‘Thanks, ants. Thants.’ [1]: Applying genetic algorithms to simulated ants to produce higher-fitness generations

Ben Goldsworthy, 33576556
Computer Science (with Industrial Experience) MSci Hons

Abstract—Genetic algorithms can be used to automated improvement of a program or problem solution, as well as potentially uncovering successful variants that a human would have been unlikely to craft by hand. In this paper this theory is applied to simulated ants, and on a mass scale that allows the oversized effects of fine-tuning small components of the algorithms to be made obvious. Starting from random ant strings, the algorithm eventually produces an ant with a fitness of $88.\overline{8}$ %, which would in theory be possible to continue mutating to 100%. The paper then discusses various avenues of future improvement on the algorithm.

Keywords—artificial intelligence, genetic algorithms, evolutionary programming, GNU Octave [1]

1 Introduction

Genetic algorithms are a relatively young field of programming, but have made a big impact, producing everything from spacecraft antenna [2] to automated program bugfixers [3]. They work using concepts taken from evolutionary biology such as natural selection and fitness. Algorithm 1 provides a simple demonstration of how a genetic algorithm is applied to an input program, that has already been encoded in one of a number of ways (e.g. binary encoded, tree encoded, etc.). Each of these permutations is referred to as a ‘chromosome’, and can be changed via mutation (i.e. randomly changing parts of the chromosome) or crossover (i.e swapping portions of two parent chromosomes to produce a new child chromosome). These chromosomes can then be tested and ranked based on some predetermined metric, with the weakest discarded and the strongest run through the process again. As time goes on, the fitness of the produced chromosomes should tend constantly upwards.

1. GNU is a free software alternative to MATLAB that provides almost the full functionality of MATLAB. For more on free software, please visit `www.gnu.org/philosophy/free-sw`

Algorithm 1 Basic genetic algorithm

```
Require: initial population
Require: max generations
  while curr. generation < max. gens do
    run population and return fitness scores
    for all chromosomes in population do
      select strongest for new population
      if small probability = true then
        apply mutation or crossover
        operations to some chromosomes
      end if
    end for
  end while
```

1.1 AntProg

This report concerns the program ‘AntProg’. This program consists of the files `cw2_goldsworthy_ben.m` (see Appendix C) and `runGeneration.m` (see Appendix D), which were written by myself, and `simluate_ant.m` [5].

`simulate_ant.m` takes a 30-digit value-encoded representation of an ant that is then simulated a 2D represenations of a John Muir trail,

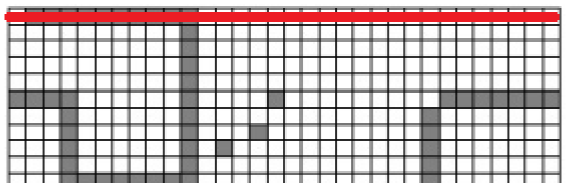
[4] which can be viewed in Appendix A). The ant starts in the top-left square, facing right. Each triplet of values represents a state, from 0-9. The first digit determines the ant's next movement, whilst the second and third determine which state it should jump to next, dependent on whether the square ahead is empty or contains food, respectively. For example, the string '121141...702' would correspond to

State 0	1	2	1
State 1	1	4	1
		⋮	
State 9	7	0	2

where state 0 moves forward one cell, then goes to state 2 if the square ahead is empty or state 1 if it contains food. If an ant enters a cell containing food, it eats it and gains one fitness point.

1.2 Initial ant

The simplest valid string that can be given to the program is '100100100100100100100100100100'. This ant would move forward one square, then constantly return to state 0 regardless of the state of the adjacent square. With 200 states, this equates to 200 moves forward, and a fitness score of 10. Below is the path taken by the ant:



Obviously, this is far from an optimum ant. However, it does demonstrate how the program works.

2 The Genetic Improvement algorithm

Appendix D contains the genetic improvement algorithm I have implemented. This algorithm takes a population p of n chromosomes and mutation and crossover probabilities $P(M)$ and $P(C)$. It returns the results of $f(p_1)\dots f(p_n)$, a new generation p' and the best-performing string in p.

For populating p' the algorithm uses rank selection to choose chromosomes from p, as this helps to increase diversity in samples that can see large variations in fitness values (as opposed to plain 'most-fit' selection or roulette wheel selection). Elite selection of the top 10 % best-performing chromosomes is also utilised as a heuristic to increase the performance of the algorithm as the strongest chromosomes are no longer potentially discarded each generation. The remaining 90 % of chromosomes then have a small chance of having a multi-point mutation or crossover (with a random other chromosome in the population) operation applied to them.

2.1 How the probabilities were chosen

One of the most powerful ways of tuning the algorithm is by changing the values of $P(M)$ and $P(C)$. Initially, these were hard-coded as 0.2 and 0.8, respectively. Once the genetic improvement algorithm was working, I implemented a mass-analysis framework to perform m runs of n generations each, with various random values for $P(M)$ and $P(C)$, and then compared the performances of each run using the fitness function

$$f(run_m) = f(n_m)_{max} \times (n - generation(f(n_m)_{max}))$$

or, the highest fitness score achieved times the number of generations minus the generation the score was achieved in.

