Genetic Programming, Logic Design and Case-Based Reasoning for Obstacle Avoidance

Andy Keane(✉)

Faculty of Engineering and the Environment, University of Southampton, Southampton SO17 1BJ, UK
ajk@soton.ac.uk

Abstract. This paper draws on three different sets of ideas from computer science to develop a self-learning system capable of delivering an obstacle avoidance decision tree for simple mobile robots. All three topic areas have received considerable attention in the literature but their combination in the fashion reported here is new. This work is part of a wider initiative on problems where human reasoning is currently the most commonly used form of control. Typical examples are in sense and avoid studies for vehicles – for example the current lack of regulator approved sense and avoid systems is a key road-block to the wider deployment of uninhabited aerial vehicles (UAVs) in civil airspaces.

The paper shows that by using well established ideas from logic circuit design (the "*espresso*" algorithm) to influence genetic programming (GP), it is possible to evolve well-structured case-based reasoning (CBR) decision trees that can be used to control a mobile robot. The enhanced search works faster than a standard GP search while also providing improvements in best and average results. The resulting programs are non-intuitive yet solve difficult obstacle avoidance and exploration tasks using a parsimonious and unambiguous set of rules. They are based on studying sensor inputs to decide on simple robot movement control over a set of random maze navigation problems.

Keywords: Decision tree · Data mining · Feature engineering · Classification · Algorithm construction

1 Introduction

Genetic Programming (GP) is the basic building block of the methods proposed here. GP was initiated in the early 1990s, when Koza applied genetic algorithms (GAs) to the evolution of computer programs (Koza 1992). GP extends the use of GAs to evolve structures of significant complexity which demand sophisticated adaptive plans to improve their performance. Since that time very many papers have used the basic ideas proposed by Koza to study a bewildering array of problems, see for example the review by Espejo *et al.* (2010) of the applications

C. Dhaenens et al. (Eds.): LION 9 2015, LNCS 8994, pp. 104–118, 2015.
DOI: 10.1007/978-3-319-19084-6_9

of GP in classification and for an application in robotics the work of (Seo *et al.* 2010). In this paper GP is used to evolve control programs for a simple mobile robot equipped with adjacency touch sensors and the ability to move forward or rotate 90° to the left or right. This is a very basic problem but one that serves to illustrate the ideas being proposed and some of the shortcomings of a straightforward application of GP to obstacle avoidance programs.

In a conventional GP approach a series of random initial programs are used to seed the GP process and these are then evolved using the standard operations of crossover and mutation under the impact of selection pressure to produce improved programs to tackle a set of trial tasks. The basic problem with this simple naïve use of GP is that the decision trees evolved are generally difficult to interpret and can rapidly become very large and cumbersome. When designs become large they then become more costly to evaluate and this slows the whole search process down. Moreover, it is quite common to find that evolutionary improvement can stall after a short period of initial improvement unless recourse is made to very large population sizes. The whole aim of the present work is to bring to bear ideas from logic design and case-based reasoning to deal with these issues.

This paper is laid out as follows: in section two we briefly introduce the path planning task being studied before going on to describe the basic ideas of how GP systems can be applied to this problem in section three. In section four we provide some illustrative results obtained using a naïve GP system, while in section five we show how the *espresso* algorithm can be used to simplify evolved program structures, speeding up the search process and improving outcomes. We then show how the *espresso* logic tool can be applied to yield an improved GP process in section six before drawing our conclusions in section seven.

2 Path Planning

In path planning and obstacle avoidance two basic approaches can be adopted: either an initial exploration phase is carried out to map the robot's world, following which planning decision can be made using the derived map, possibly with map updates or, alternatively, decisions must be based on current sensor inputs and possibly previous readings and actions without a world map. When dealing with significantly changing or new environments such as for UAV control, it is clear that the off-line building of world maps of other UAVs and obstacles, prior to decision making is not appropriate. Therefore we consider here the problem of planning based on sensor inputs. Moreover, to simplify the problems being studied, we restrict ourselves to problems where decisions must be based solely on current information states. Thus the task faced by the robot is: given a mission to accomplish and current sensor readings, should the robot move forwards, turn right or turn left. The mission considered is the commonly used one of visiting as much of the available world as possible in the least number of moves while not colliding with obstacles, see for example Bearpark and Keane (2008).

