GP-Rush: Using Genetic Programming to Evolve Solvers for the Rush Hour Puzzle

Ami Hauptman Achiya Elyasaf Moshe Sipper Assaf Karmon
Dept. of Computer Science, Ben-Gurion University, Beer-Sheva 84105, Israel
{amihau,achiya.e,sipper,assafkar}@gmail.com

ABSTRACT

We evolve heuristics to guide IDA* search for the 6x6 and 8x8 versions of the Rush Hour puzzle, a PSPACE-Complete problem, for which no efficient solver has yet been reported. No effective heuristic functions are known for this domain, and—before applying any evolutionary thinking—we first devise several novel heuristic measures, which improve (non-evolutionary) search for some instances, but hinder search substantially for many other instances. We then turn to genetic programming (GP) and find that evolution proves immensely efficacious, managing to combine heuristics of such highly variable utility into composites that are nearly always beneficial, and far better than each separate component. GP is thus able to beat both the human player of the game and also the human designers of heuristics.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance, Design

Keywords

Genetic Programming, Heuristics, Rush-Hour Puzzle, Single-Agent Search

1. INTRODUCTION

Single-player games in the form of puzzles have received much attention from the Artificial Intelligence community for some time (e.g., [14, 30]). However, quite a few NP-Complete puzzles have remained relatively neglected by researchers (See [19] for a review).

Among these difficult games we find the Rush Hour puzzle,[1] which was proven to be PSPACE-Complete (i.e., more difficult than NP-Complete problems, if $NP \subset PSPACE$) for the general $n \times n$ case [10]. The commercial version of this popular single-player game is played on a 6x6 grid, simulating a parking lot replete with several cars and trucks. The goal is to find a sequence of legal vehicular moves that ultimately clears the way for the red target car, allowing it to exit the lot through a tile that marks the exit (see Figure 1). Vehicles—of length two or three tiles—are restricted to moving either vertically or horizontally (but not both), they cannot vault over other vehicles, and no two vehicles may occupy the same tile at once. The generalized version of the game is defined on an arbitrary grid size, though the 6x6 board is sufficiently challenging for humans (we are not aware of humans playing, let alone solving, complex boards larger than 6x6). (Note: throughout this paper we use the term 'car' when referring to a two-tile element, 'truck' when referring to a three-tile element, and 'vehicle' when referring to either.)

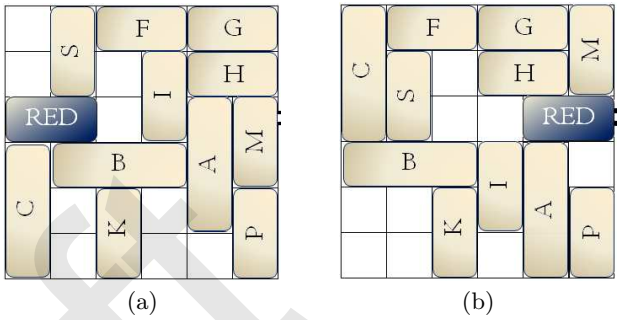

Figure 1: (a) A sample Rush Hour configuration. This is problem no. 9 of the problem set shipped with the standard version of the game by Binary Arts, Inc. In the paper we refer to this problem as $JAM09$. (b) A possible goal state: the red car has reached the exit tile on the right-hand side of the grid.

A major problem-solving approach within the field of artificial intelligence is that of heuristic search. One of the most important heuristic search algorithms is iterative-deepening A* (IDA*) [13, 20], which has several well-known enhancements, including transposition tables [11, 33], move-ordering [29], and pattern-databases [8]. This method is widely used to solve single-player games (e.g., [16, 22]). IDA* and similar algorithms are strongly based on the notion of approxi-

[1]The name "Rush Hour" is a trademark of Binary Arts, Inc.

GECCO'09, July 8–12, 2009, Montréal Québec, Canada.

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

By applying the heuristic function to states reachable from the current ones considered, it becomes possible to select more promising alternatives earlier 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).

