Evolving Both Search and Strategy for Reversi Players using Genetic Programming

Amit Benbassat and Moshe Sipper

Abstract—**We present the application of genetic programming to the zero-sum, deterministic, full-knowledge board game of Reversi. Expanding on our previous work on evolving board-state evaluation functions, we now evolve the search algorithm as well, by allowing evolved programs control of game-tree pruning. We use strongly typed genetic programming, explicitly defined introns, and a selective directional crossover method. We show that our system regularly churns out highly competent players and our results prove easy to scale.**

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 full knowledge, deterministic, zero-sum board games, expanding on our previous work on Lose Checkers [2] and other games [3, 4].

We apply tree-based Genetic Programming (GP) to evolving players for Reversi. 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, we see it as evidence that it is more likely too. We are mindful of evolutionary theory, particularly as pertaining to the gene-centered view of evolution. This view, presented by Williams [27] and expanded on by Dawkins [8], 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.

In much of the work on games the focus is on a single game, the goal being to reach a high level of play. In such research much effort goes into integrating domain-specific expert knowledge into the system in order to get the best possible player. For many games opening books of game-specific strong opening moves are created offline and used in order to give the player an edge over a less-prepared rival [10]. In Checkers, a game with only two piece types, with the number of pieces on the board tending to drop towards the end, endgame databases are often used to allow the player to "know" which moves lead to victory from numerous precomputed positions [23, 24]. This trend culminated in the construction of a database of all possible 3.9×10^{13} game states in American Checkers that contain at most 10 pieces on the board [24].

Conversely, our focus is on multi-game generality. Using our game system, which we have demonstrated to be flexible and easily applicable to multiple games, we choose to avoid using both specialized techniques and expert domain knowledge in favor of generic, easily transferable evolutionary techniques.

2012 IEEE Conference on Computational Intelligence and Games (IEEE CIG 2012). The authors are with the Dept. of Computer Science, Ben-Gurion University, Beer-Sheva, Israel. Email: {amitbenb,sipper}@cs.bgu.ac.il.

II. Reversi

Reversi, also known as Othello, is a popular game with a rich research history [15, 18**?** , 19]. The most popular Reversi variant is a board game played on an 8x8 board. Reversi is a piece-placing game, meaning that moves are made by placing a new piece on the board rather than by moving existing pieces around as in games such as Chess and Checkers. The players place their pieces on the board in turns, attempting to capture and convert opponent pieces by locking them between friendly pieces. In Reversi, the number of pieces on the board increases during play, rather than decrease as it does in Chess and Checkers. This fact makes endgame databases all but useless for Reversi. On the other hand, the number of moves (not counting the rare pass moves) in Reversi is limited by the board's size, making it a short game. There is also 10x10 variant of Reversi, which is quite popular. In this paper we focus on the 8x8 version.

III. Related Work

In the years since Strachey [26] first designed an American Checkers-playing algorithm, there has been some work on board game-playing computer programs. Notable progress was made by Samuel [21, 22], 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 [23].

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 [18] to attack the game of Reversi,

DRAFT

resulting in a competent player that employed sophisticated mobility play. ANN-based American Checkers players were evolved by Chellapilla and Fogel [5, 6] 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 [12] and an efficient solver for the Mate-in-N problem in Chess [13]. In 2010 we provided evidence that good board evaluation functions for Lose Checkers could indeed be evolved [2]. Gauci and Stanley [9] used the HyperNEAT system to evolve artificial neural networks that prune the search tree in an existing Checkers playing algorithm called Cake, allowing it to search deeper and outperform the regular version of Cake.

The 8x8 variant of Reversi has received its fair share of research attention. Early landmark work by Rosenbloom [19] yielded IAGO, an expert level Reversi program. Subsequent work by Lee and Mahajan [15] greatly improved on IAGO's level of play by utilizing Bayesian learning to improve the player's evaluation function. **?**] presented Logistello, a strong Reversi player that achieved dominance over world-class human opponents. The evolutionary approach was applied to Reversi by several researchers. A genetic algorithm (GA) with genomes representing ANNs was used in 1995 by Moriarty and Miikkulainen [18] to tackle the game of Reversi, resulting in a competent player that employed sophisticated mobility play. Chong et al. [7] presented a program using shallow search with evolved feed-forward ANNs encoded with board-spatial features as its board evaluation function. In 2011 we expanded on our previous work and evolved strong board evaluation functions for 8x8 Reversi [3].

