Comparison of Optimization Techniques for Road Networks

New Mexico
Supercomputing Challenge Final Report
April 2013

Team #31
Desert Academy

Team Members
Sara Hartse
Sean Colin-Ellerin

Teachers
Jocelyne Comstock
Jeff Mathis

Executive Summary

The goal of this project is to investigate the capabilities of two different optimization techniques for the task of approximating an optimal solution for a set of edges for a node network. The two methods we are examining are genetic algorithms, which use the principles of evolution by natural selection to 'grow' better solutions and and stochastic hill-climbing with uses random search points' descent on a surface to search for minima. The programming was done in Python and Matlab. The network optimization problem that we chose is based on a traffic flow idea with certain nodes representing start or 'work' locations and the others as destination or 'home' locations. The quality of each solution is based on the average travel time (based on a shortest path algorithm) from each work to a set of home nodes. Currently, we have succeeded in implementing the two different optimization methods and have run experiments on each method demonstrating its success with different parameters. We also developed a program that loads an image into a Python plot and allows us to identify the home and work nodes and then converts these nodes into a network that can be analyzed in the main programs. We selected a college campus map to analyze as it is smaller than a city and therefore easier to run experiments with.

We learned about the two optimization methods and the nature of their functionality. We found that the genetic algorithms program was able to successfully optimize the road network by 23% from the initial networks, while the stochastic hill-climbing program was found to require too much computation time because of the number of networks needed to create a smooth surface. Both of these methods can be improved upon, but we think that the method genetic algorithms is the better optimization technique for an optimization problem of this kind.

Contents

1 Heuristics

In the modern day, computers are relied upon to provide solutions to very intricate and complicated problems. Some of these problems can be solved exactly with a computation of the single solution, but many computational problems are so complicated that the discovery of the solution is practically infeasible. These are types of problems that require such a huge amount of computing power that the time (time to find the solution) or space (the physical space taken up by machines) requirements are unrealistic. When faced with a problem of this type, it becomes worthwhile to design a method that discovers approximate (also called partial) solutions within a certain time-frame [1]. These methods are known as heuristic algorithms and they are designed to suggest solutions to optimization problems, which seek a minimum or maximum for a certain objective function. Note that the search for a maximum or minimum is the same thing since a maximum can be transformed to a minimum by multiplying by -1. This extremum is determined by the objective function, which assess the quality of the solution, according to the particular constraints of the problem. Heuristic algorithms essentially work by searching a sample space of solutions and using various methods to find most 'fit' result according to this objective function [1]. There are numerous variations of heuristic algorithms that combine and hone techniques and our project seeks to examine the success of two different heuristic methods in the area of network optimization.

2 The Problem

Our project is based on the optimization problem of finding the ideal, or rather an approximation of the ideal, road layout for a given city. The road layout of a city is very important to its inhabitants, affecting aspects of daily life like commute time, but also larger issues like the emergency evacuation capabilities of the population and the carbon emissions of the city as a whole. The goal of our project is to compare two optimization techniques in their ability to take a set of destinations that comprise a city and generate a set of roads to connect them as well as possible. We hope that the model will provide insight as to the functionality of the two different optimization techniques as well as the characteristics of an ideal road network and possible improvements to existing networks.

For the purposes of this problem, a city and its road system is simplified into a node map, with each node representing a destination with a fixed location, and edges connecting those nodes, representing roads. In this way, the problem can be viewed as network optimization with network properties such as degree (the number of edges per node), and connectivity affecting quality of the solution. In 1997 Frank Schweitzer et al. did a similar investigation of road network optimization. They described the requirements of a 'good' road network to be these: firstly, that the destinations are well connected enough that the traveling distance between them is not too great and, secondly, that the cost of building the network (which is assumed to be proportional to the length and number of roads) is not too large either. If only the first requirement was considered, then the ideal network would clearly be one where every node is connected to every other node (the degree of everyone is equal to one less than the number of nodes in the network). Considering only the second, the best network would be a 'minimal link system,'where each node is connected to the whole structure by only one edge. Clearly these two criteria are in opposition to each other and a good solution to this problem would be a road network that balanced the two requirements as much as

possible. Schweitzer describes this problem as belonging to a set of frustrated problems by a huge number of good solutions spread out across a very rugged landscape, so clearly heuristic algorithms are the proper approach to finding a good solution [2]. Schweitzer, et al investigated the heuristic methods consisting of variations on genetic methods. We too will be examining the ability of genetic algorithms in solving this problem, but will compare them to the more classical method of stochastic hill-climbing. Another difference in our model is that that we are seeking to optimize the problem taking into account varying capacities of edges (edges that allow cars to pass at great speeds, but cost more).

3 Optimization Methods Being Tested

3.1 Hill Climbing

3.1.1 History

The simplest search algorithm is the exhaustive search which simply evaluates every single option in the search space and finds the best. This method always finds the best solution, but is as infeasible as the original problem because of the time required. The local search is similar to the exhaustive search, but it focuses on specific areas of the search space, using a hill-climbing method to move to the best solution of all neighbors. The local search is much faster than the exhaustive search, but often converges too soon. Another traditional method is the branch and bound search which moves through the search space like the exhaustive search, but seeks to eliminate certain areas or branches that cannot yield better solutions. This method is quite effective in finding approximate solutions, but often requires too much time [1].

For many optimization problems, these methods are inadequate because they are either too costly or not successful in their approximations of solutions and modern heuristic algorithms are designed to improve on these methods and overcome these drawbacks [1]. One common solution to this problem is to use a stochastic exploration of the search space. This is often done by selecting a set number of random points in the search space, which descend the surface, until it can no longer be descended. For discrete data, the descent is either executed using a small neighborhood around the point, against which the point can be compared for its 'fitness', known as hill-climbing, while for quasi-continuous data, the descent occurs using the gradient at that point, know as stochastic gradient descent. As a result of these descents, it is possible that one of the minima found is the global minimum. However, for a large search space, it is difficult to create a large enough number of points that can be

executed in a reasonable computation time yet too few points result in a low chance of finding the global minimum. Thus, many optimization programs use simulated annealing, which is based on the local search and hill-climbing method, but avoids premature convergence by occasionally accepting solutions that are worse than the current best, with the probability of this acceptance decreasing over time. This allows a point in the search space to get over or around a peak to find a better minimum. Simulated annealing is considered a stochastic hill-climbing process and is quite successful for certain problems, though is fallible in the same ways as local and branch and bound searches [3].

3.1.2 Process

Hill climbing is an iterative optimization method for finding local extrema in a search space. A set of n starting points is selected from the search space either randomly or in a pre-selected pattern, depending on the nature of the data. For each starting point a neighborhood of a certain size is selected and the points therein are compared to the starting point for their fitness with respect to the fitness function. The optimal point with regards to the fitness function then becomes the new starting point and the process is repeated until the algorithm arrives at a point whose neighborhood contains points that are all less fit than that point, thus indicating that the point is a local minimum. From the n selected starting points, one obtains a set of local minima, the minimum of which one hopes is the global minimum of the search space. However, using the hill-climbing method, it is not possible to determine whether the minimum of the set of resultant local minima is in fact the global minimum or how far it is from the global minimum. The success of the hill-climbing algorithm depends upon choosing a suitable neighborhood size and a suitable number of points. If too many points are used then the search space can be 'over-exploited', meaning that many of the same minima are found by the starting points, thereby adding unnecessary computation

time, while too few points decrease the likelihood of finding a minimum near to the global minimum [7].

3.2 Genetic Algorithms

3.2.1 History

The founder of the field of genetic programming is John Holland, a scientist who, during his undergraduate work at MIT in the mid-1940s, became interested in the similarities between biology and computational processes [4]. He worked on projects in the new field of Artificial Intelligence in the form of designing a computer program that could learn to play checkers and improve as it played more opponents. Holland drew the conclusion that machines could be designed to react and adapt to their environments, to outside stimuli, in the same manner as animals. Holland believed that as humans had a reasonable understanding of the governing principles of nature, these principles could be distilled into mathematical abstractions and applied to a system, resulting in life-like results [4] He theorized that the key was to start from a chaotic, random system, then apply the rules and let nature take its course. The book *The Genetical Theory of Natural Selection*, by evolutionary biologist R.A. Fisher, complemented Holland's theories: a mathematical approach to the theory of evolution which applied the tenants of physics and logic to those of biology. Fischer believed in the benefits of reconciling these often very separate fields, asserting that he could imagine no more beneficial change in scientific education than that which would allow each to appreciate something of the imaginative grandeur of the realms of thought explored by the other [5]. Fisher presented the process of natural selection as a definitive and logical procedure.

Holland saw how natural selection, just like learning how to do a task, is a mechanism to adapt to one's environment, a type of learning that takes place not in a life, but over generations. The fundamental idea behind genetic programming is evolving computer programs to

solve problems independently; genetic programming relates to the idea of developing a form of 'automatic programming' [6]. As the name implies, genetic programming is inspired by the natural processes of life, specifically evolution by natural selection. This takes the form of programs that are bred and optimized using the fundamentals of Darwinian evolution.

3.2.2 Process

A genetic algorithm is a computer program that is a type of formula based on the principles of evolution by natural selection. The program begins by creating a population of various possible solutions, each representing an individual. Each of these individuals is evaluated based on their aptitude in solving the problem that they were assigned, on producing the desired output. This evaluation is known as a fitness function and once all individuals are evaluated, they reproduce with varying degrees of success and die. The subsequent generations repeat the behavior. The success of reproduction is based on how well a given individual's fitness rating compares to the others in the population. The individuals with the highest ratings will reproduce and those near the bottom will die off without passing on their characteristics. This results in a tendency for the most successful traits to become more prevalent in the population and, as generations pass, a group of individuals better and better at solving the problem.

4 Models

To allow for a valid comparison of the two optimization techniques, the two models use the same parameters and mathematical equations that dictate the character of the road networks and the average time taken for the cars to arrive home. Both models begin with a node map of a circular city whose radius is 60. The distances and times are not on a direct scale in kilometers, seconds, etc, but are designed to be relative to each other and proportional throughout the model. Edges within 0.3 of the radius are designated as work nodes, while all other nodes are home nodes. Edges are placed randomly between nodes and each edge is assigned a random capacity from 1 to 5 on a standard normal distribution with $\mu = 2.5$ and $\sigma = 2.5$. Both models ensure that each node has an edge, which results in the network being fully connected. Since there are very few tunnels and overpasses, even in very large cities, the road network is made planar by adding a new node, i.e an intersection, wherever edges cross each other. Edges are added until the cost of the network reaches the budget, which is calculated as follows,

$$Cost = (\sum_{i=1}^{n} d_i \cdot c_i) + 5 \cdot (\#\ of\ intersections) \tag{1}$$

where n is the total number of edges, d_i is the distance between the two nodes of the ith edge, c_i is the capacity assigned to the ith edge, and $\#\ of\ intersections$ is the total number of intersections in the network. The factor of 5 is multiplied to the number of intersections because this value gives the number of intersections a reasonable weight relative to the distance and capacity of roads for the cost to build these features of a city. The budget for this cost function is assigned relative to the number of the nodes.