I ran a 30×30 set of generations g. Initially, both probabilities could fall in the range $0, \dots, 1$. The graph seemed to suggest that a lower mutation probability paired with a higher crossover probability produced better results, so I halved the range of $P(M)$ and re-ran g. This suggested better results with a mutation probability around 0.3, so I re-ran g with $P(M) = \{x \in \mathbb{R} : 0 \leq x \leq 0.3\}$, producing a fairly even distribution with a slight correlation between performance and a crossover probability around 0.8. Thus, mutation and crossover probability values of 0.3 and 0.8, respectively, were chosen.[2]

2. All these plots can be viewed in full in appendix B.

2.2 Individual runs

I then began running individual runs of 50 generations and populations per generation of 30. These produced occasional high-fitness strings of $f(n) > 40$, but after running between fifteen and twenty none achieved $f(n) > 50$.

2.3 Multiple analysis

I re-used the multiple analysis framework I had developed for §2.1, this time with the fixed probability values. I ran it for 40×30 generations (with a population per generation of 25), which produced a single string s that achieved $f(n) > 50$ (69).[3] I then modified `cw2_goldsworthy_ben.m` to seed the initial population p_0 with half s and half randomised chromosomes (as s could just be an example of local maxima rather than a guaranteed path towards achieving a higher-fitness chromosome):

```
newPop = zeros(populationSize,30);

for i = 1:populationSize
   chromosome = zeros(1,30);
   if mod(i,2)
      for j = 1:3:30
         chromosome(j) =
            randi([1,4],1,1);
         chromosome(j+1) =
            randi([0,9],1,1);
         chromosome(j+2) =
            randi([0,9],1,1);
      end
   else
      chromosome = [1 1 9 3 3 9 3 5
         7 3 4 3 3 7 8
         206369325129410;
   end
   newPop(i,:) = chromosome;
end
```

I then ran that algorithm for 50×50 generations (with a population per generation of 30) - see figure 1. This produced the first $f(n) \geq 80$ ant, of string '119339357343378206369325129110'. Figure 2 shows the path the ant takes to achieve it's score of 80. When both sets of generations are added together, the ant was thus produced on generation 32.

3. '119339357343378206369325129410'

Fig. 1. The first $f(n) > 80$ generation

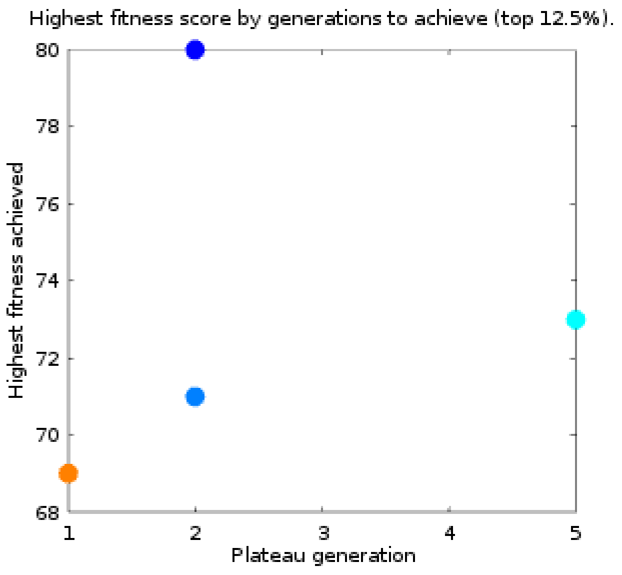
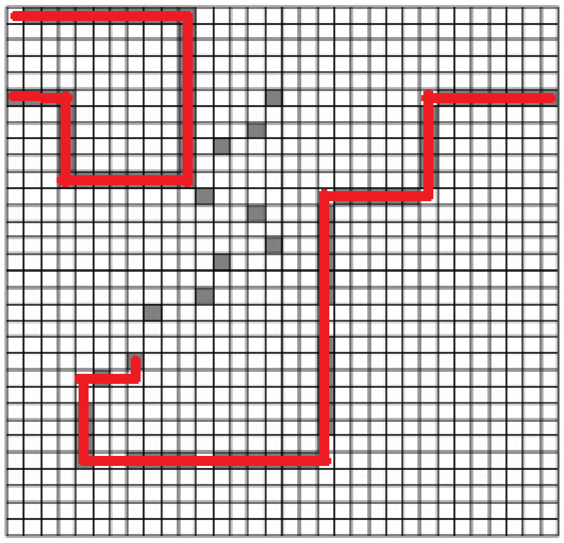


Fig. 2. Ant 119339357343378206369325129110's path



3 Conclusion

This paper will have shown that genetic algorithms can be applied to encoded program representations to produce a wide range of new programs, and that via the use of selection techniques (and a touch of randomness, to counter the impact of local maxima/minima) these techniques can trend upwards towards a given goal.

