Comparisons of the variable factors which contribute to Genetic Algorithm performance

Abstract - In this report I aim to explore conceptually and practically what a genetic algorithm is as well as the variable factors which contribute to its performance, examining the impact that different types of selection, types of mutation, probabilities of mutation, and population sizes have on the performance of the Genetic Algorithm. This will be explored through the lens of a genetic algorithm designed for the "John Muir" trail, which involves the simulation and improvement of a population of 'ants' exploring a 2D world containing food. The proposed solution is implemented in MATLAB and I make use of different selection and mutation methods (e.g. roulette selection versus ranked selection and random mutation versus order changing mutation), and analyse the results on the populations through multiple generations.

I. Introduction

A genetic algorithm (GA) is an adaptive algorithm which was originally proposed by John Holland in 1975 (Kumar and Jyotishree, 2012) and was described as an adaptive heuristic search algorithm "based on the evolutionary ideas of natural selection and natural genetics" (Kumar and Jyotishree, 2012). It is an unsupervised form of improvement where its purpose is to select the fitter individuals (also referred to as 'chromosomes') in the population which will create 'offsprings' for next generation (Kumar and Jyotishree, 2012).

Each chromosome is evaluated by a fitness value which is defined by the problem (Kumar and Jyotishree, 2012). Selection emphasises fitter individuals in an analogy to Darwin's theory of evolution – "survival of fittest" (Kumar and Jyotishree, 2012). Thus the next generation of the population is produced through simulated natural selection. There are various factors which impact the performance of genetic algorithms which I would like to explore including types of selection, types of mutation, probabilities of mutation, population sizes, and others.

The report is organised in the following sections. In section II, literature review is given on the key operations which compose a genetic algorithm and the various approaches to these operations. The specific problem set being addressed by this genetic algorithm is outlined in section III. I will then explain the process of my proposed solution in section IV. In section V, I present and analyse the results from my genetic algorithm and the different variables I experimented with to maximise performance. I then offer my conclusion and preference of selection in section VI.

II. Related work — explanation of the process of genetic algorithms

In the process of each generation in a genetic algorithm the population undergoes transformation using three primary operations — selection, crossover and mutation (Kumar and Jyotishree, 2012) the end result of which is the next generation of the population (Kumar and Jyotishree, 2012). The complete process of a genetic algorithm is illustrated with a flowchart in Fig. 1 (Kumar and Jyotishree, 2012).

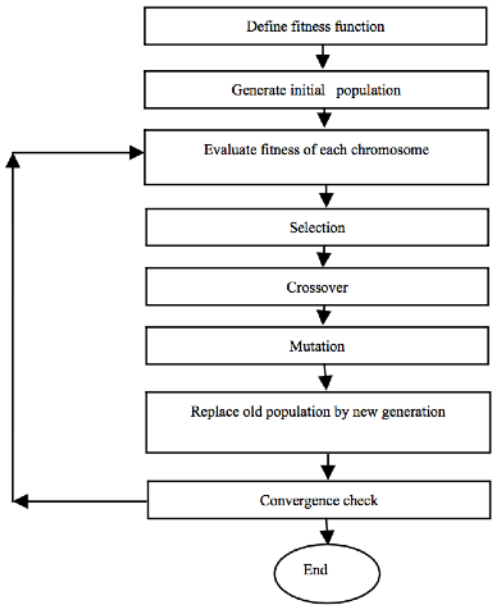


Fig. 1. Basic flowchart of a genetic algorithm

"Selection is the first genetic operation in the reproductive phase of genetic algorithm" (Kumar and Jyotishree, 2012), its purpose is to choose the fitter individuals in the population that will create offsprings for next generation. There are simple ways to do this whereby you can select the *n* fittest individuals within the population (an approach known as 'elitism'), however this is problematic in that it causes a loss of diversity by minimising the size of the population and the scope for variety (Obitko, 1998). It is however possible to use elitism in conjunction with other selection approaches though, as when creation a new population there is a possibility that the best chromosome

will be lost, elitism can increase performance as it ensures that the best individual chromosomes within the population are maintained (Obitko, 1998).

The most prominent selection approaches are roulette wheel selection and ranked selection. “In genetic algorithms, the roulette wheel selection operator has essence of exploitation while rank selection is influenced by exploration” (Kumar and Jyotishree, 2012). ‘Exploration’ is used to “investigate new and unknown areas in the search space and generate new knowledge”, while ’exploitation’ makes use of the generated knowledge to in turn maximise fitness through minor variations, “both techniques have their own merits and demerits” (Kumar and Jyotishree, 2012). There are also other more advanced approaches such as ‘blended selection’ which aims to maximise the benefits of roulette wheel and rank selection by “[showing] exploratory nature initially and [shifting] to exploitation later” (Kumar and Jyotishree, 2012).

In the case of roulette wheel selection each chromosome is assigned a segment of a virtual ‘roulette wheel’ which is proportional to its fitness whereby chromosomes with larger fitness values will be allocated a larger ‘segment’ and are more likely to be selected (Obitko, 1998). Conceptually, a marble is then thrown onto the roulette wheel and selects the chromosome, “chromosome with bigger fitness will be selected more times” (Obitko, 1998). “This may lead to biased selection towards high fitness individuals. It can also possibly miss the best individuals of a population.” (Kumar and Jyotishree, 2012). This is followed by crossover and mutation operations and the process is repeated until the desired number of individuals is selected.

Rank selection is largely similar to roulette wheel selection in that a chromosome with higher selection probability will perform better and its characteristics will persist in the population more but the way the selection probability is derived is different. Rank Selection sorts the population first according to fitness value and ranks them and chromosomes are allocated selection probability based on their rank (Kumar and Jyotishree, 2012). This can be more effective where fitness values within the population differ widely (e.g. fitness scores of [32, 34, 34, … 141]). Rather than one chromosome with high fitness taking up a large portion the roulette wheel, using ranking means that scores form a linear progression (e.g. scores of [1, 2, 3, … 50]) which results in far more proportional selection probabilities throughout the population. “Rank selection overcomes scaling problems like stagnation or premature convergence” (Kumar and Jyotishree, 2012).

The subsequent operations in each generation of a genetic algorithm are crossover and mutation. There are various ways to perform crossovers and mutations. The main act of a crossover is that one chromosome is ‘spliced’ with another chromosome so that a new chromosome is produced with characteristics from both ‘parents’, this can be done at one point or multiple points. Mutation is the act of changing specific nodes within a chromosome to introduce a level of randomness into the end result as a form of exploration. There are various methods of mutation including order changing by randomly exchanging two nodes within the chromosome sequence. The purpose of these operations is to introduce minor variations into the population which may result in improved fitness in the next generation.

III. Problem specification

The “Muir” trail artificial ant problem is a simulation of ant navigation. The ant’s task is to follow this trail and move across each black cell in sequence (Zamdborg et al., 2015). The trail itself is a series of black squares (those which contain ‘food’) on a 32x32 wraparound grid. “The trail starts off as being quite easy to follow, and gradually becomes more difficult, as the turns become more unpredictable and gaps appear” (Zamdborg et al., 2015) (Fig. 2).

The ant stands on a single cell and is oriented to the north, south, east, or west and is capable of sensing if the cell directly in front of it contains food. The ant can perform one of four actions: moves forward; rotate right; rotate left; or do nothing.

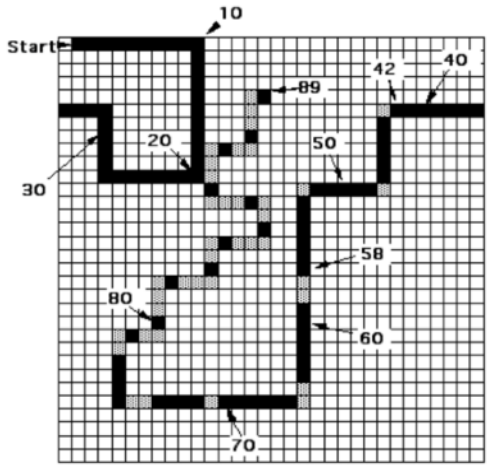


Fig. 2. Example route (Jefferson et al.)

The simulator function (Appendix 2) takes a decision tree encoded 30 digit string which is composed of 10 instruction sets (Zamdborg et al., 2015). The first digit in the set denotes the action the ant takes. If there is no food in front of the ant after performing this action the ant will transition to the instruction state indicated by the second digit, if there is food then the ant will transition to the instruction state indicated by the third digit. This represents one complete move. When the function has completed the simulation it returns a fitness score corresponding to the number of food squares covered by the ant in it's route as well as the coordinates for the ants route.

