Chapter 1

MORE OR LESS? TWO APPROACHES TO EVOLVING GAME-PLAYING STRATEGIES*

Amit Benbassat, Achiya Elyasaf, and Moshe Sipper
Department of Computer Science, Ben-Gurion University, Beer-Sheva 84105, Israel

Abstract We present two opposing approaches to the evolution of game strategies, one wherein a minimal amount of domain expertise is injected into the process, the other infusing the evolutionary setup with expertise in the form of domain heuristics. We show that the first approach works well for several popular board games, while the second produces top-notch solvers for the hard game of FreeCell.

Keywords: $\alpha\beta$ search, Checkers, Dodgem, FreeCell, Genetic Algorithms, Genetic Programing, Hyper Heuristic, Reversi.

1. Introduction

The application of computational intelligence techniques within the vast domain of games has been increasing at a breathtaking speed, with evolutionary computation being one of the major approaches used. Over the past several years our research group has produced a plethora of results in numerous games of different natures (recently expounded in the book by (Sipper, 2011)).

Our goal herein is to present two opposing approaches to the evolution of game strategies. In the first, minimalist approach we inject the evolutionary setup with as little domain knowledge as possible, while aiming to prevail in a broad spectrum of games. We show that this approach

*Genetic Programming Theory and Practice X (GPTP 2012). This research was supported by the Israel Science Foundation (grant no. 123/11). Achiya Elyasaf is partially supported by the Lynn and William Frankel Center for Computer Sciences.

indeed works well for several board games, including Checkers, Reversi, and Dodgem. The second, "maximalist" approach sets out by injecting as much human knowledge as possible into the setup. This approach produces top-notch FreeCell puzzle solvers, able to beat human players to a pulp.

2. Less

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. 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 (Schaeffer et al., 2007). 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 (Schaeffer et al., 2007).

In this section our goal is entirely different (Sipper, 2011; Benbassat and Sipper, 2012). 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. Currently our system can be applied to zero-sum, deterministic, full-knowledge board games played on a rectangular board. 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 variants of multiple games*. Our aim is to show that we can improve the play level through evolution, from total incompetency to competent play, even with little or no prior expert knowledge of the game domain.

The Games

The games we explore in this work are Lose Checkers, Reversi, and Dodgem. They are all zero-sum, deterministic, full-knowledge, two-player board games played on rectangular boards games.

Checkers. Many variants of the game of Checkers exist, several of them played by a great number of people (including tournament play).

A somewhat less-popular variant of Checkers is Lose Checkers. The basic rules are the same as American Checkers (though the existence of different organizations may cause some difference in the peripheral rules). The objective, however, is quite different: A losing position for a player in American Checkers is a winning position for that player in Lose Checkers and vice versa (i.e., one wins by losing all pieces or remaining with no legal move). (Hlynka and Schaeffer, 2006) observed that, unlike the case of American Checkers, Lose Checkers lacks an intuitive state evaluation function. Surprisingly (and regrettably) the inverse of the standard, piece differential-based evaluation function is woefully ineffective. In some cases Lose Checkers computer players rely solely on optimized deep search and an endgame state database, having the evaluation function return a random value for states not in the database.

Reversi. Reversi, also known as Othello, is a popular game with a rich research history (Rosenbloom, 1982; Lee and Mahajan, 1990; Moriarty and Miikkulainen, 1995). Though the most popular Reversi variant is a board game played on an 8x8 board, it differs widely from the Checkers variants in that it is a piece-placing game rather than a piece-moving game. In Reversi the number of pieces on the board increases during play, rather than decreasing as it does in Checkers. The number of moves (not counting the rare pass moves) in Reversi is limited by the board's size, making it a short game. The 10x10 variant of Reversi is also quite popular. International tournaments are held for both variants.

Dodgem. Dodgem is an abstract strategy game played on an $n \times n$ board with $n-1$ cars for each player. The board is initially set up with $n-1$ blue cars along the left edge and $n-1$ red cars along the bottom edge, the bottom 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.

