Evolutionary Design of FreeCell Solvers

Achiya Elyasaf, Ami Hauptman, and Moshe Sipper

***Abstract*—We evolve heuristics to guide staged deepening search for the hard game of FreeCell, obtaining top-notch solvers for this human-challenging puzzle. We first devise several novel heuristic measures using minimal domain knowledge and then use them as building blocks in two evolutionary setups involving a standard genetic algorithm and policy-based, genetic programming. Our evolved solvers outperform the best FreeCell solver to date by three distinct measures: 1) number of search nodes is reduced by over 78%; 2) time to solution is reduced by over 94%; and 3) average solution length is reduced by over 30%. Our top solver is the best published FreeCell player to date, solving 99.65% of the standard Microsoft 32K problem set. Moreover, it is able to convincingly beat high-ranking human players.**

***Index Terms*—Evolutionary Algorithms, Genetic Algorithms, Genetic Programing, Heuristic, Hyper Heuristic, FreeCell**

I. Introduction

DISCRETE puzzles, also known as single-player games, are an excellent problem domain for artificial intelligence research, because they can be parsimoniously described yet are often hard to solve [1]. As such, puzzles have been the focus of substantial research in AI during the past decades (e.g., [2], [3]). Nonetheless, quite a few NP-Complete puzzles have remained relatively neglected by academic researchers (see [4] for a review).

Search algorithms for puzzles (as well as for other types of problems) are strongly based on 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 a *heuristic function*. By applying such a function to states reachable from the current one considered, it becomes possible to select more-promising alternatives earlier 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).

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 *free cells*), whose purpose is to aid with moving the cards. Only exposed cards can be moved, either from free cells or cascades. Legal move destinations include: a home (foundation) cell, if all previous (i.e., lower) cards are already there; empty free cells; and, on top of a next-highest card of opposite color in a cascade (Figure 1). FreeCell was proven by Helmert [5] to be NP-complete. In his paper, Helmert explains that the hardness of the domain is not (or at least not exclusively) due to the difficulty in allocating free cells or empty pile positions, but rather due to the choice of which card to move on top of a pile when there are two possible choices. Computational complexity aside, even in its limited popular version (described below) 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.

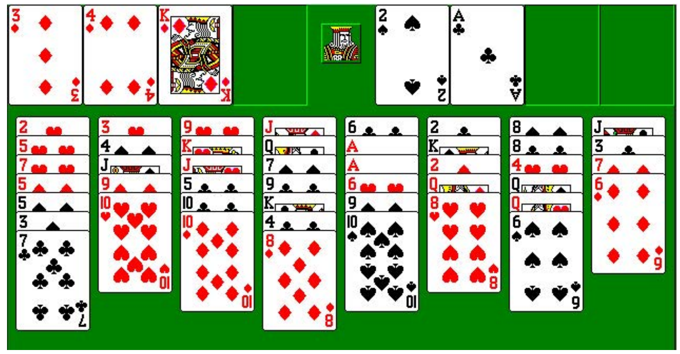

Fig. 1. A FreeCell game configuration. Cascades: Bottom 8 piles. Foundations: 4 upper-right piles. Free cells: 4 upper-left cells. Note that cascades are not arranged according to suits, but foundations are. Legal moves for the current configuration: 1) moving 7♣ from the leftmost cascade to either the 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 free cell.

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 [6]). Due to Microsoft's move, FreeCell has been claimed to be one of the world's most popular games [7]. The Microsoft version of the game comprises a standard deck of 52 cards, 8 cascades, 4 foundations, and 4 free cells. Though limited in size, this FreeCell version still requires an enormous amount of search, due both to long solutions and to large branching factors. Thus it remains out of reach for optimal heuristic search algorithms, such as A* and iterative deepening A* [8], [9], both considered standard methods for solving difficult single-player games (e.g., [10], [11]). FreeCell remains intractable even when powerful enhancement techniques are employed, such as transposition tables [12], [13] and macro moves [14].

The authors are with the Department of Computer Science, Ben-Gurion University, Beer-Sheva, Israel. Emails:{achiya.e,amihau,sipper}@gmail.com

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 is our own GA-based solver [15], [16], [17]. Using a standard GA, we were able to outperform the previous top gun—Heineman's staged deepening algorithm—which is based on a hybrid A* / hill-climbing search algorithm (henceforth referred to as the *HSD* algorithm). The HSD algorithm, along with a heuristic function, forms Heineman's FreeCell solver (we shall distinguish between the HSD algorithm, the HSD heuristic, and the HSD solver—which includes both). Heineman's system exploits several important characteristics of the game, elaborated below.

In a previous work, we successfully applied genetic programming (GP) to evolve heuristic functions for the Rush Hour puzzle—a hard, PSPACE-Complete puzzle [18], [19]. The evolved heuristics dramatically reduced the amount of nodes traversed by an enhanced "brute-force", iterative-deepening search algorithm. Although from a computational-complexity point of view the Rush Hour puzzle is harder than FreeCell (unless *NP=PSPACE*), search spaces induced by *typical* instances of FreeCell tend to be substantially larger than those of Rush Hour, and thus far more difficult to solve. This is evidenced by the failure of standard search methods to solve FreeCell, as opposed to our success in solving all 6x6 Rush Hour problems without requiring any heuristics.

The approach we take in this paper falls within the hyper-heuristic framework, wherein the system is provided with a set of predefined or preexisting heuristics for solving a certain problem, and it tries to discover the best manner in which to apply these heuristics at different stages of the search process. The aim is to find new, higher-level heuristics, or hyper-heuristics [20].

Our main set of experiments focused on evolving combinations of handcrafted heuristics we devised specifically for FreeCell. We used these basic heuristics as building blocks in a GP setting, where individuals were embodied as ordered sets of search-guiding rules (or *policies*), the parts of which were GP trees. We also used a standard genetic algorithm (GA) and standard, tree-based GP (i.e., without policies), both serving as yardsticks for assessing the policy approach's performance (in addition to comparisons with the non-evolutionary methods mentioned above). We employed three different learning methods: Rosin-style coevolution [21], Hillis-style coevolution [22], and a novel method which we call gradual difficulty (described below).

We will show that not only do we solve 99.65% of the Microsoft 32K problem set, a result far better than the best solver on record, but we also do so significantly more efficiently in terms of time to solve, space (number of nodes expanded), and solution length (number of nodes along the path to the correct solution found). The policy-based, GP solvers described herein thus substantively improve upon our previous GA-based solvers [15], [16], [17].

