Abstract
The Zombie Apocalypse has come to the United States, and the government is in need of an application (app) that will create a route to vaccinate people in designated safe zones. This app will receive a list of five or more cities and will find the shortest route that reaches each of the cities. This problem is an instance of the travelling salesman problem. The people within a one hundred mile radius of each city will be vaccinated. The result of this investigation is an Android app with a brute force algorithm that can reasonably handle up to eleven cities and display all the information to the user. The five cities that are included in the route are Boston, Massachusetts, Newark, New Jersey, Washington D.C, Pittsburgh, Pennsylvania, and Detroit, Michigan in that order. This app would be very useful if a Zombie Apocalypse actually occurs, or another catastrophic infection.

1. Introduction
A zombie apocalypse has struck the United States of America, and a virus has infected much of the population. Zombies have already overrun the top ten most populous cities, and the rest of the population needs to be vaccinated as quickly as possible. A U.S Government official asked for an Android app that can determine the most efficient way to vaccinate as many people possible in the shortest distance and time. There are currently enough resources to create five Safe Zones, each including a base city and spanning all areas one hundred miles from the city. All people within the one hundred mile radius can be vaccinated. Each Safe Zone must have an international airport, a major hospital, and a navigable waterway nearby. The final application will solve for and display information to users about the shortest five-city route with the highest persons per mile ratio.

The app includes a dynamic list of cities from which the user can pick and choose, so the user may add up to any three cities of their choosing. The list does however come with eight default cities that our research showed to be the best eight. The user gets the option to input new cities, view the current list of cities, and view all details of the cities in that list. Once the user selects the list of active cities, the app parses all of the distance between cities, calculates the optimal route, and displays that route to the user with a list and a link to a map of the solution in Google Maps.

2. Background
2.1 Traveling Salesman Problem
The problem the app solves is a version of the Traveling Salesman Problem (TSP). The TSP has many applications, which is why it is one of the most intensely studied problems in math and computer science.
The TSP solution can be manipulated to solve different real life scenarios. In this project, the app user wishes to find the shortest distance between five destinations. The shortest route will save the client time and money. Such a solution could help in planning the shortest tour for a rock concert, or programming the tour of a computer controlled drill press that needs to drill hundreds of holes in a circuit board, or even getting groceries at the store in the shortest amount of time.

Computational problems all possess a certain level of difficulty. The level of difficulty is measured by how many operations are performed to solve for a problem size \( n \). An ‘easy’ problem is where an input of size \( n \) can be solved with \( n^a \) operations, where \( a \) is some integer. Other problems, like the TSP, take on many more operations. These problems are classified as non-deterministic polynomial, meaning that no current algorithm can solve a TSP with \( n^a \) operations.

Many solutions and approximation algorithms of the TSP do exist. The solution that was chosen was brute force, where every possible solution set is created but only the shortest is selected. This solution, as with all brute force solutions, is one of the slowest approaches. A brute force algorithm for the TSP takes roughly \( n! \) operations. A genetic algorithm was considered, which is a self-improving algorithm that generates a number of random solutions, and combines the best solutions into a set of new solutions.\(^1\) This algorithm replicates the process of natural selection, but because it ultimately relies on randomness, a purely genetic algorithm is not a particularly effective solution. The algorithm could be improved if combined with a simple optimization check. Put simply, the program looks at a solution, and if it sees two crossed paths it swaps the paths, as seen in Figure 1.

![Figure 1. Crossing Path Optimization](image)

The discrete optimization algorithm utilizes recursion, a method of solving computer problems in which a program refers to itself.\(^2\) The recursive algorithm would have greatly improved efficiency in the app. While solving the TSP for 11 cities using the brute force solution took a few seconds, the recursive method would have taken a few milliseconds. However, the implementation of this algorithm would have required drastic adaptations to the majority of the already existing codebase. Considering the requirements posed by the client only called for a route to be calculated between five cities (for which the runtime is infinitesimal), it was decided that the effort to implement a completely new recursive algorithm would have dramatically increase complexity with negligible benefits based on the current system requirement.

