GA-FreeCell: Evolving Solvers for the Game of FreeCell

Achiya Elyasaf
Ben-Gurion University of the Negev, Be'er Sheva, Israel
achiya.e@gmail.com

Ami Hauptman
Ben-Gurion University of the Negev, Be'er Sheva, Israel
amihau@gmail.com

Moshe Sipper
Ben-Gurion University of the Negev, Be'er Sheva, Israel
sipper@cs.bgu.ac.il

ABSTRACT

We evolve heuristics to guide staged deepening search for the hard game of FreeCell, obtaining top-notch solvers for this NP-Complete, human-challenging puzzle. We first devise several novel heuristic measures and then employ a Hillis-style coevolutionary genetic algorithm to find efficient combinations of these heuristics. Our results significantly surpass 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.

Track: Self-* Search

Categories and Subject Descriptors

I.2.1 [**Applications and Expert Systems**]: Games; I.2.6 [**Parameter learning**]: Knowledge acquisition; I.2.8 [**Problem Solving, Control Methods, and Search**]: Heuristic methods

General Terms

Algorithms, Performance, Design

Keywords

Genetic Algorithms, Heuristics, FreeCell Puzzle, Single-Agent Search, Hyper-Heuristics

1. 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 [28]. As such, puzzles have been the focus of substantial research in AI during the past decades (e.g., [14, 31]). Nonetheless, quite a few NP-Complete puzzles have remained relatively neglected by academic researchers (see [21] for a review).

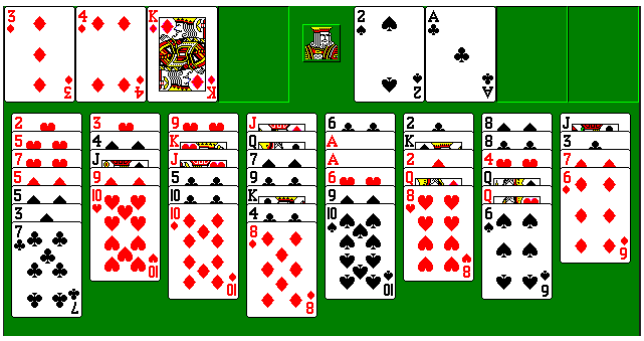

Figure 1: A FreeCell game configuration. Cascades: Bottom 8 piles. Foundations: 4 upper-right piles. FreeCells: 4 upper-left piles. Note that cascades are not arranged according to suits, but foundations are. Legal moves for current configuration: 1) moving 7♣ from the leftmost cascade to either the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

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 FreeCells; and, on top of a next-highest card of opposite color in a cascade (Figure 1). FreeCell was proven by Helmert to be NP-Complete [16]. 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

GECCO'11, July 12–16, 2011, Dublin, Ireland.

versions), along with 32,000 problems—known as *Microsoft 32K*—all solvable but one (this latter, game #11982, was proven to be unsolvable). Due to Microsoft's move FreeCell has been claimed to be one of the world's most popular games [4]. The Microsoft version of the game comprises a standard deck of 52 cards, 8 cascades, 4 foundations, and 4 FreeCells. 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* [10, 22], both considered standard methods for solving difficult single-player games (e.g., [20, 24]). FreeCell remains unsolvable even when powerful enhancement techniques are employed, such as transposition tables [9, 33] and macro moves [23].

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 that of Heineman [15], 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 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.

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 the *heuristic function*. By applying this 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 required to solve a given problem (typically measured in number of nodes expanded). 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 a recent work Hauptman et al. [13] successfully applied genetic programming (GP) to *evolving* heuristic functions for the Rush Hour puzzle—a hard, PSPACE-Complete puzzle. The evolved heuristics dramatically reduced the amount of nodes traversed by an enhanced "brute-force," iterative-deepening search algorithm. Herein, we employ a genetic algorithm (GA) to obtaining solvers for the difficult FreeCell puzzle. Note that although from a computational-complexity point of view the Rush Hour puzzle is harder (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 their success in solving all 6x6 Rush Hour problems without requiring any heuristics [13].

Our main set of experiments focused on evolving combinations of hand-crafted heuristics we devised specifically for FreeCell. We used these basic heuristics as building blocks in a GA setting, where individuals represented the heuristics' weights. We used Hillis-style competitive coevolution [18] to simultaneously coevolve good solvers and various deals of varying difficulty levels.

We will show that not only do we solve 98% 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 contributions of this work are as follows:

1. Using a genetic algorithm 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.

The paper is organized as follows: In the next section we examine previous and related work. In Section 3 we describe our method, followed by results in Section 4. Finally, we end with concluding remarks and future work in Section 5.