IV. Proposed solution

Below I will outline the process of my proposed solution and explain the code used (see Appendix 1 for complete solution) for my genetic algorithm.

1. At the top of the script I have extracted key parameters which can be manipulated to test impact on performance.

```
CrossoverProbability = 1;
MutationProbability = 1;
Generations = 300;
PopulationSize = 30;
SequenceLength = 30;
Environment = 'muir_world.txt';
```

The values of SequenceLength and Environment are specific to the "Muir" trail;
- SequenceLength - is used to specify the length of the sequence which composes a chromosome, this could be longer if the simulator function took more instructions.
- Environment - is used as a simple reference to the file name for the 2D world being used in the problem, this could be changed for a different world-map.

CrossoverProbability and MutationProbability dictate the probability that a crossover or mutation will occur on a chromosome in a generation — for testing other variable factors I set probability to 1 so that I could better draw conclusions of cause and effect on the changes I made, I will elaborate on my findings for the optimal probability for mutation in my results. Generations is the number of cycles or 'generations' which the algorithm will go through; and PopulationSize is the number of chromosomes in the population.

2. I then generate the initial random population. The population is composed of multiple chromosomes, each of which is represented by a numeric matrix. I loop to create each individual instruction set which is composed of three digits using the pseudo-random number function randi() to generate a number within a given range e.g. passing [0 9] will return a number between 0 and 9.

```
Population = zeros(PopulationSize, SequenceLength);

for i = 1:PopulationSize
    TempChromosome = zeros(1, SequenceLength);

    for j = 1: SequenceLength/3
        TempChromosome((j*3)-2) = randi([1 4]);
        TempChromosome((j*3)-1) = randi([0 9]);
        TempChromosome(j*3) = randi([0 9]);
    end

    Population(i, :) = TempChromosome;
end
```

Once the initial population has been created the main genetic operations loop for the given number of generations.

3. We begin by evaluating the fitness scores for each member of the population using the provided simulation function simulate_ant. I store the fitness in a column at the end of my matrix so that the population can then be easily ranked using the function sortrows().

```
for j = 1:PopulationSize
    [Fitness, Path] = simulate_ant(Environment, Population(j, 1:SequenceLength));
    Population(j, SequenceLength+1) = Fitness;
end

Population = sortrows(Population, SequenceLength+1);
```

4. If we are using ranked selection this is where we then convert the chromosomes ordered fitness scores to their corresponding 'ranking' whereby the worst will have fitness 1 and the best will have fitness PopulationSize (e.g. 1 to 50).

```
for j = 1:PopulationSize
    Population(j, SequenceLength+1) = j;
end
```

5. We then normalise fitness scores so that they are between the range of 0 and 1 to make the dynamic range of fitness scores more easily comparable and accumulate the normalised data for use in the roulette wheel selection.

```
NormalisedFitness = Population(:, end)/sum(Population(:, end));
CumulativeNormalisedFitness = cumsum(NormalisedFitness);
```

Normalisation can be problematic when there are significant outliers in the data being normalised, however, in this case all fitness scores are within the consistent range of 0 to 89 so this is not as much of a concern in this case.

6. To make use of partial elitism selection to ensure that the best individual chromosomes within the population are maintained we then take the two best chromosomes from the current population and add them to the new population.

```
PopulationNew = zeros(PopulationSize, SequenceLength);
PopulationNew(1:2, :) = Population(PopulationSize-1:PopulationSize,
1:SequenceLength);
PopulationNewSize = 2;
```

We now repeat the selection stage outlined below until new population is full.

7. Randomly selecting two chromosomes from the 'roulette wheel' of the population where CumulativeNormalisedFitness is used to determine proportions of segments for each chromosome.

```
R1 = rand;
PIdx1 = find(CumulativeNormalisedFitness>=R1, 1, 'first');
TempChromosome_1 = Population(PIdx1, 1:SequenceLength);

R2 = rand;
PIdx2 = find(CumulativeNormalisedFitness>=R2, 1, 'first');
TempChromosome_2 = Population(PIdx2, 1:SequenceLength);
```

8. Deciding whether two chromosomes will be crossed over based on the defined CrossoverProbability and performing a two-point crossover if it is. If no crossover is performed, the offspring is an exact copy of its parents.

```
if (rand < CrossoverProbability)
    CrossPoint_1 = randi([1 ((SequenceLength/3)-2)])*3;
    CrossPoint_2 = randi([(CrossPoint_1/3) ((SequenceLength/3)-1)])*3;

    TempChromosome_3 = [TempChromosome_1(1:CrossPoint_1),
       TempChromosome_2((CrossPoint_1+1):CrossPoint_2),
       TempChromosome_1(CrossPoint_2+1:SequenceLength)];
    TempChromosome_4 = [TempChromosome_2(1:CrossPoint_1),
       TempChromosome_1((CrossPoint_1+1):CrossPoint_2),
       TempChromosome_2(CrossPoint_2+1:SequenceLength)];

    TempChromosome_1 = TempChromosome_3;
    TempChromosome_2 = TempChromosome_4;
```

```
end
```

I used a two-point crossover whereby a start and finish point for the crossover is selected and the nodes within that range in one chromosome are transposed onto the other chromosome and vice versa. I made the choice to ensure that cross points start and end at the start/end of instruction sets. Because in this case where the problem uses a sequence of three genes to encode an “instruction", I believe that it makes more sense to exchange complete instruction sets rather than single bits to maintain more of the existing structure.

My solution ensures that the first crossover point is between the 1st and 8th instruction sets, and the second crossover point can be up to the 9th instruction set, maximising the probability that there will be a full two-point crossover even in the case where the first cross-point is at the 8th instruction set.

9. Deciding whether each chromosome will be mutated based on the defined MutationProbability (this code is repeated for TempChromosome_2 separately).

```
if (rand < MutationProbability)
    TempChromosome_1 = random_value_mutation(SequenceLength, TempChromosome_1);
end
```

I implemented and experimented with two forms of mutation. If we are performing random value mutation the code is as follows — we select one mutation point and determine which digit in the instruction set it is using the function rem(n, 3), so that we can randomly generate a correct value for it (digit 1 requires a value between 1 and 4 while digits 2 and 3 require a value between 0 and 9), the then repeat this action again for a second random mutation point.

```
MutationPoint_1 = randi([1 SequenceLength]);
MutationPoint_1_Rem = rem(MutationPoint_1, 3);

if MutationPoint_1_Rem ~= 1
    TempChromosome(MutationPoint_1) = randi([0 9]);
else
    TempChromosome(MutationPoint_1) = randi([1 4]);
end

MutationPoint_2 = randi([1 SequenceLength]);
MutationPoint_2_Rem = rem(MutationPoint_2, 3);

if MutationPoint_2_Rem ~= 1
    TempChromosome(MutationPoint_2) = randi([0 9]);
else
    TempChromosome(MutationPoint_2) = randi([1 4]);
end
```

This introduces more randomness into the genetic strings by introducing completely new values, the impact of which I will elaborate on in my results

Alternatively, if we are performing order changing mutation the logic is as follows — we use the function rem(n, 3) to determine which digit in the instruction set the mutation point is and find a corresponding digit from another instruction set to swap places with it.

```
MutationPoint_1 = randi([1 SequenceLength]);
MutationPoint_1_Rem = rem(MutationPoint_1, 3);

MutationValue_1 = TempChromosome(MutationPoint_1);

if MutationPoint_1_Rem == 2
    MutationPoint_2 = (randi([1 (SequenceLength/3)])*3)-2;

    TempChromosome(MutationPoint_1) = TempChromosome(MutationPoint_2);
    TempChromosome(MutationPoint_2) = MutationValue_1;
elseif MutationPoint_1_Rem == 1
    MutationPoint_2 = (randi([1 (SequenceLength/3)])*3)-1;

    TempChromosome(MutationPoint_1) = TempChromosome(MutationPoint_2);
    TempChromosome(MutationPoint_2) = MutationValue_1;
```

```
elseif MutationPoint_1_Rem == 0
    MutationPoint_2 = (randi([1 (SequenceLength/3)])*3);

    TempChromosome(MutationPoint_1) = TempChromosome(MutationPoint_2);
    TempChromosome(MutationPoint_2) = MutationValue_1;
end
```

This serves to maintain more of the existing structure of the genetic strings, potentially maintaining and improving positive traits from the population as genes are not replaced but reordered.