Such a task can be carried out using a range of programming structures but that most similar to the approach adopted by human navigators is a form of

case-based reasoning: i.e., a decision tree where a series of sensor predicates are tested one at a time until a match is found following which a pre-defined action is performed. Moreover, while human operators can be inconsistent in the actions they take when presented with sets of sensor inputs, it is important when considering problems subject to regulatory approval, such as vehicle navigation, that predictable behaviour is adopted. For example "rule-of-the-road" requirements on cars and aircraft lay down the expected behaviour of operators when confronted with certain known scenarios, i.e., turn to the right when confronted with an oncoming vehicle so as to pass port side to port side. The great attraction of such case-based approaches is that their logic can be studied and understood, often exhaustively, for all possible scenarios; see for example Weng et al. (2009). Our aim here is to try and build a GP system that produces well-structured case-based programs for dealing with the robot task planning problem studied. This is not as trivial as might at first be thought if the full power of the evolutionary operators of cross-over and mutation is not to be curtailed.

The key step proposed here is to adopt ideas developed for logic circuit design to allow a GP system to automatically build the case-based program structures desired. In a previous paper it was shown that the convergence of GP in producing robot path planning programs could be improved by re-writing the programs being developed in case-based form during the evolutionary search process (Bearpark and Keane 2008). In that paper arbitrary Reverse Polish Notation (RPN) control programs were re-written as large single case statements before being re-inserted into the GP system, something the authors termed conversion to "canonical form". The process used to carry out this conversion was not straightforward or simple to implement. Here a similar approach is adopted but program re-writing is accomplished by borrowing tools from VLSI circuit design. VLSI logic circuit design commonly involves millions of logic gates and these are routinely simplified using well established methods. The most well-known of these is the so called "*espresso*" algorithm originated at the University of California, Berkeley, now made available as part of the Octtools[1] package (McGeer et al., 1993). VLSI logic circuit simplification essentially boils down to taking a truth table that maps input states to output states for the proposed design and producing a new, simplified, design that produces the same truth table. *Espresso* does this in a very efficient way giving a high quality, if not always perfect, reduction in the number of logic gates needed to create a given truth table. Here we use the same algorithms to re-write the RPN structures being evolved by GP. Since the resulting structures are almost always shorter, often massively so, their evaluation may then be carried out much more quickly, allowing either faster or more exhaustive searches to be run, see for example Moraglio et al. (2012). The reduced program structures also fundamentally change the actions of mutation and crossover on a population of designs. As far as can be found by searches of ISI publication data-bases, this is the first time that the *espresso* algorithm has been applied to a GP system in this way.

[1] http://embedded.eecs.berkeley.edu/pubs/downloads/octtools/.

3 Genetic Programming and Robot Control

A Genetic Algorithm (GA) uses techniques based on the natural evolution of species to gradually improve the quality of the data structures that it produces until an optimal solution is found, or the run is terminated. Generally, an initial set of possible solutions is chosen randomly to form the first generation. The GA performs genetic operations on the population, to produce another generation, and the process is repeated for a number of generations. The principal genetic operations are those of selection and crossover. The selection process measures the success of each member of the population in performing the allotted task, and selects the better members as candidates for a mating pool, here of the same size as the population. On average, high-scoring members of the population appear multiple times in the mating pool, while low-scoring members are not selected. Consequently the average quality of the population increases with each generation. A new generation is evolved by the crossover operation using the concepts of sexual reproduction. Further genetic operations may be performed, particularly mutation, in which randomly chosen genetic material is removed from a child and randomly generated material inserted in its place. In a GP system the data structures are computer algorithms, normally represented as trees in which the nodes are function nodes, representing a sub-routine in the algorithm, or terminal nodes, representing constants or variables defined by the algorithm. A simple example is shown in Fig. 1. The algorithm is 'executed' by a depth-first traverse of the tree, starting from the root node, searching for function nodes and their operands. An examination of nodes 2, 3 and 4 yields the logical value *true* or the logical value *false*. A *true* result causes the traverse to continue by examining node 5 and its operands. A *true* value for Z causes the execution of action A while a *false* value executes B. In the case where the expression X AND Y is *false*, the algorithm executes C. This program may be written in Reverse Polish Notation (RPN) form as: C, B, A, Z, IF, Y, X, AND, IF. To encode this for GP each operator, state and action is given a numerical code and the RPN re-written as a numerical string. When carrying out mutation and crossover care must be taken to ensure that only syntactically correct strings are produced. Thus each operator has a fixed number of operands (its cardinality) and each operator must have the correct type(s) of operands (i.e., AND, OR and NOT operands require state inputs and generate state outputs while an IF

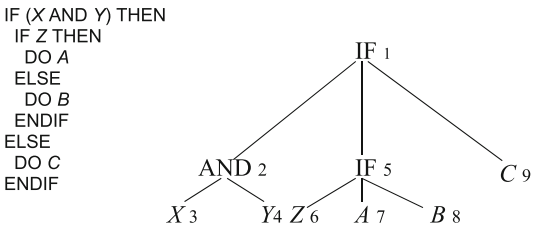

Fig. 1. A sample of control pseudo-code and the equivalent tree.