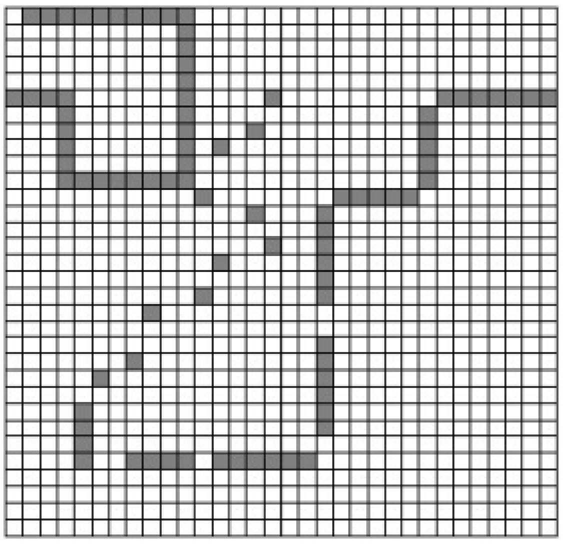
3.1 Future Improvement

As §2.1 touched on, changing the 'magic numbers' at the start of `cw2_goldsworthy_ben.m` (e.g. `populationSize`, `mutationProbability`, etc.) can have a big effect on the efficiency and performance of the algorithm. My next step would be to utilise the multi-generational analysis framework I have developed to run large numbers of tests with differing combinations of these until trends can be extrapolated. Perhaps doubling `populationSize` will prove about half as effective as doubling `generations`, and so computational time can be saved by prioritising reductions of the former rather than the latter, for example.

Another avenue of development is meta-genetic improvement [6], or using genetic algorithms to modify the way a program uses genetic algorithms. In this case, perhaps this could result in a program that learns certain patterns high-fitness strings tend to share and inserts these into a new generation's chromosomes where they appear to be forming naturally (as an additional chromosomal operation alongside mutation and crossover), theoretically speeding up the convergence to a maximum-fitness chromosome. This could also produce an effect like §2.3, where the algorithm was modified to give an improved initial state based on previous output, automatically.

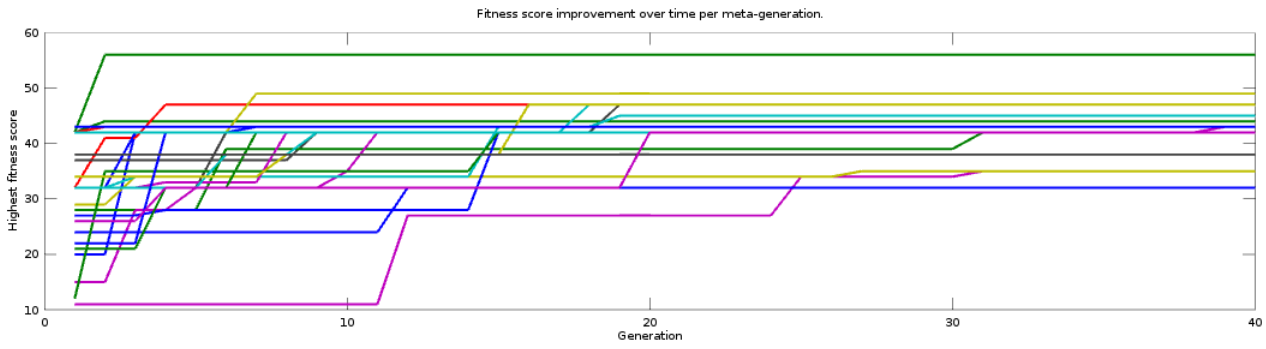
Finally, the approach to the current chromosomal operations could be modified. The current program uses only two types of operation – multi-point mutation and multi-point crossover – where others, such as uniform crossover, exist.