IV. Evolutionary Setup

In our basic system the individuals in the population act as board-evaluation functions, to be combined with a standard game-search algorithm—in our case alpha-beta. The value an individual returns 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. In this work we add another evolvable feature, namely, search. A second evaluation function is evolved, which at each internal node chooses the more-promising child nodes for expansion and further evaluation, and discards the rest.

The evolutionary algorithm was written in Java. We chose to implement a strongly typed GP framework [17] 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 [14], another form of crossover was implemented—which we designated "selective crossover"—as well as a local mutation operator. The original setup is detailed in [2, 3]. Its main points along with recent updates and novel results are detailed in this paper. To achieve good results on multiple games using deeper search we enhanced our system with the ability to run in parallel multiple threads.

TABLE I
Basic terminal nodes. F: floating point, B: boolean.

Node name	Return type	Return value
`ERC()`	F	Ephemeral Random Constant
`False()`	B	Boolean $false$ value
`One()`	F	1
`True()`	B	Boolean $true$ value
`Zero()`	F	0

TABLE II
Domain-specific terminal nodes that deal with board characteristics.

Node name	Type	Return value
`EnemeyManCount()`	F	The enemy's man count
`FriendlyManCount()`	F	The player's man count
`ManCount()`	F	`FriendlyManCount()` – `EnemeyManCount()`
`Mobility()`	F	The number of plies available to the player
`FriendlyCornerCount()`	F	Number of corners in friendly control
`EnemyCornerCount()`	F	Number of corners in enemy control
`CornerCount()`	F	`FriendlyCornerCount()` – `EnemyCornerCount()`

A. Basic Terminal Nodes

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

The ERC (Ephemeral Random Constant) returns a value in the range $[-5, 5)$ that is set at random when the node is created.

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.

A man-count terminal returns the number of men the respective player has, or a difference between the two players' man counts. The mobility node is an addition that greatly increases the playing ability of the fitter individuals in the population. This terminal allows 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 human knowledge about the game, with the possible exception of the mobility terminal. This

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
`IsManPiece(X,Y)`	B	True iff square occupied

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
NullJ(F_1,F_2)	F	F_1
Plus(F_1,F_2)	F	$F_1 + F_2$

goes against what has traditionally been done when GP is applied to board games [1, 12, 13, 25]. This is partly due to the difficulty in finding useful board attributes for evaluating game states in some games (Benbassat and Sipper [2] deals with a game that is a perfect example of this)—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, 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).

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 a GP tree containing a conditional expression. The subtree depicted in the figure returns 0 if the friendly corners count is less than double the number of enemy men on the board, and the number of enemy men plus 3.4 otherwise.

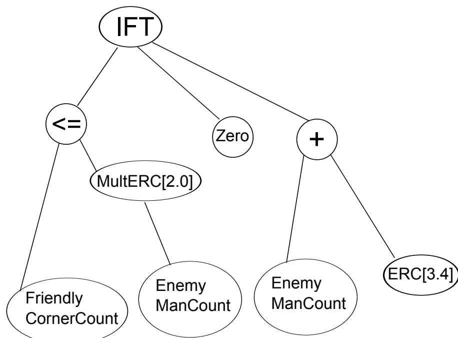

Fig. 1. Example of a subtree in our setup.

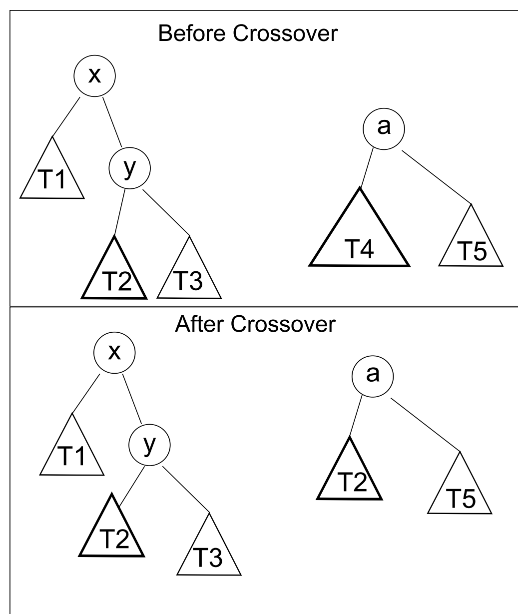

Fig. 2. One-way crossover: Subtree T2 in donor tree (left) replaces subtree T4 in receiver tree (right). The donor tree remains unchanged.

