EvoMCTS: Enhancing MCTS-Based Players through Genetic Programming

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

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

***Abstract*—We present EvoMCTS, a genetic programming method for enhancing level of play in games. Our work focuses on the zero-sum, deterministic, perfect-information board game of Reversi. Expanding on our previous work on evolving board-state evaluation functions for alpha-beta search algorithm variants, we now evolve evaluation functions that augment the MTCS algorithm. We use strongly typed genetic programming, explicitly defined introns, and a selective directional crossover method. Our system regularly evolves players that outperform MCTS players that use the same amount of search. Our results prove scalable and EvoMCTS players whose search is increased offline still outperform MCTS counterparts. To demonstrate the generality of our method we apply EvoMCTS successfully to the game of Dodgem.**

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 perfect-information, deterministic, zero-sum board games.

We apply tree-based Genetic Programming (GP) to evolving players for Reversi. Our guide in developing our algorithm parameters, aside from previous research into MCTS 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 to. We are mindful of evolutionary theory, particularly as pertaining to the gene-centered view of evolution. This view, presented by Williams [32] and expanded by Dawkins [14], 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, using techniques such as opening books [19] and endgame databases [29, 30]. Conversely, our focus has been on multi-game generality [5–7]. Using our game system, which we have demonstrated to be flexible and easily applicable to multiple games, we choose to avoid using specialized techniques and expert domain knowledge in favor of generic, easily transferable evolutionary tecniques. It is with this view in mind that we present EvoMCTS, a new method for enhancing *Monte-Carlo Tree Search* (MCTS) game players.

Section II contains some basic information on Reversi. Section III is a short presentation of MCTS and the UCT algorithm. Section IV presents previous work related to ours. Section V explains the genetic programming system we use and how we apply it to games. In Section VI we present the results of our evolutionary runs and in Section VII we demonstrate their scalability. In section VIII we demonstrate the flexibility and generality of our method by showing how the whole process can be repeated for the very different board game of Dodgem. Section IX contains conclusions and discussion of results.

II. Reversi

Reversi, also known as Othello, is a popular game with a rich research history [1, 10, 24, 27]. 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. 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 [12, 13, 23]. 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 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

978-1-4673-5311-3/13/$31.00 ©2013 IEEE

players and is responsible for the great improvement in level of play seen in games with a high branching factor such as Go [18] and Hex [2]. MCTS is also the leading approach in the field of general game playing [9, 16].

The MCTS algorithm can be seen as comprised of 3 steps that are repeated over and over again as many times as the time constraints allow:

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 preforming a quick simulation (or playout) and recording the result.
3) Update the statistics of *N* and all of its ancestors in the tree in accordance with the result.

See MCTS steps in Figure 1 below.

A. The UCT Algorithm

One of the better-known variants of MCTS is the *Upper Confidence Bounds applied to Trees* (UCT) algorithm [23]. 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.
- $W(n)$ is the sum of scores for simulations of node n (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 preforms an *argmax* operation as follows:

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

IV. Related Work

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 [11] 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.

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

Gauci and Stanley [17] used the HyperNEAT system to evolve *Artificial Neural Networks* (ANNs) that act as search guides for the Cake American Checkers engine, resulting in an improved player. Our own work on evolving heuristic evaluation functions for Reversi and other games [7] has led us to explore the possibility of evolving search in Reversi based on the traditional *alpha-beta* search algorithm by limiting the breadth of the search tree [5]. An encouraging side effect of these results is that fast players evolved with highly restricted search-tree branching factors can be improved by removing restrictions offline (i.e., the results scale as search-tree branching factor is increased).

Though we have not found any attempt in the literature to evolve a method to bias the playouts in MCTS, it is quite common to use domain knowledge in order to do so. In designing MoGo, Gelly et al. [18] use 3×3 patterns to guide to playouts. The MoHex Hex program also uses partial board patterns to decide on playout choices [3].

V. 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 MCTS. 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 the evaluation functions are used to choose between possible moves in the playouts that MCTS performs.

