Adrian Low
Dr. Wayne Wakeland
SYSC525
13 December 2020

Multi-Agent Genetic Algorithm for Vehicle Routing Problem

1. Abstract

This project aims at implementing a multi-agent genetic algorithm for the vehicle routing problem. A chromosome allowing for a dynamic number of vehicles to be searched is proposed and tested to see if a solution with the ideal number of vehicles emerges. In addition, a unique mutation operator for the proposed chromosome is tested. The crossover method, CX2, is slightly modified for the chromosome and is tested to see if the order of parents affects the system. A successful implementation of agents into the genetic algorithm process was created with the chromosome showing the ideal solution emerges most of the time with the ideal number of vehicles. The proposed mutation operator outperforms traditional genetic algorithm mutation in the program in its ability to introduce new solutions to the system. The crossover used has a better average fitness when the fittest parent is used as Parent 1 in the algorithm. Finally, future considerations are discussed that may help strengthen the implementation and validate results.

2. Background

How do companies like Amazon, FedEx, and Pepsi manage to deliver to thousands of customers daily in a timely manner? Logistics and transportation departments have employees dedicated to routing optimization figuring out the best way to allocate vehicles and decide the paths they take. Another name for this problem is the vehicle routing problem or more generally the traveling salesman problem. To develop a heuristic solution to the vehicle routing problem, genetic algorithms are chosen due to its ability to solve optimization problems. One problem with genetic algorithms is early convergence, so agent-based systems are combined to avoid this issue. This section gives background information on topics utilized to create this project and why they were selected: the vehicle routing problem, genetic algorithms, and multi-agent genetic algorithms.

2.1. Motivation

This system was chosen because of previous experience working in operations for a distribution company. During my time there, I started to learn about how the distribution routes are created and begun to take an interest in learning about the backend algorithms used in the system. Building off my previous knowledge of graph theory, I currently have a strong interest in machine learning and artificial intelligence, so genetic algorithms as a search heuristic is a new approach I wanted to explore.

2.2. Vehicle Routing Problem

The traveling salesman problem is a graph traversal problem that tries to find the most optimal loop that goes through every node in a graph once and returns to the starting node. The vehicle routing problem is similar, but instead of one traveler, there is a fleet of vehicles to divide and traverse nodes (see fig. 1). An optimum considers factors such as time and distance, but in more advanced vehicle routing problems there are other factors like delivery time windows, multiple depots, and pickups in addition to deliveries. There is no efficient algorithm to find an optimum solution, and the problem is classed as NP-hard. Instead of using bruit-force method of exhaustively testing every single possible loop, some heuristic algorithms attempt to find a quick good-enough solution. One of the heuristic approaches involves the use of genetic programming.

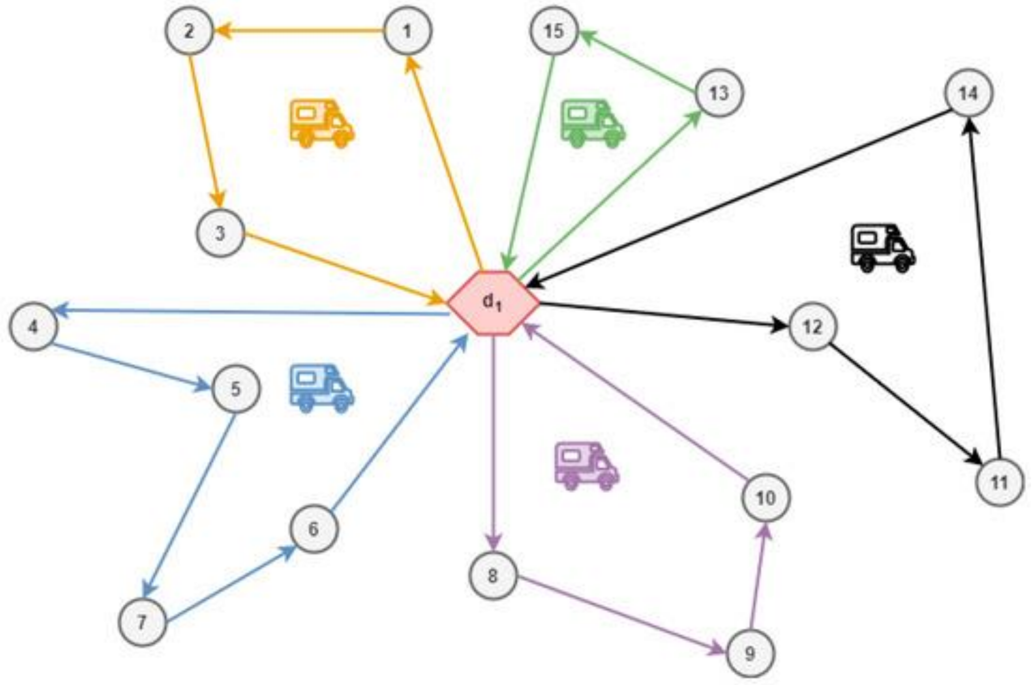


Fig 1: Diagram of the Vehicle Routing Problem. 5 vehicles divide the set of 15 stops to deliver to all starting and ending at the depot (Kovács et al.).

2.3. Genetic Algorithms

Genetic algorithms (GA) utilize strategies that mimic evolutionary processes in Nature such as selection, reproduction, and mutation (see fig. 2). The population of individuals starts off with random features that get evaluated to determine everyone's fitness with only those fit enough moving on to new generations. By selecting certain fit individuals with criteria determined by the selection method, it allows favorable features to persist and be exploited in the population. Offspring are produced through crossover, a method that mixes the chromosomes of parents together in a way allowing new potential solutions to be explored. After new offspring are introduced, a small percentage of the whole population goes through mutation. Mutation makes a small change to the chromosome to prevent evolution from coming to a standstill, introducing some variation into the population. Selection, crossover, and mutation are usually repeated until a certain stopping condition while keeping track of individual's fitness along the way (Holland 66-72). Genetic algorithms are beneficial because it allows a large variety of possibilities to be explored in a solution space having each iteration become closer to the target solution. There are other heuristic algorithms that can also be used for optimization problems such as Ant-Colony optimization, however in "Ant Colony versus Genetic Algorithm based on Travelling Salesman Problem" by Alhanjouri and Alfarra, it has been shown that using genetic algorithms produce slightly better results overall.

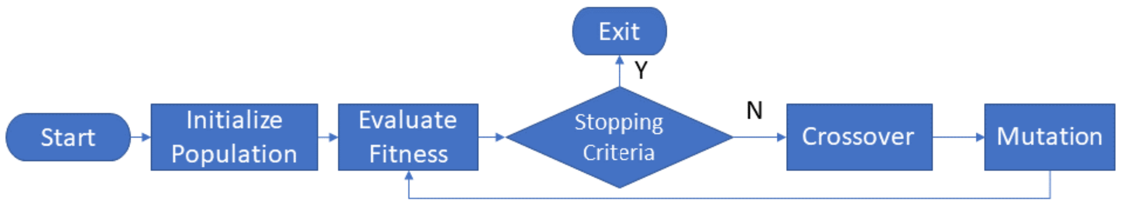


Fig. 2: Flowchart of Genetic Algorithms.

2.4. Multi-Agent Genetic Algorithms

Multi-agent genetic algorithm (MAGA) was proposed by Zhong et al. in “A Multi-Agent Genetic Algorithm for Global Numerical Optimization” and combines genetic algorithms with agent-based systems. Compared to genetic algorithms where individuals in the population are just solutions, each individual is an agent, an autonomous computational individual with properties and actions that can interact with each other or the environment (Rand and Wilensky 1). The agents represent potential solutions with an assigned fitness based on that solution. Using a Von Neumann neighborhood, agents interact with their neighbors living in a lattice environment that wraps around (see fig. 3).

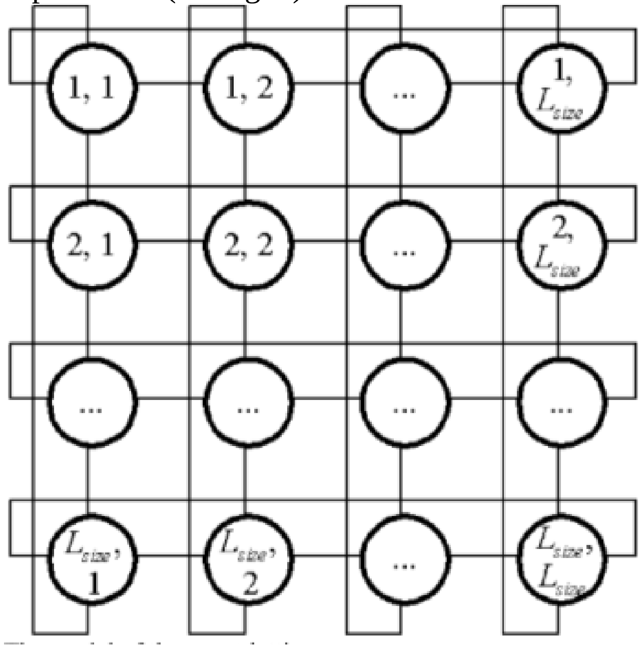


Fig 3: Agent lattice environment (Zhong et al.).

