EvoMCTS: A Scalable Approach for General Game Learning

Amit Benbassat and Moshe Sipper

***Abstract*—We present the application of genetic programming as a generic game learning approach to zero-sum, deterministic, full-knowledge board games by evolving board-state evaluation functions to be used in conjunction with Monte Carlo Tree Search (MCTS). Our method involves evolving board-evaluation functions that are then used to guide the MCTS playout strategy. We examine several variants of Reversi, Dodgem, and Hex using strongly typed genetic programming, explicitly defined introns, and a selective directional crossover method. Our results show a proficiency that surpasses that of baseline handcrafted players using equal and in some cases a greater amount of search, with little domain knowledge and no expert domain knowledge. Moreover, our results exhibit scalability.**

***Index Terms*—Genetic programming, Board Games, Monte Carlo Methods, Search**

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. We apply tree-based genetic programming (GP) to evolve players for a number of games. Our guide in developing our design, 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 [50] and expanded by Dawkins [23], 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 [31]. 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 [48, 49]. 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 [49].

A. Benbassat and M. Sipper are with the Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel

In this paper our goal is entirely different. We do not aim to use a learning technique to master a single game but rather to present a flexible, *generic* tool that allows us to learn to play any member of a group of games possessing certain characteristics with as much—or as little—domain-specific knowledge at our disposal. In previous research we applied our generic evolutionary approach to multiple full-knowledge, deterministic, zero-sum board games, focusing on evolving players that use the alpha-beta search algorithm [10–13]. In this work we expand the applicability of our system by evolving players that use the MCTS search algorithm, which is a strong choice for use in games that alpha-beta is impractical for [3, 4, 30]. MCTS is also the leading approach in general game playing [15, 25].

Currently, our system can be applied to zero-sum, deterministic, full-knowledge games. Our system can in principle be adjusted to other types of games as well. We provide evidence for the effectiveness of our approach by using our system to learn multiple games. Our aim is to show that we can improve the play level through evolution, from total incompetence to competent play, even with little or no prior expert knowledge of the game domain. We also view this work as a possible stepping stone on the way to *General Game Learning* where, given a game and time to examine it, a learning algorithm can gradually evolve a search strategy well-suited to that game.

In Section II we present the various games we explored, including Reversi, Dodgem, and Hex variants. Section III is a short presentation of MCTS and the UCT algorithm. Section IV describes the benchmark players we devised, against which our evolving players compete. In Section V we discuss past research into MCTS as well as the different variants of games that we explored. Section VI contains a detailed description of the flexible system we designed to explore board games with GP. **Our approach was used to evolve board-evaluation functions that augment MCTS and guide the search by changing the algorithm's playout strategy.** We implemented a strongly typed GP framework that supports the use of multiple trees, and two different search algorithms both of which can be tuned with runtime parameters. We also implemented several types of genetic operators on top of the typical GP crossover and mutation operator. Section VII contains the results of our evolutionary runs and in Section VIII we demonstrate their scalability. In Section IX we expound our conclusions and insights on using GP to learn board games.

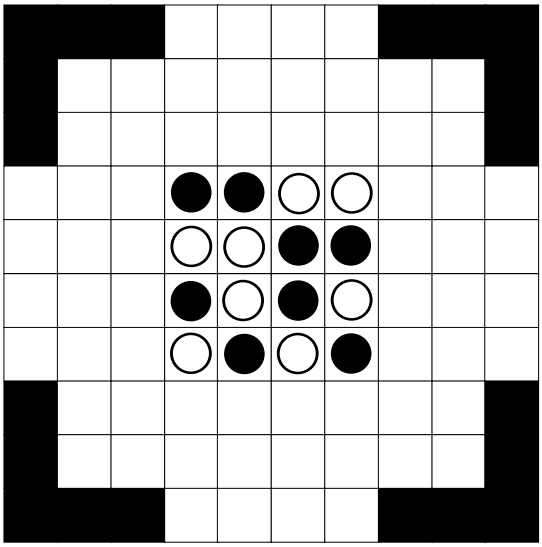


Fig. 1. Possible starting position in Reversi Plus. 16 pieces are randomly set up in the middle area of the board. 20 corner adjacent squares (blackened) are blocked and not in play.

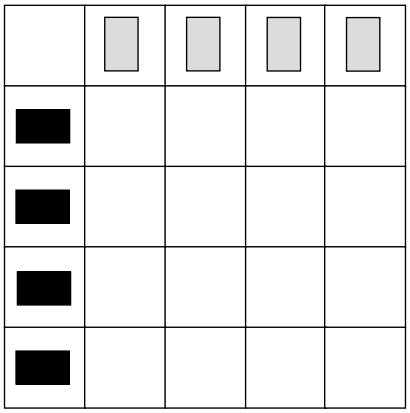


Fig. 2. 5x5 Dodgem. The board is initially set up with $n-1$ black cars along the left edge and $n-1$ white cars along the top edge, the top left square remaining empty. Players alternate turns, each allowed to move his vehicle forward or sideways. Cars may not move onto occupied spaces. They may leave the board, but only by a forward move. A car that leaves the board is out of the game. The winner is the player who first has no legal move on their turn because all their cars are either off the board or blocked in by their opponent.

II. The Games

The games we explore in this work are: Two Reversi variants, 5x5 Dodgem, and Hex on different board sizes. They are all zero-sum, deterministic, full-knowledge, two-player board games, played on an $n \times n$ board for some given n.

A. Reversi

Reversi, also known as Othello, is a popular game with a rich research history [34, 37, 41, 45, 46]. The most popular Reversi variant is a board game played on an 8x8 board. The players place their pieces on the board, 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 other popular games (e.g., Checkers and Chess). 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. The 10x10 variant of Reversi is also quite popular. International tournaments are held for both variants.

B. Reversi Plus

We also investigate a variant of Reversi of our own making dubbed *Reversi Plus*. This variant has the same basic rules and goal as Reversi but its starting position is different. This game is played on a 10x10 board; 20 squares on the board are blocked and no piece can be placed on them; and the 4 by 4 area in the middle of the board is initialized with a randomly generated pattern of 8 black pieces and 8 white pieces (we randomly select the 8 pieces on the left side of the area, and then create the opposite pattern on the right, in order to prevent any one of the players from having a structural advantage in the starting position). Figure 1 shows one possible Reversi Plus starting position.

