Decision Tree Induction Using Evolutionary Algorithms

CITS4404 – Artificial Intelligence and Adaptive Systems Project Report

Group
Ammar Abu Shamleh 21521274
Zen Ly 21491901
Brian Lee 21492167
Alex Arnold 21304455

Abstract

This paper presents a genetic algorithm for evolving decision tree parameters, using the example of a robot cleaner in a 2D grid to demonstrate the proposed solution. The solution found by the proposed algorithm demonstrates exceptional results, completely surpassing any hand-designed solution to the example problem. However, the GA's solution is shown to lack robustness when exposed to conditions not tested in the fitness function.

1. Introduction

Decision trees are widely used decision support tools that use a tree to model decisions and their outcomes. Alternatively, decision trees can be used as a decision-making tool, enabling the selection of an appropriate action in response to some set of circumstances. They are widely used in both application and research domains, for process control, decision-making, and classification, among many other tasks. Decision trees are often used in the development of video game AI, thanks to their simplicity and ease of implementation [1]. They are also used in machine learning, often for classification problems.

While there is significant active research into decision tree induction using learning techniques [2], an alternative approach is the use of an evolutionary algorithm to train a decision tree, by determining the correct leaf node values for the tree. In most decision trees, leaf nodes represent either an action to take (given the set of conditions that led to that leaf node), or a classification of some sort. This paper presents a genetic algorithm for evolving a decision tree to control a cleaning robot, by evolving the values ascribed to the tree's leaf nodes, and using a fitness function to assess the validity/effectiveness of a given set of values.

1.1 Evolutionary Algorithms

Evolutionary algorithms are a form of evolutionary computation that operate on the principle of natural selection to evolve solutions to a problem [3]. They work by generating an initial set of (often random) solutions to a problem, and then evolving them over time to produce better solutions, in a repetitive cycle.

In each iteration, fitnesses (a measure of a solution's quality) are evaluated for the current population (i.e. current working set of solutions). Based on these fitnesses, parents are selected from the population for mating. Selected parents breed children, which take on

characteristics of the parents. Children are randomly mutated with a low probability, and then fed back into the population. This cycle is repeated, in the hope that good characteristics will produce good fitnesses, and thus get bred more often, and survive longer. Over a large enough number of generations, this process can often produce excellent solutions to a given problem, while still adhering to runtime constraints. This cycle is illustrated in Figure 1.

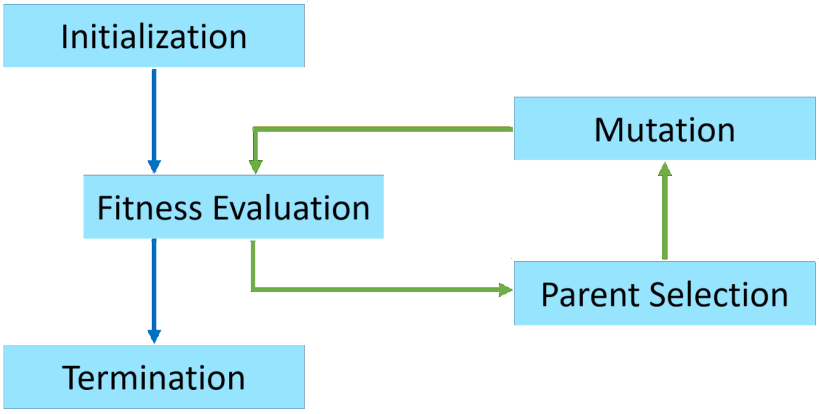

Figure 1: Evolutionary Algorithm Overview

Genetic algorithms are very commonly used for scheduling problems [4], timetabling problems [5], and engineering problems [6].

1.2 Previous Work

While the use of genetic algorithms to induce/train decision trees has been studied extensively in the past [10], [11], little research was found that mimicked the approach describe by this paper. Most alternatives presented in literature focused on using GA's to evolve the structure of a decision tree, which contrasted with this papers approach of inducing decision trees by evolving the parameters in a fixed-structure tree.

2. Robot Cleaner (The Problem)

In order to study the application of genetic algorithms to decision tree induction, we selected an appropriate example problem from literature. Melanie Mitchell describes in her book [7] a dummy problem, involving a robotic can collector on a 10x10 grid of squares.

Consider a 10x10 grid of squares, with all edges surrounded by walls. Each square is either empty, or populated with a can. Starting with a robot in the top left-hand corner of the grid, we wish to design a control scheme for the robot that will maximize the number of cans it collects in a limited number of moves, while minimizing the number of collisions it has with the walls surrounding the grid. This setup is illustrated in Figure 2.

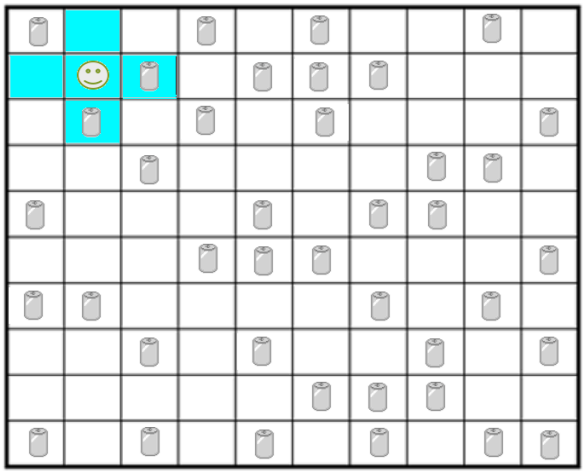

Figure 2: Grid of squares populated with cans

The robot only has awareness of its immediate Vonn-Neumann neighbourhood, as seen in Figure 2. This means that it can only see the squares immediately above and below it, and to its right and left. It can also see the square it currently occupies. Additionally, the robot has no state information, and no memory.

Using nothing other than this limited local information, the problem we have applied our genetic algorithm to is that of finding a control scheme for the robot, that will allow it to perform its task effectively.

2.1 Decision Tree Representation

As described earlier, a decision tree is a structure for representing and assisting in decision making processes. Typically, non-leaf nodes represent conditions and their edges represent possible evaluations of those conditions. Leaf nodes typically represent either actions, or classifications.

Figure 3 demonstrates this [8], with *Ai* representing a binary condition, and *Y/N* representing the possible evaluations of that condition. The leaf nodes, *L*, could either be actions, or classifications, depending on the context of the problem.