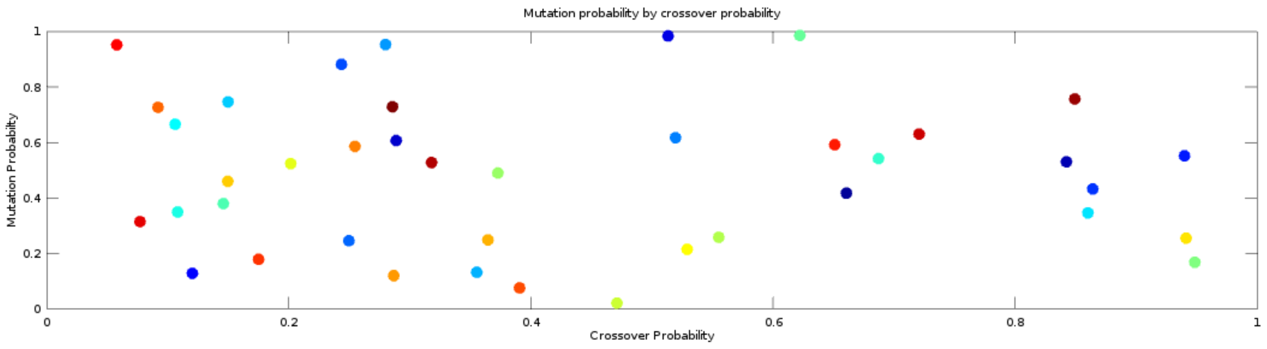
APPENDIX A
2D WORLD

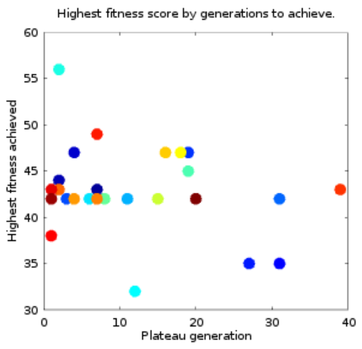


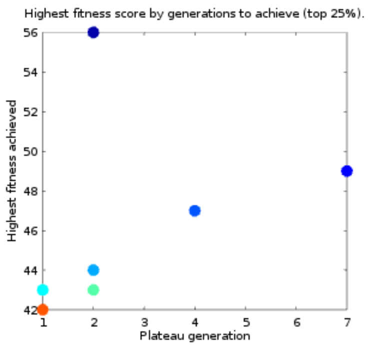
Appendix B
Probability Randomisation Results

