Solving the 8-puzzle: A Genetic Programming Approach

Jason M. Barnes, Shaddi H. Hasan, and Sanghwi Lee

***Abstract*— This paper illustrates the application of a genetic programming approach to solving the 8-puzzle, also known as the sliding block puzzle. The 8-puzzle is a 'game problem', useful for understanding concepts of machine learning in a well-defined environment. The system we have designed is broken into three functions: *GP-generate*, *solve-8puzzle*, and *test-8puzzle*. *GP-generate* uses genetic programming techniques to develop a parse tree to solve a generic 8-puzzle, while *solve-8puzzle* applies a specific parse tree generated by *GP-generate* to a specific 8-puzzle. *test-8puzzle* tests the solution generated by *solve-8puzzle* to determine if the solution is correct. We based our project on research done by Finkelstein and Markovitch on the 8-puzzle problem, specifically, our fitness function and our terminal set.**

Our parse trees were only able to solve the test cases to a specific point, depending on the function set used and the individual test cases. Our research determined that our learning agent was only able to partially solve the test cases. After modifications to our selection process which allowed all branches of the parse tree to be chosen for crossover with equal probabilities, we saw a dramatic decrease in the value of the heuristic. Further investigation would be necessary to reveal why the parse trees only reduced the value of the heuristic to this number rather than 0.

I. Introduction

THE objective of this project was to develop a program to solve the 8-puzzle (also known as the sliding block puzzle) using genetic programming. Our solution was to be broken into three programs, *GP-generate*, *solve-8puzzle*, and *test-8puzzle*. In theory, *GP-generate* would develop a parse tree using genetic programming that could solve the 8-puzzle in the general case. *solve-8puzzle* would then apply the parse tree found by *GP-generate* to a specific input puzzle, yielding a string of moves. Finally, *test-8puzzle* would apply the string of moves to the puzzle specified by *solve-8puzzle* and then determine if the puzzle were indeed solved. Thus, the bulk of our work focused on *GP-generate*, with *solve-8puzzle* and *test-8puzzle* both being relatively straightforward programs that simply interpreted the parse tree generated by *GP-generate*.

We relied heavily on research conducted by Finkelstein and Markovitch on the 8-puzzle problem. In their paper, they defined a heuristic that could be used to evaluate the degree to which a puzzle is solved. The heuristic is as follows:

$$h(s) = 36 \cdot \text{Misplaced} + 18 \cdot \text{ManhattanOfFirstMisplacedTile} \cdot \text{ManhattanOfFirstFromBlank} \quad (1)$$

The value of *h(s)* is zero for a completely solved puzzle. We determined the maximum value of this heuristic to be 402, corresponding to the theoretical 'least solved' puzzle. This puzzle would have no tiles correctly placed, the blank space in the top left corner, and the first tile in the bottom right corner (this assumes the first tile should be in the top left corner).

We used the above heuristic as our fitness function. We tested each parse tree on ten example puzzles used as test cases and then calculated the average value of the heuristic after executing the parse tree over the ten puzzles. This average value became the fitness of each parse tree. In this approach, lower fitness meant a better solution. In a similar vein, we set the fitness threshold to be zero—our goal was solving the 8-puzzle completely.

As with all genetic programming problems, we defined a number of parameters for *GP-generate*. For our terminal set, we used movements that corresponded to changing the position of the blank space in the 8-puzzle. Figure 1 illustrates this concept. The movements we used for the terminal nodes were up, down, left, and right, which we abbreviated as *u, d, l,* and *r*. In addition to these nodes, we also used a set of combinations of movements (macros) provided by Dr. Parker and based off of work done by Finkelstein and Markovitch. Thus, our terminal set consisted of the following nodes:

$$\begin{aligned}\text{Terminal set} = \{&\text{u, d, l, r, lur, rul, uld, ruuld,}\\ &\text{dllur, drrul, urrdluld, uuldrdluurd,}\\ &\text{lurrdluld, urdrulldrur, urdrulldrurd,}\\ &\text{llurdrulldrurd, uldlurdruldrurd}\}\end{aligned} \quad (2)$$

For our function set, we originally designed only one function, IF-THEN-ELSE. This function had two leaves, A and B, which of course could be any member of the terminal set or another IF function. The IF function would compare the predicted effect of leaf A on the heuristic function, *h(A)*, to the predicted effect of leaf B on the heuristic function, *h(B)*. If the value of *h(A)* were found to be less than *h(B)*, the function would return the string A up the parse tree. Otherwise, the

Manuscript submitted April 5, 2006

Jason M. Barnes is currently at the University of Tennessee, Knoxville, and can be reached at marlin336c@gmail.com.

