For office use only
T1 _____
T2 _____
T3 _____
T4 _____

For office use only
F1 _____
F2 _____
F3 _____
F4 _____

# 2016

**19th Annual High School Mathematical Contest in Modeling (HiMCM) Summary Sheet**
(Please make this the first page of your electronic Solution Paper.)

**Team Control Number:** 7211
**Problem Chosen:** B

Please paste or type a summary of your results on this page. Please remember not to include the name of your school, advisor, or team members on this page.


## Summary

A small business, whose main profits are online brick-and-mortar sales, is looking to start its shipping operations. It wishes to provide the continental United States with one-day transit by ground shipping with the United Parcel Service. Our task was to find the optimal warehouse placement to cover the continental United States with one-day transit while minimizing the number of warehouses. After finding the optimal placement, we considered state taxes as well as the tax cuts from the addition of clothing to the company's inventory.

Using transit day maps from the UPS website, we created a genetic algorithm to optimize the coverage of the United States using a certain number of warehouses. We found that it took 32 warehouses to cover 100% of the continental United States with one-day transit. However, this was highly inefficient because it unnecessarily added warehouses in order to provide one-day transit to unpopulated areas like forest preserves. We found that with 23 warehouses, it was possible to cover 95.89% of the continental United States and provide one-day transit to 99.6% of the population, allowing us to cut down on nine warehouses while only losing one-day transit to 0.4% of the population.

We then altered our program to take state tax rates into consideration when optimizing the warehouse placement. The new program produced sets containing mostly ZIP codes corresponding to places with low tax rates. However, this greatly reduced the total one-day transit coverage produced by the set of ZIP codes.

November 13, 2016

Dear Esteemed Company President,

We have decided that your business needs to expand. Extending your market to the entire nation will help you surpass the competition, and, thus, we have decided to pursue this strategy. We will achieve this by placing warehouses across the contiguous United States. Many problems arise, such as building costs, taxes, which raise our prices and thus decrease our demand, and the lack of clothing tax cuts in certain states. After 36 hours of deep consideration, we have figured out the optimal business strategy. Using a computer program, we have considered area covered, number of warehouses, state taxes, and tax exemptions for apparel in order to find the optimal number of warehouses and locations.

When we use 32 warehouses, area coverage is 83.89% of continental U.S., and the average tax rate is 2.27%.The ZIP codes for the locations are as follows:
59223, 82922, 89701, 57002, 59313, 81321, 97010, 03570, 59001, 59831, 56208, 97010, 23004, 31772, 97101, 63828, 43512, 14009, 80020, 85135, 73052, 03046, 59223, 59058, 97710, 59214, 59641, 59435, 97828, 69135, 76634, 59701

However, we saw that while the tax rates were pleasantly low, the area covered was only 83.89% of the total landmass of the continental United States. An appealing option that disregards tax fitness is as follows: If 23 warehouses are built, area coverage will be 99.56% and the average tax rate will be 5.07% The ZIP codes are
49710, 44017, 42021, 30122, 87008, 83325, 58620, 68005, 27343, 77331, 12108, 85324, 76008, 54106, 71004, 98220, 93601, 66402, 57051, 59010, 33825, 80020, 57650

As you can see, the first solution gives more weight to tax while the second gives weight to area while using less warehouses. The second solution raises the scope of our shipping coverage, it also will decrease demand and sales because of the higher tax rate.

We have also included a formula that will tell you which option is better if you input the number of warehouses you plan to build.These are the most optimized solutions we came up with, and we hope they help.

Sincerely,
Team #7211

# Table of Contents

# 1 Introduction

In the modern business world, maximizing profits is the highest priority. Businesses, especially small ones, should try to save money whenever possible. Along with cutting wages, removing competition, and increasing advertising and production, efficiency of company sites can save money. For example, a cheap 50 ft. by 100 ft. warehouse costs $35,000, but when coupled with the costs of maintenance and wages, increasing the number of warehouses significantly increases costs.

Clearly, warehouses are expensive, making it necessary to place them in optimal locations. In this problem, we attempt to reduce the number of warehouses while shipping to the entirety of the continental US via one-day transit.

In order to solve the problem, we used transit-time maps from www.ups.com/maps since the problem specifies one-day transit ground shipping via the United Parcel Service. The Python programs we created retrieved maps as image files from ups.com and allowed us to overlay combinations of ZIP codes and analyze the pixels of the resulting images. The only thing left to do was choose the best combinations.

Since covering 100% of the continental United States is not cost efficient, we created a program that optimizes the one-day transit coverage for a given number of warehouses using genetic algorithms. This program allowed us to not only calculate the minimum number of warehouses needed for 100% coverage, but also find the best coverage given a variable number of warehouses. We in turn used the data we collected in order to find the optimal locations to build warehouses.

In order to save the most amount of money, state clothing and sales tax rates were also taken into consideration when we chose warehouse locations. We modified the original program to weigh both sales and clothing tax rates and find optimal warehouse locations.

# 2 Problem Restatement/Interpretation

Because a business is looking to expand its online operations, more warehouses should be built to provide shipping to the entirety of the continental United States. Our modeling team has access to The United Parcel Service's one-day transit maps for 1846 different ZIP codes in the United States as well as tax data for the 48 contiguous states.

Using this information, our goal is to find the optimal placement of warehouses in order to maximize coverage while minimizing the number of warehouses and lowering sales and clothing tax rates for our customers by placing warehouses in low-tax states.

# 3 Assumptions

**Assumption 1**: Maps provided by UPS are accurate.
*Justification*: The problem shows a map from the UPS, so we assumed that the UPS is a valid source. Additionally, the UPS is the standard for shipping in the US, and, thus, it is reasonable that our solution should be based on the UPS maps. Furthermore, any damage caused to the customers would be the responsibility of the UPS, not the business.

**Assumption 2**: There should be around 30 warehouses built.
*Justification*: We manually covered nearly 92.5% of the area of the continental United States with 1 day shipping by building 30 warehouses in the most logical places (See Appendix A for an image of this map).

**Assumption 3:** ZIP codes in the same county essentially have the same one-day transit coverage.
*Justification*: Places in the same county will likely share the same roads and routes. The difference in travel times to certain destinations is most likely measured in minutes and is insignificant in terms of transit-days. We confirmed this with the mapping function of ups.com; all the pairs of ZIP codes in the same county that we tried gave identical or nearly identical maps.

**Assumption 4:** Maximizing the one-day transit coverage will inherently prioritize populated areas.
*Justification*: One of the biggest factors for determining the range of the one-day transit coverage is access to roads, especially highways. Obviously, access is more readily available in large cities, which means one-day transit range will be higher in large cities. Our program inherently prioritizes populated areas since our program prioritizes ZIP codes with high one-day transit ranges.

**Assumption 5**: Uncovered areas will be less populated areas.
*Justification*: Unlike large cities, less populated areas have less access to roads and highways. Thus, it is safe to assume that if an area has little coverage, there is probably a small population there.

**Assumption 6:** Covering 100% of the continental United States is possible but highly cost-inefficient.
*Justification*: Many areas have poor access to roads, which makes one-day transit difficult and inefficient. For example, in order to get access to one-day transit inside the Pfeiffer Big Sur State Park in California, we would need to build a new warehouse just to service one small area. Clearly, this money could be spent somewhere else. Additionally, according to assumption 5, these hard to reach areas tend to have small populations, and, thus, it would also be inefficient to build warehouses solely to service an extremely small percent of the population. Instead of covering the continental US with one-day transit, it would be more efficient to service 99.95% of the population.

**Assumption 7:** Optimization should be based on the population coverage, not the land area coverage.
*Justification*: We are trying to maximize profits for this business, and simply increasing the amount of land we cover does not necessarily guarantee that we will increase profits.

**Assumption 8:** Half of all sales are apparel.
*Justification:* According to www.statista.com, athletic apparel sales are 32.78 billion dollars per year. The recreation industry as a whole sells 63.65 billion dollars per year. Therefore, the apparel probably makes up around half of the sales for our company. Therefore, we can say that not paying clothing tax is the same as paying only half of our total sales tax regardless of item type.

**Assumption 9:** All people have the same likeliness to buy the company's product
*Justification*: Since we are not given an inventory of what the company is selling, we have no way of knowing who our product will appeal to. Our best option and most logical option is to assume that all people have the same likeliness to buy the company's product.

**Assumption 10:** Companies will basically pay their own taxes
*Justification*: The company must keep the same competitive price including tax, since customers can easily see tax when purchasing goods online, so if there is more tax, the company will lower the price and will basically be paying their own tax.

# 4 Model

## 4.1 Goals of Model:

Find three sets of warehouse locations so that we can:

- Cover at least 99.95% of the area of the contiguous United States using less than 33 warehouses with 1 day shipping
- Select optimal placement for the warehouses in order to minimize sales taxes while still serving at least 90% of the continental United States.
- Select optimal placement for the warehouses in order to minimize clothing taxes while still serving at least 90% of the continental United States.

## 4.2 Summary of Program

*First Program for Estimation:*

We created a program in Python that sent HTTP POST requests to the UPS to find the area a warehouse could serve with one-day transit. Using this and by logically inputting ZIP codes, we estimated that there would need to be around 30 warehouses to achieve over 90% coverage.

*Second Program for Optimization:*

First we gave a program a list of images of the one-day transit zone from 1846 different counties across the United States. We then created a genetic algorithm to select the best warehouse locations to offer one-day transit coverage to as much of the country as possible for a given number of warehouses. Knowing that a coverage of 90% or more would require around 30 warehouses from the first estimation, we inputted values around 30 into our program.

## 4.3 Influences of Program

### 4.3.1 Biological Evolution: Survival of the Fittest

In any given biological generation, only the organisms with the best adaptations, specifically adaptations that increase reproductive success or survivability, will survive. In each new generation, a random mutation is likely to occur, and if this mutation increases reproductive success or survivability, the organisms with such mutation will be more likely to survive, and, thus, this mutation will probably be added to the gene pool. If the mutations decrease

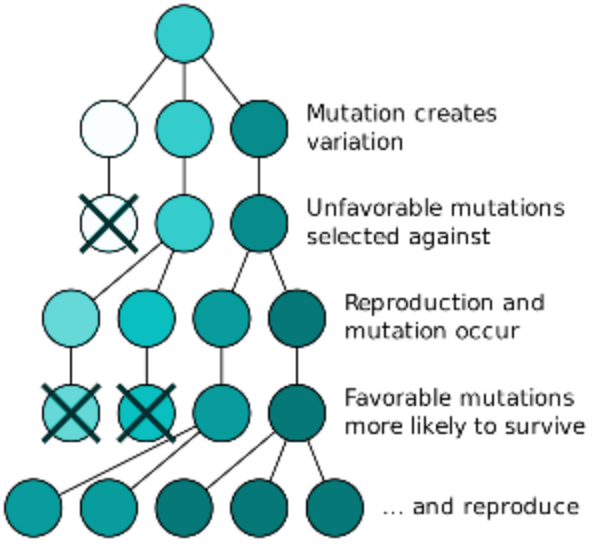


Figure 1. A basic diagram showing how evolution/adaptation works in biology, which is what our program is based on.

reproductive success or survivability, the organisms will die, and the mutations are removed from the gene pool.

### 4.3.2 Monte Carlo Method

The Monte Carlo method involves algorithmic calculations using random sampling to obtain tangible results. The Monte Carlo Method can be used to simulate the mutations in a genome when the genes are passed from one generation to the next.

## 4.4 Model Concept

We needed to find optimal coverage for each number of warehouses. It would be unfeasible to test every single combination of ZIP codes and find the best combination, so we instead created a program that first picked a somewhat optimal set of ZIP codes and continued to improve that set of ZIP codes.

We used HTTPS POST requests to take advantage of UPS's delivery map generator to download the one-day transit day range of every county in the continental United States. We then selected ten sets of of random ZIP codes (warehouses), and we calculated the total percentage of the area covered. This percentage number was referred to as the area fitness number. When sets encompassed a smaller area, or had low area fitness, we removed them from the list (specifically, the eight worst sets). To replace the eight missing sets, we duplicated the each of the desired top two sets four times. We then chose a random ZIP code from each of the eight duplicates and replaced it with a random ZIP code. This produced a new generation of ten sets. We repeated this process many times until a desirable result was achieved. We repeated this procedure for different numbers of warehouses.

For part two, we needed to take into account taxes. In order to do this, we changed the way that we calculated fitness. We added another variable called tax that would consider the region that would have to pay taxes. We then added the tax variable to the original fitness variable and changed the weighting to create a rating for the organism called tax fitness. Once again, we iterated through this procedure many times, but this time around, we used the tax fitness, not the area fitness, to eliminate the worst organisms.

## 4.5 Variables

| | |
|---|---|
| $w$ | Number of warehouses used to generate map |
| $p_y$ | Number of yellow pixels on the map (pixels that are within the zone of one-day delivery) |
| $p_o$ | Number of pixels located elsewhere on the map besides the state where the |

| | |
|---|---|
| | involved zip code is located. |
| $p_T$ | A constant representing the total number of pixels on the appropriate regions of the map (equivalent to 91,291) |
| $t_M$ | A constant representing the maximum amount of sales tax in a state (equivalent to .075 from California) |
| $t_n$ | The sales tax rate of the state of the n-th zip code of a set |
| $f_1$ | Fitness value of a set of zip code locations for warehouses which represents the percentage of continental land mass that the delivery map fills. |
| $f_2$ | Fitness value of a set of zip code locations for warehouses which prioritizes sales tax and slightly takes land mass into consideration. |
| $f_2(n)$ | Tax fitness value of the n-th zip code of a set of zip code locations. Used only in Part II. |

## 4.6 Formulas

Our model for the total fitness when not taking tax into account follows as is:

$$f_1 = \frac{p_y}{p_T}$$

Taking state sales tax into consideration, our formula for fitness was split into two different parts: calculating a fitness for each individual zip code and using the average of those fitness values multiplied by the original spatial fitness to determine an overall fitness of the set of zip codes (derivations can be found in Appendix C):

$$f_2(n) = \frac{2t_M - t_n}{2t_M} \cdot \frac{p_o}{p_T}$$

$$f_2 = \frac{3}{4} \times \left( \frac{\sum_{n=1}^{w} f_2(n)}{w} \right) + \frac{1}{4} \times f_1$$

## 4.7 Maps and ZIP codes

We got a list of all ZIP codes in the United States from https://www.aggdata.com/node/86. We narrowed down this list from 40,000 ZIP codes to 1846 ZIP codes by only including one ZIP code from each county based on our assumption that zip codes in the same area deliver to the

same locations. We then downloaded an image for each ZIP code that would only show the amount of area covered by one-day shipping. Doing this cut down the time required to retrieve images from ups.com, making our genetic algorithm much more efficient.

