ECE457A - Assignment 3

Written by: Anson Wan

Due Date: July 9th 2022

(I was granted an extension to July 11th, 2022 by Prof. Mehrannia due to a concussion injury)
(Doctors note was given to Prof. Mehrannia via email)

Question 1:

1.1. Tabu Search Source Code

Please see *a23wan_a3_q1.py* attached for the full source code.

To implement Tabu Search (TS) a priority queue was used to hold all the generated neighbors, their costs, and the swapped indices that caused it. The priority queue was sorted by their costs, this way, the first element in the priority queue would always be the most optimal neighbor. This is required when implementing TS as the most optimal neighbor is always used to compare with the currently best seen solution.

As for the Tabu List, a simple queue is used for its implementation. The length of this list acts as the tabu tenure since every iteration, a move is pushed into the Tabu List. If the tabu tenure is equal to the size of the tabu list, that means the first element can be removed and the new move can be added to the list.

For frequency memory, a dictionary was used to hold the solution and the count of how many times it was visited. Then, if that same solution is generated as a neighbor, the cost of that solution is added with the frequency count of how many times it has been seen, biasing the priority queue to not select it since it orders the neighbors based on cost.

The default parameters chosen are number of iterations=300, and tabu tenure=10. 300 iterations to allow the search to get an ample amount of iterations to find the most optimal solutions. Tabu Tenure length of 10 was simply an arbitrary value chosen mainly due to it being half of the solution length.

1.2. Running the Basic Tabu Search

Running the Tabu Search with the default parameters explained in 1.1, the result is:

> Initial Solution: [4, 0, 5, 13, 15, 18, 3, 16, 11, 1, 17, 8, 6, 12, 7, 9, 10, 14, 19, 2], Cost 3480
> Best Solution: [16, 7, 10, 3, 15, 12, 19, 14, 18, 8, 4, 6, 11, 1, 13, 5, 0, 9, 17, 2], Cost 2588

This will be referenced as the “control” run when comparing the results of the modifications later.

1.3. Modifying the Basic Tabu Search

Unless otherwise specified, the tabu tenure = 10, and the number of iterations = 300.

1.3.1 Changing the initial starting solution 10 times.

Changing the initial starting point (initial solution) 10 times results in:

Initial Solution: [15, 10, 11, 17, 3, 0, 1, 4, 5, 8, 9, 7, 13, 19, 16, 14, 2, 6, 12, 18], Cost 3322
Best Solution: [8, 15, 10, 3, 12, 13, 1, 14, 19, 18, 17, 11, 7, 6, 4, 2, 9, 0, 16, 5], Cost 2622

Initial Solution: [6, 18, 17, 14, 13, 3, 5, 19, 2, 0, 11, 9, 16, 4, 12, 10, 7, 15, 8, 1], Cost 3520
Best Solution: [17, 1, 13, 2, 8, 18, 14, 11, 9, 15, 3, 7, 10, 4, 12, 16, 19, 6, 0, 5], Cost 2580

Initial Solution: [5, 6, 14, 10, 11, 18, 17, 7, 19, 15, 12, 2, 3, 4, 1, 0, 9, 13, 8, 16], Cost 3384
Best Solution: [5, 0, 6, 4, 16, 12, 7, 19, 14, 18, 15, 10, 11, 1, 3, 8, 2, 13, 9, 17], Cost 2578

Initial Solution: [8, 0, 4, 18, 14, 11, 10, 7, 15, 3, 9, 1, 5, 17, 12, 13, 19, 6, 2, 16], Cost 3336
Best Solution: [18, 1, 13, 15, 8, 17, 14, 11, 9, 2, 3, 19, 7, 10, 12, 16, 6, 4, 0, 5], Cost 2582

Initial Solution: [12, 6, 3, 13, 19, 7, 9, 5, 11, 8, 14, 17, 2, 18, 15, 0, 10, 1, 16, 4], Cost 3492
Best Solution: [16, 0, 6, 4, 5, 10, 7, 19, 9, 12, 3, 14, 11, 17, 2, 15, 18, 1, 13, 8], Cost 2602

Initial Solution: [10, 4, 11, 7, 2, 3, 0, 6, 16, 9, 15, 14, 17, 19, 12, 1, 18, 13, 8, 5], Cost 3030
Best Solution: [16, 19, 6, 0, 5, 3, 7, 10, 4, 12, 18, 14, 11, 9, 15, 17, 1, 13, 2, 8], Cost 2580

Initial Solution: [16, 0, 17, 19, 5, 1, 15, 9, 2, 10, 6, 3, 12, 14, 8, 7, 4, 18, 13, 11], Cost 3576
Best Solution: [17, 1, 13, 2, 8, 18, 14, 11, 9, 15, 3, 19, 7, 10, 12, 16, 6, 4, 0, 5], Cost 2574

Initial Solution: [19, 0, 10, 3, 7, 18, 13, 2, 12, 11, 4, 16, 17, 8, 5, 6, 15, 14, 9, 1], Cost 3630
Best Solution: [5, 0, 4, 6, 16, 12, 10, 7, 19, 3, 15, 9, 11, 14, 18, 8, 2, 13, 1, 17], Cost 2574

Initial Solution: [6, 14, 13, 5, 4, 8, 12, 1, 0, 19, 17, 16, 18, 2, 9, 15, 11, 7, 10, 3], Cost 3374
Best Solution: [5, 0, 9, 2, 8, 4, 6, 11, 1, 13, 12, 19, 14, 18, 17, 16, 7, 10, 3, 15], Cost 2598

Initial Solution: [2, 11, 4, 12, 1, 19, 8, 9, 7, 5, 14, 15, 10, 16, 3, 17, 13, 6, 0, 18], Cost 3440
Best Solution: [16, 4, 6, 0, 5, 18, 14, 19, 7, 12, 3, 1, 11, 10, 15, 17, 13, 9, 2, 8], Cost 2570
The average cost of these 10 runs is: 2586

Comparing these results, we see that changing the initial solution does not change the effectiveness of the Tabu Search, the final cost result averages out to be 2586 which is essentially the same as the control run.

