Chapter 1

LET THE GAMES EVOLVE![*]

Moshe Sipper
Department of Computer Science, Ben-Gurion University, Beer-Sheva 84105, Israel

Abstract I survey my group's results over the past six years within the game area, demonstrating continual success in evolving winning strategies for challenging games and puzzles, including: chess, backgammon, Robocode, lose checkers, simulated car racing, Rush Hour, and FreeCell.

Keywords: backgammon, chess, FreeCell, lose checkers, policy, RARS, Robocode, Rush Hour.

It was on a bitterly cold night and frosty morning, towards the end of the winter of '97, that I was awakened by a tugging at my shoulder. It was Holmes. The candle in his hand shone upon his eager, stooping face, and told me at a glance that something was amiss.

"Come, Watson, come!" he cried. "The game is afoot. Not a word! Into your clothes and come!"

Arthur Conan Doyle, "The Adventure of the Abbey Grange"

1. GP is for Genetic Programming, GP is for Game Playing

Ever since the dawn of artificial intelligence (AI) in the 1950s games have been part and parcel of this lively field. In 1957, a year after the Dartmouth Conference that marked the official birth of AI, Alex Bernstein designed a program for the IBM 704 that played two amateur games of chess. In 1958, Allen Newell, J. C. Shaw, and Herbert Simon introduced a more sophisticated chess program (beaten in thirty-five moves by a ten-year-old beginner in its last official game played in 1960). Arthur L. Samuel of IBM spent much of the fifties working on game-playing AI programs, and by 1961 he had a checkers

[*]Draft version. Final version to appear in Genetic Programming Theory and Practice IX (GPTP 2011).

program that could play rather decently. In 1961 and 1963 Donald Michie described a simple trial-and-error learning system for learning how to play Tic-Tac-Toe (or Noughts and Crosses) called MENACE (for Matchbox Educable Noughts and Crosses Engine).

Why do games attract such interest? "There are two principal reasons to continue to do research on games," wrote (Epstein, 1999). "First, human fascination with game playing is long-standing and pervasive. Anthropologists have catalogued popular games in almost every culture... Games intrigue us because they address important cognitive functions... The second reason to continue game-playing research is that some difficult games remain to be won, games that people play very well but computers do not. These games clarify what our current approach lacks. They set challenges for us to meet, and they promise ample rewards."

Studying games may thus advance our knowledge in both cognition and artificial intelligence, and, last but not least, games possess a competitive angle which coincides with our human nature, thus motivating both researcher and student alike.

During the past few years there has been an ever-increasing interest in the application of computational intelligence techniques in general, and evolutionary algorithms in particular, within the vast domain of games. I happened to stumble across this trend early on and decided to climb aboard the gamesome boat while it was still not too far from the harbor.

The year 2005 saw the first *IEEE Symposium on Computational Intelligence and Games*, which went on to become an annually organized event. The symposia's success and popularity led to their promotion from symposium to conference in 2010, and also spawned the journal *IEEE Transactions on Computational Intelligence and AI in Games* in 2009. In 2008, a journal showcase of evolutionary computation in games seemed to be the right thing to do—so we did it (Sipper and Giacobini, 2008).

In this paper I survey my group's results over the past six years within the game area, demonstrating continual success in evolutionarily tackling hard games and puzzles. The next section presents our earliest results in three games: chess, backgammon, and Robocode. Section 3 delineates the application of GP to evolving lose checkers strategies. In Section 4 we apply GP to a single-player game—a puzzle—known as Rush Hour. Sections 5 and 6 summarize results obtained for FreeCell and RARS (Robot Auto Racing Simulator), respectively. Finally, we end with concluding remarks in Section 7.[1]

[1]Caveat lector: Given the limited space and my intended coverage some sections are less detailed than others. Full details can, of course, be found in the references provided.

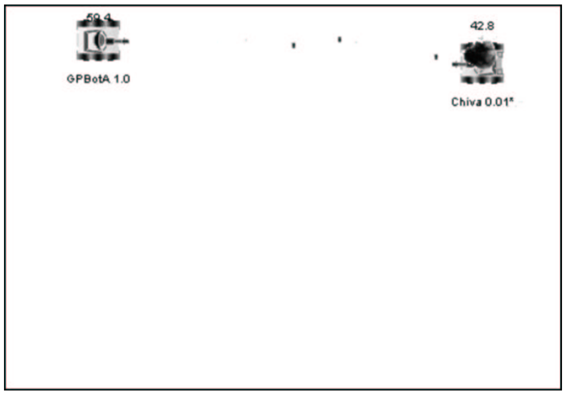

Figure 1-1 Robocode: GP bot (left) fights an enemy (right) to the death (and lives to tell the tale).

2. Genesis: A Wholly Trinity

Our ludic pursuit began in 2005 with the study of three games—chess, backgammon, and Robocode (Sipper et al., 2007):

1 **Backgammon.** The goal was to evolve a full-fledged player for the non-doubling-cube version of the game (Azaria and Sipper, 2005a; Azaria and Sipper, 2005b).

