Evolving Lose-Checkers Players using Genetic Programming

Amit Benbassat and Moshe Sipper

***Abstract*— We present the application of genetic programming (GP) to the zero-sum, deterministic, full-knowledge board game of Lose Checkers. Our system implements strongly typed GP trees, explicitly defined introns, local mutations, and multi-tree individuals. Explicitly defined introns in the genome allow for information selected out of the population to be kept as a reservoir for possible future use. Multi-tree individuals are implemented by a method inspired by structural genes in living organisms, whereby we take a single tree describing a state evaluator and split it.**

I. 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 the full knowledge, deterministic, zero-sum game of Lose Checkers.

We apply tree-based GP to evolving players for Lose Checkers. Our guide in developing our algorithm parameters, aside from previous research into 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 [4], 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.

II. Lose Checkers

Many variants of the game of Checkers exist, several of them played by a great number of people (including tournament play). Practically all Checkers variants are two-player games that contain only two types of pieces set on an $n \times n$ board. The most well-known variant of Checkers is American Checkers. It offers a relatively small search space (roughly 10^{20} legal positions compared to the 10^{43}–10^{50} estimated for Chess) with a relatively small branching factor. It is fairly easy to write a competent[1] computer player for American Checkers using minimax search and a trivial evaluation function.[2] American Checkers shares its domain with another, somewhat less-popular variant of Checkers, known as Lose Checkers. The basic rules of Lose Checkers are the same as American Checkers (though the existence of different organizations may cause some difference in the peripheral rules). The objective, however, is quite different. A losing position for a player in American Checkers is a winning position for that player in Lose Checkers and vice versa (i.e., one wins by losing all pieces or remaining with no legal move). Hlynka and Schaeffer [7] observed that, unlike the case of American Checkers, Lose Checkers lacks an intuitive state evaluation function. In some cases Lose Checkers computer players rely solely on optimized deep search and an endgame state database, having the evaluation function return a random value for states not in the database.

III. Previous Work

In the years since Strachey [20] first designed an American Checkers-playing algorithm, there has been some work on Checkers-playing computer programs. Notable progress was made by Samuel [15, 16], who was 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 it was clear that Chinook's level of play was comparable to that of the best human players when it won second place in the U.S. Checkers championship without losing a single game. Chinook continued to grow in strength, establishing its dominance [17]. In 2007, Schaeffer et al. [18] solved Checkers and became the first to completely solve a major board game.

Games attract considerable interest from AI researchers. The field of evolutionary algorithms is no exception to this rule. Over the years many games have been tackled with the evolutionary approach. A GA with genomes representing artificial neural networks (ANNs) was used in 1995 by Moriarty and Miikkulainen [12] to attack the game of Othello, resulting in a competent player that employed sophisticated mobility play. ANN-based American Checkers players were evolved by Chellapilla and Fogel [2, 3] using a GA, their long runs resulting in expert-level play. GP was used by Azaria and Sipper [1] to evolve a strong Backgammon player. GP research by Hauptman and Sipper produced both competent players for Chess endgames [5] and an efficient solver for the Mate-in-N problem in Chess [6].

The authors are with the Dept. of Computer Science, Ben-Gurion University, Beer-Sheva, Israel. Email: {amitbenb,sipper}@cs.bgu.ac.il.

[1] In this work we use "competent" to describe players that show a level of skill in their play comparable to some human players (i.e., are not trivially bad) and yet do not exhibit the level of play of the strongest players (be they computer or human) available. As it is often hard to compare levels of play between different games, we find this fluid definition of "competence" to be suitable.

[2] The generic evaluation function for Checkers is a piece differential that assigns extra value to kings on the board. This sort of player was used by Chellapilla and Fogel [2] to test their own evolved player.

TABLE I

BASIC TERMINAL NODES. F: FLOATING POINT, B: BOOLEAN.

Node name	Return type	Return value
ERC()	F	Preset random number
False()	B	Boolean $false$ value
One()	F	1
True()	B	Boolean $true$ value
Zero()	F	0

To date, there has been limited research interest in Lose Checkers, all of it quite recent [7, 19]. This work concentrates either on search [7] or on finding a good evaluation function [19]. Though both of these give rise to strong players, they can also be seen as preliminary attempts that offer much room for improvement. The mere fact that it is difficult to hand-craft a good evaluation function for Lose Checkers allows for the claim that any good evaluation function is in fact human competitive. If capable human programmers resort to having their evaluation function return random values, then any improvement on random is worth noting.

