Graduate Project Write-up

Matthew Chorney

SUNY Polytechnic Institute

Fall 2018

Background

Genetic Algorithms are a way to try a large number of solutions without putting the burden on a human to try and come up with each one. To do this, natural occurrences such as storing information in a “chromosome” and “mutation” of the chromosome as well as “reproduction” between two members of a population are used. Mutation usually involves a random change in the chromosome in the hopes to improve its ability to solve the problem. The reproduction is often done through crossover in which some of each parent’s chromosome are taken and combined to make a new chromosome. This allows for the quick generation of random answers that that build on each other and hopefully improve until a suitable solution is created.

I applied these ideas to trying to find the best path through a graph with random edges connecting the nodes. The final graph is generated by starting with a graph with all the nodes connected to their neighbors, shown in Figure 1. This graph then has random edges between nodes removed to create the final graph used in the problem of which an example is shown in Figure 2. When trying to find the best path around the graph, the algorithm isn’t allowed to either leave the graph (stopped from trying to move in a direction that goes outside the graph) or to pass through a node it already traveled through. While abiding by these rules it tries to make moves that generate the highest score. If it gets multiple chromosomes with the same score the algorithm considers the shorter one to be better. One point is added to the score when a legal move is made but one point is deducted from the score when it changes the direction it is traveling. To achieve the highest possible score, the algorithm must find a path that travels as far as it can in one direction before changing directions and then doing the same in the new direction. Each direction that can be chosen as a move is represented by the numbers 1 through 8 with 1 as North, 2 as North East, 3 as East, 4 as South East, 5 as South, 6 as South West, 7 as West, and 8 as North West. An example of this representation in a chromosome can be seen in Figure 3. This problem was implemented and tested in Python2.