Each 'work' node is then assigned a 'home' node and the shortest path is determined using a shortest path algorithm. From these paths, the fitness function is determined as the

average time per car, calculated as follows:

$$Time_{avg} = \frac{\sum_{s=1}^{n}(\sum_{i=1}^{r}\frac{d_i}{c_i} + numnodes_s \times numcars_s)}{\sum_{s=1}^{n} numcars_s} \tag{2}$$

where n is the total number of paths, r is the total number of edges on the sth path, d_i is the distance between the two nodes of the ith edge on the path, c_i is the capacity of ith edge, $numnodes_s$ is the number of nodes on the sth path, and $numcars_s$ is the number of cars that start at the sth start node.

The number of cars assigned to each start node is calculated as proportional to the size of the network because the larger the city, the more cars there will be in the business district. The number of cars also decreases with respect to the distance of the start node from the center of the city because more people work at the center of the city than at the outskirts . Thus, the number of cars is calculated as follows,

$$numcars_s = \frac{numnodes}{3}(r - |startnodes_s|) \tag{3}$$

where numnodes is the total number of nodes in the network and $\frac{numnodes}{3}$ is the proportionality constant, r is the the radius of city, and $|startnodes_s|$ is the distance of the sth start node from the center of the city.

4.1 Stochastic Hill-Climbing Model

4.1.1 Creating Networks

The hill climbing method of optimization was implemented through a Matlab model of a road network. This model begins by either creating a random node layout within the boundary of the city (radius = 60) or using a real-world city layout of nodes, each of which is assigned a value in the Argand plane. There are two reasons for giving the nodes complex values

rather than (x,y) coordinates. Firstly, it is easy to create a set city limit by limiting the magnitude of the complex values. Secondly, from a coding standpoint, it is much simpler and tidier to store one value for each node as opposed to two. Edges with a random capacity are then placed between the nodes until the budget is reached, as discussed previously. To ensure that the network is planar (no crossing of edges), an intersection is created if an edge is created on top of another.

4.1.2 Calculating Fitness

Then, each 'work' node randomly chooses a 'destination node' and, using Dijkstras algorithm, the shortest path length from the starter node to the destination node is determined. Each node is assigned a number of cars according to Equation 3. The number of intersections, the number of edges, and the average time is then stored for that network. By using intersections and edges as the two independent variables of the optimization, the results will indicate if it is beneficial to have a network with many edges and few intersections, i.e; edges that do not cross often, or many intersections and few edges, i.e; edges cross very frequently, or some middle ground (the model does not allow for a lot of edges and a lot of intersections because of the budget constraint).

4.1.3 Hill-Climbing of Surface of Data

After the above procedure is executed for a large number of networks, a surface is then fitted to the data using an interpolation function, where the x-axis is number of intersections, the y-axis is the number of edges, and the z-axis is time, since time is the variable being minimized. 10 points are then chosen at random from the surface. Since the surface is a matrix, each point has 8 surrounding points, which is used as the neighborhood for evaluating the point. For each point, the time values are compared with the 8 surrounding points and

if any are less than the starting point, that point becomes the new starting point, which is repeated until all of the 8 neighbors have time values greater than the starting point. Each initial starting point returns a local minimum, which is then stored.

4.1.4 Experiments

The hill-climbing optimization program was tested on standard surfaces to test that it is a functional optimization program and finds the minimum. It was first tested on the parabaloid shown in Figure 1 centered at, and thus with a minimum at, (31,31,0) to avoid negative numbers in the search (although the algorithm is robust with regard to negative values). When the algorithm was run on this surface, all 10 points found the minimum.

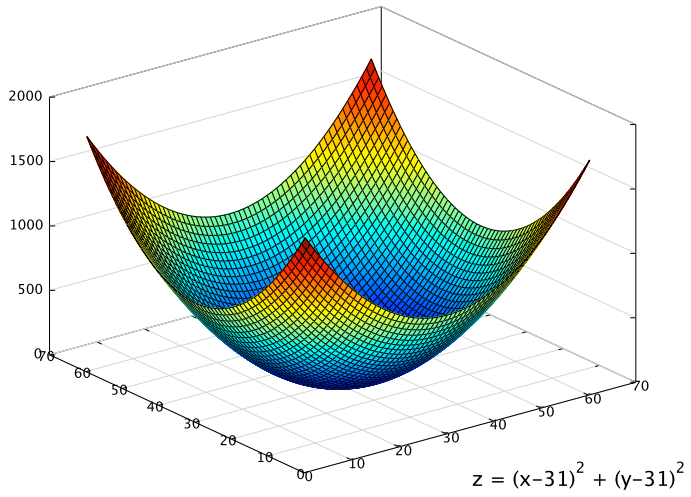


Figure 1: Parabaloid Centered at (31,31,0)

The algorithm was then tested on the Rosenbrock function, as seen in Figure 2, which is a classic optimization test function, whose equation is: $z = (1-x)^2 + 100(y-x^2)^2$, and has a global minimum at (1,1,0). Yet, to avoid negative numbers once again, the function was shifted to a global minimum of (60,60,0) and scaled down, resulting in the following

equation: $(1 - \frac{x-30}{30})^2 + 100(\frac{y-30}{30} - (\frac{x-30}{30})^2)^2$.

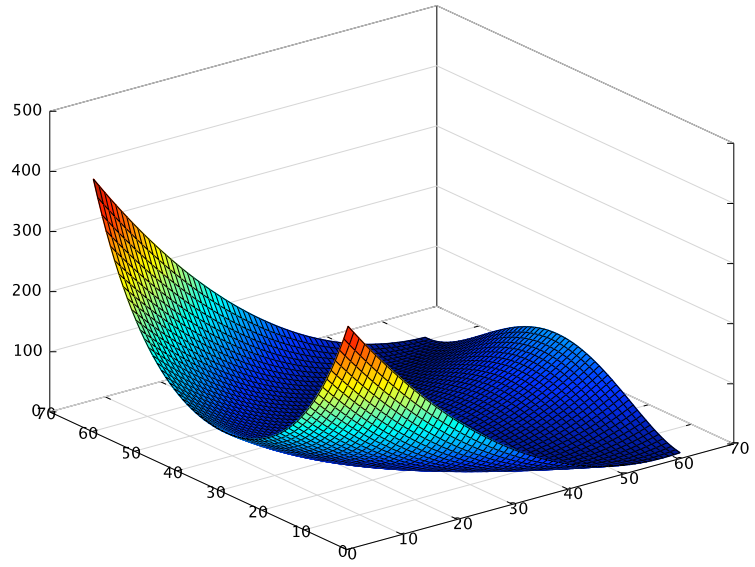


Figure 2: Rosenbrock Function with Global Min. at (60,60,0)

For this function, the algorithm finds 9 points at (50,43,0.12346) and 1 point at (58,56,0.00641). This latter point is a very good approximation of the minimum, and thus we can be satisfied that the algorithm functions adequately.

4.2 Genetic Algorithm Model

This optimization method program was written in Python. It makes use of the Python library *networkx* which was designed for creating and manipulating network graphs.

4.2.1 Creating initial population

This method begins by generating the first generation of network solutions. All members of the population are based on the same node graph, the same map of destinations, but each member of the population is given a random set of edges. The edges are randomly distributed, as long as they do not intersect with an existing node, until the entire network is connected (it is possible to reach every node from any node). Each edge is also given a random value, essentially a 'weight', which represents the capacity, or number of cars that can be handled by the given road. The other criterion for graph creation is that the cost of all the edges falls with a certain budget. The cost is calculated by multiplying the length of each edge by it's capacity (and described in Equation 1).

4.2.2 Evaluate Fitness

The next step is to evaluate the fitness of each of the graphs. This is done by averaging the shortest paths between nodes in the center (representing the workplaces) to random target nodes on the outside (representing homes), as per Equation 2. The program does not evaluate the path length from every possible work and home combination, but selects a certain portion of the population as the number of iterations to test and then averages the selection. The distance of each edge in the shortest path is converted to an approximation of time by dividing the length of the edge by its capacity. This accounts for the idea that in a maximum traffic situation a car will be able to move more quickly on a road with more lanes, so, in this model, travel time is inversely proportional to edge weight. After the fitness has been evaluated each network is ranked based on their fitness rating. The networks with the lowest times are preferable, so they are ranked highest.

4.2.3 Selection of reproducing population

Next, the networks that will go on to reproduce are selected. This is an important step of the genetic operation: survival of the fittest specifies that the most fit are most likely to reproduce and that some of the least fit are eliminated, but the criteria for selecting which set will reproduce is variable. Currently, in this model a certain percentage is decided upon and the parents are randomly picked from that pool. To add more variation that might be closer to the chaotic systems in nature so it might be useful to have a certain chance of individuals outside this pool reproducing as well. This is a commonly seen idea in heuristics: accepting some bad solutions to see if it will lead to other local minima.

4.2.4 Reproduction

Once a group of individuals is selected to reproduce, two members of the pool are selected at random. The first is designated as Parent 1 which will be the base of the new network and a copy (the child network) is made. A random node is selected which will act as the recombination point. All the edges connected to this node in the child network are deleted and replaced with all the edges connected to this node in Parent 2 (preserving the edge attribute of weight).