IV. EVOLUTIONARY SETUP

The individuals in the population act as board-evaluation functions, to be combined with a standard game-search algorithm (e.g., alpha-beta). 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 [11]. The two types implemented in code are the boolean type and a floating-point type. Support for a multi-tree interface was also implemented. We implemented the basic crossover and mutation operators described by Koza [8]. On top of this, another form of crossover was implemented—which we designated "one-way crossover"—as well as a local mutation operator. The setup is detailed below.

A. Basic Terminal Nodes

Several basic domain-independent terminal nodes were implemented. These nodes are presented in Table I.

The only node in Table I that requires further explanation is the ERC (Ephemeral Random Constant). The concept of ERC was first introduced by Koza [8]. An ERC returns a value that is decided randomly when the node is created. In our algorithm the return value of an ERC is chosen randomly from the range $[-5, 5)$, though the infrastructure supports using other value ranges as well.

B. Domain-Specific Terminal Nodes

The domain-specific terminal nodes are listed in two tables: Table II shows nodes describing characteristics that have to do with the board in its entirety, and Table III shows nodes describing characteristics of a certain square on the board.

The `KingFactor()` terminal (Table II) is a constant set to 1.4. It signifies the ratio between the value of a king and the value of a man in material evaluation of boards in

TABLE II

DOMAIN-SPECIFIC TERMINAL NODES THAT DEAL WITH BOARD CHARACTERISTICS.

Node name	Type	Return value
EnemeyKingCount()	F	The enemy's king count
EnemeyManCount()	F	The enemy's man count
EnemeyPieceCount()	F	The enemy's piece count
FriendlyKingCount()	F	The player's king count
FriendlyManCount()	F	The player's man count
FriendlyPieceCount()	F	The player's piece count
KingCount()	F	FriendlyKingCount() – EnemeyKingCount()
KingFactor()	F	King factor value
ManCount()	F	FriendlyManCount() – EnemeyManCount()
Mobility()	F	The number of plies available to the player
PieceCount()	F	FriendlyPieceCount() – EnemeyPieceCount()

TABLE III

DOMAIN-SPECIFIC TERMINAL NODES THAT DEAL WITH SQUARE CHARACTERISTICS. THEY ALL RECEIVE TWO PARAMETERS—X AND Y—THE ROW AND COLUMN OF THE SQUARE, RESPECTIVELY.

Node name	Type	Return value
IsEmptySquare(X,Y)	B	True iff square empty
IsFriendlyPiece(X,Y)	B	True iff square occupied by friendly piece
IsKingPiece(X,Y)	B	True iff square occupied by king
IsManPiece(X,Y)	B	True iff square occupied by man

American Checkers. It was included in some of the runs and plays a role also in calculating the return value of the piece-count nodes. A king-count terminal returns the number of kings the respective player has, or a difference between the two players' king counts. A man-count terminal returns the number of men the respective player has, or a difference between the two players' man counts. In much the same way a piece-count node returns the number of the respective player's men on the board and adds to it the number of that player's kings multiplied by the king factor. Again there is a node that returns the difference between the two players' piece counts.

The mobility node was a late addition that greatly increased the playing ability of the fitter individuals in the population. This terminal allowed individuals to more easily adopt a mobility-based, game-state evaluation function.

The square-specific nodes all return boolean values. They are very basic, and encapsulate no expert human knowledge about the game. In general, one could say that all the domain-specific nodes use little in the way of human knowledge about the game, with the possible exception of the king factor and mobility terminals. This goes against what has traditionally been done when GP is applied to board games [1, 5, 6]. This is partly due to the difficulty in finding useful board attributes for evaluating game states in Lose Checkers—but there is another, more fundamental, reason. Not introducing game-specific knowledge into the domain-specific nodes means the GP algorithm defined is itself not game specific,

TABLE IV

FUNCTION NODES. F_i: FLOATING-POINT PARAMETER, B_i: BOOLEAN PARAMETER.

