Genetic Algorithm Implemented in a Traveling Salesman Problem

Submitted By
Kyle Lamoureux - 7814562

Introduction

Problem Description:
What if one cold Christmas Eve Santa was delivering gifts to the last neighborhood of houses, but in his old age and after the long night realizes he missed a few houses. In order to ensure that the gifts get delivered on time, Santa needs to know what the shortest pathway is he can take to deliver the rest of the gifts.

Approach:
This description is best visualized as a traveling salesman problem (TSP). I’ve chosen to try and find the optimal path between all the missed houses by using a genetic algorithm. It uses a probabilistic selection process, thorough mutation function, as well as single child producing crossover function all of which will be described in more detail below. It is written in processing as it allows for there to be an easily understood visualization of the algorithm running. Also, it allows one to see if the solution the algorithm put forth is optimal by comparing with the human eye.

Purpose:
To explore how a genetic algorithm can be applied to find a fairly optimal solution to an extremely complex problem relatively quickly. Furthermore, I’ve taken this opportunity to test out a few of the different selection, crossover, and mutation techniques to see which would work best in this scenario. Additionally, I've experimented with altering these techniques and combining them to try and improve the resulting output of the problem. Lastly, I’ve documented the resulting output from the program so there can be an easy way to compare the performance of the algorithm when you alter the variables in the table below.

Goal:
The goal set for my genetic algorithm is to be able to solve problem sets of size 20-30 houses (nodes) optimally. If you were to look at every permutation the order of houses can be in to find the perfect solution it would take an unimaginable amount of time. That being said I’m aiming for my algorithm to find an optimal solution in approximately 2 minutes. Finding fairly optimal solutions beyond this goal is a plus and the limits of my algorithm will be tested.

Framework:
During the startup of the program, randomly selected or given house ID’s are stored in an integer array. This array is never reordered again as the individuals in the population don’t keep track of house ID’s but the indexes of those ID’s in the array. Doing this

allows for significantly simpler code in places that alter the paths ordering. Each individual also holds both the fitness (1/path cost) and normalized fitness (fitness/sum) values. The Euclidean distance between every house is precomputed during startup and stored in a two-dimensional array to speed up processing later. Each individual in the population is initialized to a random shuffling of the numbers 0-houses missed. You can run specific test cases that allow you to change variables and see their impact on the output and the difference in how it runs. In addition, you can run randomly selected cases. A spreadsheet of trials has been provided which document the performance of my algorithm. While solving my program displays the populations current best path so you can see the progression through the generations.

Table: list of variables and their initial values which if changed have a impact on the performance of the program.

#Items	20
Mutation Rate	.01
Scaling Mutation Amount	.0001
Crossover Rate	.95
Max Generations Without Change	300
Population Size	500
Keep Rate	.25

Implementation

Prototype:
Starting out I just wanted to get the simplest possible genetic algorithm running so I could start making changes to the individual parts. This simple version consisted of using the Elitism selection technique described in [6] where I would use the individuals from the current population with the best fitness to populate the next generation of individuals. There was no crossover function, which meant the algorithm relied solely on a mutation function for diversity. This mutation function would only check if a mutation would occur once for each ordering of houses then exchange two random indexes. This implementation relied heavily on having a large population as then it had a higher chance to randomly start out with a low path cost. This gave me a good starting spot where I could start implementing better selection, crossover and mutation functions.