10. Following mutation we then add the new chromosomes to the new population if there is capacity.

```
PopulationNewSize = PopulationNewSize + 1;
PopulationNew(PopulationNewSize,:) = TempChromosome_1;

if (PopulationNewSize < PopulationSize)
    PopulationNewSize = PopulationNewSize + 1;
    PopulationNew(PopulationNewSize, :) = TempChromosome_2;
end
```

11. With the new generation of the population now created we replace the pervious population, and the cycle is repeated for the given number of generations.

```
Population(:, 1:SequenceLength) = PopulationNew;
```

V. Results and findings

In a sample run of my solution the fitness score of the best individual in the first generation was 42, over 300 generations the fitness score improved to 84 (see Fig. 3). In my solution I made use of elitism and roulette wheel selection with a two-point crossover and random value mutation. The path taken by the fittest individual in the final generation can be seen in Fig. 4 and the instruction string can be seen in Fig. 5, the path is direct compared to some paths observed using order changing mutation (Appendix 9).

Fig. 3. Fitness over multiple generations

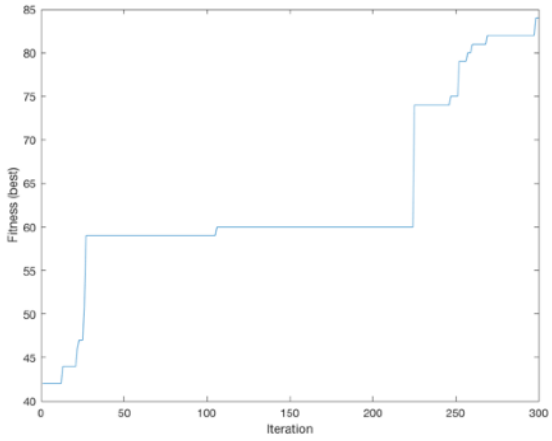


Fig. 4. Path taken by fittest individual

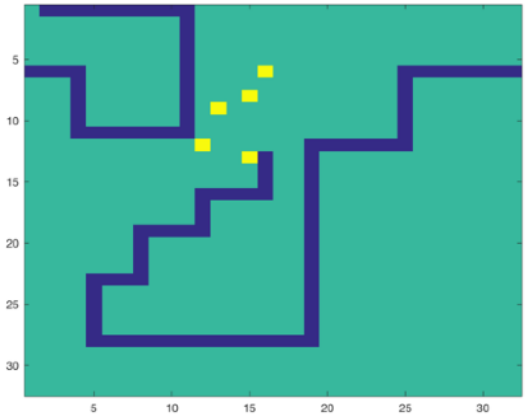


Fig. 5. Instruction string of most fit individual in final

```
2  2  3  1  9  1  2  1  2  1  6  1  1  1  8  2  0  6  1  1  7  4  1  6  2  6  2  2  5  3
```

A. *Methodology*

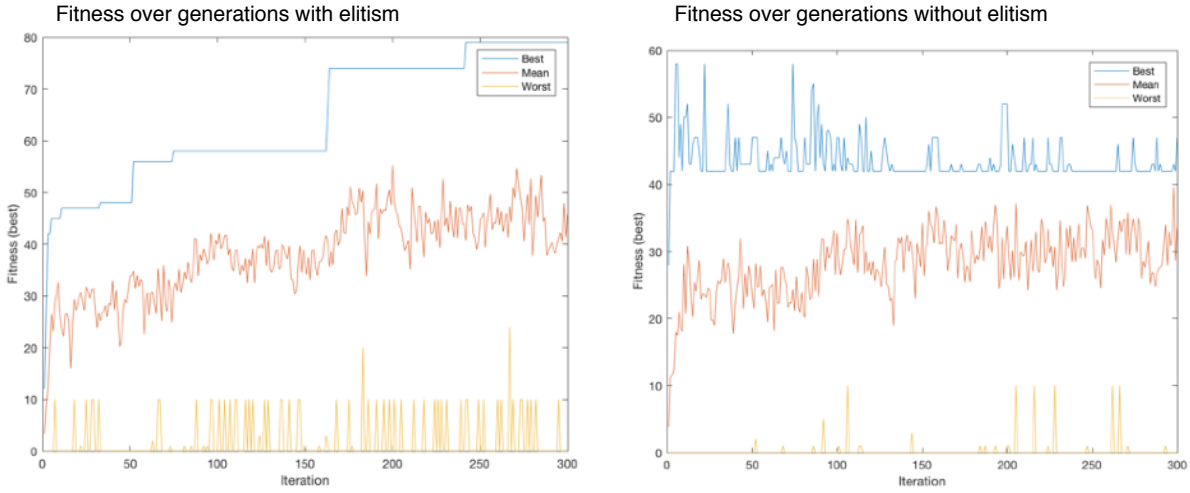
For the collection of my results I used consistent values for variables like population size, number of generations, and crossover and mutation probability (see Appendix 1 for variables used when testing) so that when changing these variables or the selection type used in tests I could better observe changes in cause and effect.

When testing changes to my algorithm I plotted best, mean, and worst fitness for the population over generations, and in each test ran the algorithm 10 times per scenario recording the best and mean fitness of the last generation each time to develop an overall average best and mean fitness for the population when using different approaches for comparison.

I began by comparing performance at different numbers of generations and found significant improvement of maximum fitness as well as the convergence with the average fitness of the overall population generally began to diminish after around 200 generations, as such for data collection I observed changes over 300 generations to maximise observable changes.

B. Elitism selection

Fig. 6. Performance comparison with and without elitism



I began be testing to observe the contribution elitism has to performance, which ensures that the fittest two chromosomes within the current population persist into the next population. The results of my comparison were clear: without elitism the fitness of the ants is erratic and does not show a clear progression. Rather, it can be seen that in some generations a chromosome attained a high fitness score but this was lost in the next generation (see Fig. 6).

While the mean for the population was in the acceptable range of 20-30 (still performing worse than with elitism) because the solution still made use of roulette wheel selection which places weighting on fitness, not using elitism limits the scope for exploration — without retaining the best performing members of the population it is harder to capitalise on them and maximise fitness over time if we are likely to lose their characteristics with each new generation. Using elitism means there is a much clearer positive progression through generations.

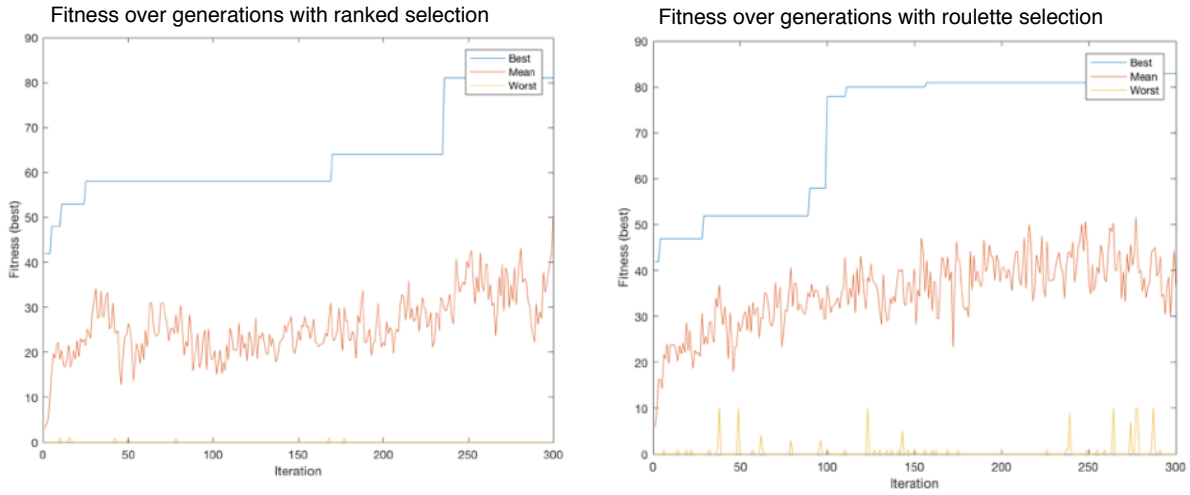
I believe that elitism offers value in maintaining population stability and promotes a better population mean fitness over time as the general population converges with the best members in a way that isn't achievable without elitism because the best members can be lost. However, selecting other individuals in addition to the best ones maintains the diversity of the population: the solutions are spread further out in the search space, and if a part of the population is stuck in a local optimum, a different part of the population can still make progress.