MAGA follows a similar structure to genetic algorithms (see fig. 4). The initialization phase creates the grid of agents with their associated chromosome and fitness. Next each agent performs the neighbor interaction operators which happen in two phases. The first phase is the neighborhood competition operator. Each agent finds its neighbor with the highest fitness. If the neighbor has a higher fitness than the agent, the agent dies and is replaced with either crossover or inversion. If the neighbor had a lower fitness than the agent, nothing happens. The second phase is the neighborhood orthogonal crossover operator. In this phase, every agent has a chance to be replaced by crossover to simulate cooperation. After the interaction operators, every agent has the chance to go through mutation just like a normal genetic algorithm. Finally, in the self-learning phase, the best agents in the system are selected to undergo its own smaller MAGA to find a more optimal agent to replace it. This simulates an agent knowing the problem and trying to improve itself. This process, excluding the initialization phase, repeats until a stopping criterion is met. According to the article, MAGA overcomes the early and local convergence problems associated with genetic algorithms, there is good scalability in terms of computational cost, and the co-evolution in an environment gives a better reflection of evolution in Nature (Zhong et al.).

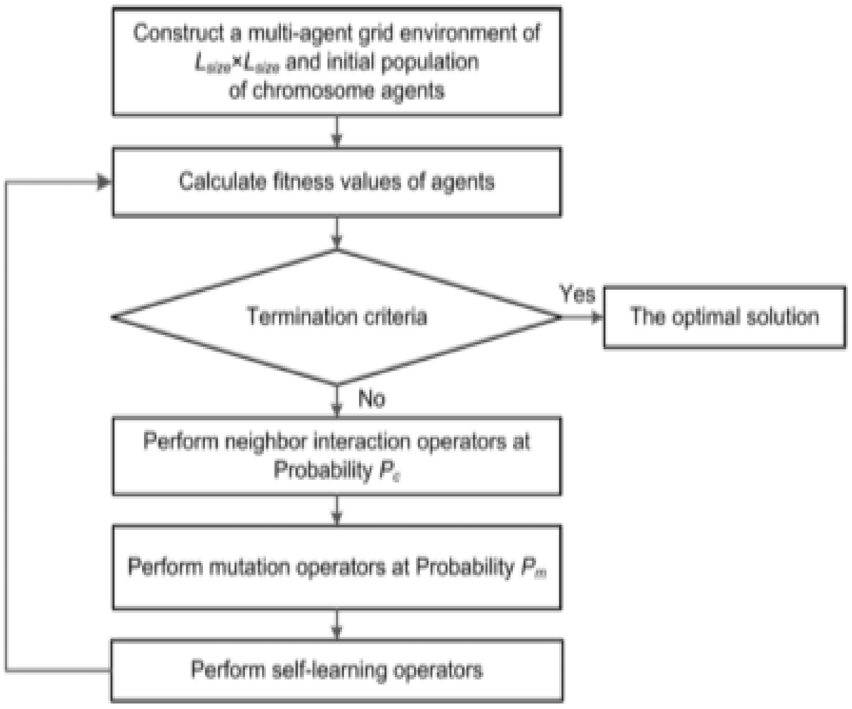

Construct a multi-agent grid environment of $L_{size} \times L_{size}$ and initial population of chromosome agents
Calculate fitness values of agents
Termination criteria
Yes
The optimal solution
No
Perform neighbor interaction operators at Probability P_c
Perform mutation operators at Probability P_m
Perform self-learning operators

Fig 4: Flowchart of Multi-Agent Genetic Algorithm (Zhong et al.).

2.5. Related Work

There does not seem to be any published research on any MAGA implementations yet for the vehicle routing problem, but there has been many for normal genetic algorithm approaches. In “A genetic algorithm for the vehicle routing problem” by Ayechew and Baker, the setup for the algorithm assumes a fixed number of vehicles. The fixed number of vehicles is an upper-bound, so solutions can be found with less vehicles. The chromosome is created such that each gene is an integer representing the vehicle number it is assigned to with the index of the gene representing the stop. Once the stops have been distributed to their vehicles, they apply a second-layer algorithm to optimally sort them (Ayechew and Baker). Another chromosome representation involves a two-dimensional array where a list of trucks has its own list of stops (Faruque et al.). Compared to the one-dimensional chromosome, this chromosome has slightly more complex calculations and algorithms involved. These approaches are like many others found when assuming a fixed number of vehicles, but this may not be the most optimal solution overall. It can cost a company up to $125,000 for a new delivery truck not including the trailer, and when other factors such as insurance and additional employee costs are added in having an unnecessary route can end up very costly. On the other hand, it may be beneficial to divide stops up to fit work hours, reduce employee fatigue, etc. Therefore, finding the optimal solution to a vehicle routing problem also involves finding the optimal number of vehicles without constrains which this project attempts to solve.

3. Model Summary

To achieve the goal to find the optimal number of vehicles, a new chromosome representation is built. For this implementation of a multi-agent genetic algorithm, a similar framework is built creating a simpler approach to the neighborhood interaction operator and excludes the self-operator (see fig. 5). The crossover operator, CX2, is chosen and modified for creating new offspring. An atypical mutation method is programmed for improvements over a traditional mutation algorithm. NetLogo was the selected environment to program and model the simulations.

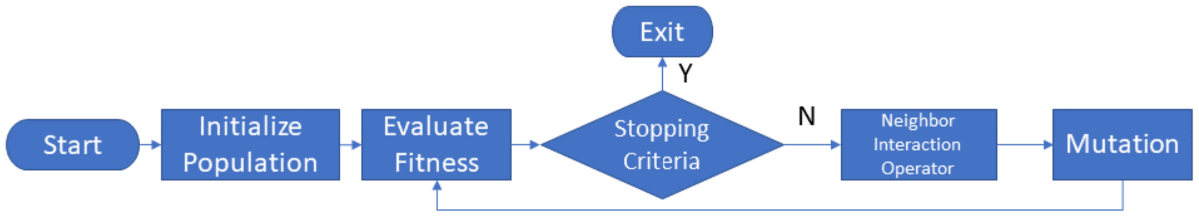


Fig 5: Flowchart of Project MAGA Implementation

3.1. Chromosome

The chromosome is a string of integers representing a stop to visit. If the integer is 0, that represents a division of the route amongst another vehicle. This can also be thought of as a single traveler or vehicle being able to return to the start multiple times making resemble more of a traveling salesman problem. This simplification makes the following algorithms and calculations easier.

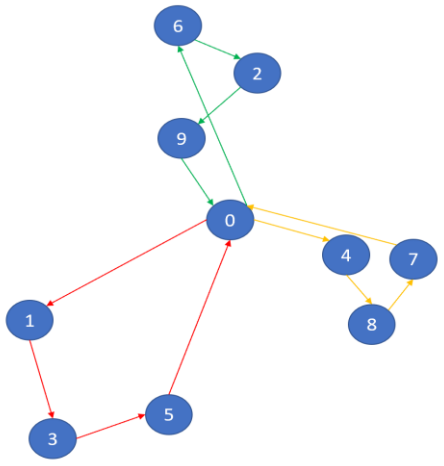


Fig 6: Graphical representation of the chromosome [1 3 5 0 6 2 9 0 4 8 7]

3.2. Fitness

Fitness represents the total cost of visiting all stops starting from and ending at the central depot. Lower cost means higher fitness. Because of the form of the chromosome, the fitness can easily be calculated by summing the costs between each pair of integers in the chromosome with an additional 0 at the beginning and the end.

$$Chromosome = [1\ 3\ 2\ 4]$$
$$Fitness = cost(0, 1) + cost(1, 3) + cost(3, 2) + cost(2, 4) + cost(4, 0)$$

3.3. Initialization

In the initialization phase, a square grid of L x L agents is created where L is an integer. Each agent is assigned a chromosome of the randomized list of stops with up to n-1, where n is the number of stops, number of 0's randomly inserted at points such that none are adjacent to each other and none are at the first or last index of the chromosome. The reason for the constrain on 0 placement is because having 0's next to each other would represent an additional vehicle that goes nowhere and would throw off calculations. Having 0's at the start or end also throw off calculations since there is inherently a 0 appended at those spots.

3.4. Neighborhood Interaction

For simplicity, instead of splitting up the neighborhood interaction into two phases, there is only a single crossover to replace agents that are killed by higher fitness neighbors using the same criteria from section 2.4. There are two methods of crossover that happens at probabilities P_c and 1-P_c.

3.5. Crossover

The crossover method implemented is CX2 proposed by Hussain et al. in "Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator." When compared, this method was shown to be better than other best known crossover methods for the traveling salesman problem (Hussain et al.). To fit with the chromosome, first the 0's are removed from the parents remembering the indexes of the higher fitness parent. CX2 creates two offspring which then gets the 0's reinserted at the indexes of the higher fitness parent. Because the crossover is meant to replace a single dead agent, the fitness' of the offspring are calculated with the higher fitness one replacing the agent.

In "Research on multi-agent genetic algorithm based on tabu search for the job shop scheduling problem" by Peng et al., they implemented a crossover method that had different results depending on the order of parents in the algorithm for their MAGA implementation. Since CX2 seemed like a similar crossover and the order of parents was not discussed in Hussain's work, the project implementation has two options for crossover. Method 1 uses the agent to be replaced as Parent 1 in the algorithm and method 2 uses the neighbor as Parent 1 with probabilities P_c and 1-P_c, respectively.

3.6. Mutation

For mutation, a random amount of 0's is inserted into the chromosome at random points maintaining the position constraints that were in the initialization phase.

3.7. NetLogo

Because the project involved agents, NetLogo was selected as the programming environment due to its ease of applying object-oriented agents to a graphical interface. In addition, NetLogo has great visualizations and tools helping experiments throughout the project. The following subsections are in the format of the ODD protocol for describing agent-based models with an additional section describing the process of building the model.

3.7.1. Purpose: The model was designed to explore parameters of a multi-agent genetic algorithm for the vehicle routing problem. Can the best solution with the best number of vehicles in the solution space emerge? How do the crossover methods and mutation rate affect the ability to find new solutions and the strength of those solutions? Is the multi-agent approach beneficial to genetic algorithms?

3.7.2. Entities, state variables, and scales

Entities: Only patches are utilized.