Node name	Type	Return value
AND(B_1,B_2)	B	Logical AND of parameters
LowerEqual(F_1,F_2)	B	True iff $F_1 \leq F_2$
NAND(B_1,B_2)	B	Logical NAND of parameters
NOR(B_1,B_2)	B	Logical NOR of parameters
NOTG(B_1,B_2)	B	Logical NOT of B_1
OR(B_1,B_2)	B	Logical OR of parameters
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 multiplied by preset random number
NullFuncJ(F_1,F_2)	F	F_1
Plus(F_1,F_2)	F	$F_1 + F_2$

and thus more flexible (it is worth noting that mobility is a universal principle in playing board games, and therefore the mobility terminal can be seen as not game-specific). As defined, the algorithm can be used on the two games played in the American Checkers domain. A very slight change in the genetic program and the appropriate game program can render our setup applicable to any variant of Checkers (the number of conceivable Checkers variants that are at least computationally interesting is virtually unlimited). Our setup can also be used with little adaptation for other board games that have no more than two types of pieces, such as Othello, or even Go, the holy grail of AI board-game research.

C. Function Nodes

Several basic domain-independent functions have been defined. These are presented in Table IV. No domain-specific functions were defined.

The functions implemented include logic functions, basic arithmetic functions, one relational function, and one conditional statement. The conditional expression renders natural control flow possible and allows us to compare values and return a value accordingly. In Figure 1 we see an example of this. The subtree depicted in the figure returns 0 if the friendly piece count is less than double the number of enemy kings on the board, and the number of enemy kings plus 3.4 otherwise.

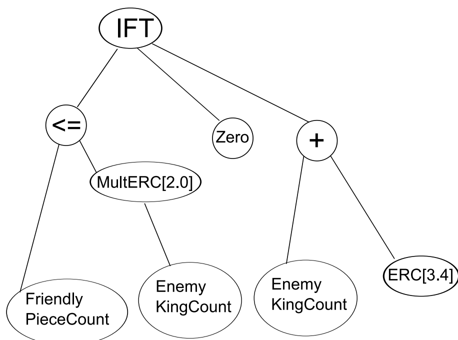

Fig. 1. Example of a subtree in our setup.

D. One-Way Crossover

One-way crossover, as opposed to the typical two-way version, 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. This can be seen as akin to an act of "aggression," where one individual pushes its genes upon another, as opposed to the generic two-way crossover operators that are more of a cooperative enterprise. 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. An example is shown in Figure 2.

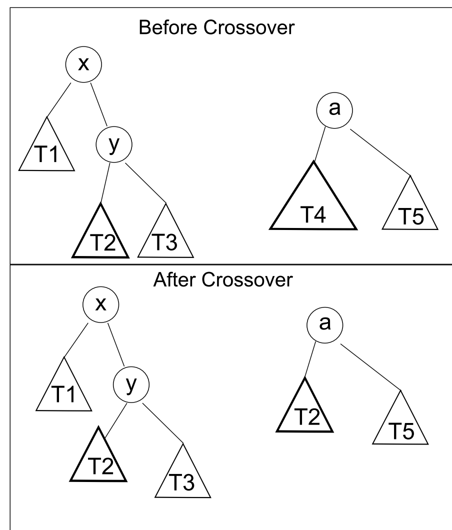

Fig. 2. One-way crossover: Subtree T2 in left, donor tree replaces subtree T4 in right, receiver tree. The donor tree remains unchanged.

This type of crossover operator is uni-directional. There is a donor and a receiver of genetic material. This directionality can be used to make one-way crossover more than a random operator. In this work, the individual with higher fitness was always chosen 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.

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. Using one-way crossover, as we do, just strengthens this trend. In one-way crossover, the donor individual pushes a copy of one of its genes into the receiver's genome at the expense of one of the receiver's own

Algorithm 1 Crossover.

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

genes. 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.

Both types of crossover used have their roots in nature. Two-way crossover is often seen as analogous to sexual reproduction. One-way crossover also has an analog in nature in the form of lateral gene transfer that exists in bacteria.