2. PREVIOUS WORK

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

2.1 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., [27]), and in numerous attempts to construct state-of-the-art planners reported in the literature (e.g., [8, 35]). 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 [4], evidenced by the poor performance of general-purpose planners.

2.2 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 [15]. 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 ahead.

2.3 Evolving Heuristics for Planning Systems

Many planning systems are strongly based on the notion of heuristics (e.g., [5, 19]). However, relatively little work has been done on *evolving* heuristics for planning.

Aler et al. [3] (see also [1, 2]) 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 [2], whose initial population was generated by

a specialized learning algorithm, called Hamlet [6]. 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 IPCs was in 2002.

Levine and Humphreys [25], and later Levine et al. [26], 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.

2.4 Evolving Heuristics for Specific Puzzles

Terashima-Marín et al. [34] compared two models to produce hyper-heuristics that solve 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 classify a state and then apply a (single) heuristic, whereas we apply a *combination* of heuristics at each state, which we believed would perform better.

Hauptman et al. [12, 13] 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 hand-crafted 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 hand-crafted heuristics).

The problem instances of Hauptman et al. 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 hand-crafted 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 hand-crafted heuristics, further evidencing that FreeCell is far more difficult.

3. METHODS

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

1. Constructing 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 algorithm's heuristic function (HSDH).
3. Implementation of the HSD algorithm (including the heuristic function).
4. Design of several novel heuristics for FreeCell.
5. Learning weights for these novel heuristics using Hillis-style coevolution.

3.1 Search Algorithms

3.1.1 Iterative Deepening

We initially implemented enhanced iterative deepening search [22] as the heart of our game engine. This algorithm may be viewed as a combination of BFS and DFS: Starting from a given configuration (e.g., the initial state) and a given depth bound, perform a DFS search for the goal state through the graph of game states (in which vertices represent game configurations and edges represent legal moves). If the goal is found, a path to it is returned; if not, the depth bound is increased and the DFS phase is restarted, unless the maximal depth bound has been reached, in which case failure is reported. The basic idea behind ID (and IDA* described below) is that of not storing the part of the search space seen so far but just the path one is "moving" along.

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.

Transposition tables [9, 33] were an important enhancement to our basic engine. Such tables are commonly used to avoid visiting states that were previously visited (thus escaping possible loops) by means of a hash table storing all states already encountered.

We observed that even when we permitted the search algorithm to use all the available memory (2GB in our case, as opposed to [13] 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.

3.1.2 Iterative Deepening A*

As stated above the HSD algorithm outperforms all other solvers. Thus we implemented the heuristic function used by HSD (described in Section 3.2). We used the HSD heuristic with an iterative deepening A* (IDA*) search algorithm [22], one of the most prominent methods for solving puzzles (e.g., [20, 24, 32]).

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 method is generally known as move ordering [30]). 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: without the hash table many instances (deals) were not solved, while using the table resulted in memory overflow time and again. Even using a strong heuristic function, IDA*—despite its success in other difficult domains—yielded inadequate performance: less than 1% of the deals we tackled were solved, with all other instances resulting in memory overflow.

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

3.1.3 Staged Deepening

Staged deepening—used by the HSD algorithm—is based on the observation that there is no need to store the entire search space seen so far in memory, as is done with IDA* search. This is so because of several significant characteristics of FreeCell:

1. For most states there is more than one distinct permutation of moves creating valid solutions. Hence, very little backtracking is needed.
2. 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.
3. 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 (characteristic 1) and a high percentage of the moves along the path are irreversible anyway (characteristic 2).

Thus, the HSD algorithm may be viewed as a 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 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 present the pseudocode of the HSD algorithm. N was empirically set by Heineman to 200,000.

Algorithm 1 HSD (Heineman's staged deepening)

```
1: // Parameter: N – size of transposition table
2: T ← initial state
3: while T not empty do
4:    s ← remove best state in T according to heuristic value
5:    U ← all states exactly k moves away from s, discovered
      by DFS
6:    T ← merge(T, U)
7:    // merge maintains T sorted by descending heuristic
      value
8:    // merge overwrites nodes in T with newer nodes from
      U of equal heuristic value
9:    if size of transposition table ≥ N then
10:      clear transposition table
11:   end if
12:   if goal ∈ T then
13:      return path to goal
14:   end if
15: end while
```

Compared with IDA*, HSD uses less heuristic evaluations (which are performed only on nodes entering the open list), and does periodical memory cleanup, resulting in significant reduction both in time and space requirements. 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, for difficult problems such as FreeCell finding *a* solution is sufficient, and there is typically no requirement to find the optimal solution.

The HSD algorithm 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.

3.2 FreeCell Heuristics

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

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

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

FreeCells. Count the number of free FreeCells and cascades.