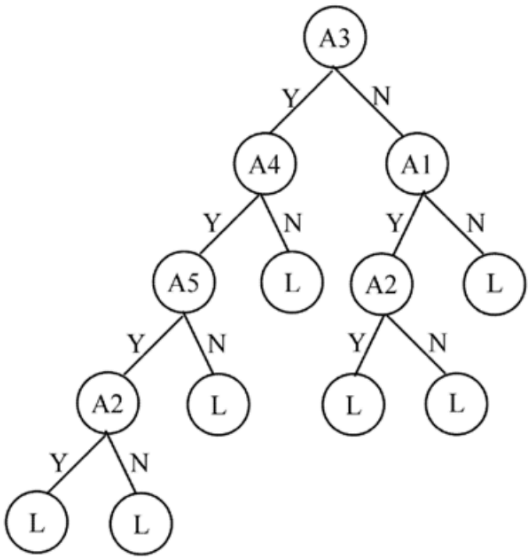

Figure 3: An Example of a Decision Tree [8]

For our problem, a decision tree is an easy and intuitive representation of a solution. Each square visible to the robot is a condition to evaluate on, and each leaf node represents an action to take given the set of condition evaluations that led to it (i.e. the state of the squares seen by the robot). Each square has 3 possible states: empty, is a wall, or contains a can. Figure 4 illustrates a small fragment of one possible decision tree that expresses a solution to this problem.

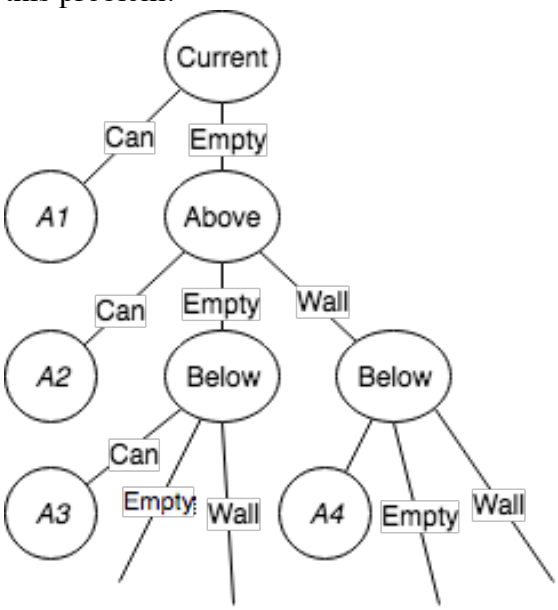

Figure 4: Decision Tree Extract for Robot Cleaner Control Scheme

Clearly, a full decision tree for the robot, given the number of squares it can see, and the number of possible states for each square, would be very large. Building such a decision tree by hand would be tedious and difficult, and selecting the right actions for a given situation may not be an easy choice. In this paper, we propose a genetic algorithm that evolves such a decision tree, by selecting the appropriate actions *Ai* for all *i*.

With 5 squares, 3 states per square (can, wall or empty), and with the assumption that all combinations of states are valid (which is not the case, as there will never be a wall in the current square, or in left/right and above/below squares simultaneously), it can be easily shown that there are 243 possible combinations of states (3^5). Thus, the full decision tree for the robot contains 243 actions *Ai* that must be specified. The selection of these actions determines the behaviour of the robot, and the quality of the solution.

We chose to give the robot 7 possible actions, and number them from 0-6. The actions are:

0. Move North
1. Move South
2. Move East
3. Move West
4. Do Nothing
5. Pick Up Can (in current square)
6. Random Action (i.e. randomly pick action between 0 and 5)

It should be noted that the approach discussed in this paper is suitable for problems where the structure of the tree is known in advance, but not the values of the leaf nodes. The genetic

algorithm discussed herein is a method of determining what values to ascribe to the leaf nodes, rather than the structure of the decision tree.

2.2 Simulation and Scoring System

In order to analyse the quality of potential solutions, we developed and used a simulation test control schemes. The simulation uses a 10x10 grid of Boolean yes/no values, with each value representing the presence/absence of a can in its associated square. At the start of each simulation, a robot is placed in the top left-hand corner (position 0,0), and the grid is randomly filled with cans, with a 50% chance of a can being placed in each square. The robot is then loaded with a control scheme, and executes 200 moves (following that control scheme). At the end of the 200 moves, a score is produced that represents the performance of the robot.

The scoring scheme used by the simulation is:

- The robot starts off with 0 points
- The robot gains 10 points for every can it picks up
- The robot loses 5 points for every collision with a wall
- The robot loses 1 point for every attempt to pick up a can in an empty square

The robot is 200 moves, as this is just enough moves for a perfect solution to pick up every can in the grid (1 move to reach a square, and another move to pick up the can in that square).

As we chose the probability of a can being in each square to be 50%, the grid is, on average, filled with 50 squares. Thus, on average, the theoretical maximum number of points attainable by any solution is 500. This provides a useful benchmark for analysing the performance of different solutions. As mentioned earlier, a solution is effectively a set of actions.

3. Algorithm

At a high level, the genetic algorithm works by following the below steps:

1. Initialize solutions randomly until population is full
2. Evaluate fitness of each solution in current population using 15 simulations
3. Select parents using k-way tournament selection
4. Randomly mutate offspring
5. Complete steps 3-4 until a new population of offspring is complete
6. Make the new population (of offspring) the current population
7. Repeat steps 2-6 until a fixed number of times

The population size is fixed at 200 individuals, a decision motivated by Haupt's finding that smaller population sizes and larger mutation rates lead to faster convergence [9].

Rather than evolving one population over time, and continually updating it, an entirely new population of children is formed with each generation. This has the downside of potentially losing good solutions if they're not selected as parents. However, it does result in more exploration of the search space.

3.1 Solution Encoding

The first and most important decision in applying a GA to any problem is that of how to represent possible solutions to the problem. That is, what encoding to use for each individual. The wrong choice of encoding can completely cripple the performance of the GA.

For this problem, possible solutions are represented as a string of 243 numbers, with each number being between 0 and 6. As described in Section 2.1, there are 243 possible combinations of states, and 7 possible actions (0-6). Thus, each position in the string (i.e. position 1) corresponds to a specific combination of states. For a given solution, the number at that position in the string denotes what action the robot should take for that combination of states.