To date, no efficient heuristics have been reported for the Rush Hour puzzle. We believe that the main reason for this is the lack of domain knowledge for the problem, which stems directly from the lack of research into this domain. Moreover, due to the specific structure of the Rush Hour puzzle, standard methods for deriving heuristics, such as solving either sub-problems (possibly with pattern databases [8]), or relaxed problems (e.g. using the Manhattan distance heuristic, augmented with linear conflicts [12]), which are typically easy to apply to other well-known domains, are not applicable here (see Section 3.3). For these reasons, using IDA* search, or similar algorithms, has probably not been possible.

In this work, we use genetic programming (GP) to *evolve* heuristic functions for the Rush Hour problem. We first construct a "brute-force", iterative-deepening search algorithm, along with several search enhancements—some culled from the literature, some of our own devise—but with no heuristic functions. As expected, this method works well on relatively simple boards, and even solves most moderately difficult ones within reasonable bounds of space and time. However, when dealing with complex problems, this method yields inadequate performance.

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

To accomplish this task, we use GP. Our main set of experiments focuses on evolving combinations of the basic heuristics devised. We use these basic heuristics as building blocks in a GP setting, where individuals are embodied as ordered sets of search-guiding rules (or *policies*), the parts of which are GP trees.

The effect on performance was profound: evolution proved immensely efficacious, managing to combine heuristics of such highly variable utility into composites that were nearly always beneficial, and far better than each separate component.

The contributions of this work are as follows:

- This is the first reported successful attempt to solve the Rush Hour puzzle using intelligent search.
- Along the way we devise several novel heuristics for this domain, some of which could be applied to other domains.
- We demonstrate how policies can be evolved to solve more difficult problems than ones previously attempted with this method.
- We show how difficult *solvable* puzzles can be generated, a task which is also considered hard (due to the fact that the decidability question, i.e., whether a given board is solvable or not, is also PSPACE-Complete).

2. PREVIOUS WORK

Little work has been done on the Rush Hour puzzle within the computer science community—work which we review herein, along with several related topics.

2.1 Rush Hour

Flake & Baum [10] examined a generalized version of Rush Hour, with arbitrary grid size and exit placements, proving that the question of whether an initial configuration is solvable is NP-Hard (the proof uses a reduction from the Satisfiability problem). They then showed the general problem's PSPACE-completeness, by emulating arbitrary recurrent circuits within generalized Rush Hour configurations. Hearn & Demaine [15] proved PSPACE-completeness of sliding block puzzles in the more general case, by demonstrating a reduction from the Quantified Boolean Formula Satisfiability problem.

These formal results imply that there is no polynomial-time algorithm able to find a solution for a given Rush Hour instance (unless $P = PSPACE$), and that the length of the shortest solution of a hard initial configuration grows exponentially with board size n (provided that $P \neq NP$ and $NP \neq PSPACE$). However, Fernau *et al.* [9] considered the parameterized complexity of the generalized version ($n \times n$) of the game, and showed that solutions can be found in polynomial time when either the total number of vehicles or the total number of moves is bounded by a constant.

Colette *et al.* [6] focused on finding hard initial configurations for 6x6 Rush Hour by modeling the game in propositional logic, and applying symbolic model-checking techniques to studying the graph of configurations underlying the game. They were able to classify all 3.6×10^{10} possible 6x6 configurations, according to the lengths of their shortest solution, within approximately 20 hours of computation time. Over 500 of the hardest configurations they found are available at `http://cs.ulb.ac.be/~fservais/rushhour/index.php`. On the downside, they proved a general theorem regarding the limitations of applying their method to board games, which stems from the fact that the underlying data structure grows exponentially with problem size.

None of the work above describes an efficient way to solve a given Rush Hour problem. The configurations database constructed by Colette *et al.* may be used for this purpose (although this is not what the authors intended [32]). However, we would need to query the database for the distance to solution of each board we encounter during the search. For difficult instances, this is highly inefficient.