Shaddi H. Hasan is currently at Bearden High School, and can be reached at shaddi.hasan@gmail.com

Rosh Lee is currently at University of Tennessee, Knoxville, and can be reached at slee@cs.utk.edu.

string B would be returned. This ensures that the parse tree as a whole would return a string corresponding to a move.

Also in our original design, the parse trees were part of a loop. That is, after every movement, the puzzle would be re-evaluated and the parse tree would execute again, making only one move per iteration.

We did make two changes to this design during experimentation, which will be described in more detail below. In short, we added the function AND to our function set, which merely concatenated leaves A and B. Also, we removed the looping aspect from our parse tree, instead allowing it to make multiple moves during each execution of the parse tree. Once again, the reasons for these changes and their effects will be described in our Experiments section.

To generate the initial population, we randomly picked from the set of terminals and functions until we had met the population size we defined. The functions AND and IF each had a likelihood of 2/13 of being chosen randomly for any given node in the initial generation of the population. All members of the terminal set were chosen with probability 1/26. However, the first node was forced to be an IF function to ensure each tree actually did something productive. The maximum depth of the initial population was originally set at 10, while the maximum depth for the successive populations was set at 30.

We also set several genetic parameters. Our initial population size was 500, and we applied crossover to 30% of our population each generation. We felt these numbers would give us a sufficiently large search space for the problem, since these numbers had worked in some of the case studies we had looked at. We did not use mutation, since crossing over two leaf nodes would be functionally equivalent. We selected members of the population for crossover using tournament selection to ensure diversity in our population. The branch of the parse tree that was selected for crossover was selected by starting at the root, and then choosing a random number. Based on this number, the program would turn left or right down the tree, or stop. The stopping point was where crossover would occur.

II. Experiments and Results

A. Initial Setup

Using the code provided in *test-8puzzle* for generating solvable 8-puzzles, we created 10 test puzzles to test our *GP-generate* program. The program executed for 51 generations, keeping track of each generation's best fitness, average fitness, complexity of the member with the best fitness, average complexity, and uniqueness.

Unfortunately, the best fit member of the population was unable to successfully solve the 8-puzzle. Our program would solve the various test cases to a point, and then stop. Using our original design, we were unable to reduce our best fitness below 353.900. While this was consistently the best fitness our program could generate, the average fitness of the population converged to this value as the generations continued. In fact, all the metrics we used to evaluate the evolution of our parse trees continued to fluctuate, with complexity remaining essentially constant before increasing, and with population uniqueness falling throughout.

This result implied that the parse trees generated by *GP-generate* were encountering a situation in which no member of the terminal set could reduce the heuristic, which was a basic assumption of our design. Our classroom peers corroborated our findings, so we modified our function set to account for this fact.

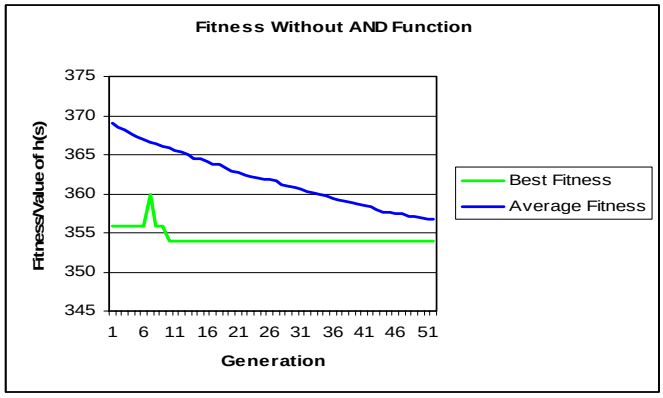

Fig. 1. Initial results using the original design for *GP-generate*. Note the relatively constant value of the best fitness; after only one significant fluctuation, the fitness reaches its minimum.

