A Puzzle to Challenge Genetic Programming

Edmund Burke, Steven Gustafson[†], and Graham Kendall

ASAP Research, School of Computer Science & IT
University of Nottingham, UK
{ ekb | smg | gxk }@cs.nott.ac.uk
[†] corresponding author

Abstract. This report represents an initial investigation into the use of genetic programming to solve the N-prisoners puzzle. The puzzle has generated a certain level of interest among the mathematical community. We believe that this puzzle presents a significant challenge to the field of evolutionary computation and to genetic programming in particular. The overall aim is to generate a solution that encodes complex decision making. Our initial results demonstrate that genetic programming can evolve good solutions. We compare these results to engineered solutions and discuss some of the implications. One of the consequences of this study is that it has highlighted a number of research issues and directions and challenges for the evolutionary computation community. We conclude the article by presenting some of these directions which range over several areas of evolutionary computation, including multi-objective fitness, coevolution and cooperation, and problem representations.

1 Introduction

Several challenging problems are used within the genetic programming literature to develop and test methods and theories. Punch et al [1] used the Royal Tree benchmark problem to test the ability of multiple populations. Langdon [2] employed the balanced bracket problem, the Dyck language and Reverse Polish expressions to compare the necessity of advanced data structures versus indexed memory. Koza [3] used the multiplexer and parity functions and protein sequence classification [4] as difficult problems for genetic programming. Daida et al [5] investigated the binomial-3 problem to study *tunably difficult* problems. Genetic programming also was developed for the robotic soccer problem in [6][7][8], using an array of novel methods. A further example is provided by Soule et al [9] who worked on the maximum clique problem.

In this paper we study the N-prisoners puzzle, as described by Ebert in his PhD thesis [10], as a problem to investigate and stretch the capabilities of genetic programming. We hypothesize that this problem will present initial difficulties for genetic programming and will force critical evaluation of its application. The N=3 puzzle of the problem is known as the 3-hat puzzle. In this case, three players are assigned a red or blue hat. Each must guess their own hat colour by only seeing the other players' hats. If at least one player guesses correctly, and

no one guesses incorrectly then all the players win, otherwise, they all lose. In addition to guessing red or blue, each player can pass, but still one player must guess correctly to win. The goal is to solve the puzzle correctly over a number of different hat combinations.

The problem is attractive to genetic programming for several reasons and this report serves as an initial investigation and as a proposal of several future research directions. As described by Ebert in [11], Hamming codes allow (with high probability) the correct guessing of the next bit in a growing sequence of random bits. The same technique can be used for solving some of the N-prisoners puzzles. The decision making aspect of solving the puzzle can be represented by computer programs and the search through possible programs with genetic programming. The problem has not been previously investigated, to the authors' knowledge, with evolutionary computation or genetic programming. Also, there is no known optimal solution for all values of N, which differs from typical benchmark and common problems in genetic programming. In this investigation we perform an empirical study to determine how effective genetic programming is in solving the problem, consider possible difficulties and propose future research directions.

The results indicate genetic programming's ability to find solutions on this challenging problem. The fact that the very difficult Hamming solution is not evolved only confirms our expectations that the problem and genetic programming will require in-depth analysis to achieve this high goal. Several research extensions are described that are currently underway. Section 2 describes the N-prisoners puzzle, Section 3 gives our genetic programming approach, with extensions and conclusions following in Sections 4 and 5.

2 Problem Description

The 3-hat problem was made popular by Ebert while working on his PhD thesis and in a subsequent article in the New York Times [12]. According to Ebert, the actual problem is attributed to Walter Wesley Winters (1905-1973). To solve the 3-hat puzzle we notice that $\frac{3}{4}$ of the time two individuals will have the same colour hat and the third will have the different colour. If each player guesses the opposite of his co-players hat colour, when they have the same colour hat, and passes when the co-players have different coloured hats, then the group only loses when all three have the same hat colour. There are 2^3 possible hat combinations with 2 of those having either all red or all blue, where these 2 combinations will cause the algorithm to fail. The N-prisoners puzzle is essentially the same, just with N people and hats, but is described next.