E. Local Mutation

It is difficult to define an effective local mutation operator for tree-based GP. Any change, especially in a function node that is not part of an intron, is likely to radically change the individual's fitness. In order to afford local mutation with limited effect, we changed the GP setup. To each node returning a floating-point value we added a floating-point variable (initialized to 1) that served as a factor. The return value of the node was the normal return value multiplied by this factor. A local mutation would then be a small change in the node's factor value.

Whenever a node returning a floating-point value was chosen for mutation, a decision had to be made on whether to activate the traditional tree-building mutation operator, or the local factor mutation operator. Toward this end we designated a run parameter that determined the probability of opting for the local mutation operator.

F. 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 [10] distinguished between two types of subtrees that are sometimes referred to as introns in the literature:

- *Unoptimized code*: Areas of code which can be trivially simplified without modifying the individual's operation, but not just replaced with anything.
- *Inviable code*: Subtrees which cannot be replaced by anything that can possibly change the individual's operation.

Luke focused on inviable introns. We will do the same in this work because unoptimized code seems to cast too wide a net and wander too far from the original meaning of the term "intron" in biology. We also make another distinction between two types of inviable code introns:

1) *Live-code introns*: Subtrees which cannot be replaced by anything that can possibly change the individual's operation, but may still generate code that will run at some point.
2) *Dead-code introns*: Subtrees whose code is never run.

Figure 3 exemplifies our definitions of introns in GP: $T1$ is a live-code intron, while $T3$ and $T5$ are dead-code introns. $T1$ is calculated when the individual is executed, but its return value is not relevant because the logical OR with a $true$ value always returns a $true$ value. $T3$, on the other hand, never gets calculated because the IFT function node above it always turns to $T2$ instead. $T3$ is thus dead code. Similarly, $T5$ is dead code because the $NullJ$ function returns a value that is independent of its second parameter.

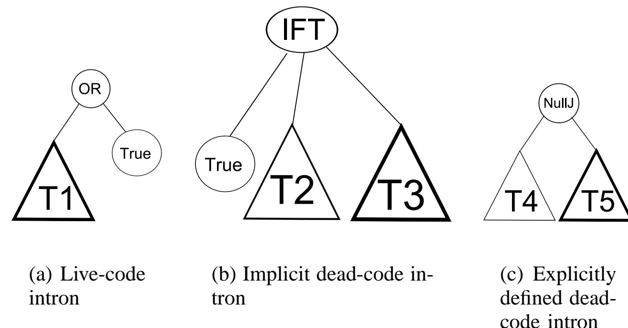

Fig. 3. Examples of different types of introns in GP trees.

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. In Figure 3, $T5$ differs from $T3$ in that $T5$ is known to be an intron the moment $NullJ$ is reached, and therefore the program can take it into account. 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.

Nordin et al. [13] explored EDIs in linear GP, finding that they tend to improve fitness and shorten runtime, as EDIs allow the evolutionary algorithm to protect important functional genes and save runtime used by live-code introns.

Earlier work showed that using introns was also helpful in GAs [9].

G. Multi-Tree Individuals

Support of multi-tree individuals was also implemented in our setup. Unlike the common case, the criteria we used were not based on deep domain knowledge. This is partly due to the fact that in Lose Checkers such information is hard to come by, but also because we were seeking a flexible setup that would be easily transferable to other board-game domains.

The idea, again based on nature, was to define our own "regulatory network." As we wanted to avoid using domain knowledge we chose instead to consider the top of the GP tree. Preliminary runs with one tree taught us a fair bit about the sort of solutions that tended to evolve. An emerging characteristic seemed to be that the majority of the population quickly focused on one design choice for the top of the tree and kept it, unchanged, in effect developing "regulatory" genes. The IFT function was a popular pick for the top part of the tree. This makes sense as the conditional expression is a good way to differentiate between cases. We decided to reinforce this trend, using multiple trees to simulate the run of a single tree with preset nodes at its top. We chose to use ten trees per individual, all returning floating-point values. The values returned by the ten trees were manipulated to simulate the behavior of the tree shown in Figure 4.

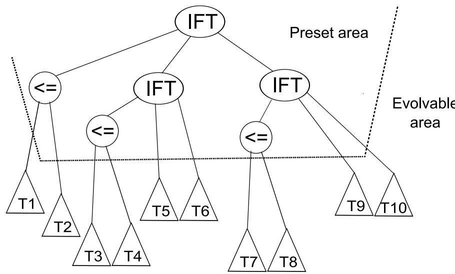

Fig. 4. A tree in our setup. Note preset (non-evolving) area (above line) and the evolving 10 subtrees, $T_1, \ldots, T_{10}$.