B. Addition of AND Function

Based on Finkelstein and Markovitch's paper, we believed that some set of movements existed that would always decrease the value of the heuristic. Thus, we added the AND function to our function set to give our parse trees the ability to concatenate strings of movements. The idea was that in addition to learning how to solve the 8-puzzle, our program would also learn the set of moves necessary to reduce the value of the heuristic given any board state. Also, we hoped that this addition would allow the parse trees to go through states that, while temporarily increasing the value of the heuristic, would eventually decrease the value of the heuristic. After adding this function, our results (illustrated by figure 2) remained essentially unchanged. In fact, the performance of the system actually seemed to decrease. Whereas the best fitness generated by *GP-generate* without the AND function continuously decreased before reaching the minimum of 353.9, the best fitness after adding the AND function exhibited some erratic fluctuation before converging to 353.9. We believe this is due to the fact that the parse trees concatenated movement strings to create new strings that were not conducive to solving the puzzle. That is, the ideal movement sequences were given in the Finkelstein and Markovitch paper, and any superfluous additions to them decreased their utility. Further research into the theory of 8-puzzles, specifically from the macro approach used in Finkelstein and Markovitch, would be needed to resolve these issues.

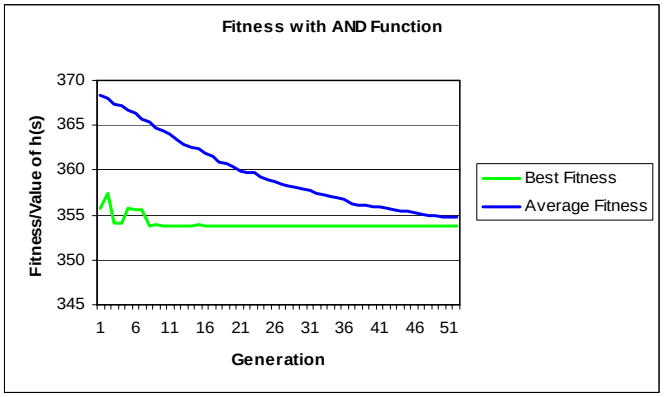

Fig. 1. Results after the addition of the AND function to the function set. Not only did the addition of the AND function not decrease the fitness of the population, it produced more erratic behavior for the value of the best fitness early in learning.

C. *Removing the Loop*

Trying a different approach, we removed the loop aspect of our parse trees. Our original design output only one move per execution of the parse tree. After removing the loop, we modified the IF function so that each IF function would contain a DO function, allowing the IF function to directly modify the puzzle. Unfortunately, this did not cause any noticeable change in our results. This is likely due to the fact that both parse trees, with or without the loop, were functionally equivalent: both approaches eventually would produce trees that acted upon the puzzles in the same way.

The fact that all these approaches brought the value of the heuristic to a single number implied that the parse trees were only able to solve the test cases to a certain point. This was our impetus for adding the AND function to the function set, hoping that function would allow *GP-generate* to generate trees that could overcome temporary decreases in fitness. After modifying the function set and main loop proved futile, we tried changing the test cases we used. Immediately, we found that the parse trees reached a new minimum when tested on the new test cases. This proved that the best fitness generated by *GP-generate* was dependent on the test cases used, and thus demonstrated that the test cases were only being solved to a point.

D. *Modifying the Crossover Selection Process*

Finally, we re-evaluated the way we selected branches of the parse trees for crossover. Analysis of our selection procedure for crossover revealed that higher-level elements of parse trees were more likely to be selected for crossover than lower-level ones. This would be logical, considering that it is in the lower levels of the trees that most of the decision-making power (and thus ability to solve the puzzles) resides. Thus, we chose a new selection procedure, henceforth known as the fair selection method. We would first choose a random level in Parent A, then choose a random node within that level. Next, we would choose a random level on Parent B that was at an equal or lower level than the one chosen on Parent A. Once again, a random node would be chosen on this level from Parent B, and the two selected nodes would be swapped. This would prevent the size of the parse tree from exceeding our limit. After modifying our selection procedure to allow crossover at any level of the parse trees with equal probability, we found that the average heuristic of the population dropped drastically, as illustrated in Figure 3.