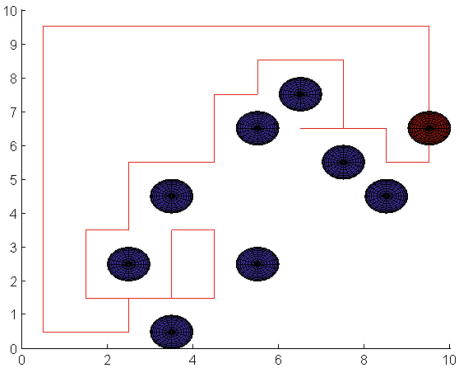

Fig. 2. The robot in its world (coloured red) with obstacles coloured blue and a possible path (the red solid line) (Color figure online).

requires either one state input and two action outputs or one state input and two state outputs and generates either an action or a state output, respectively – in this example X, Y and Z are states and A, B and C are actions).

Our GP system evolves algorithms that enable a software agent to solve basic problems in spatial exploration. An example is shown in Fig. 2. An enclosed 10 by 10 space has fixed internal obstacles coloured blue while the software robot itself is coloured red. The robot can either move forward one space, turn left or turn right before reassessing its environment. Here the robot is required to visit as many cells as possible using only a sense of touch, i.e., knowing only which of the eight adjacent cells is occupied. The robot thus has eight state inputs and three action outputs while the red line indicates a possible path. The score achieved by the robot is the number of squares visited divided by the number of vacant squares in the world. By testing a robot over a set of maze problems with random obstacle numbers and positions an average performance score can readily be computed – if the collection of random obstacle sets (mazes) is stored and used repeatedly, that score is stationary and can be used in a GP process to evolve better control programs. Notice that because the robot is blind and has no memory it can only navigate by touch and so all competitive navigation schemes involve moving towards an obstacle and then moving from obstacle to obstacle using these obstacles to aid navigation (precisely the sort of groping manoeuvre familiar to humans entering a darkened room). In an empty world a

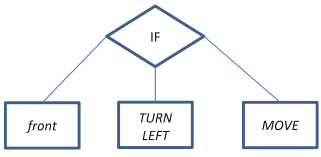

Fig. 3. A simple wall follower tree – this design scores 0.2761 when averaged over 50 of the test mazes.

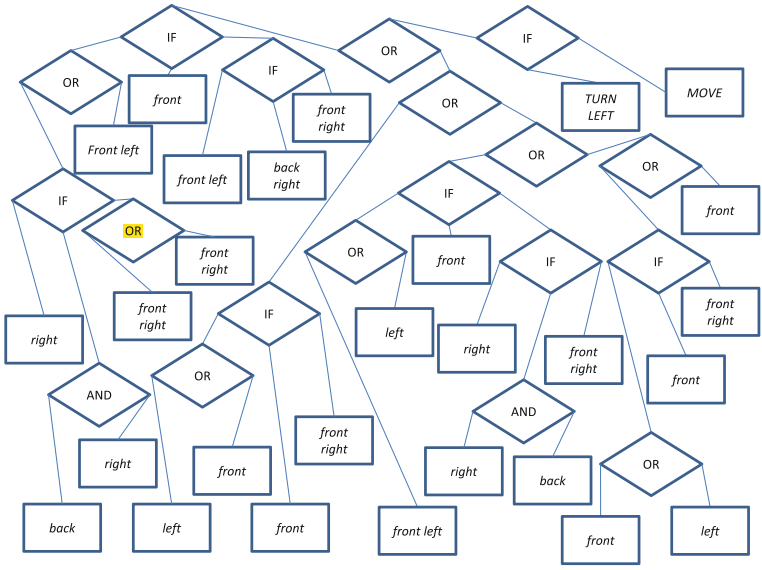

Fig. 4. A tree of 47 elements evolved by naïve GP after 100 generations – in RPN this is *MOVE, turn left, front, front right, front, front, left,* OR, IF, OR*, front right, right, back,* AND*, right,* IF*, front, front left, left,* OR, IF, OR*, front right, front, front, left,* OR, IF, OR*, front right, back right, front left, IF, front, front left, front right, front right, OR, right, back*, AND, *right*, IF, OR, IF, OR, IF – this design scores 0.3922 when averaged over 50 mazes.