C. Dodgem

Dodgem is an abstract strategy game played on an $n \times n$ board with $n-1$ cars for each player (Figure 2). Dodgem was first introduced as a 3x3 game by [14]. In spite of the small board size Dodgem is not a trivial game for human players. desJardins [24] proved, using exhaustive search, that though the first player can force a win in the 3x3 variant, the 4x4 and 5x5 variants are draw games assuming perfect play. desJardins also postulated that Dodgem is a draw game for any board size $n > 3$. In this work we explore the variant played on a 5x5 board.

D. Hex

Hex is a game played on a board shaped like a rhombus made of hexagons (easily converted into an $n \times n$ square board). Hex was invented by Danish mathematician Piet Hein in 1942 and then independently re-invented by American mathematician John Nash in 1947. It is typically played on an 11x11 or 19x19 board, with each of two players taking turns placing pieces on the board. The goal is to form a connected path of pieces linking the opposing sides of the board marked by one's colors, before one's opponent connects her sides in a similar fashion. Figure 3 shows the Hex starting position on an 11x11 board.

Along with Go, Hex serves as a classic example of high-branching-factor games where the traditional minmax-based approach fails to deliver and MCTS-based computer players outperform all other programs [4].

III. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a general method for making decisions in a given domain, initially proposed and developed by multiple research groups [18, 22, 35]. The idea of MCTS is to gradually build the domain search-tree by way of performing successive random playouts. In games, a game-tree is built one game-state at a time, with the next state to

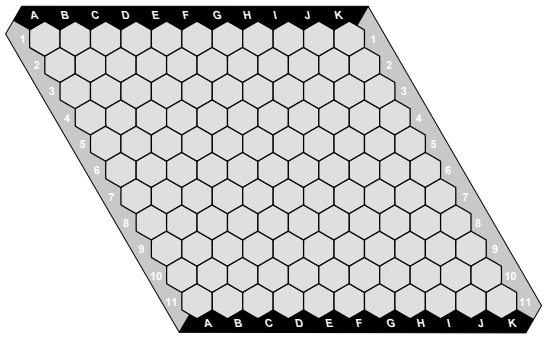


Fig. 3. 11x11 Hex. The board is initially empty. Players alternate turns placing pieces on the board. The black player's (gray player's) goal is to connect the two black (gray) sides of the board with black (gray) pieces.

be expanded and added to the game tree chosen according to the results of past random playouts (i.e., the partial game-tree is biased towards moves that yielded better results). This approach has proved useful in generating effective board game players and is responsible for the great improvement in level of play seen in games with a high branching factor such as Go [28, 30] and Hex [3]. MCTS is also the leading approach in the field of general game playing [15, 25]. For further references of MCTS research we recommend the survey paper by Browne et al. [16].

The MCTS algorithm can be seen as comprised of 3 steps that are repeated as many times as the time constraints allow (Figure 4):

1) Descend down the game tree using statistics recorded in the tree from previous playouts until an unvisited node *N* is encountered and added to the tree.
2) Evaluate node *N* by performing a quick simulation (or playout) and record the result.
3) Update the statistics of *N* and all of its ancestors in the tree in accordance with the result.

One of the better-known variants of MCTS is the *Upper Confidence Bounds applied to Trees* (UCT) algorithm [35]. UCT uses the following formula:

$$s_q(c) = \frac{W(c)}{n(c)} + C\sqrt{\frac{\log(N(q))}{n(c)}} \quad (1)$$

where:

- $s_q(c)$ is the score of child c of node q.
- $n(c)$ is the number of simulations of move c.
- $N(q)$ is the number of simulations of state q.
- $W(c)$ is the sum of scores for simulations of node c (in games this is often the number of won simulations).
- The constant C controls the compromise between exploitation of good moves and exploration of new moves

In order to choose a move from game state *q*, UCT performs an *argmax* operation as follows to select a child c for which the value of $s_q(c)$ is maximal:

$$\underset{c \in children(q)}{\operatorname{argmax}} \; s_q(c) \quad (2)$$

TABLE I
RELATIVE LEVELS OF PLAY FOR DIFFERENT BENCHMARK PLAYERS IN REVERSI. HERE AND IN TABLES II, III, AND IV: EACH LINE IN THE TABLE REPRESENTS A 10,000 GAME MATCH BETWEEN MCTS PLAYERS USING A DIFFERENT NUMBER OF PLAYOUTS. THE FIRST COLUMN REPRESENTS THE NUMBER OF PLAYOUTS USED BY BOTH PLAYERS. THE SECOND COLUMN IS THE WIN RATIO FOR THE FIRST PLAYER (E.G., A RATIO OF 0.6 MEANS 6,000 WINS). A DRAW COUNTS AS HALF A WIN.

Match	First player win Ratio
100 playouts vs 50 playouts	0.6996
200 vs 100	0.73055
400 vs 200	0.6781
800 vs 400	0.6529
1000 vs 400	0.6879
2000 vs 1000	0.6279

TABLE II
RELATIVE LEVELS OF PLAY FOR DIFFERENT BENCHMARK PLAYERS IN REVERSI PLUS.

Match	First player win Ratio
100 vs 50	0.67725
200 vs 100	0.7023
400 vs 200	0.7173
800 vs 400	0.71535
1000 vs 400	0.76395
2000 vs 1000	0.69775

IV. BENCHMARK PLAYERS

In order to test the quality of evolved players we need benchmark opponents. In this work we used standard MCTS players that used the UCT formula. We set the UCB constant to $C = 0.7$ and in all cases made sure that the baseline UCT player used to assess the value of evolved players employed the same parameters and optimizations as the evolved players (except for the evolved evaluation function, as described below). Before beginning the evolutionary experiments, we first evaluated our MCTS benchmark players by testing them against each other in matches of 10,000 games (with players alternating between playing either side). Tables I, II, III, and IV show the relative strengths of the different Reversi, Reversi Plus, 5×5 Dodgem, and 6×6 Hex players, respectively. As expected, MCTS players improve in level of play as the number of playouts increases.

We also conducted tests, omitted here for brevity's sake, to verify that our MCTS algorithm behaves similarly on Dodgem and Hex variants with larger boards.

TABLE III
RELATIVE LEVELS OF PLAY FOR DIFFERENT BENCHMARK PLAYERS IN 5×5 DODGEM.