For example, consider the solution below:

```
154355253221323256051352116652311151216320254350351055255223126254230254056321021054052200224226364056306256011610150014323312055355236050235054055335256053123133154354254624124256356353145156113154126025255150665545346156064226501306604454561
```

Position 1 in the string corresponds to the situation where the current square is occupied with a can, and all adjacent squares are empty. In the above solution, position 1 contains the number 5, which corresponds to picking up a can. Thus, according to this control scheme, when the robot is on top of a can and all adjacent squares are empty, it will pick the can up.

3.2 Initialization

The first population of solutions is generated completely randomly. For each solution, a string of 243 completely random numbers (between 0 and 6 inclusive) is generated. We found that this method of initialization was both simple (to implement) and effective.

3.3 Fitness Function

Fitnesses are evaluated by running the solutions in a simulation, and scoring them according to the scoring system described in Section 2.2. For each solution, the simulation is run 15 times, and the average score of the 15 runs is returned as the solution's fitness.

Averaging the scores of 15 runs (as compared to simply performing one simulation) provides some robustness against noise, as the grid is filled randomly with cans. As such, a solution may underperform or over perform in certain grid layouts due to simple chance. For this reason, each solution is tested 15 times, to provide some robustness against this, and to reduce noise in the fitness function. The more simulations each fitness evaluation runs, the more resilient the fitness function is to noise. However, a larger number of simulations makes the fitness function (and thus the entire GA) significantly more compute intensive, and thus increases runtime significantly. Hence a balance was required here between speed and resilience.

3.4 Parent Selection

The two main methods used for parent selection in practice and in literature are roulette wheel selection and k-way tournament selection. Due to the possibility of negative fitness values, roulette wheel selection was deemed unsuitable for the application. As such, k-way tournament selection was chosen for parent selection. This works by randomly selecting k individuals from the population, and then selecting the best of them (ranked by fitness) as the parent. K is a parameter that can be adjusted to affect the randomness of the selection: a

higher k value results in less randomness (more exploitation), and a lower k value results in more randomness (more exploration). A suitable value that balances exploration and exploitation had to be found through experimentation.

3.5 Crossover

In this algorithm, 2 parents produce 2 children. Crossover is the method used to form the children form the two parents. It is performed by simply selecting a random value between 0 and 242, and cutting the two parents in half at that point.

The first half of the first parent, and the second half of the second parent, become one child. Similarly, the second half of the first parent, and the first half of the second parent, become the second child. This is illustrated in Figure 5. This method of performing crossover never produces any invalid solutions for this problem.

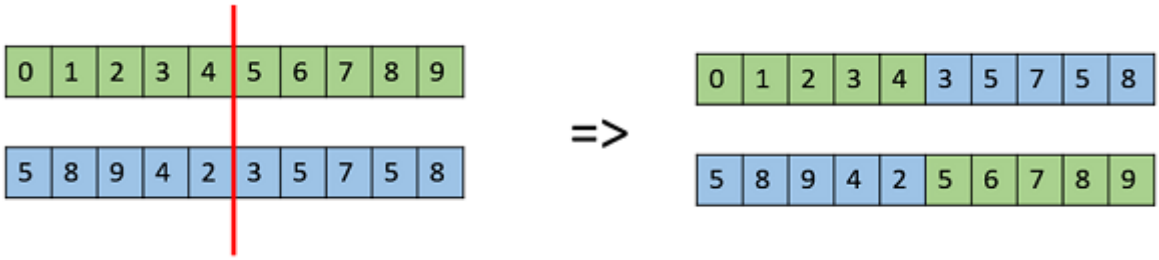

Figure 5: Crossover Using Two Parents

3.6 Mutation

Mutation is performed after crossover, and involves the alteration of a solution's characteristics in some way. The purpose of mutation is to introduce variance, and allow for the exploration of a wide region of the solution search space. Children are randomly selected with a low probability (initially 1/6, before experiments showed ¼ to be a better value), and mutated. Mutation is performed by simply selecting a random point in the string, and randomizing the value at that point (between 0 and 6, inclusive). An example of this form of mutation is illustrated in Figure 6. This method of performing mutation never produces any invalid solutions for this problem.

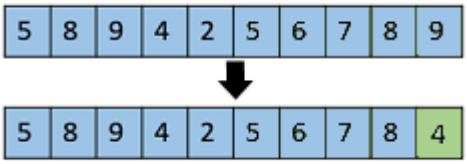

Figure 6:Mutation of an Individual

4 Experiment Suite 1

The first set of experiments we conducted aimed to find an effective set of parameters to maximize the fitness at which convergence occurs, and to minimize the time required for convergence. A preliminary run of the algorithm produced satisfying results, but revealed room for improvement. When run for 7,000 generations, the algorithm produced a final

solution that scores, on average, 360 points, picking up roughly 35-36 of the 50 cans in the grid. Figure 7 illustrates the growth of fitnesses over the 7,000 generations of evolution.