C. Comparing roulette selection and ranked selection

In my tests I compared the performance of ranked selection and roulette selection with random value mutation, both approaches incorporated elitism. The conventional belief is that roulette selection does not

work well if the fitness scale is not very regular, and that in practice it never is regular (Kumar and Jyotishree, 2012), however, I found the results generated by each approach to be very close in terms of performance. Using roulette section produced an average best fitness of 80.7, which was slightly worse than using ranked selection which was 82.3. Roulette selection produced a better overall population though with a mean fitness of 37.5 compared to 32.9 using ranked selection (see Appendix 3).

Fig.7. Comparison of roulette selection and ranked selection with random mutation



In the case of roulette selection the overall mean of the population is trending towards convergence with the best fitness but this would take a significantly larger number of generations. It can also be observed that the worst fitness within the population is more erratic compared to ranked selection (see Fig. 7) — in some generations it will achieve a weakest fitness of 10 and this will drop back down in the next generation — I think this is because with roulette selection weaker chromosomes have less probability of being selected which causes potentially good (relatively) but week chromosomes to be lost. With ranked selection there is a clear stratification in terms of the best, mean and worst within the population which might be perpetuated through the use of the ranking mechanism (see Fig. 7).

These results are still very close in performance though and results were even closer when using order changing mutation (see Appendix 3), this is presumably because random value mutation introduces more randomness into the characteristics of the population while order changing maintains more characteristics and the exploratory/exploitative nature of the two approaches benefits from more variety being introduced into the population with random mutation.

These results are illustrative of the main characteristic benefits of roulette and ranked selection, whereby ranked selection emphasises exploration which maximises the potential best fitness score while roulette emphasises exploitation which can “makes use of the generated knowledge” to improve the fitness of the whole population (Kumar and Jyotishree, 2012). The choice of which selection approach to use comes down to whether the problem is optimising for best fitness or mean fitness and this will vary depending on the problem set.

D. Comparing mutation strategies

I also chose to attempt two approaches to mutation of chromosomes, comparing random mutation which introduced completely new random values into the chromosomes as well as order changing mutation which better preserved the characteristics of chromosomes but changed the order of nodes.

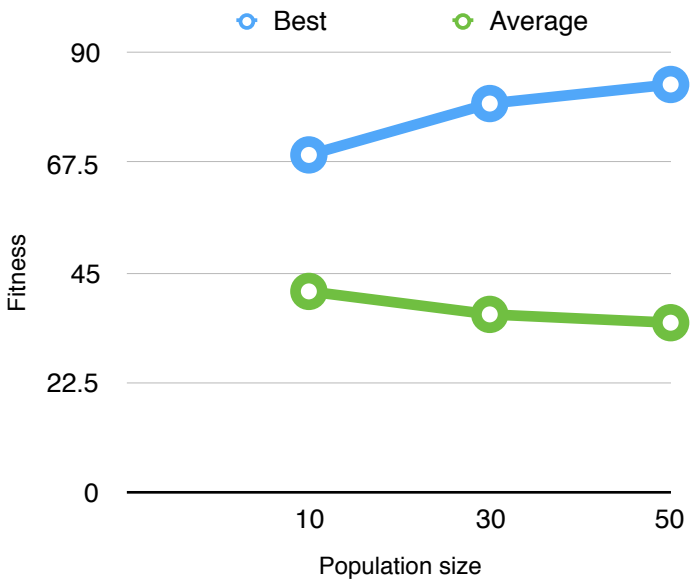
I found that using random mutation resulted in higher average best fitness with 80.7 compared to 71.9, but lower mean population fitness with 37.5 compared to 44.6 (when using roulette selection, the same pattern was also apparent with ranked selection, see Appendix 3). I believe this is because random mutation

introduces more randomness into the population allowing for better exploration to increase the overall best fitness score found and this works well with elitism as there is a gradual convergence with the mean towards the best. Order changing mutation on the other hand preserves more of the positive characteristics of the population which are passed on when chromosomes are crossed over leading to a population with an overall higher mean fitness. The choice of which mutation strategy to use again depends on the focus of the problem whether we are optimising for best fitness or mean fitness, or could be used with the counter selection approach to get a balance of both benefits.

E. Comparing performance at different population sizes

Research suggests that the optimal population size is around 20-30, however sometimes sizes 50-100 are reported as best (Obitko, 1998). When comparing population sizes I tested populations of 10, 30, and 50. I found that there was a relationship between smaller populations resulting in greater convergence (i.e. higher population mean), but with lower best fitness scores (see Fig. 8).

Fig. 8. Comparison of fitness performance at different population sizes (using roulette selection)

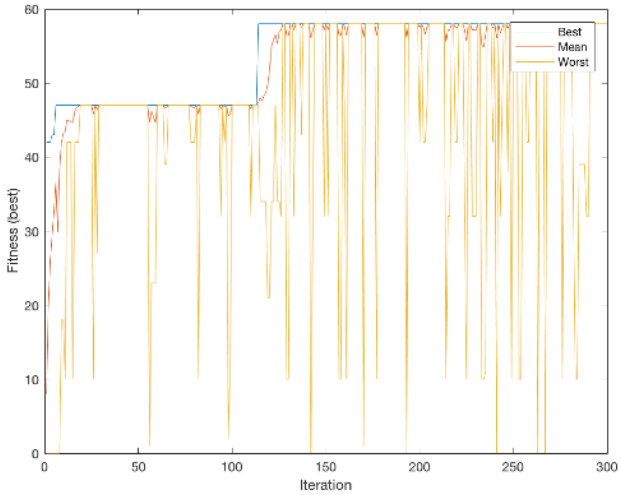


This makes sense as there is less diversity within the population at smaller sizes meaning that the population converges more quickly but there is also less of the kind of variety which enables maximisation of higher fitness scores. I found that this occurred more noticeably with ranked selection, as from my previous findings ranked selection emphasises maximising the potential best fitness score over the population mean (see Appendix 5). To this point I do believe that my findings are consistent with the general consensus that a population size of around 30 is optimal, as this is between the extremes seen in larger and smaller populations.

F. Comparing performance at different mutation probabilities

Research typically suggests that mutation probability should be very low, with the best rates reported to be about 0.5%-1% (Obitko, 1998).

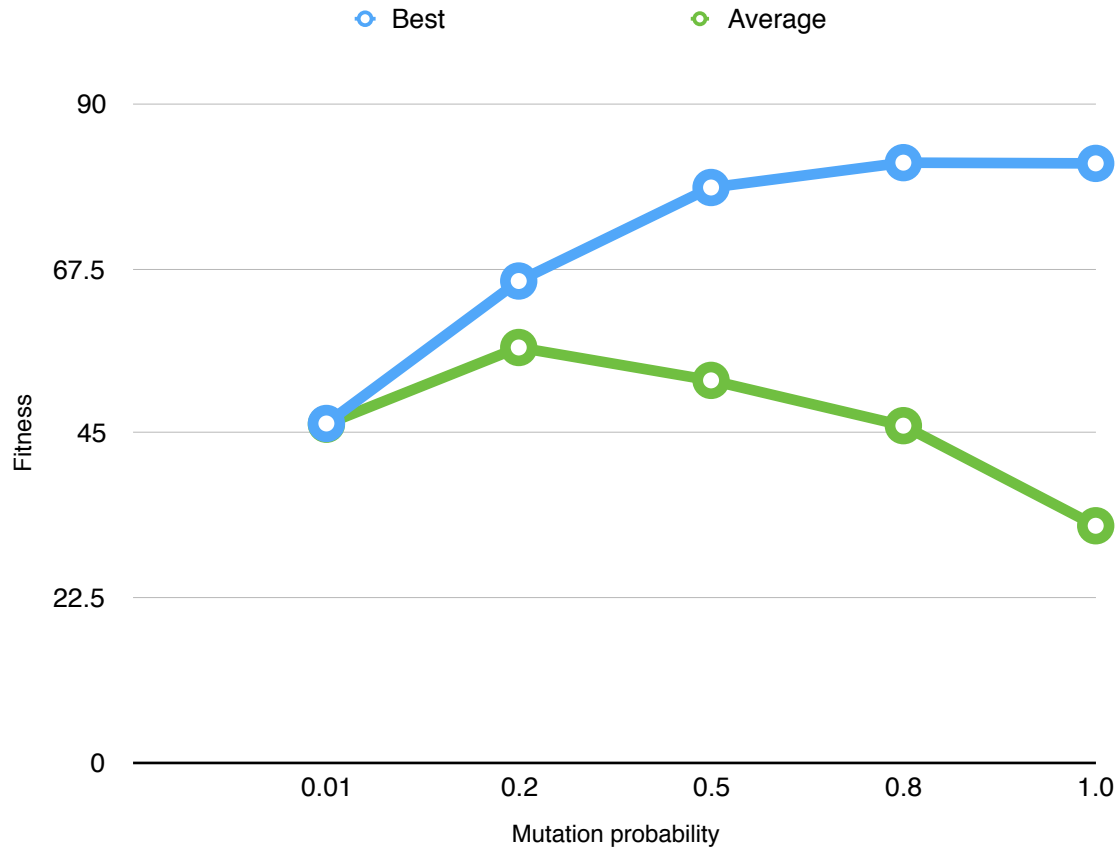
Fig. 9. Fitness over generations using mutation probability 0.01