Selection:
To improve upon my Elitism selection technique I wanted a strategy which would do the opposite of elitism and increase the diversity of the resulting population. Implementing a probabilistic or “Roulette Wheel” technique was the choice I decided would best achieve this. Described in [4], [6] the probabilistic selection allows the potential opportunity for even the individuals with the worst fitnesses to make it through the selection process. Before I could make use of this technique I had to normalize the fitness of each individual in the population. With there being so many potential permutations, some paths with lower fitnesses would have sections that were really good and I wanted there to be a chance that those sections could make it to the next generation of individuals. First two individuals are probabilistically selected then, either the one with the best fitness is added to the population or the individual produced from the crossover is added which is determined by the crossover rate. With this new selection technique, my program saw an improvement and was able to find a more optimal path on average. However, it now suffers from there being too much randomness which caused the worst cases to be truly terrible. To try and improve the current function I combined both the elitism and “roulette wheel” into one. Combining both these techniques allows there to be a guaranteed selection of the best individuals based on the keep rate, and still have diversity from filling the rest of the population with the selection technique. This improvement is documented in the spreadsheet. If you compare the A2, B2, C2 tables, which have a keep rate of near zero, to any other table which keep rates are higher my algorithm performs substantially better. Through trial and error, I was able to discover the best performing keep rate for my algorithm is approximately 10-35%. Allowing the top percentage of the population to survive each generation means their good traits will be in the next generation so they won’t have to completely relearn them. This technique performs quite well in my goal range for example in figure 1 by the eyeball test it's hard to say for sure if there is a better path. However, as shown in figure 2 you can see as the number of houses increases to the outer limits of my goal range, my algorithm becomes less optimal. You can very easily see a more optimal path between the houses and this shortcoming, and how it could potentially be improved, will be discussed later.

Mutation:
As previously mentioned, if a mutation was triggered then my initial approach was to randomly switch two indexes. This was allowed to occur a single time per individual. This technique along with a small mutation rate left my algorithm with very little diversity which in told made the improvements in the path few and far between. Experimenting with increasing the mutation rate from 1% to something such as 10% didn’t improve the average path cost as much as I desired. To increase the effectiveness of my mutation

function I made it apply the mutation rate to every index in the path array. If a mutation was triggered at an index it would be sent to a swapping function where another index will be randomly selected to swap with it. With some tinkering to find an appropriate mutation rate, this technique improved the performance of my algorithm substantially. Introducing the ability for an individual to have multiple mutations per generation increased the diversity of the population to a sufficient level for the lower bound of my goal range. However, through extensive testing I noticed two things; that the upper bound of my goal range still lacked the diversity needed to find an optimal solution, and that the upper bounds of my goal range tended to run longer and achieve a higher generation before being stopped. These two observations prompted me to try and counteract this lack of diversity by introducing a scaling mutation rate variable. Every 50 generations I would increase the mutation rate by a small amount, capping out at 2.5%. As this is a small feature whose impact can only be seen just before the end of the run it is unclear if it is worthwhile doing. That being said, it doesn't have any negative impact on the algorithm as by the time it becomes impactful the desired path has already taken over the population. These extra mutations could only positively affect it by potentially causing there to be a better section of the path found which could then be added to the most optimal path.

Crossover:
While creating this function the goal was to make sure I was creating as much opportunity for diversity as I could while trying to retain potential optimal sections of the path. I wanted my crossover function to produce one child at a time which I felt would maximize the diversity in my population. In addition to this, I wanted it to try and save sections that might be good however I found that none of the common crossover techniques suited my goal well enough. As a result, I combined aspects of two of the techniques to get my current process. The Cycle crossover technique which tries to preserve as much information as possible about the original positions of the elements [3], [2], [5]. As well as the Maximal Preservation crossover which also tries to keep pieces of the parent in the same order [1]. My crossover function works as follows.

1) Transfer the matching elements in the parent arrays to the child array.
2) Reshuffle non matching elements and insert them into the empty indexes of the child array

Example Below

Parent 1:

1	3	4	2	0

Child: 4, 1, 2 randomly put back into *'s

*	3	*	*	0

Parent 2:

2	3	1	4	0

Some aspects that make me believe this technique works well for my algorithm are as follows. First, since it's probable that two parents of high fitness will be matched together it's also probable that they will have similar paths. This means that the matching sections of the two parents paths which are most likely causing it to have a higher fitness are kept while the other parts are reshuffled. Second, it works in conjunction with my selection technique which allows for both high and low fitness individuals to be parents together occasionally. As a result, good sections in a less optimal path will be kept while the rest will be reshuffled. That being said, there are still some downsides to this technique. Since matching sections are always kept, it has the potential to run itself into a state of stagnation by filling the population with children of very similar paths if not identical. To try and mitigate this aspect I implemented a very thorough mutation function as described above. In tandem with that, I ran a plethora of trial runs to try and find the optimal mutation rate to mitigate this as much as possible. I’m confident that this crossover function works well in my algorithm for both the upper and lower bounds of my goal range. Switching it to another crossover function would have minimal to no improvement.