1.3.2. Changing the Tabu Tenure Length

Changing the Tabu List size twice, once to 10 and another to 20 (originally 15) results in:

Tabu Tenure = 5:

Initial Solution: [16, 6, 9, 2, 5, 11, 7, 18, 3, 12, 13, 1, 10, 17, 14, 4, 8, 19, 0, 15], Cost 3276

Best Solution: [16, 6, 4, 0, 5, 3, 19, 7, 10, 12, 18, 14, 11, 9, 15fffs, 17, 1, 13, 2, 8], Cost 2574

Running 10 times and calculating the average cost results in: 2594.4

Tabu Tenure = 15:

Initial Solution: [3, 8, 0, 6, 19, 13, 9, 16, 2, 12, 11, 18, 10, 17, 1, 15, 7, 5, 4, 14], Cost 3468

Best Solution: [2, 0, 6, 5, 12, 8, 9, 11, 19, 4, 17, 13, 1, 14, 18, 15, 10, 3, 7, 16], Cost 2622

Running 10 times and calculating the average cost results in: 2601.4

Comparing these results with the control run, we see that the results are worse when increasing or decreasing the tabu tenure from the default of 10. This shows that there likely needs to be a balance in the tenure size such that restrictions on allowed moves are not too persistent.

1.3.3. Using a Dynamic Tabu Tenure Length

The dynamic tabu list size was implemented so that the list size changes every 50 iterations. On the 50th iteration, a random number between 3 and 20 is generated to be the new tabu tenure length.

Using a dynamic tabu list size results in:

Initial Solution: [0, 16, 19, 14, 15, 8, 18, 1, 12, 17, 3, 5, 4, 9, 7, 6, 10, 2, 13, 11], Cost 3398

Best Solution: [5, 0, 6, 4, 16, 12, 7, 19, 14, 18, 15, 10, 11, 1, 3, 8, 2, 9, 13, 17], Cost 2570

Running 10 times and calculating the average cost results in: 2591

Comparing this to the control run, the result of the dynamic tabu list solution was very slightly worse (only worse by 2). This may be due to the method of changing the tabu tenure length. Since I chose a random length as the new tabu tenure, it is likely that the TS got to experience long and short tabu tenures over the course of the run and thus the benefits and drawbacks of each eventually average out and the final result simply ends up being extremely close to the control run.

1.3.4. Adding Aspiration Criteria

There were two aspiration criteria tested. The first being the “best solution so far” and the second being the “best solution in the neighborhood”

The “best solution so far” would allow a tabu move if it resulted in a solution that had a better solution than the best solution found in the whole search thus far.

The “best solution in the neighborhood” would allow a tabu move if it resulted in a solution better than the best solution in the neighborhood.

Using a best solution so far aspiration criteria results in:

Initial Solution: [8, 17, 11, 19, 15, 18, 16, 0, 5, 12, 2, 6, 13, 7, 1, 14, 3, 4, 9, 10], Cost 3498

Best Solution: [17, 1, 13, 2, 8, 18, 14, 11, 9, 5, 3, 19, 6, 4, 12, 16, 7, 10, 0, 15], Cost 2574

Running 10 times and calculating the average cost results in: 2582.2

Using a best solution in the neighborhood aspiration criteria results in:

Initial Solution: [19, 11, 8, 0, 10, 9, 16, 17, 1, 18, 6, 13, 4, 15, 5, 3, 14, 7, 2, 12], Cost 3594

Best Solution: [16, 17, 8, 18, 15, 2, 9, 13, 1, 3, 0, 6, 11, 14, 10, 5, 4, 12, 19, 7], Cost 2666

Running 10 times and calculating the average cost results in: 2675.2

Comparing the “‘best solution so far” AC run with the control run, the result was reasonably superior. This makes sense as the best solution is always taken even if the move is tabu thus, there is no opportunity for the TS to “miss” an optimal solution whereas in the standard TS, if an optimal solution is a result of a tabu move, it is ignored and its possible it will not be generated as a neighbor again.

Comparing the “best solution in neighborhood” AC with the control run, the result was considerably worse. This is because this AC always accepts the best neighbor regardless of it being a result of a tabu move, and since we always take the best neighbor anyways this essentially removes the purpose of the tabu list and the TS behaves like hill-climbing search as it never takes any worsening solutions that can eventually move the search to look at a more promising space. Thus, the search would reach a local maximum and not explore, leading to a suboptimal solution as we see in the results.

1.3.5. Using less than the entire neighborhood to select the next candidate solution.

This was implemented by only generating half of the entire neighborhood. Thus, when selecting the next candidate, the best candidate of only half the entire neighborhood is used.

Using less than the whole neighborhood to select the next solution results in:

Initial Solution: [18, 13, 5, 6, 19, 11, 10, 14, 9, 0, 15, 17, 1, 7, 16, 8, 3, 2, 4, 12], Cost 3176

Best Solution: [2, 9, 0, 6, 5, 8, 13, 10, 11, 12, 15, 1, 14, 7, 4, 17, 18, 3, 19, 16], Cost 2638

Running 10 times and calculating the average cost results in: 2634

When using less than the whole neighborhood to select the next candidate solution, we see that the result is considerably worse than the control run. This is because only half of the neighbors are considered and there is a significantly lower chance that an optimal solution is found in each iteration when they are generated and compared with the currently best found solution. Another observation however is that the search performed much faster, so there is a trade-off between speed and optimality of the solution.