In my findings there was again an observable trend whereby as mutation probability increases the best fitness increases and mean population fitness decreases (see Fig. 10). When using an extremely low mutation probability the population is essentially uniform (see Fig. 9) by the final generation with a best fitness of 46.3 and mean fitness of 46.29 (see Appendix 7) while at the high end using 100% probability of mutation we see high best fitness scores but a weaker average of 36.3.

I found that while a lower mutation probability increases convergence between the best and the average increasing the fitness of the overall population, this also results in a a noticeable loss of diversity and can

Fig.10. Comparison of fitness performance at different mutation probabilities (using roulette selection)



mean that the population gets stuck in a local minima. A lower mutation probability also means that the population does not see continued increases in best fitness beyond a certain point once it has fully converged. Increasing the mutation probability introduces more randomness and diversity into the population allowing it to achieve higher best fitness scores and prevent premature convergence.

Based on my findings I think the optimal probability for the problem would be 0.2 or 0.8 as these mitigate the extremes of both approaches — using a probability of 0.8 actually resulted in a better best fitness and better mean fitness than 1.0 and the mean fitness was comparable to results attained at 0.01 (see Appendix 7). I think it also makes sense to use a high probability for crossover as well, where findings typically suggest crossover probability of 80%-95% (Obitko, 1998).

VI. Conclusion

Genetic algorithms provide an effective way to perform unsupervised improvement of a population by simulating natural selection, and there are various options to operationalise for different goals (i.e. maximising for a higher best fitness or mean population fitness), as well as new emergent 'blended' approaches which can capitalise on the benefits of both exploratory and exploitative development of the population (Kumar and Jyotishree, 2012) which would be interesting to explore in future research.

Based on my findings I believe that a combination approach of elitism selection and ranked or roulette wheel selection proves effective in this problem set, and the choice of which selection approach to use depends on the goals of the algorithm. In the case of this algorithm I think the use of roulette selection to maximise exploration makes sense as if the algorithm is performed over a larger number of generations this will result in a higher best fitness as well as population convergence over time. I also believe that the use of random value mutation makes the most sense in this case to increase exploration by introducing more randomness into the population. In future research I think there is scope to examine these variable factors over more generations to conclude if there is a notable improvement in convergence over time with roulette selection compared to rank selection.

References

Jefferson, D. (1991) The Genesys System: Evolution as a Theme in Artificial Life. Available at: http://web.cs.ucla.edu/~dyer/Papers/AlifeTracker/Alife91Jefferson.html (Accessed: 12 December 2016).

Kumar, R. and Jyotishree (2012) 'Blending roulette wheel selection & rank selection in genetic Algorithms', International Journal of Machine Learning and Computing, , pp. 365–370.

Obitko, M. (1998) Introduction to Genetic Algorithms. Available at: http://www.obitko.com/tutorials/genetic-algorithms/index.php (Accessed: 12 December 2016).

Zamdborg, L., Holloway, D.M., Merelo, J.J., Levchenko, V.F. and Spirov, A.V. (2015) 'Forced evolution in silico by artificial transposons and their genetic operators: The ant navigation problem', Information Sciences, 306, pp. 88–110.