Dodgem was first introduced as a 3x3 game by (Berlekamp et al., 1982). In spite of the small board size Dodgem is not a trivial game for human players. (des Jardins, 1996) 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. (des Jardins, 1996) also postulated that Dodgem is a draw game for any board size $n > 3$.

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

Node name	Return type	Return value
`ERC()`	F	Ephemeral Random Constant in the range $[-5, 5)$
`False()`	B	Boolean $false$ value
`One()`	F	1
`True()`	B	Boolean $true$ value
`Zero()`	F	0

Handcrafted Players

In order to be able to test the quality of evolved players we first created some hand-written players. We made a point of making our players' strategy contain a random element so as to render the development of a specialized strategy against them specifically more difficult and to allow for their use as benchmark opponents. The handcrafted players essentially use the $\alpha\beta$ algorithm up to a certain depth, at which point an evaluation function is applied whenever the node reached is not a terminal one (i.e., win, loss, or draw).

Evolutionary Setup

The individuals in the population act as board-evaluation functions, to be combined with a standard game-search algorithm, in our case $\alpha\beta$. The value they return for a given board state is seen as an indication of how good that board state is for the player whose turn it is to play. The evolutionary algorithm was written in Java. We chose to implement a strongly typed GP framework supporting a boolean type and a floating-point type. Support for a multi-tree interface was also implemented. On top of the basic Koza-style crossover and mutation operators, another form of crossover was implemented—which we designated "one-way crossover"—as well as a local mutation operator. The original setup is detailed in (Benbassat and Sipper, 2010). 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.

We implemented several basic domain-independent terminal nodes were, presented in Table 1-1. The game-oriented terminal nodes are listed in several tables: Table 1-2 shows nodes describing characteristics that have to do with the board in its entirety, and Table 1-3 shows nodes describing characteristics of a certain square on the board. We originally created this setup for Lose Checkers (see (Benbassat and Sipper, 2010)), but many of the domain terminals we used for Checkers variants also proved useful for the other games.

Table 1-2. 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 plies available to the player

Table 1-3. Game-oriented 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 by man

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. In general, one could say that all the domain-specific nodes 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 (Azaria and Sipper, 2005; Hauptman and Sipper, 2005; Hauptman and Sipper, 2007). This is partly due to the difficulty in finding useful board attributes for evaluating game states in Lose Checkers—but there is another, more fundamental, reason. Not introducing game-specific knowledge into the domain-specific nodes means the GP algorithm defined is itself not game specific, and thus more flexible (it is worth noting that mobility is a universal principle in playing board games, and therefore the mobility terminal can be seen as not game-specific).

Game-specific terminal nodes are shown in Tables 1-4 through 1-6. We tried to keep these to a minimum, in line with our minimalistic approach.

We defined several basic domain-independent functions, presented in Table 1-7, and no domain-specific functions.

The functions implemented include logic functions, basic arithmetic functions, one relational function, and one conditional statement. The conditional expression rendered natural control flow possible and allowed us to compare values and return a value accordingly.

Table 1-4. Checkers-specific terminal nodes that deal with board characteristics.

Node name	Type	Return value
`EnemyKingCount()`	F	The enemy's king count
`EnemyPieceCount()`	F	The enemy's piece count
`FriendlyKingCount()`	F	The player's king count
`FriendlyPieceCount()`	F	The player's piece count
`KingCount()`	F	FriendlyKingCount() – EnemyKingCount()
`PieceCount()`	F	FriendlyPieceCount() – EnemyPieceCount()
`IsKingPiece(X,Y)`	B	True iff square occupied by king

Table 1-5. Reversi-specific terminal nodes that deal with board characteristics.

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 1-6. Dodgem-specific terminal nodes that deal with board characteristics.

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()

Table 1-7. 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
`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	$F_1 + F_2$

We implemented both one-way and two-way crossover as well as mutation. In addition we used explicitly defined introns (Sipper, 2011; Benbassat and Sipper, 2010). After initializing the fitness of all individuals in the population to 0, fitness calculation was carried out by having evolving players face two types of opponents: external "guides" and their own cohorts in the population (coevolution). Several games were then held, with the final fitness value depending on the player's performance (for the precise fitness definition see (Sipper, 2011; Benbassat and Sipper, 2010)).