These algorithms require much fewer calculations, but are much more difficult to program. In the case of the app, where only a small--less than 12--number of cities are involved, a brute force solution only requires a small number of times more calculations, and is simpler to implement.

### 2.2 Systems and Software Engineering
In almost all companies, engineers collectively work on solving a problem set by breaking up into groups that specialize in different fields. An engineering project of any size can be split into systems and software engineering. System engineers focus on how to design and manage complex engineering projects over their life cycles. They ensure that all likely aspects of a project or system are considered, and integrated into a whole. Designs and planning are often recorded on two diagrams in particular—a Case Diagram and a Class Diagram. The Use Case Diagram, as seen in figure 2, depicts all of the different users for the app and how they may want to use it. The class diagram is an illustration of relationships and dependencies among classes in UML.

Software engineers, on the other hand, take the plans that the systems engineers created and make working code to test and implement. They take the ideas and make them into a tangible, working item. Although sometimes these engineers work directly with code, a lot of the work done by software engineers is in designing how the code will be structured, testing code, and managing those whole physically type the code.

2.3 Android

Android is a leading mobile device operating system commonly found on smartphones and tablets. Developed by Google, it is currently the most popular mobile platform. In recent years, Android took over the lead from Apple’s iOS with approximately 900 million device activations in total.

Android was the optimal development platform for this project for various reasons. Firstly, it’s based around the Java programming language, with the Android SDK (Software Development Kit) providing the functions relevant to Android devices. Java is also compatible with both Windows and Mac computers whereas developing for Apple devices is only possible on a Mac.

Additionally, apps integrate and test faster on Android than iOS. This is because developer accounts are not required and apps can simply be loaded onto tablets or smartphones for real-world testing. Fortunately, there was a tablet available, which made designing and testing much more efficient.

All Android development is done within Eclipse, a Java IDE (Integrated Development Environment) that is freely available on both Windows and Mac. In conjunction with the Android SDK, Eclipse helped to simplify the development workflow by having all of the core features of Android accessible in one location.

2.4 Object Oriented Programming

Java is a very expansive language, and it can be used in many different ways. There are several ways to develop software through a variety
structures and still accomplish the same objective. The four ways to hold data are in named variables, matrices and arrays, maps, and objects.

The first way to store the name, population, and other information for a city is through creating a variable for each of these pieces of information. However, this process makes the code much longer and means that each variable has to be named in a way that is clear in case the variable changes in the future. The second way to store data is through a matrix, which is easier to organize, because it is shorter to create, store information in, and pull data from. It also means one city can have all of its information in one data structure. The downside is that matrices in Java only store one type of data—either number or strings. The third way to store data is still better, because it provides the same kind of structure as a matrix, but pulling data is done by a keyword, rather than by a number. This process typically is easier to read and understand the code, but it is limited in its dimensions. The final way to structure code is with objects. Objects can include multiple variables of different data types—integer, string, double—and each piece of data has an assigned name, which makes it both easy to access and easy to understand.

2.5 Graphical User Interface (GUI)

The Graphical User Interface (GUI) is the visual component of the Android App that the user interacts with. All the icons, fonts, and layouts are part of the GUI and enhance the app’s accessibility and marketability. Android allows the creator to easily manipulate the GUI of their app and includes an entire set of pre-designed GUI options that can be downloaded and incorporated.

3. Methods and Experimental Design

3.1 Selecting Cities

The project began by compiling a list of all the cities within the United States with an international airport, major hospital with an emergency division, and a navigable waterway—requirements created by the client. First, a list was found of all United States cities with international airports on One World Nations Online. The list narrowed down the number of possible safe zones to 52 cities. Then the top ten most populated cities were struck from the list as the client had indicated that they were no longer available as safe zones. A Google Map was then used to locate each of the fifty-two cities and ensure that each city was located near a body of water easily accessible for transportation.

By doing so, the fifty-two cities were truncated down to less than twenty cities after finding that many cities did not meet this criterion. Finally, it was ensured that each of the remaining twenty cities had a major hospital using Google Maps, which each did. A population distribution map of the United States was then examined with the purpose of seeking out the most...
densely populated clots of land. It was determined that the East Coast and California were the most populated regions, eliminating all the cities in the central US. Only fifteen cities remained afterward.