H. 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 face 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.

The method of evaluation described requires some information from the user, including the number of guides, their designations, the number of rounds per guide, and the number of games per round, for the guides array $GuideArr$ (players played X rounds of Y games each). The algorithm also needs to know the number of co-play opponents for the coevolution round. In addition, a parameter for game point value for different guides, as well as for the coevolution round, was also used in our setup. This allowed to ascribe higher significance to certain rounds than to others. All these are program run parameters that the program accepts from the user via parameter files. Tweaking these parameters allows for different setups.

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.

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 and was used to test initial runs. The alpha-beta player searched up to a preset depth in the game tree and used an evaluation function returning a random value for game states in which there was no clear winner (in states where win or loss was evident the evaluation function returned an appropriate value). To save time, not all individuals were chosen for each game round. We defined a cutoff for participation in a guide-play round. Before every guide-play round began, the best individual in the population was found. Only individuals whose fitness trailed that of the best individual by no more than the cutoff value got to play. When playing against a guide each player in the population received 1 point added to its fitness for every win, and 0.5 points for every draw.

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. This was done to make sure that no individuals received a disproportionately high fitness value by being chosen as opponents more times than others. When playing a co-play game, as when playing against a guide, each player in the population received 1 point added to its fitness for every win, and 0.5 points for every draw.

I. Selection and Procreation

The change in population from one generation to the next was divided into two stages: A selection stage and a procreation stage. In the selection stage the parents of the next generation were selected (some more than once) according to their fitness. In the procreation stage, genetic operators were applied to the parents in order to create the next generation.

Selection was done by the following simple method: Of two individuals chosen at random, a copy of the fitter individual was selected as a parent for the procreation stage (this is known as tournament selection). The pseudocode for the selection process is given in Algorithm 3.

Algorithm 3 Selection.

```
repeat
  Randomly choose two different individuals from population : I1 and I2
  if I1.Fitness > I2.Fitness then
    Select a copy of I1 for parent population.
  else
    Select a copy of I2 for parent population.
  end if
until number of parents selected is equal to original population size
```

Two more parameters are crossover and mutation probabilities, respectively denoted p_{xo} and p_m. Every individual was chosen for crossover (with a previously unchosen individual) with probability p_{xo} and self-replicated with probability $1-p_{xo}$. The implementation and choice of specific crossover operator is as in Algorithm 1. After crossover every individual underwent mutation with probability p_m. There is a slight break with traditional GP structure, where an individual goes through either mutation or crossover but not both. However our setup is in line with the GA tradition where crossover and mutation act independently of each other.

J. Players

Our setup supported two different kinds of GP players. The first kind of player examines all legal moves and uses the GP individual to assign scores to the different moves, choosing the one that scores highest. This method is essentially a minimax search of depth 1. The second kind of player mixes GP game-state evaluation with a minimax search. It uses the alpha-beta search algorithm implemented for the guides, but instead of evaluating non-terminal states randomly it does so using the GP individual. This method adds search power to our players, but creates a program wherein deeper search creates more game states to be evaluated, taking more time.

K. Summary of Run Parameters

- Number of generations (between 100–200).
- Population size (between 100–200).
- Crossover probability (typically 0.8).
- Mutation probability (0.1, or 0.2 if local mutation used).
- Local mutation ratio (0 or 0.5).
- Maximum depth of GP tree (15 without search, 12–14 with search of depth 3, 10 with search of depth 4).
- Player to serve as benchmark for the best player of each generation.
- Search depth used by GP players during run.

TABLE V

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 RANDOM EVALUATION FUNCTION.