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 (Sipper, 2011; Benbassat and Sipper, 2010).

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

Results

We divided the experiments into two sets, the first using no search, the second with search and mobility incorporated into the evolutionary algorithm. Below provide a summary of our results, with a full account given in (Benbassat and Sipper, 2012) (see also (Sipper, 2011)).

Checkers. As there is no known simple board-evaluation function for Lose Checkers we used a random evaluation for non-final board states (i.e., states that were not win, loss, or draw). Our players, using a search depth of 4, were able to outperform the strong $\alpha\beta 5$ player. Note that the player does not specialize in playing the benchmark opponent but rather faces an ever-changing array of opponents among its cohorts in the population.

Reversi. For Reversi we used the material evaluation (piece counting) to evaluate board states. We had handcrafted players randomly alternate between material evaluation functions. Our best runs to date for 8x8 Reversi are presented in full in (Benbassat and Sipper, 2012) (see also (Sipper, 2011)). Our evolved players were able to outperform the $\alpha\beta 5$ and $\alpha\beta 7$ players using a search depth of 4. Our best runs to date for 10x10 Reversi are presented in (Benbassat and Sipper, 2012).

Dodgem. For Dodgem we used the appropriate version of material evaluation to evaluate board states. Our evaluation function took into account not only the number of pieces on the board but also their distance from the edge of the board (since the goal in Dodgem is to move one's pieces off the board). As with the previous games, we had handcrafted players randomly alternate between material evaluation functions. 5x5 Dodgem afforded us the chance for deeper search than previous games. Our evolved 5x5 Dodgem players using a search depth of 5 held their own against a handcrafted opponent using a search depth of 7, with evolved 6x6 Dodgem also displaying an excellent level of play.

3. More

A well-known, highly popular example within the domain of discrete puzzles is the card game of FreeCell. Starting with all cards randomly divided into k piles (called *cascades*), the objective of the game is to move all cards onto four different piles (called *foundations*)—one per suit—arranged upwards from the ace to the king. Additionally, there are initially empty cells (called *FreeCells*), whose purpose is to aid with moving the cards. Only exposed cards can be moved, either from FreeCells or cascades. Legal move destinations include: a home (foundation) cell, if all previous (i.e., lower) cards are already there; empty FreeCells; and, on top of a next-highest card of opposite color in a cascade. FreeCell was proven by Helmert (Helmert, 2003) to be NP-complete. Computational complexity aside, many (oft-frustrated) human players (including the authors) will readily attest to the game's hardness. The attainment of a competent machine player would undoubtedly be considered a human-competitive result.

FreeCell remained relatively obscure until it was included in the Windows 95 operating system (and in all subsequent versions), along with 32,000 problems—known as *Microsoft 32K*—all solvable but one (this latter, game #11982, was proven to be unsolvable). Numerous solvers developed specifically for FreeCell are available via the web, the best of which was that of Heineman (Heineman, 2009). Although it fails to

solve 4% of Microsoft 32K, Heineman's solver significantly outperforms all other solvers in terms of both space and time.

Heineman's Staged Deepening (HSD) algorithm may be viewed as two-layered IDA* with periodic memory cleanup. The two layers operate in an interleaved fashion: 1) At each iteration, a local DFS is performed from the head of the open list up to depth k, with no heuristic evaluations, using a transposition table—storing visited nodes—to avoid loops; 2) Only nodes at *precisely* depth k are stored in the open list,[1] which is sorted according to the nodes' heuristic values. In addition to these two interleaved layers, whenever the transposition table reaches a predetermined size, it is emptied entirely, and only the open list remains in memory (Sipper, 2011).

When we ran the HSD algorithm it solved 96% of Microsoft 32K, as reported by Heineman.