To reach the best set of five cities, circles were plotted of a one-hundred mile radius on a map of the United States for each of the 15 cities using the app found on FreeMapTools. Although this app did not provide the population count of residents living within the circles, it provided a major benefit—the ability to take note of the bounding cities of each plotted circle and ensure that neither of the circles overlapped, since an overlap would mean vaccinating fewer humans. For example, Baltimore’s circle has a higher population than Washington D.C.’s, but if the circles are plotted on a map, Baltimore's would overlap with Newark’s. Washington D.C.’s circle barely touches Newark’s, therefore the total population would actually increase if Washington D.C. was selected. Ultimately, five safe zones in the Northeast region of the United States were chosen, as well as three backup cities in case the client blacked out a city in the set of five.

To find the estimated population within each city’s radius, the radius map and an interactive population map were cross referenced. The population map provided the population count of counties in each state. The radius map determined the border counties, and the population inside those border counties were added up to get an estimation of the total population vaccinated in that specific safe zone.

This early design had the major flaw of completely ignoring the populations of the cities. For instance, if three times as many people can be saved by driving twice as far, than a better program would designate that route as more optimal. The next program written did exactly this. If with the same process, and generated all five city subsets of the eight cities, and found the shortest route for each subset. The new program, however, did not simply pick the shortest best route; it divided the total distance of each best route by the total population saved for that route. This was designated as the score of that route, and the path with the highest score, or the most people per mile, was selected as the optimal route.

For example, Ontario (California) and its surrounding area have a population of 16.88 Million and would seem to be good as a safe zone, but instead Pittsburgh, with a population of only 6.04 Million, was chosen. It was realized that despite Ontario having such a large population, it was located in California, and the amount of time needed to reach California by road would cause the route to be inefficient. In other words, the ratio of people per mile with Ontario in the set of safe zones is actually lower than the same ratio with Pittsburgh is in the set.

3.2 The Initial Algorithm

The final algorithm defined optimal as the shortest total route, meaning that any city in the South or the West Coast would be less optimal than a close group of five on the East Coast. The program generated all possible subsets of five cities from eight cities in total. For each of these subsets, the program then generates all permutations—all orders of the five cities—and added up the distances city to city. Out of all of these
permutations, the one with the shortest total distance was saved as the best route for that subset of five. Once the best route for all subsets of five were found, the best of the best routes was selected as the optimal path.

3.3 Retrieving Distance Data

Mapping data is an essential component to the Android app, as it needs to have access to driving distances between cities. For flexibility purposes (the ability for the user to manually add cities), it was determined that it was not sufficient to code the distances of the pre-determined cities into the program. Instead, a more modular system that would pull driving distances from an online mapping service based on the city names provided by the user was opted for.

Initially, the Google Maps API was used, since it is one of the most robust and efficient services available. In testing, however, it was found that it was at one point unable to retrieve driving distances from Google. Google sets a 2,500 request daily limit for its distance API, and during testing that limit was reached. Though in reality it's unlikely that so many requests would be necessary in a single 24-hour period, it was reasoned that for flexibility purposes it would be optimal to use a service with a bigger limit.

Fortunately, MapQuest offers free direction data with unlimited requests. MapQuest provided the same information on distances between cities. There are several significant drawbacks to using MapQuest however. MapQuest returned a much larger file than Google, which means the more bits of data were read in by the program. It also returns the data slower than Google, meaning the wait time for the user increased by about 120%. The data, which is returned in a JSON (Javascript Object Notation) formatted file as shown in Figure 5, was then parsed by the program and saved the distances between cities in the code.

![Figure 5. JSON Format](image.png)

The original software would pull the distance data from MapQuest one pair at a time, which meant that it could take up to two minutes just to load all of the data for an eleven city problem. However, MapQuest also offers a route Matrix service, which--given a list of cities--will return the distances between the first city and all of the other cities in the list. This means that the wait time for loading distance data could be substantially reduced.