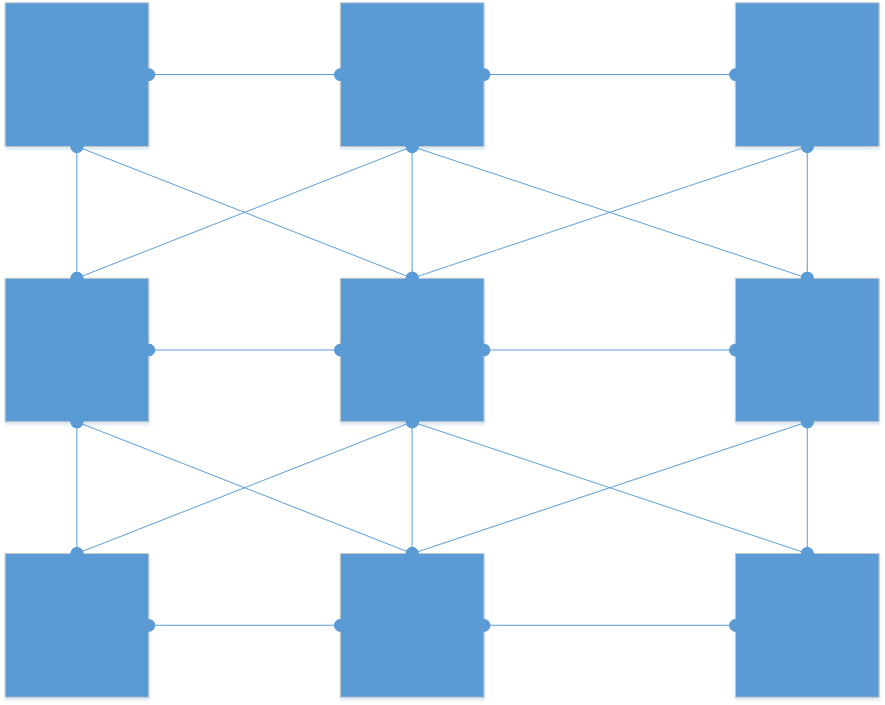


Figure 1: Full Graph

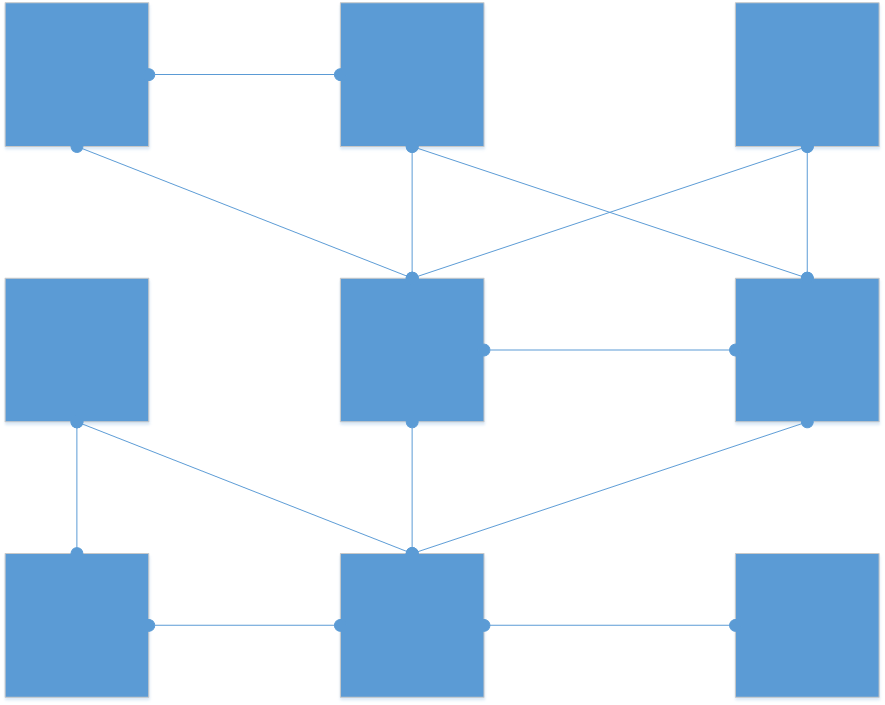


Figure 2: Removed Graph



Figure 3: Example Chromosome

Generating the Population

The population used in the program is randomly generated each time it is run, but the method for doing so changed over its development. Each member of the population, the size of which is a set length in the code, is randomly generated to have between two and a max value set number of moves. Each of the moves, 1 through 8, have an equal chance of appearing as options. Originally, I left the members of the population unchanged and continued to run the algorithm from here, but I later found that this would make it harder for the algorithm to find better scores. With each of the moves being random there is a low chance of any of them being the same as their neighbor which means there is a low chance of it trying to travel in a straight line from the start. This obviously isn’t optimal if the algorithm is trying to find the highest score possible. I later added a step after the generation which changes a third of the moves, at random, in a chromosome to be the same as their neighbor for each member of the population. This means the algorithm has a head start on finding better solutions since there are already some chains of the same moves in the chromosome.

Mutation

Mutation is the way I randomly change the moves in the chromosome in hopes of making its score improve. There are three types of mutations I had implemented originally were, adding a new move, removing a move, and swapping two moves. When adding a new move, I pick a random location in the chromosome and insert a random move into that position, which can be seen in Figure 4. I found that adding a random move however wasn’t always useful in increasing the score since moving in a line is a better way to improve the score. As such I gave the move added a fifty percent chance as being the same as its left neighbor therefore a better chance of forming a line and better improving the score. When deleting a move, I check if the number of moves in the chromosome is greater than one and if so I then remove a random move from it as seen in Figure 5. The swap is done by selecting two random moves and switching their locations in the chromosome shown in Figure 6. I later added the swapping of groups of similar moves to the possible mutations. This, as shown in Figure 7, will pick a random position and check the moves next to the one selected to see if they are the same move, for example, multiple 1s together. This single move or group of similar moves are then moved to a different location in the chromosome at random. This mutation didn’t seem affect the outcome for the trimming fitness function but was able to improve the ignoring fitness function, both of which will be discussed later.