2.2 Search Heuristics for Single-Player Games

Ample examples are found in the literature of hand-crafting heuristics to guide IDA* search in single-player games (hand-crafting—as opposed to using an automatic method, such as evolution, described in the next section). Korf [22] describes the use of pattern databases, which are pre-computed tables of the exact cost of solving various subproblems of an existing problem. This method was used to guide IDA* to solve Rubik's cube. Korf & Felner [23] dealt with disjoint pattern databases for the sliding-tiles puzzle problem. In a later work, Felner *et al.* [8] generalized this notion to include dynamic partitioning of the problem into disjoint subproblems for each state. They applied their method to three different problem domains: the sliding-tile puzzle, the 4-peg Towers of Hanoi problem, and finding an optimal vertex-cover of a graph.

Junghanns & Schaffer [18, 17] and later Botea *et al.* [5] dealt with one of the most complex single-player domains: the game of Sokoban. This is a transport puzzle in which the player pushes boxes around a maze and tries to put them in designated locations. The game is challenging due to several reasons, including: large and variable branching factors (potentially over 100); long solutions (some problems require over 500 moves); subgoals are interrelated and thus cannot be solved independently; heuristic estimators are complex; and, deadlocks exist—some moves render the problem unsolvable. Their IDA*-based program, *Rolling Stone*, equipped with several enhancements, including transposition tables, move ordering, deadlock tables, various macros, and pattern search, was able to solve 52 of the 90-problem standard test suite for Sokoban.

2.3 Evolving Heuristics for AI Planning

Some of the research on evolving heuristics for search is related to AI planning systems. However, heuristics are used to guide search in this field in a way highly similar to single-agent IDA* search, as we employ here.

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 [4]. Thus, their hybrid system (Hamlet-EvoCK) out-performed each of its sub-systems 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 far simpler than Rush Hour, mainly because they are less constrained.

Levine & Humphreys [25] 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 operators (for dealing with entire rules), as well as simple local search to augment GP crossover and mutation. They demonstrated some success in these two domains, although hand-coded policies sometimes outperformed the evolved ones.

3. METHOD

Our work on the Rush Hour puzzle developed through four main phases:

1. Construction of an iterative-deepening (uninformed) search engine, endowed with several enhancements, some of which were specifically tailored for this problem. Heuristics were not used during this phase.
2. Design of several novel heuristics, which were tested in conjunction with our engine.
3. Evolution of combinations of heuristics, along with conditions for applying them, using genetic programming.
4. Evolution of difficult 8x8 boards, using our engine augmented by heuristics to test board fitness.

First we briefly describe our test suite of problems.

3.1 Test Suite

The Rush Hour game (standard edition) is shipped along with 40 problems, grouped into four difficulty levels (Beginner, Intermediate, Advanced, and Expert). Minimal solution length (i.e., minimal number of moves) is an oft-used rough estimate of problem difficulty. We mark these problems $JAM01 \ldots JAM40$. Their solution lengths vary from 8 to 52 moves.

To add harder problems to the test suite we expanded it with the 200 most difficult configurations published online by Colette *et al.*, whose solution lengths vary from 53 to 93 moves. We mark these $SER1 \ldots SER200$. We are now in possession of an ample test suite, with problems of increasing difficulty—from simplest ($JAM1$) to hardest ($SER200$). The problem $SER200$ is the hardest 6x6 Rush Hour configuration, as reported in [6].

In order to work with more challenging problems, we evolved 15 difficult solvable 8x8 boards using the method described in Section 3.5. This resulted in much more difficult boards, which are marked $E1 \ldots E15$. Solution lengths for these problems vary from 90 to 120 moves.

3.2 Enhanced Iterative Deepening Search

We initially implemented standard Iterative Deepening search [20] 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), 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 (more precisely, near-optimal, given that the depth increase is usually larger than one), since a shorter solution would have been found in a previous iteration. However, for difficult problems, such as Rush Hour and Sokoban, finding *a* solution is sufficient, and there is typically no requirement of finding the optimal solution.