2 **Chess** (endgames). The goal was to evolve a player able to play endgames (Hauptman and Sipper, 2005a; Hauptman and Sipper, 2005b). While endgames typically contain but a few pieces, the problem of evaluation is still hard, as the pieces are usually free to move all over the board, resulting in complex game trees—both deep and with high branching factors. Indeed, in the chess lore much has been said and written about endgames.

3 **Robocode.** A simulation-based game in which robotic tanks fight to destruction in a closed arena (Figure 1-1). The programmers implement their robots in the Java programming language, and can test their creations either by using a graphical environment in which battles are held, or by submitting them to a central web site where online tournaments regularly take place. Our goal here was to evolve robocode players able to rank high in the international league (Shichel et al., 2005).

A strategy for a given player in a game is a way of specifying which choice the player is to make at every point in the game from the set of allowable choices at that point, given all the information that is available to the player at that point (Koza, 1992). The problem of discovering a strategy for playing a game can be viewed as one of seeking a computer program. Depending on the game, the program might take as input the entire history of past moves or just the current state of the game. The desired program then produces the next move as output. For some games one might evolve a complete strategy that addresses every situation tackled. This proved to work well with Robocode, which is a dynamic game, with relatively few parameters, and little need for past history.

Another approach (which can probably be traced back to (Samuel, 1959)) is to couple a current-state evaluator (e.g., board evaluator) with a next-move generator. One can go on to create a minimax tree, which consists of all possible moves, countermoves, counter-countermoves, and so on; for real-life games, such a tree's size quickly becomes prohibitive. Deep Blue, the famous machine chess player, and its offspring Deeper Blue, relied mainly on brute-force methods to gain an advantage over the opponent, by traversing as deeply as possible the game tree (Kendall and Whitwell, 2001). Although these programs achieved amazing performance levels, Chomsky criticized this aspect of game-playing research as being "about as interesting as the fact that a bulldozer can lift more than some weight lifter" (Chomsky, 1993). The approach we used with backgammon and chess was to derive a very shallow, *single-level* tree, and evolve "smart" evaluation functions. Our artificial player was thus had by combining an evolved board evaluator with a (relatively simple) program that generated all next-move boards (such programs can easily be written for backgammon and chess).

Thus, we used GP to evolve either complete game strategies (Robocode) or board-evaluation functions (chess, backgammon). Our results for these three games (which also garnered two HUMIE awards, in 2005 and 2007) are summarized below.

Backgammon. We pitted our top evolved backgammon players against *Pubeval*, a free, public-domain board evaluation function written by Gerry Tesauro to serve as a common standardized benchmark. It is a linear function of the raw board information (number of checkers at each location), which represents a linear approximation to human play. The program became the *de facto* yardstick used by the growing community of backgammon-playing program developers. Our top evolved player was able to attain a win percentage of 62.4% in a tournament against Pubeval, about 10% higher than the previous top method. Moreover, several different evolved strategies were able to surpass the 60% mark, and most of them outdid all previous works.

Chess (endgames). We pitted our top evolved chess-endgame players against two very strong external opponents: 1) A program we wrote ('Master') based upon consultation with several high-ranking chess players; 2) CRAFTY—a world-class chess program, which finished second in the 2004 World Computer Speed Chess Championship. Speed chess ("blitz") involves a time-limit per move, which we imposed both on CRAFTY and on our players. Not only did we thus seek to evolve good players, but ones that played well *and fast*.

Table 1-1. Percent of wins, material advantage without mating (having a higher point count than the opponent), and draws for best GP-EndChess player in a tournament against two top competitors.

	%Wins	%Advs	%Draws
Master	6.00	2.00	68.00
CRAFTY	2.00	4.00	72.00

Results are shown in Table 1-1. As can be seen, GP-EndChess managed to hold its own, and even win, against these top players.

Deeper analysis of the strategies developed (Hauptman and Sipper, 2005a) revealed several important shortcomings, most of which stemmed from the fact that they used deep knowledge and little search, typically developing only *one* level of the search tree. Simply increasing the search depth would not have solved the problem, since the evolved programs examined each board very thoroughly, and scanning many boards would have increased time requirements prohibitively. And so we turned to evolution to find an optimal way to overcome this problem: How to add more search at the expense of less knowledgeable (and thus less time-consuming) node evaluators, while attaining better performance. In (Hauptman and Sipper, 2007b) *we evolved the search algorithm itself*, focusing on the *Mate-In-N* problem: find a key move such that even with the best possible counterplays, the opponent cannot avoid being mated in (or before) move N. Our evolved search algorithms successfully solved several instances of the Mate-In-N problem, for the hardest ones developing 47% less game-tree nodes than CRAFTY. Improvement was thus not over the basic alpha-beta algorithm, but over a world-class program using all standard enhancements.

Finally, in (Hauptman and Sipper, 2007a), we examined a strong evolved chess-endgame player, focusing on the player's emergent capabilities and tactics in the context of a chess match. Using a number of methods, we analyzed the evolved player's building blocks and their effect on play level (e.g., comparison of evolved player's moves and CRAFTY's, study of play level of single terminals, disabling of terminals in evolved player, etc'). We concluded that evolution found combinations of building blocks that were far from trivial and could not be explained through simple combination—thereby indicating the possible *emergence* of complex strategies.

