsEmptySquare(X,Y)	B	True iff square empty
IsFriendlyPiece(X,Y)	B	True iff square occupied by friendly piece
IsManPiece(X,Y)	B	True iff square occupied by man

Table 2: Function nodes. F_i: floating-point parameter, B_i: boolean parameter.

Node name	Type	Return value
LowerEqual(F_1,F_2)	B	True iff $F_1 \leq F_2$
NOTG(B_1,B_2)	B	Logical NOT of B_1
IfTrue(B_1,F_1,F_2)	F	F_1 if B_1 is true and F_2 otherwise
Minus(F_1,F_2)	F	$F_1 - F_2$
MultERC(F_1)	F	F_1 times random number
NullJ(F_1,F_2)	F	F_1
Plus(F_1,F_2)	F	$F_1 + F_2$

highly competent Revresi player program that does not use search. Eskin and Siegel presented some preliminary work in applying GP to Reversi without using search [4]. Chong et al. [2] presented a program using shallow search, with evolved feed-forward ANNs encoded with board-spatial features as its board evaluation function.

4. EVOLUTIONARY SETUP

The individuals in the population act as board-evaluation functions, to be combined with a standard game-search algorithm. The value they return for a given board state is seen as an indication of how good that board state is for the player whose turn it is to play. The evolutionary algorithm was written in Java. We chose to implement a strongly typed GP framework [10] supporting a boolean type and a floating-point type. Support for a multi-tree interface was also implemented. On top of the basic crossover and mutation operators described by Koza [6], another form of crossover was implemented—which we designated "one-way crossover"—as well as a local mutation operator. The original setup is detailed in [1]. Its main points and recent updates are detailed below.

To achieve good results on multiple games using deeper search we enhanced our system with the ability to run in parallel multiple threads. The basic GP tree nodes used for the games are presented in Tables 1 and 2 (also supported by our setup but not presented in Table 2 for lack of space are four binary boolean functions—AND, $NAND$, NOR, and OR). The NOTG and NULLJ function nodes both contain explicitly defined introns (described below). On top of these some basic terminal node were used, including the boolean values $true$ and $false$ as well as the numeric values 0 and 1, and an ephemeral random constant (ERC) chosen at random from $[-5.0, 5)$.

4.1 One-Way Crossover

One-way crossover does not consist of two individuals swapping parts of their genomes but rather of one individual inserting a copy of part of its genome into another individual, without receiving any genetic information in return. In our case, the one-way crossover is done by randomly selecting a subtree in both participating individuals, and then inserting a copy of the selected subtree from the first individual in place of the selected subtree from the second individual.

This type of crossover operator is uni-directional. There is a donor and a receiver of genetic material. In this work, we utilized this directionality to make crossover more than a random operator by always choosing the individual with higher fitness to act as the donor in one-way crossover. This sort of nonrandom genetic operator favors the fitter individuals as they have a better chance of surviving it. Algorithm 1 shows the pseudocode representing how crossover is handled in our setup. As can be seen, one-way crossover is expected to be chosen at least half the time, giving the fitter individuals a survival advantage, but the fitter individuals can still change due to the standard "two-way" crossover.

Algorithm 1 Crossover.

```
Randomly choose two different previously unselected individuals from population for crossover: I1 and I2
if I1.Fitness ≥ I2.Fitness then
    Perform one-way crossover with I1 as donor and I2 as receiver
else
    Perform two-way crossover with I1 and I2
end if
```

Using the vantage point of the gene-centered view of evolution it is easier to see the logic of crossover in our setup. In a gene-centered world, we look at genes as competing with each other, the more effective ones out-reproducing the rest. This, of course, should already happen in a setup using the generic two-way crossover alone; our approach strengthens this trend. The individuals with high fitness that are more likely to get chosen as donors in one-way crossover, are also more likely to contain more good genes than the less-fit individuals that get chosen as receivers. This genetic operator thus causes an increase in the frequency of the genes that lead to better fitness.