The game engine receives as input a Rush Hour board, as well as some run parameters, and outputs a solution (i.e., a list of moves) or a message indicating that the given instance could not be solved within the time or space constraints given. The idea of limiting the search derives from the work of Junghanns and Schaeffer [16] for the domain of Sokoban, where a limit of $20,000,000$ nodes was set for each problem. Since the Sokoban standard test-suite, which

they used, contains problems that typically require more resources than 6x6 Rush Hour problems, we used a stricter limit of $1,500,000$ nodes (since both depth and branching factor are lower for our problem, and hence search trees are smaller, this bound is reasonable).

The basic version of the game engine also included several simple search macros (sets of moves grouped together as a single move [21]) such as moving a vehicle several tiles (if applicable) as a single move, and always moving the red car toward the exit as a single move when possible.

Transposition tables, which afford the ability to identify redundant sub-trees [11, 33], were an important enhancement to our basic engine. The common use for such tables is to avoid visiting boards which were visited previously (thus escaping loops, since all move operators are reversible by definition), using a hash table to store all boards already encountered. However, since we are using iterative deepening, it is sometimes possible to revisit a node, but along a shorter path from the initial state. In this case, we keep the node, and update its distance. This idea allows us to find (near) optimal solutions, which is beyond the accepted requirement for difficult problems such as Rush Hour. Thus, for each solution we can count the number of nodes expanded, as well as the number of transpositions (which is precisely the size of the table).

Using search alone, along with the enhancements described above, we were able to solve all boards of the first problem set, $JAM01 \ldots JAM40$, expanding less than 500,000 nodes. However, 20 problems from the group $SER150 \ldots SER200$ still took over $1,500,000$ nodes to solve, which violated our space bound.

We concluded that uninformed search, even when augmented by several enhancements, is not powerful enough to solve difficult instances of this problem. Thus, it was clear that heuristics were needed.

3.3 Heuristics for Rush Hour

In this section we describe some of the heuristics we devised, all of which are used to estimate the distance to the goal from a given board.

We encountered difficulties when attempting to implement standard methods for devising heuristics, mainly in the form of problem relaxation [27] (e.g., with pattern databases [7, 8], and more specifically, the Manhattan-Distance heuristic [12]). This methodology is difficult to apply to the Rush Hour puzzle due to the structure of the domain—every vehicle can potentially have a substantial effect over the problem as a whole. Alleviating the constraints imposed even by a single vehicle (e.g., by removing it or allowing it to move freely) in order to obtain a heuristic value for the above-mentioned methods may render a difficult problem easy (for example, if we remove vehicles M and I in problem $JAM09$ of Figure 1, the problem can be solved in a mere two steps). These ideas can, however, be refined into useful heuristics, which we describe below.

Additionally, when we use heuristics to guide search, we move from simple iterative deepening, to iterative deepening A* (or IDA*) [20]. This algorithm operates similarly, except for using the heuristic value to guide the search at each node (this method is known as move-ordering [29]).

3.3.1 Blockers Estimation

The first obvious estimate to the closeness of a board configuration to the goal is the number of vehicles blocking the red car's path to the exit, because when this number reaches zero, the problem is solved. However, simply counting the number of such vehicles is not very informative (e.g., for several difficult problems only one vehicle blocks the path in the initial configuration, yet still the distance to the solution is large).

A better measure is had by computing a lower-bound estimate of the number of moves required to move each vehicle out of the red car's path.[2] This entails estimating the number of moves needed to move each vehicle blocking these vehicles, and so on, recursively. The numbers are then added up, with some redundancy checks to avoid counting the same vehicle more than once. When we have to choose between two possible directions of moving a vehicle out of the way, we compute both and retain the minimal value.

This heuristic, which we called $BlockersLowerBound$, reduced the number of nodes for several difficult problems by 70% when tested empirically, although for some problems it actually *increased* the node count by more than 170%. This latter was probably because some parts of the solutions required moves that increased the blockers' estimate, and this heuristic guided the search away from them. What was missing was a measure of *when* to apply the heuristic. Moreover, errors are expected due to the fact that no estimator for a difficult problem can be perfect. Indeed, the ambivalent nature of this heuristic—often helpful, at times detrimental—is also true of the other heuristics introduced.

