Omniscient Traffic Lights: Using Smart Traffic Lights to Improve Collective Traffic

by

Naman Jain

A thesis
presented to the University of Waterloo
in fulfillment of
the project requirement for the course ECE499
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2022

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Access to the project repository:
https://github.com/n27jain/CrowdNavigationResearch

Abstract

When designating traffic lights to have predetermined timing, counties do not effectively manage traffic. Similarly, without the knowledge of traffic light timings, drivers are unaware of what speed they should travel at to reduce their overall trip time and or fuel consumption. While shortest trip planning techniques have been well established, the inclusion of traffic light timings in both trip planning and traffic management has not been executed.

This project creates a roadmap of random speed limits and set traffic light timings. With this model, the aim of the thesis is to create and prove that a system exists that will help reduce the trip time and idle fuel consumption of a traveller given a fixed path. The thesis aims to explore the optimal solution and its practicality.

For the second part of this project the aim is to determine the timing of traffic lights in a road network provided the information of the travellers on the road and their paths. The goal is once again to reduce the overall trip time and fuel consumption. Knowledge of various competitors entering the traffic network can allow for a system that determines traffic light timings to ensure the optimal balance for the shortest combined trip time for participants and their fuel consumption, as opposed to improvement of only a select few competitors benefiting from their predetermined route. This thesis aims to reduce collective trip time and idle fuel consumptions of travellers in a road network system by adjusting the timing of the traffic lights dynamically to better service the collective traffic.

Commented [SM1]: I feel like mentioning this here is repetitive, especially since you say it again at the end of the paragraph

Table of Contents

List of Figures

List of Tables

1 Introduction

For the last few decades, the goal of optimizing road travel has been of great interest to municipalities and businesses. Reducing the time of travel increases the economy's efficiency, as more tasks can be completed in a day when less time is consumed in travelling. Traffic congested cities often harm the mental and physical health of drivers that travel them. While shortest path finding algorithms exist, they do not consider the timings of traffic lights and the time penalty taken each time a car is stopped at a traffic light. If a traveller does not know what speed limit to travel at, they may encounter greater delays waiting at traffic lights and waste a lot of fuel idling. Additionally old school traffic light systems still depend on manual intervention for determining the traffic light timings. This means that should there be a disturbance introduced to the traffic system, there would be great delays before the issue is resolved. Therefore, the introduction of traffic light timings should be considered when conducting trip planning on county roads.

This chapter will explain in detail the problems of trip planning, the motivation for conducting the project, and the objective of the project.

1.1 Overview and Motivation

There have been many great solutions presented to help reduce travelling distance, and trip time with traffic congestion as a consideration. Many smart devices such as mobile phones and cars all have some sort of navigation system useful for directions to travel to any destination quickly. With its growing popularity apps like Google Maps and Apple Maps are being used by a large portion of the North American population. But taking these maps for granted may not be the best approach for traveling. Some traffic lights are poorly planned where if a driver travels at

the posted speed limit they may arrive at multiple red lights before arriving to their destination. This can be due to many factors such as unexpected traffic congestion, or outdated technology. Idling on red lights is frustrating for a traveller and introduces a wastage in fuel consumption. Therefore, a need exists for calibrating the traffic lights based on expected demand of travellers in a road network and the ability to forecast the speed a traveller should travel to achieve optimal travel time and save the maximum amount of fuel. When such a solution is developed, it can contribute to developing a Smart City where traffic is dynamically managed. Therefore, the main motivation for this project is to test and prove that such a solution exists so that travel time and fuel savings can be maximized for the betterment of mankind.

1.2 Research Objectives

The project has three goals:

1.) Consider what a realistic road network simulation is
2.) Consider what speed a traveler must travel at on each road along their path to achieve a short trip time and reduce idle fuel wastage
3.) Consider what traffic light timings should be fixed to cater for a set of travellers and their paths

These goals can be achieved by the following objectives:

1.) Perform research on speed limits, traffic light timings, fuel consumption, and methods to determine how to assign importance to time duration vs fuel consumption.
2.) Develop a model map that simulates a realistic road system with traffic lights and roads

3.) Develop a random path generator for a random travel that travels an explicit amount of intersections/roads
4.) For each path determine what speed limit the traveller must travel at on each road to reduce their trip time and fuel consumption using Genetic Algorithm
5.) Determine the best parameters in the algorithm to test on a generic path by experimenting with each parameter value
6.) Develop a random traffic light generator to randomly assign traffic light times to a road network to be used in a simulation
7.) Provided many travellers and their paths, determine the best traffic light timings for traffic lights so that the travellers collective trip time and fuel consumption is minimized using GA
8.) Using the formulated solution, models, and simulations prove that by utilizing traffic lights in trip planning, the potential for saving costs and time is realized.

1.3 Organization of Thesis

The thesis contains the following sections:

Section 1: Contains the introduction, motivation, and objectives of the project

Section 2: Provides background knowledge and research done to help develop the theory

Section 3: Presents the problem of shortest trip planning and lowest idle fuel consumption and then provides the solution. Then the experiment is conducted, recorded, and analyzed.

Section 4: Presents the problem of adaptive traffic light system and then provides the solution. Then the experiment is conducted, recorded, and analyzed.

Section 5: Contains the final thoughts and conclusion of the project, as well as improvements for a future project. Limitations of the current project is also discussed.

2 Background and Literature Review

2.1 Introduction

For this project there are many areas of research that need to be elaborated. This section will discuss key concepts that help aid in understanding and formulating the models and simulation for this project.

2.2 Traffic Light Timing

To determine traffic light timings there are strict guidelines. A cycle time is how long one traffic light in an intersection spends in green, yellow, and red state. All four traffic lights at the intersection share the same cycle time. The typical cycle time of a traffic light is between 30 to 120 seconds [1]. So, for the traffic light model, we will use this metric to generate sample traffic light cycle times.

Yellow Light timing is determined based on how fast the speed limit of the road is. If a road has a high expected approach speed, that traffic light must accommodate for this by raising the traffic light time accordingly. The table below is taken from the sample data shared by the Oregon State University in USA [1].

Table 2. Yellow Change Intervals

Approach Speed	Yellow
<35 mph	3.0 sec.
35-40 mph	3.5 sec.
40-45 mph	4.0 sec.
45-50 mph	4.5 sec.
>50 mph	5.0 sec.

Figure 1: Oregon State Table for Traffic Light Yellow Timings

This table is used as a reference for the traffic model designed. This may need to be modified based of the rules of the Country, State, Province or Municipality, respectively, for any experiment conducted for the respective jurisdiction.

Green light timings in our model can vary based on how the simulation is desired to be set up. A good metric is determining what percent of the cycle time is green by determining which light governs a larger road. The larger the road, the more time it should be allocated as green. This will be discussed in more detail in 3.2 Problem Formulation. This information is used in the code MapObjects.py.

2.3 Deciding Speed Limit on Simulated Road Network

According to the Ontario Highway Driving Act, section 132, a vehicle cannot drive too slowly so that it would affect traffic unreasonably unless the lowered speed is taken for emergencies [2]. Therefore, in our model, if the max possible speed of the car is over 30km/h, the simulation will

not randomly select a speed lower than 30 km/h. This information is used in the code MapObjects.py.

2.4 Pareto optimality

The Pareto optimality is the concept of finding superior solutions in an objective space with two or more variables where no improvements can be made without sacrificing one of the objectives [3]. In our scenario we will be weighing the two factors: time, and fuel consumption. Ideally, the average person wants to spend minimal time to get to their destination but may be willing to wait a bit extra if it means they will see significant cost savings and reduce their carbon emissions. This is subjective however, so in this experiment we will generalize a trend.

A measurable comparison can be made if fuel savings on a trip is comparable to the average hourly wage of a person in a county. For this experiment we will consider the average hourly rate of a person living in Ontario, Canada. For a male that is $32.47 and for a female that is $28.90 [4]. So, on average an Ontarian makes $30.69/hour.

Let's assume that a person will find it more desirable to increase their trip time if it means fuel saving's compensate the person 50% of their hourly wage. This means if

$$fuel * cost\ per\ liter \geq 0.5 * extra\ time\ in\ minutes * average\ wage\ per\ minute$$

or

$$fs * \frac{\$}{L} \geq 15.345 * w$$

the new solution is better for the driver. So, for example, if the driver takes an extra 1 minute in an alternative path, they should save $0.255 dollars or more for it to be worth their time.

Petro Canada is the largest petrol distributor in Canada. There mid-range fuel prices as of April 2021 is $1.809 / Liter [5].

Commented [SM2]: is this supposed to be one of OUR/THE objects?

Commented [SM3]: Add comma after ontario

Commented [SM4]: I don't think minute should be shortened in a sentence

So

$$fs * 1.809 \geq 15.345 * w$$

Using this information, we can determine how valuable a solution is based on how much fuel and time is saved following the simulation results.

2.4.1 Determining Fuel Consumption

According to South Carolina Department of Health and Environmental Control, an average car wastes ½ gallon (1.89 L) of fuel per hour while the car is idling [6]. We will use this statistic to determine fuel consumption in our problem formulation.

Commented [SM5]: an

Business insider states that the average car get around 25 miles per gallon mileage [7]. This means that on average, a car gets about 40.3 kilometers of travel per 4.55 liters of fuel. Or for every kilometer, the car consumes 0.113 liters. This will be a bit of a crude estimate since this value is derived from a high average of speed travelled throughout a path and our simulation may decide a path that has an overall low speed average further reducing the milage of the car. But the effects of this are very minimal since this amount of fuel could not be saved assuming that the shortest path is provided to the traveller.

For our model we need not consider the total fuel consumption but rather the wasted fuel by idling. A more robust model may consider fuel consumption and fuel economy based on speed limits, and vehicle fuel ratings. But this is outside the scope of this project.

3 Theoretic Solution for Shortest Trip Time & Lowest Idle Fuel Consumption Planning

3.1 Introduction

Traffic congestion with unplanned traffic lights is a big issue, and sometimes travelling at the speed limit (or exceeding it) increases travel delay due to the time wasted at red lights. Waiting at traffic lights increases travel anxiety and wastes fuel. So when provided a path and the knowledge of traffic light timings, this theoretical solution aims to determine the average speed that the traveller should travel on each road of their path.