Robocode. A Robocode player is written as an event-driven Java program. A main loop controls the tank activities, which can be interrupted on various occasions, called *events*. The program was limited to four lines of code, as we were aiming for the HaikuBot category, one of the divisions of the international league with a four-line code limit.

We submitted our top evolved player to the online league. At its very first tournament it came in third, later climbing to first place of 28.[2] All other 27 programs—defeated by our evolved bot—were written by humans.

3. You Lose, You Win

Many variants of the game of checkers exist, several of them played by a great number of people (including tournament play). Practically all checkers variants are two-player games that contain only two types of pieces set on an $n \times n$ board. The most well-known variant of checkers is American checkers. It offers a relatively small search space (roughly 10^{20} legal positions compared to the 10^{43}–10^{50} estimated for chess) with a relatively small branching factor. It is fairly easy to write a competent[3] computer player for American checkers using minimax search and a trivial evaluation function. The generic evaluation function for checkers is a piece differential that assigns extra value to kings on the board. This sort of player was used by (Chellapilla and Fogel, 2001) in their work on evolving checkers-playing programs.

American checkers shares its domain with another, somewhat less-popular variant of checkers, known as lose checkers. The basic rules of lose checkers are the same as American checkers (though the existence of different organizations may cause some difference in the peripheral rules). The objective, however, is quite different. A losing position for a player in American checkers is a winning position for that player in lose checkers and vice versa (i.e., one wins by losing all pieces or remaining with no legal move). (Hlynka and Schaeffer, 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 checkers 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.

To date, there has been limited research interest in lose checkers (Hlynka and Schaeffer, 2006; Smith and Sailer, 2004), leaving room for improvement.[4] The mere fact that it is difficult to hand-craft a good evaluation function for lose checkers allows for the claim that any good evaluation function is in fact human competitive. If capable human programmers resort to having their evalu-

[2] `robocode.yajags.com/20050625/haiku-1v1.html`
[3] We use "competent" to describe players that show a level of playing skill comparable to some human players (i.e., are not trivially bad) and yet do not exhibit the level of play of the strongest players (be they computer or human) available. As it is often hard to compare levels of play between different games, we find this fluid definition of "competence" to be suitable.
[4] American checkers was solved by (Schaeffer et al., 2007).

Table 1-2. Lose checkers: Basic domain-independent terminal nodes. F: floating point, B: boolean.

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

Table 1-3. Lose checkers: Domain-specific terminal nodes that deal with board characteristics.

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

Table 1-4. Lose checkers: Domain-specific terminal nodes that deal with square characteristics. They all receive two parameters—X and Y—the row and column of the square, respectively.

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

ation function return random values, then any improvement on random is worth noting.

We have recently applied GP to evolving good lose-checkers players (Benbassat and Sipper, 2010). The individuals in the population acted as board-evaluation functions, to be combined with a standard game-search algorithm such as alpha-beta. The value they returned for a given board state was seen as an indication of how good that board state was for the player whose turn it was to play.

We used strongly typed GP with two node types: Boolean and floating point. The terminal set comprised basic domain-independent nodes (Table 1-2) and domain-specific nodes. These latter are listed in two tables: Table 1-3 shows nodes describing characteristics that have to do with the board in its entirety, and Table 1-4 shows nodes describing characteristics of a certain square on the board. The function nodes are listed in Table 1-5.

Table 1-5. Lose checkers: 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
NullFuncJ(F_1,F_2)	F	F_1
Plus(F_1,F_2)	F	$F_1 + F_2$

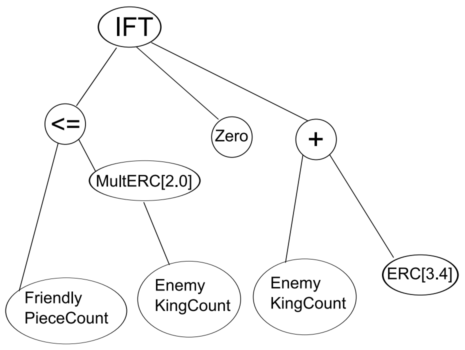

Figure 1-2 Lose checkers: Sample tree.

Figure 1-2 shows a sample tree, which returns 0 if the friendly piece count is less than or equal to double the number of enemy kings on the board, otherwise returning the number of enemy kings plus 3.4.

Aside from the standard two-point crossover we introduced a one-way crossover operator: randomly select a subtree in both participating individuals, and then insert a copy of the selected subtree from the first individual (donor) in place of the selected subtree from the second individual (receiver). The individual with higher fitness was always chosen to act as the donor. We also defined a form of local, minimally disruptive mutation, and used explicitly defined introns (Benbassat and Sipper, 2010).