Execution:
With these newly implemented selection, crossover and mutation functions my program found a significantly more optimal path more frequently than it used to. It now runs as follows; generates a new population by using the selection and crossover functions, then it mutates the population, normalizes the fitness’s, checks for an improvement and then repeats. With this current way of execution in combination with only allowing 300 generations without change the program run time is extremely quick however the average cheapest path over multiple runs is a lot higher than the best path found. This is the result of the algorithm containing a lot of randomness. I was unhappy with having this much inconsistency in the performance of my algorithm. A impracticable but easy potential fix is to set the generations without change to a very large number. This will increase the runtime by a non trivial amount but increase the probability of finding more

consistent optimal solutions. I did not do this because I would like the program to execute in under two minutes so you can see it solving.

Recognizing there is a high variability in the best path found over multiple runs, I took the approach of running the algorithm multiple times in a single run as there would be more chances to find an optimal path. With only allowing 300 generations without change the algorithm never wastes to much time on a solution that's getting stuck, so it has a fast run time. As a result of this I have time to restart the algorithm five times and one more special time and still be under my time constraint. Each restart the locations remain the same but the population is cleared and initialized again to random orderings, before this happens though the single best path is stored for later use. Since there is so much randomness in large tests its extremely rare that a rerun of the algorithm finds the same best path. Once the algorithm has restarted five times It clears the population one last time. However, instead of initializing it randomly it uses the five best runs it stored to now completely fill the new population. This strong run of the algorithm happens a single time then it outputs the best path.

As described above my crossover function works by keeping the matching indexes from both parents. Since the population is full of only really optimal paths this works extremely well to create slight optimizations in cooperation with mutation. This technique doesn't always result in extra improvements as sometimes a very optimal path has already been found in the first five runs, and the randomness of the algorithm causes there to be no change in the best path. This additional strong run will cause the most benefit when the number of houses missed increases to the upper bounds of my goal range. You can see two great examples of this in tables B3, C1 (individual runs in question have been highlighted in yellow) these final strong runs found improvements which made them the best paths found of there test case. Unfortunately this technique still hasn't made my algorithm perfect. It performs really well in the lower bounds but it still struggles in the upper bounds of my testing range but just not as much as it used to. Figure 2 displays this, as it is the best path found after running my algorithm repeatedly but it still contains unoptimized sections. This new technique is still constrained by the time limit, there is only so much improvement that can be found in the limited time i've allowed it to have since the inherent randomness of genetic algorithms. With these changes my algorithm has improved immensely since the prototype, and although there is room for improvement I left it in this state as it finds an optimal path quite well for my target range.