Appendices

Appendix 1: Software code for "Muir" trail GA

```
clear;
close all;

%%% Parameters
CrossoverProbability = 1;
MutationProbability = 1;
Generations = 40;
PopulationSize = 30;
SequenceLength = 30;
Environment = 'muir_world.txt';

Population = zeros(PopulationSize, SequenceLength);

%%% Generate random population
for i = 1:PopulationSize
    TempChromosome = zeros(1, SequenceLength);

    for j = 1:SequenceLength/3
        TempChromosome((j*3)-2) = randi([1 4]);
        TempChromosome((j*3)-1) = randi([0 9]);
        TempChromosome(j*3) = randi([0 9]);
    end

    Population(i, :) = TempChromosome;
end

%%% Include extra column at end for scores
Population = [Population zeros(PopulationSize, 1)];

BestFitnessHistory = zeros(Generations, 1);
%%%     FitnessHistory = zeros(Generations, 3);

%%% Repeat `Generations` times to create a new population
for i = 1:Generations
    %%% Evaluate fitness scores
    for j = 1:PopulationSize
        [Fitness, Path] = simulate_ant(Environment, Population(j, 1:SequenceLength));
        Population(j, SequenceLength+1) = Fitness;
    end

    %%% Sort chromosomes in increasing fitness
    Population = sortrows(Population, SequenceLength+1);

    %%% Record best fitness in the population
    MaxFitness = max(Population(:, end));

    BestFitnessHistory(i) = MaxFitness;

    %%% Uncomment to record mean and weakest fitness as well
    %%% as best (along with lines 31 and 73-77)
    %%%     FitnessHistory(i, 1) = MaxFitness;
    %%%     FitnessHistory(i, 2) = mean(Population(:, end));
    %%%     FitnessHistory(i, 3) = min(Population(:, end));

    %%% Uncomment to use rank selection
    %%%     %%% Rank population instead of using fitness
    %%%     for j = 1:PopulationSize
    %%%         Population(j, SequenceLength+1) = j;
    %%%     end

    %%% Normalise fitness and calculate cumulative fitness variables to facilitate the
selection step
    NormalisedFitness = Population(:, end)/sum(Population(:, end));
    CumulativeNormalisedFitness = cumsum(NormalisedFitness);

    if mod(i, 1) == 0
        fprintf('Iter. %d: Best solution: Fitness = %f\n', i, MaxFitness);
```

```
        figure(1);
        plot(BestFitnessHistory(1:i));
        xlabel('Iteration');
        ylabel('Fitness (best)');

        %%% Uncomment to plot best, mean, worst
        %%%     figure(2);
        %%%     plot(FitnessHistory(1:i, :));
        %%%     legend('Best', 'Mean', 'Worst')
        %%%     xlabel('Iteration');
        %%%     ylabel('Fitness (best)');
        pause(0.01);
    end

    %%% Elite, keep best 2
    PopulationNew = zeros(PopulationSize, SequenceLength);
    PopulationNew(1:2, :) = Population(PopulationSize-1:PopulationSize,
1:SequenceLength);
    PopulationNewSize = 2;

    %%% Repeat until new population is full
    while (PopulationNewSize < PopulationSize)
        %%% Use roulette wheel selection selection and pick two chromosomes
        %%% `rand` generates a (pseudo-)random number drawn from
        %%% the uniform distribution on the interval (0,1)
        R1 = rand;
        PIdx1 = find(CumulativeNormalisedFitness>=R1, 1, 'first');
        TempChromosome_1 = Population(PIdx1, 1:SequenceLength);

        R2 = rand;
        PIdx2 = find(CumulativeNormalisedFitness>=R2, 1, 'first');
        TempChromosome_2 = Population(PIdx2, 1:SequenceLength);

        %%% Perform crossover of `TempChromosome_1` and `TempChromosome_2` with
probability, `CrossoverProbability`:
        if (rand < CrossoverProbability)
            %%% `SequenceLength/3` ensures that the cross point falls at the start of an
instruction set
            %%% We `-2` to ensure that the cross point can at most be at the 8th
instruction set so that
            %%% the second cross point can be at the end of this set (so that there is a
two-point crossover)
            CrossPoint_1 = randi([1 ((SequenceLength/3)-2)])*3;
            CrossPoint_2 = randi([(CrossPoint_1/3) ((SequenceLength/3)-1)])*3;

            TempChromosome_3 = [TempChromosome_1(1:CrossPoint_1),
TempChromosome_2((CrossPoint_1+1):CrossPoint_2),
TempChromosome_1(CrossPoint_2+1:SequenceLength)];
            TempChromosome_4 = [TempChromosome_2(1:CrossPoint_1),
TempChromosome_1((CrossPoint_1+1):CrossPoint_2),
TempChromosome_2(CrossPoint_2+1:SequenceLength)];

            TempChromosome_1 = TempChromosome_3;
            TempChromosome_2 = TempChromosome_4;
        end

        %%% Perform mutation on `TempChromosome_1` and `TempChromosome_2` with
probability, `MutationProbability`:
        %%% For each chromosome, randomly picking two mutation points (but same
instruction digit) and
        %%% swap the corresponding values
        if (rand < MutationProbability)
            %%% Mutation by assigning a new random value
            TempChromosome_1 = random_value_mutation(SequenceLength, TempChromosome_1);

            %%% Mutation by order changing (also uncomment line 129 and comment out
lines 119 and 126)
            %%%     TempChromosome_1 = order_changing_mutation(SequenceLength,
TempChromosome_1);
        end
        if (rand < MutationProbability)
            %%% Mutation by assigning a new random value
```

```
            TempChromosome_2 = random_value_mutation(SequenceLength, TempChromosome_2);

            %%% Mutation by order changing (also uncomment line 122 and comment out
lines 119 and 126)
            %%%     TempChromosome_2 = order_changing_mutation(SequenceLength,
TempChromosome_2);
        end

        %%% Add to the new population if there is capacity
        PopulationNewSize = PopulationNewSize + 1;
        PopulationNew(PopulationNewSize,:) = TempChromosome_1;

        if (PopulationNewSize < PopulationSize)
            PopulationNewSize = PopulationNewSize + 1;
            PopulationNew(PopulationNewSize, :) = TempChromosome_2;
        end
    end

    Population(:, 1:SequenceLength) = PopulationNew;
end

%%% Evaluate final fitness scores and rank them
for i = 1:PopulationSize
    [Fitness, Path] = simulate_ant(Environment, Population(i, 1:SequenceLength));
    Population(i, SequenceLength+1) = Fitness;
end

Population = sortrows(Population, SequenceLength+1);

%%% Display the best (highest) fitness value
[MaxFitness, BestPath] = simulate_ant(Environment, Population(PopulationSize,
1:SequenceLength));
fprintf('Best solution: Fitness = %f\n', MaxFitness);

Map = load('muir_world.txt');
figure(3);
imagesc(Map);

for i = 1:length(BestPath)
    Map(BestPath(i, 1), BestPath(i, 2)) = -1;
end

figure(4);
imagesc(Map);

%%% Mutation by randomly selecting two mutation points and assigning new
%%% values based on which digit in the instruction set the mutation point
%%% is (can be 1-4 or 0-9)
function [TempChromosome] = random_value_mutation(SequenceLength, TempChromosome)
    MutationPoint_1 = randi([1 SequenceLength]);
    MutationPoint_1_Rem = rem(MutationPoint_1, 3);

    if MutationPoint_1_Rem ~= 1
        TempChromosome(MutationPoint_1) = randi([0 9]);
    else
        TempChromosome(MutationPoint_1) = randi([1 4]);
    end

    MutationPoint_2 = randi([1 SequenceLength]);
    MutationPoint_2_Rem = rem(MutationPoint_2, 3);

    if MutationPoint_2_Rem ~= 1
        TempChromosome(MutationPoint_2) = randi([0 9]);
    else
        TempChromosome(MutationPoint_2) = randi([1 4]);
    end
end

%%% Mutation by randomly selecting one mutation point and then randomly
%%% selecting another based on which digit in the instruction set the
%%% mutation point is and exchanging the two values
function [TempChromosome] = order_changing_mutation(SequenceLength, TempChromosome)
```

```
    MutationPoint_1 = randi([1 SequenceLength]);
    MutationPoint_1_Rem = rem(MutationPoint_1, 3);

    MutationValue_1 = TempChromosome(MutationPoint_1);

    %%% Ensures that digit is only swapped with corresponding digit from another
instruction set
    if MutationPoint_1_Rem == 2
        MutationPoint_2 = (randi([1 (SequenceLength/3)])*3)-2;

        TempChromosome(MutationPoint_1) = TempChromosome(MutationPoint_2);
        TempChromosome(MutationPoint_2) = MutationValue_1;
    elseif MutationPoint_1_Rem == 1
        MutationPoint_2 = (randi([1 (SequenceLength/3)])*3)-1;

        TempChromosome(MutationPoint_1) = TempChromosome(MutationPoint_2);
        TempChromosome(MutationPoint_2) = MutationValue_1;
    elseif MutationPoint_1_Rem == 0
        MutationPoint_2 = (randi([1 (SequenceLength/3)])*3);

        TempChromosome(MutationPoint_1) = TempChromosome(MutationPoint_2);
        TempChromosome(MutationPoint_2) = MutationValue_1;
    end
end
```

Appendix 2: Simulate ant function

```
% -this is simulator for ant
% -you do not need to change this file, but you need to use this function
% in your GA code

function [fitness, trail] = simulate_ant(filename_world, string_controller)

world_grid = dlmread(filename_world,' ');
[r_world_grid, c_world_grid] = size(world_grid);

% parse string into fsm
fsm_controller = zeros(10,3);
for i=1:10
    for j=1:3
        fsm_controller(i,j) = string_controller((i-1)*3+j);
    end
end

fitness = 0;
trail = zeros(200,2);

ant_pos = [1 1];
ant_ori = 1;
curr_state = 0;
for k=1:200
    % read gridcell in front and get false/true
    front_pos = ant_pos;
    if (ant_ori == 1)
        front_pos(2) = front_pos(2) + 1;
        if (front_pos(2) > c_world_grid) front_pos(2) = 1; end
    elseif (ant_ori == 2)
        front_pos(1) = front_pos(1) - 1;
        if (front_pos(1) == 0) front_pos(1) = r_world_grid; end
    elseif (ant_ori == 3)
        front_pos(2) = front_pos(2) - 1;
        if (front_pos(2) == 0) front_pos(2) = c_world_grid; end
    elseif (ant_ori == 4)
        front_pos(1) = front_pos(1) + 1;
        if (front_pos(1) > r_world_grid) front_pos(1) = 1; end
    end
    exist_food = world_grid(front_pos(1), front_pos(2));

    % change currstate
    curr_state = fsm_controller(curr_state+1, exist_food+2);

    % take action
    temp_action = fsm_controller(curr_state+1, 1);
    if (temp_action == 1)
        % move forward one cell
        if (ant_ori == 1)
            ant_pos(2) = ant_pos(2) + 1;
            if (ant_pos(2) > c_world_grid) ant_pos(2) = 1; end
        elseif (ant_ori == 2)
            ant_pos(1) = ant_pos(1) - 1;
            if (ant_pos(1) == 0) ant_pos(1) = r_world_grid; end
        elseif (ant_ori == 3)
            ant_pos(2) = ant_pos(2) - 1;
            if (ant_pos(2) == 0) ant_pos(2) = c_world_grid; end
        elseif (ant_ori == 4)
            ant_pos(1) = ant_pos(1) + 1;
            if (ant_pos(1) > r_world_grid) ant_pos(1) = 1; end
        end
    elseif (temp_action == 2)
        % turn right 90 degrees
        ant_ori = ant_ori - 1;
        if (ant_ori == 0) ant_ori = 4; end
    elseif (temp_action == 3)
        % turn left 90 degrees
        ant_ori = ant_ori + 1;
        if (ant_ori == 5) ant_ori = 1; end
    end
```