1st Player	2nd Player	1st Player win ratio
$\alpha\beta2$	random	0.9665
$\alpha\beta3$	$\alpha\beta2$	0.8502
$\alpha\beta5$	$\alpha\beta3$	0.82535
$\alpha\beta7$	$\alpha\beta5$	0.8478
$\alpha\beta3$	$\alpha\beta4$	0.76925
$\alpha\beta3$	$\alpha\beta6$	0.6171
$\alpha\beta3$	$\alpha\beta8$	0.41270
$\alpha\beta5$	$\alpha\beta6$	0.7652
$\alpha\beta5$	$\alpha\beta8$	0.55620

V. RESULTS

The experiments were divided into two sets, the first using no search, the second with search and mobility incorporated into the evolutionary algorithm. The same handcrafted machine players that were used as guides in fitness evaluation also served as the benchmark players. Before beginning the evolutionary experiments, we first evaluated our guide players by testing them against each other in matches of 10,000 games (with players alternating between playing black and white). For search depths greater than 2 we focused on guide players using odd search depths, as those using even search depths above 2 proved weaker than the players using lower odd search depths. Every one of the alpha-beta players participated in a match with a weaker player and a stronger player (except for the strongest searcher tested). Table V presents the results of these matches. Note the advantage of odd search depth over (higher) even search depth.

In all evolutionary runs that follow evolution ran on a single core of a dual-core Xeon 5140 2.33GHz processor. All runs took from a few days to a week (we limited runtimes to 160 hours).

A. No Search

The first set of experiments involved no search (meaning a lookahead of 1). When tested against the random player the evolved move evaluators showed a significant improvement in level of play, as can be seen in Table VI.

B. Shallow Search

Using shallow search produced better results. Experiments showed that a search depth of 3 was the best choice. This allowed for improved results, while deeper search caused runs to become too slow. In order to save time, the maximum GP-tree depth was more strongly restricted, and code optimizations were added to improve runtime. Attempts to use the multi-tree design with search did not yield good results, and this design was abandoned in favor of the standard single-tree setup. It is quite possible that another, less wasteful, multi-tree design, would have succeeded more. The results are shown in Table VII.

TABLE VI

RESULTS OF TOP RUNS USING A LOOKAHEAD OF 1 (I.E., NO SEARCH). HERE AND IN SUBSEQUENT TABLES: THE BENCHMARK (POST-EVOLUTIONARY) SCORE OF THE EVOLVED *Player* IS CALCULATED OVER 1000 GAMES AGAINST THE RESPECTIVE *Benchmark Opponent*; THE BENCHMARK SCORE IS THE NUMBER OF GAMES WON PLUS HALF THE NUMBER OF GAMES DRAWN; xRAND STANDS FOR x GAMES AGAINST THE RANDOM PLAYER; yCO STANDS FOR y GAMES OF CO-PLAY. *Benchmark Opponent* HEREIN IS THE RANDOM PLAYER.

Run identifier	Fitness Evaluation	Benchmark Score
r00032A	20Rand+30Co	554.0
r00033B	30Rand+5Co	634.0
r00034A	30Rand+5Co	660.0
r00035B	30Rand+5Co	721.0
r00036A	30Rand+10Co	705.5
r00037B	30Rand+10Co	666.5

TABLE VII

RESULTS OF TOP RUNS USING SHALLOW SEARCH. *Player* USES $\alpha\beta$ SEARCH OF DEPTH 3 COUPLED WITH EVOLVED EVALUATION FUNCTION, WHILE *Benchmark Opponent* USES $\alpha\beta$ SEARCH OF DEPTH 3 COUPLED WITH A RANDOM EVALUATION FUNCTION.

Run identifier	Fitness Evaluation	Benchmark Score
r00044A	50Co	744.0
r00046A	50Co	698.5
r00047A	50Co	765.5
r00048A	50Co	696.5
r00049A	50Co	781.5
r00056A	50Co	721.0
r00057A	50Co	786.5
r00058A	50Co	697.0
r00060A	50Co	737.0
r00061A	50Co	737.0

Analysis of Evolved Players. As we saw above, random evaluation coupled with odd search depth is a powerful heuristic. We tested some of our best evolved players against players using random evaluation with even search depth values greater than 3. The results of these tests are given in Table VIII. As evident, the evolved players, some more than others, have acquired a playing proficiency that allows them to outmaneuver players employing far deeper search and taking far greater time resources. It is worth noting that the player $\alpha\beta 8$, which uses a search depth of 8, is decisively outperformed by three distinct evolved players (which only search to a depth of 3). This player (as Table V clearly indicates) is stronger than the $\alpha\beta 3$ player that served as the evolved players' original benchmark.