We chose to implement a strongly typed GP framework [26] supporting a boolean type and a floating-point type. The original setup is detailed in [5–7]. Its main points along with recent updates and novel results are detailed in this paper. To achieve good results in reasonable time our system has 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 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 pieces (or "men") the respective player has, or a difference between the two players' man counts. The mobility terminal node allows

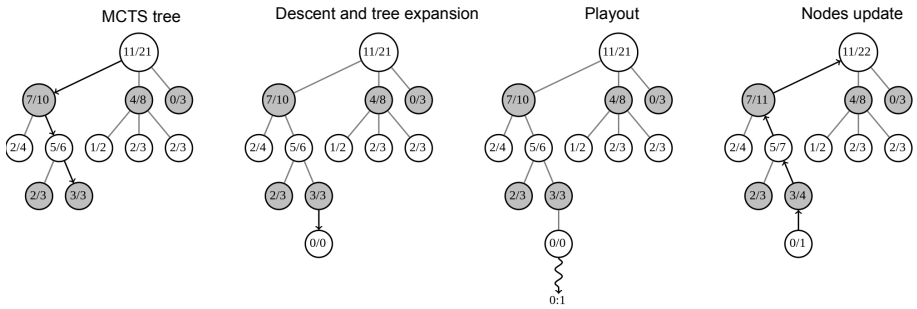

Fig. 1. Tree expansion steps of the MCTS algorithm, shown left to right. The leftmost tree is a game tree before expansion. In the second tree a new node has been added. The third tree shows a simulation from the new node. In the fourth, rightmost tree all the statistics of the relevant nodes have been updated.

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()`

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

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 the domain-specific nodes use little human knowledge about the game of Reversi. This goes against what has traditionally been done when GP is applied to board games [4, 20, 21, 31]. This is partly due to the difficulty in finding useful board attributes for evaluating game states in some games (Benbassat and Sipper [6] deals with a game that is a perfect example of this)—but there is another, more fundamental, reason. Not introducing game-specific expert knowledge into the domain-specific nodes means the GP algorithm defined is itself not game specific, and thus more flexible.

C. Function Nodes

We use several domain-independent functions. 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 2 we see an example

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	$F_1 + F_2$

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.

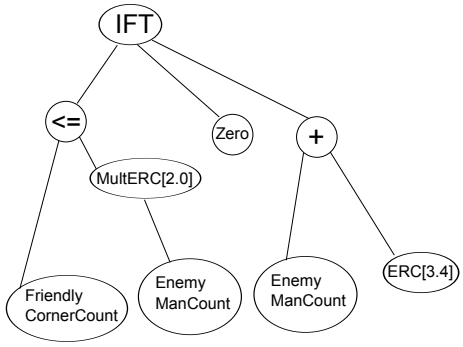

Fig. 2. Example of a subtree in our setup.

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.

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 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 [25] discusses introns in some detail. For more discussion of introns in our system see Benbassat and Sipper [6].

G. Fitness Calculation

Fitness calculation was carried out in the fashion described in Algorithm 2. Though our system supports various methods of fitness evaluation, in the evolutionary runs described in this paper fitness is decided by having evolving players face their own cohorts in the population. This method of evaluation is known as coevolution [22, 28], and is referred to below as the coevolution round.

Algorithm 2 Fitness evaluation

Every individual in the population plays *CoPlayNum* games as Black against *CoPlayNum* random opponents in the population and as a result also plays *CoPlayNum* games as White.
Assign 1 point per every game won by the individual, and 0.5 points per drawn game

In a Coevolution 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 game, 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 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 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 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 the standard MCTS this value is 0, leading to unexplored game states always being favored), 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 get better players. Based on this we can define handcrafted MCTS players to be used as yardsticks to test evolved players against.

In this paper we shall refer to MCTS players evolved using our system as EvoMCTS players. 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 asses the value of a board state.

In our system, the runtime of a single turn of an EvoMCTS player is similar to that of the standard MCTS player using the same number of playouts. This stems from the fact that EvoMCTS players spend the majority of computation time on tasks other than board-state evaluation, which the standard MCTS players also perform. Ultimately, this behavior depends upon implementation. In our case we focused on flexibility and ease of transfer between games. It may be that in highly specialized code the overhead of EvoMCTS board evaluation will have a more significant cost in computation time relative to the standard MCTS player.