3.2 Problem Formulation

Consider a traveller that travels the paths, $E = \{e_1, e_2, \dots, e_N\}$, to go from its starting point to its destination.

Each path (or edge) will have the following properties:

Speed Limit S_i

Distance D_i

Direction Dr_i

Traffic Coefficient T_i

Node start n_1 and node end n_2 (for west to east, and south to north).

where $i \in N$

Then for a network of nodes $N = \{n_1, n_2, n_z\}$ each node will have the following properties:

Commented [SM6]: some i's are subscript while others are not, idk if this is intentional or accidental

Traffic light cycle time = C (randomly generated from 30 – 120 seconds)

Red light time, $RT = \{RT_0, RT_1, RT_2, RT_3\}$ one for each direction at the intersection.

Yellow light time, $YT = \{YT_0, YT_1, YT_2, YT_3\}$ one for each direction at the intersection.

YT is determined by incoming speed at the node (Traffic Light Timing).

Commented [SM7]: is there a reason this one line isn't centered

Green light time, $GT = \{GT_0, GT_1, GT_2, GT_3\}$ one for each direction at the intersection.

Red light time, $RT = \{RT_0, RT_1, RT_2, RT_3\}$ one for each direction at the intersection.

Edges distances $Ed = \{\mathrm{Ed}_0, \mathrm{Ed}_1, \mathrm{Ed}_2, \mathrm{Ed}_3\}$ 0 – East, 1 – West, 2 – South, 3 – North

$ew_{\%g}$ is the percent of the cycle that the East and West Lights are green.

GT and RT are found by edge distances.

$$ew_{\%g} = \frac{\max(Ed_0, Ed_1)}{\max(Ed_0, Ed_1) + \max(Ed_2, Ed_3)}$$

$$GT_0 = GT_1 = C * ew_{\%g} - YT_0$$

$$RT_0 = RT_1 = C - GT_0 - YT_0$$

$$GT_2 = GT_3 = C * (1 - ew_{\%g}) - YT_2$$

$$RT_2 = RT_3 = C - GT_2 - YT_2$$

3.2.1 Generate the map

Now a map must be generated to test rigorously on. To generate this map, a random number of nodes must be generated, and a percentage of possible edges should be established. Consider a 3x3 node system. It can have up to 11 edges. So, to generate a random map, a new set of nodes that are active are selected randomly until a certain amount is made. In this model, 50% of all nodes possible will be active and connected. Then those nodes connect with one another on the x and y axis.

3.2.2 Traffic Condition Coefficient

To generate the traffic condition, a factor is selected randomly by Gaussian distribution from 0 to 1.

3.2.3 Speed limit Generation

To generate a maxim speed limit for each edge, a random number generator selects this. True maximum allowable speed is equivalent to the product of the traffic coefficient and the road speed limit. If the max allowable speed of the car is over 30km/h, this model will never allow a edge go below 30 km/h in traveling speed. If the allowable speed is less than 30km/h it is assumed at this is due to poor traffic conditions where it may be beneficial to drive slower than what is allowed. After a path defined with edges is established, the model must iterate through each edge and determine the maximum allowable speed for each edge.

3.2.4 Setting Traffic Light Timings

Derivation of traffic light timings is discussed in 2.2 Traffic Light Timing. Each traffic light intersection has a cycle. Typical cycle lasts from 30 to 120 seconds. The cycle time is chosen randomly using a uniform number distribution. Yellow light time is determined based on the speed limit of the road approaching the traffic light. Green light time is determined by the length of the road. This may not be the best measure to use, but this is sufficient for this simulation as the Adaptive Traffic Light system will aim to fix the lights without any concrete metric.

3.2.5 Creating a Path

Here are the steps for creating a path for a car to travel on for the purpose of this theoretical map:

1.) a random node will be selected as a start position

2.) A random direction will be chosen until such an edge exists that it is possible to travel in that direction.

3.) Then repeat steps 1-2 until a fixed number of desired steps for the path is established

4.) Should a dead-end node arrive before the number of desired steps is taken, a backtracking to the previous node will need to be done. If back tracking returns to the start node, return an error and run the steps again from the beginning.

The code is accessible in CreatePaths.py.

Why this works – Since the shortest path problem has already been solved by other engineers, it does not matter what path the traveller takes since the goal is to provide the best trip possible from whatever their existing path is by only adjusting the speed of the traveller when they follow that pre-determined path. In future projects perhaps inclusion of traffic lights may deduce that the traveller should take an alternative path. For example, consider if two paths exist where one is deemed to be 2 minutes faster than the other without any knowledge of traffic lights in that network. It is possible that the route has poorly planned traffic lights which leads to the alternative path being faster.

Commented [SM8]: comma after node

3.2.6 Movement with Direction (Penalties and Logic)

These are the responses to various actions possible on an intersection.

Right turn – Ignore traffic rules of current edge and proceed to add a fixed average time to take a right turn. It is very difficult to determine the exact time it takes to make a right turn on average and such data was not available. In future, it would be ideal if some live sample data could be taken. For this project right turns are accepted to have no delay because most roads allow right turns on a red light. Since this is a non-cooperative problem, there is no way of knowing if there is incoming traffic in the direction of the right turn.

Left turn – May turn when facing traffic light is green, add a delay of 5 seconds if the light was green on arrival of the car at the intersection. This is because usually left turns are not allowed after a certain time after the green light on some intersections. Additionally, those that allow it can be busy so on average a 5 second idle penalty seems reasonable.

Forwards – May proceed only if the facing traffic light is green. Add a delay for starting the car up after stopping on red light. The penalty is set to 5 seconds just like for the left turn. This is to account for the delay in return the car back to its original travel speed. This is a crude estimate because some intersections are smaller than others, and other intersections may be blocked due to traffic on the other side that has not moved.

3.3 Solution

3.3.1 Introduction

The goal is to create a list of average speed per edge to travel. So, for example, if the traveller travels through 5 edges, then the solution would be $X = \{S_1, S_2, S_3, S_4, S_5\}$.

While there are various methods for solving this problem, the method that used in this project is the Genetic Algorithm (GA), which uses the survival of the fittest methodology to

Commented [SM9]: maybe say “is the Genetic Algorithm”?

iterate over a population of solutions that change in values over time, going through Roulette selection, Crossover, and Mutation for creation of the next population to test fitness.

When the solution is used to run through the simulation, the output will be the fitness of the solution. This is determined by the time duration and fuel consumption. The initial solution will set all the S_i values to the maximum speed that the traveller can travel at. The following iterations will seek to improve the output of the initial solution.

Commented [SM10]: no comma

3.3.2 Theory

3.3.2.1 Trip Time

For GA, the input will be the X solution set.

The equation for optimization is derived as follows:

For trip time:

$$\min(Z) = t_1 + t_2 + t_3 \ldots + t_N + t_{d_1} + t_{d_2} + t_{d_3} \ldots + t_{d_N}$$

Here, t_i is the time taken to travel through edge i. N is the number of edges travelled.

t_{d_i} is delay time that must be taken depending on various penalties expected due to edge i :

1.) If the traveller arrives at an intersection with a red light and they want to go left or forwards; they must wait until the light is green and there is a 5 second delay for getting the car up to speed.
2.) If the traveller arrives at an intersection with a red light and the want to go right; this is no delay.
3.) If the traveller arrives at an intersection with a green light and wants to go left; they must wait a slight delay
4.) If the traveller arrives at an intersection with a green light and wants to go straight or right; there is no delay.

For simplicity, green refers to yellow light as well.

This problem is non-deterministic with many conditionals, which is why GA is used to solve the problem.

Here are additional conditions for this problem:

Commented [SM11]: see above

$$t_1 = \frac{D_{(A \to B)}}{V_{(A \to B)}}, t_2 = \frac{D_{(B \to C)}}{V_{(B \to C)}}, t_3 = \frac{D_{(C \to D)}}{V_{(C \to D)}}$$

Here, t_i is found by dividing the edge's distance (D_i) by the corresponding speed found in the input x_i.

These equations are conditional. The scenario bellow helps explain those equations.

3.3.2.2 Example Scenario (going forwards only)

Map:

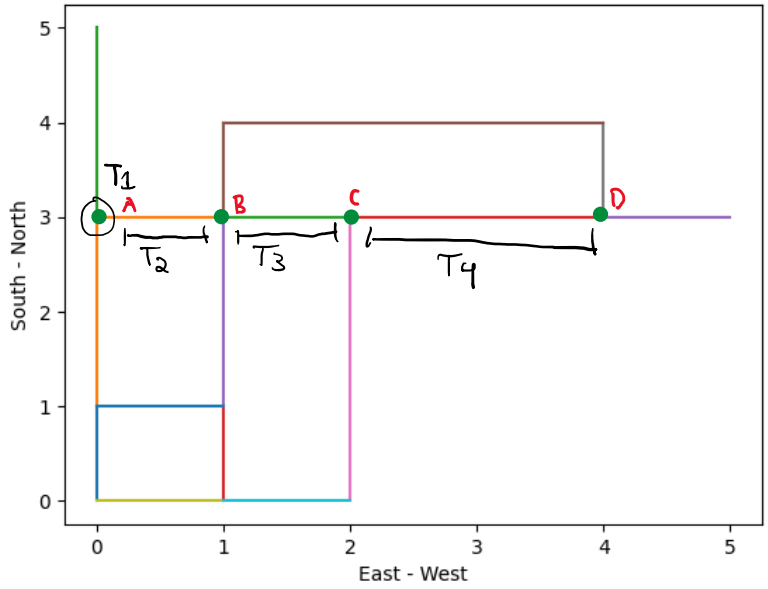

Figure 2: Example Scenario Map, Going Forwards Only

In the figure above, the green dots represent the nodes that will be visited. Point (0,3) is the start point; point (4,3) is the destination.