D. Selective 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 cooperative in nature. 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.

This type of crossover operator is uni-directional, with 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 system. 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. The algorithm can be seen as describing a new genetic operator, which we dub *selective crossover*, since it exerts selective pressure because less-fit individuals are more likely to receive genetic information from fitter ones than vice versa.

Using the vantage point of the gene-centered view of evolution it is easier to see the logic of crossover in our

Algorithm 1 Selective 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

system. 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 framework using the generic two-way crossover alone. Using selective crossover, as we do, just strengthens this trend. When selective crossover applies 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 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. The selective crossover operator thus causes an increase in the frequency of the genes that lead to better fitness.

Both basic 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

Our system also incorporates *Explicitly Defined Introns* (EDIs) that appear under each `NullJ` and `NotG`. Introns in GP are comprised of code that has no effect on overall fitness. EDIs are introns that have been designed to be introns, and therefore can be safely ignored when compiling the program, thus saving runtime. Luke [16] discusses introns in some detail. For more discussion of introns in our system see Benbassat and Sipper [2].

G. Multi-Tree Individuals

Support of multi-tree individuals was also implemented in our setup. In this work we used a second GP-tree to evaluate internal game tree nodes and decide on forward pruning. In principle, our system supports multiple GP-trees for individuals and these can be adapted to a variety of roles (see Benbassat and Sipper [2] for other uses).

H. Fitness Calculation

Fitness calculation was carried out in the fashion described in Algorithm 2. 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 [20], 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
Assign 1 point per every game won by the individual, and 0.5 points per drawn game

The method of evaluation described requires some parameter setting, 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 required. This allowed us to ascribe greater significance to certain rounds than to others. Tweaking these parameters allows for different setups.

Guide-Play Rounds. We implemented two types of guides: 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 a handcrafted evaluation function for states in which there was no clear winner. 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 we used tournament selection 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.

Selection was done by the following simple method: Of several individuals chosen at random, copies of a subset of fitter individuals was selected as parents for the procreation stage. The pseudocode for the selection process is given in Algorithm 3.

Algorithm 3 Selection(TourSize,WinTourSize)

repeat
  Randomly choose TourSize different individuals from population : $\{ I_1 \dots I_{TourSize} \}$
  Select a copy of $\{ J_1 \dots J_{WinTourSize} \}$, the subset of $\{ I_1 \dots I_{TourSize} \}$ containing the $WinTourSize$ individuals with the highest fitness score, for parent population.
until number of parents selected is equal to original population size

Two more parameters are crossover and mutation probabilities, denoted p_{xo} and p_m, respectively. 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 was as in Algorithm 1. After crossover every individual underwent mutation with probability p_m (another parameter, p_{lm}, denotes the probability of the algorithm choosing to perform local mutation). There is a slight break with traditional GP structure, where an individual goes through either mutation or crossover but not both. However our system is in line with the GA tradition where crossover and mutation act independently of each other.

J. Players with Forward Pruning

Our evolutionary system evolves GP players that use the alpha-beta search algorithm implemented for the guides, but instead of evaluating non-terminal states via a handcrafted evaluation function the system does so using the evolving GP individual, thus combining GP game-state evaluation with minimax search. 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. Therefore, we recently added selective search in the form of forward pruning. This method speeds up play if the search depth is kept constant, but at the cost of losing information about the game tree.

Our system currently supports two approaches to selective search via forward pruning. One relies on a parameter called *SelectiveSearchRatio*. This parameter is a floating-point number in the range $(0, 1]$. It sets the ratio of sibling states that get further expanded as long as the final search depth has not been reached (the number of siblings actually expanded is rounded up to the nearest integer). If, for example, *SelectiveSearchRatio*=0.25, this means that out of every four child nodes a board state has in the game tree, one will be expanded further and the others will be pruned. The other approach relies on a parameter called *MaxBranchingFactor*. This parameter is a positive integer and sets a hard limit for the effective branching factor of the searched game tree. If, for example, *MaxBranchingFactor*=5, this means that at most five sibling nodes anywhere in the game tree will be expanded for further search. Our system can use either or both parameters to limit the breadth of its search. We also implemented a method to temper the ill effects of too much forward pruning, using a third parameter, *FullSearchDepth*. This parameter is a non-negative integer and dictates that the search algorithm will behave normally up to a certain given depth. If, for example, *FullSearchDepth*=2, this means that up to depth 2 in the search tree all nodes that the base search algorithm (alpha-beta in our case) would normally expand will also be expanded by the selective search algorithm. Control of the maximal search depth is a feature that exists in our system anyway and is managed by a fourth parameter called *SearchDepth*.