Algorithm 4 Evo_Playout(Node,playoutBranchingFactor)

repeat
  VAL $\leftarrow -\infty$
  for $i \leftarrow 1$ *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 $\leftarrow EvoEval(r)$
      $ChosenMove \leftarrow r$
    end if
  end for
  $Node \leftarrow ChosenMove$
until $Node$ is a final game-state
// *GameResult()* returns information about game winner
return $GameResult(Node)$

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: 100
- UCT parameter C used: 0.7
- Runs use the option of remembering relevant parts of the game tree from previous moves
- Number of evaluated moves in playout: 4

VI. RESULTS

In order to test the quality of evolved players we need to test them against some sort of benchmark opponent—in this work we used standard MCTS players that used the UCT formula. 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). Table V shows the relative strengths of the different Reversi players. As expected,

TABLE V. RELATIVE LEVELS OF PLAY FOR DIFFERENT BENCHMARK PLAYERS IN REVERSI. 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 WOULD MEAN 6,000 WINS). A DRAW COUNTS AS HALF A WIN.

Match	First player win Ratio
100 vs 50	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 VI. 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

MCTS players improve in level of play as number of playouts increase.

In all evolutionary Reversi runs that follow we used 16 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 3–5 days.

We performed several evolutionary runs, experimenting with various parameters, in order to find a suitable parameter setup. Table VI 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.

VII. SCALABILITY OF RESULTS

Using MCTS with 100 playouts is fine if what one wants is a fast player with basic game proficiency. 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 V) so can our evolved players. Tables VII and VIII show how two top evolved players maintain their advantage when the number of playouts is scaled up.

VIII. USING EVOMCTS TO IMPROVE DODGEM PLAYERS

In order to demonstrate that our method is portable and easy to use almost "as is" on any board game, we demonstrate its use on the 5×5 variant of Dodgem. Dodgem is an abstract strategy game played on an $n \times n$ board with $n - 1$ cars for each player (Figure 3). The goal of the game is to remove one's own cars from the board via the side opposite to the player's starting side before the opponent has a chance to do so. Dodgem was first introduced as a 3x3 game by [8]. In spite of the small board size Dodgem is not a trivial game for human

TABLE VII. AN EVOLVED 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. 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. NOTE THAT EVOMCTS EVOLVED WITH ONLY 100 PLAYOUTS.

No. Playouts EvoMCTS Player	No. Playouts MCTS Player	EvoMCTS Player Benchmark Score
100	100	759.0
100	200	521.5
200	200	755.0
200	400	628.5
400	400	747.0
400	800	665.0
1000	1000	781.0
1000	2000	662.5

TABLE VIII. ANOTHER EVOLVED REVERSI PLAYER (RUN NO. 178) USING DIFFERENT PLAYOUT VALUES PLAYING AGAINST STANDARD MCTS USING EITHER THE SAME NUMBER OF PLAYOUTS OR TWICE AS MANY 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. NOTE THAT EVOMCTS EVOLVED WITH ONLY 100 PLAYOUTS.

No. Playouts EvoMCTS Player	No. Playouts MCTS Player	EvoMCTS Player Benchmark Score
100	100	727.0
100	200	530.0
200	200	773.5
200	400	605.0
400	400	763.0
400	800	647.0
1000	1000	735.0
1000	2000	637.0

players. desJardins [15] 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 $n > 3$. There has been little research or commercial interest in Dodgem, meaning expert domain knowledge is hard to come by.

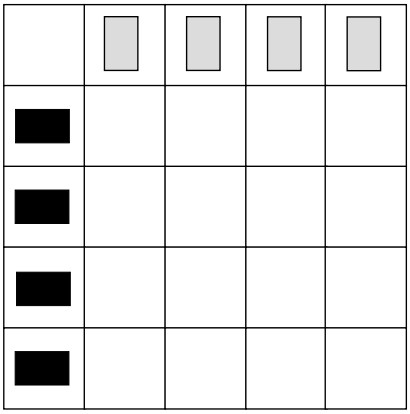

Fig. 3. 5x5 Dodgem. The board is initially set up with $n - 1$ black cars along the left edge and $n - 1$ white cars along the top edge, the top left square remaining empty. Players alternate turns, each allowed to move his vehicle forward or sideways. Cars may not move onto occupied spaces. They may leave the board, but only by a forward move. A car which 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.

TABLE IX. DODGEM-SPECIFIC TERMINAL NODES.