The contributions of this work are as follows:

1) Using genetic programing we develop the strongest known heuristic-based solver for the game of FreeCell.
2) Along the way we devise several novel heuristics for FreeCell, many of which could be applied to other domains and games.
3) We push the limit of what has been done with evolution further, FreeCell being one of the most difficult single-player domains (if not the most difficult) to which evolutionary algorithms have been applied to date.
4) We perform a thorough analysis, applying nine different settings for learning hyper-heuristics to this difficult problem domain.
5) By devising novel heuristics and evolving them into hyper-heuristics, we present a new framework for solving many heuristic problems, which proved to be efficient and successful.

The paper is organized as follows: In the next section we examine previous and related work. In Section III we describe our method, followed by results in Section IV. Next, we discuss our work in Section V. Finally, we end with concluding remarks and future work in Sections VI.

II. Previous Work

We hereby review the work done on FreeCell along with several related topics.

A. Generalized Problem Solvers

Most reported work on FreeCell has been done in the context of automated planning, a field of research in which generalized problem solvers (known as *planning systems* or *planners*) are constructed and tested across various benchmark puzzle domains. FreeCell was used as such a domain both in several International Planning Competitions (IPCs) (e.g., [23]), and in many attempts to construct state-of-the-art planners reported in the literature (e.g., [24], [25]), though in most cases the deck size was less than 52 cards [5]. The version of the game we solve herein, played with a full deck of 52 cards, is considered to be one of the most difficult domains for classical planning [7], evidenced by the poor performance of general-purpose planners.

B. Domain-Specific Solvers

As stated above there are numerous solvers developed specifically for FreeCell available via the web, the best of which is that of Heineman [6]. Although it fails to solve 4% of Microsoft 32K, Heineman's solver significantly outperforms all other solvers in terms of both space and time. We elaborate on this solver in Section III-A.

C. Evolving Heuristics for Planning Systems

Many planning systems are strongly based on the notion of heuristics (e.g., [26], [27]). However, relatively little work has been done on *evolving* heuristics for planning.

Aler et al. [28] (see also [29], [30]) proposed a multi-strategy approach for learning heuristics, embodied as ordered sets of control rules (called *policies*), for search problems in AI planning. Policies were evolved using a GP-based system called EvoCK [30], whose initial population was generated by a specialized learning algorithm, called Hamlet [31]. Their hybrid system, Hamlet-EvoCK, outperformed each of its subsystems on two benchmark problems often used in planning: Blocks World and Logistics (solving 85% and 87% of the problems in these domains, respectively). Note that both these domains are considered relatively easy (e.g., compared to FreeCell), as evidenced by the fact that the last time they were included in an IPC was in 2002.

Levine and Humphreys [32], and later Levine et al. [33], also evolved policies and used them as heuristic measures to guide search for the Blocks World and Logistic domains. Their system, L2Plan, included rule-level genetic programming (for dealing with entire rules), as well as simple local search to augment GP crossover and mutation. They demonstrated some measure of success in these two domains, although hand-coded policies sometimes outperformed the evolved ones.

D. Evolving Heuristics for Specific Puzzles

Terashima-Marín et al. [34] compared two models to produce hyper-heuristics that solved two-dimensional regular and irregular bin-packing problems, an NP-Hard problem domain. The learning process in both of the models produced a rule-based mechanism to determine which heuristic to apply at each state. Both models outperformed the continual use of a single heuristic. We note that their rules classified a state and then applied a (single) heuristic, whereas we applied a *combination* of heuristics at each state, which we believed would perform better.

Hauptman et al. [18], [19] evolved heuristics for the Rush Hour puzzle, a PSPACE-Complete problem domain. They started with blind iterative deepening search (i.e., no heuristics used) and compared it both to searching with handcrafted heuristics, as well as with evolved ones in the form of policies. Hauptman et al. demonstrated that evolved heuristics (with IDA* search) greatly reduce the number of nodes required to solve instances of the Rush Hour puzzle, as compared to the other two methods (blind search and IDA* with handcrafted heuristics).

The problem instances of [18], [19] involved relatively small search spaces—they managed to solve their entire initial test suite using blind search alone (although 2% of the problems violated their space requirement of 1.6 million nodes), and fared even better when using IDA* with handcrafted heuristics (with no evolution required). Therefore, Hauptman et al. designed a coevolutionary algorithm to find more-challenging instances.

Note that *none* of the deals in the Microsoft 32K problem set could be solved with blind search, nor with IDA* equipped with handcrafted heuristics, further evidencing that FreeCell is far more difficult.

We applied a standard genetic algorithm (GA) to evolve solvers for the game of FreeCell, surpassing the top known solver [15], [16] . We will show herein that using policy-based genetic programming we can dramatically improve upon this GA-FreeCell.

The recent book by Sipper [17] provides a thorough account of the previous work on Rush Hour and FreeCell.

III. Methods

Our work on the game of FreeCell progressed in five phases:

1) Construction of an iterative deepening (uninformed) search engine, endowed with several enhancements. Heuristics were not used during this phase.
2) Guiding an IDA* search algorithm with the HSD heuristic function (HSDH).
3) Implementation of the HSD algorithm (including the heuristic function).
4) Design of several novel heuristics and advisors for FreeCell.
5) Evolving heuristics using three different evolutionary algorithms—a standard GA, standard (Koza-style) GP, and policy-based GP—each combined with three types of evolutionary learning mechanisms: Gradual difficulty, Rosin-style coevolution, and Hillis-style coevolution.

A. Search Algorithms

1) Iterative Deepening: We initially implemented standard iterative deepening search [9] as the heart of our game engine. This algorithm may be viewed as a combination of DFS and BFS: starting from a given configuration (e.g., the initial state), with a minimal depth bound, we perform a DFS search for the goal state through the graph of game states (in which vertices represent game configurations, and edges—legal moves). Thus, the algorithm requires only $\theta(n)$ memory, where n is the depth of the search tree. If we succeed, the path is returned. If not, we increase the depth bound by a fixed amount, and restart the search. Note that since the search is incremental, when we find a solution we are guaranteed that it is optimal since a shorter solution would have been found in a previous iteration (more precisely, the solution is optimal or near optimal, depending on whether the depth increase equals 1 or is greater than 1). For difficult problems, such as Rush Hour and FreeCell, finding *a* solution is sufficient, and there is typically no requirement of finding the optimal solution.

An iterative deepening-based game engine receives as input a FreeCell initial configuration (known as a deal), as well as some run parameters, and outputs a solution (i.e., a list of moves) or an indication that the deal could not be solved.