simple thought experiment reveals that the best logic that can be achieved is to move to the edge of the world and then circumnavigate the edge either clockwise or anticlockwise, giving a maximum score in a ten by ten world of 44/100 (if one starts with one's back to the wall and walks forward to the opposite wall and then moves around the perimeter). The tree in Fig. 3 does just this and will cause the robot to move in the direction in which it is facing until it reaches the wall when it will turn left and follow the wall in an anticlockwise direction. This tree can be encoded in RPN as: *MOVE, TURN LEFT*, *front*, IF. As already noted, execution of the program requires the identification of a path from the root node to a terminal node by a depth-first traversal of the tree. Such a path is determined by the internal logical operators and the values of their operands. If the terminal node of a path is a sensor value, the traversal continues by providing this value to its parent operator. If the terminal node is an action, the action is taken and the traversal is terminated. This changes the relationship between the robot and its environment and the control program is executed again from the root node. The purpose of the GP system used here is to supply each robot in the population with an algorithm to guide it in solving the set of maze problems. The fitness of the algorithm is measured by the score achieved by the robot averaged over a set of 50 mazes. Selection for the mating pool that will

produce the next generation is based on fitness, so the better robots survive and may breed with other robots. Each breeding process produces two children who may or may not be fitter than their parents. The tools that the GP system has at its disposal are the logical operators *IF*, *AND*, *OR* and *NOT*, eight state sensors with which the robot is able to detect an obstacle in an adjacent cell (*front, front-right, right, back-right, back, back-left, left* and *front-left*) and three actions *MOVE*, *TURN RIGHT* and *TURN LEFT*. Note that two forms of the *IF* statement are used: in the first the IF statement tests a logical (sensor state) input and then selects from one of two possible actions while in the second it selects from one of two possible logic (sensor) states to output. These need to be distinguished to establish syntactically correct programs: the second kind of *IF* can provide inputs to further *IF*, *AND*, *OR* and *NOT* statements while the first cannot – the *IF* in Fig. 3 is of the first kind, while both kinds are seen in subsequent designs.

We additional use the concept of fuel to refer to the fact that each robot is limited to a fixed number of actions when it gets its turn in the mazes – here 100 actions are allowed per maze, i.e., the GP program is looped over a maximum of 100 times – this is sufficient to allow the exploration capabilities of a design to be fully assessed. In the simplest version of the GP system, each robot in the population has sole occupancy in the maze while its fitness is measured. The mazes each have on average 10 obstacles but varying between as few as five and as many as 15.

4 Some Illustrative Results

If we run a simple GP process to design control programs for this problem, permitting crossover between any valid position and mutation across any node or leaf (ensuring syntactically correct outcomes) it is possible to rapidly evolve the simple wall following design of Fig. 3 into more powerful forms. Figure 4 shows a typical program structure while Fig. 5 shows its path around a typical world. As can be seen from Fig. 4 a highly complex and non-intuitive program structure has evolved. This particular run of the naïve GP used a population size of 100 with 100 generations, each member of the population being trialled over 50 obstacle courses at each generation. The initial population is seeded with the structure from Fig. 3 plus 99 random, but syntactically valid, designs. The GP uses roulette wheel selection, 40 % cross-over probability and 20 % mutation probability (n.b., this latter probability is the probability that a member of the population undergoes a single mutation operation as opposed to the quantity of genetic material being modified). The final score for this design averaged over 50 mazes is 0.3922 as compared to 0.2761 for the original wall follower design. Not only are the programs evolved by the naïve GP often cumbersome, their structure makes them increasingly difficult to improve on using just selection, cross-over and mutation – note the duplication and occasionally redundant elements in Fig. 4 (where identical sensor inputs are ORed together – highlighted). This was the observation made by Bearpark and Keane referred to earlier (Bearpark and Keane 2008).

If this basic GP process is repeated 100 times using different random number sequences in the GP (with a fixed set of mazes) the results of Figs. 6 and 7 are produced – the best, mean and standard deviation scores are 0.4547, 0.4111 and 0.0306, respectively. The elapsed CPU time required to carry these 100 searches using a single CPU was 5.464E6 s. It is this performance that any new algorithm should seek to improve on.