Forward pruning in our system is achieved by using a second state evaluation function that allows us to sort sibling nodes according to their heuristic value and select those evaluated as better to be expanded and searched further. We can have this done either by using the same evolved heuristic evaluation function used for evaluating board states at the bottom of the search tree, or we can use a different evolved GP tree for this task. In this work we used a different evolved evaluation function to guide search.

K. Summary of Run Parameters

- Number of generations: 100
- Population size: 120
- Crossover probability: 0.8
- Mutation probability: 0.2
- Local mutation ratio: 0.5
- Maximum depth of GP tree: 15
- Player to serve as benchmark for the best player of each generation ($\alpha\beta 5p$)

TABLE V

RELATIVE LEVELS OF PLAY FOR DIFFERENT BENCHMARK (GUIDE) PLAYERS IN REVERSI. HERE AND IN THE SUBSEQUENT TABLES $\alpha\beta i$ REFERS TO AN ALPHA-BETA PLAYER USING A SEARCH DEPTH OF i AND A MATERIAL EVALUATION FUNCTION.

1st Player	2nd Player	1st Player win ratio
$\alpha\beta 2$	random	0.8471
$\alpha\beta 3$	$\alpha\beta 2$	0.6004
$\alpha\beta 5$	$\alpha\beta 3$	0.7509
$\alpha\beta 7$	$\alpha\beta 5$	0.7662

- Search depth used by GP players during run (varies for different runs)
- Selective search pruning parameters (vary for different runs)

V. RESULTS

In order to test the quality of evolved players we created hand-written players. Our approach was to use a standard algorithm employed with board games: alpha-beta search. The search proceeds up to a certain, predetermined depth, at which point a game-dependent evaluation function is called upon. As we shall see below, these alpha-beta players were used both during evolution as "guides" for fitness evaluation and also as benchmark players used to test our evolved players post-evolutionarily.

We made a point of making our players' strategy contain a random element so as to render the development of a specialized strategy against them more difficult and to allow for their use as benchmark opponents. 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 either side). Table V shows the relative strengths of the different players in the different games.

In our tests we observed a trend where players differ in level of play based not just on how deep their search is, but also on whether the depth is odd or even. This is due to what is sometimes referred to as the *Odd-Even Effect* [11], where the depth of the search being odd or even greatly affects play strategy due to the identity of the player who gets to play last in the expanded game tree. We made sure to always test our evolved players against hand-crafted players that were at least as strong as all hand-crafted players with lower search depths and in most cases also stronger than some hand-crafted players with greater search depths. In Table V we omitted the weak handcrafted players that use even search depths.

In Benbassat and Sipper [3] our handcrafted players were overwhelmed by players using far less search. In order to supply our evolved players with more of a challenge we wrote new players that use a stronger evaluation function. Table VI shows the level of the new handcrafted players that use search depths of 5 and 7 in relation to the old ones.

In all evolutionary runs that follow we used 8 cores of 3 IBM x3550 M3 servers with 2 Quad Core Xeon E5620 2.4GHz SMT processors with 12MB L3 cache and 24GB RAM. Runs took 2–4 days.

TABLE VI

RELATIVE LEVELS OF PLAY FOR DIFFERENT BENCHMARK (GUIDE) PLAYERS IN REVERSI THAT SHOW SUPERIORITY OF NEW HANDCRAFTED PLAYERS. NEW HANDCRAFTED PLAYERS ARE DENOTED BY A 'P' AT THE END OF THEIR NAME.