As for fitness, evolving players faced two types of opponents: external "guides," and their own cohorts in the population (the latter known as coevolution). Two types of guides were implemented: A random player and an alpha-beta player. The random player chose a move at random and was used to test initial runs. The alpha-beta player searched up to a preset depth in the game tree and used an evaluation function returning a random value for game states for which there was no clear winner (in states where win or loss was evident the evaluation function returned an appropriate value). An individual's fitness value was proportional to its performance in a set of games against a mix of opponent guides and cohorts.

Our setup supported two different kinds of GP players. The first examined all legal moves and used the GP individual to assign scores to the different moves, choosing the one that scored highest. This method is essentially a minimax search of depth 1. The second kind of player mixed GP game-state evaluation with a minimax search. It used the alpha-beta search algorithm implemented for the guides, but instead of evaluating non-terminal states randomly it did so using the GP individual. This method added search power to our players but required more time.

(Benbassat and Sipper, 2010) presents a plethora of experiments we performed, the bottom line being the emergence of highly competent players. Our evolved alpha-beta players were able to beat opponent alpha-beta players that searched considerably deeper. To wit, our players were able not only to win—but to do so while expending far fewer computational resources (i.e., expansion of tree nodes).

4. Help! I'm Stuck in the Parking Lot

Single-player games, or puzzles, have received much attention from the AI community for quite some time (e.g., (Hearn, 2006; Robertson and Munro, 1978)). Quite a few NP-Complete puzzles, though, have remained relatively neglected by researchers (see (Kendall et al., 2008) for a review).

Among these difficult games we find the Rush Hour puzzle.[5] The commercial version of this popular single-player game is played on a 6x6 grid, simulating a parking lot replete with several vehicles. The goal is to find a sequence of legal vehicular moves that ultimately clears the way for the red target car, allowing it to exit the lot through a tile that marks the exit (see Figure 1-3). Vehicles—of length two or three tiles—are restricted to moving either vertically or horizontally (but not both), they cannot vault over other vehicles, and no two vehicles may occupy the same tile at once. The generalized version of the game is defined on an arbitrary grid size, though the 6x6 board is sufficiently challenging for humans (we are not aware of humans playing, let alone solving, complex boards larger than 6x6).

How does one go about solving such a puzzle through computational means? A primary problem-solving approach within the field of AI is that of heuristic search. One of the most important heuristic search algorithms is iterative-deepening A* (IDA*) (Hart et al., 1968; Korf, 1985), which is widely used to solve single-player games. IDA* and similar algorithms are strongly based on

[5]The name "Rush Hour" is a trademark of Binary Arts, Inc. The game was originally invented by Nobuyuki Yoshigahara in the late 1970s.

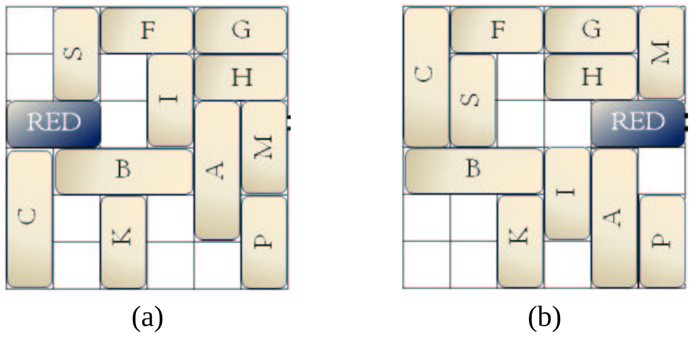

Figure 1-3. (a) A sample Rush Hour configuration. This is problem no. 9 of the problem set shipped with the standard version of the game by Binary Arts, Inc. (b) A possible goal state: the red car has reached the exit tile on the right-hand side of the grid.

the notion of approximating the distance of a given configuration (or *state*) to the problem's solution (or *goal*). Such approximations are found by means of a computationally efficient function, known as the *heuristic function*.

By applying the heuristic function to states reachable from the current ones considered it becomes possible to select more-promising alternatives earlier on in the search process, possibly reducing the amount of search effort (typically measured in number of nodes expanded) required to solve a given problem. The putative reduction is strongly tied to the quality of the heuristic function used: employing a perfect function means simply "strolling" onto the solution (i.e., no search de facto), while using a bad function could render the search less efficient than totally uninformed search, such as breadth-first search (BFS) or depth-first search (DFS).

In (Hauptman et al., 2009) we used GP to evolve heuristic functions for the Rush Hour puzzle. We first constructed a "brute-force," iterative-deepening search algorithm, along with several search enhancements—some culled from the literature, some of our own devise—but with no heuristic functions. As expected, this method worked well on relatively simple boards, and even solved most moderately difficult ones within reasonable bounds of space and time. However, when dealing with complex problems, this method yielded inadequate performance.

We moved on to hand-crafting several novel heuristics for this domain, which we then tested empirically. The effect of these heuristics on search efficiency was inconsistent, alternating between decreasing the number of nodes traversed by 70% (for certain initial configurations) and increasing this number by as much as 170% (for other configurations). It was clear at this point that using our heuristics correctly was a difficult task.