At this point we were at the limit of the current state-of-the-art for FreeCell, and we turned to evolution to attain better results. This time we took a "maximalist" approach, the idea being to inject much human expertise into the evolutionary setup. We first needed to develop additional heuristics for this domain.

Freecell Heuristics and Advisors

In this section we describe the heuristics we used, all of which estimate the distance to the goal from a given game configuration:

`Heineman's Staged Deepening Heuristic (HSDH)`: This is the heuristic used by the HSD solver. For each foundation pile (recall that foundation piles are constructed in ascending order), locate within the cascade piles the next card that should be placed there, and count the cards found on top of it. The returned value is the sum of this count for all foundations. This number is multiplied by 2 if there are no free FreeCells or empty cascade piles (reflecting the fact that freeing the next card is harder in this case).

`NumWellPlaced`: Count the number of *well-placed* cards in cascade piles. A pile of cards is well placed if *all* its cards are in descending order and alternating colors.

`NumCardsNotAtFoundations`: Count the number of cards that are not at the foundation piles.

`FreeCells`: Count the number of free FreeCells and cascades.

`DifferenceFromTop`: The average value of the top cards in cascades, minus the average value of the top cards in foundation piles.

[1] Note that since we are using DFS and not BFS we do not find all such states.

`LowestFoundationCard`: The highest possible card value (typically the king) minus the lowest card value in foundation piles.

`HighestFoundationCard`: The highest card value in foundation piles.

`DifferenceFoundation`: The highest card value in the foundation piles minus the lowest one.

`SumOfBottomCards`: Take the highest possible sum of cards in the bottom of cascades (e.g., for 8 cascades, this is $4 * 13 + 4 * 12 = 100$), and subtract the sum of values of cards actually located there.

Apart from heuristics, which estimate the distance to the goal, we also defined *advisors* (or auxiliary functions), incorporating domain features, i.e., functions that do not provide an estimate of the distance to the goal but which are nonetheless beneficial in a GP setting.

`PhaseByX`: This is a set of functions that includes a "mirror" function for each of the heuristics define above. Each function's name (and purpose) is derived by replacing `X` in `PhaseByX` with the original heuristic's name, e.g., `LowestFoundationCard` produces `PhaseByLowestFoundationCard`. `PhaseByX` incorporates the notion of applying different strategies (embodied as heuristics) at different *phases* of the game, with a phase defined by $g/(g + h)$, where g is the number of moves made so far, and h is the value of the original heuristic.

For example, suppose 10 moves have been made ($g = 10$), and the value returned by `LowestFoundationCard` is 5. The `PhaseByLowestFoundationCard` heuristic will return $10/(10 + 5)$ or $2/3$ in this case, a value that represents the belief that by using this heuristic the configuration being examined is at approximately 2/3 of the way from the initial state to the goal.

`DifficultyLevel`: This function returns the location of the current problem being solved in an ordered problem set (sorted by difficulty), and thus yields an estimate of how difficult it is. The difficulty of a problem is defined by the amount of nodes HSD needed to solve that problem.

`IsMoveToCascade` is a Boolean function that examines the destination of the last move and returns true if it was a cascade.

(Elyasaf et al., 2012; Sipper, 2011) provide a full list of the auxiliary functions, including the above functions and a number of additional ones.

Experiments with these heuristics demonstrated that each one separately (except for HSDH) was not good enough to guide search for this difficult problem. Thus we turned to evolution.

Evolving Heuristics for FreeCell

Combining several heuristics to get a more accurate one is considered one of the most difficult problems in contemporary heuristics research (Samadi et al., 2008; Burke et al., 2010). It is difficult mainly since it entails traversing an extremely large search space of possible numeric combinations, logic conditions, and game configurations. To tackle this problem we turn to evolution.

In order to properly solve these three sub-problems, we designed a large set of experiments using three different evolutionary methods, all involving hyper-heuristics: a standard GA, standard (Koza-style) GP, and policy-based GP. Each type of hyper-heuristic was paired with three different learning settings: Rosin-style coevolution, Hillis-style coevolution, and a novel method which we termed gradual difficulty. We describe herein only the winning strategy—policy-based GP with Hillis-style coevolution—with the other approaches described in (Elyasaf et al., 2012) (see also (Sipper, 2011)).