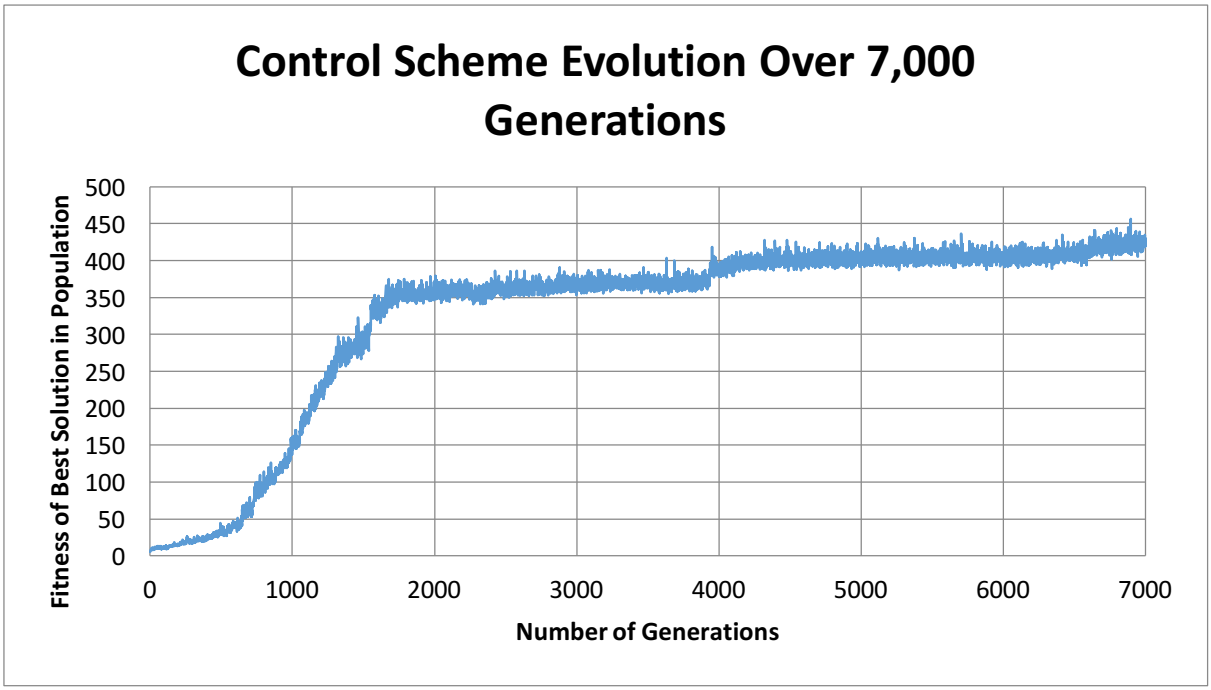

Figure 7: Control Scheme Evolution Over 7,000 Generations (Initial Run)

While these are encouraging results, the average score of 360 is still quite far off the theoretical optimum of 500. Rather than run the algorithm for a larger number of generations, we conducted a series of experiments to fine tune the parameters used by the genetic algorithm, and improve the rate and point of convergence.

For every experiment, each alternative was tested 5 times, up to a generation limit of 1000. The results were then compared to identify the parameters that consistently gave the best performance.

4.1 Parent Selection

The first experiment conducted aimed to identify:

1. The best method of parent selection for this problem
2. The best parameters to use for the chosen parent selection method

For the first choice, the three main alternatives compared were:

1. Roulette Wheel Selection
2. K-way Tournament Selection
3. Random Selection

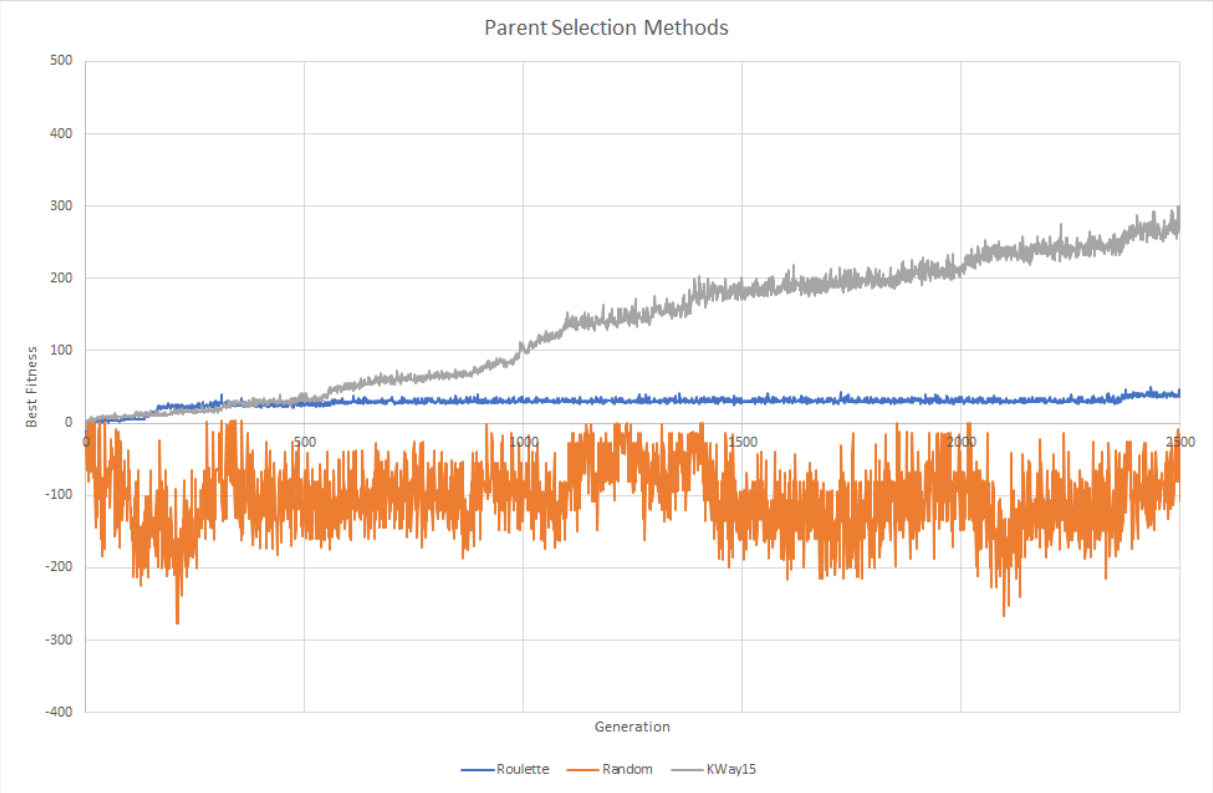

Figure 8: Parent Selection Method Effect on Convergence

The results from these experiments, which can be seen in Figure 8, suggested that k-way tournament selection provided the fastest rate of convergence.

Unsurprisingly, random selection fails to achieve any improvement over time.

Roulette wheel selection improves at an extremely slow rate, likely due to its inability to handle negative fitnesses, which are possible in this problem (as bad solutions that crash into walls often can produce negative scores). In order to allow for the use of roulette wheel selection in this problem, the method had to be adapted somewhat. Negative fitnesses are all simply zeroed, and their solutions thus occupy no chunk of the wheel, and have no chance of being selected. It is unsurprising, then, that this parent selection method performs slowly, as the elimination of bad solutions from parent selection reduces exploration significantly, and violates the careful balance between exploration and exploitation required for proper and timely convergence.

The next experiment conducted compared the performance of several k-values. The results, which are too extensive to display here (due to the large number of k values tested), suggested that a k value of 15 was ideal for this problem.