Enter GP. Our main set of experiments focused on evolving *combinations* of the basic heuristics devised—a form of hyper-heuristic search. We used the

Table 1-6. Rush Hour: Terminal set of an individual program in the population. B:Boolean, R:Real or Integer. The upper part of the table contains terminals used both in $Condition$ and $Value$ trees, while the lower part regards $Condition$ trees only.

Node name	Type	Return value
$BlockersLowerBound$	R	A lower bound on the number of moves required to remove blocking vehicles out of the red car's path
$GoalDistance$	R	Sum of all vehicles' distances to their locations in the deduced-goal board
$Hybrid$	R	Same as $GoalDistance$, but also add number of vehicles between each car and its designated location
$\{0.0, 0.1 \ldots, 1.0, 1, \ldots, 9\}$	R	Numeric terminals
$IsMoveToSecluded$	B	Did the last move taken position the vehicle at a location that no other vehicle can occupy?
$IsReleasingMove$	B	Did the last move made add new possible moves?
g	R	Distance from the initial board
$PhaseByDistance$	R	$g \div (g + DistanceToGoal)$
$PhaseByBlockers$	R	$g \div (g + BlockersLowerBound)$
$NumberOfSiblings$	R	The number of nodes expanded from the parent of the current node
$DifficultyLevel$	R	The difficulty level of the given problem, relative to other problems in the current problem set.

basic heuristics as building blocks in a GP setting, wherein individuals were embodied as ordered sets of search-guiding rules (or *policies*), the parts of which were GP trees. The effect on performance was profound: evolution proved immensely efficacious, managing to combine heuristics of highly variable utility into composites that were nearly always beneficial, and far better than each separate component.

Table 1-6 lists the heuristics and auxiliary functions we devised. Using these elemental blocks effectively is a difficult task, as it involves solving two major sub-problems:

1 Finding exact conditions regarding *when* to apply each heuristic (in order to avoid the strong inconsistent effect on performance mentioned above).

2 Combining several estimates to get a more accurate one. We hypothesized that different areas of the search space might benefit from the application of different heuristics.

As we wanted to embody both application conditions and combinations of estimates, we decided to evolve ordered sets of control rules, or *policies*. Policies typically have the following structure:

$RULE_1$: 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 a given board), returning its $Value$ part. If no rule fires, the value is taken from the last (default) tree: $Value_{N+1}$. Thus, individuals, while in the form of policies, are still board evaluators (or heuristics)—the value returned by the activated rule is an arithmetic combination of heuristic values, and is thus a heuristic value itself. This suits 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 $N+1$ $Value$ GP-trees. After experimenting with several sizes of policies, we settled on $N = 5$, providing us with enough rules per individual, while avoiding "heavy" individuals with too many rules. The depth limit used both for the $Condition$ and $Value$ trees was empirically set to 5. The function set included $\{AND, OR, \leq, \geq\}$ for condition trees and $\{\times, +\}$ for value trees.

Fitness scores were obtained by performing full IDA* search with the given individual used as the heuristic function. For each solved board we assigned to the individual a score equal to the percentage of nodes reduced, compared to searching with no heuristics. For unsolved boards this score was 0. Scores were averaged over 10 randomly selected boards from the training set.

Our results (which garnered another HUMIE in 2009) proved excellent. Indeed, our success with 6x6 boards led us to evolve more problem instances, specifically, difficult 8x8 boards. Overall, evolved policies managed to cut the amount of required search to 40% for 6x6 boards and to 10% for 8x8 boards, compared to iterative deepening.

(Baum and Durdanovic, 2000) tackled Rush Hour with an artificial economy of agents, their best reported solver able to solve 15 of the 40 standard problems (we solved all). Interestingly, they also tried a GP approach, noting, "We have tried several approaches to getting a Genetic Program to solve these problems, varying the instance presentation scheme and other parameters... it has never learned to solve any of the original problem set." Obviously, with the right GP approach, Rush Hour can be solved.

5. Lunch Isn't Free—But Cells Are

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 until 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 foundations. Legal move destinations include: a home cell, if all previous (i.e., lower) cards are already there; empty

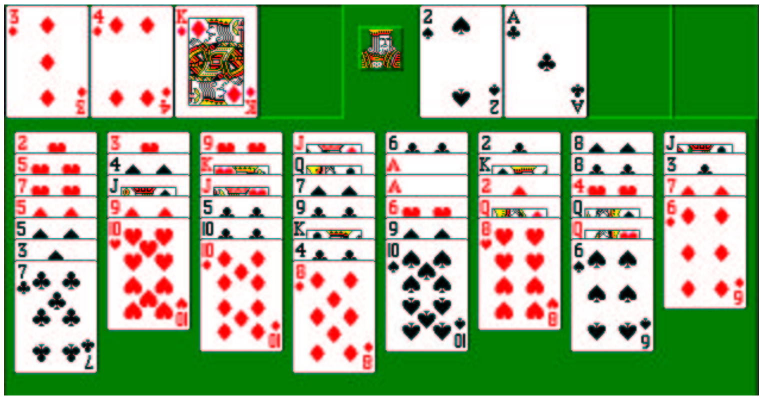

Figure 1-4. A FreeCell game configuration. Cascades: Bottom 8 piles. Foundations: 4 upper-right piles. FreeCells: 4 upper-left piles. Note that cascades are not arranged according to suits, but foundations are. Legal moves for current configuration: 1) moving 7♣ from the leftmost cascade to either the pile fourth from the left (on top of the 8♢), or to the pile third from the right (on top of the 8♡); 2) moving the 6♢ from the right cascade to the left one (on top of the 7♣); and 3) moving any single card on top of a cascade onto the empty FreeCell.