Assume the table below will show the data that was generated for this map:

Node	Time Green offset From t = 0	Tg	Tr	Cycle	Direction	Edge Movement
A 0,3	10	55.00	25.00	80	N	Forward
B 1,3	0	20.00	30	50	E	Forward
C 2,3	-10	30.00	50.00	80	E	Forward
D 4,3	-10	x	x	71.6	x	Destination

Table 1: Example Data for Example Scenario

The Time Green offset from t = 0 is used to indicate whether the light is green or red at t=0. In the example above, A is 10 seconds away from being green. B just turned green, and C and D have been green for 10 seconds already.

To avoid hitting a red light at point B,

$$t_i > T_{r_B} - O$$

Where O is the offset. This is to avoid arriving at the light to early.

But,

$$t_i \leq C_B - O$$

So that the car does not arrive too late. However, this is not the only case scenario. There are an infinite number of intervals that can be found as follows:

$$-O + T_{r_B} + n * C_B < t_1 \leq (n+1) * C_B - O \; \forall \; n \in \mathbb{N}$$

And the equation above holds for all lights that are initially red ($O > 0$).

When a light is initially green $O \leq 0$, at point C:

$$n * C_C + O < t_1 + t_2 \leq (n) * C_C + T_{g_C} + O \; \forall \; n \in \mathbb{N}$$

But for cases where this is the first light

$$0 < t_1 + t_2 \leq (n) * C_C + T_{g_C} + O \; \forall \; n \in \mathbb{N}$$

The time taken to get to point B and arrive at point C must be met to avoid the red light.

To generalize:

When $O > 0$ at node p and t_{total} is the total time traveled to arrive at p:

$$-O + T_{r_p} + n * C_p < t_{total} \leq (n+1) * C_p - O \; \forall \; n \in \mathbb{N}$$

When $O \leq 0$ at node p:

$$n * C_p < t_{total} * C_p + T_{g_p} + O \; \forall \; n \in \mathbb{N}$$

So, in summary the general optimization problem looks like the following:

$$\min(Z) = t_1 + t_2 + t_3 \ldots + t_N + t_{d_1} + t_{d_2} + t_{d_3} \ldots + t_{d_N}$$

$$s.t$$

$$-O + T_{r_p} + n * C_p < t_{arrival} \leq (n+1) * C_p - O \; \{\forall \; n \in \mathbb{N} \; , \forall \; p \; \in P^g\}$$

$$n * C_p < t_{arrival} * C_p + T_{g_p} + O \; \{\forall \; n \in \mathbb{N} | \; \forall \; p \; \in P^r\}$$

$$t_{d_N} = set\ delay + time\ remaining\ until\ cycle\ is\ complete$$

$$if\ \ t_N\ does\ not\ meet\ the\ contraints\ above$$

$t_{arrival}$ changes based on the previously travelled edges.

N is the index of the traffic light encountered (1 - # of total lights).

So for our example map,

$$(1)\quad \min(Z) = t_1 + t_2 + t_3 + t_{d_1} + t_{d_2} + t_{d_3}$$

$$s.t$$

$$(2)\ t_1 = T_r - O$$

$$(3)\ n * C_B + O < t_1 + t_2 * C_B + T_{g_B} + O\ \ \{\forall\ n \in \mathbb{N}\}$$

$$(4)\ n * C_C + \ O < t_1 + t_2 + t_3 * C_C + T_{g_C} + O\ \ \{\forall\ n \in \mathbb{N}\}$$

$$(5)$$

$$t_2 = \frac{D_{(A \to B)}}{V_{(A \to B)}}, t_3 = \frac{D_{(B \to C)}}{V_{(B \to C)}}, t_4 = \frac{D_{(C \to D)}}{V_{(C \to D)}}$$

(2) the first-time interval is waiting at the traffic light

(3) t_2 is the time spent going from point A to C under the influence of t_1

(4) t_3 is the time spent going from point C to D under the influence of $t_1 + t_2$

(5) time is found by dividing Distance over Speed. Notice t_4 is not influenced by anything else

Commented [SM12]: I don't get this phrase, but maybe I just don't have context. Just pointing it out in case you misspelled something

3.3.2.3 Fuel Consumption

Optimization equation:

$$\min(Z) = f_1 + f_2 \ldots + f_N + f_{d_1} + f_{d_2} + \cdots + f_{d_N}$$

f_i is the fuel cost for traveling edge i, and f_{d_i} is the extra fuel cost for idling at an intersection.

While the time calculation is very complex, the fuel consumption is relatively simple.

Given the speed travelled at an edge i, the fuel consumption on the edge in liters is:

$$f_i = (D_{start \to end}\ km) * 0.113 \frac{L}{km}$$

For each time the car is idle (i.e waiting for a left turn or waiting on a red light):

$$f_{d_i} = 1.89 \frac{L}{3600s} * t_{d_i}$$

3.3.3 The Algorithm

1.) Create a random path to test on the randomly generated map

2.) Create an initial solution X that sets the speed travelled at each edge to its maximum speed possible

3.) For each edge determine the time taken by following the constraints above

4.) For each edge determine the fuel consumption by following the theory above

5.) Get the base output for the solution

6.) Run the simulation again (Steps 3-5) but with 50 randomly generated solutions:

   a. These solutions must follow the speed limit

b. If the maximum speed is greater than 30, the vehicle will be restricted to speeds greater than 30

c. Otherwise, the speed can be set to any value under the maximum speed.

7.) Once the output of the tests is complete, calculate the score of the solutions as follows:

$$\Delta T = T_{total_0} - T_{total_i}$$

$$\Delta F = f_{total_0} - f_{total_i}$$

Where ΔT is the difference of time from the initial solution with the new solution and ΔF is the difference in fuel consumption from the initial solution with the new solution.

$$score = \Delta F * 1.809 - 15.345 * -\Delta T$$

Note: score can be negative. This is done so that certain solutions can be explored even if they are not improving the solution. Otherwise, the solution might converge at a suboptimal solution. For the base case solution, the score will be 0.

8.) Using the scores find the relative fitness of each solution.

First, all scores need to become positive. To achieve this, we search for the smallest score and subtract it from all scores.

$$relative\ score_i = score_i - \min(all\ \ scores)$$

Then to get relative fitness

$$fitness_i = \frac{relative\ score_i}{\sum all\ scores}$$

9.) Then store the best 2 solutions. For the remaining 48 solutions select a solution using the roulette selection method. The set of solutions will be sorted by their fitness level

Essentially, there will be a random number generated:

```
check = round(random.uniform(0, 0.99999999),8)
```

and a loop will increase a counter by the value of each fitness until a value is reached that is greater than the randomly generated number. Then, whatever solutions fitness value was used last to increase the counter value will be selected into the survivors list.

10.) Take the survivors from step 9 and apply a crossover.

Crossover is done by using the probability crossover ratio or pc. Essentially a random number generator decides if a chromosome will be crossed over. If the randomly generated number is $\geq pc$ then it will be selected for crossover. This is done for every chromosome and a list of swapped crossover chromosomes is established. That is, the first chromosome determined to be crossed over will crossover with the second chromosome determined to be crossed over, the second will cross over with the third, until finally the last chromosome will swap with the first. Afterwards a crossover point needs to be determined. The number value range is from index 1 to the length of the chromosome – 1.

```
round(random.uniform(1,len(self.X[0][0]) - 1),0)
```

Then based off this number the crossover point is established, and all values of the chromosome that occur at or after this crossover point will be swapped with the next chromosome. This will generate a new list of solutions.

11.) Take the solutions from step 10 and apply mutation

Mutation is done by using the probability mutation factor or pm . For each chromosome and its individual index, run a random number generator. If that number is $\geq pm$ then randomly select that chromosome's index's value as described in step 6. Mutating a value

means to replace it with a new random value from the set of acceptable values. In this case that would be the average speed travelled which is constrained as mentioned in the problem formulation.

12.) Repeat steps 6-11 by the number of $generations$ chosen.

This algorithm is accessible in GeneticAlgorithm.py

3.4 Results & Experimentation

The Simulator.py contains various tested scenarios.

1.) Create a new map with $n * m$ dimensions

2.) Create p paths with length l

3.) Store the data in photo format as well as an accompanied text file for clarity

4.) Access stored data to perform computations on them

5.) Print results in the terminal or a text file

3.4.1 Hypothesis:

For a path with a smaller l , solution will converge early because there are limited variations. The best solution will not stray far from the base case as there are limited penalties to be had with limited number of interruptions on traffic lights.

No variation in the $pm, pc, population, generations$ will significantly improve discovery of the optimal solution.

Commented [SM13]: Period after this?

For a path with a large l, the solution will converge later due to may variation.

pm, pc, and $genereations$ may affect the solution.

3.4.2 Results:

Experiment 1: Test Small Paths

This experiment will have an 8x8 map with five travellers each following a path length of size three edges.