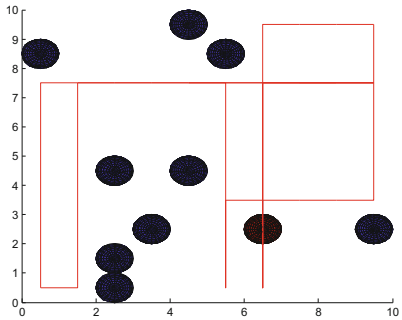

Fig. 5. The naïve GP evolved robot of Fig. 4 in its world (coloured red) with obstacles coloured blue and resulting path (the red solid line) (Color figure online).

5 Using *espresso* to Simplify Logic Trees – Case Statements

To make improvements to the design process we observe that our robot control program is a way of mapping a set of eight observable states that can be either *true* or *false* into one of three action outcomes. This is entirely similar to the task performed by VLSI logic circuits, although it is noted that VLSI designs most commonly work with two output states (0 or 1) though –1, 0, 1 output devices are also possible. We therefore proceed as follows: first a given control program is interrogated by exhaustively tabulating all possible output values for the 2^8 possible combinations of sensor input states; second the resulting truth table is passed to a VLSI logic circuit simplification routine (here a version of *espresso*) and third the resulting reduced logic table is then used to construct a case-based robot control program where each unique input state needed to recreate the program action is mapped to exactly one element in the case statement. The resulting decision tree is (a) much more highly structured, (b) generally more compact, (c) therefore faster to execute, (d) more readily understood by a human reader and e) more useful for subsequent use by the GP system since cross-over can now move entire sets of case elements between members of its population. The reduction in program size significantly speeds up the whole GP process while allowing it to produce better designs, typically halving evaluation times. Since VLSI logic programs are so highly optimized the cost of thus re-writing population members is trivially small as compared to evaluating designs

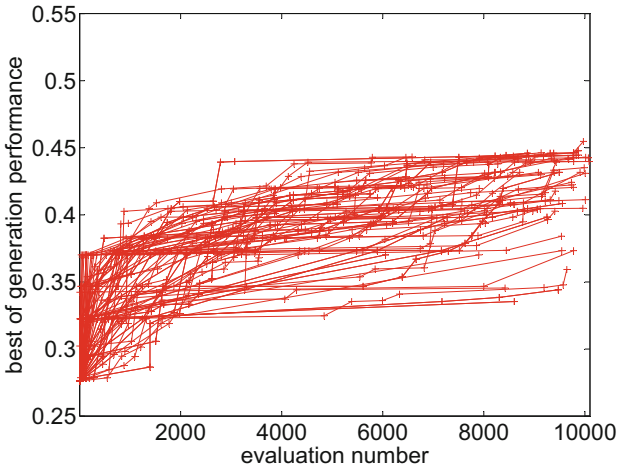

Fig. 6. Optimization histories of 100 runs of the basic GP process with a population of 100 members each being used for 100 generations – plots stop when the search stalls.

over multiple mazes. The only subtlety required to carry out this process is to adapt the essentially two output level tools available online to the task of dealing with more actions, here three. The use of the *espresso* algorithm in this context represents a considerable advance over the custom coded approach adopted earlier (Bearpark and Keane 2008), allowing faster and more compact designs to be produced. Figure 8 illustrates the *espresso* processed version of Fig. 4: it is apparent that the simplified representation is much easier to comprehend, and that its form better reflects the abilities of the robot as it tackles the different problems encountered in its exploration. Essentially it is a case-based system: each rule tackles a problem case faced by the robot. These rules may be seen to take the following actions based on four distinct cases (sensor feature sets):

1. If *front* turn *left*;
2. Else if *front right* and not *left* turn *left*;
3. Else if *front right* and *right* and not *back* and not *front left* turn *left*;
4. Else if *right* and *back* and not *left* and not *front left* turn *left*;
5. Else *move*.

It is by no means obvious that this refined structure generates the same operational behaviour as the original one but in fact they generate identical tables.