FreeCells; and, on top of a next-highest card of opposite color in a cascade (Figure 1-4). FreeCell was proven by Helmert to be NP-Complete (Helmert, 2003). 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). Despite there being numerous FreeCell solvers available via the Web, few have been written up in the scientific literature. The best published solver to date was that of Heineman (Heineman, 2009), able to solve 96% of Microsoft 32K using a hybrid A* / hill-climbing search algorithm called *staged deepening* (henceforth referred to as the *HSD* algorithm). The HSD algorithm, along with a heuristic function, forms Heineman's FreeCell solver (we made a distinction between the HSD algorithm, the HSD heuristic, and the HSD solver—which includes both).

In (Elyasaf et al., 2011) we used a genetic algorithm to develop the strongest known heuristic-based solver for the game of FreeCell, substantively surpassing that of Heineman's. Along the way we devised several novel heuristics for FreeCell (Table 1-7), many of which could be applied to other domains and games.

Table 1-7. FreeCell: List of heuristics used by the genetic algorithm. R: Real or Integer.

Node name	Type	Return value
HSDH	R	Heineman's staged deepening heuristic
NumberWellPlaced	R	Number of well-placed cards in cascade piles
NumCardsNotAtFoundations	R	Number of cards not at foundation piles
FreeCells	R	Number of free FreeCells and cascades
DifferenceFromTop	R	Average value of top cards in cascades minus average value of top cards in foundation piles
LowestHomeCard	R	Highest possible card value minus lowest card value in foundation piles
HighestHomeCard	R	Highest card value in foundation piles
DifferenceHome	R	Highest card value in foundation piles minus lowest one
SumOfBottomCards	R	Highest possible card value multiplied by number of suites, minus sum of cascades' bottom card

Combining several heuristics to get a more accurate one is considered one of the most difficult problems in contemporary heuristics research (Burke et al., 2010; Samadi et al., 2008). In (Elyasaf et al., 2011) we tackled a sub-problem, that of combining heuristics by *arithmetic* means, by summing their values or taking the maximal value. The problem of combining heuristics is difficult primarily because it entails traversing an extremely large search space of possible numeric combinations and game configurations. To tackle this problem we used a genetic algorithm.

Each individual comprised 9 real values in the range $[0, 1)$, representing a linear combination of all 9 heuristics listed in Table 1-7. Specifically, the heuristic value, H, designated by an evolving individual was defined as $H = \sum_{i=1}^{9} w_i h_i$, where w_i is the ith weight specified by the genome, and h_i is the ith heuristic shown in Table 1-7. To obtain a more uniform calculation we normalized all heuristic values to within the range $[0, 1]$ by maintaining a maximal possible value for each heuristic, and dividing by it. For example, *DifferenceHome* returns values in the range $[0, 13]$ (13 being the difference between the king's value and the ace's value), and the normalized values are attained by dividing by 13.

An individual's fitness score was obtained by performing full HSD search on deals (initial configurations) 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 by 70% compared with HSDH (i.e., 70% less 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.

To distinguish between individuals that did not solve a given deal and individuals that solved it but without reducing the amount of search (the latter case reflecting better performance than the former), we assigned to the latter a partial score of $(1 - FractionExcessNodes)/C$, where *FractionExcessNodes* is the fraction of excessive nodes (values greater than 1 were truncated to 1), and *C* is a constant used to decrease the score relative to search reduction (set empirically to 1000). For example, an excess of 30% would yield a partial score of $(1 - 0.3)/C$; an excess of over 200% would yield 0.

We used Hillis-style coevolution wherein a population of solutions coevolves alongside a population of problems (Hillis, 1990). The basic idea is that neither population should stagnate: As solvers become more adept at solving certain problems these latter do not remain in the problem set (as with a simple GA) but are rather removed from the population of problems—which itself evolves. In this form of competitive coevolution the fitness of one population is inversely related to the fitness of the other population. Specifically, in our coevolutionary scenario the first population comprised the solvers and in the second population an individual represented a *set* of FreeCell deals.

And the winner was... GA-FreeCell. Our evolutionarily produced FreeCell solver significantly surpasses the best published solver to date by three distinct measures: 1) Number of search nodes is reduced by 87%; 2) time to solution is reduced by 93%; and 3) average solution length is reduced by 41%. Our top solver is the best published FreeCell player to date, solving 98% of the standard Microsoft 32K problem set, and also able to beat high-ranking human players.