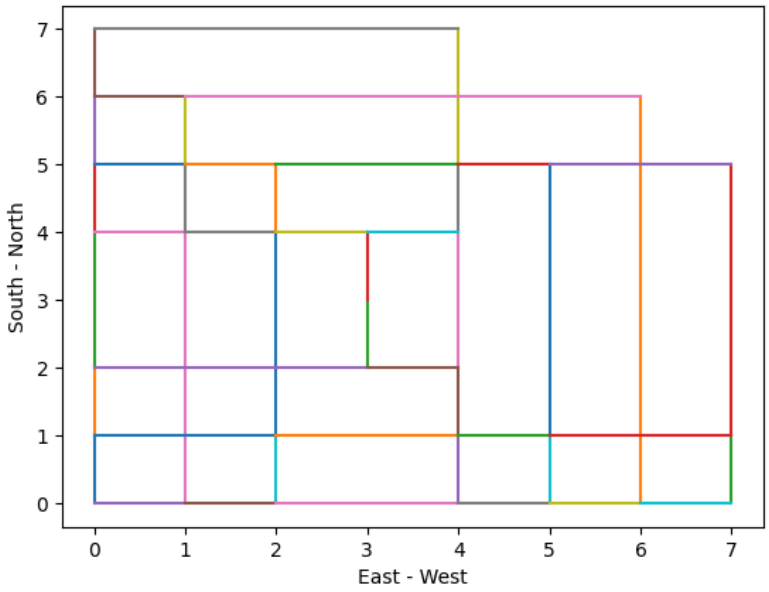

Figure 3: 8x8 Map

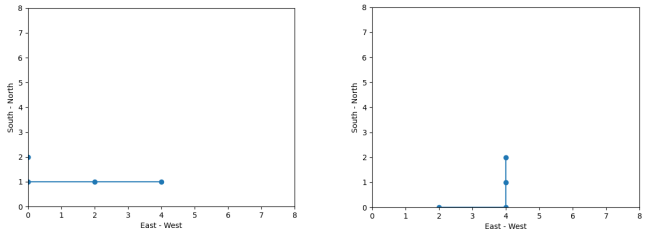

Figure 4: Paths 0-1

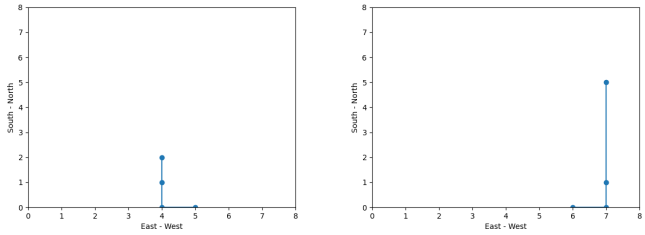

Figure 5: Paths 2-3

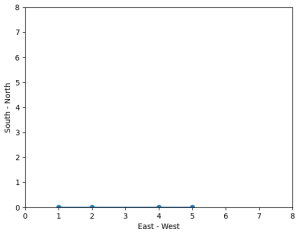

Figure 6: Path 4

Base Case:

```
population=50
generations = 100
pc = 0.6
pm = 0.4
```

Path 0:

Base Sol: [70.0, 30.0, 80.0], time = 359.0 seconds, extra fuel cost: $0.01522

Best Sol: [70.0, 29, 80.0], time = 359.0, extra fuel cost: 0.010880172413793106

The best solution converged in 6 generations and by reducing the average speed on the second edge from 30km/h to 29km/h, there is almost a negligible fuel savings difference of 0.0043 dollars. So, the number of generations should be reduced to save on computation time.

Path 1:

Base Sol: [40.0, 30.0, 11.0], 755.5454545454546, 0.005775

Best Sol: [37, 14, 11], 751.8427518427519, 0

The best solution converged in 1 generation and now there is no fuel wastage when the speed is reduced for the second edge.

Path 2:

Base Sol: [40.0, 30.0, 19.0], 290.47368421052636, 0.005775

Best Sol: [37, 14, 19], 286.77098150782365, 0

The best solution converged in 1 generation and now there is no fuel wastage when the speed is reduced for the second edge.

Path 3:

Base Sol: [28.0, 30.0, 50.0], 212.0, 0.010499999999999999

Best Sol: [4, 18, 50], 72.0, 0

Same results as Path 2, but a noticeable significant improvement

Path 4:

Same results as 2-3

Population Variation:

Reduce Population Size:

```
population=20
generations = 100
pc = 0.6
pm =0.4
```

Results summary:

Path 0:

Best Sol: [70.0, 29, 80.0]

Same as Base Case but took more generations to achieve the same result.

Path 1:

[37, 20, 11] time = 751.8427518427519, fuel cost = 0

A new solution is found but yields the same results. Basically, the speed distribution between the second and last edge is changed to reach the same result.

Path 2:

Same as path 1

Path 3:

Same as path 1

Path 4:

Same as path 1

With a reduced population size, we discover that more generations are needed to get the same result, and there are multiple optimal solutions that will yield the same score.

Increase Population Size:

```
population=100
generations = 100
pc = 0.6
pm =0.4
```

Early convergence discovered. No effect on the solution.

Generation Variance:

Decreasing the number of generations to 50 had no effect since convergence was early.

Increasing the number of generations to 150 had no effect.

pc Variance:

Decreasing the pc to 0.2 did not improve the solutions. In fact, because the probability of crossover was so low, new solutions were not being discovered, and performance was poor.

Consider path 0. The optimal solution was discovered in 6 generations where in this run it took almost 30 generations.

Increasing the pc to 0.8 did not improve the results. Convergence was faster as the best solution was exploited earlier.

pm Variance:

Decreasing the pm to 0.2 had no effect on the simulation.

Increasing the pm to 0.6 had little effect on the simulation. The optimal solution convergence took more generations then when the pm is 0.4 or less.

Experiment 2: Test Large Paths

This experiment will have use the same 8x8 map but with ten travellers each following a path length of six edges.

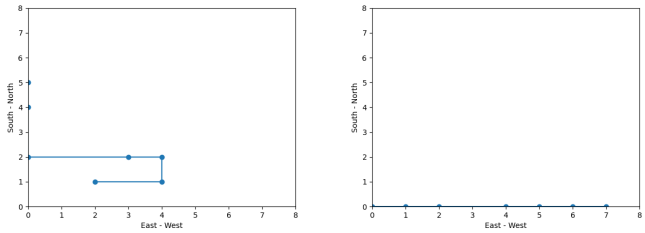

Figure 7: Paths 0-1

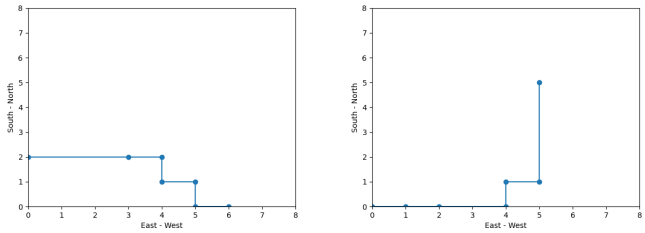

Figure 8: Paths 2-3

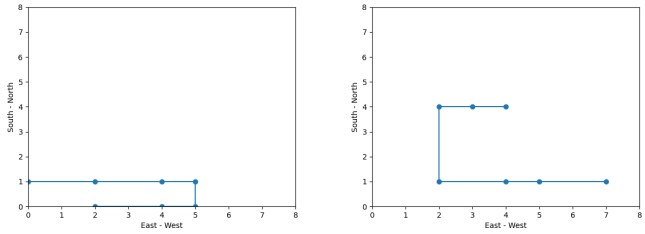

Figure 9 : Paths 4-5

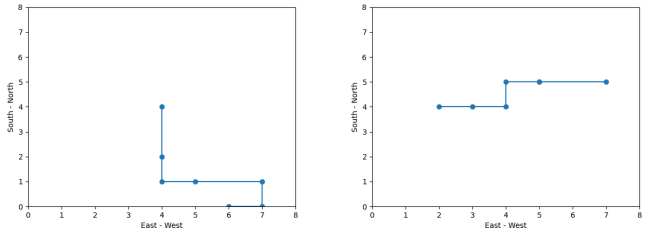

Figure 10: Paths 6-7

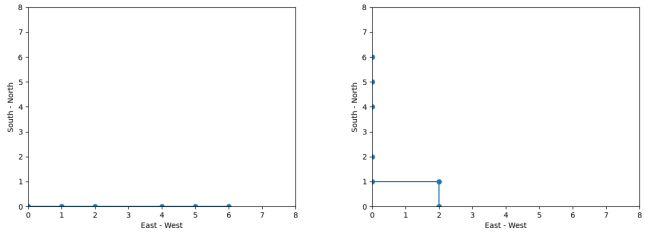

Figure 11: Paths 8-9

Base Case:

```
population=50
generations = 100
pc = 0.6
pm =0.4
```

Summary Results:

```
Base Case:
Path 0 :
 BASE : [[70.0, 69.0, 36.0, 3.0, 40.0, 80.0], 1886.0, 0.025717391304347824, 39373.446196688266, -1]
  BEST : [[57, 30, 36, 2, 39, 80], 550.4655870445343, 0.0026249999999999997, 59867.26353762575,
0.023946446126853985]
 GEN : 79

Path 1 :
 BASE : [[38.0, 50.0, 11.0, 19.0, 30.0, 50.0], 644.0, 0.050289473684210495, 109401.71543029898, -1]
  BEST : [[32, 38, 6, 19, 4, 50.0], 72.0, 0, 118179.14640395687, 0.023509554166173015]
 GEN : 17

Path 2 :
 BASE : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 124794.00569153203, -1]
  BEST : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 124794.00569153203, 0.021012892329062425]
 GEN : 0

Path 3 :
 BASE : [[57.0, 40.0, 30.0, 11.0, 50.0, 38.0], 1331.7368421052631, 0.025107057416267796, 104305.96477180802, -1]
  BEST : [[46, 40.0, 30.0, 11.0, 50.0, 38.0], 975.6156299840511, 0.008574999999999996, 109770.67467829988,
0.025731095173631571]
 GEN : 3

Path 4 :
 BASE : [[30.0, 80.0, 40.0, 60.0, 19.0, 11.0], 1263.019138755981, 0.015224999999999999, 112152.86876567807, -1]
  BEST : [[25, 57, 40.0, 60.0, 19.0, 11.0], 994.019138755981, 0, 116280.70130770307, 0.028786029035514204]
 GEN : 11

Path 5 :
 BASE : [[75.0, 49.0, 4.0, 80.0, 40.0, 13.0], 3658.846153846154, 0.050400000000000056, 171958.3540297939, -1]
  BEST : [[40, 49, 4.3, 73, 35, 13], 3269.476290832455, 0.01674892026578069, 177933.29545254234,
0.025955409241903347]
 GEN : 81

Path 6 :
 BASE : [[60.0, 40.0, 40.0, 13.0, 30.0, 50.0], 521.0, 0.015224999999999999, 112027.66527196066, -1]
  BEST : [[57, 40.0, 38, 7, 30.0, 50.0], 289.0, 0.0011881578947368385, 115587.73066460804,
0.021890760009567103]
 GEN : 14

Path 7 :
 BASE : [[60.0, 60.0, 15.0, 60.0, 75.0, 49.0], 571.469387755102, 0.01575, 59543.95245308891, -1]
  BEST : [[60.0, 45, 15.0, 60, 64, 47], 376.59574468085106, 0, 62534.316997813294, 0.023600522902309306]
 GEN : 40

Path 8 :
 BASE : [[38.0, 38.0, 50.0, 11.0, 19.0, 30.0], 437.0, 0.029151315789473664, 111971.28299518603, -1]
  BEST : [[35, 32, 35, 9, 19.0, 30.0], 120.0, 0, 116835.7007299163, 0.022836259259964794]
 GEN : 5

Path 9 :
 BASE : [[18.0, 70.0, 69.0, 70.0, 30.0, 70.0], 1076.2285714285713, 0.04082739130434784, 95778.89731988544, -1]
  BEST : [[5.7, 55, 69, 35, 29, 65], 487.9628489950828, 0.0076124999999999995, 104805.89491636568,
0.023615622574758487]
 GEN : 30
```