3.3.2 Goal Distance

The following heuristic, called $GoalDistance$, is a possible way to implement the Manhattan-Distance heuristic, as used for the sliding-tiles puzzle (e.g., [23].) To devise such a measure, we need to count each vehicle's distance from its designated place in the goal board. However, compared to the sliding tiles or Rubik's cube, the final position for each vehicle is not known in advance.

In order to solve this problem, we construct, for each initial configuration, a *deduced* goal: a board containing a clear path to the goal, where all interfering vehicles (and vehicles blocking them) have been "forcibly" positioned (i.e., ignoring move rules while still forbidding two vehicles from occupying the same tile) in possible locations in which they are no longer blocking the red car. If necessary, we also move the cars blocking their paths in the same manner. Devising a good heuristic function for deducing goal boards was not easy, as it required some complex reasoning for several cases. Moreover, there is no guarantee, especially for difficult problems, that the deduced goal board will actually be the correct goal board. However, this heuristic proved to be a useful building block for high-fitness individuals.

3.3.3 Hybrid Blockers Distance

Here we combine the essence of the previous two heuristics. Instead of merely summing up each vehicle's distance to its location in the deduced goal, we also count the number of vehicles in its path, and add it to the sum. This heuristic was dubbed $Hybrid$. Note that we do not perform a full blocker's estimation for each vehicle (only the number of

[2]Heuristic functions that never overestimate solution lengths (known as *admissible* heuristics) have been theoretically proven to guide single-agent search better than non-admissible heuristics (for example, see [31]).

blockers is summed)—this is needed since computing a more detailed measure would be both time consuming and would sometimes produce larger estimates than required (since the same vehicle may block several other vehicles, and it would be counted as a blocker for each of them).

3.3.4 Other Heuristics

Additional functions were used to assign scores to boards, including:

- $MoveFreed$: Checks if the last move made increases the number of vehicles free to move.
- $IsMoveToSecluded$: Did the last move place a car in a position to which no other car can move?
- $ProblemDifficulty$: The given difficulty level of the problem at hand (this information is also available to humans when solving the problems shipped with the game).

For a complete list of heuristics, see Table 1.

3.4 Evolving Heuristics

Using the heuristics we devised to make search more efficient is a difficult task, as it involves solving two major sub-problems:

1. Finding exact conditions regarding *when* to apply each heuristic (in order to avoid the strong inconsistent effect on performance mentioned above).
2. Combining several estimates to get a more accurate one. We hypothesized that different areas of the search space might benefit from the application of different heuristics.

Solving the above sub-problems means traversing an extremely large search space of possible conditions and combinations. This is precisely where we turn to *evolution*.

3.4.1 The Genome

As we want to embody both application conditions and combinations of estimates, we decided to evolve ordered sets of control rules, or *policies*. As stated above, policies have been evolved successfully with GP to solve search problems—albeit simpler ones (for example, see [3] and [4], mentioned above).

Policies typically have the following structure:[3]

$RULE_1$: 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 the aforementioned conditions and estimates, respectively.

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

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

The function set includes the functions $\{AND,OR,\leq,\geq\}$ for condition trees and the functions $\{\times,+\}$ for the value trees. All heuristics described above are used as terminals. For a complete list, see Table 1. To get a more uniform calculation, we normalize the values returned by terminals of $Condition$ trees to lie within the range $[0,1]$, by maintaining a maximal possible value for each terminal, and dividing by it. For example, $BlockersLowerBound$ might return an estimate of 20 moves, with the maximal value for this terminal determined empirically to be 40, thus setting the return value to 0.5.

3.4.2 Genetic Operators

We used the standard crossover and mutation operators, as detailed in [24]. However, before selecting the crossover or mutation point, we first randomly selected rules whose conditions (or values) were to be substituted. Crossover was only performed between nodes of the same type (using Strongly Typed Genetic Programming [26]).

We also added $RuleCrossover$ and $RuleMutation$ operators, whose purpose was to swap entire randomly selected rules, between individuals and within the same individual, respectively. One of the major advantages of policies is that they facilitate the use of more diverse genetic operators, such as $RuleCrossover$ and $RuleMutation$.