Match	First player win Ratio
100 vs 50	0.7333
200 vs 100	0.7380
400 vs 200	0.7137
800 vs 400	0.6785
2000 vs 800	0.6899
4000 vs 2000	0.5956

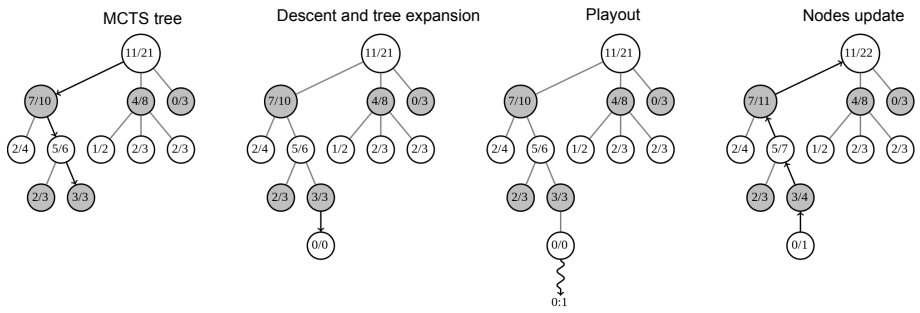


Fig. 4. An overview of Monte Carlo Tree Search.

TABLE IV
RELATIVE LEVELS OF PLAY FOR DIFFERENT BENCHMARK PLAYERS IN 6×6 HEX.

Match	First player win Ratio
100 vs 50	0.7102
200 vs 100	0.6235
400 vs 200	0.6763
800 vs 400	0.7759
1000 vs 400	0.8273
2000 vs 1000	0.6867

V. PREVIOUS WORK

Games attract considerable interest from AI researchers, with the field of evolutionary algorithms being no exception to this rule. Over the years many games have been tackled with the evolutionary approach. *Artificial Neural Networks* (ANN)-based American Checkers players were evolved by Chellapilla and Fogel [19, 20] using an evolutionary algorithm, their long runs resulting in expert-level play. GP was used by Azaria and Sipper [6] to evolve a strong Backgammon player. GP research by Hauptman and Sipper produced both competent players for Chess endgames [32] and an efficient solver for the Mate-in-N problem in Chess [33]. Our own work explored the applicability of GP to evolving players that use the alpha-beta search algorithm [10, 11], as well as evolving those players' search behavior [12, 13]. Gauci and Stanley [27] used the HyperNEAT system to evolve ANNs that act as search guides for the Cake American Checkers engine, resulting in an improved player. In our most recent work we expanded these results to MCTS [8]. **Baier and Winands [7] explored Monte-Carlo Tree Search and a Minimax Hybrid algorithm for the board games of Connect-4 and Breakthrough.**

Reversi has received its fair share of research attention. Early landmark work by Rosenbloom [46] yielded IAGO, an expert-level Reversi program. Subsequent work by Lee and Mahajan [37] greatly improved on IAGO's level of play by utilizing Bayesian learning to improve the player's evaluation function. 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 [41] to tackle the game of Reversi, resulting in a competent player that employed sophisticated mobility play. Chong et al. [21] presented a program using shallow search with evolved feed-forward ANNs encoded with board-spatial features as its board evaluation function. Hingston and Masek [34] presented an initial attempt at exploring the applicability of MCTS to Reversi in 2007, and also evolved MCTS Reversi players. In 2011 we expanded on our previous work and evolved strong board evaluation functions for 8x8 Reversi [11].

There has not been much work on *evolving* players that use MCTS. This is probably due in part to the fact that this algorithm is relatively new. Another possible reason may be the tendency to use MCTS with a high number of playouts to tackle long games with high branching factors in which traditional search algorithms fail. This results in slow search algorithms and makes the prospect of evolving players seem a very time-consuming task. Cazenave [17] used a limited *Genetic Programming* (GP) approach in order to evolve players for Go on small (7×7 and 9×9) boards that use an evolved formula to select nodes in the game tree. Cazenave's results improve on standard UCT and can be combined with other algorithmic improvements such as RAVE to generate competitive Go players on small boards. We first presented the approach in our own previous work [8], which includes some results in Reversi and Dodgem. Alhejali and Lucas [1] used GP to evolve decision trees for the simulation stage of MCTS-based MS Pac-Man players. Powley et al. [43, 44] presented work on agents that use heuristic planning as well as MCTS with macro-actions to skillfully navigate in the *Physical Traveling Salesman Problem (PTSP)*.

Besides its original presentation by Berlekamp et al. [14] we found little reference to Dodgem in the peer-reviewed scientific literature, other than results reported by [9]. 3x3 Dodgem was also implemented in GAMESMAN, presented by Garcia [26].

Hex serves as a good example of a game with a high branching factor in which the traditional search approach fails to deliver. Much research has gone into Hex and trying to create strong Hex-playing programs (e.g., [2, 51]). Currently, the dominant Hex player is MCTS based and uses board patterns to aid in the default search behavior [3, 4].

VI. 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

TABLE V
BASIC TERMINAL NODES. F: FLOATING POINT, B: BOOLEAN. AN ERC RETURNS A VALUE THAT IS CHOSEN RANDOMLY (IN OUR CASE FROM THE RANGE $[-5, 5)$) WHEN THE NODE IS CREATED.

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

TABLE VI
GAME-ORIENTED TERMINAL NODES THAT DEAL WITH BOARD CHARACTERISTICS.

Node name	Type	Return value
`EnemyManCount()`	F	The enemy's man count
`FriendlyManCount()`	F	The player's man count
`FriendlyPieceCount()`	F	The player's piece count
`ManCount()`	F	`FriendlyManCount()` – `EnemyManCount()`
`Mobility()`	F	The number of moves available to the player

player whose turn it is to play. The evolutionary algorithm was written in Java. We chose to implement a strongly typed GP framework [40] 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 [36], another form of crossover was implemented—which we designated "selective crossover"—as well as a local mutation operator. The original setup is detailed in [10]. 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.

A. Basic terminal nodes

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

B. Game-oriented terminal nodes

The game-oriented terminal nodes are listed in several tables. Table VI shows nodes defining characteristics that have to do with the board in its entirety, and Table VII shows nodes defining characteristics of a certain square on the board. We originally created this setup for Lose Checkers (see Benbassat and Sipper [10]), but many of the domain terminals we used for Checkers variants also proved useful for several other games—a point which is at the heart of our approach.