In the results above, the first value for a solution is the speed travelled array, the second is time duration, and third is fuel costs. The remaining values can be ignored as they are used for the GA. From these results we can see most solutions converge earlier than the 100 generations allocated. Path 0 had the highest generation amount needed of 79. So it is safe to assume that the base configuration will provide an optimal solution, but testing the variation in pm and pc may give a better solution than found in this base case.

```
Lower Population  = 20
Path 0 :
 BASE : [[70.0, 69.0, 36.0, 3.0, 40.0, 80.0], 1886.0, 0.025717391304347824, 38479.57704726031, -1]
  BEST : [[55, 57, 36, 1, 40.0, 80.0], 550.4545454545455, 0.0026249999999999997, 58973.56382139618,
0.059391220590442995]
 GEN : 78

Path 1 :
 BASE : [[38.0, 50.0, 11.0, 19.0, 30.0, 50.0], 644.0, 0.050289473684210495, 27344.776589597368, -1]
  BEST : [[31, 35, 8, 9, 11, 50], 72.0, 0, 36122.20756325526, 0.07423288568434266]
 GEN : 10

Path 2 :
 BASE : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 64307.08591151205, -1]
  BEST : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 64307.08591151205, 0.05276285815332576]
 GEN : 0

Path 3 :
 BASE : [[57.0, 40.0, 30.0, 11.0, 50.0, 38.0], 1331.7368421052631, 0.02510705741626796, 39138.354004671346, -1]
  BEST : [[42, 40, 30.0, 11.0, 49, 38.0], 975.6156299840511, 0.008574999999999996, 44603.06391116321,
0.06689228002298392]
 GEN : 7

Path 4 :
 BASE : [[30.0, 80.0, 40.0, 60.0, 19.0, 11.0], 1263.019138755981, 0.015224999999999999, 52213.931080125, -1]
  BEST : [[25, 78, 40, 60.0, 19.0, 11], 994.019138755981, 0, 56341.76362215, 0.06438698098801654]
 GEN : 34

Path 5 :
 BASE : [[75.0, 49.0, 4.0, 80.0, 40.0, 13.0], 3658.846153846154, 0.050400000000000056, 99439.29565420508, -1]
  BEST : [[38, 31, 4, 66, 35, 13], 3492.937062937063, 0.00728225806451619, 101985.24865420024,
0.06004361215234615]
 GEN : 53

Path 6 :
 BASE : [[60.0, 40.0, 40.0, 13.0, 30.0, 50.0], 521.0, 0.015224999999999999, 56254.76170240263, -1]
  BEST : [[47, 39, 38, 4, 30.0, 50], 289.0, 0.0011881578947368385, 59814.82709505, 0.0598753541557670 6]
 GEN : 21

Path 7 :
 BASE : [[60.0, 60.0, 15.0, 60.0, 75.0, 49.0], 571.469387755102, 0.01575, 11300.336388349395, -1]
  BEST : [[30, 45, 15.0, 60.0, 61, 48], 375.0, 0, 14315.187635201435, 0.0809758298742999]
 GEN : 25

Path 8 :
 BASE : [[38.0, 38.0, 50.0, 11.0, 19.0, 30.0], 437.0, 0.029151315789473664, 62929.84370941625, -1]
  BEST : [[33, 32, 46, 11.0, 19.0, 30.0], 120.0, 0, 67794.26144414651, 0.06583714614020264]
 GEN : 15

Path 9 :
 BASE : [[18.0, 70.0, 69.0, 70.0, 30.0, 70.0], 1076.2285714285713, 0.04082739130434784, 45562.328681963554, -1]
  BEST : [[4, 55, 69.0, 70.0, 30.0, 70.0], 491.9261998870694, 0.014488754940711439, 54528.49621986108,
0.062402004976992026]
 GEN : 18
```

When population is reduced it seems that the results are random. It is expected that the solution should be worse when the population reduces since there will be a lesser number of tests performed, but in some instances such as the solution for path 5 the lower population yielded a better result and in a smaller number of generations. This may be an outlier or possibly a random chance that a good solution was exploited earlier in the second run of the GA. In most paths the solutions were the same, and in some of them the solution worsened such as path 9.

```
Higher Population  = 100
Path 0 :
 BASE : [[70.0, 69.0, 36.0, 3.0, 40.0, 80.0], 1886.0, 0.025717391304347824, 36497.50091674977, -1]
  BEST : [[57, 41, 36, 2, 40.0, 80.0], 548.1578947368421, 0.0026249999999999997, 57026.72979614879,
0.011518540398332434]
 GEN : 48

Path 1 :
 BASE : [[38.0, 50.0, 11.0, 19.0, 30.0, 50.0], 644.0, 0.050289473684210495, 162319.41031167447, -1]
  BEST : [[30, 43, 2, 15, 21, 50], 72.0, 0, 171096.84128533237, 0.011262354503852861]
 GEN : 7

Path 2 :
 BASE : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 128746.45099646253, -1]
  BEST : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 128746.45099646253, 0.010630821357077765]
 GEN : 0

Path 3 :
 BASE : [[57.0, 40.0, 30.0, 11.0, 50.0, 38.0], 1331.7368421052631, 0.02510705741626796, 105224.70087660066, -1]
  BEST : [[52, 40, 30, 11.0, 50, 38], 975.6156299840511, 0.008574999999999996, 110689.41078309252,
0.012292513539612786]
 GEN : 93

Path 4 :
 BASE : [[30.0, 80.0, 40.0, 60.0, 19.0, 11.0], 1263.019138755981, 0.015224999999999999, 153463.40044802648, -1]
  BEST : [[25, 58, 40.0, 52, 19, 11.0], 1003.2499079867501, 0, 157449.58683620533, 0.012569260060484446]
 GEN : 77

Path 5 :
 BASE : [[75.0, 49.0, 4.0, 80.0, 40.0, 13.0], 3658.846153846154, 0.050400000000000056, 228182.43750074034, -1]
  BEST : [[40, 45, 4.4, 66, 36, 13.0], 3205.846153846154, 0.004390909090909156, 235133.8057311858,
0.012973075253648558]
 GEN : 70

Path 6 :
 BASE : [[60.0, 40.0, 40.0, 13.0, 30.0, 50.0], 521.0, 0.015224999999999999, 117043.5368613344, -1]
  BEST : [[54, 40, 38, 6, 30, 50], 289.0, 0.0011881578947368385, 120603.60225398178, 0.010969795487874226]
 GEN : 27

Path 7 :
 BASE : [[60.0, 60.0, 15.0, 60.0, 75.0, 49.0], 571.469387755102, 0.01575, 59866.84041573859, -1]
  BEST : [[55, 44, 15, 56, 66, 48], 379.2857142857143, 0, 62815.927376876345, 0.012081690577340218]
 GEN : 72

Path 8 :
 BASE : [[38.0, 38.0, 50.0, 11.0, 19.0, 30.0], 437.0, 0.029151315789473664, 124381.00622650543, -1]
  BEST : [[34, 30, 38, 6, 3, 30.0], 120.0, 0, 129245.42396123569, 0.011626409845781902]
 GEN : 8

Path 9 :
 BASE : [[18.0, 70.0, 69.0, 70.0, 30.0, 70.0], 1076.2285714285713, 0.04082739130434784, 123187.2498758676, -1]
  BEST : [[8, 69, 61, 57, 29, 63], 490.1254191405468, 0.0076124999999999995, 132181.0628334657,
0.011918347330877361]
 GEN : 54
```

When the population was increased it had the same effect as when the population was decreased. It seems apparent that due to the limited variety of solutions being prepared, there may be many duplicates in the set population. This means by increasing or decreasing the population the actual variety of solutions are limited. Path 5 was an outlier, saving almost 1 minute more than the solution found with a population of 50.

```
Lower Generation  = 30
Path 0 :
 BASE : [[70.0, 69.0, 36.0, 3.0, 40.0, 80.0], 1886.0, 0.025717391304347824, 39932.00966530794, -1]
  BEST : [[62, 43, 36, 2, 40.0, 80.0], 552.6666666666666, 0.007666129032258063, 60392.04232004139,
0.022644845683989966]
 GEN : 9

Path 1 :
 BASE : [[38.0, 50.0, 11.0, 19.0, 30.0, 50.0], 644.0, 0.050289473684210495, 69034.29397056712, -1]
  BEST : [[32, 45, 8, 3, 21, 49], 73.46938775510203, 0, 77789.17718912297, 0.024156181664786454]
 GEN : 6

Path 2 :
 BASE : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 124777.39544493803, -1]
  BEST : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 124777.39544493803, 0.021908490208975444]
 GEN : 0

Path 3 :
 BASE : [[57.0, 40.0, 30.0, 11.0, 50.0, 38.0], 1331.7368421052631, 0.02510705741626796, 101495.86917900901, -1]
  BEST : [[51, 38, 30.0, 11, 50.0, 38.0], 980.3524720893142, 0.008574999999999996, 106887.89224339562,
0.022516951459989165]
 GEN : 2

Path 4 :
 BASE : [[30.0, 80.0, 40.0, 60.0, 19.0, 11.0], 1263.019138755981, 0.015224999999999999, 164814.27429776223, -1]
  BEST : [[20, 76, 36, 58, 17, 10], 1100.500338066261, 0.0035000000000000022, 167308.14650487097,
0.024522036096555906]
 GEN : 17

Path 5 :
 BASE : [[75.0, 49.0, 4.0, 80.0, 40.0, 13.0], 3658.846153846154, 0.050400000000000056, 133972.99528755047, -1]
  BEST : [[40, 47, 4.4, 52, 37, 13], 3244.846153846154, 0.011233562713881803, 140325.89613963553,
0.02839945119791758]
 GEN : 11

Path 6 :
 BASE : [[60.0, 40.0, 40.0, 13.0, 30.0, 50.0], 521.0, 0.015224999999999999, 108068.27767766092, -1]
  BEST : [[53, 39, 40.0, 13.0, 30.0, 50.0], 289.0, 0.003675, 111628.33857161092, 0.022388808991980393]
 GEN : 13

Path 7 :
 BASE : [[60.0, 60.0, 15.0, 60.0, 75.0, 49.0], 571.469387755102, 0.01575, 59866.793251668525, -1]
  BEST : [[42, 30, 15, 57, 67, 41], 390.9627727856226, 0, 62636.69575012519, 0.024575806086359318]
 GEN : 25

Path 8 :
 BASE : [[38.0, 38.0, 50.0, 11.0, 19.0, 30.0], 437.0, 0.029151315789473664, 76280.00059776488, -1]
  BEST : [[31, 30, 36, 8, 19.0, 30.0], 120.0, 0, 81144.41833249514, 0.025512317847986164]
 GEN : 13

Path 9 :
 BASE : [[18.0, 70.0, 69.0, 70.0, 30.0, 70.0], 1076.2285714285713, 0.040827391304347844, 106778.38598878762, -1]
  BEST : [[14, 68, 66, 59, 27, 69], 495.3726652716423, 0.0076124999999999995, 115691.67995450406,
0.023749336305269 67]
 GEN : 15
```