We observed that even when we permitted the search algorithm to use all the available memory (2GB in our case, as opposed to [18] where the node count was limited) virtually all Microsoft 32K problems could not be solved. Hence, we deduced that heuristics were essential for solving FreeCell instances—uninformed search alone was insufficient.

2) Iterative Deepening A:* Given that the HSD solver outperforms all other solvers (except ours), we implemented the heuristic function used by HSD (described in Section III-B) along with the iterative deepening A* (IDA*) search algorithm [9], one of the most prominent methods for solving puzzles (e.g., [10], [11], [35]). This algorithm operates similarly to iterative deepening, except that in the DFS phase heuristic values are used to determine the order by which children of a given node are visited. This move ordering is the only phase wherein the heuristic function is used—the open list structure is still sorted according to depth alone.

IDA* underperformed where FreeCell was concerned, unable to solve many instances (deals). Even using several heuristic functions, IDA*—despite its success in other difficult domains—yielded inadequate performance: less than 1% of the deals we tackled were solved in a reasonable time.

At this point we opted for employing the HSD solver in its entirety, rather than merely the HSD heuristic function.

3) Staged Deepening: Heineman's Staged Deepening (HSD) algorithm is based on the observation that there is no need to store the entire search space seen so far in memory. This is so because of a number of significant characteristics of FreeCell:

- For most states there is more than one distinct permutation of moves creating valid solutions. Hence, very little backtracking is needed.
- There is a relatively high percentage of irreversible moves: according to the game's rules a card placed in a home cell cannot be moved again, and a card moved from an unsorted pile cannot be returned to it.
- If we start from game state s and reach state t after performing k moves, and k is large enough, then there is no longer any need to store the intermediate states between s and t. The reason is that there is a solution from t (first characteristic) and a high percentage of the moves along the path are irreversible anyway (second characteristic).

Thus, the 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. Algorithm 1 presents the pseudocode of the HSD algorithm. S was empirically set by Heineman to 200,000.

Compared with IDA*, HSD uses fewer heuristic evaluations (which are performed only on nodes entering the open list), resulting in a significant reduction in time. Reduction is achieved through the second layer of the search, which stores enough information to perform backtracking (as stated above, this does not occur often), and the size of T is controlled by overwriting nodes.

Although the staged deepening algorithm does not guarantee an optimal solution, as explained above, for difficult problems finding *a* solution is sufficient.

When we ran the HSD solver 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. However we first needed to develop additional heuristics for this domain.

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

Algorithm 1 Heineman's Staged Deepening Algorithm

// Parameter: S, size of transposition table
$T \leftarrow$ initial state
while T not empty **do**
  $s \leftarrow$ remove best state in T according to heuristic value
  $U \leftarrow$ all states exactly k moves away from s, discovered by DFS
  $T \leftarrow$ merge(T, U)
  // merge maintains T sorted by descending heuristic value
  // merge overwrites nodes in T with newer nodes from U of equal heuristic value
  if size of transposition table $\geq S$ **then**
    clear transposition table
  end if
  if *goal* $\in T$ **then**
    return path to goal
  end if
end while

B. 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 available free cells 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 available free cells and cascades.

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

`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. For example, in Figure 1, `SumOfBottomCards` is $100 - (2 + 3 + 9 + 11 + 6 + 2 + 8 + 11) = 48$.

TABLE I
LIST OF HEURISTICS. R: REAL OR INTEGER.

Node name	Type	Return value
`HSDH`	R	Heineman's staged deepening heuristic
`NumWellPlaced`	R	Number of well-placed cards in cascade piles
`NumCardsNotAtFoundations`	R	Number of cards not at foundation piles
`FreeCells`	R	Number of available free cells and cascades
`DifferenceFromTop`	R	Average value of top cards in cascades minus average value of top cards in foundation piles
`LowestFoundationCard`	R	Highest possible card value minus lowest card value in foundation piles
`HighestFoundationCard`	R	Highest card value in foundation piles
`DifferenceFoundation`	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

Table I provides a summary of all heuristics.

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 in Table I. 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 (initial state) 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 number of nodes the HSD solver needed to solve it.

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

Table II provides a list of the auxiliary functions, including the above functions and a number of additional ones.

All of the heuristics and advisors described above are intuitive and straightforward to implement and compute, with their time complexity bounded by the number of cards, i.e., problem input. Furthermore, they are not resource avaricious as are standard heuristic functions, such as relaxation (time consuming) and PDBs (memory consuming).

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.

C. 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 [35], [36].

This task typically involves solving three major subproblems:

1) How to combine heuristics by *arithmetic* means, e.g., by summing their values or taking the maximal value.
2) Finding exact conditions (i.e., *logic* functions) regarding *when* to apply each heuristic, or combinations thereof—some heuristics may be more suitable than others when dealing with specific game configurations.
3) Finding the proper set of game configurations in order to facilitate the learning process while avoiding pitfalls such as overfitting.

The problem of combining heuristics is difficult mainly because 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 call gradual difficulty.

Below we describe the elements of our setup in detail.

1) The Hyper Heuristic-Based Genome: We used three different genomic representations.

Standard GA. This representation was used by us in [15], [16], [17]. This type of hyper-heuristic only addresses the first problem of how to combine heuristics by arithmetic means. Each individual comprises 9 real values in the range $[0, 1]$, representing a linear combination of all 9 heuristics described above (Table I). Specifically, the heuristic value, H, designated by an evolving individual is 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 I. 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, *DifferenceFoundation* 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.

TABLE II
LIST OF AUXILIARY FUNCTIONS. B: BOOLEAN, R: REAL OR INTEGER.

Node name	Type	Return value
IsMoveToFreecell	B	True if last move was to a free cell, false otherwise
IsMoveToCascade	B	True if last move was to a cascade, false otherwise
IsMoveToFoundation	B	True if last move was to a foundation pile, false otherwise
IsMoveToSortedPile	B	True if last move was to a sorted pile, false otherwise
LastCardMoved	R	Value of last card moved
NumOfSiblings	R	Number of reachable states (in one move) from last state
NumOfChildren	R	Number of reachable states (in one move) from current state
DifficultyLevel	R	Index of the current problem in the problem set (sorted by difficulty)
PhaseByX	R	"Mirror" function for each heuristic
g	R	Number of moves made from initial configuration to current

A GA seemed a natural algorithm to employ given the wish to obtain a linear vector of weights. As the results will show, the GA proved quite successful and was therefore retained as a yardstick to measure against when we embarked upon our GP path.