2.1 N-prisoners Puzzle

N-prisoners are up for parole. They all go free or none go free determined by how they play a game. The game is that they each enter the parole officer's office independently and see randomly flipped coins on a desk, one representing

each prisoner, except the coin for the current prisoner is covered. The prisoner has to guess heads or tails, or pass, and then leave the room. In this manner each prisoner sees the coins, makes a guess and leaves. Again, the prisoners can formulate a strategy before beginning the game. All the prisoners go free if at least one guesses correctly and none guess incorrectly. An obvious solution to this problem is if the same prisoner always says heads and the rest always pass. This gives the group a fifty percent chance each time.

2.2 Game Representation

We will view the game as attempting to guess bit b_i in the vector $B = (b_0, b_1, \ldots, b_n)$, where each $b_i \in \{0, 1\}$. The vector $G = (g_0, g_1, \ldots, g_n)$ represents a guess for B such that each $g_i \in \{0, 1, p\}$, where p is the pass option. Thus, we want to know if $Win(G, B) = 0$, where $Win(G, B)$ can be defined as follows:

$$Win(G,B) = \begin{cases} 0 & \text{if } (\exists g_i \in G \cdot g_i = b_i, b_i \in B) \wedge \\ & (\forall g_i \in G \cdot g_i = p \vee g_i = b_i, b_i \in B) \\ 1 & \text{otherwise} \end{cases}$$

As described above, when guessing b_i, we can see the other bits in B but not b_i. Thus, winning the game is determined by assuring there is at least one b_i that is correct and all the other bits in B are either correct or equal to pass. Here, a 0 indicates correct play (a win) while a 1 denotes a penalty for losing. Notice that each b_i represents a player's hat or prisoner's coin and the guess of b_i is represented by g_i.

2.3 Hamming Code Solution

Hamming codes are used for error detection and correction in coding theory [13]. For bit strings of length $2^k - 1$ we can detect single errors and correct them using Hamming codes. Thus, the N-prisoners puzzles that can be represented as bit strings of length $2^k - 1$ are the $\{3, 7, 15, 31, \ldots\}$-puzzles. To solve our game with Hamming codes, we construct a parity check matrix P that represents the binary numbers $1 \ldots 2^k - 1$ where each binary number is a column. We then do a matrix multiplication $y = P \times B$, where the vector B represent the $N = 2^k - 1$ players' hats or prisoners' coins and player/prisoner i is guessing $b_i \in B$. The result y indicates with all 0's that we have a code word, or else which bit, read in binary, needs to be flipped in order to have a code word. For each $b_i \in B$, if we guess that $b_i = 0$, calculate y_0 and do likewise with $b_i = 1$, then we will have y_0 and y_1. Note each error word is a Hamming distance of 1 away from a code word, meaning that there is one bit in the vector that needs to be flipped to obtain a code word.

The result $y_0 \neq y_1$ indicates that we have an error word. By flipping the bit value associated with the non-zero y_i we can have a code word. The result $y_0 = y_1 \neq 0$ indicates that we have an error word and flipping either bit does not lead to a code word. If we guess the opposite of the bit that gave us a code

$$P = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}, B = \begin{bmatrix} 0 \\ ? \\ 0 \end{bmatrix}, P \times B(? = 0) = y_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, P \times B(? = 1) = y_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Fig. 1. Example of Hamming code application with vector B and unknown bit b_1. Vectors y_0 and y_1 are found, where y_1 indicates that $b_1 = 1$ should be flipped to 0 to obtain a code word.

word in the first case, and pass all other times, then we can win the game with probability $\frac{2^k-1}{2^k}$. In the case when we have a code word, every b_i will produce a $y_0 \neq y_1$ and each will guess the opposite of the bit that makes the code word causing the guess of B to lose. Figure 1 gives a small example of the 3-puzzle Hamming code application.