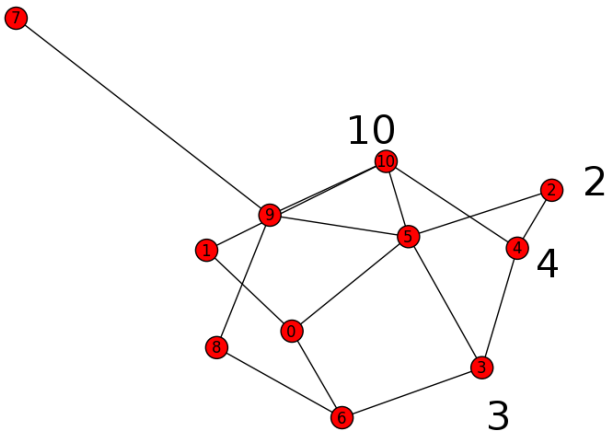


Figure 3: Parent 1

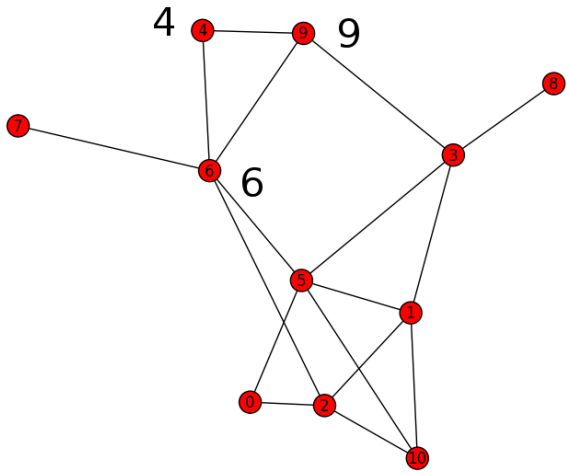


Figure 4: Parent 2

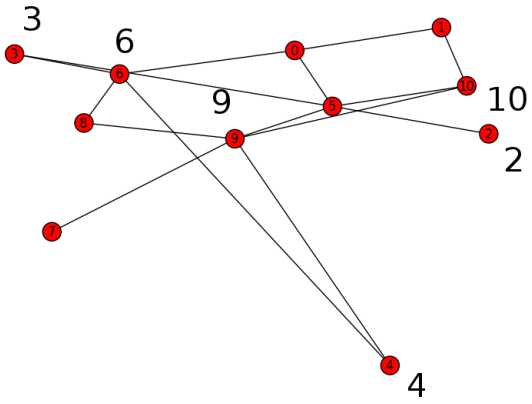


Figure 5: Child

Figures 3 and 4 show two parents. The two are recombined at Node 4. The child graph (shown in Figure 5) is a replica of Parent 1 (with respect to the connections as the nodes do not have fixed locations in this example), but instead of having Node 4 connected to 10, 3 and 2, it is connected to 9 and 6, as in Parent 2. Since all the networks are made from the same node graph, the child is a direct combination of its parents. Next the graph is checked to make sure every node is connected. Sometimes a certain amount of mutation is added because the recombination results in disconnected nodes and other edges are added to regain connectivity. Then the graph's cost is checked. If it is over budget edges are removed and if it's under budget edges are added. This additions can be considered analogous to random mutations (traits that are not inherited from either parent).

4.2.5 Experiments

There are many different variables at work within the genetic algorithm, many with regard to the 'genetic' nature of the program. For example, the population size of individuals being tested could be significant. In evolution, this is comparable to the having a large pool of organisms to select from and potentially increase the viability of offspring due to diversity. An experiment was performed on a small, randomly generated sample network. In each trial the node map, the budget and the selection function remained the same, while the number of individuals in each generation changed.

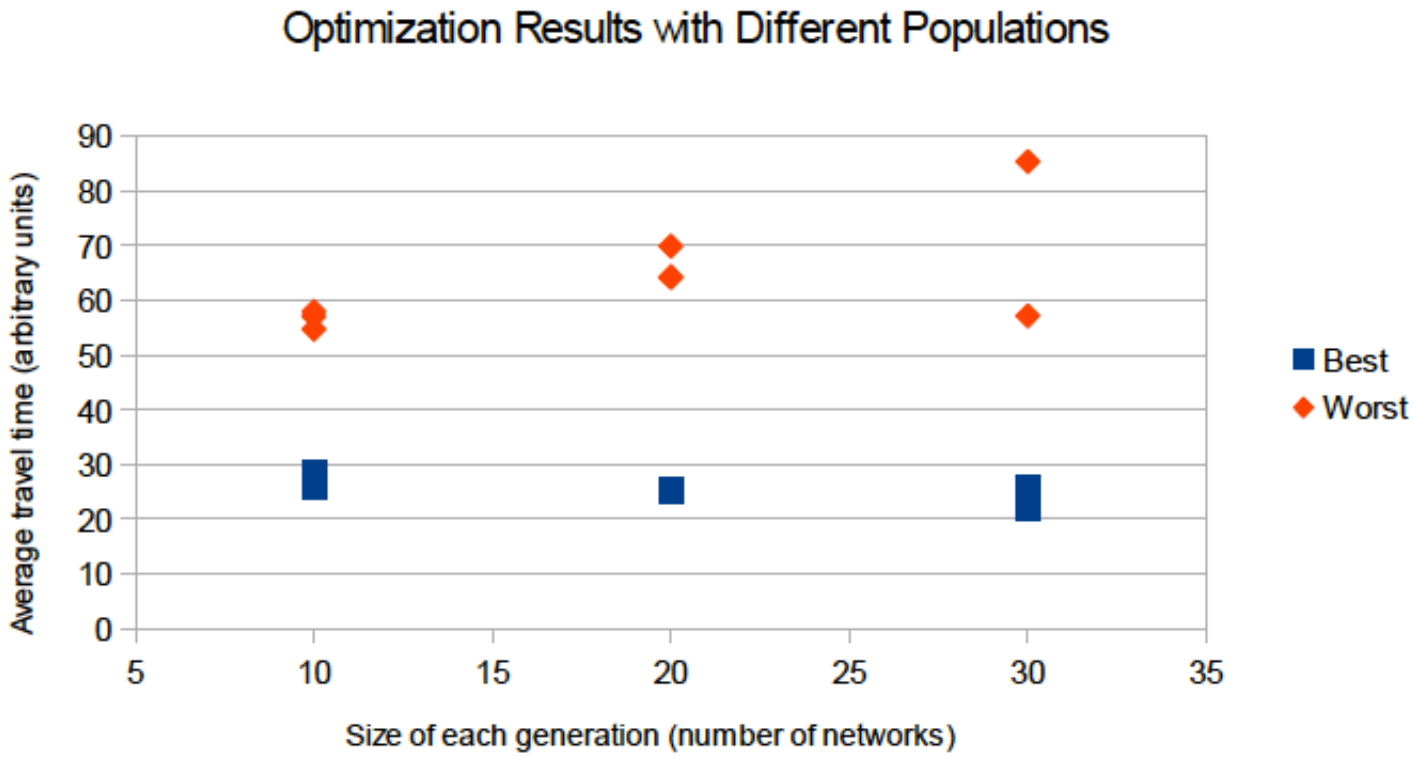


Figure 6: An experiment varying the size of the population

As seen in Figure 6, changing the generation size does seem to have an effect on the quality of solutions. Both the most and least successful graphs were discovered in the trial with the generation size of 30. This makes sense because a larger range of individuals were created so presumably there was more diversity in each generation.

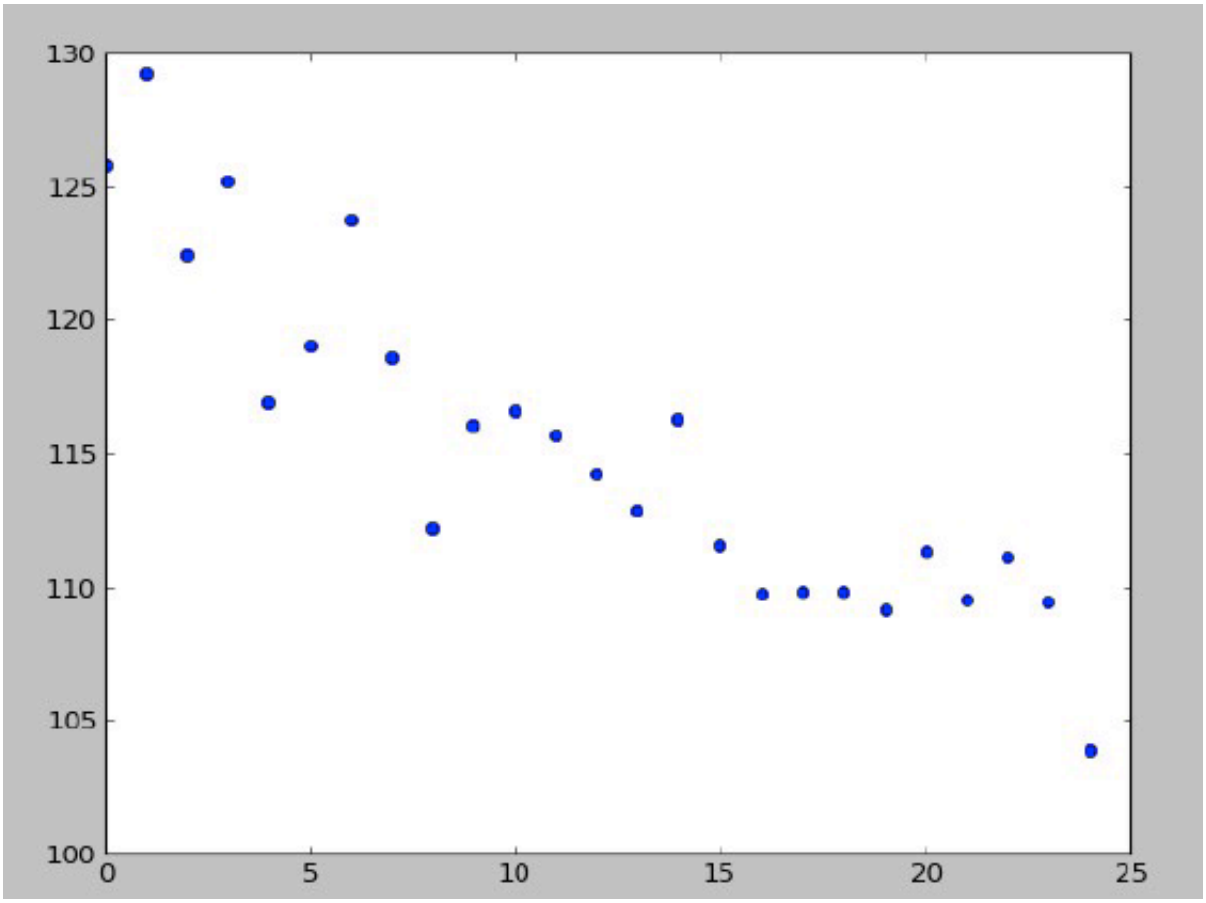


Figure 7: The fitness of the best solution from each generation

Another thing it is important to demonstrate is that the quality of the solutions improve over time, otherwise the program is just a random parameters sweep. Figure 7 is a graph of the quality of solution of the best network in each generation over 25 generations. It is clear that while the quality of solutions are not monotonically decreasing, they do have a overall downward trend. This means that the genetic algorithm is working successfully in minimizing the fitness function and improving the networks as time progresses. However,

it is important to realize that after a certain amount of time the quality of the solution increases by such a small margin that it becomes less worthwhile to keep running after many generations. Also, it is possible that after a certain amount of time the best solutions stop improving. For example, Figure 8 shows the best solutions of a simulation run of 30 generations. The best solution was found in Generation 23, but 22 and 24 are quite close, but after 23 iterations the quality of the solutions start to worsen. It is likely that they would start to improve again, but it may not be worthwhile to continue running the experiment.