Additional Topic

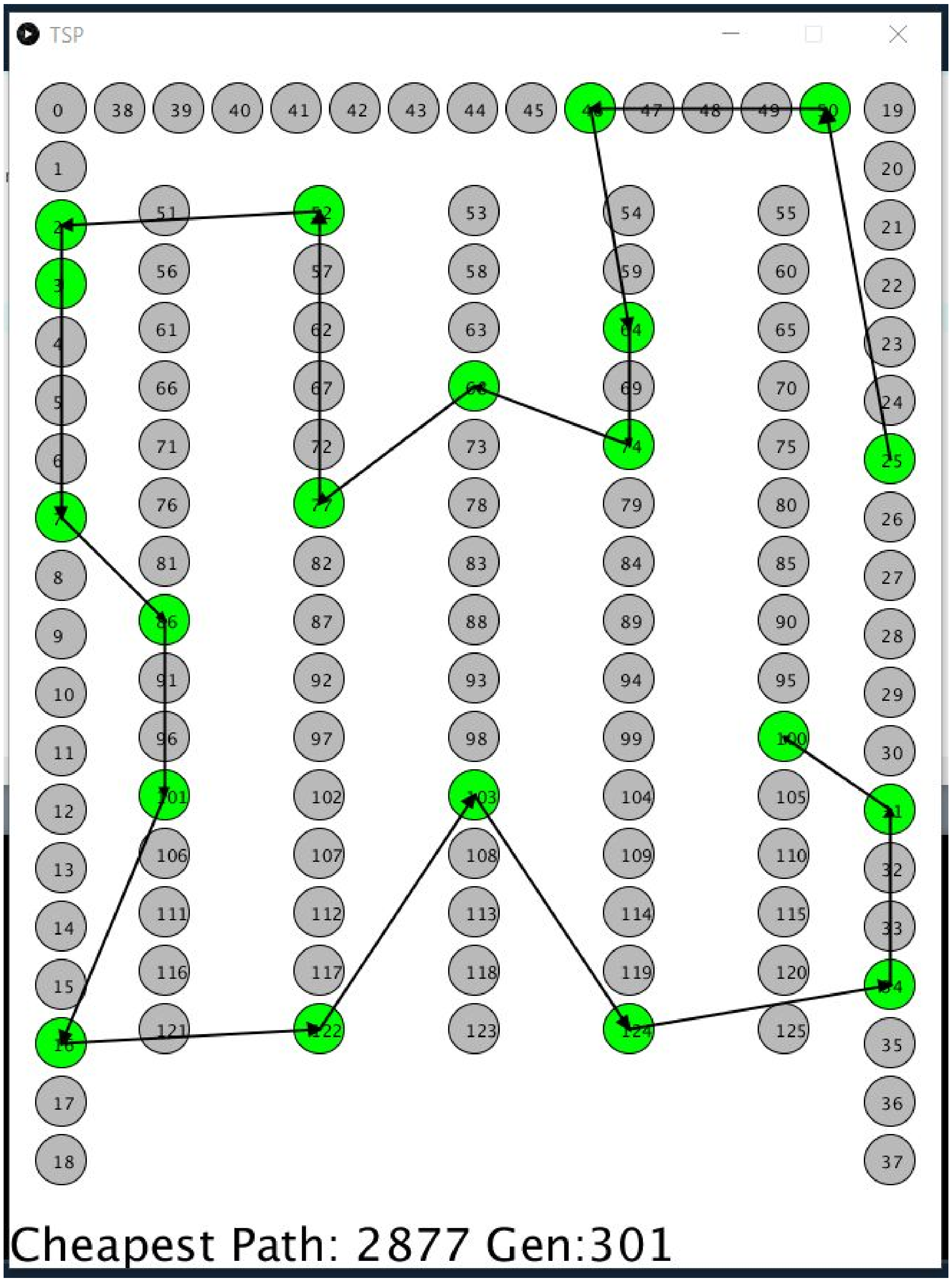
Improvements:
Improving my selection function would definitely increase the chances that an optimal path would be found in my restricted run time. Since my populations are so big using a different form of selection may have given my fitness values a bit more meaning. As discussed in [3] the ‘roulette wheel’ approach isn’t the most effective way to use probability to select parents as it doesnt give a good sampling. Changing the selection function to the Stochastic Universal Sampling which would have followed the same concept of mine but with less randomness [4]. The SUS makes sure paths with the worst fitness are still kept, but gives a more reasonable representation of the best paths. Another potential improvement is that since my population is quite large there are bound to be individuals with duplicate paths. So before the selection process took place I could have went in an added up the normalized fitness’s of individuals who had duplicate paths. Doing this would give the actual path representation of the population. Another avenue of improvements could be found if I removed the time limit I’ve placed upon myself. If you change the generations without change to say 5000 it gives the algorithm more time to randomly mutate into a better path. This change requires you to wait about 16 minutes for an optimal path to be outputted. Comparing Figure 2 to Figure 3, where figure 3 has had these limiting factors removed and let run you can see that it was able to find a more optimal path. This being said, it ran about 8 times longer and only found a path 100 better than the limited way, that is a pretty terrible trade off. Lastly i’m confident pieces of my code could be written more efficiently to reduce the run time of the program as well, which would allow me to add additional runs before doing the super run.