3 Genetic Programming Approach

The overall aim is to investigate the possibility of employing a genetic programming approach to solve the N-prisoners puzzle of finding the optimal strategy to maximize wins. We know an optimal solution for the puzzles of $N = 2^k - 1$ and that they use Hamming codes. The solution could be implemented in several interesting and novel ways and each one of those solutions would have many possible representations. What we do not investigate here, however, is what we can do with finding the optimal rates of solutions, which is the subject of on-going work.

System parameters are set to those commonly used in the research community and found in [3] and that are default in the evolutionary computation system ECJ, described in [14]. A crossover rate of 0.9, reproduction rate of 0.1, maximum tree depth of 17, ramped half-and-half tree generation, maximum number of 51 generations, and various population sizes are used. Description of other design decisions follow, with terminals and functions in Table 1.

3.1 Fitness

To determine the fitness of each individual we generate every possible combination of bit strings for the puzzle. We then evaluate each individual for each place in the string, making that place's bit value hidden. The output of the individual is either 0,1 or *pass*, found by applying the following membership function to the actual output:

$$\text{output}' = \begin{cases} 0 & \text{output} \leq 0 \\ 1 & \text{output} \geq 1 \\ \text{pass} & 0 > \text{output} < 1 \end{cases} \quad (1)$$

Table 1. Functions, terminals and descriptions

Terminals	Description
0,1	the constants 0 and 1, also the possible values of the vector B
0's,1's	the number of zeros and ones the current querying bit can see
me	the location i of the current querying bit
$\Re$	ephemeral random constants between $0 \ldots N$
loopi	loop index, set to 0, incremented each iteration of the loop, $0 \ldots 2N$
ARG0	argument of ADF1 that can be called in this tree
ADF0	automatically defined function, no internal ADFs
Functions	
$+, -, \times$	binary addition, subtraction, and multiplication
$/, \%$	binary, protected division and modulus, return 0 if denominator = 0
what	unary, returns the value of bit referenced by argument, 2 otherwise
iflte	4 arguments: if arg1 $<$ arg2 then arg3 else arg4, returning arg3 or arg4 results
prog2n	binary, executed arg1, then arg2, return arg2 result
foreachp	unary, executing argument N times and increments loopi variable, returning result of last execution
read	unary, returns local variable referenced by argument, if arg$<$ 0 the 0, if arg$>$ $2N$ then $2N$
write	binary, sets local variable referenced by arg1 to arg2, boundary condition same as read, returns value written
ADF1	unary, automatically defined function, access argument with ARG0, function and terminal set includes ADF0

Next, we compare the generated guess against the actual instance. If at least one is correct in guessing and none are incorrect, then we do nothing, else we increment by one signifying a loss. Thus, our fitness indicates the percentage of loses, $\frac{loses}{2^N}$, which we want to minimise. It should be noticed that for larger N values, say $N = 17$, this will become inadequate as 2^{17} combinations require significant evaluation time. We could make use of the fact that there is a lot of symmetry in combinations or create a reasonable number of random test combinations. However, only the 3,4,5,7 puzzles are examined in this initial investigation.

3.2 Results

Initial experiments used only the first 6 terminals and the first 7 functions in Table 1. The complete functions and terminal sets were then tried with varying population sizes and the results of the 500 population are reported, which we feel to be representative. For each puzzle, (3,4,5,7), 10 runs were done. In all attempts genetic programming was able to find a strategy that had a losing ratio below $\frac{1}{2}$. All experiments were also tried without automatically defined functions (ADFs) and several exploratory runs were made with varying population sizes and functions and terminal set combinations. For the 3-puzzle, the $\frac{3}{4}$ solution