GP. As we wanted to embody both combinations of estimates and application conditions we evolved GP-trees as described in [37]. The function set included the functions $\{IF, AND, OR, \leq, \geq, *, +\}$, and the terminal set included all heuristics and auxiliary functions in Tables I and II, as well as random numbers within the range $[0, 1]$. All the heuristic values were normalized to within the range $[0, 1]$ as performed above with the GA.

This method yielded poor results, no matter what depth limit was used for the trees.

Policies. The last genome used also combines estimates and application conditions, using ordered sets of control rules, or *policies*. As stated above, policies have been evolved successfully with GP to solve search problems—albeit simpler ones (e.g., [18], [19] and [28], mentioned above).

The structure of our policies is the same as the one in [18]:

$RULE_1$: 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 in Tables I and II. The sets of weights appearing in $Value$s all lie within the range $[0, 1]$, and correspond to the heuristics listed in Table I. All the heuristic values are normalized to within the range $[0, 1]$ as described above.

2) Genetic Operators: 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. For all types of genomes we used standard fitness-proportionate selection. We also used elitism—the best individual of each generation was passed onto the next one unchanged.

For simple GA individuals standard reproduction and single-point crossover were applied [38]. Mutation was performed in a manner analogous to bitwise mutation by replacing with independent probability 0.1 a (real-valued) weight by a new random value in the range $[0, 1]$.

We used Koza's standard crossover, mutation, and reproduction operators, for the GP hyper-heuristics [37].

For policies, however, 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 had 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 [39]).

3) GP Parameters: We experimented with several configurations, finally settling upon: population size—between 40 and 60; total generation count—between 300 and 1000, depending on the learning method, as elaborated below; reproduction probability—0.2; crossover probability—0.7; mutation probability—0.1; and elitism set size—1. These settings were applied to all types of hyper-heuristics. A uniform distribution was used for selecting crossover and mutation points within individuals, except for policies, as described above.

4) Training and Test Sets: 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.

The training set for the gradual-difficulty approach was selected anew each run, as described in Section III-D1.

5) Fitness: An individual's fitness score was obtained by running the HSD solver 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.

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 $(1-\mathit{FractionExcessNodes})/C$, where *FractionExcessNodes* was the fraction of excessive nodes (values greater than 1 were truncated to 1), and *C* was 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.

Because of the puzzle's difficulty, some deals were solved by an evolving individual or by HSDH—but not by both, thus rendering comparison (and fitness computation) problematic. To overcome this we imposed a penalty for unsuccessful search: Problems not solved within 2 minutes were counted as requiring 10^9 search nodes. For example, if HSDH did not solve within 2 minutes a deal that an evolving individual did solve using 5×10^8 nodes, the percent of nodes reduced was computed as 50%. The 10^9 value was derived by taking the hardest problem solved by HSDH and multiplying by two the number of nodes required to solve it.

An evolving solver's fitness per single deal, f_i, thus equaled:

$$f_i = \begin{cases} \textit{search-node reduction ratio} \\ \qquad \text{if solution found with node reduction} \\ \\ \max\{\textit{(1-FractionExcessNodes)/1000, 0}\} \\ \qquad \text{if solution found without node reduction} \\ \\ \textit{0} \qquad\qquad \text{if no solution found} \end{cases}$$

and the total fitness, f_s, was defined as the average, $f_s = 1/N\sum_{i=1}^{N} f_i$. Initially we computed fitness by using a constant number, N, of deals (set to 10 to allow diversity while avoiding prolonged evaluations), which were chosen randomly from the training set. However, as the test set was large, fitness scores fluctuated wildly and improvement proved difficult. To overcome this problem we devised a novel learning method which we called *gradual difficulty*.

D. Learning Methods

1) Gradual Difficulty: We first sort the entire Microsoft 32K into groups of increasing difficulty levels. During the course of learning, the difficulty of the problems encountered by individuals is increased by selecting from the more-difficult groups.

Sorting is done according to the number of nodes required to solve each deal with *HSDH*. We divided the problems into 45 groups consisting of 100 problems each. An evolutionary run begins by choosing one random problem from each of the 5 easiest groups (*group01*,. . .,*group05*). We then use only these 5 problems for fitness evaluation. The run continues for 10 generations or until an individual with a fitness score of 0.7 or above is found. Next, we drop the problem from *group01* and replace it with a random problem from *group06*, i.e., we now work with problems from *group02*,. . .,*group06*. This is repeated: drop easiest group, add more-difficult one, until *group45* is used for evaluation, i.e., until we are dealing with groups *group41*,. . .,*group45*. To reduce the effect of overfitting when evaluating with specific groups of problems, we also used a sixth problem for fitness evaluation. This problem was selected from one of the groups that had been dropped, with the number of dropped groups continually growing. The test set used was the remainder of Microsoft 32K.

Note that all the parameters described in this section—total number of groups, number of concurrently used groups, generation count per group, and maximal fitness—were determined empirically.

While some improvement was observed in node reduction and time, the individuals developed with this method failed to solve many of the problems solved by HSDH. This is further discussed in Section IV. Also, the learning process needed over 1000 generations to attain reasonable results.

The major reason for failing to solve many problems when using hyper-heuristics evolved with gradual difficulty learning, is the phenomenon of *forgetting* [40], [41], [42]: over the generations the population becomes adept at solving certain problems, at the expense of "forgetting" to solve other problems it had been adept at in earlier generations.

Coevolution, wherein the population of solutions coevolves alongside a population of problems, offers a solution to this problem . The basic idea is that neither population is allowed to stagnate: As solvers become more adept at solving certain problems these latter do not remain in the problem set but are 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.

2) Rosin-Style Coevolution: The first type of coevolution we tried was Rosin-style coevolution with a *Hall of Fame* [21]. Rosin's method may be viewed as an extension of the elitism concept. The "Hall of Fame" encourages arms races by saving good individuals from prior generations [21].

In this coevolutionary scenario the first population comprises hyper-heuristics—as described above—while the second population consists of FreeCell deals. The populations are equal in size (40). Ten top deals (in terms of difficulty to solve them) are maintained in the Hall of Fame for future testing. Each hyper-heuristic individual is given 5 deals to solve from the deals population and 2 instances from the Hall of Fame. Thus each deal is provided as training material to more than one hyper-heuristic.

The genome and genetic operators of the solver population were identical to those defined in Section III-C.

We applied GP-style evolution to the deal population in the sense that first an operator (reproduction 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. Mutation was applied by replacing a random deal with another random deal from the training set. We did not use crossover.

Fitness was assigned to a solver by averaging its performance over the 7 deals, as described in Section III-C.