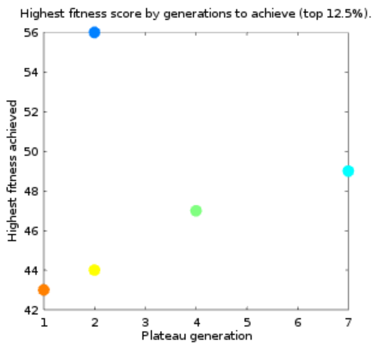
$$P(M) = [0 - 1]$$

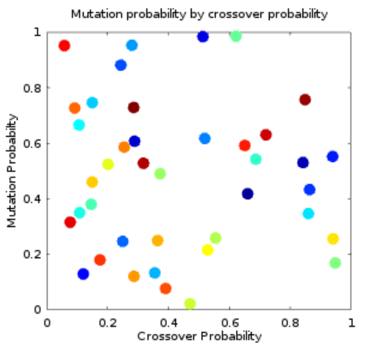


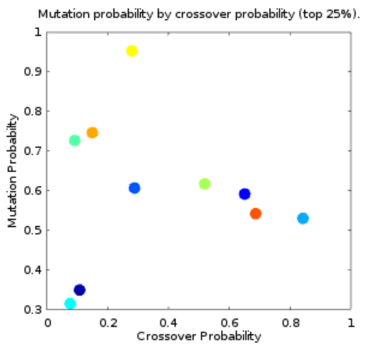


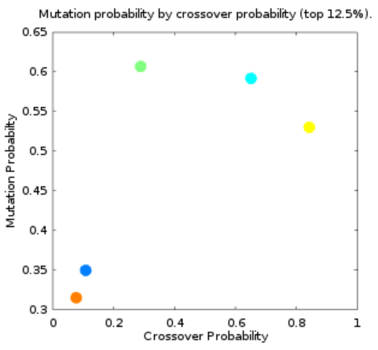




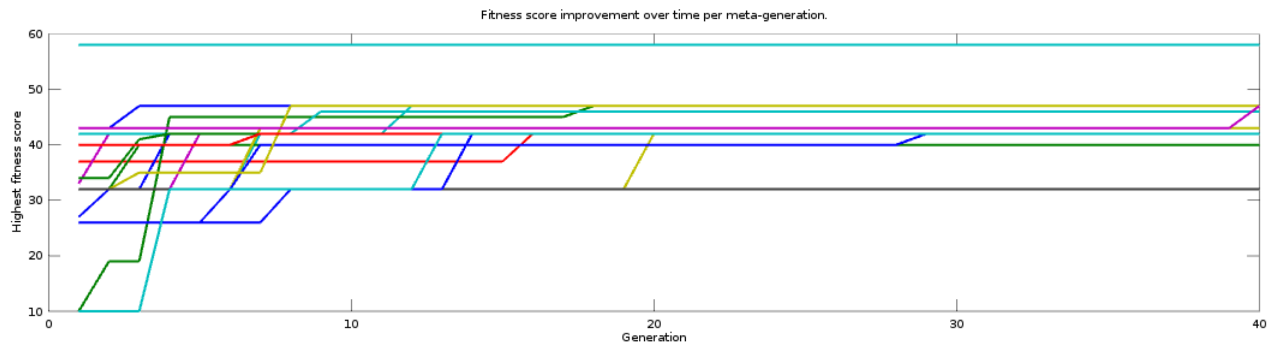


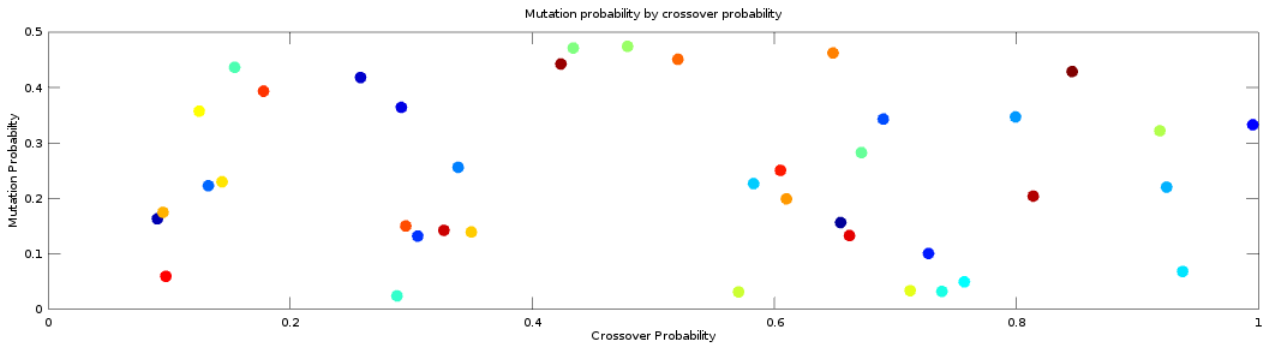


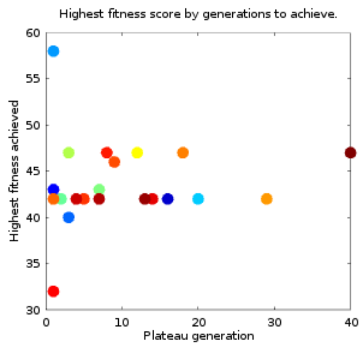


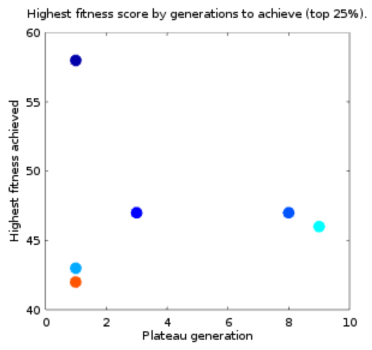


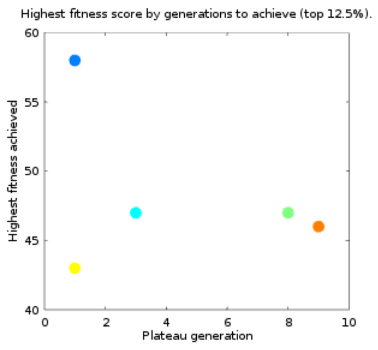
$$P(M) = [0 - 0.5]$$

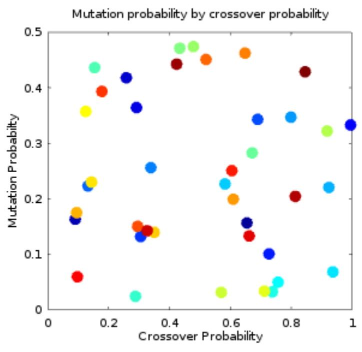

Fitness score improvement over time per meta-generation.
Highest fitness score
Generation

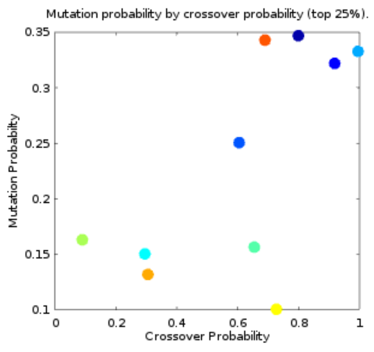

Mutation probability by crossover probability
Mutation Probability
Crossover Probability

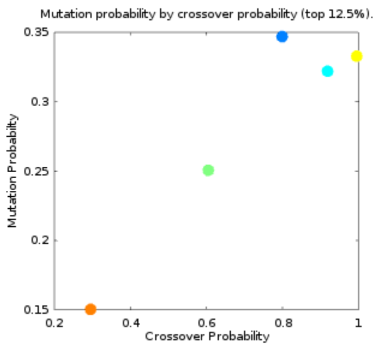

Highest fitness score by generations to achieve.
Highest fitness achieved
Plateau generation


Highest fitness score by generations to achieve (top 25%).
Highest fitness achieved
Plateau generation