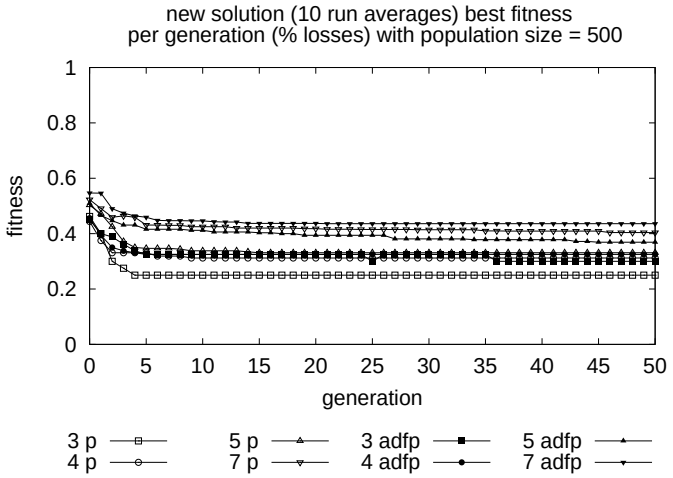

Fig. 2. Average of best fitness graph. "N p" indicates the N-puzzle and "N adfp" indicates the same configuration with ADFs.

is usually found in the initial generation. While the plots, in Figure 2, show the average fitness beginning near 1 and evolving to below $\frac{1}{2}$, we notice in the best of generation plot that best-of-run individuals are usually found in early generations. This indicates that evolution is either finding the global optima or getting stuck in a local optima, which is the case here for the 4,5,7 puzzles being trapped into local optima. Figure 3 shows two evolved individuals for the 7-puzzle that did particularly well. We next examine some hand engineered strategies, but note that Ebert points out that good solutions can be found for the non-$N = 2^k - 1$ puzzles by simply using a variant of the Hamming solution and always passing on some bits. Thus, for M-puzzles where $M > N$, if we always pass on bits $b_N \ldots b_M$, we can use the N-puzzle solution and win $\frac{N}{N+1}$ of the time.

3.3 Engineered Solutions

Using only the first 6 terminals and 7 functions in Table 1, hand engineered (coded) solutions for the 3-puzzle (fitness of $\frac{3}{4}$) and the 7-puzzle (fitness of $\frac{7}{8}$) were created. Both individuals implement the Hamming solution by using nested summations and parity checks and the modulus operator. Our solution is puzzle specific, determining whether each bit can be flipped to make a code word. The 7-puzzle solution has 701 nodes in its tree and scores a fitness of .125, or $\frac{7}{8}$ as expected.

With all the functions and terminals we can design an ADF individual with only 75 nodes that implements the ideal solution for the 7-puzzle, and a fewer nodes for the ideal 3 player solution, which is show in Figure 4. This engineered

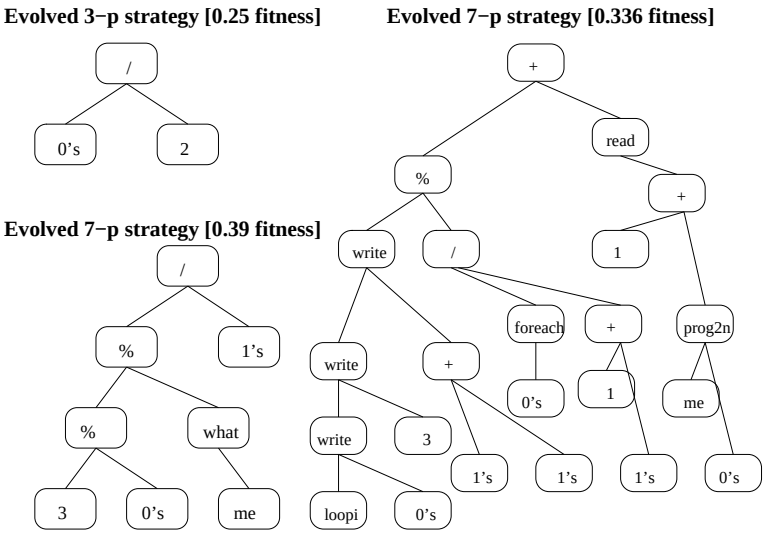

Fig. 3. Evolved solutions for 3-puzzle and two for the 7-puzzle.