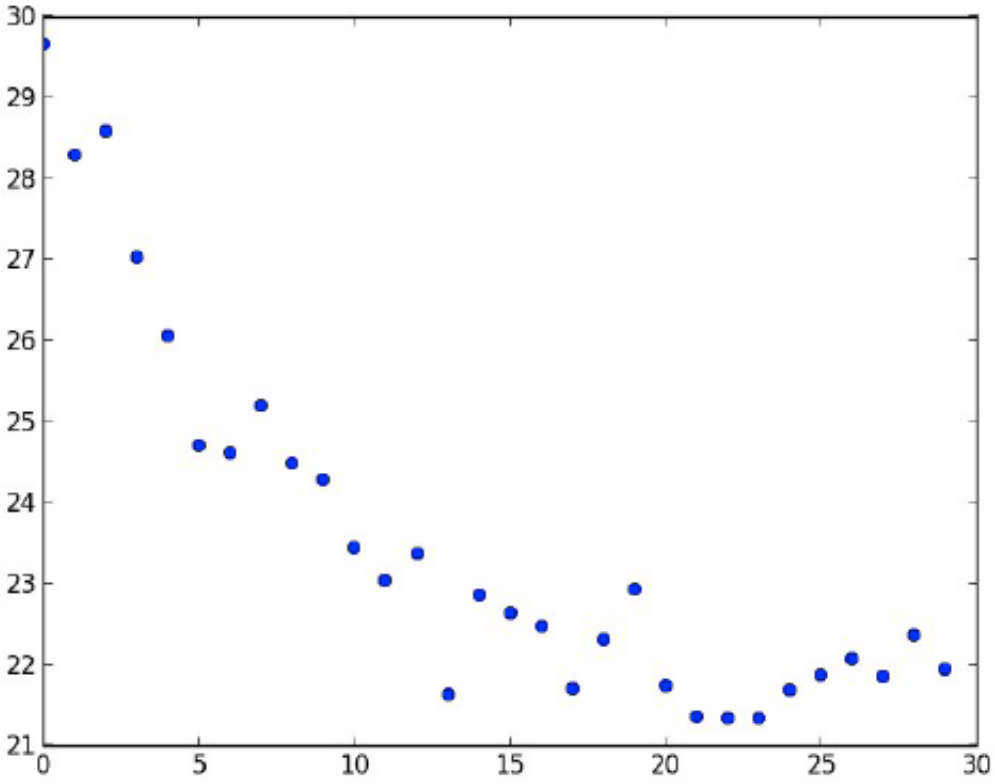


Figure 8: The best solutions of a simulation run over 30 generations

Another interesting facet of the genetic algorithm is the selection function which determines which individuals will go on to reproduce. In the previous tests, the selection function was simply taking the top 30% of each generation and using those as the parents for the next generation, but is this ideal?

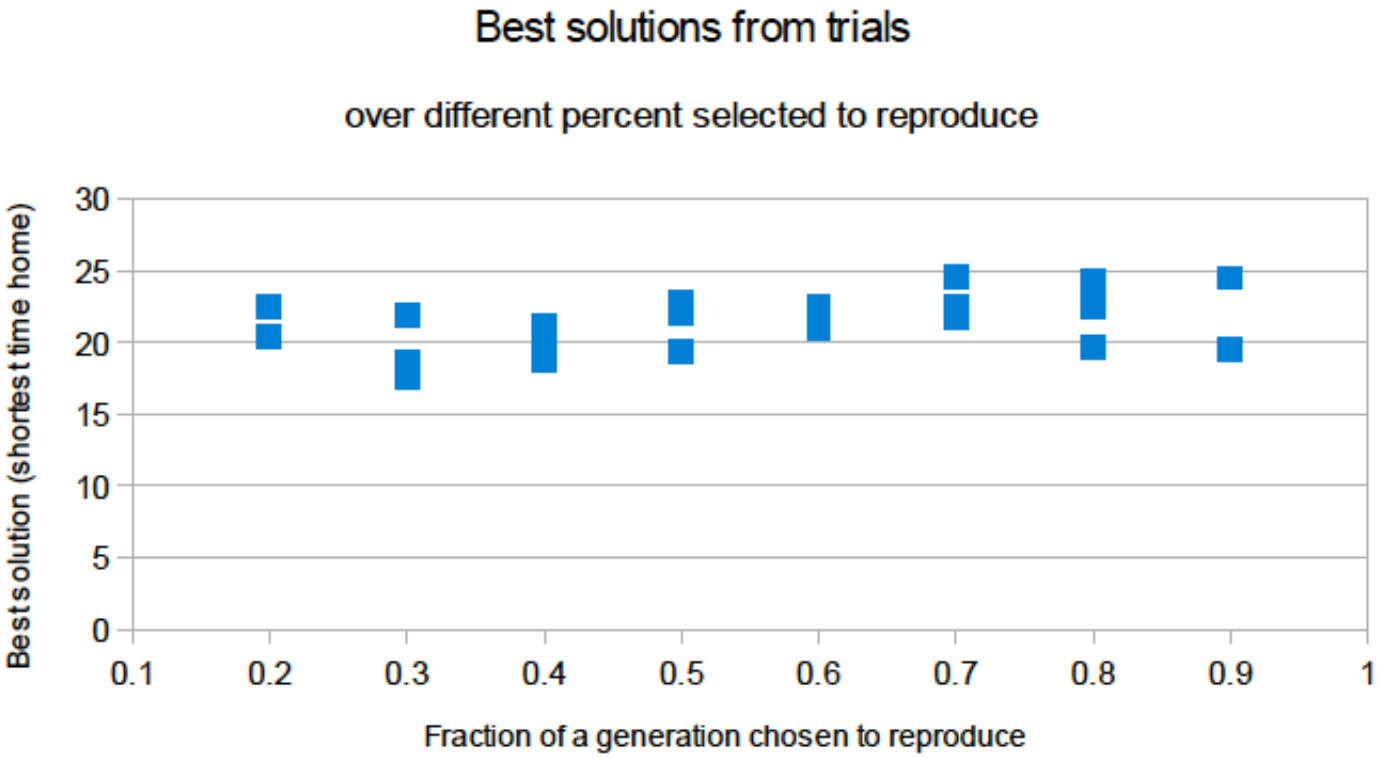


Figure 9: The minimal time found by simulations with different percentages of each generation selected to reproduce

Figure 9 shows the results of a simulation on the same node network with three tests on selection percentages from 20 to 90%.

While the solutions are all within a relatively close range (from 17.6 to 24.6), there does appear to be a difference in quality of solutions. The very best solution (17.6), as well as the best average solution, were found with a selection percentage of 30%. The largest best solution and the largest best average solution were found at 70%. There seems to be an advantage to having a percent selection that isn't too large and doesn't include too many less successful solutions, but also having one that isn't too small and allows for more variation. This conclusion mirrors the concept of evolution by natural selection which does not only allow the very best to pass on their genes, but through chance, lets a certain amount of less successful organisms reproduce as well.

4.3 Node Maps From Images

The eventual goal of our project was to apply these optimization methods to an actual city. We first attempted to determine work and home nodes computationally using an edge detection, however, the data from the edge detection was too noisy. Also, determining whether a node is a home node or work node can only be done by a human with a key. Thus, using the Python library Matplotlib we created a program that detects mouse events on an image. Currently, we have a Python script which loads images and allows the user to select locations on the image to be different types of nodes. The list of coordinates and their node type are scaled to fit on the standard plot and then converted into a new network. This network is then 'pickled' (it is saved and its data type preserved) and can be loaded into one of the two optimization programs. This allows us to take a map of an area we want to investigate, identify the key 'home' and 'work' nodes and run this abstraction of a city through our program.

4.3.1 Experiments

The model was tested on the map of Brown University's campus as shown in Figure 10 below:

Figure 10: Campus map of Brown University

where the yellow buildings are the dormitories (home nodes) and the blue buildings are the classrooms (work nodes). Using a college campus map allows for the easy assignment of nodes as home or work nodes and can be considered a microcosm of a city. From the map, we selected 30 nodes and scaled their coordinated to our 60 radius network space. For this node map, we found that a budget of 4000 is adequate.

The nodes extracted from this image are placed in our network as shown below:

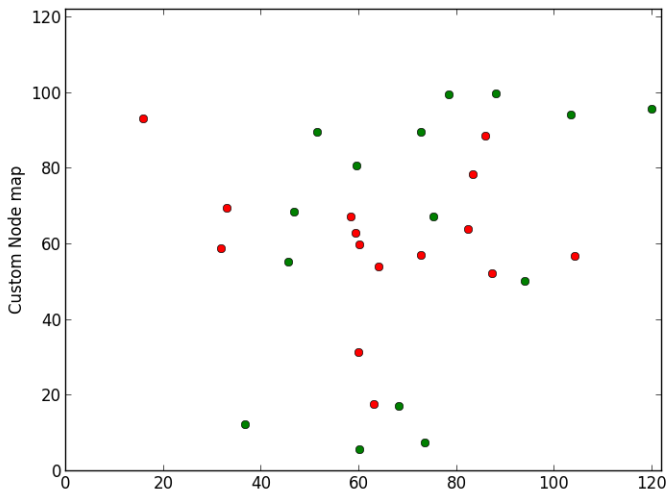


Figure 11: Node Map of Brown in Python Model

Note that the node map is rotated 180 degrees. The reason for this is that .png files have an origin in the top-left corner, while we want an origin in the bottom-left corner, therefore the node map was shifted accordingly, resulting in the rotation. However, orientation does not affect the functionality of our programs whatsoever. The following image (Figure 12) from the Matlab model of the Brown node network with edges in place demonstrates that the two models are creating the same node network and illustrates how many edges are created when the budget is set to 4000.