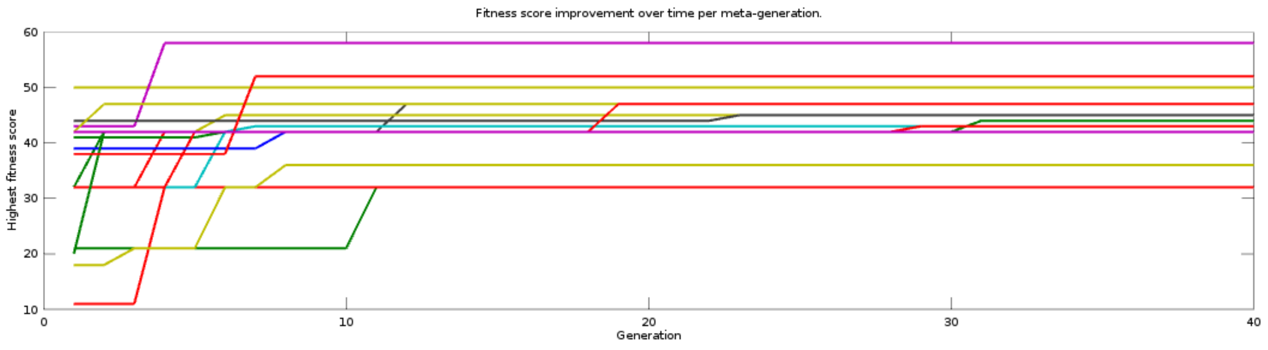

Highest fitness score by generations to achieve (top 12.5%).
Highest fitness achieved
Plateau generation

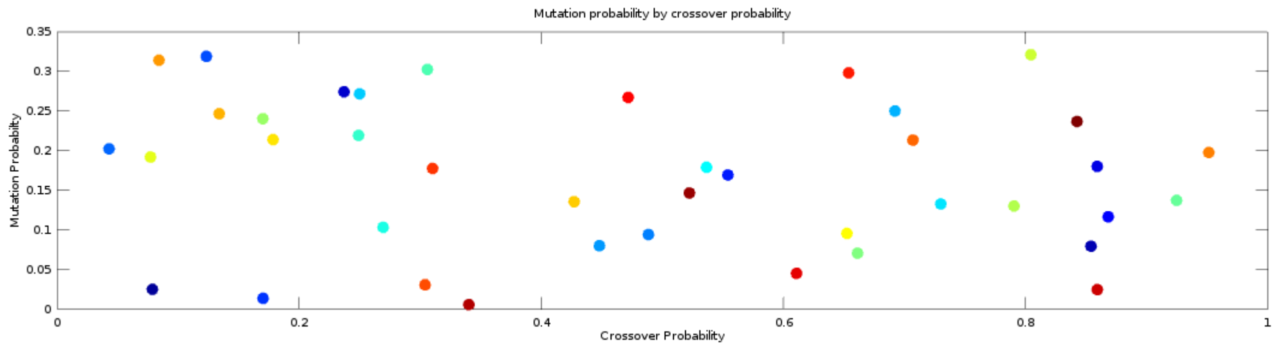

Mutation probability by crossover probability
Mutation Probability
Crossover Probability

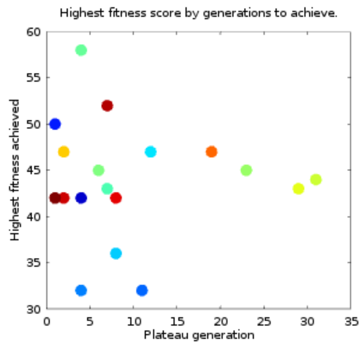

Mutation probability by crossover probability (top 25%).
Mutation Probability
Crossover Probability

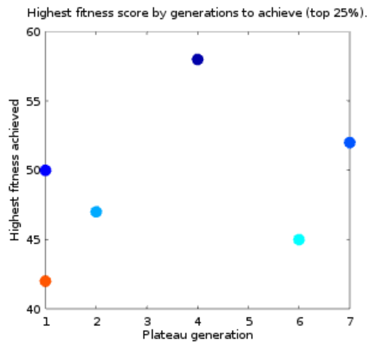

Mutation probability by crossover probability (top 12.5%).
Mutation Probability
Crossover Probability

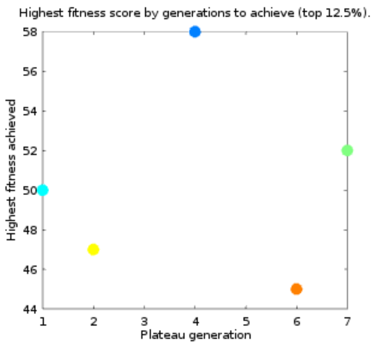
$$P\,(M) = [0 - 0.3]$$

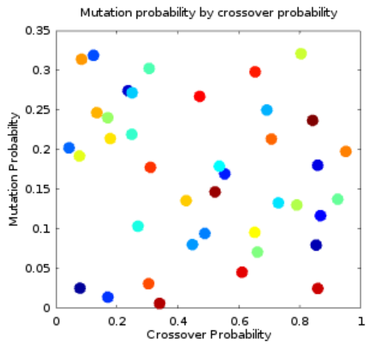

Fitness score improvement over time per meta-generation.
Highest fitness score
Generation