## 4.8 Map Rendering

We created a map rendering program to visualize the area covered with one day transit by an inputted set of ZIP codes. The program went through each ZIP code that was inputted, retrieved the map from ups.com, and cut out the colors except for the one-day transit range. The program then compiled the maps corresponding to the inputted ZIP codes into a single map. In overlapped pixels, the amount of red was reduced, allowing us to visualize overlapped areas as green.

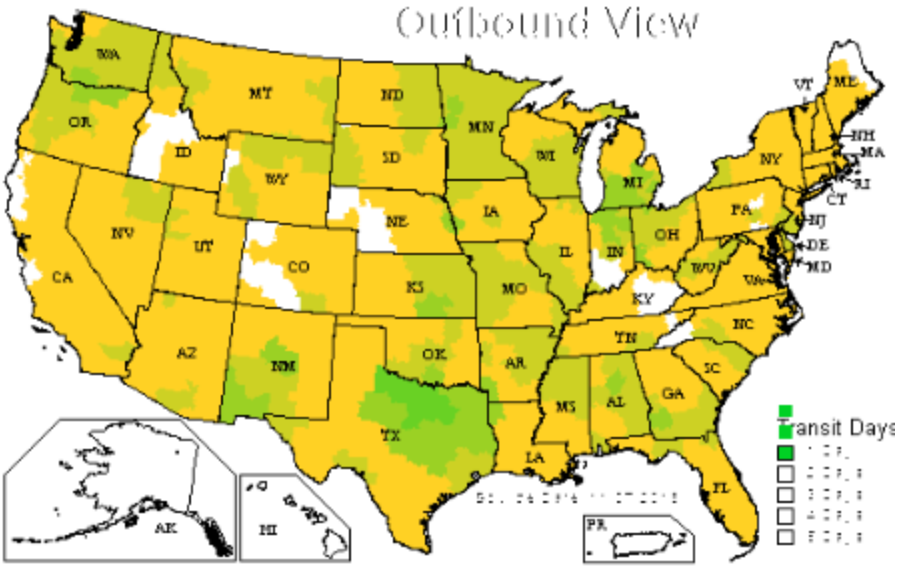


Figure 2. An example of a map generated by our map rendering program.

## 4.9 Genetic Algorithm

We developed the algorithm to generate warehouse locations as an analogy of the system of natural selection. Algorithms such as these are often referred to as "genetic algorithms." First, the program began with ten sets of $w$ random ZIP codes, with each set being similar to an organism that has traits (the ZIP codes). During each iteration, or "generation," it calculated the "fitness" value ($f_1$ or $f_2$) of each set of ZIP codes. Sorting the sets by their fitness levels from greatest to least, it removed the last eight and replaced them with four "mutations" of each of the top two. It "mutated" the top two by picking one zip code and replacing it with a random zip code. Once the last eight sets were replaced, it repeated the process. After a certain number of iterations, the process would stop, and it would output the most "fit" set from the most recent

generation as well as its fitness ranking. We recorded the set of zip codes and its corresponding fitness value in our data tables.

# 5 Model Data and Results

## 5.1 Part One - Resulting Model

Finding optimal coverage for multiple numbers of warehouses allows us to view many options for the business so that we can select the most efficient and effective number of warehouses to be built. Our genetic algorithm ran through 2000 generations to produce the optimal percent covered. Our results are as follows:

| Warehouses | Percent Covered |
|---|---|
| 15 | 81.6 |
| 16 | 86.27 |
| 17 | 88.78 |
| 18 | 89.38 |
| 19 | 92.32 |
| 20 | 92.74 |
| 21 | 93.93 |
| 22 | 94.69 |
| 23 | 95.89 |

| Warehouses | Percent Covered |
|---|---|
| 24 | 96.55 |
| 25 | 97.23 |
| 26 | 97.77 |
| 27 | 98.67 |
| 28 | 98.9 |
| 29 | 99.15 |
| 30 | 99.31 |
| 31 | 99.57 |
| 32 | 100 |

We graphed the data above, and it followed the cubic regression shown in Figure 3. The regression would be helpful when estimating best percentage covered for other numbers of warehouses.

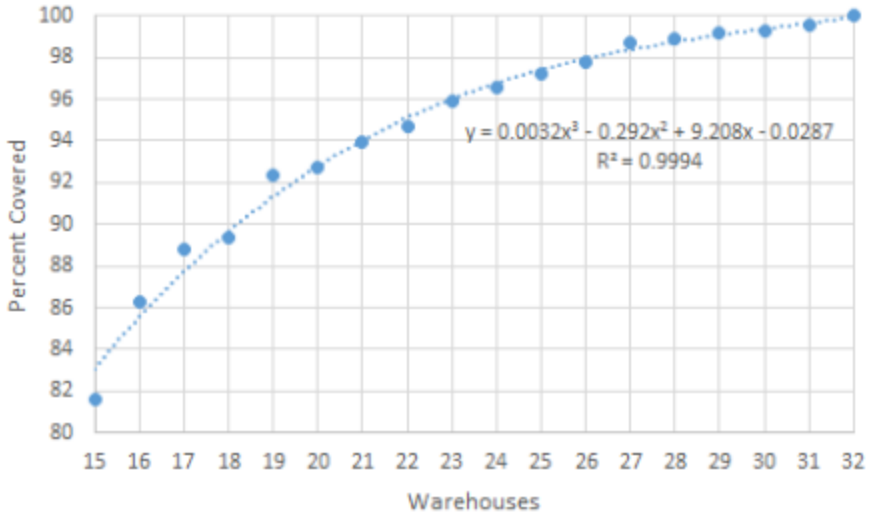


Figure 3. A plot of the number of warehouses and the percent of the continental United States covered with one-day transit. The percent covered by a certain number of warehouses can be estimated by the function below.

The cubic regression function is as follows:

$$f(x) = \begin{cases} 0.0032x^3 - 0.292x^2 + 9.208x - 0.0287 & 0 \leq x < 32 \\ 100 & x \geq 32 \end{cases}$$

According to our model, we would need at least 32 warehouses in order to service the entire continental United States with one-day transit. The following 32 ZIP codes completely fill the map:

30236, 56510, 93201, 42322, 67001, 50025, 83311, 49010, 01810, 97004, 89440, 26148, 27201, 78610, 97801, 18039, 38602, 85324, 80010, 82601, 95605, 79745, 54930, 33002, 69345, 59054, 65614, 87001, 57003, 70517, 43434, 83463

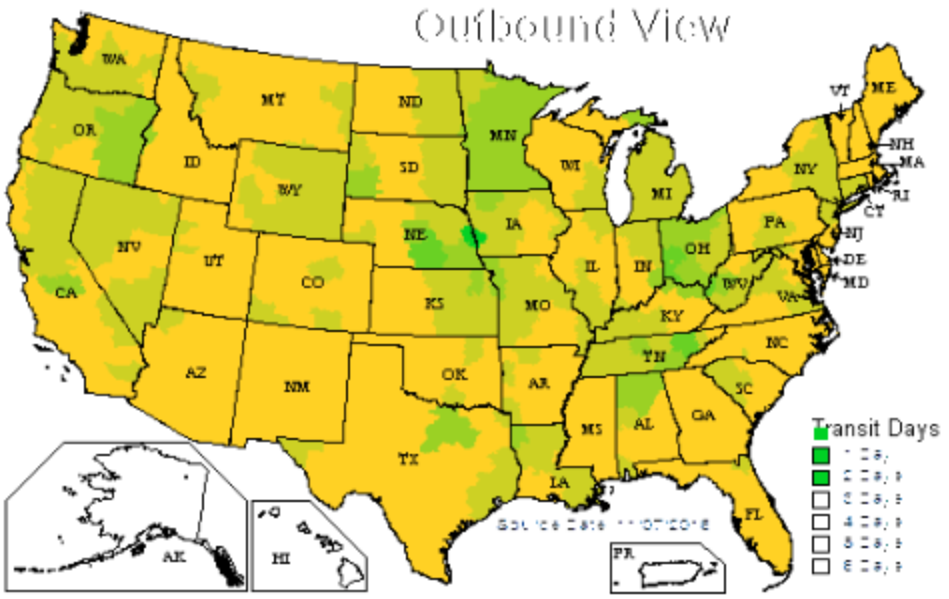


Figure 4. A map of the entire continental United States covered by one-day transit by 32 ZIP codes for warehouse locations.

## 5.2 Part Two

We then tweaked the model in order to find the optimal coverage while taking taxes into consideration. The taxes had 75 % weight while the area fitness had 25 % weight. The tax fitness data is shown below.

| Warehouses | Tax Fitness |
|---|---|
| 15 | 82.3853 |
| 16 | 81.0587 |
| 17 | 81.716 |
| 18 | 81.1486 |
| 19 | 82.2699 |
| 20 | 82.0647 |
| 21 | 83.4583 |
| 22 | 82.9763 |
| 23 | 82.8299 |

| Warehouses | Tax Fitness |
|---|---|
| 24 | 82.4549 |
| 25 | 84.5766 |
| 26 | 84.6549 |
| 27 | 85.1031 |
| 28 | 84.994 |
| 29 | 86.105 |
| 30 | 83.5205 |
| 31 | 84.953 |
| 32 | 85.2647 |

When the data above was graphed, it had a logarithmic regression with an $R^2$ value of 0.7356 (Figure 5).

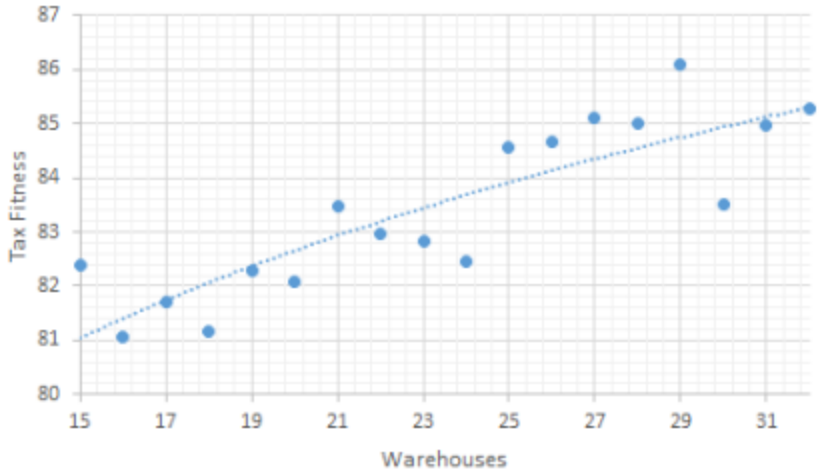


Figure 5. A graph of the tax fitness for each number of warehouses. The graph is estimated by a logarithmic function represented by the dotted line.

The formula for this regression is as follows:

$$f(x) = \begin{cases} 5.6214 \ln(x) + 65.815 & 0 < x < 438 \\ 100 & x \geq 438 \end{cases}$$

Since the tax fitness is somewhat arbitrary, we also calculated that average tax rates of the optimization for part 1 and part 2 (figure 6):

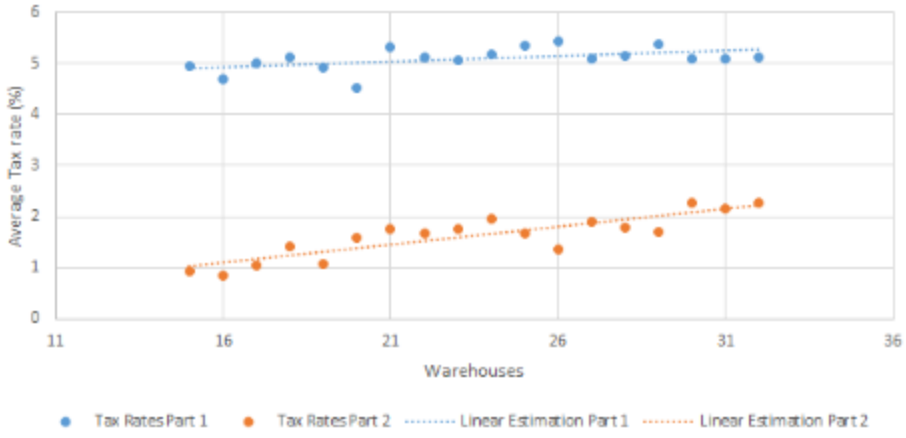


Figure 6. A graph of the average tax rates in the optimization for part 1 and 2.

The formulas for the linear regressions above for part 1 and part 2, respectively, are:

$$f(x) = 0.0209x + 4.5988$$
$$f(x) = 0.0701x - 0.0251$$

We also calculated the original fitness value using simply the area covered as a measure. Results are shown below:

| Warehouses | Area Fitness |
|---|---|
| 15 | 48.2479 |
| 16 | 41.4849 |
| 17 | 47.8054 |
| 18 | 52.8168 |
| 19 | 50.4058 |
| 20 | 60.2688 |
| 21 | 68.814 |
| 22 | 65.6144 |
| 23 | 66.4326 |

| Warehouses | Area Fitness |
|---|---|
| 24 | 69.2029 |
| 25 | 71.7946 |
| 26 | 65.6582 |
| 27 | 78.4864 |
| 28 | 75.6044 |
| 29 | 78.4612 |
| 30 | 79.4755 |
| 31 | 82.799 |
| 32 | 86.6088 |

The data was then used to calculate the area fitness, which was then graphed (Figure 7). The logarithmic regression had an $R^2$ value of 0.9262.

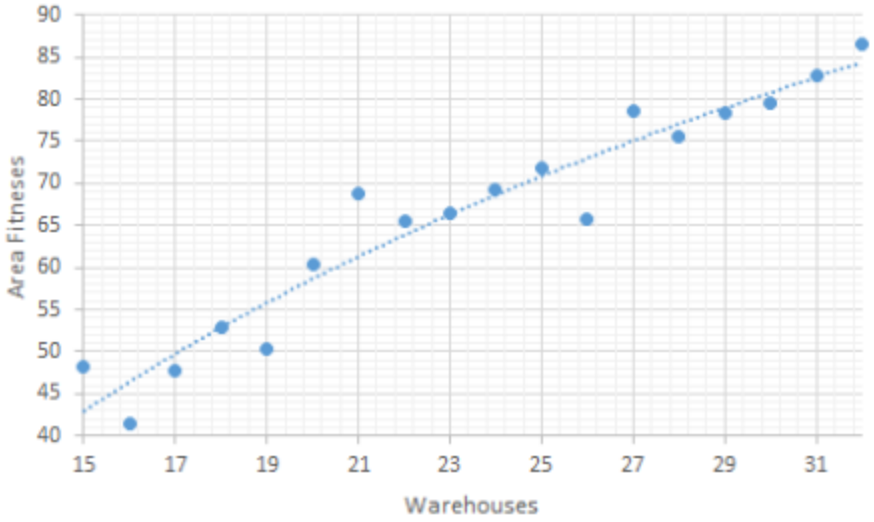


Figure 7. A graph of the area fitness (%) for the warehouses. The correlation can be estimated by dotted line that represents a logarithmic graph.