When decreasing the number of generations to 30, there is a significant drop in quality of the solutions. Most paths get a solution that have a worse performance than from the base case and this is expected as some of the paths require more than 30 generations to get a good solution.

```
Higher  Generation  = 150
Path 0 :
 BASE : [[70.0, 69.0, 36.0, 3.0, 40.0, 80.0], 1886.0, 0.025717391304347824, 38163.00305127051, -1]
  BEST : [[57, 54, 36, 2, 40, 80.0], 548.1578947368421, 0.0026249999999999997, 58692.23193066954,
0.02324869427953248]
 GEN : 120

Path 1 :
 BASE : [[38.0, 50.0, 11.0, 19.0, 30.0, 50.0], 644.0, 0.050289473684210495, 134480.6019739908, -1]
  BEST : [[31, 43, 6, 11, 22, 50], 72.0, 0, 143258.0329476487, 0.022423657939106455]
 GEN : 27

Path 2 :
 BASE : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 65615.8184357098, -1]
  BEST : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 65615.8184357098, 0.022856473586198637]
 GEN : 0

Path 3 :
 BASE : [[57.0, 40.0, 30.0, 11.0, 50.0, 38.0], 1331.7368421052631, 0.02510705741626796, 103118.41364929451, -1]
  BEST : [[42, 40, 30.0, 11.0, 50.0, 38.0], 975.6156299840511, 0.008574999999999996, 108583.12355578637,
0.024336769219161662]
 GEN : 35

Path 4 :
 BASE : [[30.0, 80.0, 40.0, 60.0, 19.0, 11.0], 1263.019138755981, 0.015224999999999999, 111049.6696529733, -1]
  BEST : [[25, 77, 40.0, 54, 19.0, 11.0], 1000.6858054226476, 0, 115075.2021949983, 0.02595061962161909]
 GEN : 86

Path 5 :
 BASE : [[75.0, 49.0, 4.0, 80.0, 40.0, 13.0], 3658.846153846154, 0.050400000000000056, 226218.29521031826, -1]
  BEST : [[47, 34, 4.4, 80, 32, 13.0], 3260.846153846154, -0.01051236488792813, 232325.71540078634,
0.022931199636461088]
 GEN : 111

Path 6 :
 BASE : [[60.0, 40.0, 40.0, 13.0, 30.0, 50.0], 521.0, 0.015224999999999999, 117851.60336456329, -1]
  BEST : [[50, 39, 38, 13.0, 30.0, 50.0], 289.0, 0.0011881578947368385, 121411.66875721066,
0.02372927182259061 3]
 GEN : 17

Path 7 :
 BASE : [[60.0, 60.0, 15.0, 60.0, 75.0, 49.0], 571.469387755102, 0.01575, 16252.326607698345, -1]
  BEST : [[40, 44, 15.0, 55, 67, 49], 378.9239332096475, 0, 19206.965099448345, 0.03941438920249504]
 GEN : 16

Path 8 :
 BASE : [[38.0, 38.0, 50.0, 11.0, 19.0, 30.0], 437.0, 0.029151315789473664, 114473.70817505222, -1]
  BEST : [[34, 30, 36, 11.0, 19.0, 30.0], 120.0, 0, 119338.12590978248, 0.024310636135314843]
 GEN : 13

Path 9 :
 BASE : [[18.0, 70.0, 69.0, 70.0, 30.0, 70.0], 1076.2285714285713, 0.040827391304347 84, 107699.18033831862, -1]
  BEST : [[16, 63, 64, 33, 29, 70.0], 483.84729064039414, 0.0076124999999999995, 116789.33117775156,
0.0232735416278457]
 GEN : 130
```

When the number of generations is increased to 150, most of the solutions improved slightly. Those that took the extra generations such ass Path 9 with 130 generations saw 5-10 second improvements in duration of the trip. Fuel costs were not significantly better if at all. In conclusion, the number of generations should remain at 100.

```
Lower PC  = 0.2
Path 0 :
 BASE : [[70.0, 69.0, 36.0, 3.0, 40.0, 80.0], 1886.0, 0.025717391304347824, 35769.17174550163, -1]
  BEST : [[55, 32, 36, 1, 40, 80.0], 550.4545454545455, 0.0026249999999999997, 56263.1585196375,
0.024839460357138887]
 GEN : 14

Path 1 :
 BASE : [[38.0, 50.0, 11.0, 19.0, 30.0, 50.0], 644.0, 0.050289473684210495, 62883.79539286598, -1]
  BEST : [[30, 45, 11, 3.2, 4, 50.0], 72.0, 0, 71661.22636652387, 0.025577807731461066]
 GEN : 16

Path 2 :
 BASE : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 87405.6785340948, -1]
  BEST : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 87405.6785340948, 0.021226211501728903]
 GEN : 0

Path 3 :
 BASE : [[57.0, 40.0, 30.0, 11.0, 50.0, 38.0], 1331.7368421052631, 0.02510705741626796, 61092.47637308442, -1]
  BEST : [[46, 37, 30.0, 11.0, 49, 37], 985.4733824733826, 0.008574999999999996, 66405.91906762749,
0.028716516758355282]
 GEN : 80

Path 4 :
 BASE : [[30.0, 80.0, 40.0, 60.0, 19.0, 11.0], 1263.019138755981, 0.015224999999999999, 149399.99377376447, -1]
  BEST : [[25, 41, 40, 60, 19.0, 11], 994.019138755981, 0, 153527.82631578945, 0.025221925419420532]
 GEN : 57

Path 5 :
 BASE : [[75.0, 49.0, 4.0, 80.0, 40.0, 13.0], 3658.846153846154, 0.050400000000000056, 155017.54444022337, -1]
  BEST : [[40, 33, 4.3, 75, 33, 12], 3313.0, -0.0019523784355179718, 160324.6483764452, 0.026889997894448674]
 GEN : 87

Path 6 :
 BASE : [[60.0, 40.0, 40.0, 13.0, 30.0, 50.0], 521.0, 0.015224999999999999, 109967.0106446939, -1]
  BEST : [[51, 40, 38, 4, 30.0, 50], 289.0, 0.0011881578947368385, 113527.07603734128, 0.021695918389182934]
 GEN : 14

Path 7 :
 BASE : [[60.0, 60.0, 15.0, 60.0, 75.0, 49.0], 571.469387755102, 0.01575, 60070.46304946172, -1]
  BEST : [[39, 30, 15, 60, 63, 49], 373.46938775510205, 0, 63108.801541211724, 0.024164768383882804]
 GEN : 74

Path 8 :
 BASE : [[38.0, 38.0, 50.0, 11.0, 19.0, 30.0], 437.0, 0.029151315789473664, 69221.30705530051, -1]
  BEST : [[34, 32, 43, 11.0, 3, 30], 120.0, 0, 74085.72479003077, 0.0269439295250494]
 GEN : 14

Path 9 :
 BASE : [[18.0, 70.0, 69.0, 70.0, 30.0, 70.0], 1076.2285714285713, 0.04082739130434784, 155125.93632204257, -1]
  BEST : [[14, 59, 67, 55, 30, 70], 493.24893570886485, 0.01518319124715405, 164071.80522251938,
0.023659830620414643]
 GEN : 41
```