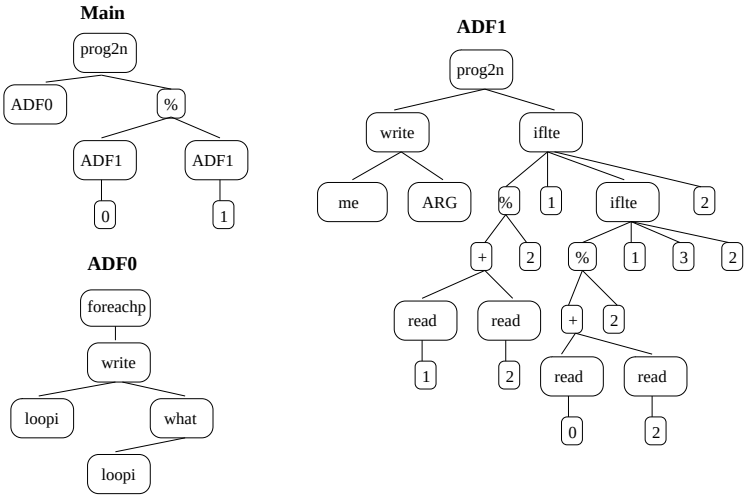

Fig. 4. Engineered 3-puzzle Hamming solution. Some nodes were left off the top of Main that force the output into the correct ranges of the membership function. The Engineered 7-puzzle Hamming solution is the same with the right child branch of ADF1 containing the correct parity checks, approx. 20 additional nodes.

solution can also be represented without ADFs and is not believed to be unique but could be represented in many ways with this set of functions and terminals.

3.4 Brief Problem Analysis

The probability of guessing N random independent bits, [0,1], is $\frac{1}{2^N}$. However, the problem allows the *pass* option, and if we assume we pass on every bit save one, the probability of winning the game increases to $\frac{1}{2}$. Also, we notice that if we generalise the 3-puzzle solution, that has probability $\frac{3}{4}$ of winning, to pass on all but the first three bits, we can have a probability of winning of $\frac{3}{4}$ for all puzzles where $N > 3$. However, the difference between individuals of this 3-puzzle solution and the 7-puzzle Hamming solution are great. Small changes in individuals would not cause small changes in fitness, creating a very rugged solution landscape. The fitness calculation for the Royal Tree [1] problem was adjusted to give partial, full and bonus credit to solutions close to the ideal. We may also need to consider something similar here. But we note that our selection of functions and terminals may in fact hinder the search, as was shown in [5] with the binomial-3 problem.

4 Future Extensions

Our next phase of research will investigate several possible extensions that we hope to be able to find the Hamming solution. We first must overcome the problems we identified but see some of the below extensions as possible solutions. In addition to these attempts to *find* an optimal solution, we are also interested in analysing the genetic programming search and discovering the effects of changing the evolutionary toolkit and the differences between different N values of the puzzles.

Multi-agent Approach. The puzzle can be won with only one bit guessed correctly, and all others passing, and using a single strategy to guess every bit may be too difficult. Instead of evolving a single strategy we could evolve N strategies that each attempt to correctly guess a different bit and learn to co-operate to maximize wins.

Multi-objective Fitness Function. It seems logical from our study here that the fitness of simply *win* or *loss* is not fully effective in driving the evolutionary search. Thus, as done with the Royal Tree problem, a fitness which incorporates other aspects of the game may need to be included. Knowles' et al [15] method of representing single objective functions as multiobjective ones may be a possibility. Other objectives that group losses together or maximize individual correct bit guesses may be helpful.

Island Model Approach. Results also indicate a difficulty of the evolutionary search to break out of local optima. Diversity pressure could be a way to keep the search going and a possible solution could be using an island model. Several possible representations would be possible for this model and problem.

Restating The Problem. It may be the case that solving the problem directly is too difficult for a straightforward evolutionary computation approach. We may benefit by restating the problem. Since we only need one bit to be guessed correctly to win the game, by forcing all other bits to always pass, we can focus on evolving a single bit strategy.