The equation for this logarithmic regression is as follows:

$$f(x) = \begin{cases} 54.664 \ln(x) - 105.07 & 0 < x < 38 \\ 100 & x \geq 38 \end{cases}$$

We then graphed both tax fitnesses on the same graph, and this resulted in the following figure:

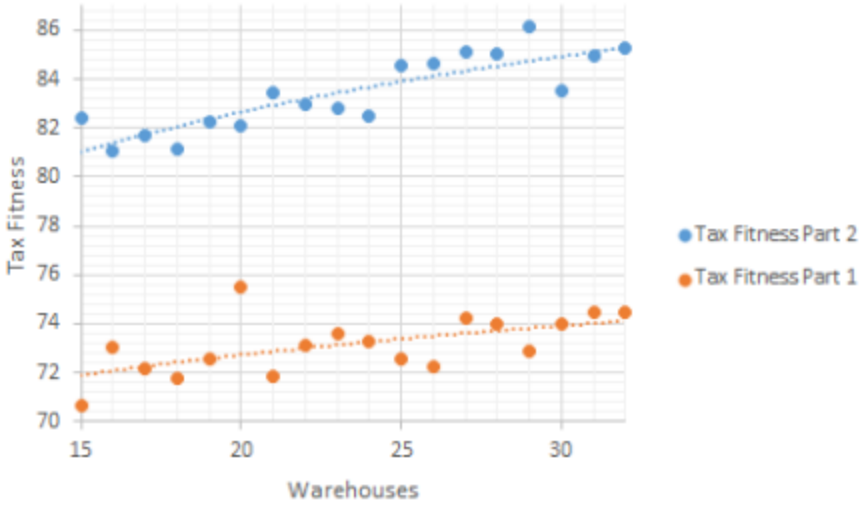


Figure 8. A graph comparing the general tax fitness (f2) of the optimization done in Model Part 1 and Model Part 2.

We then used a t-test to compare the population averages for the tax fitnesses from the two parts. Results are shown in Figure 9.

t-Test: Two-Sample Assuming Equal Variances

| | *Tax Fitness Part 2* | *Tax Fitness Part 1* |
|---|---|---|
| Mean | 83.41858014 | 73.13526551 |
| Variance | 2.38183003 | 1.401196365 |
| Observations | 18 | 18 |
| Pooled Variance | 1.891513198 | |
| Hypothesized Mean Difference | 0 | |
| df | 34 | |
| t Stat | 22.4310545 | |
| P(T<=t) one-tail | 2.94341E-22 | |
| t Critical one-tail | 1.690924255 | |
| P(T<=t) two-tail | 5.88683E-22 | |
| t Critical two-tail | 2.032244509 | |

Figure 9. This figure shows the t-test comparing the means the tax fitnesses of part 1 and part 2. The p-value is obviously very small.

## 5.3 Part Three

The introduction of clothing slightly changes things. Since only eleven states have limited or no clothing tax, we want to place warehouses in those states. Thus, we modified the program in part two to halve the weight of the states that little clothing tax.

The data when we ran the program is shown below:

| Warehouses | Tax Fitness |
|---|---|
| 15 | 81.94823 |
| 16 | 80.74726 |
| 17 | 82.23446 |
| 18 | 83.05837 |
| 19 | 82.06103 |
| 20 | 81.9032 |
| 21 | 82.80486 |
| 22 | 82.36359 |
| 23 | 84.48168 |

| Warehouses | Tax Fitness |
|---|---|
| 24 | 83.39286 |
| 25 | 83.89302 |
| 26 | 84.39762 |
| 27 | 83.91007 |
| 28 | 83.38867 |
| 29 | 84.64853 |
| 30 | 85.32594 |
| 31 | 85.76969 |
| 32 | 84.62019 |

We then graphed the data (Figure 10) and there was a logarithmic regression with an $R^2$ value of 0.7652.

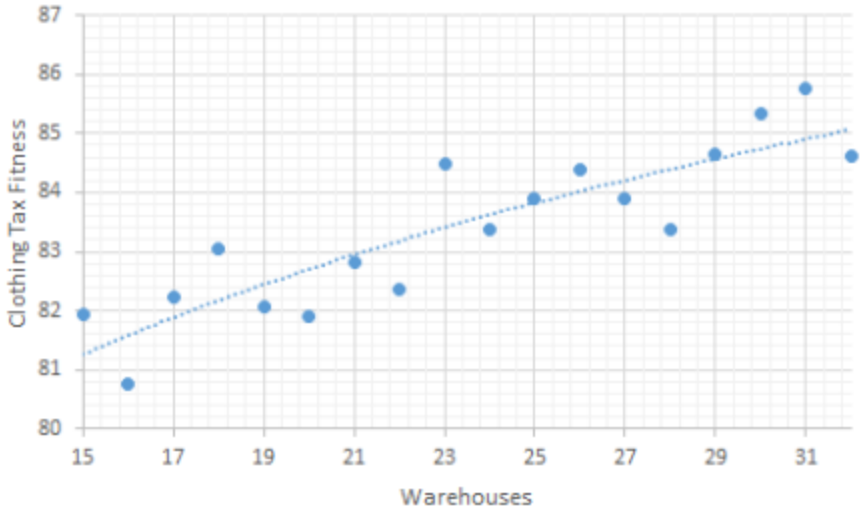


Figure 10. This is a graph of the warehouses vs tax fitness with apparel tax exemptions taken into consideration. It is a logarithmic function. The more warehouses, the better tax fitness.

The graph above had the regression formula of:

$$f(x) = 5.0264 \ln(x) + 67.646$$

We then graphed the data from Part 2 on the same graph as the data from Part 3 (Figure 11).

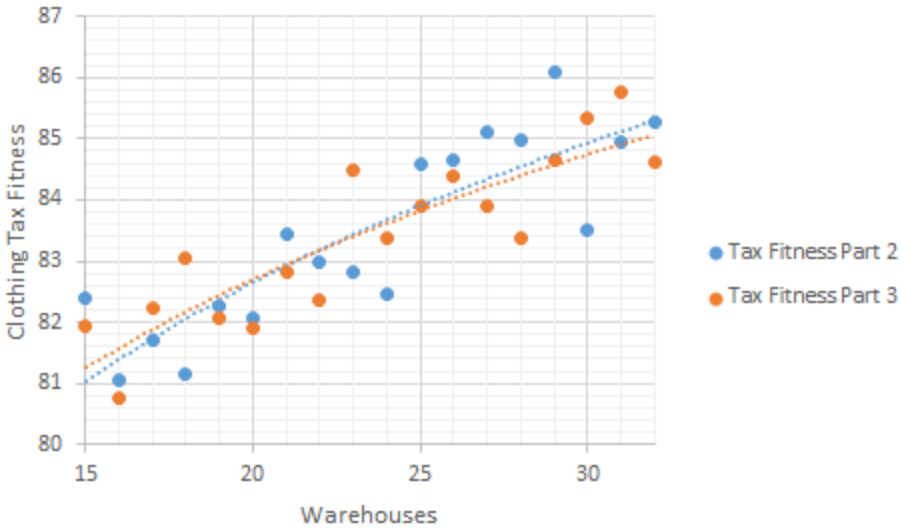


Figure 11. This is the graph of the Tax Fitnesses found in both part 2 and part 3. The dotted lines are the respective logarithmic regressions.

Finally, we ran a t-test (Figure 12) to see if there was a significant difference between the means.

t-Test: Two-Sample Assuming Equal Variances

| | *Tax Fitness Part 2* | *Tax Fitness Part 3* |
|---|---|---|
| Mean | 83.41858014 | 83.38607077 |
| Variance | 2.38183003 | 1.830529549 |
| Observations | 18 | 18 |
| Pooled Variance | 2.106179789 | |
| Hypothesized Mean Difference | 0 | |
| df | 34 | |
| t Stat | 0.067201984 | |
| P(T<=t) one-tail | 0.473407272 | |
| t Critical one-tail | 1.690924255 | |
| P(T<=t) two-tail | 0.946814544 | |
| t Critical two-tail | 2.032244509 | |

Figure 12. The results of the t-test between the Part 2 fitnesses and Part 3 fitnesses. The p-value seems to be rather large.

# 6 Discussion

## 6.1 Optimal placement considering only coverage

We can cover 100% of the continental United States using 32 warehouses, as mentioned in the results. However, this is highly inefficient. If we cover everything, we are providing one-day transit to forest preserves and other unpopulated areas. Furthermore, the addition of one warehouse had decreasing effects on the total area coverage as the total number of warehouses increased. This means that adding extra warehouses when the total number of warehouses is around 30 is cost-inefficient. We found that only 23 warehouses allow for 95.89% coverage of the United States (see Appendix B), and we can still provide one-day shipping to the vast majority of the population. The following 23 ZIP codes fill 95.89% of the continental United States:

49710, 44017, 42021, 30122, 87008, 83325, 58620, 68005, 27343, 77331, 12108, 85324, 76008, 54106, 71004, 98220, 93601, 66402, 57051, 59010, 33825, 80020, 57650

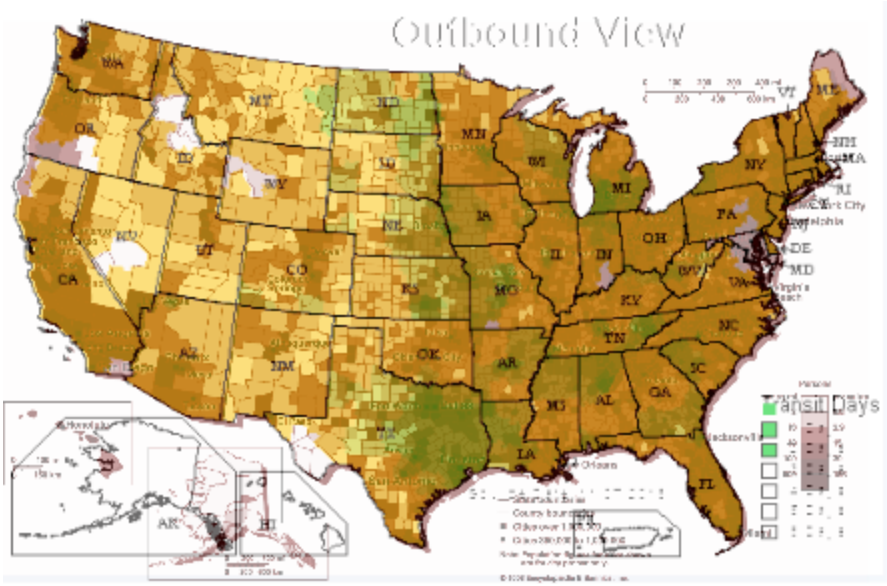


Figure 13. A semi-transparent map of optimal coverage with 23 warehouses placed over a population density map. The population density map was made darker so that the areas not covered by one-day transit (white and purple areas) could clearly be seen. See Appendix B for the original maps.

The continental United States is about 314 million square miles (http://www.comparea.org/). We know that 4.11 % is not covered by our model, which is 1.291 million square miles. In the uncovered areas, we estimated the population density to be 10 people per square mile. This means that one-day transit service would not be available to 1.291 million people, which is only 0.3997 % of the population of the continental United States.

Overall, while it takes a minimum of 32 warehouses to cover the entire continental United States with one-day transit, we would suggest that the company use 23 warehouses at the ZIP codes presented above to cover 99.6% of the continental United States population with one-day transit, which produces almost the same result as covering the entire continental United States with one-day transit in terms of business.

## 6.2 Optimal placement from part 1 vs part 2

Since the sets of ZIP codes form part 2 do much better in terms of average tax rates but cover much less of the United States with one-day transit, we must to compare the profits from using placement from part 1 and 2.

We can assume that the company will pay for its own sales tax (see assumption 10) and that all people will be equally likely to buy the company's products (see assumption 9). Assuming our profit per year is $q_1(w)$ for $w$ warehouses using warehouse distribution from part

1, we can calculate the profit using warehouse distribution from part 2, which we will call $q_2(w)$ in terms of $q$ ($f$ is the area fitness and $t$ is the average tax rate):

$$q_2(w) = q_1(w) * \frac{f_2(w) * \frac{100}{100+t_2(w)}}{f_1(w) * \frac{100}{100+t_1(w)}}$$

$$\frac{q_2(w)}{q_1(w)} = \frac{(54.664ln(w)-105.07) * \frac{100}{99.749+0.0701w}}{(0.0032w^3-0.292w^2+9.205w-0.0287) * \frac{100}{104.5988+0.0209w}}$$

The graph of this formula is as follows:

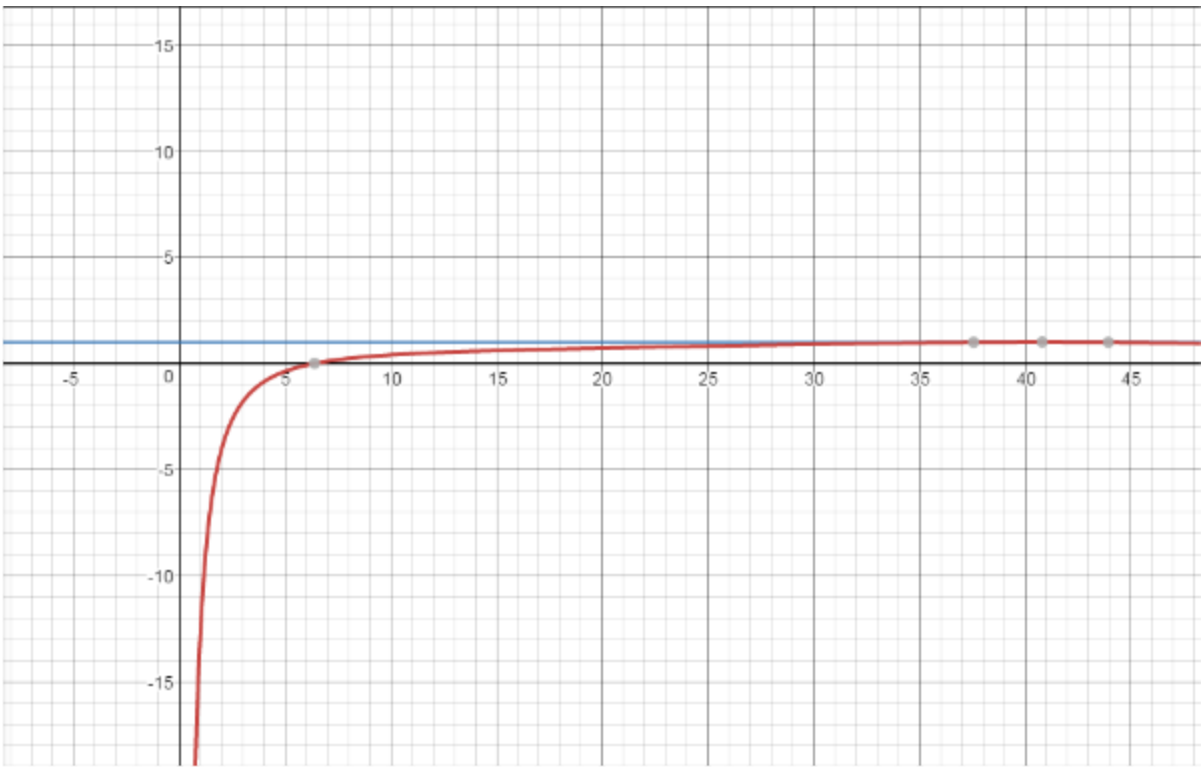


Figure 14. A graph of the the ratio of profits from the optimal warehouse distribution of $w$ warehouses from part 2 to part 1, represented by $y$. The number of warehouses is represented by $x$. The blue line is the line $y = 1$.