Conclusion

Goals and Further Possibilities:
After a significant amount of testing to fine tune variables, improving the selection, mutation and crossover functions, the overall performance of my algorithm saw massive improvement from my starting prototype. These improvements allowed the algorithm to meet the goals I had set to a sufficient level. It routinely finds a very optimal path in the lower bounds of my target range. As it approaches the upper bounds it still finds fairly optimal paths quite often given the time constraint however there's room for improvement. Set sizes below my target range are very easily solved. Set sizes that exceed my upper bound are poorly handled though. This occurs as I’m limiting the generations without change to ensure the program run time doesn’t exceed 2 minutes.

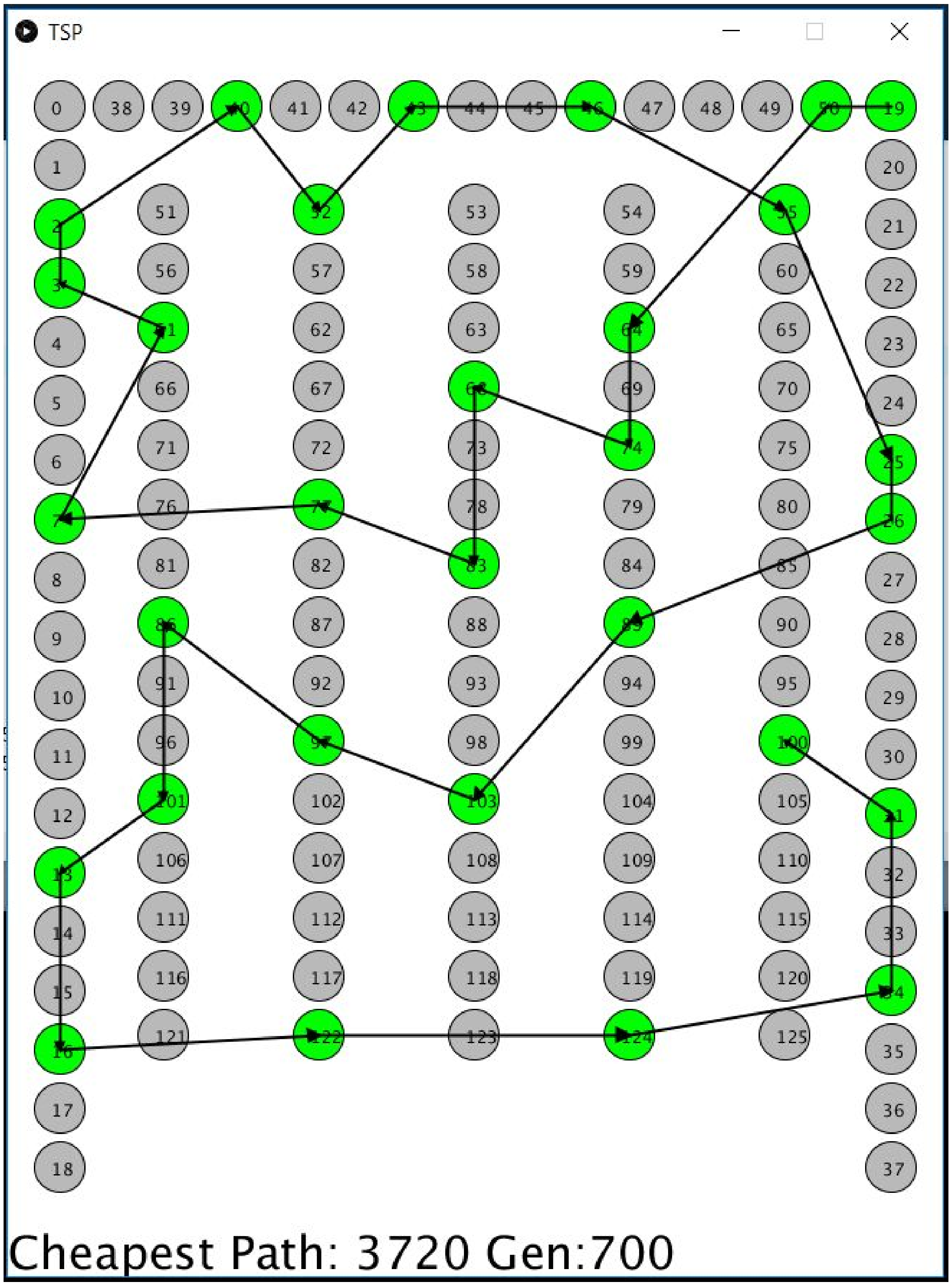
Additionally I’d liked to have be able to further explore the interconnectedness of population size and generations without change. Fiddling with these numbers to see how far I could have pushed my algorithms ability to solve sets in my goal range more optimally as well as sets beyond my goal range. This would have been an exciting avenue to explore. However, it’s just not that practical with my current testing process because as you increase the generations without change cap along with the population size, the run time grows exponentially. All in all I'm satisfied with the outcome of the project.