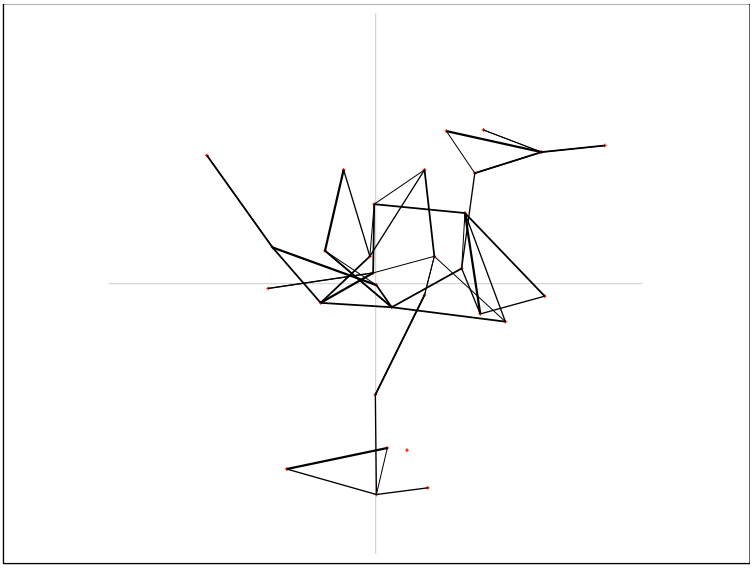


Figure 12: Sample Brown network from Matlab model

5 Results & Analysis

5.1 Genetic Algorithms

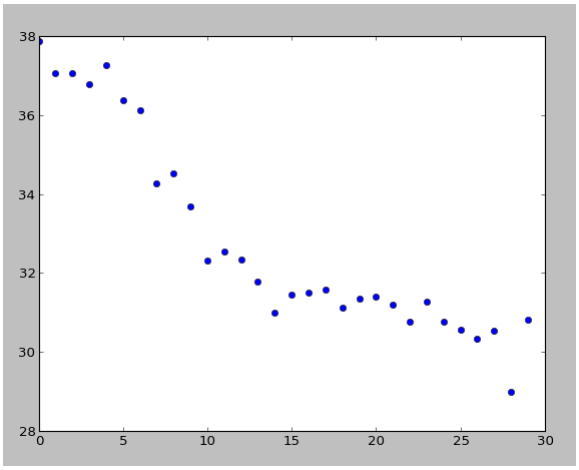


Figure 13: Best road network from each generation

In the experiment optimizing the nodes based on the Brown campus map, we ran a genetic algorithm simulation with a generation size of 20, 30 generations in total and a selection to reproduce of 30%. The results of the simulation, namely the time of the best graph in each generation are plotted in Figure 13. The optimal solution is showed in Figure 14 and the worst solution is in Figure 15. It is evident that the genetic algorithm did succeed in reducing the travel time of the network. The best solution was a reduction in the travel time from the best graph in the first generation by 23.5%. An interesting behavior of this method is that as the generations pass the program begins to lock in on a certain graph and the difference between each member of the population as well as between generations becomes less significant. Near the end of the simulation even the positions of edges stop changing and

only their capacities vary. This is most likely a feature of there being little genetic mutation in children as well as the relatively small generation sizes. It could be problematic due to the fact that initial graphs constrain the features of the overall population and could lead to premature convergence. It would be interesting to add more pronounced mutation within the reproduction process and see what kind of effect that had on the quality of the solutions.

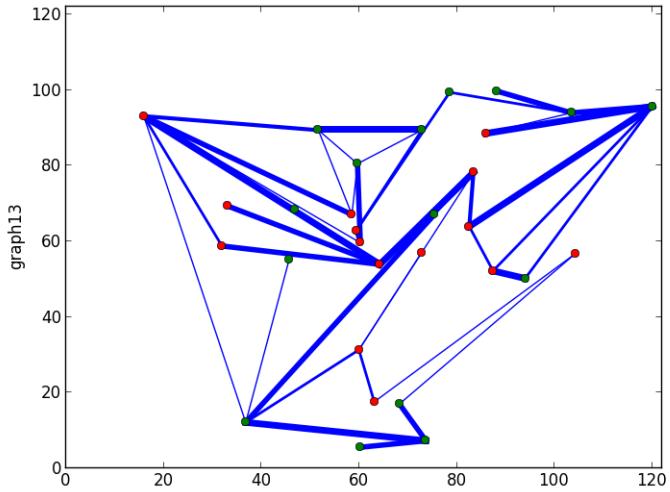
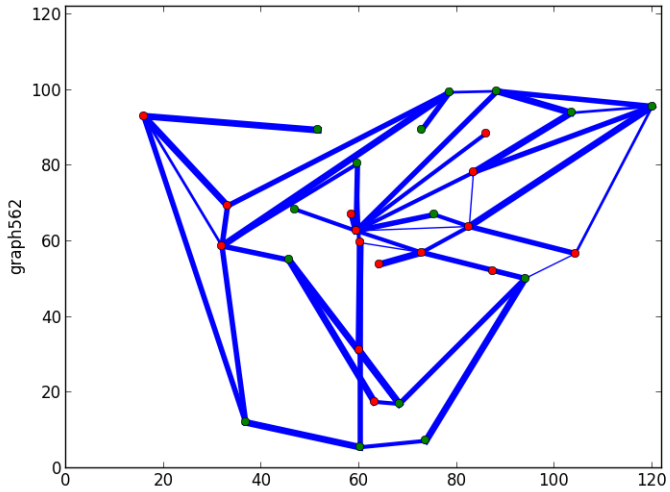


Figure 14: Worst road network overall

Figure 15: Best road network overall

5.2 Stochastic Hill-Climbing

For the implementation of this optimization technique on the Brown campus map, the model could only create 20 networks in reasonable computation time (approx. 3 hours). Therefore, the surface was not nonzero throughout the surface space, rather each data point was a smooth peak (as opposed to a single point peak because of the interpolation of the data), which resulted in all the random points of the search space returning a minimum of 0 for time. In building the 20 networks, the computation time was very similar to the time taken for the genetic algorithm program in building networks, therefore the models were similar in their



efficiency. However, the stochastic hill-climbing method requires many more initial networks than the genetic algorithms optimization to create a surface that is able to be descended to a meaningful minimum. Thus, to find minima this method is more computationally costly than the genetic algorithms method.

6 Conclusion

At this point in our project we have succeeded in creating methods to generate random, connected networks based on a given set of nodes. We also programmed two different optimization methods that proved successful, to at least some degree, in optimization of these networks. Finally, we developed a program that allowed us to approximate new node maps based on existing images and maps. We ran various simulations on each of the optimization techniques to gain a greater understanding of their functionality and behavior under varying circumstances. We also began to compare the the success of the two programs on the same network, one that was an approximation of Brown University's campus.

One idea that we concluded is that genetic algorithms seem particularly suited to this kind of problem. This network optimization problem was easily adapted to genetic algorithms because, while complex, it consists of something with a physical structure that be replicated and slightly altered for the optimization. On the other hand, a classical optimization technique, in this case stochastic hill-climbing, seems to be less suited to a network optimization problem. We believe this has to do with the complexity of the solutions and because so many points are needed to conduct a search of the search space, which leads to unreasonable computation time. In addition, we found that for a problem of this type, the complexity makes it difficult to determine what to minimize in the hill-climbing method. Although many parameters can be included in the fitness function, the surface minimized only allows two parameters. We conclude that stochastic hill-climbing is better suited to less complex programs, where many points can be created in reasonable computation time to allow for the creation of a surface that covers the parameter space, and thus accurate descent of that surface. Genetic algorithms allow for a more encompassing and dynamic optimization method which was also faster computationally, requiring fewer start nodes.

There are several limitations on our comparison between these two optimization tech-

niques. Firstly it is important that we resolve certain inconsistencies between the two methods, especially with regards to the question of intersections. At present, the genetic algorithm seeks to eliminate all intersections and the stochastic hill-climbing method creates a certain number of intersection nodes with the budget. Essentially, this is a function of the two programs not being quite at the same point in development and unfortunately makes it unrealistic to directly compare the results of the two simulations. We intend to fix this inconsistency soon, but in the meantime it does give a certain amount of insight about the benefits of creating intersections.

Another extension of our model that we've been considering in the implementation of parallel processing in the program, in particular for the stochastic hill-climbing method. Currently, the models are relatively quick in their main optimization functions, but they do take quite a long time creating the initial set of points. Because each generation of a new network can be done independently of the others, threading might be a good way to improve the computation time. One of the most challenging things about writing the genetic algorithm was generating the initial generation of road networks. This is because one of the criteria for an edge to be created is that it does not cross any other existing nodes and testing every existing edge in the network for intersection becomes computationally difficult once there are a large enough number of nodes in the network. Another issue is that the method in the python code of testing intersection is based on converting edges to a linear equation and testing values of that equation over a range where both edges are defined. This works well unless an edge is a straight vertical line which means it does not have a domain to be compared to. Figuring out a more efficient and more successful method of finding edge intersections would certainly improve the Python model.

7 Acknowledgments

We would like to thank our two teachers, Jocelyn Comstock and Jeff Mathis for all their help. We would also like to thank user tcaswell from stackoverflow.com for his advice on detecting nodes from images by clicks and we would like to thank our parents for their support. Finally, we would like to thank all those involved with the Supercomputing Challenge for their extensive work in organizing this program. We have been doing this program for the past four years and it has made a big difference in our high-school education.

References

[1] N. Kokash. An introduction to heuristic algorithms. University of Trento, Department of Informatics and Telecommunications. Available at `http://www.researchgate.net/publication/228573156_An_introduction_to_heuristic_algorithms`.

[2] F. Schweitzer, et al. Optimization of road networks using evolutionary strategies. *Evolutionary Computation 5.4* (1997): 419-438.

[3] M. E. Ayndin, T. C. Fogarty. A Distributed Evolutionary Simulated Annealing Algorithm for Combinatorial Optimization Problems. *Journal of Heuristics* vol. 24, no. 10, Mar. 2004.

[4] Steven Levy. Artificial Life. N.p.: Cape, 1992. Print.

[5] R. A. Fisher. The Genetical Theory of Natural Selection. Oxford: n.p., 1930. *Universal Library.*

[6] Matthew Walker. Introduction to Genetic Programming. Tech. N.p.: University of Montana, 2001. Print.

[7] Zbigniew Michalewicz, David B. Fogel. How to Solve It: Modern Heuristics. Springer: April, 2004. Print.