A deal individual's fitness was defined as the average number of nodes needed to solve it, 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 10^9 nodes was assigned to it. This way the fitness of deals was inversely proportional to the hyper-heuristics' fitness, so that if a deal was solved easily (with a relatively small number of nodes) on average—it was assigned a low fitness.

Unfortunately, this method proved unsuccessful for our problem domain, regardless of the parameter settings. Rosin-style coevolution is based on the assumption that the more the FreeCell deals that accumulate in the Hall of Fame are harder, the more the hyper-heuristics will improve. Although this assumption might hold for some domains it is untrue for FreeCell due to the difficulty of defining *hard* problems. While for some states a heuristic function might provide a good estimate, for other states it might provide bad estimates [43]. This means that there is no inherently hard or easy state for a heuristic; therefore, a hard-to-solve Hall of Fame deal in a certain generation will be easy to solve a few generations later when the hyper-heuristic individuals have specialized in the new type of deals and have "forgotten" how to solve the previous ones. If at some point a hyper-heuristic performs badly on some deals in the Hall of Fame, we do not know whether the hyper-heuristic is bad all around or perhaps it performs well on other types of deals. The evolutionary process exploits this for the benefit of the deal population, and every few generations "hard" deals become "easy" and vice-versa.

Given the fundamental problem of forgetting, a new method for training the hyper-heuristics to classify states and apply different values thereof was needed. Although policies were designed to maintain rules for different states, they need an effective training method to learn the correct questions and values.

Thus we come to Hillis-style coevolution, which proved to be the most successful learning method for FreeCell.

3) Hillis-Style Coevolution: We assumed that if we could train each hyper-heuristic with a subset of deals that somehow represented the entire search space, we would obtain better results. Although Hillis-style coevolution [22] did not originally address this problem, it does provide a solution.

In our new coevolutionary scenario 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 in Section II-C.

The genome of an individual in the deals population contained 6 FreeCell deals, represented as integer-valued indexes from the training set $\{v_1, v_2, \dots, v_{1000}\}$, 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., $\{v_1, v_2, \dots, v_{1000}\}$ (Figure 2). 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).

We experimented with several parameter settings, finally settling on: population size—between 40 and 60, generation count—between 60 and 80, reproduction probability—0.2, crossover probability—0.7, mutation probability—0.1, and elitism set size—1.

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, as described in Section III-C.

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 10^9 nodes was assigned to it.

TABLE III
AVERAGE NUMBER OF NODES, TIME (IN SECONDS), AND SOLUTION LENGTH REQUIRED TO SOLVE ALL MICROSOFT 32K PROBLEMS, ALONG WITH THE NUMBER OF PROBLEMS SOLVED. TWO SETS OF MEASURES ARE GIVEN: 1) UNSOLVED PROBLEMS ARE ASSIGNED A PENALTY, AND 2) UNSOLVED PROBLEMS ARE EXCLUDED FROM THE COUNT. *HSDH* IS THE HEURISTIC FUNCTION USED BY HSD, *GA-FreeCell* IS OUR TOP EVOLVED GA SOLVER [15], AND *Policy-FreeCell* IS THE TOP EVOLVED HYPER-HEURISTIC POLICY, ALL SELECTED ACCORDING TO PERFORMANCE ON THE TRAINING SET.

Heuristic	Learning method	Nodes	Time	Length	Solved
Unsolved problems penalized					
HSDH	-	75,713,179	709	4,680	30,859
GA	Gradual Difficulty	290,209,299	2,612	17,512	17,748
Policy	Gradual Difficulty	261,331,656	2,352	15,782	18,470
GA-FreeCell	Hillis-style coevolution	**16,626,567**	**150**	**1,132**	**31,475**
Policy-FreeCell	Hillis-style coevolution	**3,977,932**	**34.94**	**392**	**31,888**
Unsolved problems excluded					
HSDH	-	1,780,216	44.45	255	30,859
GA	Gradual Difficulty	182,132	1.77	151	17,748
Policy	Gradual Difficulty	178,202	1.71	149	18,470
GA-FreeCell	Hillis-style coevolution	**230,345**	**2.95**	**151**	**31,475**
Policy-FreeCell	Hillis-style coevolution	**385,568**	**2.61**	**177**	**31,888**

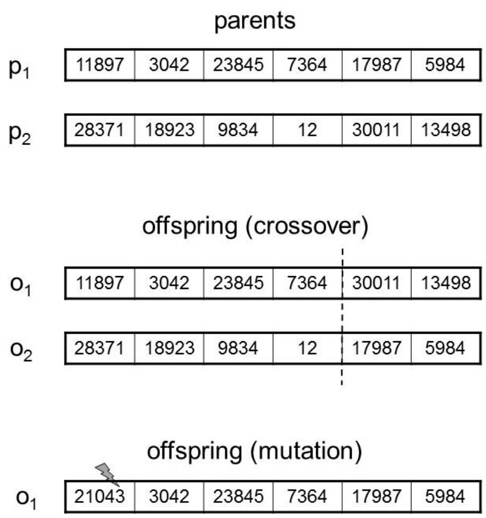

Fig. 2. Crossover and mutation of individuals in the population of problems (deals).

Not only did this method surpass the previous ones, it also outperformed HSDH 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.

IV. RESULTS

We evaluated the performance of evolved heuristics with the same scoring method used for fitness computation, except we averaged over all Microsoft 32K deals instead of over the training set. We also measured average improvement in time, solution length (number of nodes along the path to the correct solution found), and number of solved instances of Microsoft 32K, all compared to the HSD heuristic, HSDH.

We compared the following heuristics: `HSDH` (Section III-B), `HighestFoundationCard` and `DifferenceFoundation` (Section III-B)—both of which proliferated in evolved individuals, and the top hyper-heuristic developed via each of the learning methods.

Table III shows our results. `HighestFoundationCard`, `DifferenceFoundation`, and all GP individuals proved worse than HSD's heuristic function in all of the measures and in all of the experiments and therefore were not included in the tables. In addition, all Rosin-style coevolution experiments failed to solve more than 98% of the problems, and therefore this learning method was not included in the tables as well.

The results for the test set (Microsoft 32K minus 1K training set) and for the entire Microsoft 32K set were very similar, and therefore we report only the latter. The runs proved quite similar in their results, with the number of generations being 1000 on average for gradual difficulty and 300 on average for Hillis-style coevolution. The first few generations took more than 8 hours (on a Linux-based PC, with processor speed 3GHz, and 2GB of main memory) since most of the solvers did not solve most of the deals within the 2-minute time limit. As evolution progressed a generation came to take less than an hour.