4.2 Mutation

The mutation experiments aimed to identify:

1. The probability of mutation (for each individual) that maximized convergence rate and quality
2. The best mutation method

We experimented by varying the probability of mutation, and varying the mutation method to observe how these changes impacted the convergence over 1000 generations. Our experiments consisted of:

1. Using ½ chance of mutation
2. Using ¼ chance of mutation
3. Using 1/6 chance of mutation
4. Using 1/8 chance of mutation
5. Mutation only on the first half of an individual's genome
6. Mutation only on the second half of an individual's genome
7. Mutation at two points of the individual's genome
8. Swapping two different points in the individual's genome

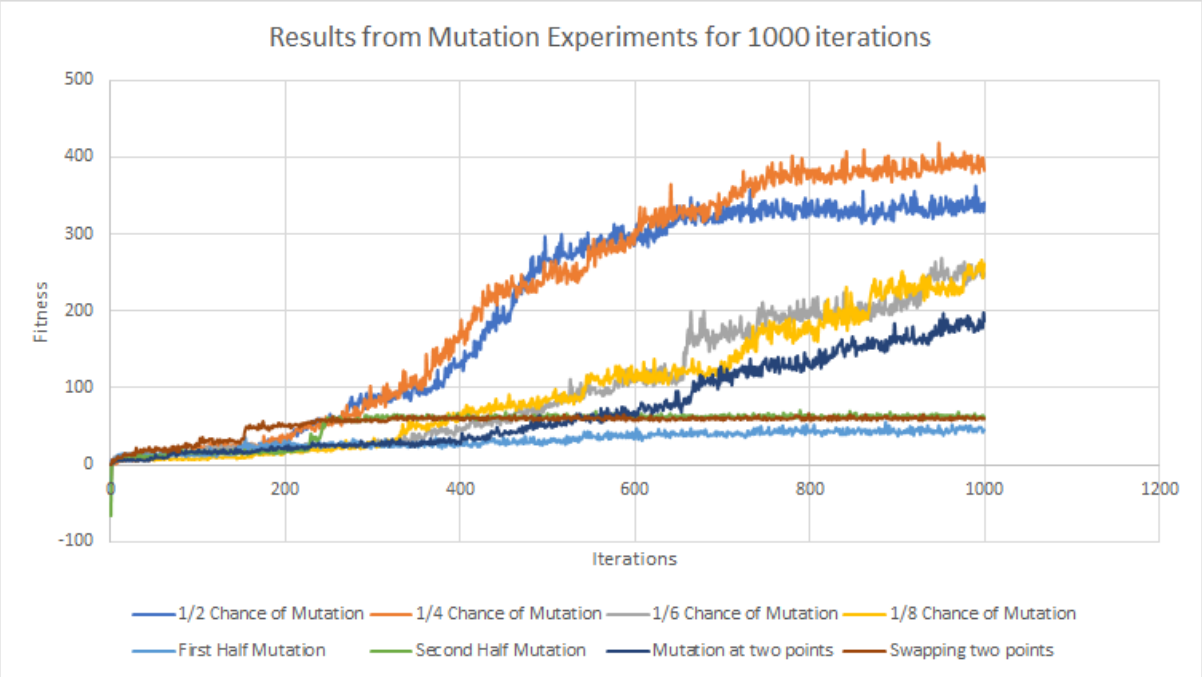

Figure 9: Mutation Experiments and the Effect on Convergence

The results of the mutation experiments are shown above in Figure 9. The initial run of the GA (Figure 5) used a 1/6 chance of mutation for each child. As can be seen above, this produces only average results compared to the rest of the variants assessed.

Only mutating points in the first or second half of the genome resulted in poor results, with the convergence rate for both of these methods being very slow, and the fitnesses never exceeding 100 (when run over 1000 generations).

Mutating two points instead of one in the individual's 243-character string also did not seem to demonstrate any improvement compared to the initial solution. The same can be said for swapping two points with one another. Both of these methods produced slower convergence rates compared to the initial run.

With regards to mutation probability, the results suggest that using a ¼ chance of mutation is ideal. This probability produced the most consistent results, and also the greatest fitnesses; it seems to be the probability that best balances exploration and exploitation for this problem. The results also suggested that the initial choice of mutating only one point in any individual's genome produced the best results, when compared with the variants evaluated.

4.3 Crossover

The crossover experiments aimed to identify the number of crossover splice points that maximised population fitness. These experiments included running the GA with one split point through to ten split points (5 runs for each variant). The results of the crossover experiments are shown below in Figure 10.

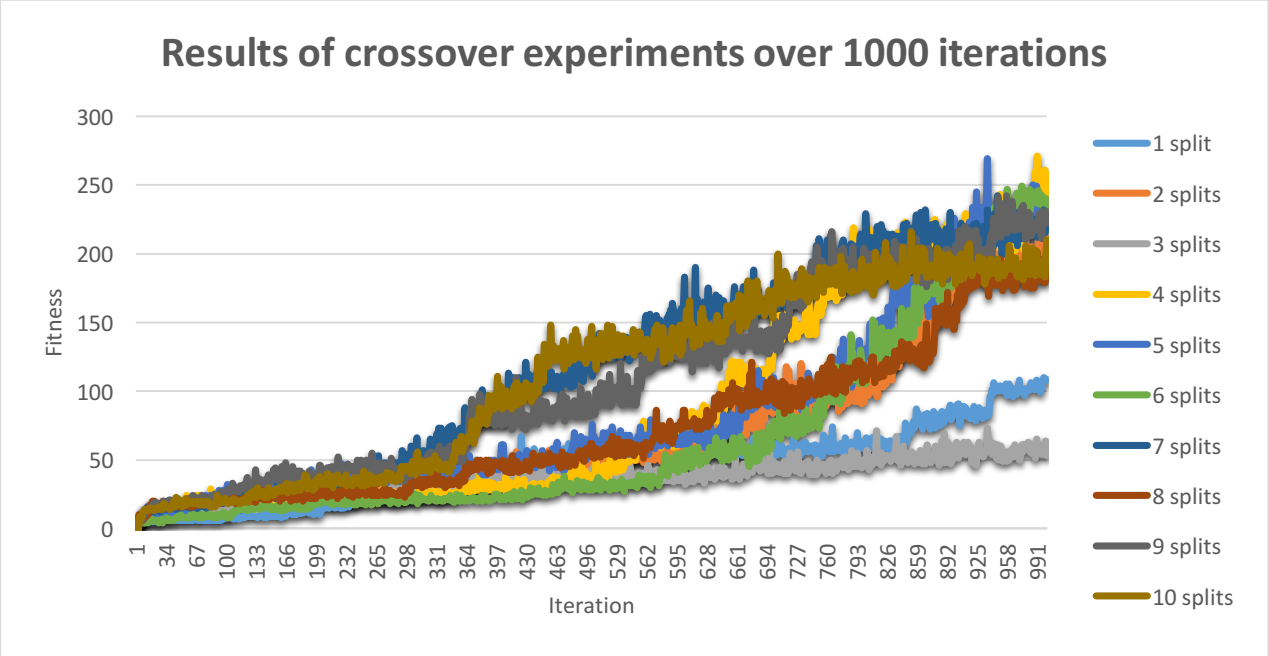

Figure 10: Crossover experiments, and their effect on population fitness and algorithm convergence