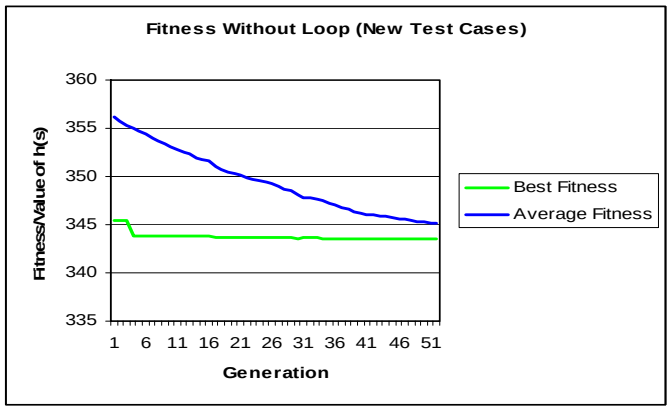

Fig. 2. Results after removal of the loop aspect of the parse tress and the addition of new test cases. While removal of the loop did produced no noticeable changes in either the best or average fitness, the addition of new test cases allowed the best fitness to reach a new minimum.

We also noticed a rise in both best and average complexity due to more complex branches being crossed over, which was in line with our expectations. While the tree was still unable to solve the test cases, the best parse tree did progress much farther than any other previous one in solving the puzzles.

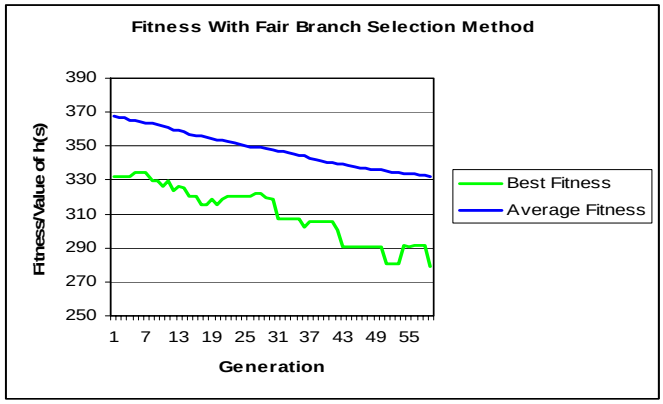

Fig. 4. Results after implementing a fair branch selection method. The value of the heuristic demonstrated a drastic decrease, more than 60. However, rate at which the average fitness decreased did not keep up with the rate of decrease of the best fitness value.

These results exhibit an interesting characteristic. With the fair selection method in place, the best fitness value decreased relatively rapidly, but the average fitness decreased at a much slower pace.

E. *Uniqueness and Complexity*

Comparing the population uniqueness of the various approaches to the learning problem, we noticed an interesting phenomenon. Every single approach saw a decrease in population uniqueness early in the learning. The lower the minimum value for uniqueness, the later the uniqueness reached that minimum, as illustrated in Figure 5. The best uniqueness was, of course, from the population that used the

fair branch selection method, since this approach allowed for the most variety in crossover outcomes.

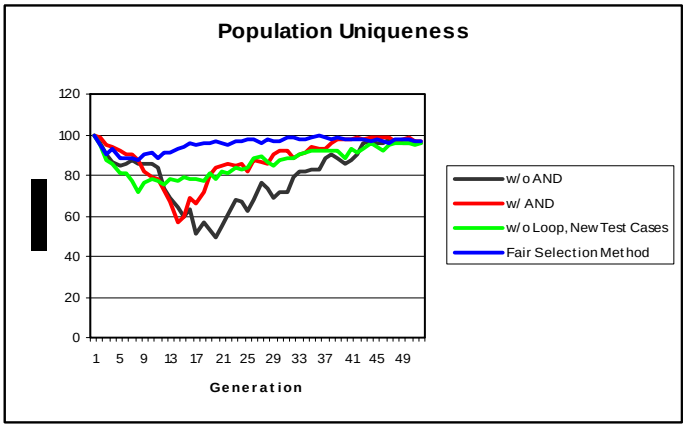

Fig. 5. Population uniqueness for the various approaches to the learning problem. Low uniqueness, combined with an average fitness that is close to the best fitness, is an indication that the best fit members are taking over the population.