1.3.6. Using a Frequency-based Tabu Search

The frequency bias was implemented by having a dictionary/map that contained all seen solutions as the key and a count of how many times it was visited as the value. Then when generating the neighbors, the count of how many times it was visited would be multiplied by 10 and added to the cost of the solution so that in the priority queue, it would bias solutions that were not seen before to be in the front and thus be taken as the next candidate solution.

Using a frequency based tabu list resulted in:

Initial Solution: [7, 6, 3, 9, 4, 10, 13, 16, 15, 5, 2, 18, 17, 11, 0, 14, 8, 1, 12, 19], Cost 3592

Best Solution: [16, 19, 6, 0, 5, 3, 7, 10, 4, 12, 18, 14, 11, 9, 15, 17, 1, 13, 2, 8], Cost 2580

Running 10 times and calculating the average cost results in: 2589

Comparing this run to the control, the results were essentially the same (different by 1). Adding frequency bias encourages exploration as unseen solutions are given an incentive to be taken as the next candidate so it is possible that the problem space is not large enough to benefit heavily from adding a frequency based tabu list.

1.3.7. Table of Results

Table 1. Results of Various Tabu Search Modifications.

Tabu Search Modification	Average final solution cost of 10 Runs
Basic Tabu Search (Tabu Tenure = 10)	2586
Basic Tabu Search with Tabu Tenure = 5	2594.4
Basic Tabu Search with Tabu Tenure = 15	2601.4
Tabu Search with Dynamic Tabu Tenure	2591
TS with “best solution so far” AC	2582.2
TS with “best solution in neighborhood” AC	2675.2
TS using less than entire neighborhood	3060.2
Frequency Biased TS	2589

Question 2:

2.a.

The representation that is used for the solution of this problem is as follows:

$$S = [\, K_P,\, T_i,\, T_d \,],\ where\ K_p \in (2, 18),\ T_i \in (1.05, 9.42),\ T_d \in (0.26, 2.37)$$

The three parameters K_P, T_i, T_d directly affect the performance of a given solution and are real-values rounded to 2 decimal places. Since the values in the representation array are real-valued, it follows that Real-Valued Genetic Algorithm will be used. This representation is superior to a binary-encoded one as using binary encoding for a real-value would require a very long array (chromosome) representation which would result in difficult computation when performing crossover. Whereas using a real-value representation results in only 3 genes in the chromosome with mating and crossover being simpler.

2.b.

Within the domain of a PID controller, we know that a fitness function must be based on the performance of the system. It follows that the performance of a PID controller is based on the Integral Squared Error (ISE), the rise time (t_r), the step response (t_s), and the maximum overshoot magnitude (M_p). We want to minimize all of these performance measures thus, generally, we want all 4 of these values to be minimized for the solution to have the highest performance. When selecting a parent using Fitness Proportionate Selection (FPS) we want the best parents (solutions with lowest valued ISE, t_r, t_s, M_p) to have the highest chance of being selected, thus we take the inverse of the sum of the values as the fitness function.

Thus the fitness function for the RV-GA used is:

$$Fitness = \frac{10000}{ISE + t_r + t_s + M_p}$$

$$where,\ ISE = \int_0^T (e(t)^2)dt$$

$$t_r = rise\ time$$

$$t_s = settling\ time$$

$$M_p = maximum\ overshoot\ magnitude$$

The value was scaled by 10000 so the fitness value would be more readable.

2.c

See *a23wan_a3_q2.py* attached for full source code.

Real-Valued Genetic Algorithm (RVGA) was implemented to solve this problem. As required in the solution, the default value for the number of generations is 150, crossover probability is set to 0.6, and mutation probability is set to 0.25.

FPS parent selection was implemented by simulating a "roulette wheel" where the total performance/fitness was calculated of the entire population, and a random number was selected within that total, then a running sum was incremented with the fitness values of each individual of the population one-by-one until that sum was above or equal to the random number. If it was, that individual was selected as the parent. This mimics a roulette wheel because if the individual's fitness was higher, it would have a higher chance to increase the running sum's value past the threshold (the random number).

This implementation uses Gaussian Mutation for child mutation, where a random number within a normal distribution from 0 to sigma=1 is added to each element in the child.As for crossover, Whole Arithmetic Crossover for child generation was used.

In terms of data structures, a dictionary/map was used to hold children and their performance values. This map was then sorted to ensure that the best two children were always selected to persist into the next generation as required by elitism survivor selection.

2.d

The fitness of the best solution for each generation is plotted below in Figure 1.

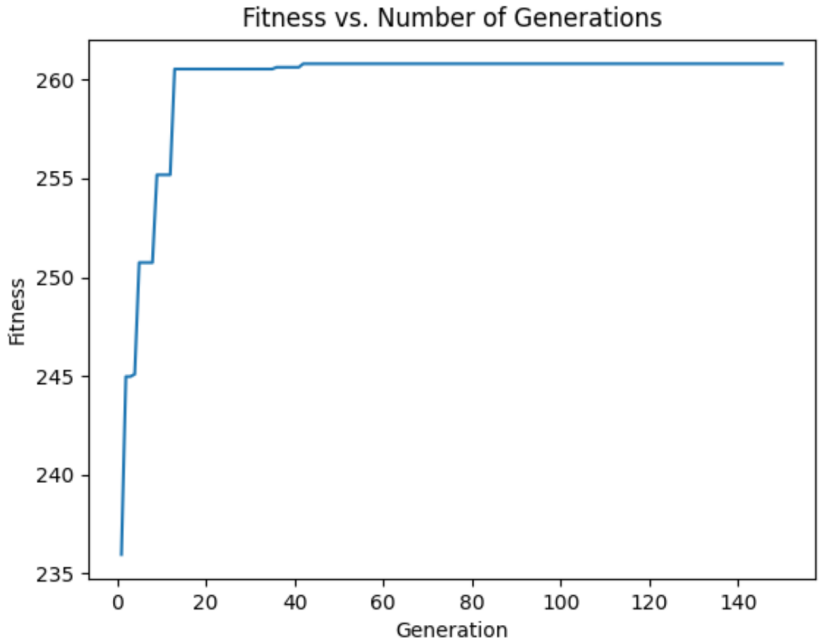

Figure 1. Best fitness per each generation, with RVGA run on default parameters

2.e

Running RVGA with number of generations = 50, 150, and 225 resulted in:

num_gens: 50, pop_size: 50, crossover_prob: 0.6, mutation_prob: 0.25
Initial performance: 230.6418101406067, Parameters: [3.99, 9.35, 2.15]
Final performance: 260.6226896796177, Parameters: [4.62, 5.92, 2.36]
num_gens: 150, pop_size: 50, crossover_prob: 0.6, mutation_prob: 0.25
Initial performance: 240.89548566663169 , Parameters: [5.26, 6.6, 1.97]
Final performance: 260.7697176356946, Parameters: [4.63, 5.82, 2.36]
num_gens: 225, pop_size: 50, crossover_prob: 0.6, mutation_prob: 0.25
Initial performance: 258.21235463243625, Parameters: [4.59, 5.66, 2.33]
Final performance: 260.68879232007026, Parameters: [4.62, 5.88, 2.36]

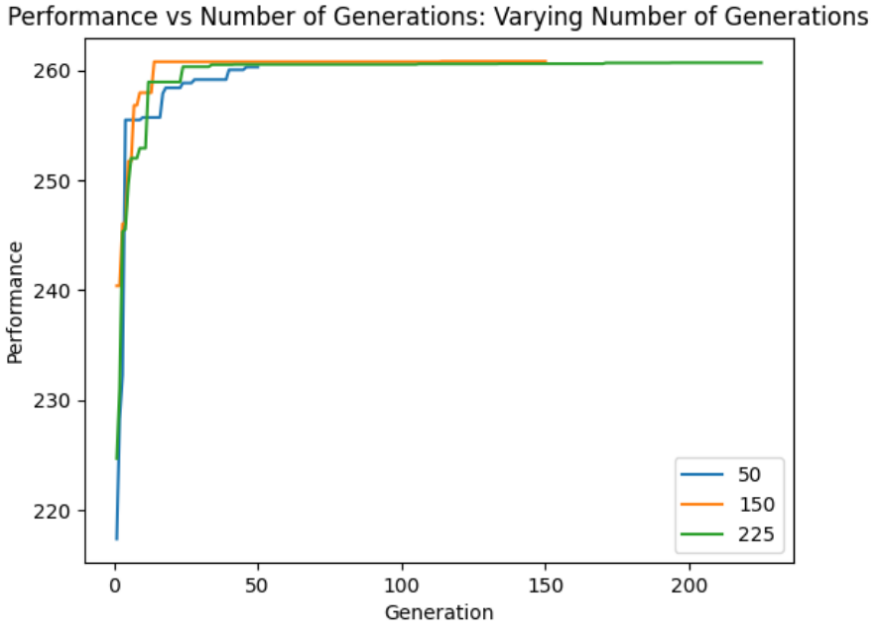

Figure 2: Performance vs Number of Generations, with varied number of generations

Comparing the results of varying the number of generations, we can see that the results are relatively similar as they all converged near a value of 260 (see Figure 2). Unsurprisingly, we see that the runs with a higher number of generations resulted in the better solution even if it was only a negligibly superior solution. This makes sense as there are more iterations (generations) for a new child with a superior fitness to appear. Knowing this however, we see that the run where num_gens = 150 had a superior solution to the one with num_gens = 225. This may show that there is a point in the algorithm where finding a superior solution may come down to luck/randomness even though more iterations should eventually lead to a better solution. Additionally, the greater the number of generations, the longer the algorithm had to run. This means there is a balance between time vs increased probability of finding a new solution that must be taken into account when selecting the optimal number of generations. As for convergence speed, as we can see from Figure 2, all three runs converged at similar speeds.

2.f

Running RVGA with population size = 25, 50, and 85 resulted in:

num_gens: 150, pop_size: 25, crossover_prob: 0.6, mutation_prob: 0.25
Initial performance: 216.31086296961317, Parameters: [3.97, 8.39, 1.41]
Final performance: 260.68879232007026, Parameters: [4.62, 5.88, 2.36]

num_gens: 150, pop_size: 50, crossover_prob: 0.6, mutation_prob: 0.25
Initial performance: 240.89548566663169 , Parameters: [5.26, 6.6, 1.97]
Final performance: 260.7697176356946, Parameters: [4.63, 5.82, 2.36]

num_gens: 150, pop_size: 85, crossover_prob: 0.6, mutation_prob: 0.25
Initial performance: 256.69053515891216, Parameters: [4.67, 5.44, 2.24]
Final performance: 260.7594988353001, Parameters: [4.67, 5.42, 2.36]