Policy-based GP combines estimates and application conditions, using ordered sets of control rules, or *policies*. Policies have been evolved successfully with GP to solve search problems—albeit simpler ones (e.g., (Elyasaf et al., 2012; Sipper, 2011; Hauptman et al., 2009; Hauptman et al., 2010; Aler et al., 2002)).

The structure of a policy is:

$RULE_1$: IF $Condition_1$ THEN $Value_1$
.
.
.
$RULE_N$: IF $Condition_N$ THEN $Value_N$
$DEFAULT$: $Value_{N+1}$

where $Condition_i$ and $Value_i$ represent conditions and estimates, respectively.

Policies are used by the search algorithm in the following manner: The rules are ordered such that we apply the first rule that "fires" (meaning its condition is true for the current state being evaluated), returning its $Value$ part. If no rule fires, the value is taken from the last (default) rule: $Value_{N+1}$. Thus individuals, while in the form of policies, are still heuristics—the value returned by the activated rule is an arithmetic combination of heuristic values, and is thus a heuristic value itself. This accords with our requirements: rule ordering and conditions

control when we apply a heuristic combination, and values provide the combinations themselves.

Thus, with N being the number of rules used, each individual in the evolving population contains N *Condition* GP trees and $N + 1$ $Value$ sets of weights used for computing linear combinations of heuristic values. After experimenting with several sizes of policies, we settled on $N = 5$, providing us with enough rules per individual, while avoiding cumbersome individuals with too many rules. The depth limit used for the *Condition* trees was empirically set to 5.

For *Condition* GP trees, the function set included the functions $\{AND, OR, \leq, \geq\}$, and the terminal set included all heuristics and auxiliary functions define above. The sets of weights appearing in $Value$s all lie within the range $[0, 1]$, and correspond to the above heuristics. All the heuristic values are normalized to within the range $[0, 1]$ as described above.

The crossover and mutation operators were performed as follows: First, one or two individuals were selected (depending on the genetic operator). Second, we randomly selected the rule (or rules) within the individual(s). This we did with uniform distribution, except that the most oft-used rule (we measured the number of times each rule fired) had a 50% chance of being selected. Third, we chose with uniform probability whether to apply the operator to either of the following: the entire rule, the condition part, or the value part.

We thus have 6 sub-operators, 3 for crossover—*RuleCrossover*, *ConditionCrossover*, and *ValueCrossover*—and 3 for mutation—*RuleMutation*, *ConditionMutation*, and *ValueMutation*. One of the major advantages of policies is that they facilitate the use of such diverse genetic operators.

For both GP-trees and policies, crossover was only performed between nodes of the same type (using Strongly Typed Genetic Programming).

The Microsoft 32K suite contains a random assortment of deals of varying difficulty levels. In each of our experiments 1,000 of these deals were randomly selected for the training set and the remaining 31K were used as the test set.

An individual's fitness score was obtained by running the HSD algorithm on deals taken from the training set, with the individual used as the heuristic function. Fitness equaled the average search-node reduction ratio. This ratio was obtained by comparing the reduction in number of search nodes—averaged over solved deals—with the average number of nodes when searching with the original HSD heuristic (HSDH). For example, if the average reduction in search was 70% compared with HSDH (i.e., 70% fewer nodes expanded on average), the fitness score was set

to 0.7. If a given deal was not solved within 2 minutes (a time limit we set empirically), we assigned a fitness score of 0 to that deal (for a fuller explanation see (Elyasaf et al., 2011; Sipper, 2011)).