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

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

HighestHomeCard. The highest card value in foundation piles.

DifferenceHome. 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 1 lists the above heuristics.

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

3.3 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 [7, 32]. Herein we tackle a subproblem, that of combining heuristics by *arithmetic* means, e.g., by summing their values or taking the maximal value.

The problem of combining heuristics is difficult primarily

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

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

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

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. Below we describe the elements of our setup in detail.

3.3.1 Genome

Each individual comprises 9 real values in the range $[0, 1)$, representing a linear combination of all 9 heuristics described above (Table 1). 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 1. 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.

3.3.2 GA Operators and Parameters

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 a (real-valued) weight by a new random value in the range $[0, 1)$. We experimented with several parameter settings, finally settling on: population size—between 40 and 60, generation count—between 300 and 400, reproduction probability—0.2, crossover probability—0.7, mutation probability—0.1, and elitism set size—1. We used a uniform distribution for selecting crossover points within individuals.

3.3.3 Training and Test Sets

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

3.3.4 Fitness

An individual's fitness score was obtained by performing full HSD search 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 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.

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 seconds were counted as requiring 1000M search nodes. For example, if HSDH did not solve within 2 seconds a deal that an evolving individual did solve using 500M nodes, the percent of nodes reduced was computed as 50%. The 1000M 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 equals:

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

and the total fitness, f_s, is 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 turned to coevolution.

3.4 Hillis-Style Coevolution

We used Hillis-style of coevolution wherein a population of solutions coevolves alongside a population of problems [18]. 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.

In our 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 will contain various deals of varying difficulty levels. This multi-deal individual makes life harder for the evolving solvers: They must maintain a consistent level of play over several deals. With single-deal individuals, which we initially experimented with, either the solvers did not improve if the deal population evolved every generation (i.e., too fast), or the solvers became adept at solving certain deals and failed on others if the deal population evolved more slowly (i.e., every k generations, for a given $k > 1$).

The genome and genetic operators of the solver population were identical to those defined above.

The genome of an individual in the deal population contained 6 FreeCell deals, represented as integer-valued indexes from the set $\{v_1, v_2, \ldots, v_{1000}\}$, where v_i is a random index in the range $[1, 32000]$. We applied GP-style evolution in the sense that first an operator (reproduction, crossover, or mutation) was selected with a given probability, and then one or two individuals were selected in accordance with the operator chosen. We used standard fitness-proportionate selection and single-point crossover. Mutation was performed in a manner analogous to bitwise mutation by replacing with independent probability 0.1 an (integer-valued) index with a randomly chosen deal (index) from the training set (i.e., $\{v_1, v_2, \ldots, v_{1000}\}$). We used a uniform distribution for selecting crossover points within individuals (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 represent. The fitness value was an average of all 12 deals, as described in Section 3.3.4.

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

Coevolution outperformed HSD by a wide margin, solving all but 525 instances of Microsoft 32K, and doing so using significantly less time and space requirements. Additionally, the solutions found were shorter and hence better. We expand upon these results in the next section.

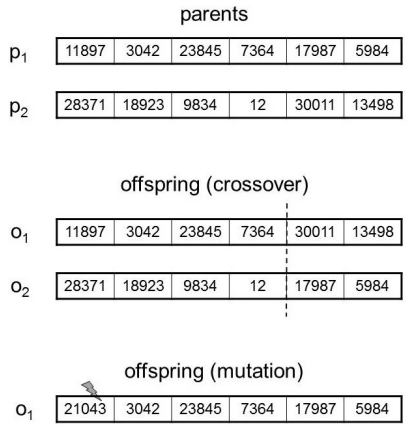

Figure 2: Crossover and mutation of individuals in the population of problems (deals).

4. 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.

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 150 on average. The first few generations took more than 8 hours since most of the solvers did not solve most of the boards within the 2-minute time limit. As evolution progressed a generation came to take less than an hour.

We compared the following heuristics: HSDH (Section 3.2), HighestHomeCard and DifferenceHome (Section 3.2)—both of which proliferated in evolved individuals, and GA-FreeCell—the top evolved individual.

We performed numerous experiments, but due to lack of space we only report herein the main results, summarized in Table 2. The HighestHomeCard and DifferenceHome heuristics proved worse than HSD's heuristic function in all of the measures and therefore were not included in the tables. For comparing unsolved deals we applied the 1000M penalty scheme described in Section 3.3.4 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.

GA-FreeCell reduced the amount of search by 87%, solution time by 93%, and solution length by 41%, compared to HSDH. In addition, GA-FreeCell solved 98% of Microsoft 32K, thus outperforming HSDH, the (now) previously top solver, which solved only 96% of Microsoft 32K. Note that although GA-FreeCell solves "only" 2% more instances, these 2% are far harder to solve due to the long tail of the learning curve.