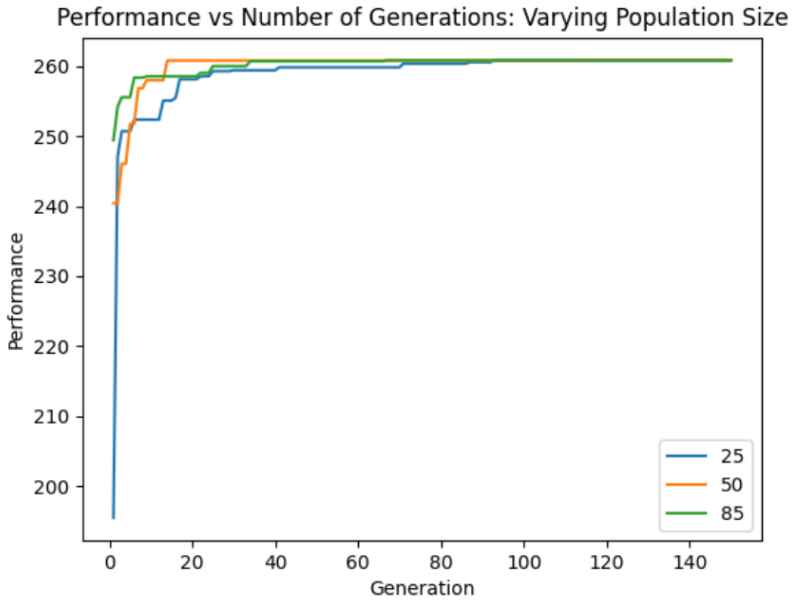

Figure 3: Performance vs Number of Generations, with varied population size

Comparing the results of varying the population size, we see that all the results converge to a fitness value of 260 and are relatively similar. Unsurprisingly, the best results are found using a larger population size. This is intuitive as a larger population will allow for a larger sample to be compared for the best solution. From that however, we see that the run where pop_size=50 had a superior solution to the run with pop_size=85. This may show that nearing the end of the algorithm, finding a superior child is likely a result of randomness or luck more so than the population size, even though a larger population size should help aid it. Additionally, the larger the population size, the longer the algorithm took to run. Thus, when choosing a population size, time to run vs increased probability to find a solution must be weighted. As for convergence speed, as we can see from Figure 3, all three runs converged at a similar speed.

2.g

Running RVGA with crossover probability = 20% and 90% resulted in:

num_gens: 150, pop_size: 50, crossover_prob: 0.2, mutation_prob: 0.25
Initial performance: 248.49512939603983, Parameters: [4.41, 4.73, 2.32]
Final performance: 260.81338008293113, Parameters: [4.64, 5.72, 2.36]

num_gens: 150, pop_size: 50, crossover_prob: 0.6, mutation_prob: 0.25
Initial performance: 240.89548566663169 , Parameters: [5.26, 6.6, 1.97]
Final performance: 260.7697176356946, Parameters: [4.63, 5.82, 2.36]

num_gens: 150, pop_size: 50, crossover_prob: 0.9, mutation_prob: 0.25
Initial performance: 232.16059750365926, Parameters: [5.26, 6.85, 1.61]
Final performance: 260.79984998209676, Parameters: [4.67, 5.41, 2.36]

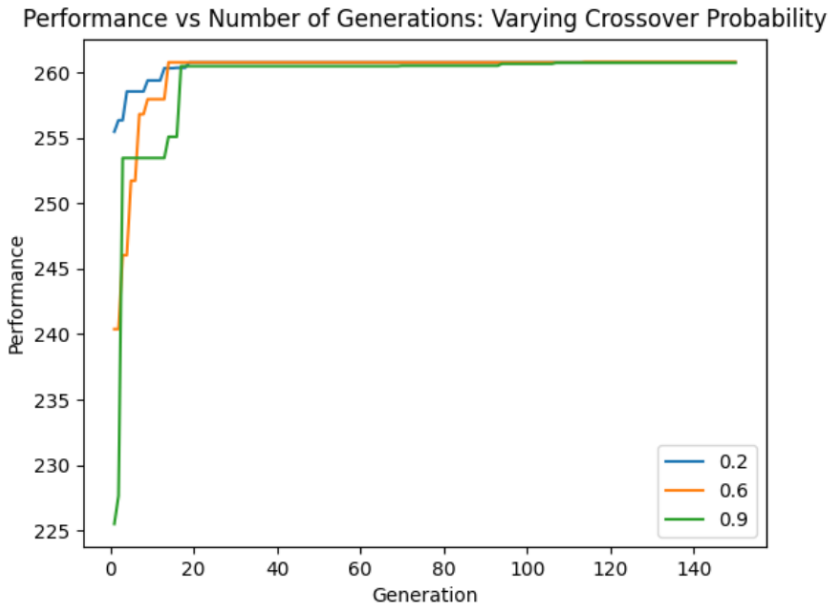

Figure 4: Performance vs Number of Generations, with varied crossover probability

Comparing the results of varying the crossover probability, we see that generally, all three runs converged to a value around 260. There is not an obvious correlation between crossover probability with finding the most optimal solution as it is seen that decreasing crossover probability to 20% and increasing it to 90% both resulted in better solutions to the original 60%. It is known that crossover is explorative, so perhaps the problem place is too small for changes in crossover (exploration) to have an effect since the entire space will be checked regardless. As for convergence speed, from Figure 4, we can see that the higher the crossover, the slower the convergence speed, this makes sense as crossover is explorative, and if the probability of crossover is lower, the algorithm will try to converge around the current best solution rather than explore.

2.h

Running RVGA with mutation probability = 5% and 50% resulted in:

num_gens: 150, pop_size: 50, crossover_prob: 0.6, mutation_prob: 0.05
Initial performance: 222.7087118998194, Parameters: [3.94, 8.97, 1.74]
Final performance: 260.7946352979925, Parameters: [4.63, 5.8, 2.36]

num_gens: 150, pop_size: 50, crossover_prob: 0.6, mutation_prob: 0.25
Initial performance: 240.89548566663169 , Parameters: [5.26, 6.6, 1.97]
Final performance: 260.7697176356946, Parameters: [4.63, 5.82, 2.36]

num_gens: 150, pop_size: 50, crossover_prob: 0.6, mutation_prob: 0.5
Initial performance: 239.01470955546796, Parameters: [5.41, 4.48, 1.96]
Final performance: 260.74988913526926, Parameters: [4.67, 5.45, 2.36]