For comparing unsolved deals we applied the 10^9 penalty scheme described in Section III-C to the node reduction measure. Since we also compared time to solve and solution length, we applied the penalties of 9,000 seconds and 60,000 moves to these measures, respectively. Since we used this penalty scheme during fitness evaluation we included the penalty in the results as well.

Compared to HSDH, GA-FreeCell [15] and Policy-FreeCell reduced the amount of search by more than 78%, solution time by more than 93%, and solution length by more than 30% (with unsolved problems excluded from the count). In addition, Policy-FreeCell solved 99.65% of Microsoft 32K, thus outperforming both HSDH and GA-FreeCell. Note that although Policy-FreeCell solves "only" 1.3% more instances than GA-FreeCell, these additional deals are far harder to solve due to the long tail of the learning curve.

One of our best Policy solvers is shown in Table IV.

How does our evolution-produced player fare against humans? A major FreeCell website[2] provides a ranking of human FreeCell players, listing solution times and win rates (alas, no data on number of deals examined by humans, nor on solution lengths). This site contains thousands of entries and has been

[2]http://www.freecell.net

TABLE IV
EXAMPLE OF AN EVOLVED POLICY-BASED SOLVER. H_i IS THE iTH HEURISTIC OF TABLE I.

Rule	Condition	Value H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8	H_9
1	(AND (OR (OR ($\leq$ PhaseBySumOfBottomCards 0.58) ($\leq$ NumCardsNotAtFoundations 0.82)) (OR ($\leq$ PhaseBySumOfBottomCards 0.58) ($\leq$ NumCardsNotAtFoundations 0.58))) (OR (OR ($\geq$ PhaseByDifferenceFromTop 0.77) ($\leq$ PhaseByLowestFoundationCard 0.16)) (AND ($\leq$ PhaseByNumWellPlaced 0.21) ($\geq$ IsMoveToSortedPile 0.59))))	0	0.02	0.03	0.41	0	0	0.51	0.02	0.01
2	(OR (OR (OR ($\geq$ PhaseByDifferenceFromTop 0.77) ($\leq$ PhaseByNumWellPlaced 0.16)) (AND ($\leq$ PhaseByNumWellPlaced 0.21) ($\geq$ PhaseByNumWellPlaced 0.59))) (OR (OR ($\geq$ PhaseByDifferenceFromTop 0.77) ($\leq$ PhaseByLowestFoundationCard 0.16)) (AND ($\leq$ PhaseByNumWellPlaced 0.21) ($\geq$ IsMoveToSortedPile 0.59))))	0.2	0.11	0.02	0	0.15	0.03	0.03	0.32	0.14
3	(AND (AND ($\geq$ PhaseByLowestFoundationCard 0.63) ($\geq$ PhaseByLowestFoundationCard 0.63)) ($\geq$ PhaseByLowestFoundationCard 0.63))	0.01	0	0.02	0	0.28	0	0.68	0.01	0
4	(AND ($\leq$ NumCardsNotAtFoundations 0.78) ($\geq$ PhaseByLowestFoundationCard 0.63))	0	0.04	0.09	0	0.02	0.47	0.07	0.26	0.05
5	(OR ($\leq$ HighestFoundationCard 0.44) ($\leq$ HSDH 0.83))	0.3	0.41	0	0.13	0	0	0.09	0.06	0.01
default	—	0.26	0.07	0.03	0.06	0.01	0	0.02	0.52	0.03

TABLE V
THE TOP THREE HUMAN PLAYERS (WHEN SORTED ACCORDING TO NUMBER OF GAMES PLAYED), COMPARED WITH HSDH, GA-FREECELL, AND POLICY-FREECELL. SHOWN ARE NUMBER OF DEALS PLAYED, AVERAGE TIME (IN SECONDS) TO SOLVE, AND PERCENT OF SOLVED DEALS FROM MICROSOFT 32K. TABLE ARRANGED IN DESCENDING ORDER OF WIN RATE (PERCENTAGE OF SOLVED DEALS).

Rank	Name	Deals played	Time	Solved
1	Policy-FreeCell	32,000	3	99.65%
2	GA-FreeCell	32,000	3	98.36%
3	HSDH	32,000	44	96.43%
4	volwin	159,478	190	96.03%
5	deemde	160,237	111	96.02%
6	caralina	151,102	67	65.82%

TABLE VI
THE TOP THREE HUMAN PLAYERS WITH WIN RATE OVER 90% (WHEN SORTED ACCORDING TO AVERAGE TIME TO SOLVE), COMPARED WITH HSDH, GA-FREECELL, AND POLICY-FREECELL. SHOWN ARE NUMBER OF DEALS PLAYED, AVERAGE TIME (IN SECONDS) TO SOLVE, AND PERCENT OF SOLVED DEALS FROM MICROSOFT 32K. TABLE ARRANGED IN DESCENDING ORDER OF WIN RATE (PERCENTAGE OF SOLVED DEALS).

Rank	Name	Deals played	Time	Solved
1	Policy-FreeCell	32,000	3	99.65%
2	GA-FreeCell	32,000	3	98.36%
3	DoubleDouble	48,828	107	96.64%
4	caribsoul	61,617	104	96.56%
5	HSDH	32,000	44	96.43%
6	deemde	160,237	111	96.02%

active since 1996, so the data is reliable. It should be noted that the game engine used by this site generates random deals in a somewhat different manner than the one used to generate Microsoft 32K. Yet, since the deals are randomly generated, it is reasonable to assume that the deals are not biased in any way. Since statistics regarding players who played sparsely are not reliable, we focused on humans who played over 30K games—a figure commensurate with our own.

The site statistics, which we downloaded on December 13, 2011, included results for 83 humans who met the minimal-game requirement—all but two of whom exhibited a win rate greater than 91%. Sorted according to the number of games played, the no. 1 player played 160,237 games, achieving a win rate of 96.02%. This human is therefore pushed to the fourth position, with our top player (99.65% win rate) taking the first place, our GA-FreeCell taking the second place, and HSDH coming in third (Table V).

When sorted according to average solving time, the fastest human player with win rate above 90% solved deals in an average time of 104 seconds and achieved a win rate of 96.56%. This human is therefore pushed to the fourth position, with HSDH in the third place, GA-FreeCell in the second place and Policy-FreeCell taking the first place (Table VI). Note that the fastest human player—caralina—takes 67 seconds on average to reach a solution (Table V). HSDH reduces caralina's average time by 34.3%, while our evolved solvers reduce the average time by 95.5%. These values suggest that outperforming human players in time-to-solve is not a trivial task for a computer. Yet, our evolved solvers manage to shine with respect to time as well.