Note that as long as $y < 1$, $\frac{q_2(w)}{q_1(w)} < 1$, which means that $q_1(w) > q_2(w)$ for almost all values of $w$ except for $37.558 < w < 43.932$, or $38 \leq w \leq 43$ for integers $w$. Thus, for any number of warehouses except for $38 \leq w \leq 43$, using the warehouse distribution from part 1 will be more profitable than the warehouse distribution of part 2.

### 6.3 Optimal placement with the addition of clothing to inventory

The p-value for the t-test calculating the difference between the fitnesses in part 2 and part 3 is about 0.946, meaning that there is no significant difference. Therefore, optimal placement will not be affected very much by taking into account the clothing tax.

# 7 Model Strengths and Weaknesses

## 7.1 Strengths

1. **Converting Number of Warehouses to Coverage**
   Our program found the optimal coverage for an inputted number of warehouses. This allowed us to find the minimum number of warehouses to cover the whole continental United States as well as the optimal number of warehouses to cover a significant portion of the continental United States population.

2. **Map Reliability**
   The simulation of uncertain conditions needed for the representation of an overall transit day map would require many variables, i.e. traffic and weather, and increasing the number of variables creates a greater uncertainty in the model. Our model reduced this uncertainty by simply using pre-generated maps from the United Parcel Service. These maps are guaranteed to be correct, and allow us to make minimal assumptions.

3. **Efficiency**
   Our genetic algorithm for Part 1 was efficient. Downloading 1846 transit-day maps and processing them outside the program allowed for us not to send requests to ups.com and waste valuable time retrieving an image for every ZIP code. We also used an efficient method for counting the amount of yellow pixels in an image by using a built-in function from the Image class of the Python Imaging Library that quickly records the frequency of colors in an image, which saved us more valuable time than iterating through every pixel and counting how many of them were yellow.

   Our algorithm was efficient enough that we could test thousands of generations in a reasonable amount of time. While the genetic algorithm might not always find the most optimal set of ZIP codes due to the indeterminate process, our algorithm still produced one of the best possible set of ZIP codes almost every time.

## 7.2 Weaknesses

1. **Errors from random selection**
   Since our program used random selections of the 1846 ZIP codes, our program did not produce the best set of ZIP codes all the time. While we used 2000 generations in part 1 and 250 generations in part 2 to reduce the effects of the random selection, increasing the number of generations could not completely solve the problem. If the program selects a suboptimal set of ZIP codes in the first 50 or so generations, it may never correct it. The program will improve that set of ZIP codes, the chance that it completely fixes that set of core ZIP codes is extremely small even with extremely large amounts of generations. At times, the program would give a set of ZIP codes that covered less than a set of ZIP codes with one less warehouse, which obviously does not make sense. To fix this, we reran the program to get better optimization. While the process was slightly biased, we achieved a more accurate and more sensible result.

2. **Difficulty of achieving perfect optimization**
   Since our program produces one of the best possible set of ZIP codes but not the perfect set of ZIP codes, we were not sure how good a set of ZIP codes was. Even if we did at some point did achieve perfect optimization, we had no way of knowing it because of the uncertainty caused by the genetic algorithm.

3. **Overweighted tax**
   In our second and third genetic algorithms, our fitness function for an entire set of ZIP codes gave too much prioritization to the tax rates. This resulted in much less importance to spatial fitness than in our first genetic algorithm, meaning that our warehouse locations failed to be in range of a large portion of the continental landmass.

# 8 Code Analysis

## 8.1 Map Downloader Algorithm

*(Full code in Appendix F)*

The program begins by opening the CSV file containing the 1846 ZIP codes we used and appending each code to `ZIP_LIST`, an array. Then, it begins to iterate through each ZIP code in `ZIP_LIST`, where it saves the image from UPS using the function `get_img(zip)`.

`get_img(zip)` functions by sending an HTTPS `POST` request to "`https://www.ups.com/maps/results`" with data `{'zip': zip, 'stype': '0'}`, which means that the program requests the UPS server to generate a delivery time map for the provided ZIP code where the times are for delivering from the ZIP code (as opposed to delivering to the ZIP code). UPS will then return a full HTML page containing the image, and the program uses the `str.find()` function to look for the URL to the map image.

Once the URL of the image is retrieved, the program uses the `urlopen(url)` function from the `urllib.request` Python module to retrieve the binary data of the image. This data is then stored as an `img` file, a variable of datatype `PIL.Image` (from Pillow, which is a user-maintained version of the Python Imaging Library). `img`, which is originally in values of RGB only, is converted to RGBA to allow for transparent pixels (which are needed for the genetic algorithms). The program then iterates through every pixel in `img`, turning any pixel that is not black (map borders) or yellow (one-day delivery zone color) into full transparency. `img` is saved as a PNG file inside the folder "`images`", which should be in the same directory as the script.

## 8.2 Map Visualizer Algorithm

*(Full code in Appendix D)*

The Map Visualizer program begins by defining a new function `get_img`, which, opposed to retrieving the map from the Internet, retrieves the edited map from the "`images`" folder that was created in 7.1. A variable `base_img` is defined as the first image retrieved from the first ZIP code from the `zips` array (provided by the user). The program iterates through the rest of the ZIP codes in `zips`, overlaying their corresponding images on top of `base_img`, denoting overlap by reducing the amount of red in the overlapping pixels (therefore giving it a more greenish color). Once all images of the ZIP codes from `zips` are added, the program calls method `show()` on `base_img`, which displays the map on the screen.

## 8.3 Genetic Algorithm

*(Full code in Appendix E/G)*

The genetic algorithm begins by generating 10 arrays of $w$ random ZIP codes. It starts a repetitive process of sorting the 10 arrays by their generation using the `fitness(zips)` function and mutating the best 2 arrays to replace the last 8 arrays with the `mutate(zips)` function. Once it has repeated this process the user-given amount of times, it provides the best array from the most recent generation of arrays.

The `mutate(zips)` function operates by choosing a random ZIP code from the array of ZIP codes and replacing it with a new random ZIP code. This is how incremental change is made through mutation in the genetic algorithm.

The `fitness(zips)` function works differently for the $f_1$ and $f_2$ algorithms. The $f_1$ algorithm simply overlays all of the ZIP code images on top of one another and counts the total amount of yellow pixels in the final image by using the `PIL.Image` function `getcolors()`. It then divides the amount of yellow pixels by the amount of total pixels ($p_T = 91291$) to get the fitness value. Meanwhile, the $f_2$ algorithm adds the individual ZIP code fitness values (see Appendix C for formula) together, averages them by dividing the total by the amount of ZIP codes, and then uses that value as well as the outside pixel count to find the total fitness value of the set of ZIP codes.

## 8.4 Fitness Value Calculator Algorithm

*(Full code in Appendix H)*

This program essentially takes in `ZIPS`, an array containing ZIP codes, and calculates both the $f_1$ and $f_2$ values for the array. It uses both fitness functions from both algorithms to determine these values.

## 8.5 Other Pieces of Code

In Appendix I, a new sales tax table is included as a modification for the second genetic algorithm to find new fitness values when clothing taxes are included. The only modifications to make were to divide the sales tax of states which had no clothing tax by 2 (per Assumption 8).

In Appendix J, a small algorithm is included to find the average tax rate of a set of ZIP code

# 9 References

"Complete List of United States Zip Codes." *AggData*. Geonames.org, 16 Feb. 2012. Web. 13 Nov. 2016.

Jaaskelainen, Liisa. "The Sporting Goods Industry." *www.statista.com*. Statista, 01 Nov. 2016. Web. 13 Nov. 2016.

"The 2015 Tax Resource." *Sales Tax Rates By State 2016*. Tax-rates.org, 23 Feb. 2016. Web. 13 Nov. 2016.

"United States Area." *United States (Contiguous 48) Area*. Natural Earth Data, 13 Oct. 2014. Web. 13 Nov. 2016.

"United States. Ground Time-in-Transit Maps." *United Parcel Service.* The United Parcel Service, 10 July 2012. Web. 13 Nov. 2016.

"U.S. and World Population Clock Tell Us What You Think." *Population Clock*. United States Department of Commerce, 1 Jan. 2016. Web. 13 Nov. 2016.

U.S., The Conversation. "Simulating Evolution: How Close Do Computer Models Come to Reality?" *The Huffington Post*. TheHuffingtonPost.com, 6 May 2016. Web. 13 Nov. 2016.

"U.S. Zip Code Map." *, USA Zip Codes by State*. WhereIG, 3 Apr. 2013. Web. 13 Nov. 2016.

# 10 Appendices

## Appendix A

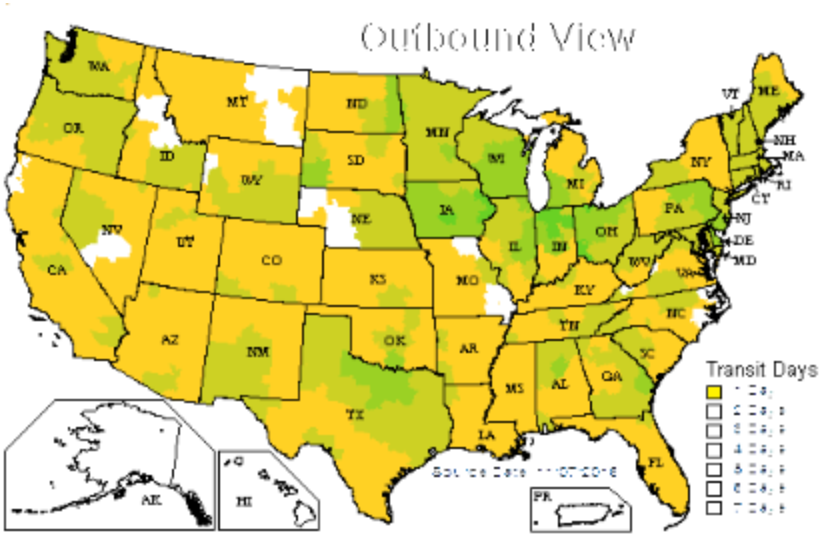


Figure 15. A map of the one-day transit coverage of the United States done manually. Using 30 warehouses, 92.5 % coverage was achieved. Thus, it is proven that it is possible to service at least 99 % of the United States using only 30 warehouses.

The following cities were used:
Sacramento, CA; Helena, MO; Arvada, CO; Madison, WI; Louisville, KY; El Paso, TX; Austin, TX; Seattle, WA; Los Angeles, CA; Portland, OR; Boise, ID; Aurora, IL; Bismarck, ND; Minneapolis, MN; Kansas City, KS; Sioux Falls, SD; Chandler, AZ; Sheridan, WY; Manchester, NH; Albany, NY; Washington DC; Pittsburgh, PA; Shreveport, LA; Atlanta, GA; Orlando, FL; Columbia, SC; Albuquerque, NM; Oklahoma City, OK; Island Falls, ME; and Detroit, MI.

## Appendix B

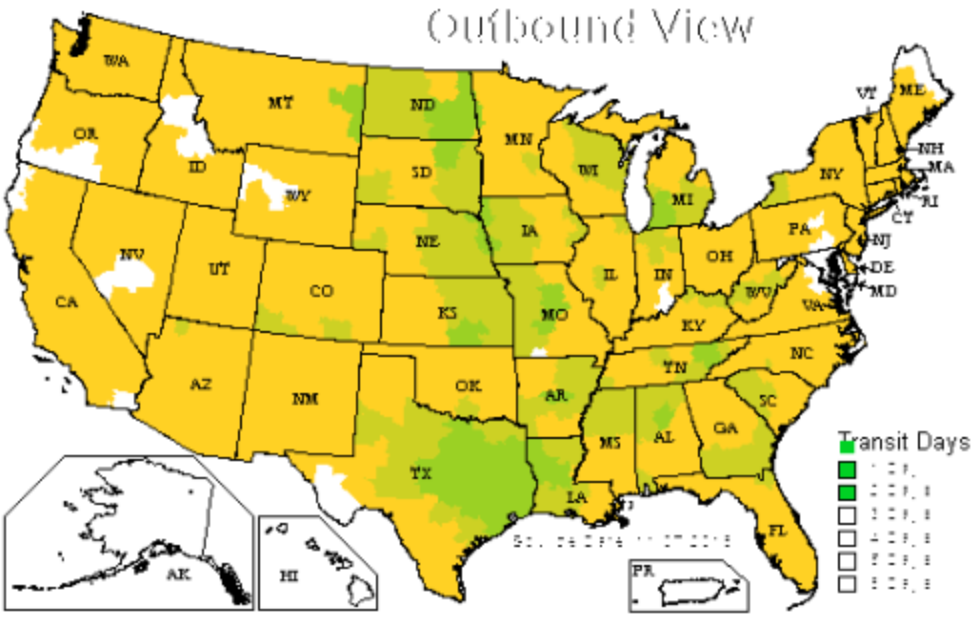


Figure 16. A map of the optimal one-day transit coverage from 23 ZIP code locations for warehouses. One-day transit was provided to 95.89% of the continental United States.