4.2 Explicitly Defined Introns

In natural living systems not all DNA has phenotypic effect. This non-coding DNA, sometimes referred to as Junk DNA, is prevalent in virtually all eukaryotic genomes. In GP, so-called introns are areas of code that do not affect survival and reproduction (usually this can be replaced with "do not affect fitness"). In the context of tree-based GP the term "areas of code" applies to subtrees.

Introns occur naturally in GP, provided that the function and terminal sets allow for it. As bloat progresses, the number of nodes that are part of introns tends to increase. Luke [9] focused on inviable code introns: Introns that cannot be replaced by anything that can possibly change the individual's operation. In [1] we made another distinction between two types of inviable code introns: Live-code introns, which though inviable may still be translated into code that will run, and dead-code introns, whose code is never run.

Explicitly defined introns (EDIs) in GP are introns that reside in an area of the genome specifically designed to hold introns. As the individual runs it will simply ignore these introns. In our setup, EDIs exist under every $NullJ$ and $NOTG$ node. In both functions the rightmost subtree does not affect the return value in any way.

This means that every instance of one of these function nodes in an individual's tree defines an intron, which is always of the dead-code type—a fact the program can take into account (by not generating any code for that subtree). In our setup, when converting individuals into C code, the EDIs are simply ignored, a feat that can be accomplished with ease as they are dead-code introns that are easy to find.

Our search of the literature discovered no exploration of EDIs in tree-based GP, but the prevalence of explicit introns in EA research as well as in junk DNA in nature suggested that this is an avenue worth exploring. Nordin et al. [12] explored EDIs in linear GP, finding that they tend to improve fitness and shorten runtime. Earlier work showed that using introns was also helpful in GAs [8].

4.3 Fitness Calculation

After initializing the fitness of all individuals in the population to 0, fitness calculation was carried out in the fashion described in Algorithm 2. Essentially, evolving players faced two types of opponents: external "guides" (described below) and their own cohorts in the population. The latter method of evaluation is known as coevolution [14], and is referred to below as the coevolution round.

Algorithm 2 Fitness evaluation.

// Parameter: $GuideArr$ – array of guide players
for $i \leftarrow 1$ *to* $GuideArr.length$ **do**
  for $j \leftarrow 1$ *to* $GuideArr[i].NumOfRounds$ **do**
    Every individual in population deemed fit enough plays $GuideArr[i].roundSize$ games against guide i.
  end for
end for
Every individual in the population plays $CoPlayNum$ games as black against $CoPlayNum$ random opponents in the population.

The method of evaluation described requires some information from the user, including the number of guides, their designations, and the number of co-play opponents for the coevolution round. All these are program run parameters that the program accepts from the user via parameter files. Tweaking these parameters allows for different setups.

Guide-Play Rounds. Two types of guides were implemented: A random player and an alpha-beta player. The random player chose a move at random. The alpha-beta player searched up to a preset depth in the game tree and used a hand-crafted evaluation function for states in which there was no clear winner.

Coevolution Rounds. In a co-play round each member of the population in turn played Black in a number of games equal to the parameter $CoPlayNum$ against $CoPlayNum$ random opponents from the population playing White. The opponents were chosen in a way that ensured that each individual also played exactly $CoPlayNum$ games as White.

When playing against a guide or against one of its cohorts, each player in the population received 1 point added to its fitness for every win, and 0.5 points for every draw.

4.4 Selection and Procreation

In the selection stage we used tournament selection with a tournament size of 2 to select the parents of the next generation from the population according to their fitness. In the procreation stage, genetic operators were applied to the parents in order to create the next generation.

During procreation every individual was chosen for crossover with probability p_{xo} and self-replicated with probability $1 - p_{xo}$.

Table 3: Lose Checkers: Relative levels of play for different benchmark (guide) players. Here and in the subsequent tables, $\alpha\beta i$ refers to an alpha-beta player using a search depth of i and a hand-crafted evaluation function.