If the statistics are sorted according to win rate then our Policy-FreeCell player takes the first place with a win rate of 99.65%, while GA-FreeCell attains the respectable 11th place. Either way, it is clear that when compared with strong, persistent, and consistent humans, Policy-FreeCell emerges as the new best player to date, leaving HSDH far behind

TABLE VII
THE TOP THREE HUMAN PLAYERS (WHEN SORTED ACCORDING TO WIN RATE), COMPARED WITH HSDH, GA-FREECELL, AND POLICY-FREECELL. SHOWN ARE NUMBER OF DEALS PLAYED, AVERAGE TIME (IN SECONDS) TO SOLVE, AND PERCENT OF SOLVED DEALS FROM MICROSOFT 32K. TABLE ARRANGED IN DESCENDING ORDER OF WIN RATE (PERCENTAGE OF SOLVED DEALS).

Rank	Name	Deals played	Time	Solved
1	Policy-FreeCell	32,000	3	99.65%
2	JonnieBoy	39,102	270	99.33%
3	time.waster	37,286	191	99.20%
4	Nat_King_C.	54,599	207	98.97%
...				
11	GA-FreeCell	32,000	3	98.36%
...				
66	HSDH	32,000	44	96.43%

(Table VII).

V. Discussion

Although policies can be seen as a special case of GP trees they yielded good results for this domain while GP did not. A possible reason for this is that the policy structure is more apt for this type of problems. The policy conditions classify states while the values combine the available heuristics. When standard tree-GP is used, the structure is not clear and many meaningless trees are generated.

Another interesting point is the difference in the results between GA-FreeCell and Policy-FreeCell. 80% of the problems not solved by GA-FreeCell were solved by Policy-FreeCell, leaving only 112 unsolved problems by the latter. On the other hand, the search reduction measures were similar. We thus concluded that for most of the states a simple GA individual would have sufficed, but in order to attain a leap in success rate the use of policies proved necessary.

In general, when the evaluation time of an individual is short, large populations may be used; moreover, we can afford to evaluate each individual on many randomly selected instances, perhaps even on the entire training set, thereby attaining a reliable fitness measure. In such cases gradual difficulty might contribute to the evolutionary process. However, with long evaluation times an individual can be tested against but a small subset of the entire training set, and this part will not be representative of the whole. The learning process will then exhibit “forgetfulness” and “specialization”, as described in Section III-D. As we saw, Hillis-style coevolution solved these problems since we did not need to know a priori which deals to use for the learning process.

Lastly, the heuristics and advisors used as building blocks for the evolutionary process are intuitive and straightforward to implement and compute. Yet, our evolved solvers are the top solvers for the game of FreeCell, suggesting that in some domains good solvers can be achieved with minimal domain knowledge and without the use of much domain expertise. It should be noted that complex heuristics and memory-consuming heuristics (e.g., landmarks and pattern databases) can be easily used as building blocks as well. Such solvers might outperform the simpler ones at the expense of increased run time or code complexity.

VI. Concluding Remarks

We evolved a solver for the FreeCell puzzle, one of the most difficult single-player domains (if not the most difficult) to which evolutionary algorithms have been applied to date. Policy-FreeCell and GA-FreeCell beat the previous top published solver by a wide margin on several measures, with the former emerging as the top gun. By classifying states and assigning different values to different states, Policy-FreeCell was able to solve 99.65% of Microsoft 32K, a result far better than any previous solver.

There are a number of possible extensions to our work, including:

1) It is possible to implement FreeCell macro moves and thus decrease the search space. Implementing macro moves will yield better results, and we believe that we might even solve the entire Microsoft 32K (not including unsolvable game #11982).
2) As mentioned in Section V, complex heuristics and memory-consuming heuristics (e.g., landmarks and pattern databases) can easily be used as building blocks as well. Such solvers might outperform the simpler ones at the expense of increased run time or code complexity.
3) The HSD algorithm, enhanced with evolved heuristics, is more efficient than the original version. This is evidenced both by the amount of search reduction and the increased number of solved deals. It remains to be determined whether the algorithm, when aided by evolution, can outperform other widely used algorithms (such as IDA*) in different domains. The fact that the algorithm is based on several properties of search problems, such as the high percentage of irreversible moves and the small number of deadlocks, already points the way towards several domains. A good candidate may be the Satellite game, previously studied in [44], [45].
4) Handcrafted heuristics may themselves be improved by evolution. This could be done by breaking them into their elemental components and evolving their combinations thereof.
5) Many single-agent search problems fall within the framework of AI-planning problems (e.g., with ADL [46]). However, using evolution in conjunction with these techniques is not trivial and may require the use of techniques such as GP policies [18].

Acknowledgments

Achiya Elyasaf is partially supported by the Lynn and William Frankel Center for Computer Sciences.

References

[1] J. Pearl, *Heuristics*. Reading, Massachusetts: Addison–Wesley, 1984.
[2] E. Robertson and I. Munro, “NP-completeness, puzzles and games,” *Utilas Mathematica*, vol. 13, pp. 99–116, 1978.
[3] R. A. Hearn, “Games, puzzles, and computation,” Ph.D. dissertation, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2006.
[4] G. Kendall, A. Parkes, and K. Spoerer, “A survey of NP-complete puzzles,” *ICGA Journal*, vol. 31, pp. 13–34, 2008.
[5] M. Helmert, “Complexity results for standard benchmark domains in planning,” *Artificial Intelligence*, vol. 143, no. 2, pp. 219–262, 2003.

[6] G. T. Heineman, "Algorithm to solve FreeCell solitaire games," January 2009, http://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.

[7] F. Bacchus, "AIPS'00 planning competition," *AI Magazine*, vol. 22, no. 1, pp. 47–56, 2001.

[8] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for heuristic determination of minimum path cost," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, February 1968.

[9] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial Intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[10] A. Junghanns and J. Schaeffer, "Sokoban: A challenging single-agent search problem," in *Workshop on Using Games as an Experimental Testbed for AI Research, Proceedings IJCAI-97*, 1997, pp. 27–36.

[11] R. E. Korf, "Finding optimal solutions to Rubik's cube using pattern databases," in *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, ser. AAAI'97/IAAI'97. AAAI Press, 1997, pp. 700–705.

[12] P. W. Frey, *Chess Skill in Man and Machine*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1979.

[13] L. A. Taylor and R. E. Korf, "Pruning duplicate nodes in depth-first search," in *Proceedings of the Eleventh National Conference on Artificial Intelligence*, ser. AAAI'93. AAAI Press, 1993, pp. 756–761.