C. Expanding Search Depth

In order to produce good results against stronger players, without incurring high runtime costs, we took some of the existing players evolved using shallow search of depth 3 and modified them to use deeper search with the same (evolved) evaluation function. We then tested the modified players against a benchmark player using deeper search, hoping that expanding search depth would preserve the quality of the evaluation function (Chellapilla and Fogel [3] succeeded in doing this with American Checkers). In these select runs,

TABLE VIII

RESULTS OF TOP RUNS PLAYING AGAINST PLAYERS USING RANDOM EVALUATION AND VARIOUS SEARCH DEPTHS, FOCUSING ON EVEN DEPTHS GREATER THAN 3. *Player* USES $\alpha\beta$ SEARCH OF DEPTH 3 COUPLED WITH EVOLVED EVALUATION FUNCTION.

Run identifier	vs $\alpha\beta 3$	vs $\alpha\beta 4$	vs $\alpha\beta 6$	vs $\alpha\beta 8$
r00044A	744.0	944.5	816.0	758.0
r00047A	765.5	899.0	722.5	476.0
r00049A	781.5	915.0	809.0	735.5
r00057A	786.5	909.0	745.5	399.5
r00060A	737.0	897.0	627.0	408.5
r00061A	737.0	947.0	781.5	715.5

TABLE IX

RESULTS OF TOP RUNS BEFORE AND AFTER DEPTH EXPANSION. NOTE THAT *Player* WAS EVOLVED USING A SEARCH DEPTH OF 3, BUT IS HERE TESTED WITH ITS DEPTH EXTENDED TO 5, VS. THE RANDOM-EVALUATION $\alpha\beta 5$, $\alpha\beta 6$, AND $\alpha\beta 8$ *Benchmark Opponents*.

Run identifier	vs $\alpha\beta 5$	vs $\alpha\beta 6$	vs $\alpha\beta 8$
r00044A	438.5	774.0	507.5
r00047A	437.5	807.0	482.5
r00049A	420.5	856.0	449.0
r00057A	494.5	874.5	463.5
r00060A	459.0	834.0	583.5
r00061A	483.5	967.0	886.0

we chose the evolved individual that had the best benchmark score against the $\alpha\beta 3$, altered its code to extend its search depth to 5, and ran a 1000-game match between the altered, deeper-search player and $\alpha\beta$ random-evaluation players. The results of this experiment are presented in Table IX.

D. Using Deeper Search

After optimizing the code to some extent we used a search depth of 4 in our runs. To try and prevent exceedingly long runtimes population size, number of generations, and maximum tree depth, were severely limited (to 50, 70, and 10, respectively). Table X presents the results. We see that players have evolved to beat the strong $\alpha\beta 5$ player but at the cost of some overspecialization against that player.

VI. CONCLUDING REMARKS AND FUTURE WORK

We presented the genetic programming approach as a tool for discovering effective strategies for playing the game

TABLE X

RESULTS OF TOP RUNS PLAYING AGAINST PLAYERS USING RANDOM EVALUATION AND VARIOUS SEARCH DEPTHS. *Player* USES $\alpha\beta$ SEARCH OF DEPTH 4 COUPLED WITH EVOLVED EVALUATION FUNCTION.

Run identifier	Fitness Evaluation	vs. $\alpha\beta 5$	vs. $\alpha\beta 6$	vs. $\alpha\beta 8$
r00064A	20$\alpha\beta$5+20Co	582.0	603.5	395.0
r00065A	20$\alpha\beta$5+20Co	537.0	782.5	561.5
r00066A	20$\alpha\beta$5+20Co	567.0	757.5	483.5
r00067A	20$\alpha\beta$5+20Co	598.5	723.0	385.5
r00068A	20$\alpha\beta$5+20Co	548.0	787.0	524.0
r00069A	20$\alpha\beta$5+20Co	573.5	715.5	523.0
r00070A	20$\alpha\beta$5+20Co	577.0	691.5	476.0
r00071A	20$\alpha\beta$5+20Co	551.5	582.5	401.5

of Lose Checkers. Guided by the gene-centered view of evolution, which describes evolution as a process in which segments of self-replicating units of information compete for dominance in their genetic environment, 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 Lose Checkers—a full, nontrivial board game. Our use of search with GP is another novelty (though Hauptman and Sipper [5] also used search, differently, in their work on Chess endgames).