These results indicate the most crossover experiments have a similar outcome for 1000 iterations. It should be noted that a higher number of crossover points (such as 7, 9 and 10) converge on high fitness values faster than lower numbers, although smaller values (such as 2, 4 and 6) were able to match or beat the fitness values when near the iteration limit of the tests. Following this trend, we determined that the crossover point was ideal at 2, due to the large increase in fitness over the original 1 split point, and the greater increase in fitness at higher iteration values. 2 points also achieved the most consistent results over the 5 runs, when compared to the higher values.

4.4 Results

To analyse the results achieved by the GA, we needed some benchmarks against which to compare. For this reason, we designed 2 hand-coded solutions and measured their results.

4.4.1 Random Solution

The first solution designed simply moves randomly until it sees a can underneath it, and then picks it up. On average, this solution only scores about 90 points. It tends to collect roughly 20/50 cans on the grid, but collides with many walls.

4.4.2 Hand Coded Solution

The second solution designed never collides with a wall, picks up cans it sees, moves towards cans that are visible, and otherwise moves in arbitrary directions. On average, this solution only scores about 70 points, gathering on average 7/50 cans in the grid.

4.4.3 GA Solution

After re-running the GA with all of the optimized parameters that were discovered using the above experiments, the GA converged on a near optimal solution much faster. Compared with the first run, which took 7000 generations to find a solution that averaged 360 points, this run took roughly 5700 generations to find a solution that averages about 470 points. A trace of this run can be seen in Figure 11.

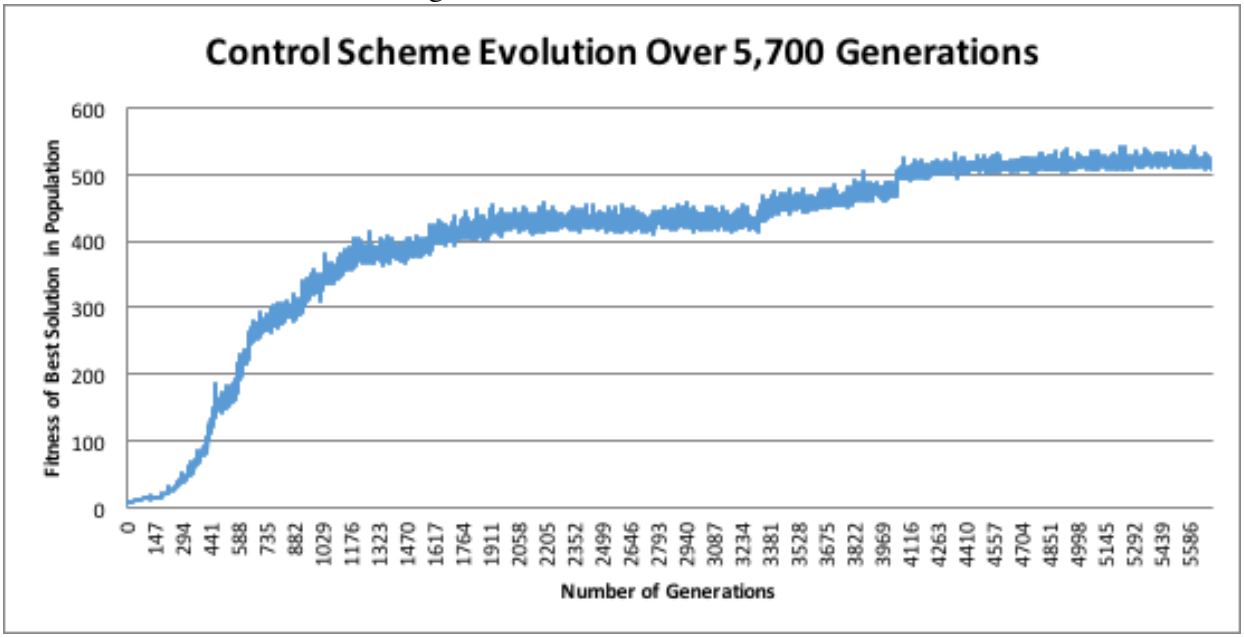

Figure 11: Optimised GA evolution over 5700 generations

This solution is nearly optimal. A score of 470 points suggests that the robot picks up at least 47 cans in the grid. As there are, on average, 50 cans placed in the grid, this control scheme achieves scores that are nearly perfect, and completely surpass the two hand designed solutions described above.

5 Experiment Suite 2

The second set of experiments we conducted aimed to identify how well the solution functioned when tested under different conditions to those under which it was trained. We tested the solution's robustness to different grid sizes, and to different can densities within the grid.

5.1 Grid Size Variance

This experiment aimed to observe how increasing the size of the grid affected the performance of the GA's solution. As the GA's fitness function only ever ran its simulations on 10x10 grid sizes, the goal here was to observe how robust the solution was to different types of grids.

The results of this experiment can be seen below, in Figure 12. As the grid size increases, the solution moves further away from optimality, as can be seen.