TABLE VII
GAME-ORIENTED TERMINAL NODES THAT DEAL WITH SQUARE CHARACTERISTICS. THEY RECEIVE TWO PARAMETERS—X AND Y—THE ROW AND COLUMN OF THE SQUARE, RESPECTIVELY.

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

TABLE VIII
REVERSI-SPECIFIC TERMINAL NODES.

Node name	Type	Return value
`FriendlyCornerCount()`	F	Number of corners in friendly control
`EnemyCornerCount()`	F	Number of corners in enemy control
`CornerCount()`	F	`FriendlyCornerCount()` – `EnemyCornerCount()`

TABLE IX
DODGEM-SPECIFIC TERMINAL NODES.

Node name	Type	Return value
`FriendlyPosCount()`	F	Distance measure from victory for friendly player
`EnemyPosCount()`	F	Distance measure from victory for enemy player
`PosCount()`	F	`FriendlyPosCount()` – `EnemyPosCount()`

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 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 games. We believe that the domain-specific nodes discussed below also use little in the way of human knowledge about the games. This goes against what has traditionally been done when GP is applied to board games [6, 32, 33]. This is partly due to the difficulty in finding useful board attributes for evaluating game states for some board games, but there is another, more fundamental, reason: Not introducing game-specific knowledge into the domain-specific nodes means the GP setup defined is itself not game specific, and thus more flexible.

1) Reversi-specific terminal nodes: Reversi-specific terminals, shown in Table VIII, essentially deal with corners. These terminals constitute use of domain knowledge—but at a very basic level. Any human playing Reversi quickly realizes that the corner squares are highly significant.

For Reversi Plus we used the same terminal nodes except we counted all the 12 corners in the Reversi Plus board. We did this because we had a general feeling that the corners might still be somewhat important in Reversi Plus.

2) Dodgem-specific terminal nodes: Dodgem-specific terminals, shown in Table IX, essentially return a distance measure from victory for the players. As each player in Dodgem is attempting to move her pieces from one side of the board to the other, a natural metric for measuring a board state is to check how close the pieces are to the target edge of the board. Again, the level of domain knowledge is very basic.

3) Hex-specific terminal nodes: In Hex the evolved board-state evaluators do not evaluate the entire board but rather just the square being considered for the current move and its 2-neighborhood. We chose this approach in order to evolve evaluation functions that work independently of board size. This attribute affects the way the general terminals behave in

TABLE X
HEX-SPECIFIC TERMINAL NODES. X AND Y STAND FOR A BOARD SQUARE IN THE 2-NEIGHBORHOOD OF THE CURRENTLY CONSIDERED MOVE.

Node name	Type	Return value
`IsEnemyContact()`	B	True if current move position is a contact position for enemy player
`IsFriendlyContact()`	B	True if current move position is a contact position for friendly player
`IsEmptySquare(X,Y)`	B	True if square empty
`IsFriendlyPiece(X,Y)`	B	True if square occupied by friendly piece
`IsEnemyPiece(X,Y)`	B	True if square occupied by enemy piece
`IsManPiece(X,Y)`	B	True if square occupied by man

TABLE XI
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 if $F_1 \leq F_2$
`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
`IfT`(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	$F_1 + F_2$

the Hex game (e.g., the counting terminals only count pieces in the 2-neighborhood of the current move location). There are also some square-specific terminal nodes (see Table X).

C. Function nodes

We defined several basic domain-independent functions, presented in Table XI, and no domain-specific functions.

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 5 we see an example of a Reversi 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 (3.4 is an `ERC` terminal as defined in Table V).

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, one-way crossover is done by randomly selecting a subtree

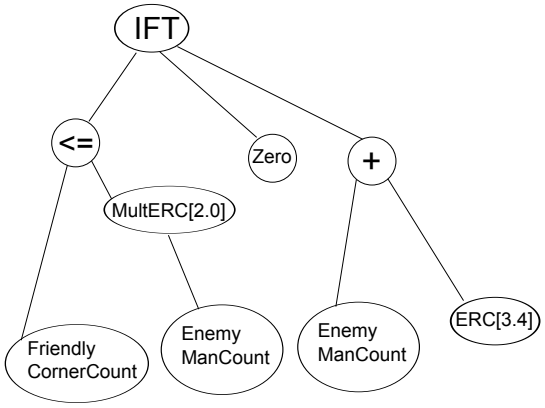


Fig. 5. Example of a subtree for the game of Reversi.

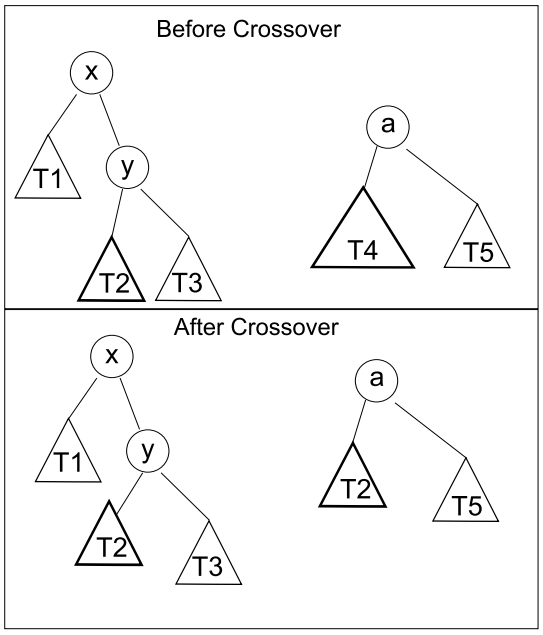


Fig. 6. One-way crossover: Subtree T2 in donor tree (left) replaces subtree T4 in receiver tree (right). The donor tree remains unchanged.

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 6. This type of crossover is similar to the asymmetric single-parent crossover operator defined by Ashlock and Ashlock [5] except that in the case of the single-parent operator the donated genetic information comes from an archive of unchanging individuals and not from cohorts in the population.

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.

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

Using the vantage point of the gene-centered view of evolution it is easier to see the logic of crossover in our 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 in many cases has an effect on the return value, is likely to radically change the individual's fitness. In order to afford local mutation with limited effect we modified the GP setup as follows: 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.