1st Player	2nd Player	1st Player win ratio
$\alpha\beta 5p$	$\alpha\beta 5$	0.6342
$\alpha\beta 7p$	$\alpha\beta 5p$	0.8418
$\alpha\beta 7p$	$\alpha\beta 7$	0.62855

TABLE VIII

COMPARISON OF GAME-STATE EXPANSION BETWEEN DIFFERENT PLAYERS. PLAYER 161 CONDUCTS FULL $\alpha\beta$ SEARCH OF DEPTH 4. THE OTHER PLAYERS USE FORWARD PRUNING TO LIMIT SEARCH BREADTH AND SEARCH DEEPER.

Run identifier	Search depth	Branching factor limit	Average # of states expanded per turn	Standard Deviation
161	4	–	563.14	124.89
162	5	5	186.73	46.66
167	6	3	162.34	25.36

We ran eight different simulations. In the first two, runs 160 and 161 (we tagged every run with a unique integer identifier), individuals used a full search of depth 4. The rest used forward pruning with deeper search. Runs 162 and 163 used a search depth of 5 with *MaxBranchingFactor*=5. Runs 164 and onward used a search depth of 6 with *MaxBranchingFactor*=3. Fitness was evaluated by having each individual play 25 games as Black and 25 games as White against other individuals in the population (coevolution). Table VII contains the results of the best individuals of these runs against $\alpha\beta 5p$ and $\alpha\beta 7p$. In the last two runs we added guide play to fitness evaluation in an attempt to correct erratic changes in benchmark score behavior caused by coevolutionary fitness evaluation. We tweaked the run parameters so that fitness that could potentially be accrued during a guide play round was half the fitness that could potentially be accrued in a coevolution round.

As we can see our evolved players using forward pruning hold their own against the deeper-searching handcrafted players, though the results seem to be weaker than the results from the runs using full search (run 162 does very well against $\alpha\beta 7p$ but this result may be a fluke). All these players, however, are significantly faster than the players using a full search of depth 4.

A. Speed advantage of players using forward pruning

A major advantage of using forward pruning in search is that a player can search deeper into the game tree in less time. We ran an analysis comparing three players that use different search strategies by having each one play 100 games against a stochastic opponent and checked how many game states each player expanded on average per turn before selecting a move. The results are presented in Table VIII. The winner of run 161, which uses full search, expands more states per move than the deeper-searching winners from runs 162 and 167.

TABLE VII
REVERSI: RESULTS OF TOP RUNS. *Benchmark Opponent* USES $\alpha\beta$ SEARCH OF DEPTHS 5 AND 7 COUPLED WITH A MATERIAL EVALUATION FUNCTION.

Run identifier	Fitness Evaluation	Search depth	Branching factor limit	Benchmark Score vs $\alpha\beta 5p$	Benchmark Score vs $\alpha\beta 7p$
160	25Co	4	–	891.5	575.0
161	25Co	4	–	935.0	605.0
162	25Co	5	5	922.5	881.5
163	25Co	5	5	602.0	286.0
164	25Co	6	3	824.5	458.0
165	25Co	6	3	851.5	607.0
166	20Co40 $\alpha\beta 3$	6	3	742.5	408.0
167	20Co40 $\alpha\beta 3$	6	3	897.0	443.5

TABLE IX
REVERSI: RESULTS OF RUNS. *Benchmark Opponent* USES $\alpha\beta$ SEARCH OF DEPTH 7 COUPLED WITH A MATERIAL EVALUATION FUNCTION.

Run identifier	Branching factor limit	Benchmark Score vs $\alpha\beta 7p$
164	3	458.0
164	4	637.5
164	5	799.0
165	3	607.0
165	4	728.5
165	5	817.0
166	3	408.0
166	4	824.5
166	5	811.0
167	3	443.5
167	4	629.0
167	5	862.0

VI. SCALING SELECTIVE SEARCH

In Benbassat and Sipper [2] results did not scale well when changing the search depth. Here we have an option to try and scale our results without changing the search depth, by changing the forward pruning parameters after the fact (i.e., after evolution). Runs 163 to 167 use a search depth of 6 with a maximum branching factor of 3. This low branching factor made possible evolving strong individuals. Now, by altering the *MaxBranchingFactor* parameter a posteriori in the best individuals from evolutionary runs we can create stronger players that search more thoroughly for more time, without having to pay the high cost of *evolving* them that way.

Table IX shows how well the best players from runs 163 to 167 fared against $\alpha\beta 7p$ when we increased their branching factor. As can be seen, the level of play increases as breadth of search increases (with but one exception, where a high level of play is achieved for *MaxBranchingFactor=4* and slightly goes down for *MaxBranchingFactor=5*).