5 Conclusions

The N-prisoners puzzle gives evolutionary computation and genetic programming, in particular, a new challenge to overcome. This initial study investigated the problem and showed genetic programming solutions and presented some possible future extensions. The problem's rough solution space, the possibility of evolving complex data structures and algorithms, and the possibilities for future research all make it an attractive area for further genetic programming research. As shown here, genetic programming is able to evolve solutions that consistently do better than 50%, and we hypothesize that through methods described in the previous section we will be able to greatly improve upon this. Also, by studying the evolutionary search on this hard problem we hope to learn more about why genetic programming can find solutions and how to improve the process by which it works. In short, we hope to gain a deeper understanding of how and when genetic programming works well on difficult problems. We feel that the investigation of this problem will lead to a worthwhile examination of evolutionary computation systems and genetic programming.

Acknowledgments. The authors appreciate the review of early versions of this work by Dr. David Gustafson at Kansas State University and members of the ASAP Research Group at the University of Nottingham.

References

1. W.F. Punch, D. Zongker, and E.D. Goodman. The royal tree problem, a benchmark for single and multi-population genetic programming. In P.J. Angeline and K.E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 15, pages 299–316. The MIT Press, Cambridge, MA, 1996.
2. W.B. Langdon. *Data Structures and Genetic Programming: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming*. Kluwer, Boston, 24 April 1998.
3. J.R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
4. J. Koza, F. Bennett, and D. Andre. Using programmatic motifs and genetic programming to classify protein sequences as to extracellular and membrane cellular location. In V. William Porto et al, editor, *Proceedings of the Seventh Annual Conference on Evolutionary Programming*, volume 1447 of *LNCS*, San Diego, California, USA, 25-27 March 1998. Springer-Verlag.
5. J.M. Daida, J.A. Polito, S.A. Stanhope, R.R. Bertram, J.C. Khoo, and S.A. Chaudhary. What makes a problem GP-hard? analysis of a tunably difficult problem

in genetic programming. In Wolfgang Banzhaf et al, editor, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 982–989, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

6. S. Luke. Genetic programming produced competitive soccer softbot teams for robocup97. In J.R. Koza et al, editor, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 214–222, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.
7. D. Andre and A. Teller. Evolving Team Darwin United. In M. Asada and H. Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*, volume 1604 of *LNCS*, pages 346–351, Paris, France, July 1998 1999. Springer Verlag.
8. S.M. Gustafson and W.H. Hsu. Layered learning in genetic programming for a cooperative robot soccer problem. In J.F. Miller et al, editor, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, Lake Como, Italy, April 2001. Springer-Verlag.
9. T. Soule, J.A. Foster, and J. Dickinson. Using genetic programming to approximate maximum clique. In J.R. Koza et al, editor, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 400–405, Stanford University, CA, USA, 28–31 July 1996. MIT Press.
10. T. Ebert. *Applications of Recursive Operators to Randomness and Complexity.* Ph.D. thesis, University of California at Santa Barbara, 1998.
11. T. Ebert. On the autoreducibility of random sequences. Unpublished. http://www.ics.uci.edu/~ebert/, 2001.
12. S. Robinson. Why mathematicians now care about their hat color. *The New York Times: Science Desk*, 10 April 2001.
13. R.W. Hamming. *Coding and Information Theory.* Prentice-Hall, Inc, New Jersey, USA, 1980.
14. S. Luke. *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat.* PhD thesis, Department of Computer Science, University of Maryland, University of Maryland, College Park, MD 20742 USA, 2000.
15. J.D. Knowles, R.A. Watson, and D.W. Corne. Reducing Local Optima in Single-Objective Problems by Multi-objectivization. In E. Zitzler et al, editor, *First International Conference on Evolutionary Multi-Criterion Optimization*, pages 268–282. Springer-Verlag. LNCS no. 1993, 2001.