3.4.3 Test and Training Sets

Individuals were evolved with fixed groups of problems (one group per run): The suite of all 6x6 problems ($JAM01$ through $SER200$) was divided into 5 equally sized groups (48 problems per group). Additionally, we used a sixth group containing 15 difficult 8x8 problems, discovered through evolution (see Section 3.5).

For each group, 10 problems (taken from the 20 most difficult ones) were tagged as *test* problems, and the remaining ones were used as *training* problems. Training problems were used for fitness purposes (see below), while test problems were used to test the best individual, in order to assess the overall progress of the run.

3.4.4 Fitness

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

[3]Actually, policies are commonly defined as rules where the result is an *action*, not a value. However, actions lead to the selection of a child node, and are thus effectively similar to heuristic values.

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

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

3.4.5 GP Parameters

We experimented with several configurations, finally setting upon: population size – between 50 and 100, generation count – between 100 and 400, reproduction probability – 0.5, crossover probability – 0.4 , and mutation probability – 0.1. For both the crossover and mutation operators, we used a uniform distribution for selecting trees inside individuals.

3.5 Evolving Difficult Solvable 8x8 Boards

Since our enhanced IDA* search solved over 90% of the 6x6 problems (including 30% of the 50 most difficult problems reported in [6]), well within the space bounds (in fact, with far fewer requirements), and, moreover, we wanted to demonstrate our method's scalability to larger boards, we needed to design more challenging problems. This we did through evolution.

We generated the initial population by taking solvable 6x6 boards and expanding each one to size 8x8 by "wrapping" it with a perimeter of empty cells (i.e., each 6x6 board was embedded in the center of an empty 8x8 board). Then, using simple mutation operators, which randomly either add, swap, or delete vehicles, we assigned to each board a fitness score equal to the number of boards required to solve it using our enhanced IDA* search. A board that could not be solved within 15 minutes (on a Linux-based PC, with processor speed 3GHz, and 2GB of main memory) received a fitness score of 0.

We repeated this process until evolution showed no further improvement. While this mutation-based process might generate genotypically similar boards, they are phenotypically different due to the domain structure, described above. The most difficult 8x8 board found required 26,000,000 nodes to solve with no-heuristic, iterative deepening (the None column in Table 2).

Table 2: Average percentage of nodes required to solve *test* problems, with respect to the number of nodes scanned by iterative deepening (shown as 100% in the second column). $H1$: the heuristic function *BlockersLowerBound*; $H2$: *GoalDistance*; $H3$: *Hybrid*. Hc is our hand-crafted policy, and GP is the best evolved policy, selected according to performance on the training set. 6x6 represents the test cases taken from the set $\{JAM01 \ldots SER200\}$. 8x8 represents the 15 most difficult 8x8 problems we evolved. Numbers are rounded to nearest integer.

Heuristic: Problem	None	$H1$	$H2$	$H3$	Hc	GP
6x6	100%	72%	94%	102%	70%	40%
8x8	100%	69%	75%	70%	50%	10%

4. RESULTS

We assess the performance of heuristics with the same scoring method used for fitness computation, except we average over the entire test set instead of boards taken from the training set.

We compare several heuristics: the three hand-crafted heuristics described in Section 3.3; a hand-crafted policy that we designed ourselves, by combining the basic (hand-crafted) heuristics; and the top full-fledged policy developed via GP, which we took from the best run.

Results are summarized in Table 2. As can be seen, the average performance of our hand-crafted heuristics did not show significant improvement over iterative deepening with no heuristic (although *BlockersLowerBound* proved better than the other two). While our hand-crafted policy fared somewhat better, the evolved policy yielded the best results by a wide margin, especially given the increasing difficulty of node reduction as search gets better. **Overall, evolved policies managed to cut the amount of search required to 40% for 6x6 boards and to 10% for 8x8 boards, compared to iterative deepening.**

It should also be noted that performance over 8x8 boards was better relative to 6x6 boards. This may be ascribed to the fact that while the entire space of difficult 6x6 boards is covered by our test and training sets, this is not the case for our 8x8 boards. Still, considering that the evolved 8x8 boards we used proved immensely difficult for no-heuristic iterative deepening (requiring over 20,000,000 nodes to solve in some cases) **results show that our method is scalable, which is non-trivial for a PSPACE-Complete problem.**

Next, we turn to comparing the performance of evolution to that of humans. Since we have no exact measure for the number of boards examined by humans for this problem, we turn to another measure: solution time. All comparisons performed so far treated only the number of nodes expanded, due to the fact that the amount of time required to solve a problem is linearly related to the number of nodes (i.e., less nodes implies less time). This is obvious since the engine's speed (or nodes per second) is constant. The time data was collected along with the number of nodes for all our runs.