I found through testing though that by mutating like this and taking the changes blindly would sometimes cause the mutation to make the chromosome’s score worse. This isn’t necessarily a problem, but it caused the algorithm to stagnate when trying to find new solutions because they could only get so long before a mutation randomly destroyed a part of it. To solve this, I created a greedy version of the mutation which only saves the mutation if it improves the score of the chromosome it is operating on. This solves the earlier mentioned problem by no longer allows mutations to destroy parts of answers.



Figure 4: Insert Mutation



Figure 5: Remove Mutation



Figure 6: Swap Mutation



Figure 7: Swap Section Mutation

Fitness

The fitness function is used to calculate the score of each chromosome in the population. I developed two fitness functions over the course of this project based on the secondary function I have each do. The first fitness function I made to help trim illegal moves from the chromosomes while the second one is made to ignore the illegal moves and continue evaluating the score.

The trimming functionality was added to the original fitness function to help deal with the accumulation of illegal moves in the chromosome. It would originally stop evaluating the fitness of the chromosome when it detected an illegal move and return the fitness up to that point. This was fine in terms of calculating a score but because the illegal move and all the moves after that weren’t being evaluated and were left in the chromosome the mutation of the chromosome could mutate parts of it that weren’t being evaluated which would cause no change in the score. As such I had the fitness function also return the position of the illegal move in the chromosome which I would then pass into a trimming function along with the chromosome where I would delete the illegal move on. This meant that whenever the chromosome is mutated a move that is important to the score is always being affected.

A fitness function that ignores illegal moves was made as an alternative to the trimming function to see if a different method of scoring could increase the scores of chromosomes. As opposed to the previous fitness function this one ignores any illegal moves it finds and continues as if they weren’t there. This is done based in the hope that some moves beyond the illegal ones are useful to finding the optimal path. While there are still illegal moves in the chromosome there is the thought that maybe through mutation, either swapping an illegal move with another, adding a move before it, or removing a move before it, it might become a legal move and therefore it still has value in being part of the chromosome. The chromosome being littered with illegal moves though can make it hard to read if it needs to be outputted to the screen to view so I created a cleaning function that like the fitness function goes through the chromosome and when it finds an illegal move it deletes only that move. This makes sure that the chromosome contains only the actual moves it made through the graph to get its score.

Running the Algorithm

When running the program there are two sections of code that can be run if they are set to. One runs the trim fitness function and the other the ignore bad moves fitness function, which is the only difference between them. The program will start from every node in the graph and run the algorithm until it does so for the number of times set by the “limit” variable. This gives every node a chance to be the start of the best path since not all nodes have the same edges connecting them. Before starting at each node, the population is recreated using the earlier mentioned method to ensure each one has a fresh start.

The main loop of the algorithm is then executed a set number of times. First, each member in the population is mutated using the mutate function, after which their score is calculated with the appropriate fitness function. If the trimming version of the fitness function was used, then the results of the function are then sent into a trimming function to remove the unused moves if needed. Each member of the population is then checked against the best path found so far and if it has a better score or has the same score as the best, but a shorter path it is saved as the new best. When a new path is saved as the best its moves, score, and starting location are saved. The population is then sorted and the worst 1/3 of population, determined by score, are deleted. The number deleted are then re-added as either newly generated members created the same way the population is, or a crossover from two surviving members of the population with a fifty percent chance of each member re-added being either one. When all the runs for every node are finished, and the ignoring fitness function was used, the whole population and the best stored path are run through a “cleaning” function where the previously ignored moves are removed since the algorithm is finished and they are no longer needed.