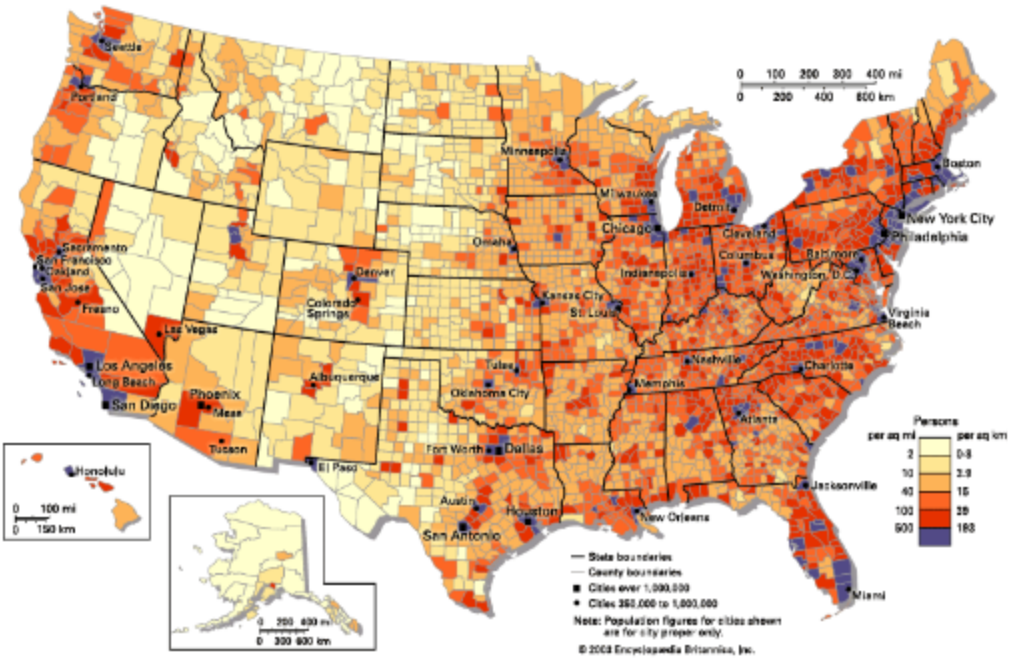


Figure 17. A population density map of the United States.

## Appendix C

*Derivations of Part II formulas*

Since sales tax is only applied when purchasing from the same state, it is clear that a warehouse will prioritize selling to out-of-state locations in order to increase sales. It is also evident that warehouses located in states with high taxes should have the least amount of intrastate pixels as possible within its one-day delivery zone. Therefore, the fitness formula for individual ZIP codes should weigh against state sales tax and weigh for exostate pixels.

In order to weigh against state sales tax, we can subtract state sales tax for the ZIP code from the maximum sales tax and divide by the maximum sales tax.

$$\frac{t_M - t_s}{t_M}$$

Therefore, the higher the sales tax, the lower this value will be. However, for California, which has the maximum sales tax, this value would be the minimum, which is zero. Thus, we decided to lock the minimum to .5 by altering this expression slightly.

$$\frac{2t_M - t_s}{2t_M}$$

In order to weigh for outside pixels from the state of the ZIP code, we would simply divide the number of outside pixels by the total number of pixels that are within the ZIP code's reach.

$$\frac{p_o}{p_T}$$

Therefore, the final fitness formula for a singular ZIP code would be represented as such:

$$f_{II}(n) = \frac{2t_M - t_s}{2t_M} \cdot \frac{p_o}{p_T}$$

Now, all of these individual ZIP code fitness values must be compiled together to form a singular fitness value for the entire set of ZIP codes. However, consideration must also be given to the $f_I$ value. These two different values need to be weighted in such a manner that the algorithm would produce a >50% filled map with decent tax efficiency. After weighing them equally, we discovered that the spatial fitness ($f_I$) was still given large priority, so we settled on weighing the compiled ZIP code tax fitnesses to spatial fitness in a ratio of 3 : 1.

The compiled value of all individual ZIP code fitness values should just simply be the average of those values, like so:

$$\frac{\sum_{n=1}^{w} f_{II}(n)}{w}$$

Thus, the final formula for sales tax fitness would result in:

$$f_{II} = \frac{3}{4} \times \left( \frac{\sum_{n=1}^{w} f_{II}(n)}{w} \right) + \frac{1}{4} \times f_{I}$$

## Appendix D

*Full code of Delivery Map Visualizer*

```
# COMAP 7211
# Program that displays zip code map with overlap represented by
green

import PIL
import PIL.Image
import io
import numpy

zips = [] # INSERT ZIPS HERE

top_zips = []

def get_img(zip):
     try:
          img = PIL.Image.open("images/" + zip + ".png")
          return img
     except:
          return False

base_img = None

def add_img(zip, img):
     w, h = img.size
     pix = img.load()
     count = 0
     overlap_count = 0
     b_pix = base_img.load()
     for x in range(w):
          for y in range(h):
               r, g, b, a = pix[x, y]
               if (r == 255 and g == 209 and b == 36):
                    count += 1
                    b_r, b_g, b_b, b_a = b_pix[x, y]
                    if (b_g == g and b_b == b):
                         b_pix[x, y] = (b_r - 50, g, b)
```

```
                    overlap_count += 1
                else:
                    b_pix[x, y] = (r, g, b)
    print(zip, "Count:", count, "Overlap:", overlap_count)
    if count >= 6000:
        top_zips.append((zip, count))

for zip in zips:
    img = get_img(zip)
    if img != False:
        w, h = img.size
        pix = img.load()
        count = 0
        if base_img == None:
            base_img = img
            for x in range(w):
                for y in range(h):
                    r, g, b, a = pix[x, y]
                    if (r == 255 and g == 209 and b == 36):
                        count += 1
            print(zip, ":", count-56)
        else:
            add_img(zip, img)
        if count >= 6000:
            top_zips.append((zip, count))
    else:
        print("Failed to open image of", zip)

base_img.show()
print(top_zips)

while True:
    inp = input("add zip code or type q to quit: ")
    if inp == "q":
        break
    elif int(inp) != None:
        add_img(get_img(inp))
        base_img.show()
```

## Appendix E

*Full code of Genetic Algorithm to generate Delivery Map for Part I*

```
# COMAP 7211
# Program that generates an efficient set of warehouse locations
through a 'genetic' process

import PIL
import PIL.Image
import io
import os
import numpy
import random

# Zip List
ZIP_LIST = []
for file_name in os.listdir("images"):
     ZIP_LIST.append(file_name[:5])

# Constants
QTY = 30 # Amount of Warehouses
POP = 10 # Amount of Warehouse Location Sets per Generation
ITER = 2000 # Amount of iterations
TOTAL_PIXELS = 91291 # Total number of pixels in the UPS map
(excluding Alaska, Hawaii, P.R., and border pixels)
COLOR = (255, 209, 36, 255) # Color of one-day delivery zone

# Zip cache
zip_cache = {}

# Function that determines if a zip code is legit by UPS
def is_legit_zip(zip):
     return get_img(zip) != False

# Function that generates a beginning list of 50 zip codes
def gen_random_zips(q):
     random_zips = []
     for j in range(q):
          zip = ZIP_LIST[random.randint(0, len(ZIP_LIST)-1)]
```

```
          i = int(zip)
          while (i >= 601 and i <= 988) or (i >= 96701 and i <=
96898) or (i >= 99501 and i <= 99950) or not is_legit_zip(zip):
               zip = ZIP_LIST[random.randint(0,
len(ZIP_LIST)-1)]
               i = int(zip)
          random_zips.append(zip)
     return random_zips

# Function that retrieves the UPS delivery time map given the
zip code
def get_img(zip):
     if zip in zip_cache:
          return zip_cache[zip]

     try:
          img = PIL.Image.open("images/" + zip + ".png")
     except:
          return False

     zip_cache[zip] = img

     return img

# Function that counts the number of times a certain color
occurs in an image
def count_color(img, color):
     colors = img.getcolors()
     pixels = None
     for tup in colors:
          if tup[1] == color:
               return tup[0]

# Function that determines the 'fitness' of a set of warehouse
locations (a.k.a. its efficiency) as a
# float between 0. and 1.
def fitness(zips, show=False):
     comb_img = get_img(zips[0]).copy()
     for i in range(1, len(zips)-1):
```

```
            img = get_img(zips[i])
            comb_img.paste(img, (0,0), img)
      if show == True:
            comb_img.show()
      filled_pixels = count_color(comb_img, COLOR)
      filled_pixels -= count_color(comb_img.crop((466, 213, 545,
352)), COLOR)
      #print("Filled pixels:", filled_pixels)
      del comb_img
      return filled_pixels / TOTAL_PIXELS

# Function that clones an array
def clone(array):
      new_array = []
      for elem in array:
            new_array.append(elem)
      return new_array

# Function that replaces one zip code with another random one.
def mutate(zips):
      new_zips = clone(zips)
      ind = random.randint(0, len(new_zips)-1)
      new_zips[ind] = gen_random_zips(1)[0]
      return new_zips

gen = []
for i in range(POP):
      gen.append(gen_random_zips(QTY))

for i in range(ITER):
      gen = sorted(gen, key=fitness, reverse=True)
      print("Generation:", i, "Best:", fitness(gen[0]))
      for j in range(2, POP, 1):
            gen[j] = mutate(gen[j%2])

print("FINAL BEST:", fitness(gen[0], show=True))
print(gen[0])
```

## Appendix F

*Full code for retrieving UPS map images and saving them to hard-drive in a folder named "images"*

```
# COMAP 7211
# Downloads all of the zip code images

import requests
import PIL
import PIL.Image
import urllib.request as urllib
import io
import numpy
import random
import csv

# Zip List
ZIP_LIST = []
with open('validpostalcodes.csv', 'r') as csvfile:
     linereader = csv.reader(csvfile)
     for row in linereader:
          zip = row[0]
          if len(zip) == 4:
               zip = "0" + zip
          ZIP_LIST.append(zip)

print(len(ZIP_LIST))

# Function that determines if a zip code is legit by UPS
def is_legit_zip(zip):
     try:
          get_img(zip)
          return True
     except:
          return False

# Function that retrieves the UPS delivery time map given the
zip code
def get_img(zip):
```

```
     r = requests.post("https://www.ups.com/maps/results",
data={'zip': zip, 'stype': 'O'})
     index = r.text.find('id="imgMap" src="')
     index2 = r.text.find('" alt="US Time in Transit Map"')
     url_ending = r.text[index+17:index2]
     whole_url = "https://www.ups.com" + url_ending

     fd = urllib.urlopen(whole_url)
     image_file = io.BytesIO(fd.read())
     img = PIL.Image.open(image_file)
     img = img.convert('RGBA')

     w, h = img.size
     pix = img.load()
     for x in range(w):
          for y in range(h):
               r, g, b, a = pix[x, y]
               if (r != 255 or g != 209 or b != 36) and (r != 0
or g != 0 or b != 0):
                    pix[x, y] = (255, 255, 255, 0)

     img.save("images/" + zip + ".png")

     return img

i = 0
for zip in ZIP_LIST:
     try:
          get_img(zip)
     except:
          print("Image", i, "of", len(ZIP_LIST), "(Zip Code",
zip + ") failed.")
     i += 1
     print("Downloaded image", i, "of", len(ZIP_LIST), "(Zip
Code", zip + ")")
```