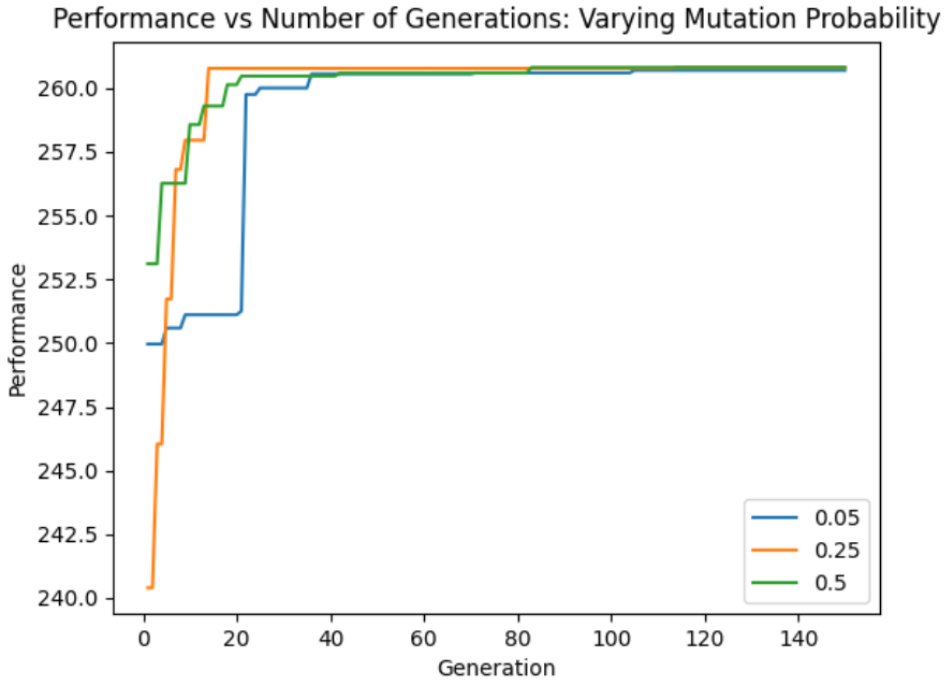

Figure 5: Performance vs Number of Generations, with varied mutation probability

Comparing the results of varying the mutation probability, we see that again, the results are similar and all are converging towards a fitness of 260. There appears to be a slight trend where a lower mutation probability results in superior final results although it seems to be negligible. As for convergence speed, we see that when the mutation probability was the smallest (5%) the algorithm converges the slowest. This makes sense as mutation is exploitative in nature, and if the probability to mutate is low, there is a lower chance for that child to change slightly and have a better performance/fitness which results in slower convergence.

2.1 Table of Results

The results of runs 2.e - 2.h are placed into Table 2 below:

Table 2: Results of RVGA with varying parameters.

Parameter Changed	Final Performance	Solution
Number of Generations = 50	260.622689	[4.62, 5.92, 2.36]
Number of Generations = 150	260.769717	[4.63, 5.82, 2.36]
Number of Generations = 225	260.688792	[4.62, 5.88, 2.36]
Population Size = 25	260.688792	[4.62, 5.88, 2.36]
Population Size = 50	260.769717	[4.63, 5.82, 2.36]
Population Size = 85	260.759498	[4.67, 5.42, 2.36]
Crossover Probability = 20%	260.813380	[4.64, 5.72, 2.36]
Crossover Probability = 60%	260.769717	[4.63, 5.82, 2.36]
Crossover Probability = 90%	260.799849	[4.67, 5.41, 2.36]
Mutation Probability = 5%	260.794635	[4.63, 5.8, 2.36]
Mutation Probability = 25%	260.769717	[4.63, 5.82, 2.36]
Mutation Probability = 50%	260.749889	[4.67, 5.45, 2.36]

Question 3:

The parameters that are to be adjusted and observed are the following:

- Population (30,50,100)
- Diffusion Rate (40, 80). This is the amount of pheromone deposited in the trails (Q in the ant density model)
- Evaporation Rate (10, 20). The rate at which pheromone disappears.
- Placement of the food sources (far, close, medium)

To test the effects of each parameter, the food locations are kept constant. One food stack is placed close, another somewhat close, and a third placed far from the ant hole. Food locations are shown in Figure 6. Additionally, the term “performance” will be used to refer to the speed at which the ants deplete all the food.

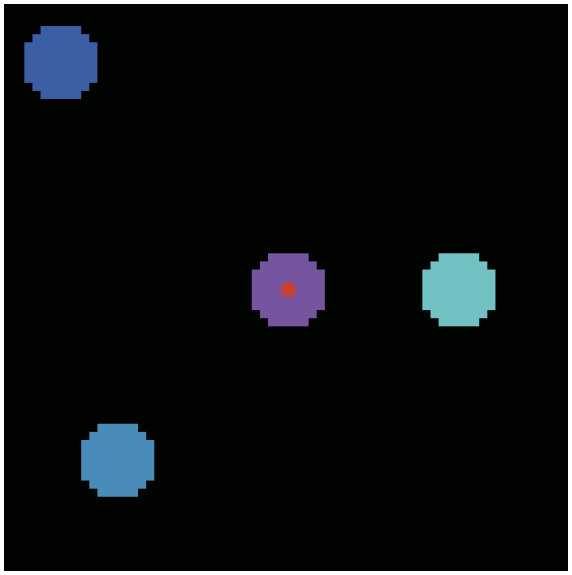

Figure 6. Location of Food Sources

The result of the experiment are are as follows in Table 3:

Table 3: Results of Ant Colony Simulation Experiment

Population	Diffusion Rate	Evaporation Rate	Time to Finish Food (Ticks)
30	40	10	8400
30	40	20	7212
30	80	10	7811
30	80	20	7307
50	40	10	5418
50	40	20	4512
50	80	10	6256
50	80	20	5225
100	40	10	1771
100	40	20	3180
100	80	10	1740
100	80	20	3739

With these results, the effects of changing population, diffusion rate, and evaporation rate can be analyzed.