Data regarding human performance is available online at `http://trafficjamgame.com/`, in the form of High Scores (sorted by time to solution) for each of the problems $JAM01$ to $JAM40$. This site contains thousands of entries for each problem, so the data is reliable, although it doesn't neces-

Table 3: Time (in seconds) required to solve problems $JAM01 \ldots JAM40$ by: ID – iterative deepening, Hi – average of our three hand-crafted heuristics, Hc – our hand-crafted policy, GP – our best evolved policy, and human players (average of top 5). Problems are divided into 5 groups, and the average is presented below.

Problems	ID	Hi	Hc	GP	Humans
$JAM01 \ldots JAM08$	0.2	0.65	0.06	0.03	2.6
$JAM09 \ldots JAM16$	1.7	0.35	1.74	0.6	8.15
$JAM17 \ldots JAM24$	2.4	1.8	1.08	0.83	10.32
$JAM25 \ldots JAM32$	6.3	1.6	3.94	1.17	14.1
$JAM33 \ldots JAM40$	7.65	2.8	7.71	2.56	20.00
Average	3.65	1.44	2.69	1.04	11.03

sarily reflect the *best* human performance. We compared the time required to solve these 40 problems by humans to the runtime of several algorithms: iterative deepening, Hi (representing the average time of our three hand-crafted heuristics), our hand-crafted policy, and our best evolved policy. Results are presented in Table 3. Clearly, all algorithms tested are much faster than human players, and evolved policies are the fastest.

This emphasizes the fact that evolved policies save *both* search time *and* space.

5. CONCLUSIONS

We designed an IDA*-based solver for the Rush Hour domain, a problem to which intelligent search has not been applied to date. With no heuristics we managed to solve most 6x6 problems within reasonable time and space limits, but only a few of our newly evolved, difficult 8x8 problems. After designing several novel heuristics for the Rush Hour domain, we discovered that their effect on search was limited, and somewhat inconsistent, at times reducing node count by 70%, but in several cases actually *increasing* the node count to over 170% for many configurations. Solving the problem of correctly applying our heuristics was done by evolving policies with GP (which outperformed a less successful attempt to devise policies by hand). To push the limit yet further, we evolved difficult 8x8 boards, which aided in the training of board-solving individuals by augmenting the fitness function.

Our results show that the improvement attained with heuristics increased substantially when evolution entered into play: search with evolved policies required less than 50% of the nodes required by search with non-evolved heuristics. As a result, 85% of the problems, which were unsolvable before, became solvable within the 1,500,000 node limit, including several difficult 8x8 instances.

There are several conceivable extensions to our work, including:

1. We are confident that better heuristics for Rush Hour remain to be discovered. For example, it is possible to take the ideas underlying the *GoalDistance* heuristic, and apply them to deducing more configurations along the path to the goal (and calculating distances to them, as we did with *GoalDistance*). While this calculation requires more preprocessing, we are certain that it will yield a more efficient algorithm, since we'd be providing search with a more detailed map to the goal.

2. Hand-crafted heuristics may themselves be improved by evolution. This could be done by breaking them into their elemental pieces, and evolving their combinations thereof. For example, the values we add when computing $BlockerLowerBound$ might be real numbers, not integers, whose values evolve, subject to more domain knowledge. It is both possible to evolve a given heuristic as the only one used in IDA* search, or to evolve it as part of a larger structure of heuristics, itself subject to (piecewise) evolution. Totally new heuristics may also be evolved using parts comprising several known heuristics (just like the $Hybrid$ heuristic was conceptualized as a combination of $BlockersLowerBound$ and $GoalDistance$).