Node name	Type	Return value
`FriendlyPosCount()`	F	Distance-from-win measure for friendly player
`EnemyPosCount()`	F	Distance-from-win measure for enemy player
`PosCount()`	F	`FriendlyPosCount()` − `EnemyPosCount()`

TABLE X. RELATIVE LEVELS OF PLAY FOR DIFFERENT BENCHMARK PLAYERS IN 5×5 DODGEM. EACH LINE IN THE TABLE REPRESENTS A 10,000 GAME MATCH BETWEEN MCTS PLAYERS USING A DIFFERENT NUMBER OF PLAYOUTS. THE FIRST COLUMN REPRESENTS NUMBER OF PLAYOUTS USED BY BOTH PLAYERS. THE SECOND COLUMN IS THE WIN RATIO OF 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 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

We chose to evolve players for Dodgem on a 5×5 board. Since as a piece-moving game a Dodgem game can in principle go on on for an infinite number of moves, we limited playout length to 100 moves, but otherwise the MCTS algorithm remained unchanged. Table IX shows three terminal nodes we used for Dodgem instead of the Reversi-specific corner count nodes. The Dodgem-specific nodes essentially return a distance-from-win measure 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. Tables X and XI show relative levels of play of benchmark players and level of play of EvoMCTS players, respectively.

As Table XI shows, 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.

Finally, in Table XII we see that the Dodgem results are also scalable in a very convincing manner.

In all evolutionary Dodgem runs above we used a 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.

TABLE XI. 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 XII. EVOMCTS DODGEM PLAYERS USING DIFFERENT PLAYOUT VALUES PLAYING AGAINST STANDARD MCTS EITHER THE SAME NUMBER OF PLAYOUTS OR TWICE AS MANY PLAYOUTS. THE FIRST COLUMN REPRESENTS THE NUMBER OF PLAYOUTS USED BY THE EVOMCTS PLAYER FROM RUN NUMBER 181 (EVOMCTS ORIGINALLY EVOLVED WITH 100 PLAYOUTS). THE SECOND COLUMN REPRESENTS THE NUMBER OF PLAYOUTS USED BY THE MCTS BENCHMARK PLAYER

No. Playouts EvoMCTS Player	No. Playouts MCTS Player	EvoMCTS Player Benchmark Score
100	100	920.0
100	200	822.0
200	200	905.0
200	400	807.0
400	400	879.0
400	800	777.0
2000	2000	756.0
2000	4000	649.5

IX. CONCLUDING REMARKS

As the results show, using GP to evolve heuristic board evaluation functions for playouts has proved useful in improving MCTS players in two very different board games. 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 EvoMCTS players offline by increasing the number of playouts employed by the MCTS algorithm.

In addition to being game nonspecific, our method 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.

This work opens several avenues for future research. Firstly, the EvoMCTS approach presented here can be applied to high branching-factor games for which MCTS-based methods have proven especially effective (e.g., Go or Hex). Evolving players for these "heavier" games may require more computational resources, but we believe this goal is within reach even with current available hardware. Another possible avenue would be to expand EvoMCTS and have it evolve algorithm behavior within the search tree itself. This may help in improving on standard methods like UCT in domains where a fine-tuned domain-specific implementation does not exist.

ACKNOWLEDGMENT

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 world-championship-level othello program," *Artificial Intelligence*, vol. 19, no. 3, pp. 279 – 320, 1982.

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

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

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

[5] A. Benbassat and M. Sipper, "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.

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

[7] ——, "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.

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

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

[10] M. Buro, "Improving heuristic mini-max search by supervised learning," *Artificial Intelligence*, vol. 134, no. 12, pp. 85 – 99, 2002.

[11] T. Cazenave, "Evolving monte-carlo tree search algorithms," *Dept. Inf., Univ. Paris*, vol. 8, 2007.

[12] 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.

[13] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," *Computers and Games*, pp. 72–83, 2007.

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

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

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

[17] 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.

[18] S. Gelly, Y. Wang, R. Munos, O. Teytaud *et al.*, "Modification of uct with patterns in monte-carlo go," INRIA, Tech. Rep. 6062, 2006.

[19] 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.

[20] 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.

[21] ——, "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.

[22] W. D. Hillis, "Co-evolving parasites improve simulated evolution as an optimization procedure," *Physica D: Nonlinear Phenomena*, vol. 42, no. 1, pp. 228–234, 1990.

[23] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," *Machine Learning: ECML 2006*, pp. 282–293, 2006.

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

[25] 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.

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

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

[28] 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.

[29] 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.

[30] 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.

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

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