3.1 Effect of Population

Starting with the effects of population, we see that there is a general trend that if population increases, performance increases. This makes sense because if there are more ants, there are more possibilities to find food, and thus there are more possibilities to excrete pheromones. Additionally, there are more ants that can find an existing pheromone trail which can help it find more food, and the loop continues. This means having a higher population allows for more opportunities to create a positive feedback loop and thus, a consistent stream of pheromones and ants is formed between a food source and the ant hill. This was in fact observed as when viewing the simulations with a population value of 30 and 50, no matter what the diffusion or evaporation rates were, the furthest food source never got a consistent stream of ants (see Figure 7) whereas with a population value of 100, there was always a consistent stream of pheromone and ants from the farthest food source to the ant hill (see Figure 8).

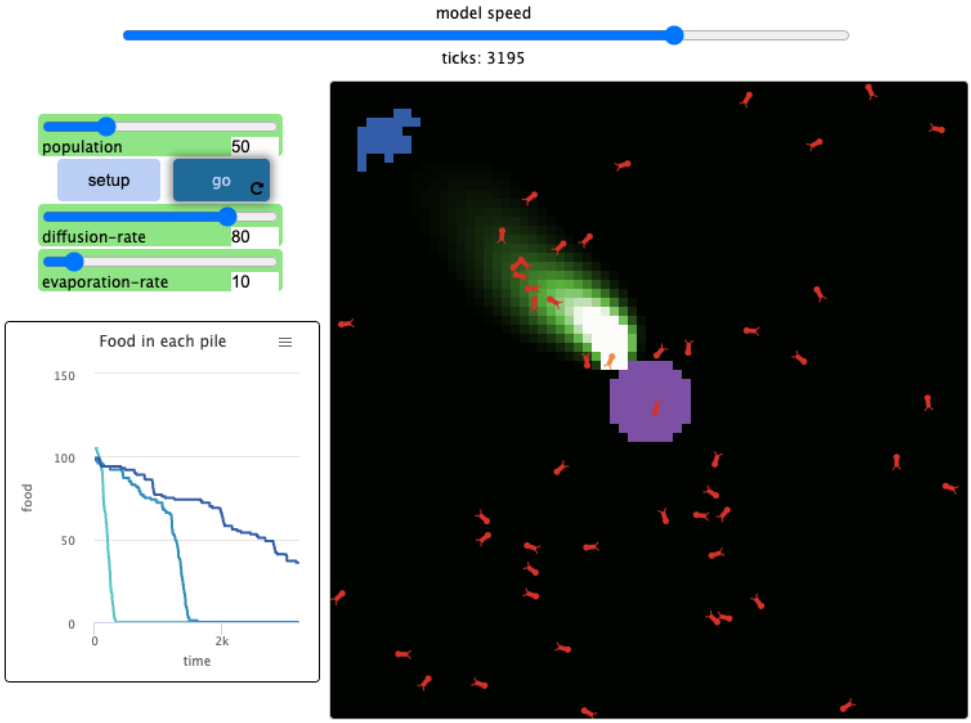

Figure 7. No consistent pheromone trail to furthest food source with population 50.

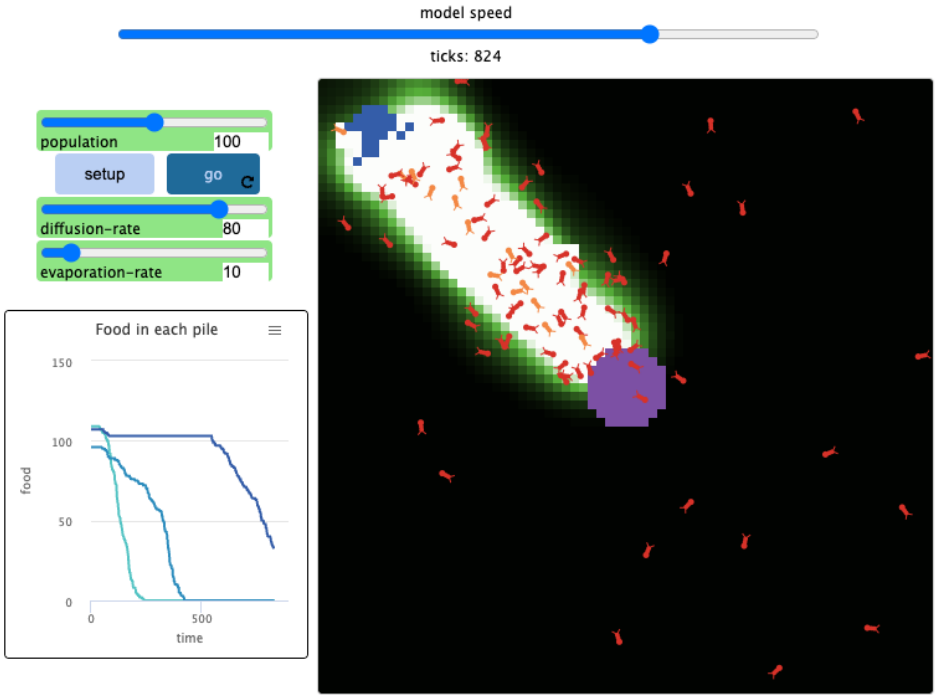

Figure 8. Strong and consistent pheromone trail to furthest food source with population 100.

3.2 Effect of Diffusion Rate

As for diffusion rate, when diffusion rate is increased from 40 to 80, there is a negligible effect to the time it takes for the ants to finish the food. Sometimes performance increases and sometimes it decreases. This is because when the pheromone area gets larger, it may bring more ants to follow the one with food back to the ant hill, decreasing exploration. On the flipside however, a larger pheromone area will result in easier pheromone bridge construction between the food source and the ant hill. These two results of the larger pheromone area result in differing performance results depending on which situation arises.