One of our best solvers is the following: (+ (* DifferenceToGoal 0.09) (* DifferenceToNextStepHome 0.01) (* Free-

Table 2: 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 score of 1000M nodes (as done during fitness computation), and 2) unsolved problems are excluded from the count. HSDH is the heuristic function used by HSD. GA-FreeCell is our top evolved solver.

Heuristic	Nodes	Time	Length	Solved
	unsolved problems penalized			
HSDH	75,713,179	709	4,680	30,859
GA-FreeCell	**16,626,567**	**150**	**1,132**	**31,475**
	unsolved problems excluded			
HSDH	1,780,216	44.45	255	30,859
GA-FreeCell	**230,345**	**2.95**	**151**	**31,475**

Table 3: The top three human players (when sorted according to number of games played), compared with HSDH and GA-FreeCell. Shown are number of deals played and average time (in seconds) to solve.

Name	Deals played	Time	Solved
sugar357	147,219	241	97.61%
volwin	146,380	190	96.00%
caralina	146,224	68	66.40%
HSDH	32,000	44	96.43%
GA-FreeCell	32,000	**3**	**98.36%**

Cells 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). This site contains thousands of entries and has been 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 April 12, 2011, included results for 76 humans who met the minimal-game requirement—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 (Table 3). 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.

5. 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. GA-FreeCell beats the previous top published solver by a wide margin on several measures. A simple genetic algorithm proved adequate, the main thrust of our approach being the designing of basic heuristics and their fruitful combination.

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

1. 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 upon several properties of search problems, such as the high percentage of irreversible moves along with the small number of deadlocks, already points the way towards several domains. A good candidate may be the Satellite game, previously studied in [11, 17].

2. Hand-crafted heuristics may themselves be improved by evolution. This could be done by breaking them into their elemental components and evolving their combinations thereof.

3. Many single-agent search problems fall within the framework of AI-planning problems (e.g., with ADL [29]). However, using evolution in conjunction with these techniques is not trivial and may require the use of techniques such as GP policies [13].

6. ACKNOWLEDGMENTS

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

7. REFERENCES

[1] R. Aler, D. Borrajo, and P. Isasi. Evolving heuristics for planning. In V. Porto, N. Saravanan, D. Waagen, and A. Eiben, editors, *Evolutionary Programming VII*, volume 1447 of *Lecture Notes in Computer Science*, pages 745–754. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0040825.

[2] R. Aler, D. Borrajo, and P. Isasi. Learning to solve planning problems efficiently by means of genetic programming. *Evolutionary Computation*, 9(4):387–420, Winter 2001.

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

[4] F. Bacchus. AIPS'00 planning competition. *AI Magazine*, 22(1):47–56, 2001.

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

[6] D. Borrajo and M. M. Veloso. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *Artificial Intelligence Review*, 11(1-5):371–405, 1997.

[7] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and J. R. Woodward. A classification of

hyper-heuristic approaches. In M. Gendreau and J. Potvin, editors, *Handbook of Meta-Heuristics 2nd Edition*, pages 449–468. Springer, 2010.

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

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

[10] 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*, 4(2):100–107, February 1968.

[11] P. Haslum, B. Bonet, and H. Geffner. New admissible heuristics for domain-independent planning. In M. M. Veloso and S. Kambhampati, editors, *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*, pages 1163–1168. AAAI Press / The MIT Press, 2005.

[12] 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)*, pages 149–150, 2010.

[13] 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*, pages 955–962, New York, NY, USA, 2009. ACM.

[14] R. A. Hearn. *Games, Puzzles, and Computation*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2006.

[15] 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.

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

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

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

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

[20] 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*, pages 27–36, 1997.

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

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

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

[24] 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*, AAAI'97/IAAI'97, pages 700–705. AAAI Press, 1997.

[25] J. Levine and D. Humphreys. Learning action strategies for planning domains using genetic programming. In 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, editors, *EvoWorkshops*, volume 2611 of *Lecture Notes in Computer Science*, pages 684–695. Springer, 2003.

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

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

[28] J. Pearl. *Heuristics*. Addison–Wesley, Reading, Massachusetts, 1984.

[29] 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*, pages 324–332, 1989.

[30] A. Reinefeld and T. A. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.

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

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

[33] L. A. Taylor and R. E. Korf. Pruning duplicate nodes in depth-first search. In *Proceedings of the eleventh national conference on Artificial intelligence*, AAAI'93, pages 756–761. AAAI Press, 1993.

[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*, 179(1):369–392, 2010.

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