Patch Variables:

- *chromosome* – an ordered list of stops and divisions to multiple vehicles
- *fitness* – the fitness of the chromosome
- *num-vehicles-represented* – the number of 0's in the chromosome +1
- *pcolor* – the color displayed on the view based on its fitness scaled to the rest of the population

Global variables:

- *best-fitness* – the highest fitness found in the population
- *worst-fitness* – the lowest fitness found in the population
- *best-chromosome* – the chromosomal representation of the highest fitness found in the population
- *best-generation* – the generation that the *best-fitness* was found at
- *locations* – a symmetric adjacency matrix storing the cost to travel between locations including the origin. Each value in the matrix is randomly assigned from 1 to 100 and is of size *num-locations+1 x num-locations+1*.
- *changes-in-best* – the number of times *best-fitness* was updated

Scales:

- *n-generations* – the number of generations to run the simulation for
- *sqrt-population* – the square root of the number of patch agents to create
- *num-locations* – the number of locations needed to visit in the problem
- *probability-best-parent-first-cross* – the probability method 1 is used for crossover
- *probability-mutation* – the probability of an agent undergoing mutation

3.7.3. Process Overview and Scheduling: On each time step, everyone in the population first goes through the neighborhood interaction operator. Once all offspring are generated, some go through mutation. After the mutation phase, all global variables except *locations* are updated reflecting the new population. Finally, all patches on the grid are recolored to reflect new fitness'. The process repeats for *n-generations*.

3.7.4. Design Concepts: To view emergence of higher fitness solutions, variables tracked include the best fitness and the corresponding chromosome, the number of times a new higher fitness was found, and the generation that the best fitness was found. The chromosome was designed to allow a dynamic number of vehicles to be represented. Because of the constrains on 0-placement, it allows there to be a realistic upper-bound that can easily scale as *num-locations* increases. In the extreme case, there will be vehicles going to a single stop each. The locations are represented as a symmetric adjacency matrix for ease of calculation through quick lookup. Initially each location had coordinates, but the need to continuously apply a distance formula seemed tedious and computationally costly. Also assigning a cost for weights between locations allow cost to conceptually include not only distance but other factors like time and traffic into a single value. To simplify implementation, the neighborhood interaction operator was reduced retaining the agent-neighborhood comparison, and the self-learning operator was not included. For the sake of exploring new methods of crossover, CX2 was chosen and modified to fit the chromosome. Traditional mutation operators involve methods such as moving locations of bits or inverting sections of the chromosome. Because there is a focus on finding the optimal number of vehicles, the insertion of 0's was chosen instead. Since mutation inserts 0's, the initial population of chromosomes have a right-skewed distribution of 0's where there are more single vehicle chromosomes. The coloration of patches by fitness help the user observe the distribution of fitness across the model, and the plot of the best fitness over generations help observations on how the system as a whole evolves.

3.7.5. Initialization: The *locations* matrix is created assigning values of weights between 1 and 100. A *sqrt-population x sqrt-population* square grid of patches are created with a randomized chromosome upon initialization and the associated fitness. The global variables are then updated, and patches are recolored.

3.7.6. Submodels:

- *calculate-fitness* – this function takes a chromosome and returns the fitness of the chromosome by summing the costs found in *locations* between each pair of integers in the chromosome with an additional 0 at the beginning and the end.
- *compete* – the agent that calls this function finds the neighbor with the highest fitness. If *fitness* of the agent is less than *fitness* of the neighbor, 0's are removed from the two agent's chromosomes and passed to *crossover*. If a random float generated is less than *probability-best-parent-first-cross*, the neighbor's chromosome is passed as Parent 1. Otherwise, the agent's chromosome is passed as Parent 1. When *crossover* produces two offspring chromosomes, 0's are reinserted at the indexes it was in the neighbor's original *chromosome*. The agent's *chromosome* and *fitness* are reassigned to the offspring with the higher fitness' chromosome and fitness values.
- *crossover* – the algorithm occurs in the following steps given Parent 1 (P2) and Parent 2 (P2):
    1. The 1st gene from P2 is the 1st gene of Offspring 1 (O1)
    2. Find the gene from Step 1 in P1, pick the exact same position gene in the P2, find it in P1, the same position gene in P2 will be the 1st gene of Offspring 2 (O2).
    3. Find the gene from Step 2 in P1, and the same position gene in P2 will be the next gene for O2
    4. Repeat Steps 2 and 3 until the 1st gene of P1 will not come in O2

5. If there are remaining genes, repeat the process starting from Step 1 with the unused gene instead of the 1st gene from P2.

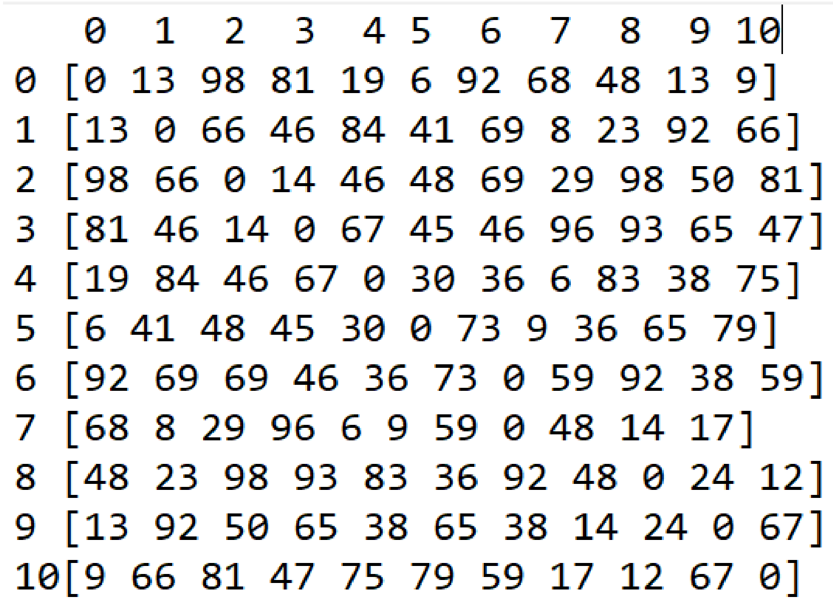
- *mutate* – insert a random amount of 0's into the agent's *chromosome* retaining the position constraints from section 3.3. After insertion, reassign the agent's *fitness* by calling *calculate-fitness*.

3.7.7. Initial Approach to Model

Initially the chromosome was to be of a form like in Ayechew and Baker's implementation described in section 2.5, however, to make calculations as simple as possible the current form was chosen more resembling a simple Traveling Salesman Problem. The locations were initially going to have their own coordinates in two-dimensional space, but again to simplify calculations and generalize 'cost' an adjacency matrix was selected. The model was supposed to be built in parts over a duration of three weeks with research being one week, but due to the complexity of research done, the model was programmed in a couple days with two weeks of research. Due to the high-level description of the neighborhood interaction operator in MAGA, a simpler implementation was done, and the self-learning operator was excluded completely to focus more on other aspects of MAGA. Throughout implementation, the test-driven development process of Agile development was applied where test cases are built before functions to validate functionality.

4. Experiments

4.1. Environment: the following experiments was performed utilizing BehaviorSpace in NetLogo using a set adjacency matrix for *num-locations = 10* (see fig. 7). For this matrix, the highest fitness that can be achieved is *-227*. There are 4 possible chromosomes to attain that fitness: [5, 0, 10, 8, 1, 7, 4, 2, 3, 6, 9], [9, 6, 3, 2, 4, 7, 1, 8, 10, 0, 5], [5, 0, 9, 6, 3, 2, 4, 7, 1, 8, 10], and [10, 8, 1, 7, 4, 2, 3, 6, 9, 0, 5]. All runs for experiments are for 500 generations.



	0	1	2	3	4	5	6	7	8	9	10
0	[0	13	98	81	19	6	92	68	48	13	9]
1	[13	0	66	46	84	41	69	8	23	92	66]
2	[98	66	0	14	46	48	69	29	98	50	81]
3	[81	46	14	0	67	45	46	96	93	65	47]
4	[19	84	46	67	0	30	36	6	83	38	75]
5	[6	41	48	45	30	0	73	9	36	65	79]
6	[92	69	69	46	36	73	0	59	92	38	59]
7	[68	8	29	96	6	9	59	0	48	14	17]
8	[48	23	98	93	83	36	92	48	0	24	12]
9	[13	92	50	65	38	65	38	14	24	0	67]
10	[9	66	81	47	75	79	59	17	12	67	0]

Fig 7: Adjacency Matrix for 10 locations used for experiments

4.2. Experiment 1: ability to find the best solution: This experiment takes the value of *best-fitness* of each run over 100 and 1000 runs with parameters *probability-best-parent-first-cross*=0.5 and *probability-mutation=0.1*. The same experiment is also run on a modified version of this project that represents traditional genetic algorithms to compare the two different methods. That model removes the agent grid and uses tournament selection of size 15 instead of the neighbor interaction operator with the same crossover and mutation operators.

4.3. Experiment 2: varying *probability-best-parent-first-cross*: This experiment varies *probability-best-parent-first-cross* from 0 to 1 with intervals of 0.1. For each interval, 20 runs are simulated with *probability-mutation=0.1* taking the value of *best-fitness* and *changes-in-best* to see the effects of the two methods on those variables.

4.4. Experiment 3: varying *probability-mutation*: This experiment varies *probability-mutation* from 0 to 0.5 with intervals of 0.05. For each interval, 20 runs are simulated with *probability-best-parent-first-cross=0.5* taking the value of *best-fitness*, *best-generation* and *changes-in-best* to see the effect mutation has on introducing variation into the system. In addition, a more common mutation method (swapping) was coded into the model and ran through the same experiment in order to compare the effectiveness of the proposed mutation algorithm.