When reducing the *pc* to 0.2 the results were mixed. Some solutions improved slightly, some got slightly worse, and some remained the same.

```
Higher PC  = 0.8
Path 0 :
 BASE : [[70.0, 69.0, 36.0, 3.0, 40.0, 80.0], 1886.0, 0.025717391304347824, 35898.65061747328, -1]
  BEST : [[58, 51, 36, 1, 40, 80], 552.6666666666666, 0.005563793103448273, 56358.68707533243,
0.02284515250783028]
 GEN : 87

Path 1 :
 BASE : [[38.0, 50.0, 11.0, 19.0, 30.0, 50.0], 644.0, 0.050289473684210495, 113729.46442881136, -1]
  BEST : [[30, 40, 1, 19.0, 21, 50.0], 72.0, 0, 122506.89540246926, 0.023274381947751353]
 GEN : 23

Path 2 :
 BASE : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 66472.76937084485, -1]
  BEST : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 66472.76937084485, 0.021977052843386306]
 GEN : 0

Path 3 :
 BASE : [[57.0, 40.0, 30.0, 11.0, 50.0, 38.0], 1331.7368421052631, 0.02510705741626796, 106402.22352005815, -1]
  BEST : [[46, 40, 30.0, 11.0, 50, 37], 978.1760851760853, 0.008574999999999996, 111827.64324162825,
0.023301720239337728]
 GEN : 87

Path 4 :
 BASE : [[30.0, 80.0, 40.0, 60.0, 19.0, 11.0], 1263.019138755981, 0.015224999999999999, 124828.8810962665, -1]
  BEST : [[25, 34, 40.0, 51, 19.0, 11.0], 1004.6073740500985, 0, 128794.23716770326, 0.027227983949081044]
 GEN : 16

Path 5 :
 BASE : [[75.0, 49.0, 4.0, 80.0, 40.0, 13.0], 3658.846153846154, 0.050400000000000056, 170976.2929886057, -1]
  BEST : [[40, 47, 4, 72, 38, 13.0], 3439.846153846154, 0.004937234042553166, 174336.93023074933,
0.02511754691324899]
 GEN : 15

Path 6 :
 BASE : [[60.0, 40.0, 40.0, 13.0, 30.0, 50.0], 521.0, 0.015224999999999999, 108261.1149598909, -1]
  BEST : [[49, 30, 38, 6, 30, 50.0], 289.0, 0.0011881578947368385, 111821.18035253827, 0.022442265680686004]
 GEN : 39

Path 7 :
 BASE : [[60.0, 60.0, 15.0, 60.0, 75.0, 49.0], 571.469387755102, 0.01575, 25945.94316406886, -1]
  BEST : [[30, 30, 15, 59, 70, 46], 379.2778187177597, 0, 28895.15128269688, 0.026739018771714262]
 GEN : 50

Path 8 :
 BASE : [[38.0, 38.0, 50.0, 11.0, 19.0, 30.0], 437.0, 0.029151315789473664, 110826.70179122397, -1]
  BEST : [[34, 30, 36, 11, 5, 30.0], 120.0, 0, 115691.11952595423, 0.02260575229984079]
 GEN : 7

Path 9 :
 BASE : [[18.0, 70.0, 69.0, 70.0, 30.0, 70.0], 1076.2285714285713, 0.04082739130434784, 72636.9060109827, -1]
  BEST : [[14, 59, 69, 54, 28, 68], 489.94880885294424, 0.0076124999999999995, 81633.42905344407,
0.025544304875899995]
 GEN : 39
```

When increasing the pc to 0.8 most of the solutions got worse. In fact, Path 5 experiences a delay of:

$$3439.85 - 3269.48 = 170.37\ seconds$$

$$or\ 2\ mins\ 50.37\ seconds$$

This is likely because a large variety of bad solutions are being mixed in each iteration and potentially good solutions may be getting overwritten by the bad solutions. No solution improved.

```
Lower PM  = 0.2
Path 0 :
 BASE : [[70.0, 69.0, 36.0, 3.0, 40.0, 80.0], 1886.0, 0.025717391304347824, 40116.18885210381, -1]
  BEST : [[57, 38, 36, 2, 40.0, 80], 548.1578947368421, 0.0026249999999999997, 60645.417731502836,
0.02389237705086155]
 GEN : 60

Path 1 :
 BASE : [[38.0, 50.0, 11.0, 19.0, 30.0, 50.0], 644.0, 0.050289473684210495, 42330.662467975846, -1]
  BEST : [[32, 30, 6, 19.0, 21, 50.0], 72.0, 0, 51108.09344163374, 0.026218510318665286]
 GEN : 34

Path 2 :
 BASE : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 24668.05603755987, -1]
  BEST : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 24668.05603755987, 0.024673222335614503]
 GEN : 0

Path 3 :
 BASE : [[57.0, 40.0, 30.0, 11.0, 50.0, 38.0], 1331.7368421052631, 0.02510705741626796, 49201.497293584885, -1]
  BEST : [[51, 40.0, 30.0, 11.0, 45, 38.0], 975.6156299840511, 0.008574999999999996, 54666.20720007675,
0.024912221602586337]
 GEN : 12

Path 4 :
 BASE : [[30.0, 80.0, 40.0, 60.0, 19.0, 11.0], 1263.019138755981, 0.015224999999999999, 81928.62134042595, -1]
  BEST : [[25, 80.0, 40.0, 60.0, 19.0, 11.0], 994.019138755981, 0, 86056.45388245095, 0.025416850642170537]
 GEN : 28

Path 5 :
 BASE : [[75.0, 49.0, 4.0, 80.0, 40.0, 13.0], 3658.846153846154, 0.050400000000000056, 136035.73961663715, -1]
  BEST : [[40, 36, 4.4, 80.0, 30, 13], 3189.846153846154, -0.004486363636363615, 143232.64390606896,
0.024590395473307245]
 GEN : 64

Path 6 :
 BASE : [[60.0, 40.0, 40.0, 13.0, 30.0, 50.0], 521.0, 0.015224999999999999, 117020.99197935, -1]
  BEST : [[48, 38, 38, 1, 30.0, 50.0], 289.0, 0.0011881578947368385, 120581.05737199738, 0.021919635039424783]
 GEN : 20

Path 7 :
 BASE : [[60.0, 60.0, 15.0, 60.0, 75.0, 49.0], 571.469387755102, 0.01575, 50937.23700450676, -1]
  BEST : [[57, 44, 15.0, 60, 65, 49.0], 373.46938775510205, 0, 53975.57549625676, 0.023910137169581007]
 GEN : 33

Path 8 :
 BASE : [[38.0, 38.0, 50.0, 11.0, 19.0, 30.0], 437.0, 0.029151315789473664, 52050.214660619065, -1]
  BEST : [[32, 31, 44, 7.7, 14, 30], 120.0, 0, 56914.63239534933, 0.025470008193627477]
 GEN : 7

Path 9 :
 BASE : [[18.0, 70.0, 69.0, 70.0, 30.0, 70.0], 1076.2285714285713, 0.040827391304347784, 156782.91378453505, -1]
  BEST : [[16, 64, 68, 52, 28, 70.0], 485.203781512605, 0.0076124999999999995, 165852.24927153392,
0.02174300748725962]
 GEN : 27
```

When the *pm* value is lowered to 0.2, every solution either remained the same or improved.

```
Higher PM  = 0.6
Path 0 :
 BASE : [[70.0, 69.0, 36.0, 3.0, 40.0, 80.0], 1886.0, 0.025717391304347824, 38872.92305985868, -1]
  BEST : [[57, 55, 36, 2, 40, 79], 549.2971352431712, 0.0026249999999999997, 59384.67029368809,
0.024347766342470264]
 GEN : 82

Path 1 :
 BASE : [[38.0, 50.0, 11.0, 19.0, 30.0, 50.0], 644.0, 0.050289473684210495, 110449.22691682701, -1]
  BEST : [[31, 39, 4, 5, 4, 50], 72.0, 0, 119226.6578904849, 0.027385310584318718]
 GEN : 48

Path 2 :
 BASE : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 66502.15419198954, -1]
  BEST : [[36.0, 3.0, 40.0, 40.0, 60.0, 30.0], 360.0, 0, 66502.15419198954, 0.025232094945661022]
 GEN : 0

Path 3 :
 BASE : [[57.0, 40.0, 30.0, 11.0, 50.0, 38.0], 1331.7368421052631, 0.02510705741626796, 97962.48215392178, -1]
  BEST : [[38, 39, 29, 11, 49, 37], 984.6217085182602, 0.008574999999999996, 103288.9937853062,
0.02515850280014859]
 GEN : 21

Path 4 :
 BASE : [[30.0, 80.0, 40.0, 60.0, 19.0, 11.0], 1263.019138755981, 0.015224999999999999, 134743.04936337654, -1]
  BEST : [[20, 80.0, 40, 53, 19, 11], 1001.9436670578677, 0, 138749.2800186091, 0.026208819715880897]
 GEN : 30

Path 5 :
 BASE : [[75.0, 49.0, 4.0, 80.0, 40.0, 13.0], 3658.846153846154, 0.050400000000000056, 171467.3783340952, -1]
  BEST : [[69, 32, 4.3, 77, 36, 13], 3335.352647352647, 0.006141544489383172, 176431.46625478409,
0.03294764610452469]
 GEN : 56

Path 6 :
 BASE : [[60.0, 40.0, 40.0, 13.0, 30.0, 50.0], 521.0, 0.015224999999999999, 110715.26727252858, -1]
  BEST : [[51, 37, 38, 7, 30, 50], 289.0, 0.0011881578947368385, 114275.33266517596, 0.022787182215072744]
 GEN : 58

Path 7 :
 BASE : [[60.0, 60.0, 15.0, 60.0, 75.0, 49.0], 571.469387755102, 0.01575, 36866.86900718267, -1]
  BEST : [[39, 44, 15.0, 60.0, 65, 49.0], 373.46938775510205, 0, 39905.20749893267, 0.026698934890054333]
 GEN : 14

Path 8 :
 BASE : [[38.0, 38.0, 50.0, 11.0, 19.0, 30.0], 437.0, 0.029151315789473664, 75006.37103251973, -1]
  BEST : [[34, 32, 43, 11, 19, 30.0], 120.0, 0, 79870.78876725, 0.0240036435793957]
 GEN : 4

Path 9 :
 BASE : [[18.0, 70.0, 69.0, 70.0, 30.0, 70.0], 1076.2285714285713, 0.04082739130434784, 108920.95067951665, -1]
  BEST : [[17, 67, 66, 50, 28, 66], 489.01056406280287, 0.0076124999999999995, 117931.87108828274,
0.023167578720599286]
 GEN : 57
```

When the *pm* was increased to 0.8, the results worsened for the most part. In conclusion, the best solutions found on average were when the base case scenario was followed.

So, keeping the base case we can perform experiment #3

Experiment 3: Testing a massive route

For this experiment, the same map is used but a route of size 20 edges is used.

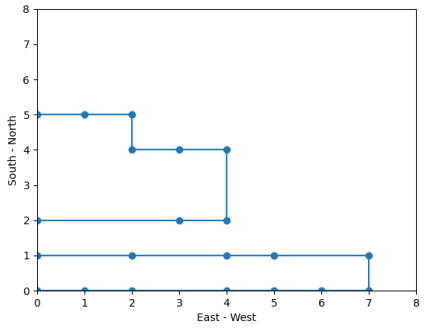

Figure 12: Experiment 3 Path of Size 20

```
BASE:
[9.0, 80.0, 50.0, 49.0, 75.0, 60.0, 3.0, 36.0, 70.0, 30.0, 80.0, 40.0, 13.0, 30.0, 50.0, 30.0, 19.0, 11.0,
50.0, 38.0], 5531.736842105263, 0.10375488721804489

BEST: [3.9, 62, 32, 33, 68, 60.0, 3.0, 35, 68, 29.9, 73, 38, 5.2, 25.5, 40, 5.4, 9.6, 7.7, 41, 37],
2790.2972972972975, 0.05814568588844537
GEN: 17
```

3.4.3 Conclusion

All files are accessible in the submission_files folder in the code repository.