## Appendix G

*Full code for second genetic algorithm, which generates a map for reducing tax liability for customers*

```
# COMAP 7211
# Program that generates an efficient set of warehouse locations
through a 'genetic' process
# For use in Part II

import PIL
import PIL.Image
import io
import os
import numpy
import random

# Zip Code to State Color
ZIP_COLOR = [
     [(1000, 2799), (0, 0, 207, 255)], #MA
     [(2800, 2999), (0, 0, 191, 255)], #RI
     [(3000, 3899), (0, 0, 239, 255)], #NH
     [(3900, 4999), (0, 0, 255, 255)], #ME
     [(5000, 5999), (0, 0, 223, 255)], #VT
     [(6000, 6999), (0, 0, 175, 255)], #CT
     [(7000, 8999), (0, 0, 143, 255)], #NJ
     [(10000, 14999), (0, 0, 159, 255)], #NY
     [(15000, 19699), (0, 0, 127, 255)], #PA
     [(19700, 19999), (0, 0, 111, 255)], #DE
     [(20600, 21999), (0, 0, 95, 255)], #MD
     [(22000, 24699), (0, 0, 63, 255)], #VA
     [(24700, 26999), (0, 0, 79, 255)], #WV
     [(27000, 28999), (0, 0, 47, 255)], #NC
     [(29000, 29999), (0, 0, 31, 255)], #SC
     [(30000, 31999), (0, 255, 0, 255)], #GA
     [(32000, 34999), (0, 0, 15, 255)], #FL
     [(35000, 36999), (0, 127, 0, 255)], #AL
     [(37000, 38599), (0, 143, 0, 255)], #TN
     [(38600, 39999), (0, 111, 0, 255)], #MS
     [(40000, 42999), (0, 159, 0, 255)], #KY
     [(43000, 45999), (0, 239, 0, 255)], #OH
```

```
      [(46000, 47999), (0, 207, 0, 255)], #IN
      [(48000, 49999), (0, 223, 0, 255)], #MI
      [(50000, 52999), (0, 47, 0, 255)], #IA
      [(53000, 54999), (0, 191, 0, 255)], #WI
      [(55000, 56799), (0, 31, 0, 255)], #MN
      [(57000, 57999), (255, 0, 0, 255)], #SD
      [(58000, 58999), (0, 15, 0, 255)], #ND
      [(59000, 59999), (111, 0, 0, 255)], #MT
      [(60000, 62999), (0, 175, 0, 255)], #IL
      [(63000, 65999), (0, 63, 0, 255)], #MO
      [(66000, 67999), (223, 0, 0, 255)], #KS
      [(68000, 69999), (239, 0, 0, 255)], #NE
      [(70000, 71599), (0, 95, 0, 255)], #LA
      [(71600, 72999), (0, 79, 0, 255)], #AR
      [(73000, 74999), (207, 0, 0, 255)], #OK
      [(75000, 79999), (191, 0, 0, 255)], #TX
      [(80000, 81999), (159, 0, 0, 255)], #CO
      [(82000, 83199), (127, 0, 0, 255)], #WY
      [(83200, 83999), (95, 0, 0, 255)], #ID
      [(84000, 84999), (143, 0, 0, 255)], #UT
      [(85000, 86999), (79, 0, 0, 255)], #AZ
      [(87000, 88499), (175, 0, 0, 255)], #NM
      [(88900, 89999), (63, 0, 0, 255)], #NV
      [(90000, 96199), (47, 0, 0, 255)], #CA
      [(97000, 97999), (31, 0, 0, 255)], #OR
      [(98000, 99499), (15, 0, 0, 255)]  #WA
]

# State Color to State Tax
COLOR_TAX = {
      (0, 0, 207, 255): 0.0625, #MA
      (0, 0, 191, 255): 0.07, #RI
      (0, 0, 239, 255): 0.0, #NH
      (0, 0, 255, 255): 0.055, #ME
      (0, 0, 223, 255): 0.06, #VT
      (0, 0, 175, 255): 0.0635, #CT
      (0, 0, 143, 255): 0.07, #NJ
      (0, 0, 159, 255): 0.04, #NY
      (0, 0, 127, 255): 0.06, #PA
```

```
(0, 0, 111, 255): 0.0, #DE
(0, 0, 95, 255): 0.06, #MD
(0, 0, 63, 255): 0.053, #VA
(0, 0, 79, 255): 0.06, #WV
(0, 0, 47, 255): 0.0475, #NC
(0, 0, 31, 255): 0.06, #SC
(0, 255, 0, 255): 0.04, #GA
(0, 0, 15, 255): 0.06, #FL
(0, 127, 0, 255): 0.04, #AL
(0, 143, 0, 255): 0.07, #TN
(0, 111, 0, 255): 0.07, #MS
(0, 159, 0, 255): 0.06, #KY
(0, 239, 0, 255): 0.0575, #OH
(0, 207, 0, 255): 0.07, #IN
(0, 223, 0, 255): 0.06, #MI
(0, 47, 0, 255): 0.06, #IA
(0, 191, 0, 255): 0.05, #WI
(0, 31, 0, 255): 0.0688, #MN
(255, 0, 0, 255): 0.04, #SD
(0, 15, 0, 255): 0.05, #ND
(111, 0, 0, 255): 0.0, #MT
(0, 175, 0, 255): 0.0625, #IL
(0, 63, 0, 255): 0.0423, #MO
(223, 0, 0, 255): 0.065, #KS
(239, 0, 0, 255): 0.055, #NE
(0, 95, 0, 255): 0.04, #LA
(0, 79, 0, 255): 0.065, #AR
(207, 0, 0, 255): 0.045, #OK
(191, 0, 0, 255): 0.0625, #TX
(159, 0, 0, 255): 0.029, #CO
(127, 0, 0, 255): 0.04, #WY
(95, 0, 0, 255): 0.06, #ID
(143, 0, 0, 255): 0.0595, #UT
(79, 0, 0, 255): 0.056, #AZ
(175, 0, 0, 255): 0.0513, #NM
(63, 0, 0, 255): 0.0685, #NV
(47, 0, 0, 255): 0.075, #CA
(31, 0, 0, 255): 0.0, #OR
(15, 0, 0, 255): 0.065  #WA
```

```
}

# Zip List
ZIP_LIST = []
for file_name in os.listdir("images"):
     ZIP_LIST.append(file_name[:5])

# Constants
QTY = 30 # Amount of Warehouses
POP = 10 # Amount of Warehouse Location Sets per Generation
ITER = 250 # Amount of iterations
TOTAL_PIXELS = 91291 # Total number of pixels in the UPS map
(excluding Alaska, Hawaii, P.R., and border pixels)
COLOR = (255, 209, 36, 255) # Color of one-day delivery zone
MAX_TAX = 0.075 # Maximum Tax (from California)

# Zip cache
zip_cache = {}

# Function that determines if a zip code is legit by UPS
def is_legit_zip(zip):
     return get_img(zip) != False

# Function that generates a beginning list of 50 zip codes
def gen_random_zips(q):
     random_zips = []
     for j in range(q):
          zip = ZIP_LIST[random.randint(0, len(ZIP_LIST)-1)]
          i = int(zip)
          while (i >= 601 and i <= 988) or (i >= 96701 and i <=
96898) or (i >= 99501 and i <= 99950) or (i >= 20000 and i <=
20599) or not is_legit_zip(zip):
               zip = ZIP_LIST[random.randint(0,
len(ZIP_LIST)-1)]
               i = int(zip)
          random_zips.append(zip)
     return random_zips
```

```
# Function that retrieves the UPS delivery time map given the
zip code
def get_img(zip):
     if zip in zip_cache:
          return zip_cache[zip]

     try:
          img = PIL.Image.open("images/" + zip + ".png")
     except:
          return False

     zip_cache[zip] = img

     return img

# Function that counts the number of times a certain color
occurs in an image
def count_color(img, color):
     colors = img.getcolors()
     pixels = None
     for tup in colors:
          if tup[1] == color:
               return tup[0]

# Function that determines the state color based on zip
def zip_to_color(zip):
     i_zip = int(zip)
     for zip_color in ZIP_COLOR:
          if i_zip >= zip_color[0][0] and i_zip <=
zip_color[0][1]:
               return zip_color[1]

# Function that determines the 'fitness' of a set of warehouse
locations (a.k.a. its efficiency) as a
# float between 0. and 1.
def fitness(zips, show=False):
     zip_fitness = 0.
     for zip in zips:
          zip_color = zip_to_color(zip)
```

```
          zip_img = get_img(zip)
          col_img = PIL.Image.open("colormap.png")
          col_img.convert("RGBA")
          col_img.paste(zip_img, (0,0), zip_img)
          zip_tax = COLOR_TAX[zip_color]
          other_pixels = 0
          total_pixels = 0
          for tup in col_img.getcolors():
               color = tup[1]
               if color != (0, 0, 0, 255) and color != (255,
255, 255, 0):
                    total_pixels += 1
                    if color != zip_color:
                         other_pixels += 1
          zip_fitness += (2 * MAX_TAX - zip_tax) / (2 * MAX_TAX)
* other_pixels/total_pixels
     zip_fitness /= len(zips)
     comb_img = get_img(zips[0]).copy()
     for i in range(1, len(zips)-1):
          img = get_img(zips[i])
          comb_img.paste(img, (0,0), img)
     if show == True:
          comb_img.show()
     filled_pixels = count_color(comb_img, COLOR)
     filled_pixels -= count_color(comb_img.crop((466, 213, 545,
352)), COLOR)
     print("Filled pixels:", filled_pixels)
     del comb_img
     return 3*zip_fitness/4 + (filled_pixels / TOTAL_PIXELS) / 4

# Function that clones an array
def clone(array):
     new_array = []
     for elem in array:
          new_array.append(elem)
     return new_array

# Function that replaces one zip code with another random one.
def mutate(zips):
```

```
     new_zips = clone(zips)
     ind = random.randint(0, len(new_zips)-1)
     new_zips[ind] = gen_random_zips(1)[0]
     return new_zips

gen = []
for i in range(POP):
     gen.append(gen_random_zips(QTY))

for i in range(ITER):
     gen = sorted(gen, key=fitness, reverse=True)
     print("Generation:", i, "Best:", fitness(gen[0]))
     for j in range(2, POP, 1):
          gen[j] = mutate(gen[j%2])

print("FINAL BEST:", fitness(gen[0], show=True))
print(gen[0])
```

## Appendix H

*Full code for calculating both $f_I$ and $f_{II}$ values given the set of zip codes*

```
# COMAP 7211
# Program that determines the f1/f2 value (spatial fitness) of a
set of zip codes

import PIL
import PIL.Image
import io
import os
import numpy
import random

# Zip Code to State Color
ZIP_COLOR = [
     [(1000, 2799), (0, 0, 207, 255)], #MA
     [(2800, 2999), (0, 0, 191, 255)], #RI
     [(3000, 3899), (0, 0, 239, 255)], #NH
     [(3900, 4999), (0, 0, 255, 255)], #ME
     [(5000, 5999), (0, 0, 223, 255)], #VT
     [(6000, 6999), (0, 0, 175, 255)], #CT
     [(7000, 8999), (0, 0, 143, 255)], #NJ
     [(10000, 14999), (0, 0, 159, 255)], #NY
     [(15000, 19699), (0, 0, 127, 255)], #PA
     [(19700, 19999), (0, 0, 111, 255)], #DE
     [(20600, 21999), (0, 0, 95, 255)], #MD
     [(22000, 24699), (0, 0, 63, 255)], #VA
     [(24700, 26999), (0, 0, 79, 255)], #WV
     [(27000, 28999), (0, 0, 47, 255)], #NC
     [(29000, 29999), (0, 0, 31, 255)], #SC
     [(30000, 31999), (0, 255, 0, 255)], #GA
     [(32000, 34999), (0, 0, 15, 255)], #FL
     [(35000, 36999), (0, 127, 0, 255)], #AL
     [(37000, 38599), (0, 143, 0, 255)], #TN
     [(38600, 39999), (0, 111, 0, 255)], #MS
     [(40000, 42999), (0, 159, 0, 255)], #KY
     [(43000, 45999), (0, 239, 0, 255)], #OH
     [(46000, 47999), (0, 207, 0, 255)], #IN
```

```
      [(48000, 49999), (0, 223, 0, 255)], #MI
      [(50000, 52999), (0, 47, 0, 255)], #IA
      [(53000, 54999), (0, 191, 0, 255)], #WI
      [(55000, 56799), (0, 31, 0, 255)], #MN
      [(57000, 57999), (255, 0, 0, 255)], #SD
      [(58000, 58999), (0, 15, 0, 255)], #ND
      [(59000, 59999), (111, 0, 0, 255)], #MT
      [(60000, 62999), (0, 175, 0, 255)], #IL
      [(63000, 65999), (0, 63, 0, 255)], #MO
      [(66000, 67999), (223, 0, 0, 255)], #KS
      [(68000, 69999), (239, 0, 0, 255)], #NE
      [(70000, 71599), (0, 95, 0, 255)], #LA
      [(71600, 72999), (0, 79, 0, 255)], #AR
      [(73000, 74999), (207, 0, 0, 255)], #OK
      [(75000, 79999), (191, 0, 0, 255)], #TX
      [(80000, 81999), (159, 0, 0, 255)], #CO
      [(82000, 83199), (127, 0, 0, 255)], #WY
      [(83200, 83999), (95, 0, 0, 255)], #ID
      [(84000, 84999), (143, 0, 0, 255)], #UT
      [(85000, 86999), (79, 0, 0, 255)], #AZ
      [(87000, 88499), (175, 0, 0, 255)], #NM
      [(88900, 89999), (63, 0, 0, 255)], #NV
      [(90000, 96199), (47, 0, 0, 255)], #CA
      [(97000, 97999), (31, 0, 0, 255)], #OR
      [(98000, 99499), (15, 0, 0, 255)]  #WA
]

# State Color to State Tax
COLOR_TAX = {
      (0, 0, 207, 255): 0.0625, #MA
      (0, 0, 191, 255): 0.07, #RI
      (0, 0, 239, 255): 0.0, #NH
      (0, 0, 255, 255): 0.055, #ME
      (0, 0, 223, 255): 0.06, #VT
      (0, 0, 175, 255): 0.0635, #CT
      (0, 0, 143, 255): 0.07, #NJ
      (0, 0, 159, 255): 0.04, #NY
      (0, 0, 127, 255): 0.06, #PA
      (0, 0, 111, 255): 0.0, #DE
```

```
    (0, 0, 95, 255): 0.06, #MD
    (0, 0, 63, 255): 0.053, #VA
    (0, 0, 79, 255): 0.06, #WV
    (0, 0, 47, 255): 0.0475, #NC
    (0, 0, 31, 255): 0.06, #SC
    (0, 255, 0, 255): 0.04, #GA
    (0, 0, 15, 255): 0.06, #FL
    (0, 127, 0, 255): 0.04, #AL
    (0, 143, 0, 255): 0.07, #TN
    (0, 111, 0, 255): 0.07, #MS
    (0, 159, 0, 255): 0.06, #KY
    (0, 239, 0, 255): 0.0575, #OH
    (0, 207, 0, 255): 0.07, #IN
    (0, 223, 0, 255): 0.06, #MI
    (0, 47, 0, 255): 0.06, #IA
    (0, 191, 0, 255): 0.05, #WI
    (0, 31, 0, 255): 0.0688, #MN
    (255, 0, 0, 255): 0.04, #SD
    (0, 15, 0, 255): 0.05, #ND
    (111, 0, 0, 255): 0.0, #MT
    (0, 175, 0, 255): 0.0625, #IL
    (0, 63, 0, 255): 0.0423, #MO
    (223, 0, 0, 255): 0.065, #KS
    (239, 0, 0, 255): 0.055, #NE
    (0, 95, 0, 255): 0.04, #LA
    (0, 79, 0, 255): 0.065, #AR
    (207, 0, 0, 255): 0.045, #OK
    (191, 0, 0, 255): 0.0625, #TX
    (159, 0, 0, 255): 0.029, #CO
    (127, 0, 0, 255): 0.04, #WY
    (95, 0, 0, 255): 0.06, #ID
    (143, 0, 0, 255): 0.0595, #UT
    (79, 0, 0, 255): 0.056, #AZ
    (175, 0, 0, 255): 0.0513, #NM
    (63, 0, 0, 255): 0.0685, #NV
    (47, 0, 0, 255): 0.075, #CA
    (31, 0, 0, 255): 0.0, #OR
    (15, 0, 0, 255): 0.065  #WA
}
```

```
# Constants
TOTAL_PIXELS = 91291 # Total number of pixels in the UPS map
(excluding Alaska, Hawaii, P.R., and border pixels)
COLOR = (255, 209, 36, 255) # Color of one-day delivery zone
MAX_TAX = 0.075 # Maximum Tax (from California)
ZIPS = [] # INSERT ZIPS HERE

# Zip List
ZIP_LIST = []
for file_name in os.listdir("images"):
     ZIP_LIST.append(file_name[:5])

# Zip cache
zip_cache = {}

# Function that retrieves the UPS delivery time map given the
zip code
def get_img(zip):
     if zip in zip_cache:
          return zip_cache[zip]

     try:
          img = PIL.Image.open("images/" + zip + ".png")
     except:
          return False

     zip_cache[zip] = img

     return img

# Function that counts the number of times a certain color
occurs in an image
def count_color(img, color):
     colors = img.getcolors()
     pixels = None
     for tup in colors:
          if tup[1] == color:
               return tup[0]
```

```
# Function that determines the 'fitness' of a set of warehouse
locations (a.k.a. its efficiency) as a
# float between 0. and 1.
def fitness1(zips, show=False):
     comb_img = get_img(zips[0]).copy()
     for i in range(1, len(zips)-1):
          img = get_img(zips[i])
          comb_img.paste(img, (0,0), img)
     if show == True:
          comb_img.show()
     filled_pixels = count_color(comb_img, COLOR)
     filled_pixels -= count_color(comb_img.crop((466, 213, 545,
352)), COLOR)
     del comb_img
     return filled_pixels / TOTAL_PIXELS

# Function that determines the state color based on zip
def zip_to_color(zip):
     i_zip = int(zip)
     for zip_color in ZIP_COLOR:
          if i_zip >= zip_color[0][0] and i_zip <=
zip_color[0][1]:
               return zip_color[1]

# Function that determines the 'fitness' of a set of warehouse
locations (a.k.a. its efficiency) as a
# float between 0. and 1.
def fitness2(zips, show=False):
     zip_fitness = 0.
     for zip in zips:
          zip_color = zip_to_color(zip)
          zip_img = get_img(zip)
          col_img = PIL.Image.open("colormap.png")
          col_img.convert("RGBA")
          col_img.paste(zip_img, mask=zip_img)
          zip_tax = COLOR_TAX[zip_color]
          other_pixels = 0
          total_pixels = 0
```

```
          for tup in col_img.getcolors():
               color = tup[1]
               if color != (0, 0, 0, 255) and color != (255,
255, 255, 0):
                    total_pixels += 1
                    if color != zip_color:
                         other_pixels += 1
          zip_fitness += (2 * MAX_TAX - zip_tax) / (2 * MAX_TAX)
* other_pixels/total_pixels
     zip_fitness /= len(zips)
     comb_img = get_img(zips[0]).copy()
     for i in range(1, len(zips)-1):
          img = get_img(zips[i])
          comb_img.paste(img, (0,0), img)
     if show == True:
          comb_img.show()
     filled_pixels = count_color(comb_img, COLOR)
     filled_pixels -= count_color(comb_img.crop((466, 213, 545,
352)), COLOR)
     del comb_img
     return 3*zip_fitness/4 + (filled_pixels / TOTAL_PIXELS)/4

print("Calculating Fitness 1...")
print("Fitness 1", str(fitness1(ZIPS)*100) + "%")
print("Calculating Fitness 2...")
print("Fitness 2", str(fitness2(ZIPS)*100) + "%")
```