We applied Hillis-style coevolution (Hillis, 1992), wherein the first population comprises the solvers, as described above. In the second population an individual represents a *set* of FreeCell deals. Thus a "hard"-to-solve individual in this latter, problem population contains several deals of varying difficulty levels. This multi-deal individual made life harder for the evolving solvers: They had to maintain a consistent level of play over several deals. With single-deal individuals, which we used in Rosin-style coevolution, either the solvers did not improve if the deal population evolved every generation (i.e., too fast), or the solvers became adept at solving certain deals and failed on others if the deal population evolved more slowly (i.e., every k generations, for a given $k > 1$).

The genome and genetic operators of the solver population were identical to those defined above. The genome of an individual in the deals population contained 6 FreeCell deals, represented as integer-valued indexes from the training set $\{v_1, v_2, \ldots, v_{1000}\}$, where v_i is a random index in the range $[1, 32000]$. We applied GP-style evolution in the sense that first an operator (reproduction, crossover, or mutation) was selected with a given probability, and then one or two individuals were selected in accordance with the operator chosen. We used standard fitness-proportionate selection and single-point crossover. Mutation was performed in a manner analogous to bitwise mutation by replacing with independent probability 0.1 an (integer-valued) index with a randomly chosen deal (index) from the training set, i.e., $\{v_1, v_2, \ldots, v_{1000}\}$. Since the solvers needed more time to adapt to deals, we evolved the deal population every 5 solver generations (this slower evolutionary rate was set empirically).

Fitness was assigned to a solver by picking 2 individuals in the deal population and attempting to solve all 12 deals they represented. The fitness value was an average of all 12 deals. Whenever a solver "ran" a deal individual's 6 deals its performance was recorded in order to derive the fitness of the deal population. A deal individual's fitness was defined as the average number of nodes needed to solve the 6 deals, averaged over the solvers that "ran" this individual, and divided by the average number of nodes when searching with the original HSD heuristic. If a particular deal was not solved by any of the solvers—a value of 1000M nodes was assigned to it.

Not only did this method surpass all previous learning-based methods, but it also outperformed HSD by a wide margin, solving all but 112

deals of Microsoft 32K when using policy individuals, and doing so using significantly less time and space requirements. Additionally, the solutions found were shorter and hence better (Elyasaf et al., 2012; Sipper, 2011).

Results

Perhaps the most impressive feat is the following: Policy-FreeCell beat *all* human players from a major FreeCell website (`www.freecell.net`), ranking number one. Note that GA-FreeCell, our HUMIE-winning, GA-based player of last year (Elyasaf et al., 2011) came in at number 11. Full results are provided in (Elyasaf et al., 2012) (see also (Sipper, 2011)).

4. Concluding Remarks

We have seen two approaches to evolutionary game design, the first using a flexible generic system and attempting to evolve players while injecting as little domain knowledge as possible (and thus allowing for multiple games to be tackled by the same system with few adjustments), the second doing just the opposite. The minimalist approach seems to work well when a group of games exhibiting similar features can be identified. Note that even then we needed to make some domain-specific adaptations in order to attain high performance. We are currently expanding this approach to widen its spectrum of applicability, tackling the card game Hearts, and the non-deterministic game Backgammon.

We see here the age-old trade-off between flexibility and generality on the one hand and specialization on the other. The specialized system has an inherent advantage in that all its parts and parameters are optimized for the goal of succeeding in the one domain for which it was built. Conversely, the generic system allows the quick application and optimization to a new domain, though its best results may fall short of a highly specialized system. To try and close this gap we are constantly adding parts and features to our generic system, while endeavoring to leave an opening for the application of expert domain knowledge when it is available (meaning that even with the generic system knowing your domain well is still a good thing).

The maximalist approach is well suited for single-agent search problems. The domain-specific heuristics used as building blocks for the evolutionary process are intuitive and straightforward to implement and compute. Yet, the evolved solver is the top solver for the game of FreeCell. It should be noted that complex heuristics and memory-consuming heuristics (e.g., landmarks and pattern databases) can be used as build-

ing blocks as well. Such solvers might outperform the simpler ones at the expense of increased code complexity.

References

Aler, R., Borrajo, D., and Isasi, P. (2002). Using genetic programming to learn and improve knowledge. *Artificial Intelligence*, 141(1–2):29–56.