A Python Code

A.1 Node Map from Image

```
#!/usr/bin/env python

import math
import matplotlib.pyplot as plt
import networkx as nx
import numpy as np

#Imports and image and opens it as a plot
image_name = 'brown_campus_map'
im = plt.imread('brown_map.PNG')
ax = plt.gca()
fig = plt.gcf()
implot = ax.imshow(im, origin = 'lower')

#allows the user to chose atype of node to place
Node_type = input('type: work, home or intersection')
if Node_type != 'home' and Node_type != 'work' and Node_type != 'intersection':
    print('invalid node type: choose work home or intersection')
Node_locations =[]

#Registers click events and saves the coordinates of the click
def onclick(event):
    if event.xdata != None and event.ydata != None:
        Node_locations.append((event.xdata, event.ydata, Node_type))

cid = fig.canvas.mpl_connect('button_press_event', onclick)
plt.show()
plt.close()

#repeats the input and plotting section for the next node type
ax = plt.gca()
fig = plt.gcf()
implot = ax.imshow(im, origin = 'lower')
Node_type = input('type: work, home or intersection')
if Node_type != 'home' and Node_type != 'work' and Node_type != 'intersection':
    print('invalid node type: choose work home or intersection')

def onclick(event):
    if event.xdata != None and event.ydata != None:
        Node_locations.append((event.xdata, event.ydata, Node_type))

cid = fig.canvas.mpl_connect('button_press_event', onclick)

plt.show()

#Scales all the coordinates to fit within a certain sized graph
side_length = 120

max_x = Node_locations[0][0]
max_y = Node_locations[0][0]
for each in Node_locations:
    if each[0] > max_x:
        max_x = each[0]
    if each[1] > max_y:
```

```
        max_y = each[1]

if max_x > max_y:
    scale_factor = side_length / max_x
else:
    scale_factor = side_length / max_y

New_Node_locations =[]

for each in Node_locations:
    new_x = each[0] * scale_factor
    new_y = each[1] * scale_factor
    New_Node_locations.append ((new_x, new_y, each[2]))

Workplaces = []
Houses = []
Intersections = []

#Adds all the nodes to a new network
My_graph = nx.Graph()
id_num = 0
for each in New_Node_locations:
    id_num = id_num +1
    My_graph.add_node(id_num, btype = each[2], x = each[0], y = each[1])

for each in My_graph.nodes():
    if My_graph.node[each]['btype']=='work':
        Workplaces.append(each)
    elif My_graph.node[each]['btype']=='home':
        Houses.append(each)
    elif My_graph.node[each]['btype']=='intersection':
        Intersections.append(each)

def graph_maps(Map):
    plt.clf()
    work_xs = list(Map.node[n]['x'] for n in Workplaces)
    work_ys = list(Map.node[n]['y'] for n in Workplaces)

    home_xs = list(Map.node[n]['x'] for n in Houses)
    home_ys = list(Map.node[n]['y'] for n in Houses)

    inter_xs = list(Map.node[n]['x'] for n in Intersections)
    inter_ys = list(Map.node[n]['y'] for n in Intersections)

    edge_points = []
    for one in Map.edges():
        node1 = one[0]
        node2 = one[1]
        node1x = Map.node[node1]['x']
        node1y = Map.node[node1]['y']
        node2x = Map.node[node2]['x']
        node2y = Map.node[node2]['y']
        plt.plot((node1x, node2x), (node1y,node2y), 'b')

    plt.plot(work_xs, work_ys, 'ro', home_xs, home_ys , 'go', inter_xs, inter_ys, 'yo' )
    plt.ylabel(Map)
    plt.axis([ 0,side_length +2, 0,  side_length+2])

    plt.show()
My_graph.graph['name'] = 'Custom Node map'
graph_maps(My_graph)
#pickles the graph to be used in the other Python program
nx.write_gpickle(My_graph, image_name + "_nodemap.gpickle")
```

```
#writes the node locations to file for use in Matlab code
f = open('brown_file.txt','w')
for item in New_Node_locations:
    f.write(str(item) + ',')

f.close()
```

A.2 Genetic Algorithm

```
#!/usr/bin/env python
##############################
##Libraries used in program###
##############################
import networkx as nx
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import numpy as np
import random, math
from datetime import datetime
startTime= datetime.now()
from scipy.interpolate import interp1d

##############################
##Funcrions used in program###
##############################

#Function used to make sorted lists of graphs based on their path value
def bubbleSort(numbers):
    changes = 1
    while changes > 0:
        changes = 0
        for n in range(len(numbers) - 1):
            if numbers[n].graph['path'] > numbers[n+1].graph['path']:
                #Swaps the numbers if the first is larger
                numbers[n], numbers[n+1] = numbers[n+1], numbers[n]
                #Keeps track of whether or not the list was changed
                changes = changes + 1

    return(numbers)

#Makes a new edge between two nodes
def new_edge(graph, start_node, end_node):
    u = start_node
    v = end_node
    x1 = graph.node[u]['x']
    x2 = graph.node[v]['x']
    y1 = graph.node[u]['y']
    y2 = graph.node[v]['y']
    node_weight = random.randint(1,max_road_capacity)
    distance_between_nodes = math.sqrt(abs(((x1-x2)**2)- ((y1-y2)**2) ))
    graph.add_edge(u,v, distance = distance_between_nodes, weight = node_weight )

#checks if there will be an intersection if an edge is added between two nodes
def is_intersection(graph,u, v):
    if u != v:
        if graph.node[u]['x'] < graph.node[v]['x']:
            new_node_xs =  (graph.node[u]['x'],graph.node[v]['x'] )
            new_node_ys = (graph.node[u]['y'],graph.node[v]['y'] )
```

```
        else:
            new_node_xs =  (graph.node[v]['x'],graph.node[u]['x'] )
            new_node_ys = (graph.node[v]['y'], graph.node[u]['y'] )
        new_function = interp1d(new_node_xs, new_node_ys)

        if len(graph.edges()) > 0:
            possible_edges = graph.edges()
            for each in graph.edges():
                if each[0] == v or each[1] == v or each[0] == u or each[1] == u:
                    possible_edges.remove(each)
            for each_edge in possible_edges:
                if graph.node[each_edge[0]]['x'] < graph.node[each_edge[1]]['x']:
                    xs = (graph.node[each_edge[0]]['x'] , graph.node[each_edge[1]]['x'])
                    ys = (graph.node[each_edge[0]]['y'] , graph.node[each_edge[1]]['y'])
                else:
                    xs = ( graph.node[each_edge[1]]['x'], graph.node[each_edge[0]]['x'] )
                    ys = ( graph.node[each_edge[1]]['y'], graph.node[each_edge[0]]['y'] )

                function = interp1d(xs,ys)
                new_range = list(np.arange(new_node_xs[0], new_node_xs[1],0.1))
                old_range = list(np.arange(xs[0], xs[1], 0.1))
                test_range = []
                if len(old_range) > 2 and len(new_range) > 2:
                    new_range.pop(0)
                    new_range.pop(-1)
                    old_range.pop(0)
                    old_range.pop(-1)
                for each in old_range:
                    for every in new_range:
                        if abs(each- every) < 0.1:
                            test_range.append((every + each) / 2)

                test_range.sort()
                if len(test_range) > 0:
                    for n in test_range:
                        if abs(function(n) - new_function(n)) < 1 and (u,v) != each_edge:
                            return True
            return False
        else:
            new_edge(graph, u, v)
            print('new edge because too few', u, v)

#makes a new edge to a random node as long as it does not intersect
def edge_to_random_node(graph, node):
    V = random.randint(1, len(graph.nodes()))
    while node == V:
        V = random.randint(1, len(graph.nodes()))
    if is_intersection(graph, node, V) == False:
        new_edge(graph, node, V)
        print('edge added')
    else:
        print('''can't add edge''')

#Caluclated the amount of time a given path takes to travel
def find_path_Time(List, item):
    time_per_person = 0
    if len(List) > 0:
        for x in range(0, len(List) - 1):
            #print(List[x],List[x+1])
            time_per_link = (item.edge[List[x]][List[x+1]]['distance']/item.edge[List[x]][List[x+1]]['weight'])
            time_per_person += time_per_link

        return time_per_person
```

```
#Finds the cost of a whole graph
def calcCost(a_graph):
    total_cost = 0
    for road in a_graph.edges():
        cost = a_graph.edge[road[0]][road[1]]['weight']* a_graph.edge[road[0]][road[1]]['distance']

        total_cost = total_cost + cost
    return total_cost


def Reproduce(parent1, parent2, budget):
    child_graph = parent1.copy()
    recombination_point = random.choice(parent1.nodes())
    edges_to_delete = child_graph.edges(recombination_point)
    child_graph.remove_edges_from(edges_to_delete)
    edges_to_add = parent2.edges(recombination_point)
    for each in edges_to_add:
         if is_intersection(child_graph,each[0],each[1]) != True:
            new_edge(child_graph, each[0], each[1])
    while nx.is_connected(child_graph) == False:
        for i in range(1,len(parent1.nodes())+1):
            if i not in nx.connected_components(child_graph)[0]:
                edge_to_random_node(child_graph,i)

    while calcCost(child_graph) > budget:
        remove = child_graph.edges()[random.randint(0,len(child_graph.edges())-1)]
        child_graph.remove_edge(*remove)
        while nx.is_connected(child_graph) == False:
            print('not connected')
            for i in range(1,len(parent1.nodes())+1):
                if i not in nx.connected_components(child_graph)[0]:
                    edge_to_random_node(child_graph,i)
    return child_graph


def graph_maps(Map):
    plt.clf()
    work_xs = list(Map.node[n]['x'] for n in Workplaces)
    work_ys = list(Map.node[n]['y'] for n in Workplaces)

    home_xs = list(Map.node[n]['x'] for n in Houses)
    home_ys = list(Map.node[n]['y'] for n in Houses)

    inter_xs = list(Map.node[n]['x'] for n in Intersections)
    inter_ys = list(Map.node[n]['y'] for n in Intersections)

    edge_points = []
    for one in Map.edges():
        node1 = one[0]
        node2 = one[1]
        node1x = Map.node[node1]['x']
        node1y = Map.node[node1]['y']
        node2x = Map.node[node2]['x']
        node2y = Map.node[node2]['y']
        plt.plot((node1x, node2x), (node1y,node2y), 'b', lw = Map.edge[node1][node2]['weight'])

    plt.plot(work_xs, work_ys, 'ro', home_xs, home_ys, 'go', inter_xs, inter_ys, 'yo' )
    plt.ylabel(Map)
    plt.axis([ 0,122, 0,  122])
    plt.savefig(Map.graph['name']  + '.png')
```

```
############################
####Constants for the city###
############################
import math

num_intersections = 0
id_num = 0
radius_city = 20
radius_home = 60
Budget = 4000
max_road_capacity = 5

#######################################
##Creates the city map without roads###
#######################################
#loads a pre-made network
City = nx.read_gpickle('brown_campus_map_nodemap.gpickle')
#City = nx.read_gpickle('nodemap.gpickle')

#scales the locations of the nodes to fit a certain graph size
side_length = 120

max_x = City.node[1]['x']
max_y = City.node[1]['y']
for each in City.nodes():
    if City.node[each]['x'] > max_x:
        max_x = City.node[each]['x']
    if City.node[each]['y'] >  max_y:
        max_y = City.node[each]['y']

if max_x > max_y:
    scale_factor = side_length / max_x
else:
    scale_factor = side_length / max_y

for each in City.nodes():
    new_x = City.node[each]['x'] * scale_factor
    new_y = City.node[each]['y'] * scale_factor
    City.node[each]['x'] = new_x
    City.node[each]['y'] = new_y

Houses = [] #a list of all nodes that are houses
Workplaces = []#a list of all nodes that are workplaces
Intersections = []

for each in City.nodes():
    if City.node[each]['btype']=='work':
        Workplaces.append(each)
    elif City.node[each]['btype']=='home':
        Houses.append(each)
    elif City.node[each]['btype']=='intersection':
        Intersections.append(each)

graph_maps(City)

print('made node graph')

#######################################
##Initializes first round of duplicates
#######################################

#Duplicates of node graph
```

```
pool_size = 20
number = 0
Duplicates = []

for individual in range(pool_size):

    individual = City.copy()
    individual.graph['name'] ='graph' + str(number)
    Duplicates.append(individual)
    number = number + 1

#Gives each duplicate a different road system

for graph in Duplicates:
    print(graph)
    U = random.randint(1, len(graph.nodes()))
    V = random.randint(1, len(graph.nodes()))
    while U == V:
        V = random.randint(1, len(graph.nodes()))
    new_edge(graph, U, V)

    while nx.is_connected(graph) == False:
        for each_node in graph.nodes():
            if each_node not in nx.connected_components(graph):
                V = random.randint(1, len(graph.nodes()))
                while each_node == V:
                    V = random.randint(1, len(graph.nodes()))
                if is_intersection(graph, each_node, V) == False:
                    new_edge(graph, each_node, V)

        if nx.is_connected(graph) == False:
            graph.remove_edge(*random.choice(graph.edges()))
    while calcCost(graph) > Budget:
        remove = random.choice(graph.edges())
        graph.remove_edge(*remove)
        while nx.is_connected(graph) == False:
            print('not connected')
            for i in graph.nodes():
                if i not in nx.connected_components(graph)[0]:
                    edge_to_random_node(graph,i)
    graph_maps(graph)

    print ('made', graph)
    print('costs', calcCost(graph))
    if nx.is_connected(graph) == False:
        print('not connected')


##############################
##   Main Loop  #############
##############################
All_graphs =[]
Best_graphs = []
generations = 30

for x in range(generations): #number of iterations through generations
    for each_graph in Duplicates: #each graph in population
        path_time_Total = 0 #overall travel time in graph
        for n in Workplaces: #for each of the workplaces in the graph
            num_workers = 30
            path_time_Building = 0#travel time from each building
            for person in range(num_workers):#for each worker in the building
                target = random.choice(Houses)#a random home
```

```
                Path_home = nx.shortest_path(each_graph, n, target, 'distance')#shortest path home
                path_time_Person = find_path_Time(Path_home, each_graph)#time to travel shortest path home
                path_time_Building = path_time_Building + path_time_Person#time of all people in building
            path_time_Total = path_time_Total + (path_time_Building/num_workers)#sum of building times
        average_path = path_time_Total/len(Workplaces)

        each_graph.graph['path'] = average_path#assigns a path time value to the network
        All_graphs.append(each_graph)
    Ordered_graphs = bubbleSort(Duplicates)#Ranks the graphs
    Best_graphs.append(Ordered_graphs[0])
    #print('did fitness function')

#Selects the top layouts
#Mutates the layouts
    Duplicates = []
    top_group_number = int (0.3*pool_size)#the number of graphs selected to reproduce
    if top_group_number < 2:
        top_group_number = 2
    while len(Duplicates) < pool_size:
        random_parent1 = Ordered_graphs[ random.randint(0,top_group_number)]
        random_parent2 = Ordered_graphs[ random.randint(0,top_group_number)]
        new_graph = Reproduce(random_parent1,random_parent2, Budget)
        new_graph.graph['name'] ='graph' + str(number)
        Duplicates.append(new_graph)
        graph_maps(new_graph)
        number = number + 1
    total_weight = 0
    for each in Ordered_graphs[0].edges():
        total_weight = total_weight + Ordered_graphs[0].edge[each[0]][each[1]]['weight']
    average_weight = total_weight / len(Ordered_graphs[0].edges())

    print (Ordered_graphs[0].graph['path'], len(Ordered_graphs[0].edges()), average_weight)

#Keeps track of the runtime of the program
time_taken = (datetime.now()-startTime)
print(time_taken)
################################
##Visualization#################
################################

def graphSolutions(best_list):
    plt.clf()
    Pointx =[]
    for n in range(len(best_list)):
        Pointx.append(n)
    Pointy =[]
    for n in (best_list):
        Pointy.append(n.graph['path'])
    Solutions = plt.plot(Pointx,Pointy, 'bo')
    plt.show()

graphSolutions(Best_graphs)
```