As already noted the process used here to simplify logic trees involves first converting any given control program into its equivalent truth table. Since the robot has eight proximity sensors there are $2^8 = 256$ possible combinations of input sensor states. Each state is presented, one at a time, to the control logic using the same code that is used to simulate the software robot and the resulting action noted – this can take one of three values: turn left, move or turn right, which are then represented as -1, 0 and 1. Table 1 shows part of the truth table for a robot program. Truth tables of this kind cannot be presented

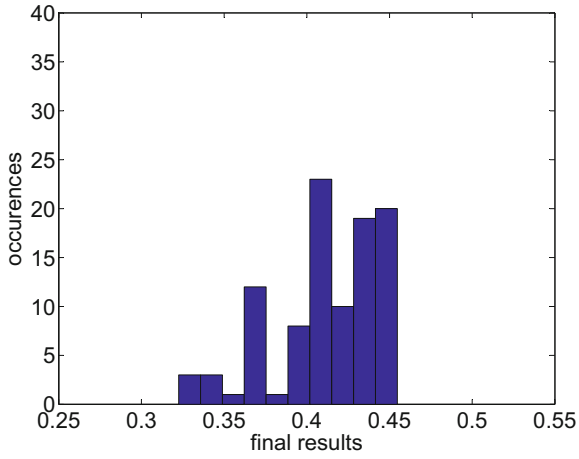

Fig. 7. Histogram showing variations of final results for the optimization histories of Fig. 6 – the best, mean and standard deviation scores are 0.4547, 0.4111 and 0.0306, respectively.

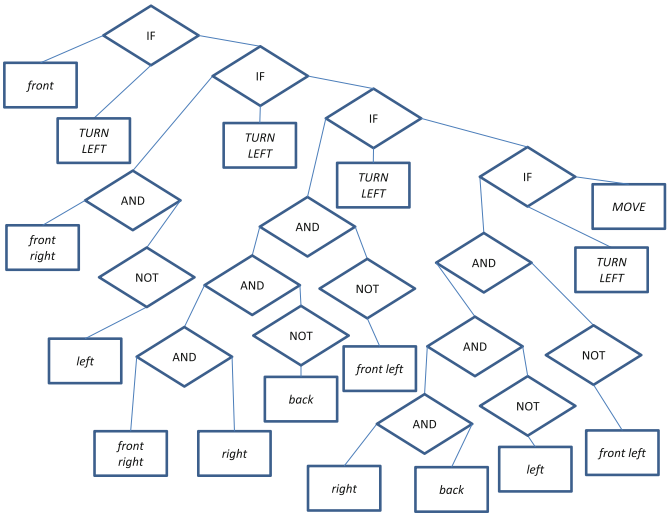

Fig. 8. The *espresso* simplified tree of 32 elements derived from Fig. 4 – in RPN this is *MOVE, TURN LEFT, front left,* NOT, *left,* NOT, *back, right,* AND, AND, AND, IF, *TURN LEFT, front left,* NOT, *back,* NOT, *right, front right,* AND, AND, AND, IF, *TURN LEFT, left,* NOT, *front right,* AND, IF, *turn left, front,* IF – this design scores 0.3922 when averaged over 50 mazes.

directly to *espresso* as it is designed to work with binary valued inputs and binary valued outputs – while the robot sensor map is a binary one, the action list is not. *Espresso* does, however, permit multiple function outputs providing

each one is only binary. Thus the next step is to convert the single three valued output in the table into two binary valued ones. To do this a *MOVE* output is encoded as [0 0], *TURN RIGHT* as [1 0] and *TURN LEFT* as [0 1], while [1 1] is not used, see the right hand columns of Table 1. Such truth tables can then be presented to *espresso* and rewritten in compact form. The resulting output will show which compound input states must be explicitly dealt with and what the appropriate action is for each. Table 2 shows the re-written table for the full truth table from which Table 1 is drawn. Notice that the simplified table contains only seven rows as compared to the 256 needed for the full table derived from the initial program. Note also the presence of "–" symbols in the table which are "don't care" symbols meaning the relevant line can be used for multiple matching entries, e.g., the final line of the table says if there is something in front of the robot and no other line has fired then turn left. This re-write process works for almost all input truth tables but it does permit an output action of [1 1] which has no meaning for a three action problem. *Espresso* rule re-writing can lead to such outcomes because it is designed for logic circuits where there is no concept of one-at-a-time testing of input states and thus multiple input states

Table 1. Part of a truth table for a typical robot control program and the three valued outputs converted to dual binary outputs

Inputs	Outputs	Binary outputs	
00001100	0	0	0
00001101	0	0	0
00001110	1	1	0
00001111	1	1	0
00010000	1	1	0
00010001	-1	0	1
00010010	1	1	0
00010011	-1	0	1

Table 2. *espresso* simplified truth table for a typical robot control program.