5. Results

5.1. Experiment 1 Results

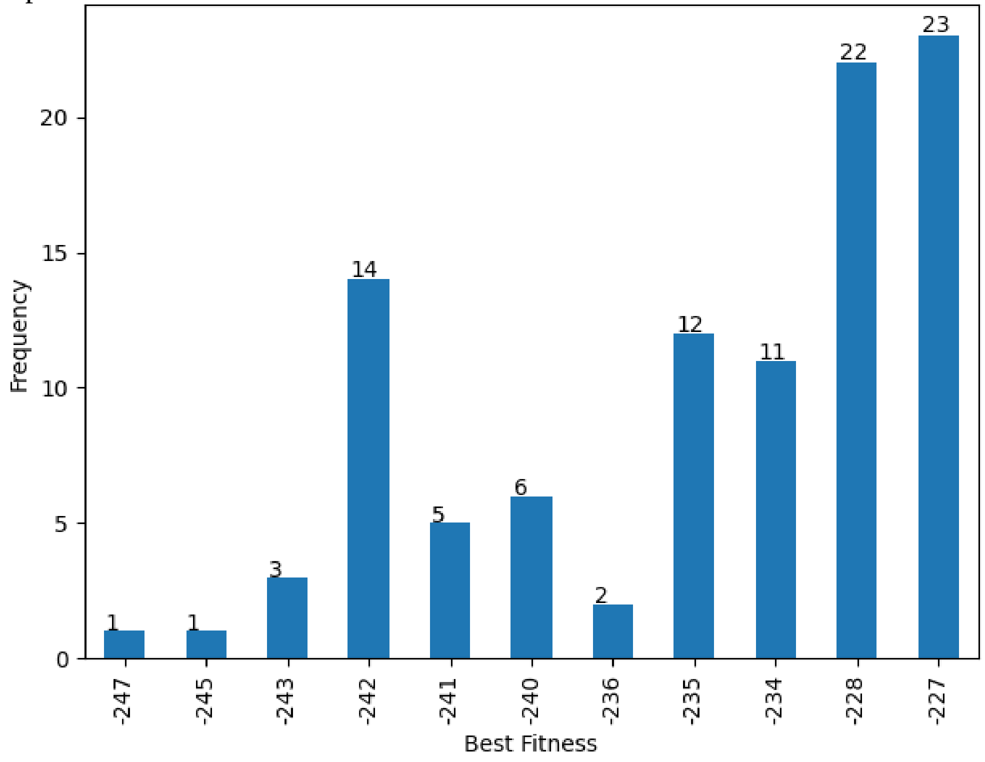


Fig 8: Frequency of *best-fitness* over 100 runs using MAGA

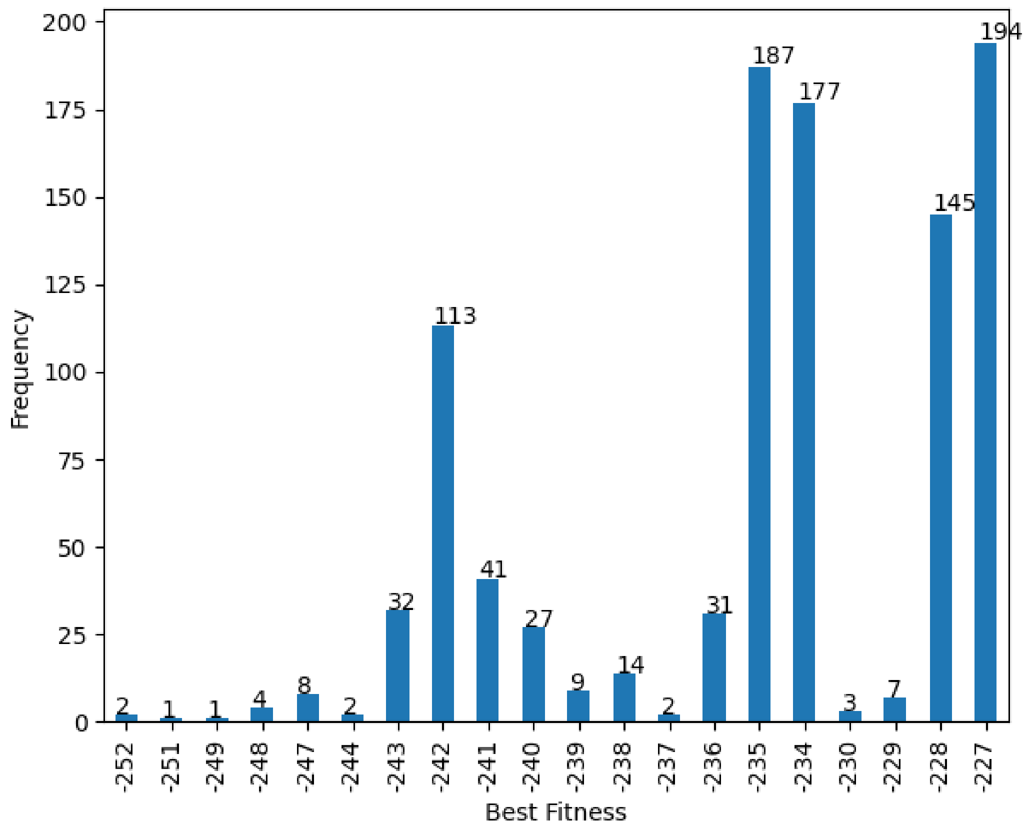


Fig 9: Frequency of *best-fitness* over 1000 runs using MAGA

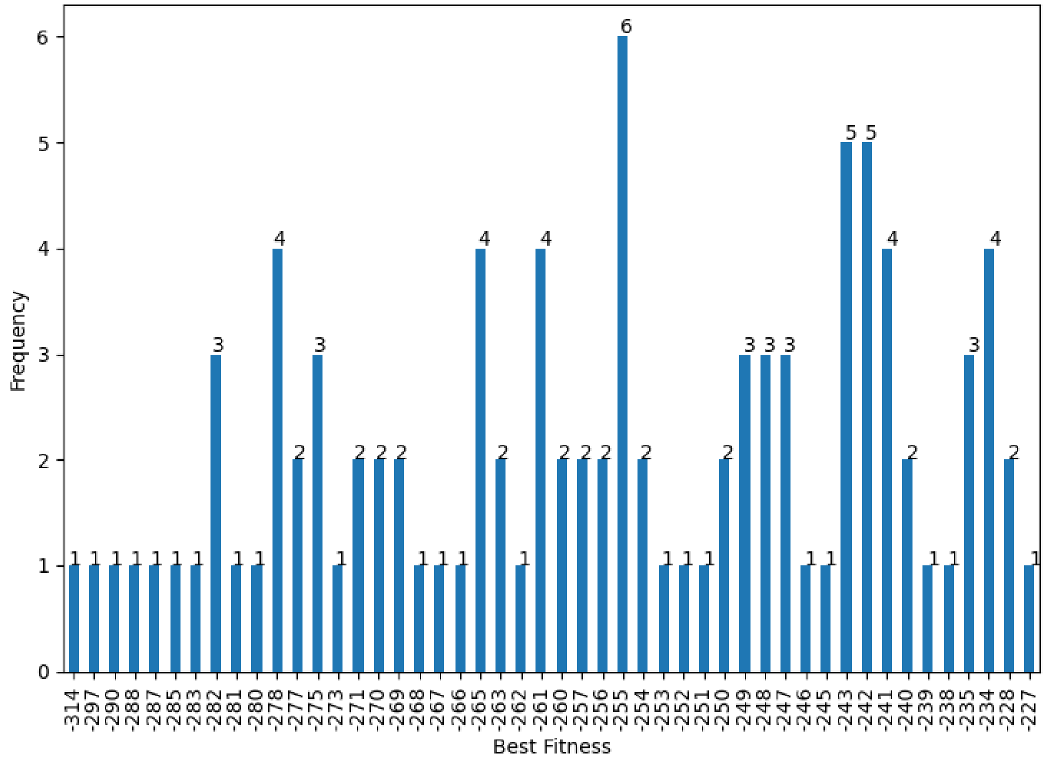


Fig 10: Frequency of *best-fitness* over 100 runs using tournament selection GA

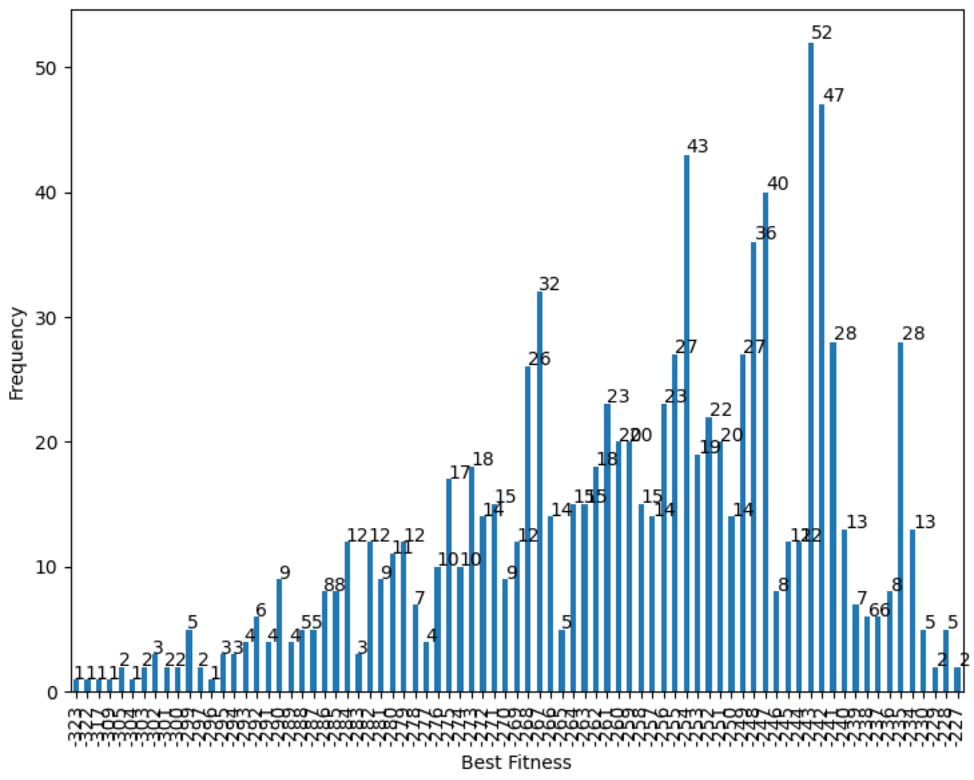


Fig 11: Frequency of *best-fitness* over 1000 runs using tournament selection GA