The removal of the loop also seemed to have a marked impact on the uniqueness. This is probably due to the fact with only one iteration, a more complex tree would be needed to solve a given puzzle. Thus, the fitness function (i.e., how far the parse tree solved the test puzzles) would reward a more complex solution.

Regarding complexity, we noticed a general trend toward increased complexity for all our approaches to the problem.

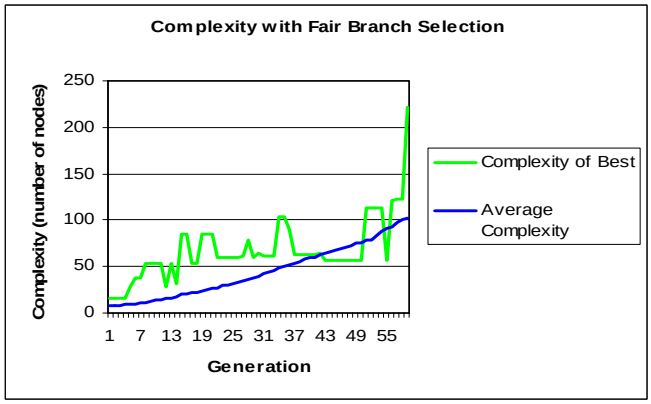

Fig. 6. Complexity of the best fit individual vs. the average complexity of the population using the fair branch selection method during crossover. The trends seen here were identical to those seen in the rest of the data.

Figure 6 illustrates the complexity of the best solution versus the average complexity of the population. The general trend shown here, with steadily increasing average complexity and erratically increasing complexity of the best member, was seen from all our other data. The only notable difference is that the complexity of the best member tended to remain constant for longer periods of time in some of the other approaches. For example, the population that used the AND function, the loop, and the old test cases reported back a best fit member with fitness 353.8 and complexity 63 nine

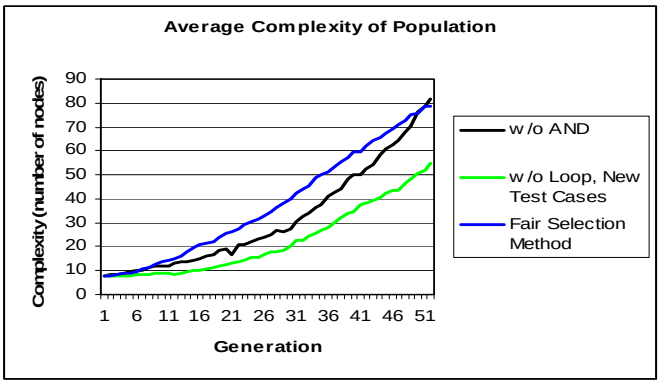

Fig. 7. The average complexity of the population as a function of the generation. No complexity data was available for the population generated with both AND in the function set and the loop.

separate times. As all of these instances were toward the end of learning, this is in line with the relatively low uniqueness of that population which would tend toward an abundance of the same, highly fit members.

III. Conclusions

The ability of *GP-generate* to create parse trees that are able to solve puzzles is most dependent on two factors: the function set, and the method by which branches are selected for crossover. Changes to the terminal set, particularly after the addition of the AND function, had little or no bearing on the performance of our parse trees, as the trees were able to concatenate strings of movements together on their own simply from the up, down, left, and right movements.

Even after changing the way branches were chosen for crossover, our system still did not solve the puzzle. Since the heuristic was decreasing even up to the last recorded generation, we have reason to believe that *GP-generate* would have eventually discovered the optimal solution. The issue, then, would be learning speed. While more testing would be necessary, it is possible that different, better, macros could assist the learning speed. By letting the designer combine the movements beforehand instead of relying on the learning agent to determine the best set of macros, the agent would be able to 'focus' its learning on solving the 8-puzzles. That is, the agent would learn faster since it would only have one problem to learn instead of the two it currently has.

Acknowledgment

We all wish to thank Dr. Parker for providing the results of Finkelstein and Markovitch's work in a concise, condensed form. Shaddi Hasan would like to thank Elinor Benami for her assistance in proofreading this paper.