Inputs	Outputs
0001- - -0	10
0-00- -1-	10
00011- - -	10
- - -10- -1	01
- -1-0111	01
-101- -1-	01
1- - - - - - -	01

can sometimes be handled by concatenation. If such output lines do occur in the re-written tables the re-write operation is abandoned – such events are very rare. The final step in the process is to convert tables like that of Table 2 back into programmatic form. This is readily accomplished by creating a case statement where each line of the truth table maps to one case which is then terminated with a final action of *MOVE*. Each individual case is simply a sequence of sensor readings unioned together with AND statements if the sensor column shows a 1 and AND NOT statements if the sensor column shows a zero with the relevant action as specified in the output line, otherwise control passes to the next line in the table. It will be obvious to the reader that the truth table in Table 2 maps to the case structure in Fig. 6. Notice that the re-written statements do not make use of the OR statement at all. Because of this fact it is important not to over use the *espresso* algorithm as it can lead to a loss of genetic diversity.

6 Results: The Impact of *espresso* on the GP Process

To make use of the *espresso* capability an additional re-write operator must be added to the GP system. Re-writing 20% of the population after the actions of cross-over and mutation in every generation has been found to work effectively with the problem being studied here, though the results are not particularly sensitive to this setting – values ranging from 10% to 40% work similarly well. If this enhanced GP process is repeated 100 times using different random number sequences in the GP (with the same fixed set of mazes as used before) the results of Figs. 9 and 10 are produced – the best, mean and standard deviation scores are now 0.5133, 0.4151 and 0.0319, respectively as compared to the previous

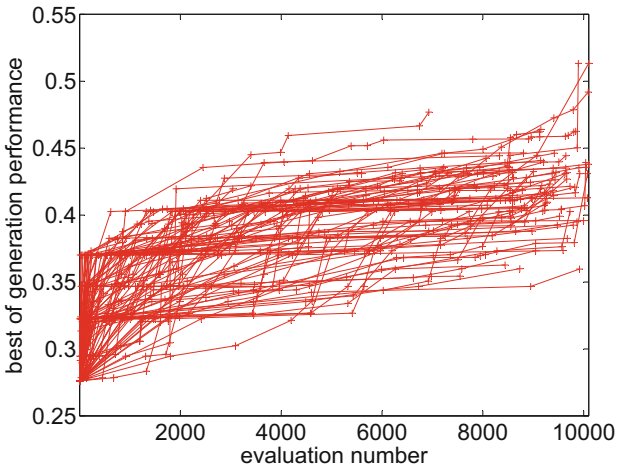

Fig. 9. Optimization histories of 100 runs of the *espresso* enhanced GP process with a population of 100 members each being used for 100 generations – plots stop when the search stalls.

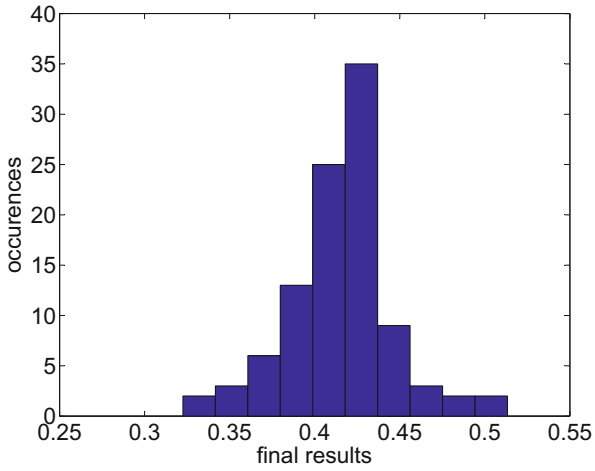

Fig. 10. Histogram showing variations of final results for the optimization histories of Fig. 9 – the best, mean and standard deviation scores are 0.5133, 0.4151 and 0.0319, respectively.