## Appendix I

*Modification for second genetic algorithm to include no clothing tax (Using assumption 8) to fulfill Part III*

```
# State Color to State Tax
COLOR_TAX = {
     (0, 0, 207, 255): 0.03125, #MA
     (0, 0, 191, 255): 0.035, #RI
     (0, 0, 239, 255): 0.0, #NH
     (0, 0, 255, 255): 0.055, #ME
     (0, 0, 223, 255): 0.03, #VT
     (0, 0, 175, 255): 0.0635, #CT
```

```
(0, 0, 143, 255): 0.035, #NJ
(0, 0, 159, 255): 0.02, #NY
(0, 0, 127, 255): 0.03, #PA
(0, 0, 111, 255): 0.0, #DE
(0, 0, 95, 255): 0.06, #MD
(0, 0, 63, 255): 0.053, #VA
(0, 0, 79, 255): 0.06, #WV
(0, 0, 47, 255): 0.0475, #NC
(0, 0, 31, 255): 0.06, #SC
(0, 255, 0, 255): 0.04, #GA
(0, 0, 15, 255): 0.06, #FL
(0, 127, 0, 255): 0.04, #AL
(0, 143, 0, 255): 0.07, #TN
(0, 111, 0, 255): 0.07, #MS
(0, 159, 0, 255): 0.06, #KY
(0, 239, 0, 255): 0.0575, #OH
(0, 207, 0, 255): 0.07, #IN
(0, 223, 0, 255): 0.06, #MI
(0, 47, 0, 255): 0.06, #IA
(0, 191, 0, 255): 0.05, #WI
(0, 31, 0, 255): 0.0344, #MN
(255, 0, 0, 255): 0.04, #SD
(0, 15, 0, 255): 0.05, #ND
(111, 0, 0, 255): 0.0, #MT
(0, 175, 0, 255): 0.0625, #IL
(0, 63, 0, 255): 0.0423, #MO
(223, 0, 0, 255): 0.065, #KS
(239, 0, 0, 255): 0.055, #NE
(0, 95, 0, 255): 0.04, #LA
(0, 79, 0, 255): 0.065, #AR
(207, 0, 0, 255): 0.045, #OK
(191, 0, 0, 255): 0.0625, #TX
(159, 0, 0, 255): 0.029, #CO
(127, 0, 0, 255): 0.04, #WY
(95, 0, 0, 255): 0.06, #ID
(143, 0, 0, 255): 0.0595, #UT
(79, 0, 0, 255): 0.056, #AZ
(175, 0, 0, 255): 0.0513, #NM
(63, 0, 0, 255): 0.0685, #NV
```

```
     (47, 0, 0, 255): 0.075, #CA
     (31, 0, 0, 255): 0.0, #OR
     (15, 0, 0, 255): 0.065  #WA
}
```

## Appendix J

*Code that finds average tax rate of set of zip codes*

```
# COMAP 7211
# Program that generates an efficient set of warehouse locations
through a 'genetic' process
# For use in Part II

import PIL
import PIL.Image
import io
import os
import numpy
import random

# Zip Code to State Color
ZIP_COLOR = [
     [(1000, 2799), (0, 0, 207, 255)], #MA
     [(2800, 2999), (0, 0, 191, 255)], #RI
     [(3000, 3899), (0, 0, 239, 255)], #NH
     [(3900, 4999), (0, 0, 255, 255)], #ME
     [(5000, 5999), (0, 0, 223, 255)], #VT
     [(6000, 6999), (0, 0, 175, 255)], #CT
     [(7000, 8999), (0, 0, 143, 255)], #NJ
     [(10000, 14999), (0, 0, 159, 255)], #NY
     [(15000, 19699), (0, 0, 127, 255)], #PA
     [(19700, 19999), (0, 0, 111, 255)], #DE
     [(20600, 21999), (0, 0, 95, 255)], #MD
     [(22000, 24699), (0, 0, 63, 255)], #VA
     [(24700, 26999), (0, 0, 79, 255)], #WV
     [(27000, 28999), (0, 0, 47, 255)], #NC
     [(29000, 29999), (0, 0, 31, 255)], #SC
     [(30000, 31999), (0, 255, 0, 255)], #GA
```

```
    [(32000, 34999), (0, 0, 15, 255)], #FL
    [(35000, 36999), (0, 127, 0, 255)], #AL
    [(37000, 38599), (0, 143, 0, 255)], #TN
    [(38600, 39999), (0, 111, 0, 255)], #MS
    [(40000, 42999), (0, 159, 0, 255)], #KY
    [(43000, 45999), (0, 239, 0, 255)], #OH
    [(46000, 47999), (0, 207, 0, 255)], #IN
    [(48000, 49999), (0, 223, 0, 255)], #MI
    [(50000, 52999), (0, 47, 0, 255)], #IA
    [(53000, 54999), (0, 191, 0, 255)], #WI
    [(55000, 56799), (0, 31, 0, 255)], #MN
    [(57000, 57999), (255, 0, 0, 255)], #SD
    [(58000, 58999), (0, 15, 0, 255)], #ND
    [(59000, 59999), (111, 0, 0, 255)], #MT
    [(60000, 62999), (0, 175, 0, 255)], #IL
    [(63000, 65999), (0, 63, 0, 255)], #MO
    [(66000, 67999), (223, 0, 0, 255)], #KS
    [(68000, 69999), (239, 0, 0, 255)], #NE
    [(70000, 71599), (0, 95, 0, 255)], #LA
    [(71600, 72999), (0, 79, 0, 255)], #AR
    [(73000, 74999), (207, 0, 0, 255)], #OK
    [(75000, 79999), (191, 0, 0, 255)], #TX
    [(80000, 81999), (159, 0, 0, 255)], #CO
    [(82000, 83199), (127, 0, 0, 255)], #WY
    [(83200, 83999), (95, 0, 0, 255)], #ID
    [(84000, 84999), (143, 0, 0, 255)], #UT
    [(85000, 86999), (79, 0, 0, 255)], #AZ
    [(87000, 88499), (175, 0, 0, 255)], #NM
    [(88900, 89999), (63, 0, 0, 255)], #NV
    [(90000, 96199), (47, 0, 0, 255)], #CA
    [(97000, 97999), (31, 0, 0, 255)], #OR
    [(98000, 99499), (15, 0, 0, 255)]  #WA
]

# State Color to State Tax
COLOR_TAX = {
    (0, 0, 207, 255): 0.0625, #MA
    (0, 0, 191, 255): 0.07, #RI
    (0, 0, 239, 255): 0.0, #NH
```

```
(0, 0, 255, 255): 0.055, #ME
(0, 0, 223, 255): 0.06, #VT
(0, 0, 175, 255): 0.0635, #CT
(0, 0, 143, 255): 0.07, #NJ
(0, 0, 159, 255): 0.04, #NY
(0, 0, 127, 255): 0.06, #PA
(0, 0, 111, 255): 0.0, #DE
(0, 0, 95, 255): 0.06, #MD
(0, 0, 63, 255): 0.053, #VA
(0, 0, 79, 255): 0.06, #WV
(0, 0, 47, 255): 0.0475, #NC
(0, 0, 31, 255): 0.06, #SC
(0, 255, 0, 255): 0.04, #GA
(0, 0, 15, 255): 0.06, #FL
(0, 127, 0, 255): 0.04, #AL
(0, 143, 0, 255): 0.07, #TN
(0, 111, 0, 255): 0.07, #MS
(0, 159, 0, 255): 0.06, #KY
(0, 239, 0, 255): 0.0575, #OH
(0, 207, 0, 255): 0.07, #IN
(0, 223, 0, 255): 0.06, #MI
(0, 47, 0, 255): 0.06, #IA
(0, 191, 0, 255): 0.05, #WI
(0, 31, 0, 255): 0.0688, #MN
(255, 0, 0, 255): 0.04, #SD
(0, 15, 0, 255): 0.05, #ND
(111, 0, 0, 255): 0.0, #MT
(0, 175, 0, 255): 0.0625, #IL
(0, 63, 0, 255): 0.0423, #MO
(223, 0, 0, 255): 0.065, #KS
(239, 0, 0, 255): 0.055, #NE
(0, 95, 0, 255): 0.04, #LA
(0, 79, 0, 255): 0.065, #AR
(207, 0, 0, 255): 0.045, #OK
(191, 0, 0, 255): 0.0625, #TX
(159, 0, 0, 255): 0.029, #CO
(127, 0, 0, 255): 0.04, #WY
(95, 0, 0, 255): 0.06, #ID
(143, 0, 0, 255): 0.0595, #UT
```

```
     (79, 0, 0, 255): 0.056, #AZ
     (175, 0, 0, 255): 0.0513, #NM
     (63, 0, 0, 255): 0.0685, #NV
     (47, 0, 0, 255): 0.075, #CA
     (31, 0, 0, 255): 0.0, #OR
     (15, 0, 0, 255): 0.065  #WA
}

# Zip List
ZIP_LIST = []
for file_name in os.listdir("images"):
     ZIP_LIST.append(file_name[:5])

# Constants
TOTAL_PIXELS = 91291 # Total number of pixels in the UPS map
(excluding Alaska, Hawaii, P.R., and border pixels)
COLOR = (255, 209, 36, 255) # Color of one-day delivery zone
MAX_TAX = 0.075 # Maximum Tax (from California)

ZIPS = [] # INSERT ZIPS HERE

# Zip Cache
zip_cache = {}

# Function that retrieves the UPS delivery time map given the
zip code
def get_img(zip):
     if zip in zip_cache:
          return zip_cache[zip]

     try:
          img = PIL.Image.open("images/" + zip + ".png")
     except:
          return False

     zip_cache[zip] = img

     return img
```

```
# Function that counts the number of times a certain color
occurs in an image
def count_color(img, color):
     colors = img.getcolors()
     pixels = None
     for tup in colors:
          if tup[1] == color:
               return tup[0]

# Function that determines the state color based on zip
def zip_to_color(zip):
     i_zip = int(zip)
     for zip_color in ZIP_COLOR:
          if i_zip >= zip_color[0][0] and i_zip <=
zip_color[0][1]:
               return zip_color[1]

avg_tax_total = 0.
for zip in ZIPS:
     zip_color = zip_to_color(zip)
     avg_tax_total += COLOR_TAX[zip_color]

print(avg_tax_total/len(ZIPS))
```

## Appendix K

*All of the data collected by the model. F1 represents area fitness, and F2 represents tax fitness.*

**Part I Data**

**15 - F1 = 81.60278669310228% F2 = 70.69736333994223%**
['40339', '98901', '71006', '75103', '29301', '85533', '87002', '59001', '80010', '64018', '12007', '57017', '83314', '93210', '26238']
**16 - F1 = 86.27246935623446% F2 = 73.05561733905861%**
['12015', '97004', '27343', '59831', '46103', '56324', '38602', '76008', '64018', '87001', '93210', '82212', '83301', '31701', '85001', '80727']
**17 - F1 = 88.78093130757687% F2 = 72.14229165042363%**
['12108', '57051', '93623', '31701', '85001', '83301', '54101', '80201', '59054', '87002', '76043', '23824', '71004', '66012', '40347', '98901', '67003']
**18 - F1 = 89.38449573342389% F2 = 71.7544572666893%**
['80020', '42021', '49010', '35007', '66012', '93601', '16910', '76001', '57435', '32754', '87011', '98012', '85321', '83301', '27239', '54926', '59002', '31623']
**19 - F1 = 92.3190675970249% F2 = 72.58045806671436%**
['85321', '44003', '93210', '81220', '58027', '42022', '68019', '31772', '77412', '12025', '82922', '98612', '75135', '54201', '59010', '27239', '87001', '13737', ‘22352’]
**20 - F1 = 92.74079591635539% F2 = 75.52769897908885%**
['42021', '59313', '12019', '27343', '31772', '97812', '87001', '66012', '38004', '59801', '80020', '44001', '85321', '77363', '82922', '75135', '54101', '93210', '57003', '49805']
**21- F1 = 93.93368459103307% F2 = 71.8858021001392%**
['57435', '80020', '71019', '87002', '32099', '54409', '90001', '49010', '59054', '78837', '40010', '75135', '26707', '85901', '29325', '83322', '98901', '12031', '89418', '64018', '94506']
**22 - F1 = 94.69389096405998% F2 = 73.10983637737863%**
['75103', '97801', '89406', '87002', '58201', '27501', '59054', '72354', '31701', '01001', '40339', '85602', '80201', '93401', '66007', '57002', '16910', '48809', '83311', '54102', '77510', '48435']
**23 - F1 = 95.88677963873766% F2 = 73.61734708359745%**
['49710', '44017', '42021', '30122', '87008', '83325', '58620', '68005', '27343', '77331', '12108', '85324', '76008', '54106', '71004', '98220', '93601', '66402', '57051', '59010', '33825', '80020', '57650']
**24 - F1 = 96.54840017088212% F2 = 73.25585004272051%**
['93210', '95531', '58216', '67005', '68922', '51009', '42001', '83322', '15062', '59010', '98061', '87002', '01810', '59831', '78837', '57017', '31772', '85602', '49010', '89406', '80201', '27212', '71027', '29069']
**25 - F1 = 97.22973787120308% F2 = 72.58943446780076%**

['59002', '98612', '71019', '83311', '42038', '83824', '56324', '89418', '79821', '93401', '32628', '87512', '79033', '27214', '12025', '57002', '64018', '48809', '85611', '80201', '15012', '68005', '78010', '30006', '46530']

**26 - F1 = 97.77415079252062% F2 = 72.28392231351476%**

['31701', '18056', '01002', '80020', '42033', '79018', '71065', '27214', '93623', '49001', '58009', '69024', '83325', '78007', '57002', '87544', '59001', '95531', '85135', '79843', '89406', '66012', '54926', '98236', '45701', '43408']

**27 - F1 = 98.66690035162283% F2 = 74.20746582864645%**

['99017', '85325', '68005', '97101', '40339', '77510', '83322', '57003', '66843', '48809', '12007', '89440', '17017', '32008', '80010', '62009', '93426', '30002', '79843', '27201', '74829', '58477', '54814', '87544', '71006', '59002', '39041']

**28 - F1 = 98.89802937858059% F2 = 74.02629305893088%**

['49010', '71019', '83325', '44003', '79830', '01810', '54926', '93426', '97004', '67005', '89701', '87008', '78332', '27011', '66012', '32008', '58520', '22626', '59010', '36701', '42027', '83501', '85324', '57012', '12007', '69001', '80436', '56323']