Azaria, Yaniv and Sipper, Moshe (2005). GP-gammon: Genetically programming backgammon players. *Genetic Programming and Evolvable Machines*, 6(3):283–300. Published online: 12 August 2005.

Benbassat, A. and Sipper, M. (2012). Evolving competent board game players for mutiple games with little domain knowledge. (in preparation).

Benbassat, Amit and Sipper, Moshe (2010). Evolving lose-checkers players using genetic programming. In *IEEE Conference on Computational Intelligence and Game*, pages 30–37, IT University of Copenhagen, Denmark.

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

Burke, Edmund K., Hyde, Matthew, Kendall, Graham, Ochoa, Gabriela, Ozcan, Ender, and Woodward, John R. (2010). A classification of hyper-heuristic approaches. In Gendreau, M. and Potvin, J-Y., editors, *Handbook of Meta-Heuristics 2nd Edition*, pages 449–468. Springer.

des Jardins, D. (1996). "dodgem" . . . any info?

Elyasaf, A., Hauptman, A., and Sipper, M. (2012). Evolutionary design of freecell solvers. (submitted).

Elyasaf, Achiya, Hauptman, Ami, and Sipper, Moshe (2011). GA-FreeCell: Evolving Solvers for the Game of FreeCell. In Krasnogor, N. et al., editors, *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1931–1938, Dublin, Ireland. ACM.

Hauptman, A., Elyasaf, A., and Sipper, M. (2010). Evolving hyper heuristic-based solvers for Rush Hour and FreeCell. In *SoCS '10: Proceedings of the 3rd Annual Symposium on Combinatorial Search (SoCS 2010)*, pages 149–150.

Hauptman, Ami, Elyasaf, Achiya, Sipper, Moshe, and Karmon, Assaf (2009). GP-rush: using genetic programming to evolve solvers for the rush hour puzzle. In Raidl, Guenther et al., editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 955–962, Montreal. ACM.

Hauptman, Ami and Sipper, Moshe (2005). GP-endchess: Using genetic programming to evolve chess endgame players. In Keijzer, Maarten,

Tettamanzi, Andrea, Collet, Pierre, van Hemert, Jano I., and Tomassini, Marco, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 120–131, Lausanne, Switzerland. Springer.

Hauptman, Ami and Sipper, Moshe (2007). Evolution of an efficient search algorithm for the mate-in-N problem in chess. In Ebner, Marc, O'Neill, Michael, Ekárt, Anikó, Vanneschi, Leonardo, and Esparcia-Alcázar, Anna Isabel, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 78–89, Valencia, Spain. Springer.

Heineman, G. T. (2009). Algorithm to solve FreeCell solitaire games. broadcast.oreilly.com/2009/01/january-column-graph-algorithm.html. Blog column associated with the book "Algorithms in a Nutshell book," by G. T. Heineman, G. Pollice, and S. Selkow, O'Reilly Media, 2008.

Helmert, M. (2003). Complexity results for standard benchmark domains in planning. *Artificial Intelligence*, 143(2):219–262.

Hillis, W. Daniel (1992). Co-evolving parasites improve simulated evolution as an optimization procedure. In Langton, Christopher G., Taylor, Charles E., Farmer, J. Doyne, and Rasmussen, Steen, editors, *Artificial Life II*, volume X of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 313–324. Addison-Wesley, Santa Fe Institute, New Mexico, USA.

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

Lee, Kai-Fu and Mahajan, Sanjoy (1990). The development of a world class othello program. *Artificial Intelligence*, 43(1):21–36.

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

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

Samadi, M., Felner, A., and Schaeffer, J. (2008). Learning from multiple heuristics. In Fox, Dieter and Gomes, Carla P., editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, pages 357–362. AAAI Press.

Schaeffer, J. et al. (2007). Checkers is solved. *Science*, 317(5844):1518–1522.

Sipper, Moshe (2011). *Evolved to Win*. Lulu. available at http://www.lulu.com/.