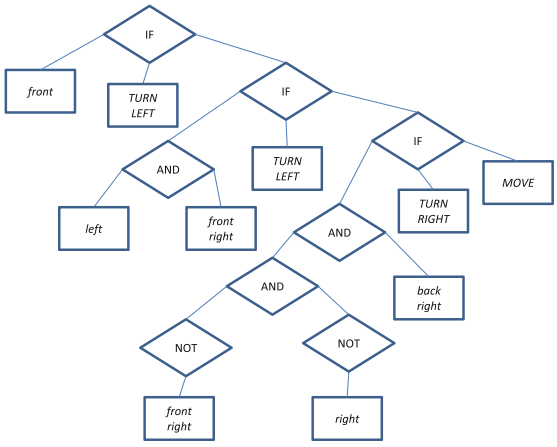

Fig. 11. A tree evolved by the *espresso* enhanced GP – this design scores 0.5133 when averaged over 50 mazes.

results of 0.4547, 0.4111 and 0.0306. The elapsed CPU time required to carry these 100 searches using a single CPU has reduced to 4.329E6 s from the previous run time of 5.464E6 s, a saving of some 20%. The search is faster, achieves better peak results and produces broadly similar average results (Welch's t-test shows the *espresso* based approach is better but only with a 36% significance level). While these gains are not overwhelming, they are very useful, particularly the improvement in peak performance. Figure 11 shows the best structure evolved with this enhanced approach and Fig. 12 shows its path through the same maze

as illustrated in Fig. 5. Its control program is appealingly simple, having the desired case-based structure and now with only three distinct cases (a compact set of useful features has been engineered):

1. If *front* turn *left*;
2. Else if *left* and *front right* turn *left*;
3. Else if not *front right* and not *right* and *back right* turn *right*;
4. Else *move.*

It also contains the important capability of being able to turn right or left. The logic of the final rule-base is, however, still not completely obvious and it is not at all clear that a human navigator would develop such an approach. Indeed the problem being studied here is extremely difficult for humans to write successful decision trees for. This is the great attraction of computer generated control programs – powerful yet readily studied logics can be produced without needing any special insights into the problem being confronted.

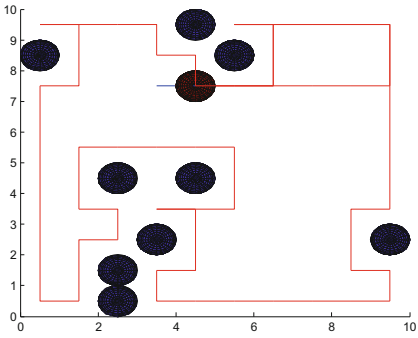

Fig. 12. The *espresso* enhanced GP evolved robot of Fig. 11 in its world (coloured red) with obstacles coloured blue and resulting path (the red solid line) (Color figure online).

7 Conclusions

This paper has shown how tools developed for VLSI logic circuit design can be used to improve the performance of a naïve GP system in producing a simple robot path planning task. The key observation is that restructuring GP derived programs as case statements not only helps improve subsequent understanding of the programs but also aids the GP system in developing them – the GP runs faster and it produces better designs through cross-over. Moreover logic simplification tools are extremely robust and powerful and so have negligible impact on the cost of evolving new structures. Here an eight sensor, three action system is explored but the ideas presented are capable of dealing with arbitrary numbers of inputs and outputs by adopting suitable encoding approaches to switch between decision trees, Reverse Polish Notation (RPN) and truth tables.

References

Bearpark, K., Keane, A.J.: Canonical representation in genetic programming. In: Parmee, I.C. (ed.) Proceedings of the Conference on Adaptive Computing in Design and Manufacture ACDM08, Bristol (2008)

Espejo, P.G., Ventura, S., Herrera, F.: A Survey on the application of genetic programming to classification. IEEE Trans. Syst. Man Cybern. **40**(2), 121–144 (2010)

Koza, J.: Genetic Programming: on the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)

McGeer, P.C., Sanghavi, J.V., Brayton, R.K., Sangiovanni-Vicentelli, A.L.: ESPRESSO-SIGNATURE: a new exact minimizer for logic functions. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **1**(4), 432–440 (1993)

Mencar, C., Castiello, C., Cannone, R., Fanelli, A.M.: Design of fuzzy rule-based classifiers with semantic cointension. Inf. Sci. **181**(20), 4361–4377 (2011)

Moraglio, A., Krawiec, K., Johnson, C.G.: Geometric semantic genetic programming. In: Coello, C.A.C., Cutello, V., Deb, K., Forrest, S., Nicosia, G., Pavone, M. (eds.) PPSN 2012, Part I. LNCS, vol. 7491, pp. 21–31. Springer, Heidelberg (2012)

Seo, K., Hyun, K.S., Goodman, E.D.: Genetic programming-based automatic gait generation in joint space for a quadruped robot. Adv. Robot. **24**(15), 2199–2214 (2010)

Weng, M., Wei, X., Qu, R., Cai, Z.: A path planning algorithm based on typical case reasoning. Geo-spat. Inf. Sci. **12**(1), 66–71 (2009)