5.2. Experiment 2 Results

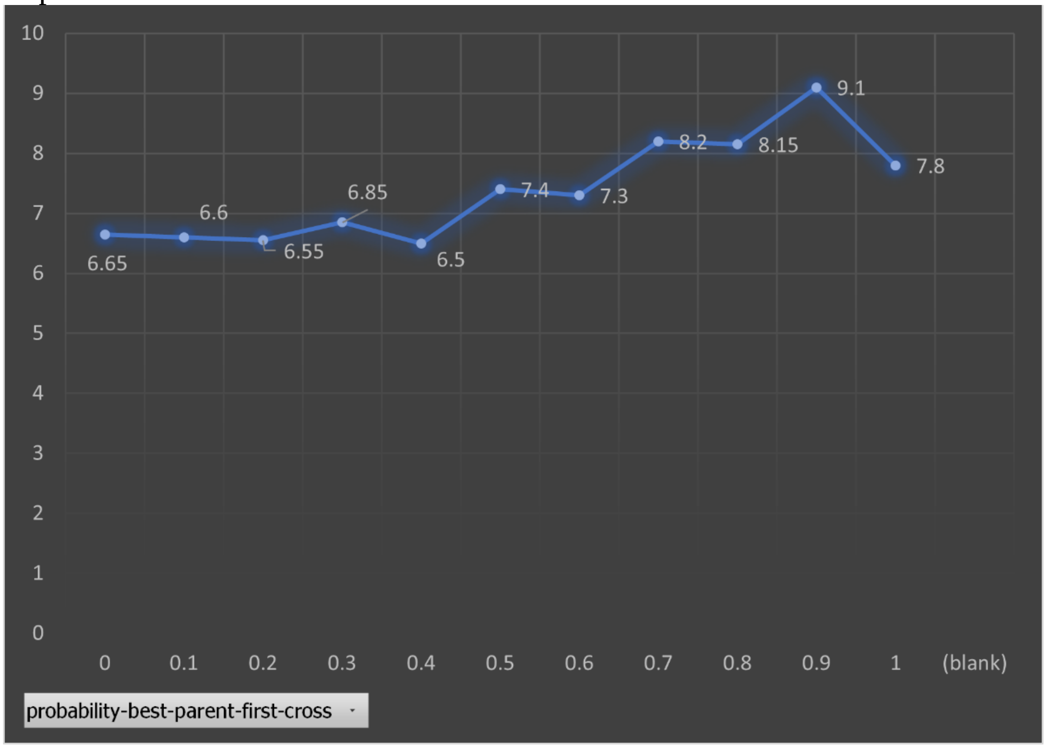


Fig 12: Average of 20 runs of *changes-in-best* varying *probability-best-parent-first-cross*

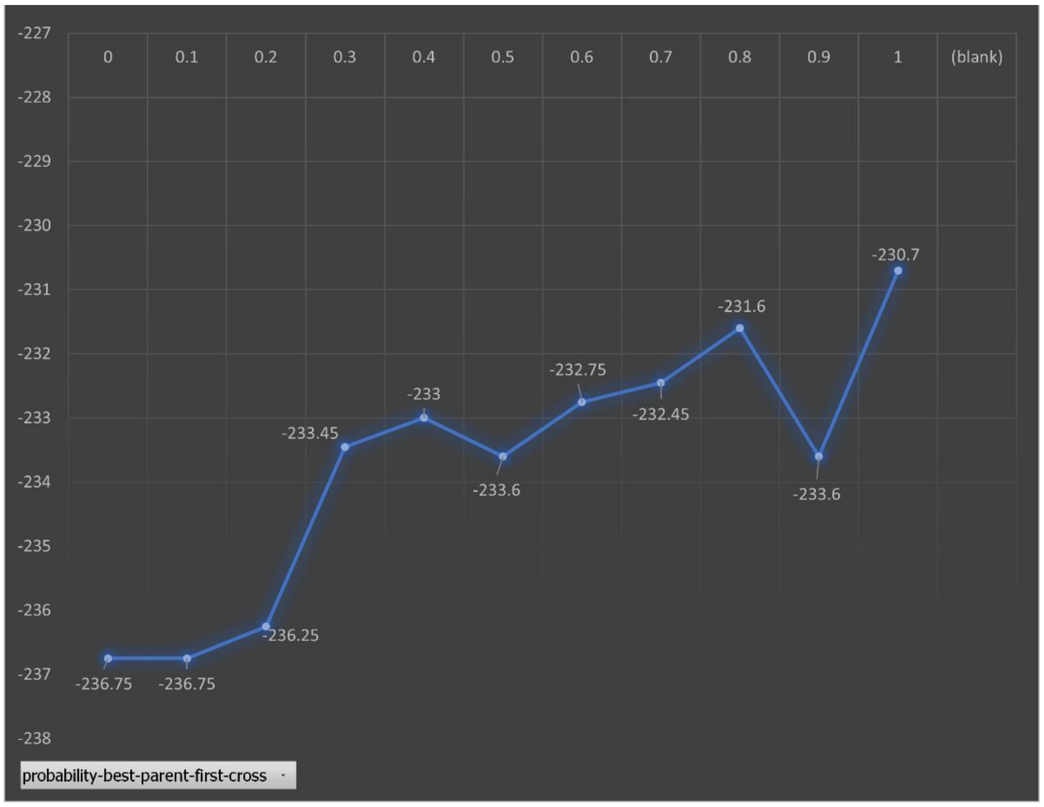


Fig 13: Average of 20 runs of *best-fitness* varying *probability-best-parent-first-cross*

5.3. Experiment 3 Results

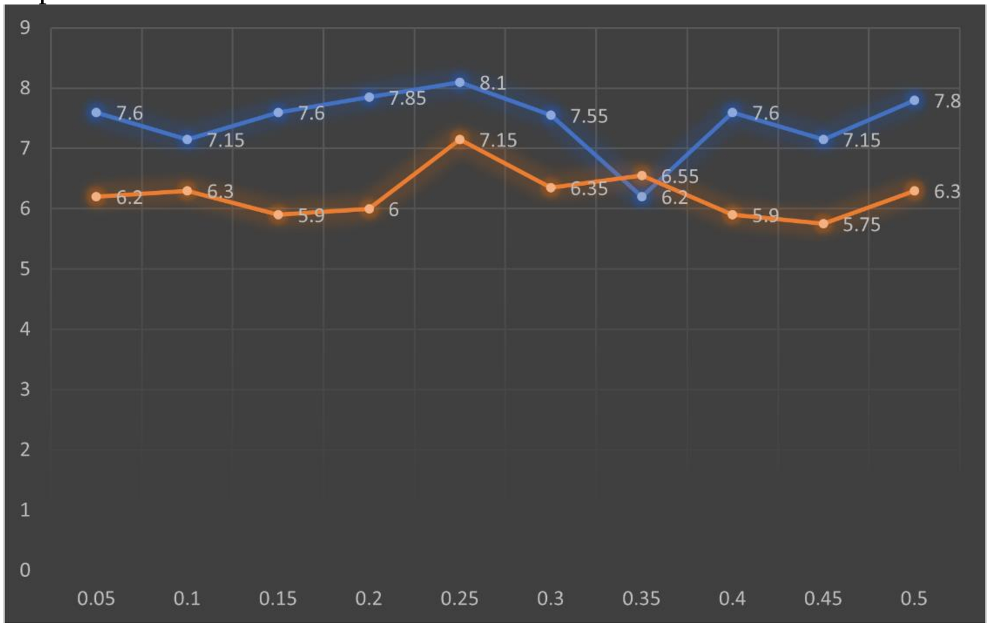


Fig 14: Average of 20 runs of *changes-in-best* varying *probability-mutation* (orange=swapping mutation, blue=inserting 0's)