One of our best solvers is the following: (+ (* DifferenceToGoal 0.09) (* DifferenceToNextStepHome 0.01) (* FreeCells 0.0) (* DifferenceFromTop 0.77) (* LowestHomeCard 0.01) (* UppestHomeCard 0.08) (* NumOfArranged 0.01) (* DifferenceHome 0.01) (* BottomCardsSum 0.02)). (In other good solvers DifferenceFromTop was less weighty.)

How does our evolution-produced player fare against humans? The website `www.freecell.net` provides a ranking of human FreeCell players, listing solution time and win rate (alas, no data on number of boards examined by humans, nor on solution lengths). The site statistics, which we downloaded on April 12, 2011, included results for 76 humans who met our minimal-game requirement of 30K games—all but two of whom exhibited a win rate greater than 91%. Sorted according to number of games played, the no. 1 player played 147,219 games, achieving a win rate of 97.61%. This human is therefore pushed to the second position, with our top player (98.36% win rate) taking the first place. If the statistics are sorted according to win rate then our player assumes the no. 9 position. Either way, it is clear that when compared with strong, persistent, and consisted humans GA-FreeCell emerges as a highly competitive player.

This work may well have pushed the limit of what has been done with evolution, FreeCell being one of the most difficult single-player domains (if not the most difficult) to which evolutionary algorithms have been applied to date.

6. A Racy Affair

Controlling a moving vehicle is considered a complex problem, both in simulated and real-world environments. Dealing with physical forces, varying road conditions, unexpected opponent behavior, damage control, and many other factors, render the car-racing problem a fertile ground for artificial intelligence research.

In (Shichel and Sipper, 2011) we applied GP to creating car controller programs for the game of RARS—Robot Auto Racing Simulator, which is an open-source, car-race simulator. We chose RARS mainly because of its extensive human-written driver library, and the substantive amount of published works that describe machine-learning approaches applied to RARS—enabling us to perform significant comparisons between our results and both human- and machine-designed solutions.

A RARS controller is a C++ class with a single method, which receives the current race *situation* and determines the desired speed and wheel angle of the car. The simulation engine queries the controller approximately 20 times per "game second," and advances the car according to the returned decisions and physical constraints. The *situation* argument provides the agent (car controller) with detailed information about the current race conditions, such as current speed and direction, road curvature, fuel status, and nearby car positions.

Controlling the car is done by two actuators: speed and steering. The speed actuator specifies the desired speed of the car, while the steering actuator specifies the desired wheel angle. The simulation engine uses both values to calculate the involved physical forces and compute the car's movement. Extreme values, such as high speed or a steep steering angle, may result in slippage or skidding, and must be taken into consideration when crafting a controller.

Using GP to evolve driver controllers we created highly generalized game-playing agents, able to outperform most human-crafted controllers and all machine-designed ones on a variety of game tracks (Shichel and Sipper, 2011).

The evolved drivers demonstrated a high degree of generalization, enabling them to perform well on most tracks—including ones that were not used during the evolutionary process. We noted that using a complex track for fitness evaluation, coupled with a comprehensive yet simple set of genetic building blocks, contributed greatly to our controllers' generalization capabilities. We also analyzed the evolved code, observing the emergence of useful patterns, such as a time-to-crash indicator. Such patterns, not pre-coded into the system, were repeatedly used in the evolved individuals' code, acting as evolution-made genetic building blocks.

7. In Conclusion... Evolution Rocks!

Our extensive experience shows that evolutionary algorithms, and in particular genetic programming, are able to parlay fragmentary human intuition into full-fledged winning strategies. Such algorithms are therefore a good candidate when confronted with the problem of finding a complex, successful game strategy or puzzle solver.

We believe that a major reason for our success in evolving winning game strategies is genetic programming's ability to readily accommodate human expertise. When tackling a new game we begin our experimentation with small sets of functions and terminals, which are then revised and added upon through our examination of evolved players and their performance. For example, (Sipper et al., 2007) described three major steps in the development of the evolutionary chess setup.

Genetic programming represents a viable means to automatic programming, and perhaps more generally to machine intelligence, in no small part due to its ability to "cooperate" with humans: more than many other adaptive-search techniques genetic programming's representational affluence and openness lend it to the ready imbuing of the designer's own intelligence within the genotypic language. While AI purists may wrinkle their noses at this, taking the AI-should-emerge-from-scratch stance, we argue that a more practical path to AI involves man-machine cooperation. Genetic programming, as evidenced herein, is a forerunning candidate for the 'machine' part.

AI practitioners sometimes overlook the important distinction between two phases of intelligence (or knowledge) development: 1) from scratch to a mediocre level, and 2) from mediocre to expert level. Traditional AI is often better at handling the first phase. Genetic programming allows the AIer to focus his attention on the second phase, namely, the attainment of true expertise. When aiming to develop a winning strategy, be it in games or any other domain, the genetic-programming practitioner will set his sights at the mediocre-to-expert phase of development, with the scratch-to-mediocre handled automatically during the initial generations of the evolutionary process. Although the designer is "imposing" his own views on the machine, this affords the "pushing" of the artificial intelligence frontier further out.