Our approach opens the door to further GP research involving games and search. Below are some potential avenues for future exploration:

1) Applying the GP approach to other computationally interesting Checkers variants. This has the advantage that the setup hardly needs to be changed and most effort can be invested in applying new methods to improve the evolutionary algorithm's performance.
2) Applying the GP approach to other board games. This work is, as far as we know, the first time tree-based GP was used to evolve players for a full board game, and one of very few attempts at applying the evolutionary approach to board games in general. Our attempt demonstrates that a game like Lose Checkers can be tackled even with little domain knowledge and expertise. Considering the rate of growth in available computational power, it is quite possible that the big names in board games, Chess and Go, will be within reach in the coming years.
3) GP can be applied to guiding search itself in games and puzzles, deciding which nodes to develop first [6].
4) GP can also be applied to more complicated games that are not full-knowledge, or contain a stochastic element. This applies to many turn-based computer strategy games, and is also a better approximation of real-world problems.
5) Other fields where search is used in conjunction with heuristic functions, such as planning and AI research.

There are many possibilities for further research. As long as the strategies for solving the problem can be defined and the quality of solvers can be evaluated in reasonable time, there is an opening for using GP to evolve a strong problem-solving program. All that is required is that solvers be evaluated in such a way that those solvers that are closer to doing the job right get higher fitness, and that the search space defined by the GP design be such that good solvers tend to be clumped together (i.e., be similar to each other), or that the same code segments tend to appear in many good solvers, so as to allow the gradual change and improvement that is a hallmark of the evolutionary process.

References

[1] Y. Azaria and M. Sipper, "GP-Gammon: Genetically programming backgammon players," *Genetic Programming and Evolvable Machines*, vol. 6, no. 3, pp. 283–300, 2005.

[2] K. Chellapilla and D. B. Fogel, "Evolving an expert checkers playing program without using human expertise," *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 4, pp. 422–428, 2001.

[3] ——, "Evolving neural networks to play checkers without relying on expert knowledge," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 5, pp. 1382–1391, 1999.

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

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

[6] ——, "Evolution of an efficient search algorithm for the mate-in-n problem in chess," in *Proceedings of 10th European Conference on Genetic Programming (EuroGP2007)*, ser. Lecture Notes in Computer Science, M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi, and A. I. Esparcia-Alcázar, Eds., vol. 4445. Springer-Verlag, Heidelberg, 2007, pp. 78–89.

[7] M. Hlynka and J. Schaeffer, "Automatic generation of search engines," in *Advances in Computer Games*, 2006, pp. 23–38.

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

[9] 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*. Morgan Kaufmann, 1991, pp. 123–127.

[10] S. Luke, "Code growth is not caused by introns," in *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, D. Whitley, Ed., Las Vegas, Nevada, USA, July 2000, pp. 228–235.

[11] D. J. Montana, "Strongly typed genetic programming," *Evolutionary Computation*, vol. 3, pp. 199–230, 1993.

[12] D. E. Moriarty and R. Miikkulainen, "Discovering complex Othello strategies through evolutionary neural networks," *Connection Science*, vol. 7, no. 3, pp. 195–210, 1995.

[13] 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*, 1996, pp. 6–22.

[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*, vol. 9, no. 6, pp. 628–640, 2005.

[15] A. L. Samuel, "Some studies in machine learning using the game of Checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, July 1959.

[16] ——, "Some studies in machine learning using the game of Checkers II - recent progress," *IBM Journal of Research and Development*, vol. 11, no. 6, pp. 601–617, 1967.

[17] J. Schaeffer, R. Lake, P. Lu, and M. Bryant, "Chinook: The world man-machine checkers champion," *AI Magazine*, vol. 17, no. 1, pp. 21–29, 1996.

[18] J. Schaeffer, N. Burch, Y. Bjornsson, A. Kishimoto, M. Muller, R. Lake, P. Lu, and S. Sutphen, "Checkers is solved," *Science*, vol. 317, no. 5844, pp. 1518–1522, 2007.

[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)*, 1952, pp. 46–49.

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