3.3 Effect of Evaporation Rate

For evaporation rate, in the experiments with populations of 30 and 50, increasing evaporation rate leads to better performance whereas when the population is 100, increasing evaporation rate actually causes a significant drop in performance. To explain why increasing evaporation leads to better performance in the lower populations, it is likely because there is a lower likelihood that a nearby ant will follow another back to the ant hill, thus allowing it to explore further out and find the food source. If there is a high population however, we know that pheromone bridges are built from all food sources, and thus increasing evaporation rate will lead to a pheromone bridge that is harder to build and sustain since the pheromone disappears quicker and if the pheromone bridge cannot be built or breaks, it will lead to a huge drop in performance.

3.4 Effect of Food Source Location

Now testing different food source locations, we will try putting all the food sources equally far from the ant hill. The parameters are set as population=100, diffusion rate=80, evaporation rate=10. These parameter values were chosen to promote pheromone bridge building between the food sources and ant hill. When the food sources are equally far away, the ant colony converges towards depleting one first, then the next (see Figure 9 on next page). Whereas when the food sources are close to the ant hill, they are both depleted at equal speed at the same time (see Figure 10 on next pag).

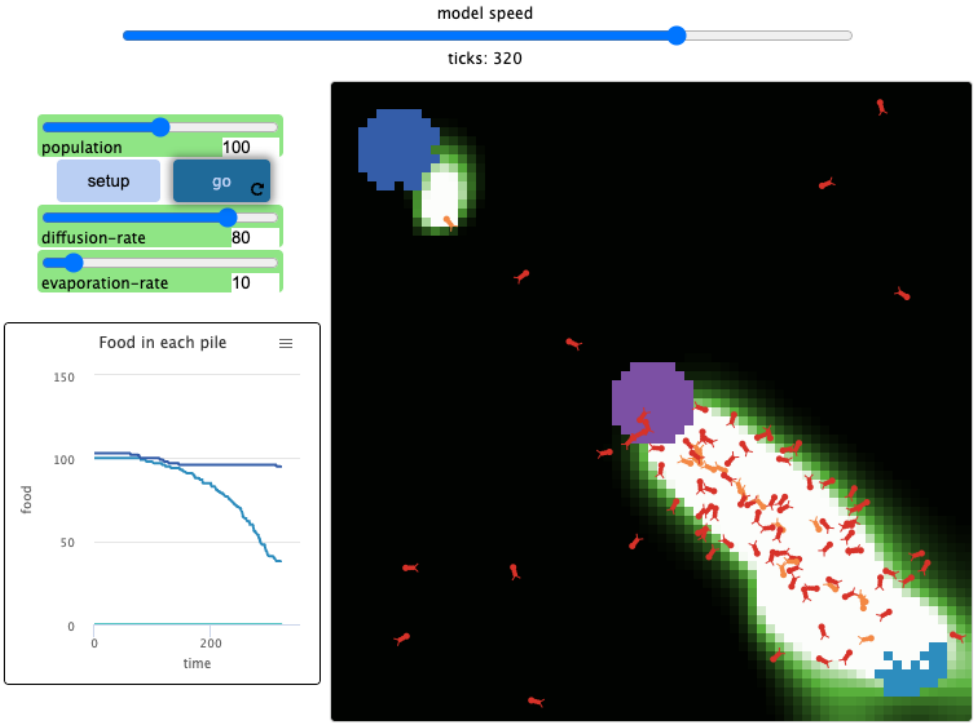

Figure 9. Two equal distance food sources far from ant hill.

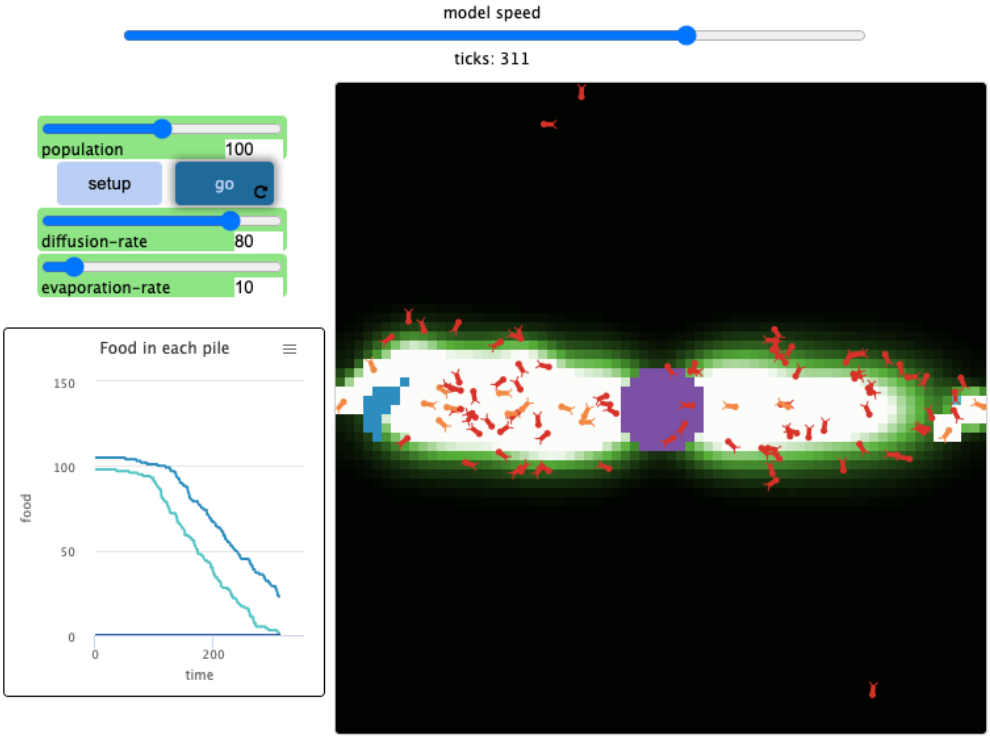

Figure 10. Two equal distance food sources close to ant hill.