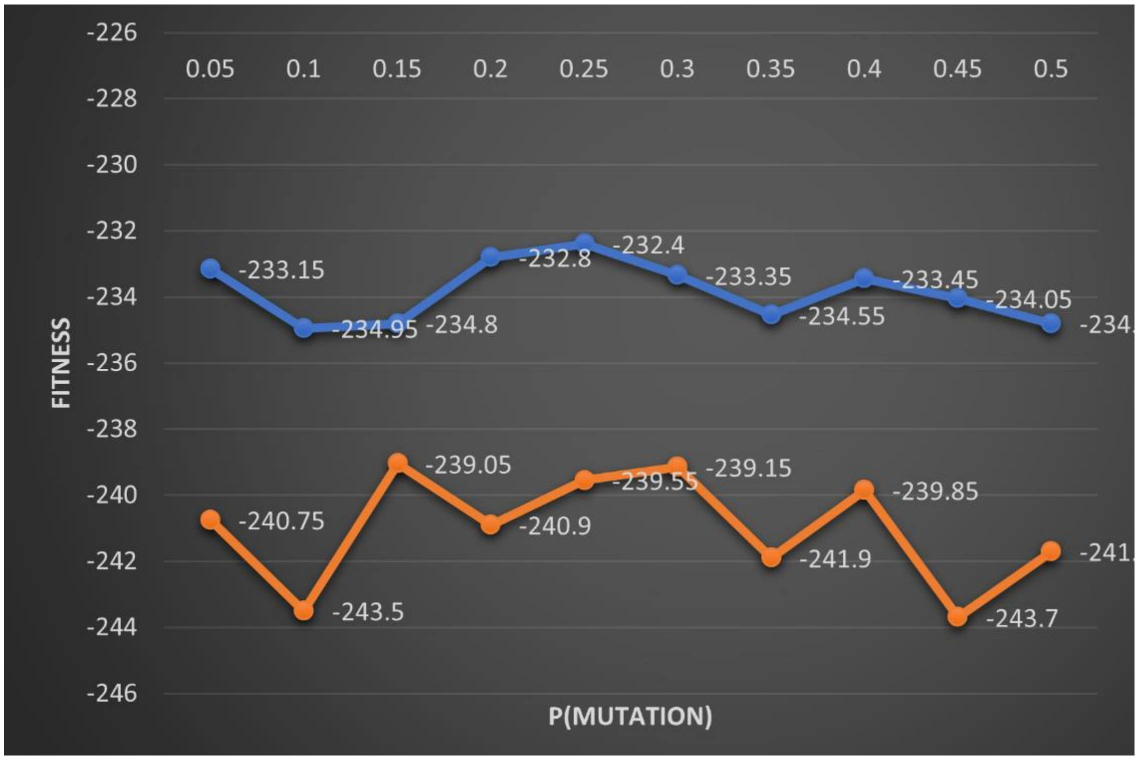


Fig 15: Average of 20 runs of *best-fitness* varying *probability-mutation* (orange=swapping mutation, blue=inserting 0's)

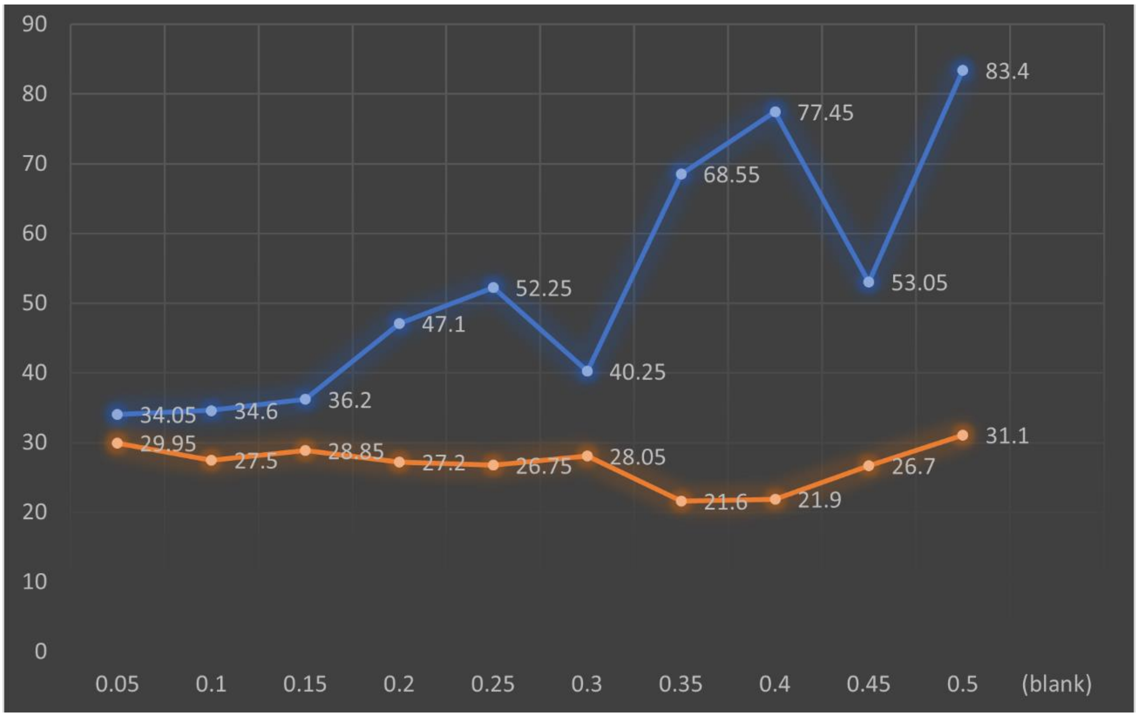


Fig 16: Average of 20 runs of *best-generation* varying *probability-mutation* (orange=swapping mutation, blue=inserting 0's)

6. Discussion
    6.1. Experiment 1: Looking at the MAGA data, in both 100 and 1000 runs the program was able to converge on the best fitness solution the most out of all other solutions. The results of the tournament selection genetic algorithm implementation show a much larger spread of fitness' that were converged on. The most frequent fitness found in that implementation was not the best fitness. This supports the research of MAGA avoiding early and local convergence stated earlier. The ability to find the best fitness solution also supports the goal of finding a dynamic number of vehicles. The time to run BehaviorSpace for 1000 runs on the MAGA and GA were less than 20 minutes and 1 hour 20 minutes respectively with the same hardware, although it may be due to overhead caused by the programmer rather than the methods itself.
    6.2. Experiment 2: By collecting data varying the use of the two approaches to the CX2 crossover, it can be seen that using the higher fitness parent as Parent 1 in the algorithm produces both higher fitness results and slightly more variation into the system. Thus, to maximize results from this MAGA implementation, the neighbor should always be used as Parent 1 in the crossover.
    6.3. Experiment 3: When comparing the two mutation methods and varying the probability of mutation, not much can be said for its affects in introducing new solutions into the system shown by Fig. 14. On average, using the proposed mutation operator does result in higher fitness individuals, however the probability of mutation does not seem to have any effect. The last test had very interesting results. When looking at swapping mutation, the average generation that the best fitness was found in did not change as the probability increase and was significantly lower than the proposed mutation. It can be suggested then that the proposed mutation helps the system avoid early convergence. It is also interesting to see that as the probability of mutation increases for the proposed mutation, on average the best fitness is found in later generations even though the number of times the best fitness changes does not.

7. Summary

In comparison to a traditional genetic algorithm, the modified MAGA implementation for the vehicle routing problem performed better using agent neighborhood interactions rather than selection. The chromosome representation was successful in producing strong solutions with the ideal number of vehicles emerging. The modified CX2 algorithm produced higher fitness when the higher fitness parent is used as Parent 1 in the algorithm. The proposed mutation algorithm was successful in preventing early convergence of the system compared to swapping mutation.

Using NetLogo was very beneficial in developing this project. The innate object-oriented patches made the agent lattice very simple to implement along with patch-patch interactions. The user-interface did not seem very critical in extrapolating results, but in certain runs there was an unexpected observation of fitness improvements dispersing like diffusion models. BehaviorSpace was very useful to run the experiments and collect data that was exported to Excel and Python for analysis.

Future considerations include collecting and analyzing data with a larger number of stops for testing a larger solution space, adding the full neighborhood interaction operator and self-learning operator from the MAGA framework, and comparing run-time to an exhaustive search.

8. References

Alhanjouri, Mohammed and Belal Alfarra. “Ant Colony versus Genetic Algorithm based on Travelling Salesman Problem”. *International Journal of Computer Technology and Applications. 2*, June 2011, 570-578.

Ayechew, M.A., and Barrie Baker. “A genetic algorithm for the vehicle routing problem.” *Computers & Operations Research, Volume 30, Issue 5,* April 2003, 787-800.

Faruque, Faisal, et al. “Solving the Vehicle Routing Problem using Genetic Algorithm.” *(IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 2, No. 7*, 2011, 126-131.

Holland, John. “Genetic Algorithms.” *Scientific American*, July 2019, 66-72.

Hussain, Abid, et al. "Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator", *Computational Intelligence and Neuroscience, vol. 2017, Article ID 7430125*, 7 pages, 2017. https://doi.org/10.1155/2017/7430125

Kovács, László, et al. “Fitness Landscape Analysis and Edge Weighting-Based Optimization of Vehicle Routing Problems.” MDPI, Multidisciplinary Digital Publishing Institute, 28 Oct. 2020, www.mdpi.com/2227-9717/8/11/1363.

Peng, Chong, et al. (2019). Research on multi-agent genetic algorithm based on tabu search for the job shop scheduling problem. PLOS ONE. 14. e0223182. 10.1371/journal.pone.0223182.

Rand, William, and Uri Wilensky. *An Introduction to Agent-Based Modeling.* The MIT Press, 2015.

Zhong, Weicai et al. "A multiagent genetic algorithm for global numerical optimization." *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), vol. 34, no. 2,* April 2004, 1128-1141.