Figure 1:
Best performing path of 20 houses
25% keep rate



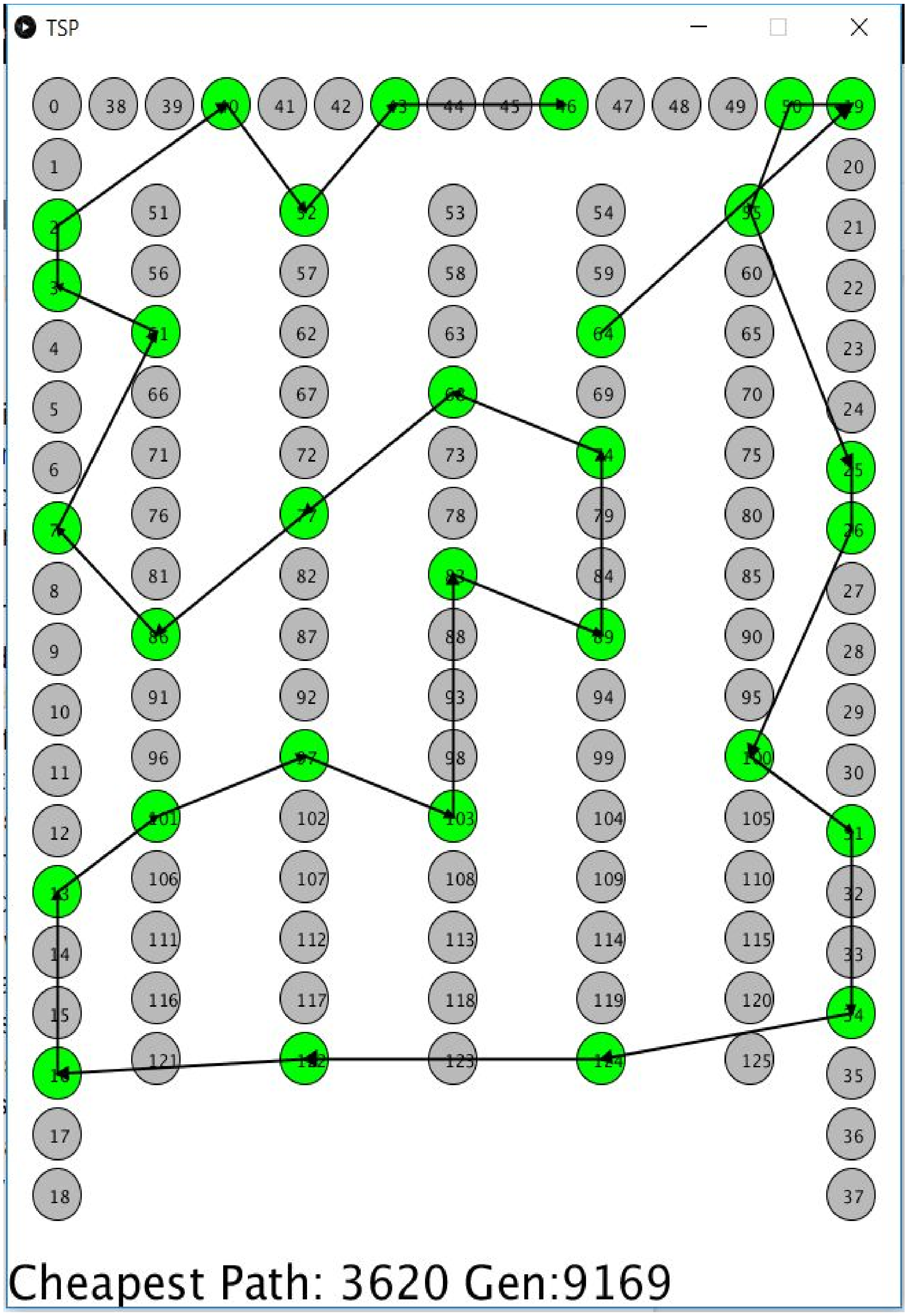
Cheapest Path: 2877 Gen:301

Figure 2:
The best performing path for 30 houses
Keep rate 10%



Cheapest Path: 3720 Gen:700

Figure 3:
Long execution
0 scaling mutation, 5000 generations without change, population of 500



Cheapest Path: 3620 Gen:9169

References:

[1]: A.J., U. and P.D., S. (2015). CROSSOVER OPERATORS IN GENETIC ALGORITHMS: A REVIEW. *ICTACT Journal on Soft Computing*, [online] 06(01), pp.1083-1092. Available at: http://ictactjournals.in/paper/IJSC_V6_I1_paper_4_pp_1083_1092.pdf [Accessed 8 Dec. 2018].

[2]: Carr, J. (2014). *An Introduction to Genetic Algorithms*. [online] pp.20-21. Available at: https://www.whitman.edu/Documents/Academics/Mathematics/2014/carrjk.pdf [Accessed 10 Dec. 2018].