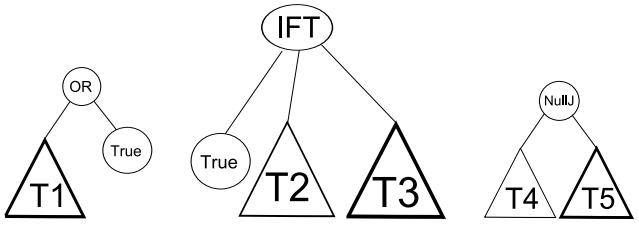
Introns occur naturally in GP, provided that the function and terminal sets allow for it. As bloat progresses, the number of nodes that are part of introns tends to increase. Luke [39] distinguished two types of subtrees that are sometimes referred to as introns in the literature:

- *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 and we will do the same because unoptimized code seems to cast too wide a net to be of much use in our case, and also wanders too far from the original meaning of the term "intron" in biology. We also make another distinction between two types of inviable code introns:

- *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.
- *Dead-code introns*: Subtrees whose code is never run.

Figure 7 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.



(a) Live-code intron (b) Implicit dead-code intron (c) Explicitly defined dead-code intron

Fig. 7. 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 system 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 7, $T5$ differs from $T3$ in that $T5$ is known to be an intron the moment $NullJ$ is reached, and therefore the system can take this into account. In our system, when converting individuals into C code the EDIs were simply ignored, a feat that could be accomplished with ease as they were dead-code introns that were easy to find.

Nordin et al. [42] explored EDIs in linear GP, finding that they tended 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 [38]. Our search of the literature discovered no exploration of EDIs in tree-based GP prior to [10], 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.

G. Fitness calculation

Fitness calculation in our system is 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 [47], and is referred to below as the coevolution round. While in our previous work we used both guides and coevolution, herein we used coevolution alone to evaluate fitness.

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

Our evaluation method 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 coevolutionary 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 several types of guides, including a random player and different search-based players. The random player chose a move at random and was used to test initial runs. The search-based players used a hand-coded search-based approach. 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—according to partial fitness values from games already played—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.

H. 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 as follows: Of several individuals chosen at random, copies of a subset of fitter individuals were 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 scores, 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.

I. EvoMCTS Players

Our system supports several kinds of GP players (see [9, 10]). The results we present in this paper involve evolved players that use MCTS as their basic search algorithm. We dubbed the players evolved using our system *EvoMCTS players* and refer to them thus henceforth.

Our evolutionary system evolves GP players that use the MCTS algorithm. We implemented a UCT variant of MCTS. The parameters we can tune control the number of playouts used before each move, the initial value of unexplored nodes in the game tree (in our implementation of the standard MCTS this value is 0, leading to unexplored game states always being favored; tuning this parameter is akin to using a limited version of *First Play Urgency* as defined by Gelly and Wang [29]), the C constant from the UCT formula (Equation 1), and a parameter used to enhance search by having players remember the search tree from previous turns (this way MCTS gains some of the playouts from its previous turns "for free"). We decided on values for these parameters empirically in order to define better players. Based on this we can define handcrafted MCTS players to be used as yardsticks to test evolved players against.

The EvoMCTS players use the same MCTS parameters as the handcrafted player. Instead of using random playouts, the players use evolved board evaluation functions in the following fashion: An additional parameter dubbed *playoutBranchingFactor* is used in the EvoMCTS players. Before each simulated move in the playout, the players evaluate *playoutBranchingFactor* randomly chosen legal moves and select the move evaluated as best by the evolved evaluation function. Algorithm 4 describes how EvoMCTS players' playouts work in our system. In order to allow even the moves evaluated as bad a chance to be selected, *playoutBranchingFactor* is a maximum value of moves to be considered. With a low probability the algorithm can choose the same move more than once, thus allowing even the move evaluated as worst a chance to be chosen. We did this because of the inherent limitation of even the best fast evaluation functions that sometimes fail to correctly assess the value of a board state.

J. Summary of Run Parameters

- Number of generations: 100
- Population size: 120
- Value of *CoPlayNum* in fitness calculation: 25
- Crossover probability: 0.8
- Mutation probability: 0.2
- Local mutation ratio: 0.5
- Selection Method: Tournament selection with tournament size 2 and 1 tournament winner
- Maximum depth of GP tree: 15
- Number of playouts used by evolved players: 50, 100, or 200

Algorithm 4 Evo_Playout(Node,playoutBranchingFactor)

```
repeat
  VAL ← −∞
  for i ← 1 to playoutBranchingFactor do
    Select at random a move r from game-state Node.
    // EvoEval() is the evolved evaluation function.
    if EvoEval(r) > VAL then
      VAL ← EvoEval(r)
      ChosenMove ← r
    end if
  end for
  Node ← ChosenMove
until Node is a final game-state
// GameResult() returns information about game winner
return GameResult(Node)
```

TABLE XII
REVERSI: RESULTS OF TOP RUNS. *EvoMCTS Player* USES MCTS WITH 100 PLAYOUTS COUPLED WITH EVOLVED EVALUATION FUNCTION, WHILE *Benchmark Opponents* USE STANDARD MCTS. HERE AND IN THE SUBSEQUENT TABLES: *MCTSi* REFERS TO A STANDARD MCTS PLAYER USING *i* PLAYOUTS; BENCHMARK SCORES ARE THE NUMBER OF WINS OUT OF 1000 GAMES (A DRAW COUNTS AS HALF A WIN).

Run identifier	Benchmark Score vs MCTS100	Benchmark Score vs MCTS200
172	759.0	521.5
173	701.5	522.5
176	717.0	482.5
177	730.0	505.0
178	727.0	530.0
179	719.0	529.0

- UCT parameter C: 0.7
- Runs use the option of remembering relevant parts of the game tree from previous moves (except in Hex)
- Number of evaluated moves in playout: 4 or 5

VII. RESULTS

In all evolutionary Reversi and Reversi Plus runs that follow we used 14–16 cores (depending on availibiliy) 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 3–5 days. In all evolutionary Dodgem and Hex runs that follow we used a personal computer with an ASUS SABERTOOTH 990FX board with an AMD Phenom II X6 1100T @ 3400MHz 6-core processor with 6MB L3 cache and 16GB RAM. Runs took 1–2 days.

The results we show in this section are from runs conducted after some manual parameter tuning. We only show results from runs we conducted after this tuning. We refer to these runs as the "best runs". Other runs with unsuccessful parameter setups are not reported here.

Table XII shows the results from some of our best Reversi runs. The table clearly demonstrates that our players not only beat the Standard MCTS player that uses the same number of playouts, but also hold their own against a much stronger MCTS player that uses *twice* as many playouts. Table XIII shows 95% confidence intervals for the benchmark scores in Table XII.