**29 - F1 = 99.15435256487496% F2 = 72.85065710673598%**

['98612', '92328', '53014', '18011', '30003', '74829', '79714', '71404', '57213', '78111', '42020', '94503', '83325', '93013', '64018', '27006', '01810', '87011', '83802', '69135', '34601', '49719', '80436', '56510', '59001', '40339', '85325', '26374', '12018']

**30 - F1 = 97.8376838899782% F2 = 73.9827543058279%**

['27343', '37707', '48627', '18403', '03570', '79731', '49902', '39039', '73010', '59002', '83314', '94503', '97101', '85324', '52216', '83522', '68943', '46702', '31701', '57017', '91319', '79223', '87001', '65018', '21520', '58009', '77404', '72636', '80422', '57621']

**31 - F1 = 99.56731769834923% F2 = 74.43376490845827%**

['30236', '56510', '93201', '42322', '67001', '50025', '83311', '49010', '01810', '97004', '89440', '26148', '27201', '78610', '97801', '18039', '38602', '85324', '80010', '82601', '95605', '79745', '54930', '33002', '69345', '59054', '65614', '87001', '57003', '70517', '43434']

**32 - F1 = 100% F2 = 74.43376490845827%**

['30236', '56510', '93201', '42322', '67001', '50025', '83311', '49010', '01810', '97004', '89440', '26148', '27201', '78610', '97801', '18039', '38602', '85324', '80010', '82601', '95605', '79745', '54930', '33002', '69345', '59054', '65614', '87001', '57003', '70517', '43434', '83463']

---

**Part II Data**

**15 Warehouses - F1 = 48.247910527872406% F2 = 82.38531096530143 %**

['59002', '19701', '59801', '03280', '97004', '80010', '59353', '64018', '03218', '97828', '71027', '81325', '59001', '03215', '97701']

**16 Warehouses - F1 = 41.48492184333614% F2 = 81.05873046083404%**

['59001', '03280', '97901', '80422', '71065', '97101', '97102', '97014', '81321', '59036', '59353', '35901', '59501', '59410', '59447', '59419']

**17 Warehouses - F1 = 47.8053696421334% F2 = 81.71604829288629%**
['81433', '59326', '03215', '97101', '59223', '97497', '71404', '59054', '35901', '59222', '59501', '59313', '57012', '80436', '59010', '59831', '59016']
**18 Warehouses - F1 = 52.81681655365809% F2 = 81.148648582895 %**
['59313', '59410', '59353', '97101', '30122', '66402', '97901', '59827', '57002', '81121', '59201', '59010', '82922', '59001', '71019', '59223', '97107', '59801']
**19 Warehouses - F1 = 50.4058450449661% F2 = 82.26988231387309%**
['59416', '59011', '03031', '97901', '57012', '03570', '03280', '87008', '97497', '59501', '80422', '97801', '59018', '35007', '59036', '65614', '59353', '59054', '03218']
**20 Warehouses - F1 = 60.26881072613949% F2 = 82.06470268153487%**
['59326', '56520', '59701', '03218', '59222', '82922', '40348', '03227', '30442', '59054', '97014', '59062', '97001', '97828', '71404', '81433', '59832', '19701', '65634', '59416']
**21 Warehouses - F1 = 68.81401233418409% F2 = 83.45826498830794%**
['19701', '59420', '97801', '59010', '59641', '59353', '81321', '59711', '59523', '03215', '97101', '45102', '85321', '03280', '57002', '64018', '59036', '75007', '82922', '30012', '59054']
**22 Warehouses - F1 = 65.61435490984 3248% F2 = 82.9763157973354%**
['59018', '45102', '97812', '59018', '03046', '97102', '31804', '59016', '59523', '81121', '59831', '80422', '97812', '71019', '64018', '59002', '97101', '23014', '82930', '57031', '97710', '59901']
**23 Warehouses - F1 = 66.43261657775684% F2 = 82.829893274874%**
['80201', '97497', '66007', '85360', '81320', '03431', '59411', '97001', '63826', '03218', '76401', '59831', '57002', '59054', '59084', '19701', '59313', '97828', '59353', '30018', '97004', '59018', '71301']
**24 Warehouses - F1 = 69.20287870655377 % F2 = 82.45488634330511 %**
['59401', '31701', '97710', '97102', '71065', '58031', '63826', '80436', '27343', '68337', '59261', '97004', '43408', '59420', '03280', '83336', '59410', '59002', '87009', '59416', '59801', '59701', '59001', '03031']
**25 Warehouses - F1 =71.79459092353025% F2 = 84.57664773088256%**
['81030', '97812', '40348', '57003', '30230', '97010', '97101', '59054', '59222', '97497', '71019', '97014', '59201', '59827', '85001', '59002', '87008', '19701', '83322', '97812', '03046', '59701', '64018', '59053', '59420']
**26 Warehouses - F1 = 65.65817002771358% F2 = 84.65492712231301%**
['59313', '97812', '59222', '80511', '97102', '82922', '03431', '89418', '59222', '66012', '59223', '59641', '59016', '57003', '03031', '59831', '03227', '31727', '97497', '19701', '81325', '59701', '59447', '59010', '71065', '59313']
**27 Warehouses - F1 = 78.48637872298474% F2 = 85.10307616222768%**
['59003', '03431', '59201', '81321', '82930', '59523', '03227', '76238', '31316', '03570', '97901', '59011', '80020', '59353', '97101', '97801', '85360', '59416', '19701', '71002', '68879', '57002', '97710', '40348', '62613', '59062', '59062']
**28 Warehouses - F1 = 75.60438597452104% F2 = 84.99395363648741%**

['59032', '35004', '97101', '59711', '59724', '43009', '59447', '76570', '27212', '97102', '03046', '85001', '82922', '03218', '87002', '66012', '59313', '59827', '97801', '59353', '03431', '54601', '03280', '59214', '80020', '59054', '59016', '59326']

**29 Warehouses - F1 = 78.46118456364811% F2 = 86.10495131332584%**

['97101', '19701', '80422', '85602', '97828', '03227', '59214', '97107', '59054', '03431', '57012', '35901', '59410', '03031', '59711', '97801', '59832', '87011', '59420', '59201', '40355', '65634', '59084', '59411', '59313', '71027', '93210', '83401', '97497']

**30 Warehouses - F1 = 79.47552332650535% F2 = 83.520547498293%**

['14029', '88121', '59435', '97101', '79025', '59401', '59016', '82322', '59641', '59901', '81101', '64018', '71004', '59501', '03280', '63101', '35013', '23011', '85611', '59411', '59831', '93623', '58009', '03227', '59010', '97014', '59831', '84023', '59084', '97901']

**31 Warehouses - F1 = 82.79896156247604% F2 = 84.95296619707062%**

['76238', '14041', '87544', '94503', '68019', '59058', '03570', '31805', '19701', '53015', '28018', '97004', '03215', '57003', '83201', '97102', '85135', '97901', '59419', '59410', '97102', '59353', '59901', '97107', '59010', '97001', '80020', '59827', '42027', '59801', '59801']

**32 Warehouses - F1 = 86.60875661346683% F2 85.26468915336672%**

['59724', '64018', '15012', '81201', '93601', '71004', '97812', '59214', '59201', '03431', '59326', '57003', '85001', '03215', '59701', '68337', '59827', '59032', '81320', '59447', '59419', '42211', '59261', '59201', '59003', '53501', '83314', '27013', '59054', '31701', '73052', '03218']

---

## Part III Data

**15 Warehouses - F1 = 43.79292591821757% F2 = 81.94823147955439%**

['59001', '59801', '59054', '59214', '97828', '71002', '19701', '82922', '97497', '57012', '59002', '03227', '59724', '97001', '97001']

**16 Warehouses - F1 = 52.73904327918415%  F2 = 80.74726081979604%**

['97010', '71404', '03570', '59054', '35007', '81321', '57012', '80422', '15012', '59222', '97014', '97801', '59214', '59701', '97828', '03280']

**17 Warehouses - F1 = 53.82020133419505% F2 = 82.23446209825465%**

['82930', '59313', '97102', '97004', '97101', '76230', '35013', '59901', '81121', '59032', '57012', '59411', '19701', '03431', '59353', '59002', '59523']

**18 Warehouses - F1= 48.78903725449387%, F2 = 83.05837042473458%**

['59313', '82922', '19701', '57003', '97004', '81122', '59801', '97014', '97901', '71404', '59223', '97004', '03280', '59724', '59901', '97001', '59501', '97014']

**19 Warehouses -  F1 = 56.45463408222059% F2 = 82.06102694160777%**

['59831', '97828', '87008', '97710', '59827', '82212', '97101', '59010', '14029', '64421', '59084', '30014', '03227', '56324', '59801', '97107', '71404', '03215', '59036']

**20 Warehouses - F1 = 64.79280542441205% F2 = 81.90320135610303%**
['66012', '59002', '71019', '81121', '59711', '19701', '97497', '82930', '59201', '03570', '56324', '97101', '97107', '97801', '80436', '59353', '97828', '40348', '12018', '59701']
**21 Warehouses - F1 = 62.64801568610269% F2 = 82.8048610643828%**
['71027', '59711', '59001', '97101', '59827', '81325', '82922', '03280', '68943', '97812', '59724', '30122', '17005', '57017', '85324', '97014', '97701', '97801', '59401', '97901', '59831']
**22 Warehouses - F1 = 67.03618100360386% F2 = 82.36359070544641%**
['82930', '56510', '14009', '35013', '59901', '97497', '77801', '59831', '81611', '97828', '03046', '59018', '27343', '68879', '59032', '59011', '81433', '85360', '97710', '59001', '03280', '59313']
**23 Warehouses -F1 = 66.37017887853129% F2 = 84.48167515441543%**
['59801', '59036', '03031', '59416', '59711', '03431', '64018', '80020', '59016', '14414', '84624', '35004', '19701', '59002', '59016', '87008', '57003', '73301', '59214', '59711', '97101', '97828', '59058']
**24 Warehouses - F1 = 65.97145392207336% F2 = 83.39286348051834%**
['03570', '97004', '59011', '59501', '65018', '59435', '82930', '57003', '71006', '59016', '59001', '89406', '59214', '59058', '59062', '03280', '59054', '97701', '30012', '81325', '59832', '16028', '81123', '59353']
**25 Warehouses - F1 = 69.49206383980896% F2 = 83.89301595995223%**
['68019', '59326', '85001', '59701', '40046', '17003', '59353', '97901', '81320', '59827', '31620', '73530', '59523', '59401', '03227', '71004', '97107', '57002', '03218', '97004', '59827', '80511', '59010', '59084', '59001']
**26 Warehouses - F1 = 79.61354350374078% F2 = 84.39761664516595%**
['40348', '59058', '59326', '83311', '03218', '59032', '59054', '59010', '59501', '87009', '80010', '97107', '31701', '97101', '59411', '59326', '14029', '59313', '85324', '71404', '59801', '93623', '68937', '59501', '57017', '12501']
**27 Warehouses - F1: 78.03288385492546% F2: 83.91007281558323%**
['03227', '10451', '57012', '59054', '89701', '97101', '03227', '59831', '97801', '97107', '53005', '03215', '76539', '16001', '67005', '59801', '87002', '81235', '97102', '85931', '83322', '35013', '97010', '59003', '97701', '59420', '97102']
**28 Warehouses - F1 = 78.26182208541915% F2 = 83.38866980706906%**
['87011', '42022', '75135', '59701', '59724', '97812', '59401', '64446', '16028', '97004', '19701', '59001', '59054', '59036', '03570', '30439', '82922', '59003', '80436', '89701', '97001', '56520', '28612', '03218', '85920', '59084', '59420', '97710']
**29 Warehouses - F1 = 79.89067925644368% F2 = 84.64853188307645%**
['63828', '59261', '03031', '59401', '73301', '59010', '97014', '54106', '97014', '59058', '81433', '97710', '83217', '80422', '59058', '66407', '85001', '13737', '35007', '15012', '97801', '59010', '73547', '97497', '28006', '57213', '59002', '19701', '59002']
**30 Warehouses - F1 = 75.33710880590638% F2 = 85.32594386814325%**

['59641', '12108', '59435', '15006', '97901', '59003', '71019', '59222', '83203', '59711', '51501', '31622', '59001', '97701', '56623', '59003', '87001', '59058', '63101', '19701', '59901', '59001', '59062', '80010', '85324', '59011', '59010', '59214', '27343', '59801']
**31 Warehouses - F1 = 78.18843040387333% F2 = 85.76968824612963%**
['81122', '19701', '82212', '55111', '97107', '03570', '19701', '59214', '35007', '75135', '59401', '59901', '59641', '03280', '68924', '59701', '85325', '63821', '28702', '59724', '57017', '44805', '97004', '82930', '97107', '59222', '59419', '03570', '97701', '59058', '59831']
**32 Warehouses - F1 = 83.89326439627126% F2 = 84.62019109906781%**
['59223', '82922', '89701', '57002', '59313', '81321', '97010', '03570', '59001', '59831', '56208', '97010', '23004', '31772', '97101', '63828', '43512', '14009', '80020', '85135', '73052', '03046', '59223', '59058', '97710', '59214', '59641', '59435', '97828', '69135', '76634', '59701']

## Average Tax Rates

*1 represents data from the first genetic algorithm (prioritizing spatial fitness)*
*2 represents data from the second genetic algorithm (prioritizing less tax liability for customers)*
*'W' stands for 'Warehouses'*

**15W**
1) **4.9406666666666675%**
2) **0.09353333333333333%**

**16W**
1) **4.7025000000000004%**
2) **0.08625%**

**17W**
1) **5.010588235294118%**
2) **1.047058823594119%**

**18W**
1) **5.118333333333334%**
2) **1.4111111111111111%**

**19W**
1) **4.917368421052633%**
2) **1.0663157894736841%**

**20W**
1) **4.531500000000001%**
2) **1.6005000000000002%**

**21W**
1) **5.319523809523811%**
2) **1.7490476190476188%**

**22W**

1) **5.1127272727272736%**
2) **1.6854545454545452%**

**23W**

1) **5.070869565217392%**
2) **1.7556521739130432%**

**24W**

1) **5.176250000000001%**
2) **1.9691666666666666%**

**25W**

1) **5.3436000000000004%**
2) **1.6744000000000002%**

**26W**

1) **5.431923076923077%**
2) **1.3519230769230771%**

**27W**

1) **5.091851851851852%**
2) **1.9037037037037036%**

**28W**

1) **5.139642857142857%**
2) **1.7814285714285717%**

**29W**

1) **5.387586206896552%**
2) **1.7020689655172415%**

**30W**

1) **5.095333333333333%**
2) **2.269666666666664%**

**31W**

1) **5.091612903225807%**
2) **2.149355838709677%**

**29W**

1) **5.12%**
2) **0.227755000000004%**