[3]: Eiben, A. and Smith, J. (2003). *Introduction to Evolutionary Computing*. 2nd ed. [ebook] Berlin: springer, pp.28-34, 67-74, 79-84. Available at: https://link.springer.com/content/pdf/10.1007%2F978-3-662-44874-8.pdf [Accessed 8 Dec. 2018].

[4]: Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. [ebook] Cambridge, MA: MIT, pp.7-10, 124-132. Available at: http://www.boente.eti.br/fuzzy/ebook-fuzzy-mitchell.pdf [Accessed 9 Dec. 2018].

[5]: Shiffman, D. (2016). *Genetic Algorithms - Intelligence and Learning*. [video] Available at: https://www.youtube.com/watch?v=c8gZguZWYik&list=PLRqwX-V7Uu6bw4n02JP28QDuUdNi3EXxJ [Accessed 8 Dec. 2018].

[6]: Shiffman, D. (2012). *The Nature Of Code*. [ebook] Available at: https://natureofcode.com/book/chapter-9-the-evolution-of-code/ [Accessed 8 Dec. 2018].

[7]: En.wikipedia.org. (2018). *Stochastic universal sampling*. [online] Available at: https://en.wikipedia.org/wiki/Stochastic_universal_sampling [Accessed 15 Dec. 2018].

Performance Spreadsheet