TABLE XIII
REVERSI: 95% CONFIDENCE INTERVALS FOR BENCHMARK SCORES.

Run identifier	vs MCTS100	vs MCTS200
172	718.0–800.0	477.0–566.0
173	659.0–744.0	478.0–567.0
176	675.0–759.0	438.0–527.0
177	688.0–772.0	461.0–549.0
178	685.0–769.0	486.0–574.0
179	677.0–761.0	485.0–573.0

TABLE XIV
REVERSI PLUS: RESULTS OF TOP RUNS. *EvoMCTS Player* USES MCTS WITH 50 PLAYOUTS COUPLED WITH EVOLVED EVALUATION FUNCTION, WHILE *Benchmark Opponents* USE STANDARD MCTS.

Run identifier	Benchmark Score vs MCTS50	Benchmark Score vs MCTS100
186	693.5	521.5
187	652.0	485.5
188	731.5	564.5
189	668.0	486.5

Table XIV shows the results from some of our best Reversi Plus runs. Once again the table demonstrates that our players beat the Standard MCTS player that uses the same number of playouts and hold their own against a much stronger MCTS player that uses twice as many playouts. Table XV shows 95% confidence intervals for the benchmark scores in Table XIV.

Table XVI shows the results from some of our best Dodgem runs. The table demonstrates that EvoMCTS Dodgem players outperform both the MCTS player that uses the same number of playouts, and the much stronger MCTS player that uses twice as many playouts, by a wide margin. Table XVII shows 95% confidence intervals for the benchmark scores in Table XVI.

Table XVIII shows the results from some of our best Hex runs. The table demonstrates that EvoMCTS Hex players outperform the MCTS player that uses the same number of playouts playing Hex on a 6x6 board. Table XIX shows 95% confidence intervals for the benchmark scores in Table XVIII.

TABLE XV
REVERSI PLUS: 95% CONFIDENCE INTERVALS FOR BENCHMARK SCORES.

Run identifier	vs MCTS50	vs MCTS100
186	651.0–736.0	477.0–566.0
187	609.0–695.0	441.0–530.0
188	690.0–773.0	520.0–609.0
189	625.0–711.0	442.0–531.0

TABLE XVI
DODGEM: RESULTS OF TOP RUNS. *EvoMCTS Player* USES MCTS WITH 100 PLAYOUTS COUPLED WITH EVOLVED EVALUATION FUNCTION, WHILE *Benchmark Opponents* USE STANDARD MCTS.

Run identifier	Benchmark Score vs MCTS100	Benchmark Score vs MCTS200
180	865.0	731.0
181	920.0	822.0
182	880.0	745.0
183	920.0	821.0
184	814.0	634.0
185	884.0	773.0

TABLE XVII
DODGEM: 95% CONFIDENCE INTERVALS FOR BENCHMARK SCORES.

Run identifier	vs MCTS100	vs MCTS200
180	828.0–902.0	689.0–773.0
181	887.0–953.0	783.0–861.0
182	844.0–916.0	704.0–786.0
183	887.0–953.0	782.0–860.0
184	775.0–853.0	590.0–678.0
185	848.0–920.0	732.0–814.0

TABLE XVIII
6×6 HEX: RESULTS OF TOP RUNS. *EvoMCTS Player* USES MCTS WITH 200 PLAYOUTS COUPLED WITH EVOLVED EVALUATION FUNCTION, WHILE *Benchmark Opponent* USES STANDARD MCTS.

Run identifier	Benchmark Score vs MCTS200
192	609.0
193	710.0
194	679.0
195	636.0

VIII. SCALABILITY OF RESULTS

Using MCTS with 50–200 playouts is fine if what one wants is a fast player with basic game proficiency. It is also necessary to limit the number of playouts when evolving players play thousands of games in every generation. But to obtain strong players more playouts are needed. Just as the standard MCTS players can be tuned and improved by increasing the number of playouts (see Table I) so too can our evolved players.

Table XX shows how a top evolved Reversi player maintains its advantage when the number of playouts is scaled up post-evolutionarily.

Table XXI shows how a top evolved Reversi Plus player maintains its advantage when the number of playouts is scaled up.

Table XXII shows how a top evolved Dodgem player maintains a clear advantage when the number of playouts is scaled up.

Table XXIII shows how a top evolved Hex player maintains a clear advantage when the number of playouts is scaled up. In the case of Hex we also scaled the board size. Hex isn't often played on a 6x6 board by humans, but our choice of local domain terminals makes our evolved evaluation functions oblivious to the size of the board, and potentially renders the EvoMCTS players scalable to larger boards. Table XXIV shows how the same top evolved Hex player maintains its advantage when the board size is increased post-evolutionarily.

TABLE XIX
6×6 HEX: 95% CONFIDENCE INTERVALS FOR BENCHMARK SCORES.

Run identifier	vs MCTS200
192	565.0–653.0
193	668.0–752.0
194	636.0–722.0
195	593.0–679.0

TABLE XXIV
EVOMCTS HEX PLAYER OF TABLE XXIII EVOLVED WITH 200 PLAYOUTS ON A 6×6 BOARD, PLAYING ON LARGER BOARD SIZES AGAINST STANDARD MCTS USING THE SAME NUMBER OF PLAYOUTS. THE FIRST COLUMN REPRESENTS THE NUMBER OF PLAYOUTS USED BY THE EVOMCTS PLAYER. THE SECOND COLUMN REPRESENTS THE NUMBER OF PLAYOUTS USED BY THE MCTS BENCHMARK PLAYER. THE THIRD COLUMN REPRESENTS THE SIZE OF THE BOARD.

No. Playouts EvoMCTS Player	No. Playouts MCTS Player	Board Size	EvoMCTS Player Benchmark Score	95% confidence interval
1000	1000	6x6	716.0	674.0–758.0
1000	1000	8x8	723.0	681.0–765.0
1000	1000	11x11	692.0	649.0–735.0
1000	1000	14x14	630.0	586.0–674.0

TABLE XX
AN EVOMCTS REVERSI PLAYER (RUN NO. 172) USING DIFFERENT PLAYOUT VALUES PLAYING AGAINST STANDARD MCTS USING EITHER THE SAME NUMBER OF PLAYOUTS OR TWICE AS MANY PLAYOUTS. NOTE THAT EVOMCTS EVOLVED WITH ONLY 100 PLAYOUTS. HERE AND IN TABLES XXI, XXII, AND XXIII: THE FIRST COLUMN REPRESENTS THE NUMBER OF PLAYOUTS USED BY THE EVOMCTS PLAYER. THE SECOND COLUMN REPRESENTS THE NUMBER OF PLAYOUTS USED BY THE MCTS BENCHMARK PLAYER.