B Stochastic Hill-Climbing Algorithm (Matlab Code)

```
function RoadNetwork_basic(runs,filename)

tic

% Read in node coordinates, if given in input

if nargin == 2
file = textread(filename,'\%s','delimiter',','),(',');
data = zeros(length(file),1);

for c = 1:length(file)
    if mod(c,3) == 0
        if isempty(setdiff(double(file{c}),(double(file{end})))) == 0
            data(c) = 1; % 1 means 'home', 0 means 'work'
        end %if
        else data(c) = str2num(char(file(c)));
            data(c) = data(c)-60;
    end %if
end %for

numnodes = length(data)/3;

else numnodes = 50;

end %if

intersect_runs = [];
edges_runs = [];
time_runs = [];

% Create runs number of node networks

for g = 1:runs

% Initialize variables

nodes = zeros(numnodes,1);
nodes_poss = zeros(length(nodes),length(nodes));
diff = floor(rand(1,1)*5);
edges = zeros(2*length(nodes)-3-diff,2);

%numedges = size(edges(:,1));

for ed = 1:length(edges(:,1))
    capacity(ed) = ceil(normrnd(2.5,2.5));
end %for

xboundary = 60;
yboundary = 60;
rad = 60;

intersect_time = 5;
intersect_cost = 5;

% city is circular with limits at radius = sqrt(xboundary*yboundary)

% Randomly place nodes
```

```
if nargin == 1

    for n = 1:length(nodes)
        posneg = randperm(4);
        a(n) = rand(1,1)*xboundary;
        b(n) = rand(1,1)*yboundary;
            if posneg(1) == 2 | posneg(1) == 3
                a(n) = -a(n);
            end %if
            if posneg(1) == 3 | posneg(1) == 4
                b(n) = -b(n);
            end %if
        nodes(n) = a(n) + b(n)*i;
        destination_theta(n) = arg(nodes(n));
        destination_rho(n) = sqrt(a(n)^2+b(n)^2);
    end %for

else

    for n = 1:length(data)/3
        a(n) = data((n-1)*3 + 1);
        b(n) = data((n-1)*3 + 2);
        nodes(n) = a(n) + b(n)*i;
        destination_theta(n) = arg(nodes(n));
        destination_rho(n) = sqrt(a(n)^2+b(n)^2);
    end %for
end %if

all_destination_theta(g,1:n) = destination_theta;
all_destination_rho(g,1:n) = destination_rho;

figure(g);
clf;
shg;
polar(destination_theta,destination_rho,'.r')
axis off

% Place edges randomly between nodes

budget = 0;
newedges = edges;

for n = 1:numnodes
    for m = 1:numnodes
        if m == n
            continue
        end %if
        dist(m) = sqrt((a(m)-a(n))^2 + (b(m)-b(n))^2);
        if dist(m) < rad*(1/2)
            nodes_poss(n,m) = nodes(m);
        end %if
    end %for
    ind = find(nodes_poss(n,:));
    if isempty(ind)
        continue
    end %if
    node_chosen = randperm(length(ind));

% Edges are not constructed where they already exist

    for ed = 1:length(edges(:,1))
        if edges(ed,1) == nodes(ind(node_chosen)) & edges(ed,2) == nodes(n)
```

```
            node_chosen = rand_nodes_chosen(2);
        end %if
    end %for
    edges(n,1) = n;
    edges(n,2) = ind(node_chosen(1));
    edges_distance(n) = abs(nodes(edges(n,1))-nodes(edges(n,2)));
    edgespropcap(n) = edges_distance(n)*capacity(n);
    line([a(n),a(ind(node_chosen(1)))],[b(n),b(ind(node_chosen(1)))],'linewidth',capacity(n)/3);

% Make planar

    if n > 1
        for ed = 1:n-1
            try
            [x0,y0] = intersections([real(nodes(edges(n,1))),real(nodes(edges(n,2)))],[imag(nodes(edges(n,1))),
             imag(nodes(edges(n,2)))],[real(nodes(edges(ed,1))),real(nodes(edges(ed,2)))],  \\
             [imag(nodes(edges(ed,1))),imag(nodes(edges(ed,2)))],'true');
            does_intersect1 = intersect(arg(x0+y0*i),destination_theta);
            does_intersect2 = intersect(abs(x0+y0*i),destination_rho);
        if (isempty(x0) == 0) && (isempty(y0) == 0) && (isempty(does_intersect1 == 1)) && \\
         (isempty(does_intersect2 == 1))
            nodes(length(nodes)+1) = x0 + y0*i;
            newedges(length(newedges(:,1))+1:length(newedges(:,1))+4,:) = [edges(n,1),length(nodes); \\
            length(nodes),edges(n,2);edges(ed,1),length(nodes);length(nodes),edges(ed,2)];
           capacity(length(capacity)+1:length(capacity)+4) = [capacity(n),capacity(n),capacity(ed),capacity(ed)]
               if edges(ed,:) == newedges(ed,:)
                     newedges(ed,:) = [];
                     capacity(ed) = [];
               end %if
               if edges(n,:) == newedges(n,:)
                     newedges(n,:) = [];
                     capacity(n) = [];
               end %if
        end %if
            catch continue
            end % try&catch
        end %for
    end %if
        budget = intersect_cost*(length(nodes)-numnodes) + sum(edgespropcap);
end %for

% Reiterate over nodes to place excess edges between them (w/ budget constraint based on number of nodes)

while budget < 4000

for n = 1:length(edges(:,1))-numnodes
        for m = 1:numnodes
            if m == n
                continue
            end %if
            dist(m) = sqrt((a(m)-a(n))^2 + (b(m)-b(n))^2);
            if dist(m) < rad*(1/2)
                nodes_poss(n,m) = nodes(m);
            end %if
        end %for
        ind = find(nodes_poss(n,:));
        if isempty(ind)
            continue
        end %if
        rand_nodes_chosen = randperm(length(ind));
        node_chosen = rand_nodes_chosen(1);
% Edges are not constructed where they already exist
        for ed = 1:length(edges(:,1))
```

```
            if (edges(ed,1) == nodes(ind(node_chosen))) && (edges(ed,2) == nodes(n))
                node_chosen = rand_nodes_chosen(2);
            end %if
        end %for
        edges(numnodes+n,1) = n;
        edges(numnodes+n,2) = ind(node_chosen);
        edges_distance(numnodes+n) = abs(nodes(edges(numnodes+n,1))-nodes(edges(numnodes+n,2)));
        edgespropcap(numnodes+n) = edges_distance(numnodes+n)*capacity(numnodes+n);
        line([a(n),a(ind(node_chosen))],[b(n),b(ind(node_chosen))],'linewidth',capacity(numnodes+n)/3);

% Make planar
            for ed = 1:numnodes+n-1
                try
        [x0,y0] = intersections([real(nodes(edges(numnodes+n,1))),real(nodes(edges(numnodes+n,2)))], \\
        [imag(nodes(edges(numnodes+n,1))),imag(nodes(edges(numnodes+n,2)))],[real(nodes(edges(ed,1))), \\
        real(nodes(edges(ed,2)))],[imag(nodes(edges(ed,1))),imag(nodes(edges(ed,2)))],'true');
            if (isempty(x0) == 0) && (isempty(y0) == 0)
                nodes(length(nodes)+1) = x0 + y0*i;
                newedges(length(newedges(:,1))+1:length(newedges(:,1))+4,:) = [edges(numnodes+n,1), \\
                length(nodes);length(nodes),edges(numnodes+n,2);edges(ed,1),length(nodes);length(nodes), \\
                edges(ed,2)];
               capacity(length(capacity)+1:length(capacity)+4) = [capacity(numnodes+n),capacity(numnodes+n), \\
               capacity(ed),capacity(ed)];
                    if edges(ed,:) == newedges(ed,:)
                        newedges(ed,:) = [];
                        capacity(ed) = [];
                    end %if
                    if edges(numnodes+n,:) == newedges(numnodes+n,:)
                        newedges(numnodes+n,:) = [];
                        capacity(numnodes+n) = [];
                    end %if
            end %if
            catch continue
            end % try&catch
            end %for
                budget = intersect_cost*(length(nodes)-numnodes) + sum(edgespropcap);
    end %for