Result

I generated and used a seven by seven graph shown in Figure 8 for the gathering and evaluating my implementation of solving this problem. In the Figure ‘\*’ represents a node, ‘-’, ‘|’, ‘/’, and ‘X’ represent edges between the nodes with the ‘X’ representing two

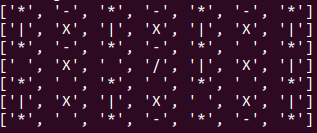


Figure 8: Graph Used for Results

edges crossing. I then ran both the trim fitness and ignore fitness versions of the algorithm and recorded the best score found for each with different values set for limit. The results of this are shown in Figure 9. As limit is set to higher values, allowing the algorithm more time to search, both methods are able to find better

|  |  |  |
| --- | --- | --- |
| Limit | Trim | Ignore |
| 10 | 3 | 4 |
| 100 | 5 | 6 |
| 1,000 | 6 | 7 |
| 5,000 | 6 | 8 |
| 10,000 | 6 | 8 |

Figure 9: Best score for Limit runs

solutions up to a certain value of limit. For trim, after 1,000 it can no longer find a higher score than 6 while ignore is able to find better values until it finds a score of 8 at 5,000. After this the higher value of limit, allowing for more search time, is wasted time for a graph of this size. I also recorded the best found over time for both trim and ignore fitness in the 10,000-limit run and graphed their improvement in Figure 10 and Figure 11. The graph of trim fitness’s improvement over time shows that it finds new bests infrequently compared to the number of iterations it’s

Figure 10: 10,000 Trim Fitness

running but it also never finds a shorter path with the same score. This is most likely do to the fact that it both trims illegal moves so the path it finds is already the shortest. Ignore fitness however has many more bests recorded over time with many being recorded with the same score but with a shorter path. I also recorded the bests that were saved throughout the run and their starting positions which can

Figure 11: 10,000 Ignore Fitness

be seen in Figures 12 and 13. Trim fitness’s scores all start from the node at position 0,0 in the generated graph meaning it never found a path either with a better score or with the same score but a shorter path when starting from any other node in the graph. This is again likely because of the way it mutates to find new paths it already has a very short path for that score. Ignore Fitness also has many paths saved as the best that start at the node 0,0 but it later finds better ones from other starting points. It was able to find a path with a score of 8 starting at that node, so it took starting from many of the other nodes before it could find a path of a similar score but shorter. For ignore fitness, the idea of the path being shorter could be wrong since many of the moves in it could be illegal but are left in because of how the fitness function’s main mechanic works. This means that while the number of moves in the best may be shorter the path that is actually used in calculating the score could be the same

|  |  |  |
| --- | --- | --- |
| Score | Path | Start Node Position |
| 1 | 5 | 0,0 |
| 2 | 3, 4, 6, 6 | 0,0 |
| 3 | 3,3,3 | 0,0 |
| 4 | 3, 3, 3, 6, 6 | 0,0 |
| 5 | 3, 3, 3, 6, 6, 6 | 0,0 |
| 6 | 3, 3, 3, 5, 5, 5, 7, 7 | 0,0 |