3. As our search keeps improving, and we use it to find more difficult solvable configurations, which, in turn aid in evolving search, we feel that the limits of our method (i.e., solving the most difficult boards possible within the given bounds) have not yet been reached. As we are dealing with a PSPACE-Complete problem, it is certain that if we take large-enough boards, solving them would become infeasible. However, for the time being we plan to continue discovering the most challenging configurations attainable.

4. Many single-agent search problems fall within the framework of AI-planning problems, and Rush Hour is no exception. Algorithms for generating and maintaining agendas, policies, interfering sub-goals, relaxed problems, and other methodologies mentioned above are readily available, provided we encode Rush Hour as a planning domain (e.g., with ADL [28]). However, using evolution in conjuncture with these techniques is not trivial.

Rush Hour is a fun game, which challenges the mind both of player and of AI practitioner alike. Encouraged by our positive results reported herein, we are vigorously pursuing the avenues of future research described above, and hope to be able to report upon more exciting results in the future.

Acknowledgments

Ami Hauptman is partially supported by the Lynn and William Frankel Center for Computer Sciences.

6. REFERENCES

[1] R. Aler, D. Borrajo, and P. Isasi. Evolving heuristics for planning. *Lecture Notes in Computer Science*, 1447:745–754, 1998.

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

[5] A. Botea, M. Muller, and J. Schaeffer. Using abstraction for planning in Sokoban. In *CG: International Conference on Computers and Games*. LNCS, 2003.

[6] S. Collette, J.-F. Raskin, and F. Servais. On the symbolic computation of the hardest configurations of the Rush Hour game. In *Proc. of the 5^{th} International Conference on Computers and Games*, LNCS 4630, pages 220–233. Springer-Verlag, 2006.

[7] J. C. Culberson and J. Schaeffer. Searching with pattern databases. In G. McCalla, editor, *Proceedings of the Eleventh Biennial Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, volume 1081 of *LNAI*, pages 402–416, Berlin, May 21–24 1996. Springer.

[8] A. Felner, R. E. Korf, and S. Hanan. Additive pattern database heuristics. *J. Artif. Intell. Res. (JAIR)*, 22:279–318, 2004.

[9] H. Fernau, T. Hagerup, N. Nishimura, P. Ragde, and K. Reinhardt. On the parameterized complexity of the generalized Rush Hour puzzle. In *Canadian Conference on Computational Geometry*, pages 6–9, 2003.

[10] G. W. Flake and E. B. Baum. Rush Hour is pspace-complete, or "why you should generously tip parking lot attendant". *Theor. Comput. Sci.*, 270(1-2):895–911, 2002.

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

[12] O. Hansson, A. Mayer, and M. Yung. Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, 63(3):207–227, Sept. 1992.

[13] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Trans. on SSC*, 4:100, 1968.

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

[15] R. A. Hearn and E. D. Demaine. Pspace-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1-2):72–96, 2005.

[16] A. Junghanns and J. Schaeffer. Sokoban: A challenging single-agent search problem. In *IJCAI*, pages 27–36. Universiteit, 1997.

[17] A. Junghanns and J. Schaeffer. Domain-dependent single-agent search enhancements. In T. Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pages 570–577. Morgan Kaufmann, 1999.

[18] A. Junghanns and J. Schaeffer. Sokoban: Improving the search with relevance cuts. *TCS: Theoretical Computer Science*, 252, 2001.

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

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

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

[22] R. E. Korf. Finding optimal solutions to rubik's cube using pattern databases. In *AAAI/IAAI*, pages 700–705, 1997.

[23] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *AIJ: Artificial Intelligence*, 134, 2002.

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

[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] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

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

[28] E. P. D. Pednault. Adl: exploring the middle ground between strips and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pages 324–332, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

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

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

[31] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs , NJ, 1995.

[32] F. Servais. Private communication.

[33] L. A. Taylor and R. E. Korf. Pruning duplicate nodes in depth-first search. In *AAAI*, pages 756–761, 1993.