1st Player	2nd Player	1st Player win ratio
$\alpha\beta 2$	random	0.9665
$\alpha\beta 3$	$\alpha\beta 2$	0.8502
$\alpha\beta 5$	$\alpha\beta 3$	0.82535
$\alpha\beta 7$	$\alpha\beta 5$	0.8478
$\alpha\beta 8$	$\alpha\beta 3$	0.5873
$\alpha\beta 5$	$\alpha\beta 8$	0.55620

Table 4: Lose Checkers: Results of top runs using shallow search. *Player* uses $\alpha\beta$ search of depth 4 coupled with evolved evaluation function, while *Benchmark Opponent* uses $\alpha\beta$ search of depth 5 coupled with a random evaluation function. Here and in the subsequent tables, the XCo notation in the Fitness Evaluation column implies that the $CoPlayNum$ parameter discussed in text was set to X. $Y\alpha\beta i$ implies that each individual played Y games against guide $\alpha\beta i$ during fitness evaluation. Here and in the subsequent tables, Benchmark Score is the result of having an evolved player play a 1000-game tournament against a benchmark opponent.

Run identifier	Fitness Evaluation	Benchmark Score
r00090	10$\alpha\beta$2+20Co	632.0
r00091	10$\alpha\beta$2+20Co	645.0
r00096	25Co	608.0
r00097	25Co	575.0
r00098	40Co	575.5
r00099	40Co	633.5

The implementation and choice of specific crossover operator is as in Algorithm 1. After crossover every individual underwent mutation with probability p_m.

5. RESULTS

Below we summarize our main novel results using the new multi-threaded system. We reported on previous results in [1]. The guides used in fitness evaluation also served as the post-evolutionary benchmark players. We evaluated our guide players by testing them against each other in matches of 10,000 games. The overall trend when increasing search depth was towards improvement in level of play, though in some cases and for some search depths the hand-crafted players did poorly. We avoided testing our players against these latter hand-crafted players.

Note that in all the results presented below the player does not specialize in playing the benchmark opponent but rather faces an ever-changing array of opponents among its cohorts in the population.

As there is no known simple board-evaluation function for Lose Checkers we used a random evaluation for nontrivial board states. Table 3 demonstrates the relative levels of play of the different hand-crafted players. The players that used search depths of 4 and 6 did poorly and we removed them from this table.

Our best runs to date for Lose Checkers are presented in Table 4. As the table demonstrates, our players, using a search depth of 4, were able to outperform the strong $\alpha\beta 5$ player.

For 10x10 checkers and Reversi we used the simple yet effective method of material evaluation (piece counting) to evaluate board states. We had hand-crafted players randomly alternate between

Table 5: 10x10 Checkers: Results of top runs using shallow search. *Player* uses $\alpha\beta$ search of depth 3 (runs 84, 85) or 2 (runs 92–95) coupled with evolved evaluation function, while *Benchmark Opponent* uses $\alpha\beta$ search of depth 3 coupled with a material evaluation function.

Run identifier	Fitness Evaluation	Benchmark Score
r00084	50Co	889.0
r00085	50Co	927.0
r00092	25Co	732.0
r00093	25Co	615.5
r00094	25Co	554.0
r00095	25Co	631.0

Table 6: Reversi: Results of top runs using shallow search. *Player* uses $\alpha\beta$ search of depth 4 coupled with evolved evaluation function, while *Benchmark Opponent* uses $\alpha\beta$ search of depths 5 and 7 coupled with a material evaluation function.

Run identifier	Fitness Evaluation	Benchmark Score vs $\alpha\beta 5$	Benchmark Score vs $\alpha\beta 7$
r00100	25Co	875.0	758.5
r00101	25Co	957.5	803.0
r00102	40Co	942.5	640.5
r00103	40Co	905.5	711.5
r00108	40Co	956.0	760.0
r00109	40Co	912.5	826.0
r00110	40Co	953.5	730.5
r00111	40Co	961.0	815.5

two different material evaluation functions in order to generate a playing strategy that was not entirely predictable. The average score of $\alpha\beta 2$ playing 10x10 Checkers against the random player was 0.99885. $\alpha\beta 3$ scored an average of 0.5229 against $\alpha\beta 2$.

Our best runs to date for 10x10 Checkers are presented in Table 5. As the table demonstrates our players were able to outperform the $\alpha\beta 3$ player using a search depth of 2 (runs 92–95), and to overwhelm it using a search depth of 3 (runs 84–85).