Figure 12: 10,000 Trim Fitness

|  |  |  |
| --- | --- | --- |
| Score | Path (before “cleaned”) | Start Node Position |
| 1 | 4,4,6 | 0,0 |
| 1 | 5,4 | 0,0 |
| 3 | 3, 3, 6, 6, 3, 5 | 0,0 |
| 3 | 3, 3, 6, 6, 3 | 0,0 |
| 3 | 3, 3, 3 | 0,0 |
| 4 | 3, 3, 3, 4, 1, 1, 5, 5, 1 | 0,0 |
| 4 | 3, 3, 3, 6, 6, 3 | 0,0 |
| 5 | 3, 3, 3, 6, 6, 6, 3 | 0,0 |
| 5 | 3, 3, 3, 6, 6, 6 | 0,0 |
| 6 | 3, 3, 3, 5, 5, 5, 3, 7, 7, 5, 3, 7 | 0,0 |
| 6 | 3, 3, 3, 5, 5, 5, 5, 7, 7, 5, 5 | 0,0 |
| 6 | 3, 3, 3, 5, 5, 5, 3, 3, 7, 7 | 0,0 |
| 6 | 3, 3, 3, 5, 5, 5, 7, 7, 4 | 0,0 |
| 6 | 3, 3, 3, 5, 5, 5, 7, 7 | 0,0 |
| 7 | 3, 3, 3, 5, 5, 5, 7, 7, 6, 2, 6, 6, 7, 4, 1, 2, 7, 7 | 0,0 |
| 7 | 3, 3, 3, 5, 5, 5, 7, 6, 8, 8, 8, 8, 7, 7, 5, 3, 3 | 0,0 |
| 8 | 3, 3, 3, 7, 8, 8, 5, 5, 5, 7, 5, 6, 7, 8, 5, 6, 3, 3, 2, 7, 2, 7, 7, 2 | 0,0 |
| 8 | 2, 2, 2, 5, 5, 5, 7, 8, 3, 2, 3, 7, 8, 5, 2, 2, 7, 2, 4, 2, 7 | 6,0 |
| 8 | 3, 3, 1, 1, 1, 6, 6, 6, 1, 2, 2, 7, 7, 5, 5, 5, 7, 2, 3, 2 | 6,2 |
| 8 | 3, 3, 1, 1, 1, 6, 6, 6, 1, 2, 2, 7, 7, 5, 5, 5, 2, 3, 2 | 6,2 |

Figure 13: 10,000 Ignore Fitness

length. The actual path taken by the best found for Ignore fitness can be seen in Figure 14. The path saved as best is 3, 3, 1, 1, 1, 6, 6, 6, 1, 2, 2, 7, 7, 5, 5, 5, 2, 3, 2 but after “cleaning” we can see that the path it actually takes is 3, 3, 1, 1, 1, 6, 6, 6, 1, 2, 2, 7, 7, 5. This path has many long straight paths that go as far as possible which is what earned it the score of 8. Since it also travels through almost every node it is likely the best score available for this graph which is why as shown in Figure 9, a score of 8 was found when limit was set to 5,00 and 10,000. Looking at Figure 15 though, which is the path saved as best for the trim fitness, it is clear there are more moves it could take to add to its score. It could be having trouble finding better paths because of its trimming functionality. Since it is trimming illegal moves it is possible that the swapping moves and removing moves mutations don’t help it as much as the adding new moves mutation. Taking a move that means “go east” and moving it to a new spot in the chromosome will likely either cause a set of move to become illegal or for the score to lower since there are no “extra” moves to move around that aren’t important to the path.

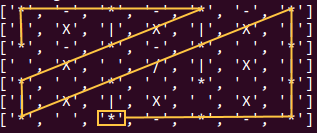


Figure 14: 10,000 Ignore Path

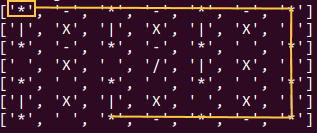


Figure 15: 10,000 Trim Path

Conclusion

These are two solutions to this problem that I implemented out of many possible ones. While the trim fitness solution didn’t end up doing very well at solving the problem, the ignore fitness solution did significantly better and was perhaps able to find a path with the highest score possible for the testing graph. In the future, perhaps, this problem could be approached with spanning trees. A spanning tree is a subset of a graph with all the nodes included but missing edges so that there is no path where there is a loop. This is acceptable because any loops would break the rules of the problem which could allow for this to be a quality solution. Dijkstra's algorithm could also be a notable solution to this particular problem. Dijkstra’s algorithm is used to find the shortest path between two nodes in a graph which could be used so slowly build paths between nodes until a longer path with a higher score is created. In these ways there are many more ways to approach solving this problem and perhaps ones could be found that are able to create better paths faster than the two ways I presented in this paper here.