No. Playouts EvoMCTS Player	No. Playouts MCTS Player	Benchmark Score	95% confidence interval
100	100	759.0	718.0–800.0
100	200	521.5	477.0–566.0
200	200	755.0	714.0–796.0
200	400	628.5	585.0–672.0
400	400	747.0	706.0–788.0
400	800	665.0	622.0–708.0
1000	1000	781.0	741.0–821.0
1000	2000	662.5	619.0–706.0

TABLE XXI
AN EVOMCTS REVERSI PLUS PLAYER (RUN NO. 188) USING DIFFERENT PLAYOUT VALUES PLAYING AGAINST STANDARD MCTS USING EITHER THE SAME NUMBER OF PLAYOUTS OR TWICE AS MANY PLAYOUTS. EVOMCTS EVOLVED WITH ONLY 50 PLAYOUTS.

No. Playouts EvoMCTS Player	No. Playouts MCTS Player	Benchmark Score	95% confidence interval
50	50	731.5	690.0–773.0
50	50	564.5	520.0–609.0
100	100	732.0	690.0–774.0
100	200	532.0	488.0–576.0
200	200	743.0	702.0–784.0
200	400	567.5	523.0–612.0
400	400	727.0	685.0–769.0
400	800	565.0	521.0–609.0
1000	1000	734.0	692.0–776.0
1000	2000	607.0	563.0–651.0

TABLE XXII
AN EVOMCTS DODGEM PLAYER (RUN NO. 181) USING DIFFERENT PLAYOUT VALUES PLAYING AGAINST STANDARD MCTS EITHER THE SAME NUMBER OF PLAYOUTS OR TWICE AS MANY PLAYOUTS. EVOMCTS EVOLVED WITH ONLY 100 PLAYOUTS.

No. Playouts EvoMCTS Player	No. Playouts MCTS Player	Benchmark Score	95% confidence interval
100	100	920.0	887.0–953.0
100	200	822.0	783.0–861.0
200	200	905.0	871.0–939.0
200	400	807.0	768.0–846.0
400	400	879.0	843.0–915.0
400	800	777.0	737.0–817.0
2000	2000	756.0	715.0–797.0
2000	4000	649.5	606.0–693.0

TABLE XXIII
AN EVOMCTS HEX PLAYER (RUN NO. 193) USING DIFFERENT PLAYOUT VALUES PLAYING AGAINST STANDARD MCTS USING THE SAME NUMBER OF PLAYOUTS. EVOMCTS EVOLVED WITH ONLY 200 PLAYOUTS.

No. Playouts EvoMCTS Player	No. Playouts MCTS Player	Benchmark Score	95% confidence interval
200	200	710.0	668.0–752.0
400	400	647.0	604.0–690.0
1000	1000	716.0	674.0–758.0

A. Comparing Evolved Players to Alpha-Beta Players

Since MCTS is not often the search algorithm used in computer players for games with low branching factors we decided to test our best evolved and scaled-up Reversi and Dodgem players against handcrafted Alpha-Beta-based players we used in previous research [8, 9]. In order to avoid a technical difficulty that caused games to occasionally crash, we chose to disable one of the optimizations in our EvoMCTS players. In the results described below the EvoMCTS players do not use the optimization that allows them to remember relevant parts of the game tree from previous moves.

We pitted the evolved Reversi player scaled up in Table XX to use 1000 playouts against an Alpha-Beta player that looks 5 moves ahead. In a 1000-game match the EvoMCTS player obtained a benchmark score of 883.5 (848.0–919.0 95% confidence interval), clearly outperforming its opponent.

We pitted the evolved Reversi player scaled up in Table XXII to use 2000 playouts against an Alpha-Beta player that looks 5 moves ahead. In a 1000-game match the EvoMCTS player obtained a benchmark score of 470.0 (426.0–514.0 95% confidence interval). **This result shows that the evolved player is competitive with the Alpha-Beta player, the two playing at an approximately equal level**. In a match against an Alpha-Beta player that looks 3 moves ahead the EvoMCTS player obtained a benchmark score of 948.0 (918.0–978.0 95% confidence interval), clearly outperforming its opponent.

IX. CONCLUDING REMARKS

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 several distinct nontrivial board games.

We presented genetic programming as a tool for discovering effective strategies for playing several zero-sum, deterministic, full-knowledge board games using the MCTS search algorithm, expanding on previous results with Alpha-Beta. As the results show, using GP to evolve heuristic board evaluation functions for playouts has proved useful in improving MCTS players in several different game domains.

The scalability of results means that although time constraints render our evolutionary approach limited to producing only very fast players, we can later improve those evolved players offline by increasing the number of playouts employed by the MCTS algorithm. Our choice to focus on local patterns in Hex allowed us to scale the board size as well, having fast players that evolved on a small board and played short games then play well on larger board variants in which the games are longer. This approach is in principle applicable to other games, most notably Go.

In addition to being game-nonspecific, EvoMCTS is also to a great degree algorithm-nonspecific within the MCTS algorithm family. We used our method in conjunction with the standard UCT algorithm (with an added enhancement of game-tree memory) for which we hand-tuned some parameters. This method can, however, be used together with a more specialized game-specific version of MCTS that navigates the game tree in any other way. As our method focuses on evolving playout behavior it is indifferent to changes in the particulars of the MCTS implementation.

Our approach now affords the evolution of players in a variety of games using two very different approaches to search. It is also search-algorithm flexible in that we can use our system with another search algorithm (e.g., *Expectimax*, *Max*^{n}).

With EvoMCTS being based on the search algorithm that is currently the leading approach in General Game Playing, we suggest that our system may be seen as something very close to a *General Game Learning* system. By this we mean a system that receives a game, and with some or no guidance from the human developer can learn how to improve in said game. Any two-player, zero-sum game can in principle be explored by our system, as long as there is access to its game-state features. Herein we tackled a plethora of board games because they share many features, but our system can work on games with no board. Though we dealt here with two-player, zero-sum games our system can be adapted to deal with n-player and nonzero-sum games.