The app should also display some sort of map of the best solution to the user. The three possible ways to do that would be a static image, an embedded map from Google or MapQuest that would by dynamic, or a link to the outside maps app. Every Android device has a maps app already loaded on to it, so with a simple click of a button, the user could be redirected to a map with all the information on the optimal route already plugged in. This has the advantage of being a dynamic, rather than a static image. Because the no API key from Google to embed a dynamic Google Maps map into the app could be obtained, and time constraints necessitated linking the user to the native maps app.

3.4 Graphical User Interface Development
Fluidity and simplicity is incredibly crucial to an app’s success, as it is what the user sees and interacts with. First, stencils were made showing potential layouts and designs on paper, referring to the Android developer guide’s suggestions on design practices to maximize the user experience. From there, sketches were compared to the graphical capabilities of Android to determine what designs would be feasible in such a short period of time. Based on that feedback, the graphics were redesigned and represented to the software team. Once the designs were agreed upon, the designs were turned in code using the Eclipse Android Graphic Design Interface. The final designs for the app are displayed below in figure 4.

The GUI for an app also includes the architecture, or how the different screens within the app will connect to each other. Our app contains five different screens. The first screen presented to the user is a simple start screen, which has the title of the app and a simple start button in the middle, as seen in the left most image in figure 4.1. Upon clicking the button, the city menu screen, which is the right most image in figure 4.1. This is where the user can select cities, and this screen connects to the last three screens. The info buttons next to each city name take the user to the city information screen—right of figure 4.2—which simply lists the name, population, waterway, airport, and hospital for a city. The last three boxes in the city menu screen, when clicked, will lead the user to the add city screen—right of figure 4.2—where the user can add in information for a new city. The calculate button at the button of the city menu screen will bring the user to a screen with the route listed, total population saved, and the distance of the route, as seen on the right of figure 4.3. There is also a button on this final solution screen which can take the user to a Google map of the five city route generated by the app.

4. Results and Discussion
The final set of five cities is comprised of:
- Boston, Massachusetts
- Newark, New Jersey
- Washington, D.C.
- Pittsburgh, Pennsylvania
- Detroit, Michigan

The three backup cities are:
- Ontario, California
- Charlotte, North Carolina
- Orlando, Florida

The total distance traveled is 967 miles and the total population is sixty-four million two hundred ninety thousand people. The five safe zones primarily cover the northeast region of the United States. When plotted on a map, the five safe zones appear to form a “V” shape as shown in the Concept of Operation Map (figure 6). This set of five cities was chosen because the ratio of population per distance travelled was the highest possible in any combination of five USA cities.

Furthermore, the app uses MapQuest’s Route Matrix service, which pulls data faster than MapQuest’s directions service. The Route Matrix service halves the time for a five city problem, and it was reduced by a factor of five for an eleven city problem. One factor that was looked at was the time the app takes to load city data and to solve the answer. This was investigated as a function of the number of cities input, and the graph, shown in figure 7, shows a linear increase in runtime as problem size increases. This result does not match the theoretical speed of the brute force algorithm, which is roughly $n$-factorial. This is because the runtime below includes the time to load the city data. The time it takes to solve the TSP is insignificant when compared to the time to load the distance data, and so the exponential increase in time to solve the TSP is masked by the time it takes to load the distance data.

The app is an object-oriented program, and two objects were settled on: a City object and a Solution object. The City object contained the name, population, hospital name, waterway name, and airport name. The Solution object held the total population, total distance, and list of cities in a route. These objects made code concise and easy to read.

5. Conclusions:

In creating an Android app to solve the zombie apocalypse version of the TSP, a firm understanding was grasped of the many interactions and complexities that takes place among a team of engineers. Both system and software engineers play very different, but equally vital, roles when working
together. An understanding was gained of android development and experience in software design.

5.2 Improvements

One possible improvement would be finding a better algorithm to accommodate for a higher number of cities in the app. If more than 15 cities are added to the current program, the run time would be too long for the user to realistically utilize the app. Any algorithm that has a run time smaller than $n$-factorial would run faster than the current algorithm.

This algorithm could also account for a possible die-off rate. This would take into account the fact that the population of a certain area would change as time goes on, rather than stay at the same number as before the apocalypse started, decreasing at a certain rate. This would make the results of the algorithm more realistic.