Table X contains an analysis of players using a search depth of 6 and various maximal branching factors. As the table shows a maximal branching factor of 5 means more states are expanded than in a full search of depth 4. Time considerations may deter us from evolving this type of player directly, and the fact that players can easily be scaled post-evolutionarily is a very strong property of our system. Note that all these players conduct less search than our benchmark opponents that conduct a full search of depths 5 and 7.

TABLE X
COMPARISON OF GAME-STATE EXPANSION BETWEEN DIFFERENT PLAYERS WITH DIFFERENT BRANCHING FACTORS. THE "DEPTH 4 FULL" PLAYER IS THE WINNER OF RUN 161 AND SERVES AS A BASELINE. ALL PLAYERS USING A DEPTH OF 6 ARE BASED ON THE WINNER OF RUN 167. THE "DEPTH 5 FULL" AND "DEPTH 7 FULL" PLAYERS ARE BENCHMARK PLAYERS $\alpha\beta 5p$ AND $\alpha\beta 7p$, RESPECTIVELY.

Search parameters	Average # of States expanded per Turn	Standard Deviation
Depth 6 branching 3	162.34	25.36
Depth 6 branching 4	432.18	77.46
Depth 4 full	563.14	124.89
Depth 6 branching 5	925.21	198.16
Depth 5 full	1545.03	688.99
Depth 7 full	111328.99	72323.21

VII. CONCLUDING REMARKS AND FUTURE WORK

Expanding on our previous work we presented the genetic programming approach as a tool for discovering effective strategies for playing zero-sum, deterministic, full-knowledge board games. Using our extant GP gaming system, we introduced several tools that allow us to apply GP to evolving game players that use selective search with forward pruning and a heuristic evaluation function, with the search method itself being an evolvable feature. We have established that our approach yields players that achieve similar or better results than full-search players, with less actual search work. Our tool allows us to evolve fast players that look deeper into the game-tree than we could before. As these results scale with branching factor increase we can create even stronger players that search more thoroughly without paying the higher computational price for evolving them.

ACKNOWLEDGMENTS

Amit Benbassat is partially supported by the Lynn and William Frankel Center for Computer Sciences. This research was supported by the Israel Science Foundation (grant no. 123/11).

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

[3] ——, "Evolving board-game players with genetic programming," in *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, ser. GECCO '11. New York, NY, USA: ACM, 2011, pp. 739–742.

[4] ——, "Evolving competent players for multiple board games with little domain knowledge," *IEEE Transactions on Computational Intelligence and AI in Games*, submitted for publication.

[5] 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.

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

[7] 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*, vol. 3, December 2003, pp. 2222 – 2229 Vol.3.

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

[9] J. Gauci and K. Stanley, "Evolving neural networks for geometric game-tree pruning," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. ACM, 2011, pp. 379–386.

[10] R. D. Greenblatt, D. E. Eastlake, III, and S. D. Crocker, "The Greenblatt chess program," in *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, ser. AFIPS '67 (Fall). New York, NY, USA: ACM, 1967, pp. 801–810.

[11] T. Hauk, M. Buro, and J. Schaeffer, "*-minimax performance in backgammon," in *Computers and Games*, ser. Lecture Notes in Computer Science, H. van den Herik, Y. Bjrnsson, and N. Netanyahu, Eds. Springer Berlin / Heidelberg, 2006, vol. 3846, pp. 51–66.

[12] 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.

[13] ——, "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.

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

[15] K.-F. Lee and S. Mahajan, "The development of a world class othello program," *Artificial Intelligence*, vol. 43, no. 1, pp. 21 – 36, 1990.

[16] 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.

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

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

[19] P. S. Rosenbloom, "A world-championship-level othello program," *Artificial Intelligence*, vol. 19, no. 3, pp. 279 – 320, 1982. [Online]. Available: http://www.sciencedirect.com/science/article/B6TYF-47YY20N-1K/2/32f4945a2e00291e7af0eafd01900e34

[20] 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.

[21] 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.

[22] ——, "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.

[23] 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.

[24] 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.

[25] M. Sipper, *Evolved to Win*. Lulu, 2011, available at http://www.lulu.com/.

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

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