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

1) Applying our system to other board games. The flexibility of the system makes it easy to learn how to play more games, and since we do not require expert knowledge new games can easily be accommodated.
2) Applying our system to non-zero game domains, and thus exploring evolution's ability to evolve complex playing strategies in a non-zero sum environment.
3) Our system can also be applied to more-complicated games that are not full-knowledge, or contain a stochastic element, or are n-player games. This applies to many turn-based computer strategy games, and is also a better approximation of real-world problems. Along this vein we are currently exploring the application of the EvoMCTS approach to Backgammon.

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 our system to evolve a strong problem-solving program in a game environment.

ACKNOWLEDGMENTS

The authors would like to thank Neta-Li Robman and Ben Shalom for their implementation of the MCTS code used in this work. 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] A. M. Alhejali and S. M. Lucas, "Using genetic programming to evolve heuristics for a monte-carlo tree search ms pac-man agent," in *2013 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2013, pp. 57–64.

[2] V. V. Anshelevich, "A hierarchical approach to computer Hex," *Artificial Intelligence*, vol. 134, no. 1, pp. 101–120, 2002.

[3] B. Arneson, R. Hayward, and P. Henderson, "MoHex wins Hex tournament," *ICGA journal*, vol. 32, no. 2, p. 114, 2009.

[4] B. Arneson, R. B. Hayward, and P. Henderson, "Monte Carlo tree search in Hex," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 251–258, 2010.

[5] W. Ashlock and D. Ashlock, "Single parent genetic programming," in *The 2005 IEEE Congress on Evolutionary Computation*, vol. 2. IEEE, 2005, pp. 1172–1179.

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

[7] H. Baier and M. H. M. Winands, "Monte-carlo tree search and minimax hybrids," in *2013 IEEE Conference on Computational Intelligence in Games (CIG)*. IEEE, 2013, pp. 1–8.

[8] A. Benbassat and M. Sipper, "Evomcts: Enhancing mcts-based players through genetic programming," in *2013 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2013, pp. 57–64.

[9] A. Benbassat, A. Elyasaf, and M. Sipper, "More or less? two approaches to evolving game-playing strategies," in *Genetic Programming Theory and Practice X (GPTP 2012)*, R. Riolo, E. Vladislavleva, M. D. Ritchie, and J. H. Moore, Eds. Heidelberg: Springer-Verlag, 2013.

[10] 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.

[11] ——, "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.

[12] ——, "Evolving both search and strategy for reversi players using genetic programming," in *2012 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2012, pp. 47–54.

[13] ——, "Evolving players that use selective game-tree search with genetic programming," in *Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference Companion*. ACM, 2012, pp. 631–632.

[14] E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways for your Mathematical Plays*. New York, NY, USA: Academic Press, 1982.

[15] Y. Bjornsson and H. Finnsson, "Cadiaplayer: A simulation-based approach to general game playing," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 1, pp. 4–15, 2009.

[16] C. Browne, E. J. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *IEEE Trans. Comp. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, 2012.

[17] T. Cazenave, "Evolving Monte-Carlo Tree Search Algorithms," Univ. Paris 8, Dept. Inform., Tech. Rep., 2007.

[18] G. Chaslot, M. Winands, H. van den Herik, J. Uiterwijk, and B. Bouzy, "Progressive strategies for Monte Carlo tree search," *New Mathematics and Natural Computation*, vol. 4, no. 3, pp. 343–357, 2008.

[19] 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.

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

[21] 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.

[22] R. Coulom, "Efficient selectivity and backup operators in Monte Carlo tree search," *Computers and Games*, pp. 72–83, 2007.

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

[24] D. desJardins. (1996, March) "Dodgem" . . . any info? [Online]. Available: http://www.ics.uci.edu/~eppstein/cgt/dodgem.html

[25] H. Finnsson and Y. Björnsson, "Simulation-based approach to general game playing," *The Twenty-Third AAAI Conference on Artificial Intelligence*, pp. 259–264, 2008.

[26] D. D. Garcia, "Gamesman," Ph.D. dissertation, University of California, 1990.

[27] 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.

[28] S. Gelly and D. Silver, "Monte-carlo tree search and rapid action value estimation in computer go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, 2011.

[29] S. Gelly and Y. Wang, "Exploration exploitation in go: Uct for monte-carlo go," 2006.

[30] S. Gelly, Y. Wang, R. Munos, and O. Teytaud, "Modification of UCT with patterns in Monte Carlo go," INRIA, Tech. Rep. 6062, 2006.

[31] 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.

[32] 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.

[33] ——, "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.

[34] P. Hingston and M. Masek, "Experiments with monte carlo othello," in *IEEE Congress on Evolutionary Computation*. IEEE, 2007, pp. 4059–4064.

[35] L. Kocsis and C. Szepesvári, "Bandit based Monte Carlo planning," *Machine Learning: ECML 2006*, pp. 282–293, 2006.

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

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

[38] 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.

[39] 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.

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

[41] D. E. Moriarty and R. Miikkulainen, "Discovering complex Othello strategies through evolutionary neural net-

works," *Connection Science*, vol. 7, no. 3, pp. 195–210, 1995.

[42] 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.

[43] E. J. Powley, D. Whitehouse, and P. I. Cowling, "Monte carlo tree search with macro-actions and heuristic route planning for the physical travelling salesman problem," in *IEEE Conference on Computational Intelligence and Games*. IEEE, 2012, pp. 234–241.

[44] ——, "Monte carlo tree search with macro-actions and heuristic route planning for the multiobjective physical travelling salesman problem," in *IEEE Conference on Computational Intelligence in Games*. IEEE, 2013, pp. 1–8.

[45] D. Robles, P. Rohlfshagen, and S. M. Lucas, "Learning non-random moves for playing othello: Improving monte carlo tree search," in *IEEE Conference on Computational Intelligence and Games*. IEEE, 2011, pp. 305–312.

[46] P. S. Rosenbloom, "A world-championship-level Othello program," *Artificial Intelligence*, vol. 19, no. 3, pp. 279 – 320, 1982.

[47] 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.

[48] 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.

[49] 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.

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

[51] J. Yang, S. Liao, and M. Pawlak, "New winning and losing positions for $7\times$ 7 Hex," in *Computers and games*. Springer, 2003, pp. 230–248.