Another way that the app could be improved could be using traveling distances while both flying and driving. Being able to toggle between the two would make the app more versatile in usage as it could take into account the availability of multiple transportation methods.

Acknowledgments

We gratefully acknowledge the members of Lockheed Martin, including Amanda Weissman, Janet Mekker, David Coar, Eddie Kahn, Lee Mahoney, Joe Mirizio and Joe Grossmann for taking the time to guide us through the project and further teach us about engineering. We also thank our Residential and Technical Assistant, Jon Scheaffer, the Assistant Director of the Governor’s School of Engineering and Technology, Jean Patrick Antoine, the Governor’s School Program Director, Dean Ilene Rosen, and the head RTA, Tara Nealon, for advising us. Lastly, we would like to thank the following sponsors of the 2013 Governor’s School of Engineering and Technology for making this opportunity possible: Rutgers University, the State of New Jersey, Morgan Stanley, NJ Resources, South Jersey Industries, PSE&G, and the GSET Alumni Community.

References


Appendix

<table>
<thead>
<tr>
<th>Safe Zone City</th>
<th>Population (million)</th>
<th>Major Hospital</th>
<th>International Airport</th>
<th>Nearest Body of Water</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston, Massachusetts</td>
<td>10.10</td>
<td>Boston Medical Center</td>
<td>Logan International Airport</td>
<td>Charles River</td>
</tr>
<tr>
<td>Newark, New Jersey</td>
<td>26.85</td>
<td>Saint Michael’s Medical Center</td>
<td>Newark Liberty</td>
<td>Passaic River</td>
</tr>
<tr>
<td>Detroit, Michigan</td>
<td>9.50</td>
<td>Detroit Receiving Hospital</td>
<td>Coleman A. Young</td>
<td>Detroit River</td>
</tr>
<tr>
<td>Pittsburgh, Pennsylvania</td>
<td>6.04</td>
<td>UPMC Presbyterian Emergency</td>
<td>Pittsburgh International Airport</td>
<td>Allegheny River</td>
</tr>
<tr>
<td>City</td>
<td>Distance</td>
<td>Department</td>
<td>Airport</td>
<td>Ocean</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------</td>
<td>---------------------------------</td>
<td>-----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Ontario, California</td>
<td>16.88</td>
<td>Corona Regional Medical Center</td>
<td>Ontario Airport</td>
<td>Pacific Ocean</td>
</tr>
<tr>
<td>Orlando, Florida</td>
<td>6.63</td>
<td>Arnold Palmer Hospital</td>
<td>Orlando Airport</td>
<td>Atlantic Ocean</td>
</tr>
<tr>
<td>Charlotte, North Carolina</td>
<td>4.71</td>
<td>Carolinas Medical Center</td>
<td>Charlotte-Douglas International Airport</td>
<td>Lake Wylie</td>
</tr>
</tbody>
</table>

Code For Solving TSP for N cities

```java
public Solution shortestOfNCities(int[][] distances, ArrayList<City> cities, int[] combination) {
    Solution bestSolution = new Solution();
    double score = 0;
    int minPathDistance = Integer.MAX_VALUE, totalPopulation = 0;
    int[][] permutations = generatePermutations(5);
    for (int i = 0; i < permutations[0].length; i++) {
        totalPopulation += cities.get(combination[i]).getPopulation();
    }
    for (int i = 0; i < permutations.length; i++) {
        int sum = 0;
        for (int j = 1; j < permutations[0].length; j++) {
            int from = combination[permutations[i][j - 1]];
            int to = combination[permutations[i][j]];
            sum += distances[from][to];
        }
        if (sum <= minPathDistance) {
            minPathDistance = sum;
            score = totalPopulation / (double) (minPathDistance);
        }
    }
    return bestSolution;
}
```
bestSolution.setScore(score);
bestSolution.setDistance(minPathDistance);
bestSolution.setPopulationSave(totalPopulation);
int[] route = new int[5];
for (int j = 0; j < permutations[0].length; j++) {
    route[j] = combination[permutations[i][j]];
}
bestSolution.setOrder(route);

return bestSolution;