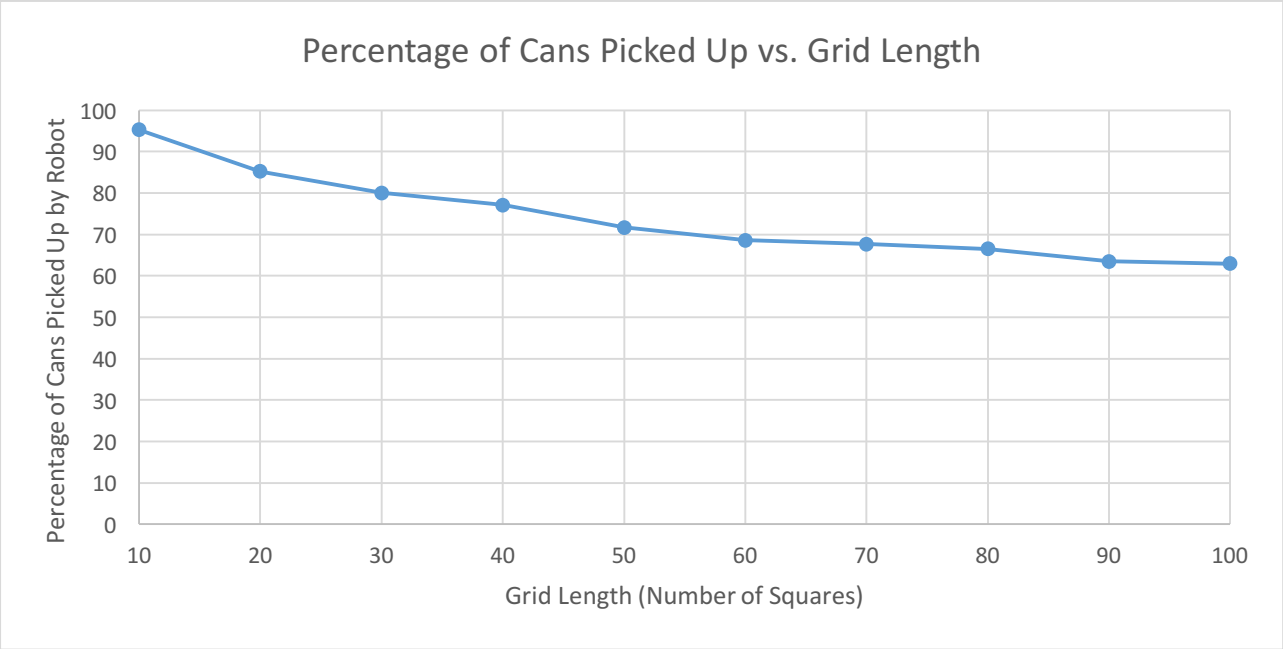

Figure 12: Performance of Solution as Grid Size Increases

The solution collects less of the cans in the grid as the grid size increases, suggesting that the solution does not have ideal robustness to this factor. This is perhaps a result of the fact that the genetic algorithm was run with a fitness function that always tested solutions with a constant grid size (of 10).

5.2 Can Sparsity Variance

The next experiment aimed to observe how changing the density of the cans in the grid affected the performance of the solution. The GA's fitness function always ran simulations that used a 50% chance of a can in each square. As such, we wanted to observe how robust the solution was to different can densities. The results can be seen below in Figures 13 and 14.