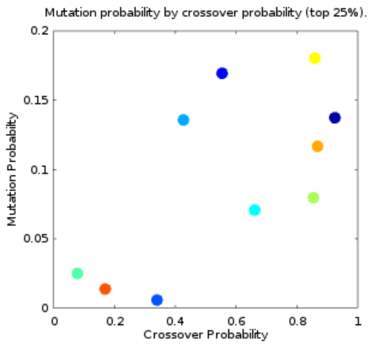

Mutation probability by crossover probability
Mutation Probability
Crossover Probability

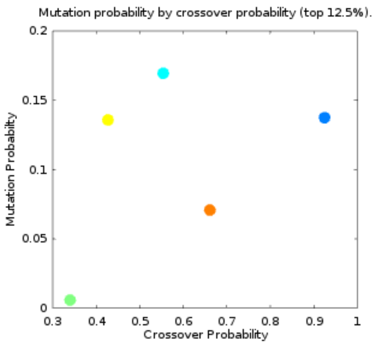

Highest fitness score by generations to achieve.
Highest fitness achieved
Plateau generation


Highest fitness score by generations to achieve (top 25%).
Highest fitness achieved
Plateau generation


Highest fitness score by generations to achieve (top 12.5%).
Highest fitness achieved
Plateau generation


Mutation probability by crossover probability
Mutation Probability
Crossover Probability


Mutation probability by crossover probability (top 25%).
Mutation Probability
Crossover Probability


Mutation probability by crossover probability (top 12.5%).
Mutation Probability
Crossover Probability

Appendix C

CW2_GOLDSWORTHY_BEN.M

```
% INITIALISATION

% ~change these variables for multi-generational analysis~
populationSize = 30;
generations = 50;
mGenerations = 50;
crossoverProbability = 0.8;
mutationProbability = 0.3;

% generates a random population of however many chromsomes
initPop = zeros(populationSize,30);
% initialises various matrices
fittest = zeros(mGenerations,generations+2);
highpoint = zeros(mGenerations,2);
bestTrails = zeros(mGenerations*2,200);
bestString = zeros(1,30);
bestStrings = zeros(mGenerations,31);

% GENETIC ALGORITHM

for y = 1:mGenerations
   % if we are doing a multi-generational analysis with changing
   % values, set them below
   if mGenerations > 1
      crossoverProbability = rand;
      mutationProbability = rand/3;
   end

   % assembles the initial population of 25 chromosomes with random alleles
   newPop = zeros(populationSize,30);

   for i = 1:populationSize
      chromosome = zeros(1,30);
      for j = 1:3:30
        chromosome(j) = randi([1,4],1,1);
        chromosome(j+1) = randi([0,9],1,1);
        chromosome(j+2) = randi([0,9],1,1);
      end
      newPop(i,:) = chromosome;
      initPop = newPop;
   end

   % runs the generation and gets the fittest results from it
   for x = 1:generations
      [resultPop, newPop, string] = runGeneration(newPop, crossoverProbability,
          mutationProbability);

      fittest(y,x) = resultPop(1,31);
      % records the highest points achieved across all generations for
      % use in multi-generation analysis (if applicable)
      if fittest(y,x) > highpoint(y,1)
        highpoint(y,1) = fittest(y,x);
        highpoint(y,2) = x;
        bestString = string;
      end
   end

   % gets the best result from this m-generation, and the probability
   % conditions that caused it
   bestStrings(y,:) = [bestString highpoint(y,1)];
   fittest(y, generations+1) = crossoverProbability;
   fittest(y, generations+2) = mutationProbability;
end

% plots the improvement lines of each m-generation, and the colour scores
% of each below (if performing multi-generational analysis)
figure
if mGenerations != 1
  subplot(2,1,1)
end
plot(fittest(:,1:generations)','LineWidth',2),
xlabel('Generation'),
ylabel('Highest fitness score'),
if mGenerations == 1
   title('Fitness score improvement over time.');
   sprintf('%d', bestString)
else
   title('Fitness score improvement over time per meta-generation.');

   subplot(2,1,2)
   scatter(fittest(:,generations+1),
      fittest(:,generations+2),10,jet(length(1:mGenerations)),'filled');
   xlabel('Crossover Probability'),
   ylabel('Mutation Probabilty'),
   title('Mutation probability by crossover probability');

   % if performing multi-generation analysis, plots graphs of improvement
   % rates across all m-generations
   figure

   subplot(2,3,1)
   scatter(highpoint(:,2), highpoint(:,1),10,jet(length(1:mGenerations)),'filled'),
   xlabel('Plateau generation'),
   ylabel('Highest fitness achieved'),
   title('Highest fitness score by generations to achieve.');

   subplot(2,3,4)
   scatter(fittest(:,generations+1),
      fittest(:,generations+2),10,jet(length(1:mGenerations)),'filled');
   xlabel('Crossover Probability'),
   ylabel('Mutation Probabilty'),
   title('Mutation probability by crossover probability');

   % plots the same graphs, but with only the 25% best performing
   % condition setups - fitness is determined by:
   % fitness_score * (num_of_generations - plateau_generation)
   fittest = [fittest highpoint zeros(mGenerations,1)];
   for x = 1:mGenerations
     fittest(x,generations+5) = fittest(x,generations+3)*(100 -
        fittest(x,generations+4));
   end
```

```
    fittest = sortrows(fittest, -(generations+5));

    subplot(2,3,2)
    m = round(mGenerations/4);
    scatter(fittest(1:m,generations+4),
        fittest(1:m,generations+3),10,jet(length(1:m)),'filled');
    xlabel('Plateau generation'),
    ylabel('Highest fitness achieved'),
    title('Highest fitness score by generations to achieve (top 25%).');

    subplot(2,3,5)
    scatter(fittest(1:m,generations+1),
        fittest(1:m,generations+2),10,jet(length(1:m)),'filled');
    xlabel('Crossover Probability'),
    ylabel('Mutation Probabilty'),
    title('Mutation probability by crossover probability (top 25%).');

    % plots them again for only the top 12.5%
    subplot(2,3,3)
    m = round(mGenerations/8);
    scatter(fittest(1:m,generations+4),
        fittest(1:m,generations+3),10,jet(length(1:m)),'filled');
    xlabel('Plateau generation'),
    ylabel('Highest fitness achieved'),
    title('Highest fitness score by generations to achieve (top 12.5%).');

    subplot(2,3,6)
    scatter(fittest(1:m,generations+1),
        fittest(1:m,generations+2),10,jet(length(1:m)),'filled');
    xlabel('Crossover Probability'),
    ylabel('Mutation Probabilty'),
    title('Mutation probability by crossover probability (top 12.5%).');
end
```