To summarize:

- Genetic programming has proven to be an excellent tool for automatically generating complex game strategies.
- A crucial advantage of genetic programming lies in its ability to incorporate human intelligence readily.
- As such, genetic programming is an excellent choice when complex strategies are needed, and human intelligence is there for the imbuing.

I evolve therefore I win.

"Why do you not solve it yourself, Mycroft? You can see as far as I."

"Possibly, Sherlock... No, you are the one man who can clear the matter up. If you have a fancy to see your name in the next honours list –"

My friend smiled and shook his head.

"I play the game for the game's own sake," said he.

Arthur Conan Doyle, "The Adventure of the Bruce-Partington Plans"

References

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

Azaria, Yaniv and Sipper, Moshe (2005b). Using GP-gammon: Using genetic programming to evolve backgammon 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 132–142, Lausanne, Switzerland. Springer.

Baum, E. B. and Durdanovic, I. B. (2000). Evolution of cooperative problem solving in an artificial economy. *Neural Computation*, 12:2743–2775.

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.

Burke, E. K., Hyde, M., Kendall, G., Ochoa, G., Ozcan, E., and Woodward, J. 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.

Chellapilla, K. and Fogel, D. B. (2001). Evolving an expert checkers playing program without using human expertise. *IEEE Transactions on Evolutionary Computation*, 5(4):422–428.

Chomsky, N. (1993). *Language and Thought*. Moyer Bell, Wakefield, RI.

Elyasaf, A., Hauptman, A., and Sipper, M. (2011). GA-FreeCell: Evolving solvers for the game of FreeCell. In *GECCO 2011: Proceedings of the Genetic and Evolutionary Computation Conference*, New York, NY, USA. ACM. (accepted).

Epstein, S. L. (1999). Game playing: The next moves. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 987–993. AAAI Press, Menlo Park, California USA.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107.

Hauptman, A. and Sipper, M. (2007a). Emergence of complex strategies in the evolution of chess endgame players. *Advances in Complex Systems*, 10(suppl. no. 1):35–59.

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, Rothlauf, Franz, Squillero, Giovanni, Drechsler, Rolf, Stuetzle, Thomas, Birattari, Mauro, Congdon, Clare Bates, Middendorf, Martin, Blum, Christian, Cotta, Carlos, Bosman, Peter, Grahl, Joern, Knowles, Joshua, Corne, David, Beyer, Hans-Georg, Stanley, Ken, Miller, Julian F., van Hemert, Jano, Lenaerts, Tom, Ebner, Marc, Bacardit, Jaume, O'Neill, Michael, Di Penta, Massimiliano, Doerr, Benjamin, Jansen, Thomas, Poli, Riccardo, and Alba, Enrique, editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 955–962, Montreal. ACM.

Hauptman, Ami and Sipper, Moshe (2005a). Analyzing the intelligence of a genetically programmed chess player. In Rothlauf, Franz, editor, *Late breaking paper at Genetic and Evolutionary Computation Conference (GECCO'2005)*, Washington, D.C., USA.

Hauptman, Ami and Sipper, Moshe (2005b). 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 (2007b). 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.

Hearn, R. A. (2006). *Games, puzzles, and computation*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science.

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

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

Hillis, D. W. (1990). Co-evolving parasites improve simulated evolution in an optimization procedure. *Physica D*, 42:228–234.

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

Kendall, G., Parkes, A., and Spoerer, K. (2008). A survey of NP-complete puzzles. *International Computer Games Association Journal (ICGA)*, 31:13–34.

Kendall, G. and Whitwell, G. (2001). An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In *Proceedings of the 2001 Congress on Evolutionary Computation (CEC2001)*, pages 995–1002. IEEE Press.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109.

Koza, John R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

Robertson, E. and Munro, I. (1978). NP-completeness, puzzles and games. *Utilas Mathematica*, 13:99–116.

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.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229.

Schaeffer, J., Burch, N., Bjornsson, Y., Kishimoto, A., Muller, M., Lake, R., Lu, P., and Sutphen, S. (2007). Checkers is solved. *Science*, 317(5844):1518–1522.

Shichel, Y. and Sipper, M. (2011). GP-RARS: Evolving controllers for the Robot Auto Racing Simulator. *Memetic Computing*. (accepted).

Shichel, Yehonatan, Ziserman, Eran, and Sipper, Moshe (2005). GP-robocode: Using genetic programming to evolve robocode 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 143–154, Lausanne, Switzerland. Springer.

Sipper, Moshe, Azaria, Yaniv, Hauptman, Ami, and Shichel, Yehonatan (2007). Designing an evolutionary strategizing machine for game playing and beyond. *IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews*, 37(4):583–593.

Sipper, Moshe and Giacobini, Mario (2008). Introduction to special section on evolutionary computation in games. *Genetic Programming and Evolvable Machines*, 9(4):279–280.

Smith, M. and Sailer, F. (2004). Learning to beat the world Lose Checkers champion using TDLeaf(λ).