The average score of $\alpha\beta 2$ playing Reversi against the random player was 0.8471. $\alpha\beta 3$ scored an average of 0.6004 against $\alpha\beta 2$. $\alpha\beta 5$ scored an average of 0.7509 against $\alpha\beta 3$. $\alpha\beta 7$ scored an average of 0.7509 against $\alpha\beta 5$.

Our best runs to date for Reversi are presented in Table 6. As the table demonstrates, our players were able to outperform the $\alpha\beta 5$ and $\alpha\beta 7$ players using a search depth of 4.

6. CONCLUSIONS

Expanding on previous work [1] we presented the genetic programming approach as a tool for discovering effective strategies for playing zero-sum, deterministic, full-knowledge board games. Guided by the gene-centered view of evolution, we introduced several new ideas and adaptations of existing ideas for augmenting the GP approach. Having evolved successful players, we established that tree-based GP is applicable to board-state evaluation in three distinct nontrivial board games.

7. ACKNOWLEDGMENTS

Amit Benbassat is partially supported by the Lynn and William Frankel Center for Computer Sciences.

8. REFERENCES

[1] A. Benbassat and M. Sipper. Evolving lose-checkers players using genetic programming. In *IEEE Conference on Computational Intelligence and Games (CIG'10)*, pages 30–37, August 2010.

[2] S. Chong, D. Ku, H. Lim, M. Tan, and J. White. Evolved neural networks learning othello strategies. In *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, volume 3, pages 2222 – 2229 Vol.3, December 2003.

[3] R. Dawkins. *The Selfish Gene*. Oxford University Press, Oxford, UK, 1976.

[4] E. Eskin and E. Siegel. Genetic programming applied to othello: introducing students to machine learning research. *SIGCSE Bull.*, 31:242–246, March 1999.

[5] M. Hlynka and J. Schaeffer. Automatic generation of search engines. In *Advances in Computer Games*, pages 23–38, 2006.

[6] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[7] K.-F. Lee and S. Mahajan. The development of a world class othello program. *Artificial Intelligence*, 43(1):21 – 36, 1990.

[8] J. R. Levenick. Inserting introns improves genetic algorithm success rate: Taking a cue from biology. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 123–127. Morgan Kaufmann, 1991.

[9] S. Luke. Code growth is not caused by introns. In D. Whitley, editor, *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, pages 228–235, Las Vegas, Nevada, USA, July 2000.

[10] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3:199–230, 1993.

[11] D. E. Moriarty and R. Miikkulainen. Discovering complex Othello strategies through evolutionary neural networks. *Connection Science*, 7(3):195–210, 1995.

[12] P. Nordin, F. Francone, and W. Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 6–22, 1996.

[13] P. S. Rosenbloom. A world-championship-level othello program. *Artificial Intelligence*, 19(3):279 – 320, 1982.

[14] T. P. Runarsson and S. M. Lucas. Coevolution versus self-play temporal difference learning for acquiring position evaluation in small-board Go. *IEEE Transactions on Evolutionary Computation*, 9(6):628–640, 2005.

[15] A. L. Samuel. Some studies in machine learning using the game of Checkers. *IBM Journal of Research and Development*, 3(3):210–229, July 1959.

[16] A. L. Samuel. Some studies in machine learning using the game of Checkers II - recent progress. *IBM Journal of Research and Development*, 11(6):601–617, 1967.

[17] J. Schaeffer, N. Burch, Y. Bjornsson, A. Kishimoto, M. Muller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[18] J. Schaeffer, R. Lake, P. Lu, and M. Bryant. Chinook: The world man-machine checkers champion. *AI Magazine*, 17(1):21–29, 1996.

[19] M. Smith and F. Sailer. Learning to beat the world Lose Checkers champion using TDLeaf(λ), December 2004.

[20] C. S. Strachey. Logical or nonmathematical programming. In *ACM '52: Proceedings of the 1952 ACM national meeting (Toronto)*, pages 46–49, 1952.

[21] G. Williams. *Adaptation and Natural Selection*. Princeton University Press, Princeton, NJ, USA, 1966.