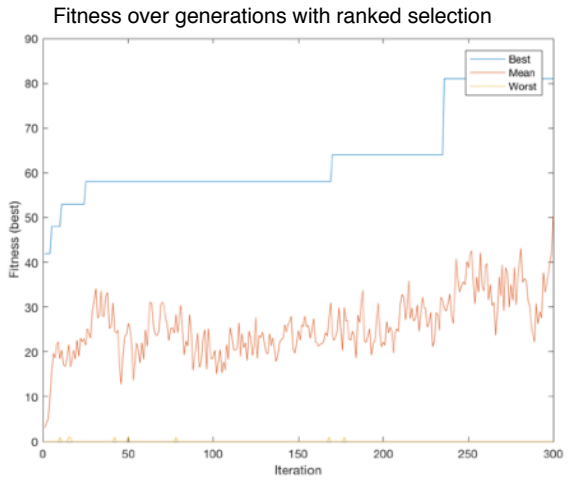
```
    % possibly eat food and increase fitness, and store trail
    if (world_grid(ant_pos(1), ant_pos(2)) == 1)
        world_grid(ant_pos(1), ant_pos(2)) = 0;
        fitness = fitness + 1;
    end
    trail(k,:) = ant_pos;
end
```

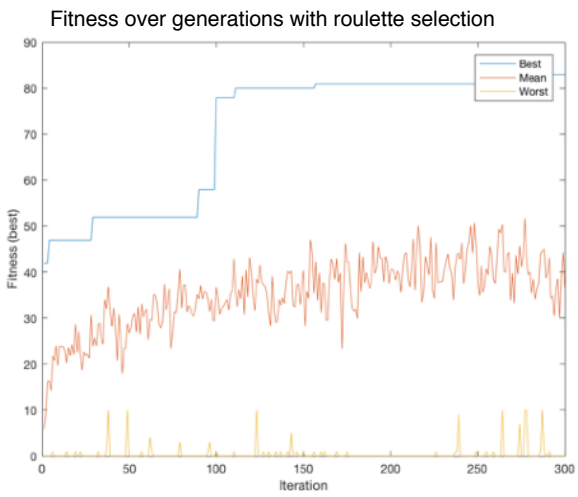
Appendix 3: Roulette selection and ranked selection comparison results

	Roulette wheel selection				Rank selection			
	Random mutation		Order changing mutation		Random mutation		Order changing mutation	
Attempt	Best	Average (mean)	Best	Average (mean)	Best	Average (mean)	Best	Average (mean)
1	83	25.9	74	41.6	81	27.2	59	34.6
2	84	36.6	74	49.6	84	40	58	45.7
3	83	39.4	82	48.7	83	34.4	82	51.3
4	83	40.6	58	39.6	84	34.1	79	43.3
5	83	37.5	82	56	81	25.2	52	47.6
6	81	50.6	74	49.3	83	36.5	82	34.9
7	84	27	81	38.6	80	32.6	81	60.4
8	82	45	56	28.9	81	31.3	80	35.1
9	63	31.6	57	40.1	82	34.3	79	46.6
10	81	41.1	81	53.8	84	34	58	47.3
Average (mean):	**80.7**	**37.53**	**71.9**	**44.62**	**82.3**	**32.96**	**71**	**44.68**

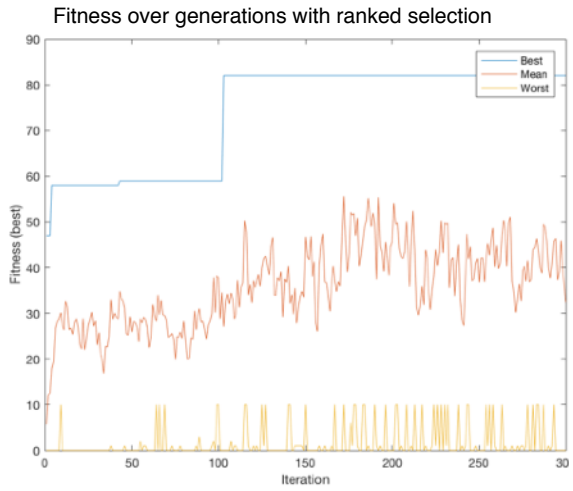
Appendix 4: Roulette selection and ranked selection comparison plotting

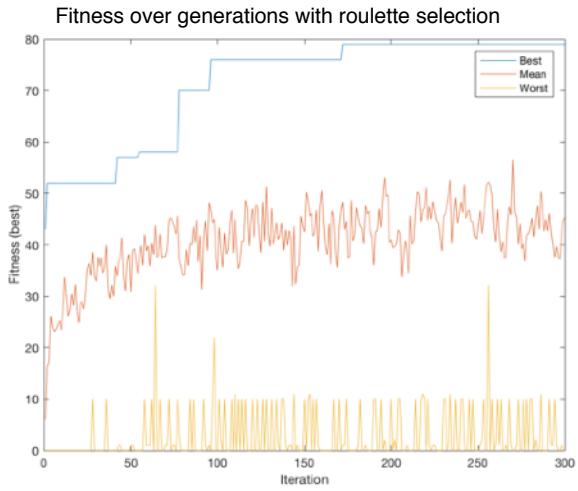
Random mutation:





Order changing mutation:



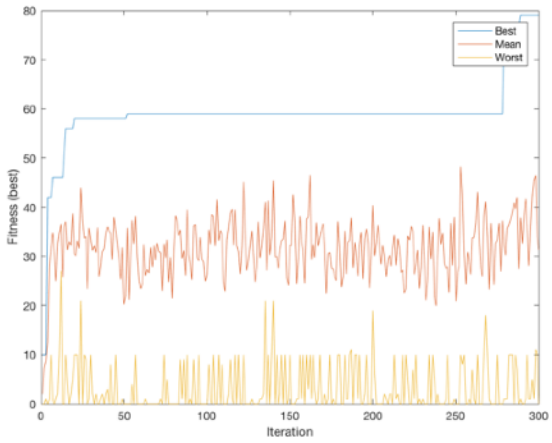


Appendix 5: Population size comparison results

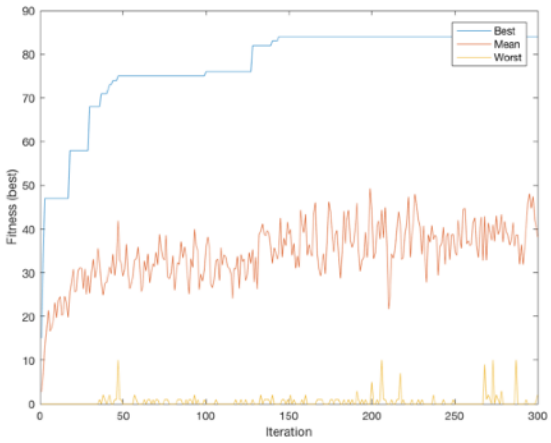
	Roulette selection						Rank selection					
	Population size											
	10		30		50		10		30		50	
Attempt	Best	Average (mean)	Best	Average (mean)	Best	Average (mean)	Best	Average (mean)	Best	Average (mean)	Best	Average (mean)
1	58	23.5	83	34	83	35.7	68	27	84	36.4	83	35.1
2	59	34.4	82	33.1	83	29	81	47.3	79	29.3	81	35.6
3	81	47.7	82	43.9	83	30.9	74	38.1	82	34.5	84	28.2
4	58	44.7	62	26.6	84	30.7	58	29.7	82	24.8	82	30.8
5	52	41	79	26.4	84	27.7	58	28	81	23	83	30.8
6	83	48.4	81	39.3	84	33.1	64	32.2	83	20	81	33.1
7	71	27	82	37.5	82	38.9	58	32	85	25	84	27.5
8	75	49.9	81	46.6	84	42	83	37	84	38	83	32.3
9	81	59.5	83	35.9	84	38.9	82	43.6	81	39.2	83	37
10	73	36	81	42.2	84	41.5	80	50.6	84	39	82	19.7
Average (mean):	**69.1**	**41.21**	**79.6**	**36.55**	**83.5**	**34.84**	**70.6**	**36.55**	**82.5**	**30.92**	**82.6**	**31.01**
	* collected over 300 generations with random mutation with with mutation and crossover probability of 1. Best and average are sampled from the final generation.											