9. Appendices

9.1. Appendix A: Project Code

```
extensions[ rnd ]

patches-own [
  chromosome
  fitness
  num-vehicles-represented
]

globals [
  best-fitness
  worst-fitness
  best-chromosome
  best-generation
  locations
  changes-in-best
]

to setup
  clear-all
  reset-ticks
  ;;resizes the world into grid of sqrt-population x sqrt-population patches
  ;;ex: sqrt-population = 10 -> 10x10 grid of 100 patches
  resize-world (sqrt-population * -0.5) (sqrt-population - 1) / 2 (sqrt-population * -0.5) (sqrt-population - 1)
/ 2
  set locations create-locations
  set changes-in-best 0
  initialize-patches
end

to setup-fixed
  clear-all
  reset-ticks
  resize-world (sqrt-population * -0.5) (sqrt-population - 1) / 2 (sqrt-population * -0.5) (sqrt-population - 1)
/ 2
  set locations [[0 13 98 81 19 6 92 68 48 13 9] [13 0 66 46 84 41 69 8 23 92 66] [98 66 0 14 46 48 69 29
98 50 81] [81 46 14 0 67 45 46 96 93 65 47] [19 84 46 67 0 30 36 6 83 38 75] [6 41 48 45 30 0 73 9 36 65
79] [92 69 69 46 36 73 0 59 92 38 59] [68 8 29 96 6 9 59 0 48 14 17] [48 23 98 93 83 36 92 48 0 24 12]
[13 92 50 65 38 65 38 14 24 0 67] [9 66 81 47 75 79 59 17 12 67 0]]
  set num-locations 10
  initialize-patches
end

to go
  tick
  ask patches [compete]
  ask patches [if random-float 1 < probability-mutation [ifelse use-traditional-mutation = false
[mutate][traditional-mutation]]]
  update-globals
  ask patches [color-patches]
  if (ticks = n-generations) [stop]
end

to update-globals
```

```
  let temp [fitness] of max-one-of patches [fitness]
  if temp > best-fitness [
    set best-fitness temp
    set best-generation ticks
    set best-chromosome [chromosome] of max-one-of patches [fitness]
    set changes-in-best changes-in-best + 1
  ]
  set worst-fitness [fitness] of min-one-of patches [fitness]
end

;;This function is used in setup and creates the locations needed to travel to
to initialize-patches
  ask patches [
    set chromosome create-chromosome
    set fitness calculate-fitness chromosome
    set num-vehicles-represented count-num-vehicles-represented
  ]
  ;;update-globals
  set best-fitness [fitness] of max-one-of patches [fitness]
  set best-generation ticks
  set best-chromosome [chromosome] of max-one-of patches [fitness]
  set worst-fitness [fitness] of min-one-of patches [fitness]
  ;;color patches based on fitness
  ask patches [color-patches]
end

to color-patches
  ifelse worst-fitness != best-fitness [
    set pcolor scale-color green fitness worst-fitness best-fitness
    ifelse fitness = best-fitness [set plabel-color black set plabel "best"]
    [ifelse fitness = worst-fitness [set plabel-color white set plabel "worst"][set plabel ""]]
  ][ set pcolor blue set plabel "done"]
end

;;creates a list of location coordinates with the index corresponding to the chromosomal representation
to-report create-locations
  let matrix []
  let i 0

  while [i <= num-locations][
    let arr []
    let j 0
    while [j <= num-locations][
      let val 0
      ifelse i = j [ set val 0 ]
      [ ifelse i > j [set val (item i (item j matrix))]
        [ set val ((random 99) + 1) ] ] ;;values 1 to 100
      set arr lput val arr
      set j j + 1
    ]
    set matrix lput arr matrix
    set i i + 1
  ]
  report matrix
end
```

```
;;This function is called in setup and creates a chromosome for each strategy
to-report create-chromosome
  let temp shuffle (range 1 ( num-locations + 1 )) ;;randomly shuffle list of locations
  ;;The rest of the code is to insert the return trips into the chromosome
  let returns-to-home find-return-trips
  repeat returns-to-home [ set temp insert-zero temp ]
  report temp
end

to-report insert-zero [ genome ]
  let g genome
  let insert-success false
    let index-range (range 1 (length g))
    let random-index one-of index-range
    while [not insert-success] [
      ;;if no neighboring 0's insert, else look for new space
      ifelse item random-index g != 0 and item ( random-index - 1 ) g != 0 [
        set g insert-item random-index g 0
        set insert-success true
      ]
      [
        ;;if pointed at the end of the list, start from the beginning
        ;;else increment pointer 1 over
        ifelse random-index = ( length g - 1) [ set random-index 1] [ set random-index random-index + 1 ]
      ]
    ]
  report g
end

;;This function is used in create-chromosome
;;it randomly calculates return trips to home simulating multiple-vehicle routes
;;the number of extra vehicles is right-skewed so single vehicle routes are more frequent
to-report find-return-trips
  let returns n-values num-locations [i -> i] ;;ex 3 locations [0, 1, 2]
  let weights reverse n-values num-locations [ i -> i + 1 ] ;; [3, 2, 1]
  ;; https://ccl.northwestern.edu/netlogo/docs/rnd.html#rnd:weighted-one-of-list
  let pairs (map list returns weights)
  report first rnd:weighted-one-of-list pairs [ [p] -> last p ] ;; random number of return trips right-skewed
so 0 appears more often than 1
end

to-report count-num-vehicles-represented
  let num 1
  foreach chromosome [ x -> if x = 0 [set num num + 1]]
  report num
end

to-report zero-indexes [ c ]
  let temp []
  let i 1
  while [i < length c] [
    if 0 = (item i c) [ set temp lput i temp ] ;;if finds 0 add index to end of temp
   set i i + 1
  ]
  report temp
end
```

```
to-report add-back-zeros [ c zero-locations ]
  let temp c
  foreach zero-locations [ x ->
    set temp insert-item x temp 0
  ]
  report temp
end

;;This function is run by each patch
;;Visits every location in the chromosome then returns home
;;and records total distance traveled as 'fitness'
to-report calculate-fitness [ c ]
  let total 0
  let prev-loc 0
   ;;for each gene in chromosome
  foreach c [ gene ->
      set total total - (item prev-loc (item gene locations))
      set prev-loc gene
    ]
  ;;return home
  set total total - (item prev-loc (item 0 locations))

  report total
end

to-report crossover [g1 g2]
  let o1 []
  let o2 []
  let bit2 0
  let pos1 0
  let pos2 0

  let continue true
  let start-index 0
  while [continue] [
    ;;Step 2 in algorithm
    set bit2 item start-index g2
    set o1 lput bit2 o1 ;;1st bit from parent2 is 1st bit of offspring1

    let continue2 true
    while [continue2] [
      ;;Step 3
      set pos1  position bit2 g1 ;;find position of bit2 in parent1
      set bit2 item pos1 g2 ;;find bit in parent2 at the same position of parent1
      set pos1 position bit2 g1 ;;find position of bit2 in parent1 again
      set bit2 item pos1 g2 ;;find bit in parent2 at same position of parent1
      set o2 lput bit2 o2

      ;;Step 4 in algorithm
      set pos1 position bit2 g1
      ifelse pos1 != start-index [
        set bit2 item pos1 g2
        set o1 lput bit2 o1
      ]
      [set continue2 false]
    ]

    let ciac check-if-algorithm-complete o2 g1
    ifelse ciac = true [set continue false] [set start-index ciac]
  ]

  report list o1 o2
end

;;reports value not in offspring or true if all values are in the offspring
to-report check-if-algorithm-complete [offspring parent]
  foreach parent [x ->
    if position x offspring = false [report position x parent] ;;report value missing
  ]
  report true
end

to mutate
  let insertions (random (num-locations - 2)) + 1 ;;1 to num-locations-1 insertions
  let i 0
  while [ i < insertions ][
    if num-vehicles-represented < num-locations [
      set chromosome insert-zero chromosome
      set num-vehicles-represented count-num-vehicles-represented ;; or just +1
    ]
    set i i + 1
  ]
  set fitness calculate-fitness chromosome
end

to traditional-mutation
  let zeros zero-indexes chromosome
  let c-zero-removed remove 0 chromosome
  let bit-one random length c-zero-removed
  let bit-two random length c-zero-removed
  while [bit-one != bit-two][
   set bit-two random length c-zero-removed
  ]
  let temp item bit-one c-zero-removed
  set c-zero-removed replace-item bit-one c-zero-removed (item bit-two c-zero-removed)
  set c-zero-removed replace-item bit-two c-zero-removed temp
  set chromosome add-back-zeros c-zero-removed zeros
  set fitness calculate-fitness chromosome
end

to compete
  let max-neighbor-fitness [fitness] of max-one-of neighbors4 [fitness]
  if fitness < max-neighbor-fitness [
    let max-neighbor-chromosome [chromosome] of max-one-of neighbors4 [fitness]
    let crossover-results 0
    let zeros zero-indexes max-neighbor-chromosome
    let c-zero-removed remove 0 chromosome
    let c2-zero-removed remove 0 max-neighbor-chromosome
    ifelse random-float 1 < probability-best-parent-first-cross [
      set crossover-results crossover c2-zero-removed c-zero-removed
    ]
    [set crossover-results crossover c-zero-removed c2-zero-removed]
    let os1 add-back-zeros (item 0 crossover-results) zeros
    let os2 add-back-zeros (item 1 crossover-results) zeros
    let os1-fitness calculate-fitness os1
    let os2-fitness calculate-fitness os2
    ifelse os1-fitness > os2-fitness [
      set chromosome os1
      set fitness os1-fitness
    ][
      set chromosome os2
      set fitness os2-fitness
    ]
    set num-vehicles-represented count-num-vehicles-represented
  ]
end
```