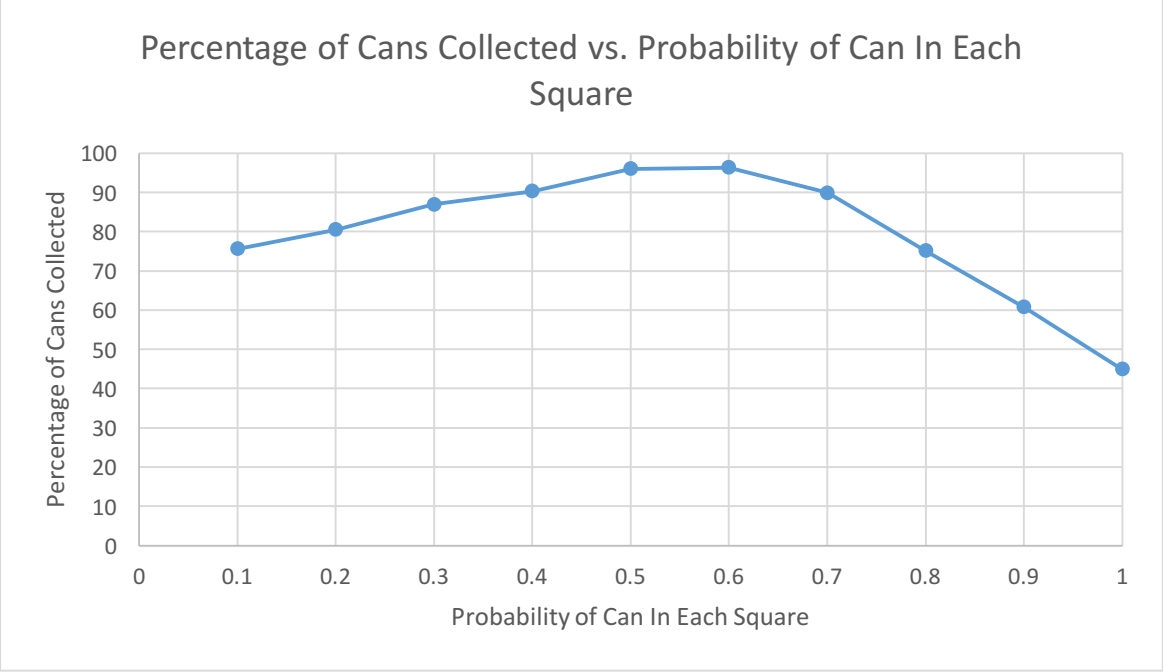

Figure 13: Robustness of Solution to Different Can Densities

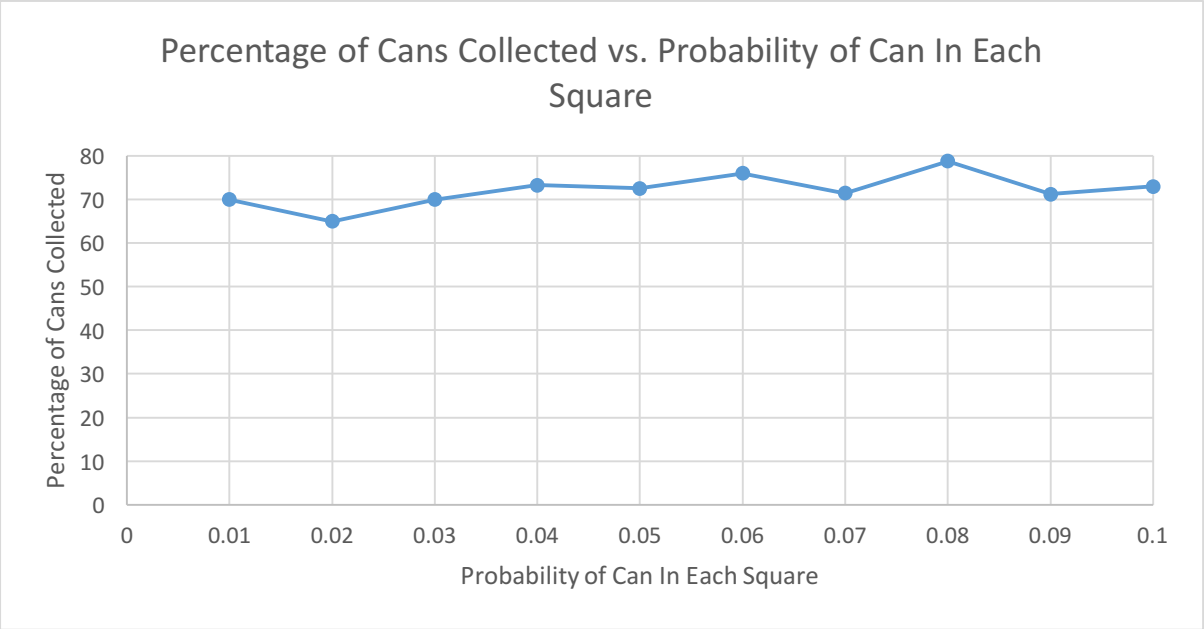

Figure 14: Robustness of Solution to Different Can Densities

As can be seen above, the solution performs best with a 50% probability, under which it was trained. Either an increase or a decrease in the probability leads to degradation in the performance of the solution. Surprisingly, however, increasing the probability leads to a larger degradation in the performance of the solution than decreasing the probability. This result is somewhat surprising, as solutions guided by random movements would be expected to perform better in denser grids, and worse in sparser grids. It suggests that the GA developed tactics that exploited the specific probability of 50%, and perhaps don't work well in dense grids.

The GA's tactic of leaving cans behind as markers (discussed in Section 6) is perhaps part of the reason for this large degradation in performance as the probability increases, as it leads to lots of wasted movements. Furthermore, as the grid becomes denser, each movement becomes more valuable, as it takes more moves (theoretically) to collect all the cans (there are more of them). The GA's solution, having been trained with a probability of 50% (where some leeway exists and some wastefulness is permitted), likely does not work well with this new factor in tow.

6 Solution Analysis

While it is difficult to determine the strategies that were encoded into the GA's solution, observations from the simulation's graphical interface showed that the robot had evolved to circle around the outer walls and work its way toward the centre of the grid. In addition, the robot would often pass over cans that it was able to pick up in favour of "exploring" nearby cans. This allowed the robot to return to these cans later (as they would still be visible on the path back) and to pick up more distant cans that would no longer be adjacent if the robot had collected an earlier can.

7 Future Work

While the results achieved by the GA were impressive, we felt that there were several avenues for improvement and extension. As shown in section 5, the solution found by the GA

does not have a desired amount of robustness to different environmental conditions, such as grid layout and can density. This is likely a result of the GA having only explored for solutions on 10x10 grid sizes, and with a fixed 50% chance of there being a can in each square. Had these factors been randomized more during the evolutionary process, the final solution may have had more robustness to such conditions.

Due to computational limitations, we were unable to perform the extent of parameter testing that we desired. We had to impose a strict generation limit of 1000, and were only able to run 5 GA instances for each parameter choice. Furthermore, the range of values we could test was severely limited. Executing more tests, and more thoroughly exploring the space of possible parameter choices, may have increased the reliability and accuracy of the results, and helped to reduce the randomness factor inherent to GAs when examining the results to select good parameters.

Finally, this paper only explored the use of genetic algorithms to find values for leaf nodes in a decision tree. A more valuable and general-purpose genetic algorithm would be one that can evolve the decision tree itself (i.e. its structure). This would enable the application of these techniques to more complex problems, where the decision tree structure may not be known in advance, or where it may not be intuitively obvious.

8 Conclusions

This paper demonstrated an alternative approach to decision tree induction using genetic algorithms. By implementing and expanding previous work in this field [7], a decision tree for a can-cleaning robot was evolved using a GA. Through extensive testing, the GA's parameters were optimised to produce decision trees that far exceeded hand-coded solutions. While these solutions were found to be less optimal when exposed to varying grid sizes and can densities, expanded testing to include these situations will improve robustness.

9 References

[1] J. Funge and I.Millington, "Artificial Intelligence for Games 2nd Edition", 2nd ed., 2009, pp. 320
[2] S. Russel and P. Norvig, "Artificial Intelligence: A Modern Approach", 3rd ed., 2010, pp. 716
[3] M. Mitchell, "An Introduction to Genetic Algorithms", 1999, pp. 11
[4] Wall B W, "A Genetic Algorithm for Resource Constrained Scheduling", PhD Thesis, Department of Mechanical Engineering, Massachusetts Institute of Technology, USA, 1996
[5] A. Colorni, M. Dorigo, and V. Maniezzo, "A genetic algorithm to solve the timetable problem," Politecnico di Milano, Milan, Italy, Tech. Rep. 90-060 revised, 1992.
[6] B. Tomoiaga, M. Chidris, A. Sumper, A. Sudria-Andreu, and R. Villafafila-Robles, "Pareto Optimal Reconfiguration of Power Distribution Systems Using a Genetic Algorithm Based on NSGA-II", Energies, vol. 6, pp. 143901455, 2013.
[7] M. Mitchell, "Complexity: A Guided Tour", 2009
[8] Z. Fu, B. Golden, and S. Lele, "A Genetic Algorithm based approach for building accurate decision trees," *INFORMS Journal on Computing*, vol. 15, no. 5, pp. 3-23, 2003.
[9] R. L. Haupt and S. E. Haupt, "Optimum population size and mutation rate for a simple real genetic algorithm that optimizes array factors," *Applied Computation Electromagn. Soc. J,* vol. 15, no. 2, pp. 94-102, July 2000

[10] M. Kretowski and M. Grzes, “Global learning of decision trees by an evolutionary algorithm.” In *Information processing and Security Systems,* 2005.
[11] J. Bala, J. Huang, H. Vafaie, K. DeJong, and H. Wechsler, “Hybrid learning using genetic algorithms and decision tress for pattern classification”, in *Proceedings of the 14th International Joint Conference on Artificial Intelligence, Montreal, Canada,* 1995, pp. 719-724.