end %while

%nodes = unique(nodes);
intersect_theta = [];
intersect_rho = [];

for ex = numnodes+1:length(nodes)
    intersect_theta(length(intersect_theta)+1) = arg(nodes(ex));
    intersect_rho(length(intersect_rho)+1) = sqrt((real(nodes(ex)))^2+(imag(nodes(ex)))^2);
end %for

all_intersect_theta(g,1:length(nodes)-numnodes) = intersect_theta;
all_intersect_rho(g,1:length(nodes)-numnodes) = intersect_rho;

hold on;

% If you want to plot the intersections

%polar(intersect_theta,intersect_rho,'.b')
%circle_ang = 0.01:0.01:2*pi;
%circle_rad = ones(length(circle_ang),1).*rad;
%polar(circle_ang,circle_rad)
%axis off

% Determine shortest path to neighbourhood for each node & number of nodes crossed to calculate time taken
```

```
% Add edges for intersections created and remove old edges

edges = [edges; newedges(length(edges(:,1))+1:length(newedges(:,1)),:)];

% Find weights for each pair of nodes for shortest path algorithm

weights = zeros(length(edges(:,1)),length(edges(:,1)));
edges(all(edges)==0) = [];
capacity(all(capacity==0)) = [];

for ed = 1:length(edges(:,1))
    weights(edges(ed,1),edges(ed,2)) = abs(nodes(edges(ed,1))-nodes(edges(ed,2)));
end %for


% Choose start nodes within a certain ring and end nodes out of that ring

start_nodes = [];
end_nodes = [];
routes = zeros(length(start_nodes),1);
maxrad = max(nodes);
pathnodes = cell(length(start_nodes),1);
numcars = [];

if nargin == 1

    for n = 1:numnodes
        if abs(nodes(n)) < maxrad*0.3
            start_nodes(length(start_nodes)+1) = n;
        else end_nodes(length(end_nodes)+1) = n;
        end %if
    end %for

else

    for c = 1:length(data)
        if mod(c,3) == 0
            if data(c) == 0
                start_nodes(length(start_nodes)+1) = c/3;
            else end_nodes(length(end_nodes)+1) = c/3;
            end %if
        end %if
    end %for

end %if

% For each start-node choose an end-node at random out of end-nodes not already chosen and calculate time taken

time_total = 0;

for sn = 1:length(start_nodes)
    end_nodes_chosen = randperm(length(end_nodes));
    [distance,route_st,J,route] = shortestPathDP(weights,start_nodes(sn),end_nodes(end_nodes_chosen(1)),100);

% Stop paths that equal infinity
        if distance == Inf;
            for n = 1:length(end_nodes)
            [distance,route_st,J,route] = shortestPathDP(weights,start_nodes(sn),end_nodes(n),100);
                if distance ~= Inf;
                    break
                end %if
            end %for n
```

```
        end %if
    route_length(sn) = distance;
    pathnodes(sn,1) = route_st;
    capacitydist_total(sn) = 0;
        for ed = 1:length(route_st)-1
            try
            route_edgedist(ed) = abs(nodes(route_st(ed))-nodes(route_st(ed+1)));
            [nodes_of_edges,same_ind,capacity_ind] = intersect([route_st(ed),route_st(ed+1)],edges,'rows');
            capacity_path(ed) = capacity(capacity_ind);
            capacitydist_total(sn) = capacitydist_total(sn) + capacity_path(ed)/route_edgedist(ed);
            catch continue
            end %try&catch
        end %for ed
    numcars(sn) = floor((rad-abs(nodes(start_nodes(sn))))*(length(nodes)/3));
    time_total = time_total + (route_length(sn)/capacitydist_total(sn) + length(pathnodes(sn,1)) \\
    *intersect_time)*numcars(sn);
end %for sn


time_avg = time_total/sum(numcars)

% Save the number of intersections, number of edges, and average time for one car


if (time_avg == Inf) || (time_avg == 5) || (isnan(time_avg));
runs = runs+1
continue
else intersect_runs(length(intersect_runs)+1) = length(nodes)-numnodes;
edges_runs(length(edges_runs)+1) = length(edges(:,1));
time_runs(length(time_runs)+1) = time_avg;
end %if

all_edges(g,1:length(edges(:,1)),1:2) = edges;

end %for g

keyboard

%figure(runs+1);
%clf;
%shg;

minplot = zeros(max(intersect_runs),max(edges_runs));

for g = 1:length(intersect_runs)
minplot(intersect_runs(g),edges_runs(g)) = time_runs(g);
end %for


figure(runs+2);
clf;
shg;

edges_runs_v = 1:max(edges_runs);
intersect_runs_v = 1:max(intersect_runs);
[y_smooth,x_smooth] = meshgrid(1:0.5:max(intersect_runs_v),1:0.5:max(edges_runs_v));
smoothsurf = interp2(edges_runs_v,intersect_runs_v,minplot,x_smooth,y_smooth);


%---------------------------------- Stochastic hill-climbing ------------------------

% Initialize variables
```

```
numpts = 10;
[smoothsurf_rows,smoothsurf_cols] = size(smoothsurf);

randxvals = randperm(smoothsurf_rows);
randyvals = randperm(smoothsurf_cols);
localmins = zeros(numpts,3);
absstartpt = zeros(numpts,3)

% Starting values

for i = 1:numpts

surround = zeros(9,2);
smoothsurf_surround = [];

    if (randxvals(i)+1 < smoothsurf_rows) && (randxvals(1)-1 > 0) && (randyvals(i)+1 < smoothsurf_cols) \\
        && (randyvals(i) - 1 > 0)
        randpt(i,1) = randxvals(i);
        randpt(i,2) = randyvals(i);
    else randpt(i,1) = smoothsurf_rows-ceil(randxvals(i)/2);
        randpt(i,2) = smoothsurf_cols-ceil(randyvals(i)/2);
    end %if
        startpt = [randpt(i,1),randpt(i,2)];
        if (startpt(1)+1 < max(intersect_runs)) && (startpt(1)-1 > min(intersect_runs)) && \\
         (startpt(2)-1 > min(edges_runs)) && (startpt(2)+1 < max(edges_runs)) % No boundaries
            absstartpt = [startpt(1),startpt(2),smoothsurf(startpt(1),startpt(2))];
            for xval = startpt(1)-1:startpt(1)+1
                for yval = startpt(2)-1:startpt(2)+1
                [minval,minindex] = min(surround(:,1));
                surround(minindex,:) = [xval,yval];
                end %for
        end %for
        else numpts = numpts+1;
        continue
        end %if
        for surpt = 1:9
        smoothsurf_surround(length(smoothsurf_surround)+1) = smoothsurf(surround(surpt,1),surround(surpt,2));
        end %for

% Descend surface

        while  (any(gt(smoothsurf(startpt(1),startpt(2)),smoothsurf_surround)) = 1)
            [r,c]=find(smoothsurf_surround==min(min(smoothsurf_surround)));
            startpt = [surround(c,1),surround(c,2)];
            surround = zeros(9,2);
            if (startpt(1)+1 < max(intersect_runs)) && (startpt(1)-1 > min(intersect_runs)) && \\
            (startpt(2)-1 > min(edges_runs)) && (startpt(2)+1 < max(edges_runs)) % No boundaries
                for xval = startpt(1)-1:startpt(1)+1
                    for yval = startpt(2)-1:startpt(2)+1
                    [minval,minindex] = min(surround(:,1));
                    surround(minindex,:) = [xval,yval];
                end %for
                    end %for
            else numpts = numpts+1
            continue
            end %if
            smoothsurf_surround_last = smoothsurf_surround;
            smoothsurf_surround = [];
            for surpt = 1:length(find(surround(:,1)))
                smoothsurf_surround(length(smoothsurf_surround)+1) = smoothsurf(surround(surpt,1), \\
                surround(surpt,2));
            end %for
            if smoothsurf_surround == smoothsurf_surround_last
```

```
                [absmin,absminindex] = min(smoothsurf_surround)
                xyvals = surround(absminindex,:);
                break
            end %if
        end %while
    localmins(i,1:3) = [xyvals(1),xyvals(2),smoothsurf(xyvals(1),xyvals(2))]
end %for i

keyboard

toc
```