9.2. Appendix B: Genetic Algorithm with Tournament Selection Code

```
extensions[ rnd ]

turtles-own [
  chromosome
  fitness
  num-vehicles-represented
]

globals [
  best-fitness
  worst-fitness
  best-chromosome
  best-generation
  locations
  changes-in-best
  tournament-size
]

to setup
  clear-all
  reset-ticks
  ;;resizes the world into grid of sqrt-population x sqrt-population patches
  ;;ex: sqrt-population = 10 -> 10x10 grid of 100 patches
  resize-world (sqrt-population * -0.5) (sqrt-population - 1) / 2 (sqrt-population * -0.5) (sqrt-population - 1)
/ 2
  set locations create-locations
  set changes-in-best 0
  set tournament-size 15
  initialize-turtles
end

to setup-fixed
  clear-all
  reset-ticks
  resize-world (sqrt-population * -0.5) (sqrt-population - 1) / 2 (sqrt-population * -0.5) (sqrt-population - 1)
/ 2
  set tournament-size 15
  set locations [[0 13 98 81 19 6 92 68 48 13 9] [13 0 66 46 84 41 69 8 23 92 66] [98 66 0 14 46 48 69 29
98 50 81] [81 46 14 0 67 45 46 96 93 65 47] [19 84 46 67 0 30 36 6 83 38 75] [6 41 48 45 30 0 73 9 36 65
79] [92 69 69 46 36 73 0 59 92 38 59] [68 8 29 96 6 9 59 0 48 14 17] [48 23 98 93 83 36 92 48 0 24 12]
[13 92 50 65 38 65 38 14 24 0 67] [9 66 81 47 75 79 59 17 12 67 0]]
  set num-locations 10
  initialize-turtles
end

to go
  tick
  compete
  ask turtles [if random-float 1 < probability-mutation [ifelse use-traditional-mutation = false
[mutate][traditional-mutation]]]
  update-globals
  ;;ask patches [color-patches]
  if (ticks = n-generations) [stop]
end

to update-globals
  let temp [fitness] of max-one-of turtles [fitness]
  if temp > best-fitness [
    set best-fitness temp
    set best-generation ticks
    set best-chromosome [chromosome] of max-one-of turtles [fitness]
    set changes-in-best changes-in-best + 1
  ]
  set worst-fitness [fitness] of min-one-of turtles [fitness]
end

;;This function is used in setup and creates the locations needed to travel to
to initialize-turtles
  crt (sqrt-population ^ 2) [
    set chromosome create-chromosome
    set fitness calculate-fitness chromosome
    set num-vehicles-represented count-num-vehicles-represented
  ]

  ;;update-globals
  set best-fitness [fitness] of max-one-of turtles [fitness]
  set best-generation ticks
  set best-chromosome [chromosome] of max-one-of turtles [fitness]
  set worst-fitness [fitness] of min-one-of turtles [fitness]

  ;;ask patches [color-patches]
end

to color-patches
  ifelse worst-fitness != best-fitness [
    set pcolor scale-color green fitness worst-fitness best-fitness
    ifelse fitness = best-fitness [set plabel-color black set plabel "best"]
    [ifelse fitness = worst-fitness [set plabel-color white set plabel "worst"][set plabel ""]]
  ][ set pcolor blue set plabel "done"]
end

;;creates a list of location coordinates with the index corresponding to the chromosomal representation
to-report create-locations
  let matrix []
  let i 0
  while [i <= num-locations][
    let arr []
    let j 0
    while [j <= num-locations][
      let val 0
      ifelse i = j [ set val 0 ]
      [ ifelse i > j [set val (item i (item j matrix))]
        [ set val ((random 99) + 1) ] ] ;;values 1 to 100
      set arr lput val arr
      set j j + 1
    ]
    set matrix lput arr matrix
    set i i + 1
  ]
  report matrix
end

;;This function is called in setup and creates a chromosome for each strategy
to-report create-chromosome
  let temp shuffle (range 1 ( num-locations + 1 )) ;;randomly shuffle list of locations
  ;;The rest of the code is to insert the return trips into the chromosome
  let returns-to-home find-return-trips
  repeat returns-to-home [ set temp insert-zero temp ]
  report temp
end

to-report insert-zero [ genome ]
  let g genome
  let insert-success false
    let index-range (range 1 (length g))
    let random-index one-of index-range
    while [not insert-success] [
      ;;if no neighboring 0's insert, else look for new space
      ifelse item random-index g != 0 and item ( random-index - 1 ) g != 0 [
        set g insert-item random-index g 0
        set insert-success true
      ]
      [
        ;;if pointed at the end of the list, start from the beginning
        ;;else increment pointer 1 over
        ifelse random-index = ( length g - 1) [ set random-index 1] [ set random-index random-index + 1 ]
      ]
    ]
  report g
end

;;This function is used in create-chromosome
;;it randomly calculates return trips to home simulating multiple-vehicle routes
;;the number of extra vehicles is right-skewed so single vehicle routes are more frequent
to-report find-return-trips
  let returns n-values num-locations [i -> i] ;;ex 3 locations [0, 1, 2]
  let weights reverse n-values num-locations [ i -> i + 1 ] ;; [3, 2, 1]
  ;; https://ccl.northwestern.edu/netlogo/docs/rnd.html#rnd:weighted-one-of-list
  let pairs (map list returns weights)
  report first rnd:weighted-one-of-list pairs [ [p] -> last p ] ;; random number of return trips right-skewed
so 0 appears more often than 1
end

to-report count-num-vehicles-represented
  let num 1
  foreach chromosome [ x -> if x = 0 [set num num + 1]]
  report num
end

to-report zero-indexes [ c ]
  let temp []
  let i 1
  while [i < length c] [
    if 0 = (item i c) [ set temp lput i temp ] ;;if finds 0 add index to end of temp
   set i i + 1
  ]
  report temp
end

to-report add-back-zeros [ c zero-locations ]
  let temp c
  foreach zero-locations [ x ->
    set temp insert-item x temp 0
  ]
  report temp
end

;;This function is run by each patch
;;Visits every location in the chromosome then returns home
;;and records total distance traveled as 'fitness'
to-report calculate-fitness [ c ]
  let total 0
  let prev-loc 0
   ;;for each gene in chromosome
  foreach c [ gene ->
      set total total - (item prev-loc (item gene locations))
      set prev-loc gene
    ]
  ;;return home
  set total total - (item prev-loc (item 0 locations))

  report total
end

to-report crossover [g1 g2]
  let o1 []
  let o2 []
  let bit2 0
  let pos1 0
  let pos2 0

  let continue true
  let start-index 0
  while [continue] [
    ;;Step 2 in algorithm
    set bit2 item start-index g2
    set o1 lput bit2 o1 ;;1st bit from parent2 is 1st bit of offspring1

    let continue2 true
    while [continue2] [
      ;;Step 3
      set pos1  position bit2 g1 ;;find position of bit2 in parent1
      set bit2 item pos1 g2 ;;find bit in parent2 at the same position of parent1
      set pos1 position bit2 g1 ;;find position of bit2 in parent1 again
      set bit2 item pos1 g2 ;;find bit in parent2 at same position of parent1
      set o2 lput bit2 o2

      ;;Step 4 in algorithm
      set pos1 position bit2 g1
      ifelse pos1 != start-index [
        set bit2 item pos1 g2
        set o1 lput bit2 o1
      ]
      [set continue2 false]
    ]

    let ciac check-if-algorithm-complete o2 g1
    ifelse ciac = true [set continue false] [set start-index ciac]
  ]

  report list o1 o2

end

;;reports value not in offspring or true if all values are in the offspring
to-report check-if-algorithm-complete [offspring parent]
  foreach parent [x ->
    if position x offspring = false [report position x parent] ;;report value missing
  ]
  report true
end

to mutate
  let insertions (random (num-locations - 2)) + 1 ;;1 to num-locations-1 insertions
  let i 0
  while [ i < insertions ][
    if num-vehicles-represented < num-locations [
      set chromosome insert-zero chromosome
      set num-vehicles-represented count-num-vehicles-represented ;; or just +1
    ]
    set i i + 1
  ]
  set fitness calculate-fitness chromosome
end

to traditional-mutation
  let zeros zero-indexes chromosome
  let c-zero-removed remove 0 chromosome
  let bit-one random length c-zero-removed
  let bit-two random length c-zero-removed
  while [bit-one != bit-two][
   set bit-two random length c-zero-removed
  ]
  let temp item bit-one c-zero-removed
  set c-zero-removed replace-item bit-one c-zero-removed (item bit-two c-zero-removed)
  set c-zero-removed replace-item bit-two c-zero-removed temp
  set chromosome add-back-zeros c-zero-removed zeros
  set fitness calculate-fitness chromosome
end

to compete
  let old-generation (turtle-set turtles)
  let crossover-count (count turtles) / 2

  repeat crossover-count [

    ; We use "tournament selection". So for example if tournament-size is 15
    ; then we randomly pick 15 individuals from the previous generation
    ; and allow the best-individuals to reproduce.

    let parent1 max-one-of (n-of tournament-size old-generation) [fitness]
    let parent2 max-one-of (n-of tournament-size old-generation) [fitness]

    let zeros-p1 zero-indexes [chromosome] of parent1
    let zeros-p2 zero-indexes [chromosome] of parent2

    ; get a two-element list containing two new chromosomes
    let child-chromosomes []
    ifelse random-float 1 < probability-best-parent-first-cross [
      set child-chromosomes crossover ([remove 0 chromosome] of parent1) ([remove 0 chromosome] of
parent2)
    ]
    [set child-chromosomes crossover ([remove 0 chromosome] of parent1) ([remove 0 chromosome] of
parent2)]

    ; create the two children, with their new genetic material
    ask parent1 [
      hatch 1 [
        rt random 360 fd random-float 3.0
        set chromosome add-back-zeros (item 0 child-chromosomes) zeros-p1
        set fitness calculate-fitness chromosome
        set num-vehicles-represented count-num-vehicles-represented
      ]
    ]
    ask parent2 [
      hatch 1 [
        rt random 360 fd random-float 3.0
        set chromosome add-back-zeros (item 1 child-chromosomes) zeros-p2
        set fitness calculate-fitness chromosome
        set num-vehicles-represented count-num-vehicles-represented
       ]
    ]
  ]
  ask old-generation [ die ]
end
```