Appendix D

runGeneration.m

```
function [resultPop, newPop, bestString] = runGeneration(newPop, crossoverProbability,
    mutationProbability)
   %% INITIALISATION

   % determines the population size
   populationSize = length(newPop(:,1));
   % adds the 31st column for fitness values
   resultPop = [newPop zeros(populationSize,1)];
   % creates an array of indicies for movement values
   movementValues = [1 4 7 10 13 16 19 22 25 28];
   % creates the rank selection array, the size of which will
   % increase in a triangular sequence (cf: OEIS A000217)
   triangular = populationSize * (populationSize + 1) / 2;
   rankSelection = zeros(triangular,1);
   n = 0;
   for i = 1:populationSize
      for j = 1:i
         n = n + 1;
         rankSelection(n) = i;
      end
   end

   %% GENERATION RUN

   % simulates all 25 chromosomes in the world
   for i = 1:populationSize
     [resultPop(i, 31),trail] = simulate_ant('muir_world.txt', sprintf('%d',
         newPop(i,:)));
   end

   % sorts the results by fitness score
   resultPop = sortrows(resultPop, -31);

   %% SELECTION & GENETIC MODIFICATION

   % elite selection picks the top 10% best-performing chromosomes
   newPop = zeros(populationSize,30);
   bestString = resultPop(1,1:30);
   newPop(1:round(populationSize/10),:) = resultPop(1:round(populationSize/10),1:30);
   n = round(populationSize/10);

   % with those two as a base, populates the rest of a new population
   % by rank selecting chromosomes and then applying a random chance
   % of random allele mutation or crossover
   while (n < populationSize)
      % use rank selection and pick parent chromosomes a and b
      rankSelect = rankSelection(randi([1,triangular]));
      a = resultPop(rankSelect, 1:30);
      rankSelect = rankSelection(randi([1,triangular]));
      b = resultPop(rankSelect, 1:30);

      % on a crossover picks a random three sections from a and replaces
      % them with the three equivalent sections from b
      if (rand < crossoverProbability)
```

```
            for i = 1:3
               r = randi([1 29],1);
               a(r:r+1) = b(r:r+1);
            end
         end

         % on a mutation picks two random values from the chromsome,
         % determines the role (and thus valid values) and changes the value
         % randomly
         if (rand < mutationProbability)
            for i = 1:2
               r = randi([1,30]);
               if any(r == movementValues)
                  a(r) = randi([1,4]);
               else
                  b(r) = randi([0,9]);
               end
            end
         end

         % adds chromosome a to the new population
         n = n + 1;
         newPop(n,:) = a;
      end
   end
```

References

[1] *"Water."* Look Around You. *BBC Two, 2002.*

[2] Hornby, Gregory S., et al. "Automated antenna design with evolutionary algorithms." *AIAA Space*. 2006.

[3] Le Goues, Claire, et al. "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each." *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012.

[4] Koza, John R. *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems*. Stanford University, Department of Computer Science, 1990.

[5] Vidler, John. "simulate_ant.m" [Computer software]. Lancaster University, 2016.

[6] Schmidhuber, Jurgen. "Evolutionary principles in self-referential learning." *On learning how to learn: The meta-meta-... hook.) Diploma thesis, Institut f. Informatik, Tech. Univ. Munich*. 1987.