[14] R. E. Korf, "Macro-operators: A weak method for learning," *Artificial Intelligence*, vol. 26, pp. 35–77, 1985.

[15] A. Elyasaf, A. Hauptman, and M. Sipper, "GA-FreeCell: Evolving Solvers for the Game of FreeCell," in *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, N. Krasnogor *et al.*, Eds. Dublin, Ireland: ACM, 12-16 Jul. 2011, pp. 1931–1938.

[16] A. Elyasaf, Y. Zaritsky, A. Hauptman, and M. Sipper, "Evolving solvers for FreeCell and the sliding-tile puzzle," in *Proceedings of the Fourth Annual Symposium on Combinatorial Search, SOCS 2011*, D. Borrajo, M. Likhachev, and C. L. López, Eds. AAAI Press, 2011.

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

[18] A. Hauptman, A. Elyasaf, M. Sipper, and A. Karmon, "GP-Rush: using genetic programming to evolve solvers for the Rush Hour puzzle," in *GECCO'09: Proceedings of 11th Annual Conference on Genetic and Evolutionary Computation Conference*. New York, NY, USA: ACM, 2009, pp. 955–962.

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

[20] M. Bader-El-Den, R. Poli, and S. Fatima, "Evolving timetabling heuristics using a grammar-based genetic programming hyper-heuristic framework," *Memetic Computing*, vol. 1, no. 3, pp. 205–219, November 2009.

[21] C. D. Rosin, "Coevolutionary search among adversaries," Ph.D. dissertation, University of California, San Diego, CA, 1997.

[22] D. W. Hillis, "Co-evolving parasites improve simulated evolution in an optimization procedure," *Physica D*, vol. 42, pp. 228–234, 1990.

[23] D. Long and M. Fox, "The 3rd international planning competition: Results and analysis," *Journal of Artificial Intelligence Research*, vol. 20, pp. 1–59, 2003.

[24] A. Coles and K. A. Smith, "Marvin: A heuristic search planner with online macro-action learning," *Journal of Artificial Intelligence Research*, vol. 28, pp. 119–156, 2007.

[25] S. Yoon, A. Fern, and R. Givan, "Learning control knowledge for forward search planning," *Journal of Machine Learning Research*, vol. 9, pp. 683–718, Apr. 2008.

[26] B. Bonet and H. Geffner, "mGPT: A probabilistic planner based on heuristic search," *Journal of Artificial Intelligence Research*, vol. 24, pp. 933–944, 2005.

[27] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, May 2000.

[28] R. Aler, D. Borrajo, and P. Isasi, "Using genetic programming to learn and improve knowledge," *Artificial Intelligence*, vol. 141, no. 1–2, pp. 29–56, 2002.

[29] ——, "Evolving heuristics for planning," in *Evolutionary Programming VII*, ser. Lecture Notes in Computer Science, V. Porto, N. Saravanan, D. Waagen, and A. Eiben, Eds. Springer Berlin / Heidelberg, 1998, vol. 1447, pp. 745–754.

[30] ——, "Learning to solve planning problems efficiently by means of genetic programming," *Evolutionary Computation*, vol. 9, no. 4, pp. 387–420, Winter 2001.

[31] D. Borrajo and M. M. Veloso, "Lazy incremental learning of control knowledge for efficiently obtaining quality plans," *Artificial Intelligence Review*, vol. 11, no. 1-5, pp. 371–405, 1997.

[32] J. Levine and D. Humphreys, "Learning action strategies for planning domains using genetic programming," in *EvoWorkshops*, ser. Lecture Notes in Computer Science, G. R. Raidl, J.-A. Meyer, M. Middendorf, S. Cagnoni, J. J. R. Cardalda, D. Corne, J. Gottlieb, A. Guillot, E. Hart, C. G. Johnson, and E. Marchiori, Eds., vol. 2611. Springer, 2003, pp. 684–695.

[33] J. Levine, H. Westerberg, M. Galea, and D. Humphreys, "Evolutionary-based learning of generalised policies for AI planning domains," in *Proceedings of the 11th Annual conference on Genetic and Evolutionary Computation (GECCO 2009)*, F. Rothlauf, Ed. New York, NY, USA: ACM, 2009, pp. 1195–1202.

[34] H. Terashima-Marín, P. Ross, C. J. F. Zárate, E. López-Camacho, and M. Valenzuela-Rendón, "Generalized hyper-heuristics for solving 2D regular and irregular packing problems," *Annals OR*, vol. 179, no. 1, pp. 369–392, 2010.

[35] M. Samadi, A. Felner, and J. Schaeffer, "Learning from multiple heuristics," in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, D. Fox and C. P. Gomes, Eds. AAAI Press, 2008, pp. 357–362.

[36] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and J. R. Woodward, "A classification of hyper-heuristic approaches," in *Handbook of Meta-Heuristics 2nd Edition*, M. Gendreau and J. Potvin, Eds. Springer, 2010, pp. 449–468.

[37] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge Massachusetts: MIT Press, May 1994.

[38] J. H. Holland, *Adaptation in Natural Artificial Systems*. Ann Arbor: University of Michigan Press, 1975.

[39] D. J. Montana, "Strongly typed genetic programming," *Evolutionary Computation*, vol. 3, no. 2, pp. 199–230, 1995.

[40] S. G. Ficici and J. B. Pollack, "A game-theoretic memory mechanism for coevolution," in *Genetic and Evolutionary Computation – GECCO-2003*, E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, Eds. Berlin: Springer-Verlag, 2003, pp. 286–297.

[41] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Englewood Cliffs , NJ: Prentice-Hall, 2010.

[42] C. Birchenhall, N. Kastrinos, and S. Metcalfe, "Genetic algorithms in evolutionary modelling," *Journal of Evolutionary Economics*, vol. 7, no. 4, pp. 375–393, 1997.

[43] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, ser. Addison-Wesley series in artificial intelligence. Addison-Wesley, 1984.

[44] P. Haslum, B. Bonet, and H. Geffner, "New admissible heuristics for domain-independent planning," in *AAAI '05: Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, M. M. Veloso and S. Kambhampati, Eds. AAAI Press / The MIT Press, 2005, pp. 1163–1168.

[45] M. Helmert, *Understanding Planning Tasks: Domain Complexity and Heuristic Decomposition*, ser. Lecture Notes in Computer Science. Springer, 2008, vol. 4929.

[46] E. Pednault, "ADL: Exploring the middle ground between STRIPS and the situation calculus," in *Proceedings of 1st international conference on Principles of Knowledge Representation and Reasoning*, 1989, pp. 324–332.