Appendix 6: Population size comparison plotting

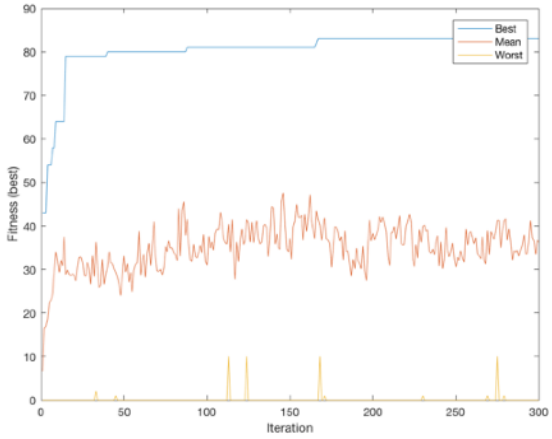
Fitness over generations with 10 population (roulette)



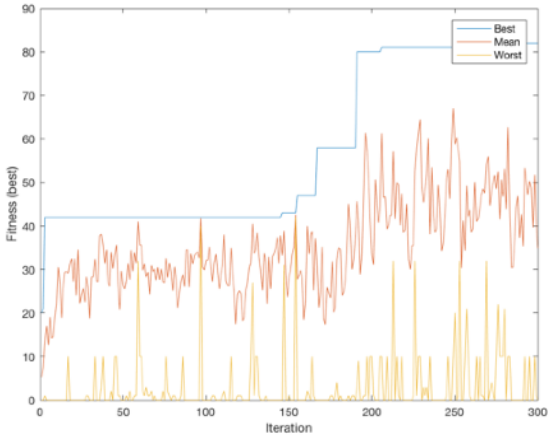
Fitness over generations with 30 population (roulette)



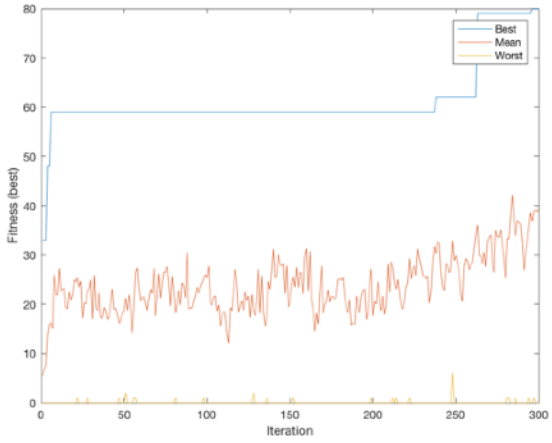
Fitness over generations with 50 population (roulette)



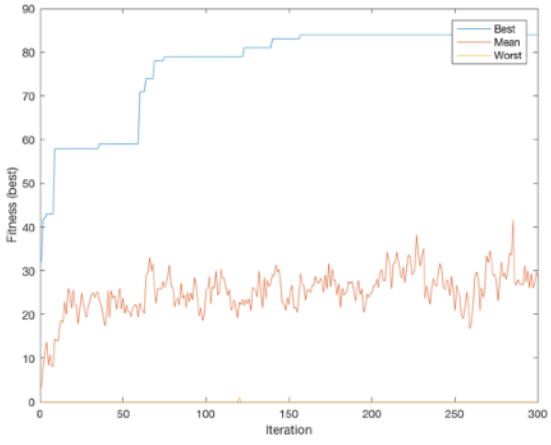
Fitness over generations with 10 population (ranked)



Fitness over generations with 30 population (ranked)



Fitness over generations with 50 population (ranked)

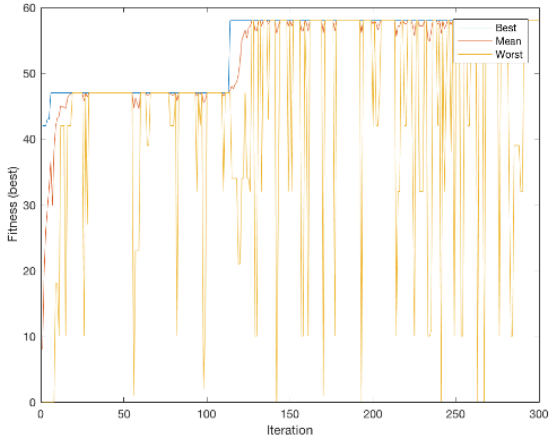


Appendix 7: Mutation probability comparison results

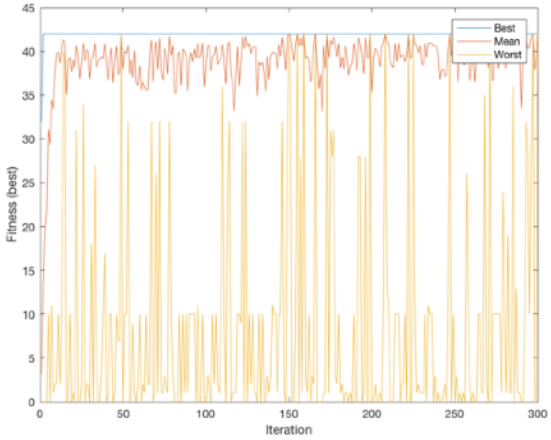
	Mutation probability									
	0.01		0.2		0.5		0.8		1.0	
Attempt	**Best**	**Average (mean)**	**Best**	**Average (mean)**	**Best**	**Average (mean)**	**Best**	**Average (mean)**	**Best**	**Average (mean)**
1	42	42	80	67.8	81	62.4	81	55.4	80	31.8
2	52	52	62	50.5	80	51.5	83	38.3	81	29.9
3	42	42	57	47.2	80	49.3	82	51.6	81	40
4	47	47	42	38.8	81	51.9	85	49.2	83	29.9
5	52	51.9	79	70.8	81	52	84	49.7	84	28.8
6	42	42	81	71.7	58	37.3	83	45.1	81	50
7	43	43	64	50.5	81	54.7	80	47.2	83	35.1
8	43	43	76	67.4	83	59.9	84	41.4	84	38.1
9	42	42	59	47.2	82	57	77	35.2	79	33.3
10	58	58	58	55.2	79	46.2	81	47.7	83	46.3
Average (mean):	**46.3**	**46.29**	**65.8**	**56.71**	**78.6**	**52.22**	**82**	**46.08**	**81.9**	**36.32**
	* collected using population of 30 over 300 generations with roulette wheel selection and random mutation with a crossover probability of 1. Best and average are sampled from the final generation.									

Appendix 8: Mutation probability comparison plotting (roulette selection)

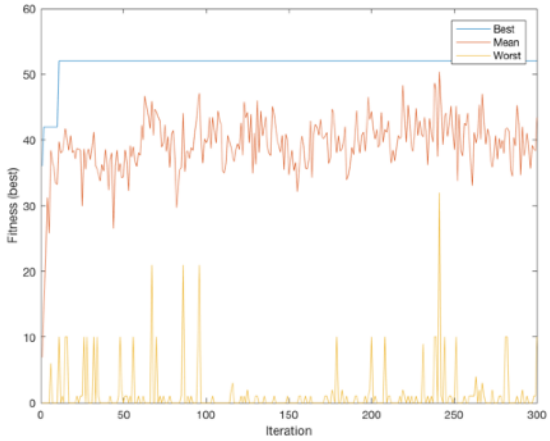
0.01 mutation probability



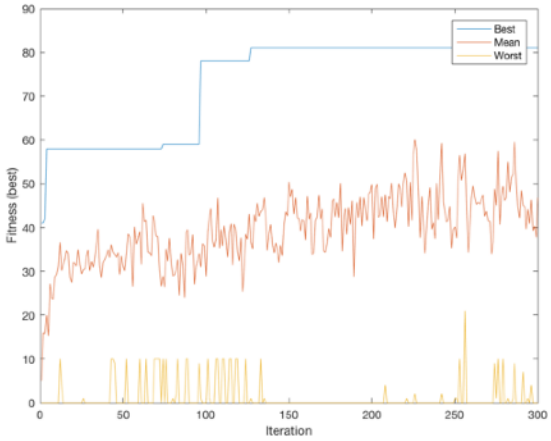
0.2 mutation probability



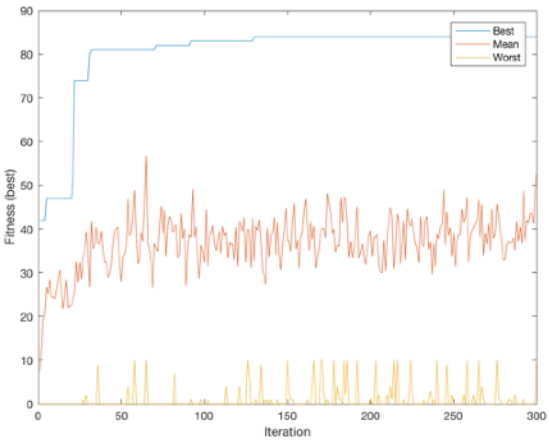
0.5 mutation probability



0.8 mutation probability



1.0 mutation probability



Appendix 9: Example paths using roulette selection and order changing mutation