From this experiment it is established that there is a better way for travellers to travel on their determined routes. As more roads are travelled the traffic light system impacts the expected travel time and fuel consumption of a journey.

To determine the best values for the GA, experiment #1 and experiment #2 show that the base case tested yields the best solutions on average.

Using the determined GA values, we can test experiment #3 with a path that contains 20 edges. Significant results are deduced. Travel time is reduced from 5531.74 seconds to 2790.30 seconds. This is a 49.56% improvement in travel time. This can however depend on case to case. In an instance where travelling the maximum available speed allows a traveller to avoid all red-light interruptions, the GA will be finished in the first generation. Fuel costs are also reduced from 0.104 to 0.058, a 44.23% improvement. While this improvement seems very miniscule, with a large network of cars the wasted fuel during idle time adds up.

4 Theoretic Solution for Adaptive Traffic Light System

4.1 Introduction

While it is great to have a solution for any given traveller based off traffic conditions and expected traffic light timings, a better solution may exist when the cars communicate with city traffic lights to encourage the traffic light timings dynamically. For instance, on Family Day certain roads may expect greater traffic which the predetermined traffic lights do not consider.

Commented [SM14]: I think this should be capitalized

4.2 Problem Formulation

This problem will be formulated in a similar manner to the Theoretic Solution for Shortest Trip Time Planning. Initially, the traffic lights are fixed with their respective timings. The simulation will be run for that traffic light system and N number of travellers travelling in their respective paths. Once the best fit route is determined for each traveller using the Solution for the Shortest Trip Time Planning, then the problem becomes adjusting the traffic light timings such that the collective groups travel fitness improves.

Let a solution for any n traveller where $n \in N$ be $X_n = \{x_{n_1}, x_{n_2}, \dots, x_{n_J}\}$. Where J is the number of edges travelled at a specific speed limit.

Let a fixed solution for a set of traffic light be $G = \{g_1, g_2, g_3, \dots g_{num\_nodes}\}$ where num_nodes is the number of traffic light nodes.

Each g_k solution will have the all the properties of a node.

Traffic light cycle time = CT $\{CT_0, CT_1,\ CT_2, CT_3\}$ one for each direction at the intersection.

Red light time, RT = $\{RT_0, RT_1,\ RT_2, RT_3\}$

Yellow light time, YT = $\{YT_0, YT_1,\ YT_2, YT_3\}$

Green light time, GT = $\{GT_0, GT_1,\ GT_2, GT_3\}$

Edges distances Ed = $\{\mathrm{Ed}_0, \mathrm{Ed}_1, \mathrm{Ed}_2, \mathrm{Ed}_3\}$

Additionally for each node the following random solution will be generated.

YT remains the same

We will allow a larger $cycle\ time = random(30,120)$

We will no longer determine the percent green by distance of an edge, instead we will randomly select a percentage for the E-W lights. To ensure that there is adequate time for the E-W lights we will allow the range of percentages to be from 20-80%. This will prevent theoretical cars that are not in the system from facing dire consequences. For example, if the best solution finds $ew_{\%g}$ to be equal to zero, then any cars not using this navigation system will be trapped indefinitely at the traffic light.

$$ew_{\%g} = 0.2 + random(0,0.6)$$

The optimization equation is as follows:

$$\max(Z) = \sum_{n=1}^{N} scoreof(X_n)$$

4.3 Solution

Using GA cyclically, we can solve this problem to get a good estimate of the best configuration of traffic lights and the travellers that must go through them. The algorithm works as follows:

1.) Create initial solution using fixed traffic light data
2.) Create travellers with variable path lengths
3.) Solve for each traveller in this network for their cost of path based off trip time and fuel consumption (as described in Theoretic Solution for Shortest Trip Time Planning).
4.) Sum their travel time and fuel consumption
5.) Using GA, modify the traffic light timing
6.) Save data.
7.) Run Steps 3-5 over a fixed number of generations
8.) Select best solution found
9.) Compare with initial solution

For step 5, the modification done is quite complex. There are multiple approaches but the aim to is reduce as many computations done as possible.

Commented [SM15]: "the modification"

Let a solution $Y = [\, node_1, node_2, \ldots node_n]$,

where n is the number of nodes that are of interest in this problem.

One approach is to take a map and modify every node in the network appropriately. This is very costly because it would require the modification of 64 nodes in each generation run. To mitigate this, only the nodes that are listed in the path of the travellers will be considered. This ideal especially when the entire county uses this navigation system, as only busy lights will need to be modified.

This algorithm is accessible in GeneticAlgorithm.py

4.4 Results & Experimentation

4.4.1 Hypothesis:

Provided numerous travellers that take variable length paths and the initial fixed traffic light timings it is expected that the system will perform sub-optimally. When the travellers find the optimal speed limit that they must travel at using GA, the collective trip time for the travellers and the idle time fuel wastage will decrease, but with the introduction of adaptive lights the solution should improve further. The collective difference is expected to be significant.

Using GA, the traffic lights will adjust their timings to reduce the sum of total trip times of the participants. Most participants will experience an improvement in their trip time, but some may experience a worse trip time. This is because the system will consider success of the collective over the individual. For example, if 10 cars take traffic light A going East, and 1 car takes traffic light B going North, it would be more beneficial to improve the trip time for the 10 cars over the single one.

4.4.2 Results:

Experiment # 4:

Provided the same 8x8 map. We will generate the following travellers:

10 travellers that follow the same path of length random (6-25).

5 travellers that follow the same path of length random (6-25).

5 travellers that follow the same path of length random (6-25).

1 traveller that follow the same path of length random (6-25).

This means 21 travellers will travel with 8 different unique paths.

Due to the large runtime the various trails for this method were conducted on a collection of virtual machines. This way many tests can occur simultaneously.

Commented [SM16]: trials

Trial 1 (MacOS):

Base Solution:

Total Time: 4975.295381310419

Total Fuel: 0

Best Last 3 Solutions:

Total Time : 2526.315789473684

Total Fuel : 0

Total Time : 2526.315789473684

Total Fuel : 0

Total Time : 2526.315789473684

Total Fuel : 0

From this trial, by adjusting the traffic light timings the total time of all travellers in the path is reduced by 49.20%. Since each path in the network experiences no idle time, there is no

fuel wasted in this example. Since the last 3 solution from the last three generations is identical, the solution has converged to an optimal one. Total runtime on MacOS was 2 hours for this run.

Trial 2 (MacOS):

Base Solution:

Total Time: 229391.05333140626

Total Fuel: 1.7007409869197483

Best Last 3 Solutions:

Total Time : 220411.8001411686

Total Fuel : 0.790416346491012

Total Time : 221465.8029602006

Total Fuel : 1.0179505003234832

Total Time : 217916.06428633636

Total Fuel : 0.6599202502330488

In this trial, there isn't a large improvement in saving trip time for the travellers, but there is a significant improvement on cost savings for fuel. The second last solution seems to select a worse solution than its previous generation. On inspected this run's best solution was identical to the previous run's. This means that the node timings were the same, but the discovered Total Time and Total fuel was different.

Commented [SM17]: inspection

This is because for the same set of node solution, when running GA for individual paths the yield score was different. This is because the more optimal solution may not have been discovered on the second run for some of the paths.

The final solution is taken for analysis. The overall improvement in time is 5.00 % and the improvement in fuel savings is 61.20%. Total runtime on MacOS was 2 hours for this run.

Trial 3 (GPU Intense Windows)

Base Solution:

Total Time: 28826.842996431234

Total Fuel: 0.16888026170344733

Best Last 3 Solutions:

Total Time : 27047.82217381706

Total Fuel : 0.045584762605815204

Total Time : 27117.908923582967

Total Fuel : 0.05407987623053528

Total Time : 26882.73740545169

Total Fuel : 0.08599300289813115

The results for this run are very similar to Trial 2. The major difference is that with this accelerated hardware the run took only 1 hour. That is a runtime saving of 50%.

4.4.3 Conclusion

Some changes will need to be made with optimizing the solution of the smallest trip time and fuel consumption, because solving for each path takes a ton of computational power. One way to do so is to reduce the number of generations per path, but that can prevent the discovery of a more optimal solution. With a better computer, this trade-off will not be necessary.

So,

$$\#\ gen\ for\ each\ path = 50$$

$$\#\ gen\ for\ adaptive\ GA = 50$$

But with the time complexity it seems that this is also not ideal. After running a time measurement tool with python called *timeit,* it is revealed that each cycle of optimizing a path takes roughly 1.5 seconds on average. This means that the time to complete the run of the adaptive traffic lights optimization algorithm is:

$$O(1.5 * \#\ Paths * \#\ gen\ for\ adaptive\ GA * \ \#\ population\ for\ adaptive\ GA)$$

That means for an experiment that tests 8 unique paths and follows the current base case for the GA it would take:

$$1.5 * 8 * 50 * 50 = 30\ 000\ seconds = 8.33\ hrs$$

So, to get more samples for this experiment, the number of unique paths that were tested was 5. So, the simulation time will take:

$$1.5 * 4 * 50 * 50 = 15\ 000\ seconds = 4.17\ hrs$$

Even with a powerful computer this is not well optimized solution. Hence, memory will need to be used to improve the computational time. For each new solution set of nodes, the result will be added it will be added to a list of known past solutions. This repeated over many instances can significantly save computational time assuming that the variance in solutions is

Commented [SM18]: "not a well optimized solution", you missed the a

Commented [SM19]: You repeated added here

limited. To save on search time, a HashMap structure may be beneficial to store the discovered solutions to prevent computing for them again. This is especially useful for any survivor solutions that are identical. But to ensure that the survivors are identical requires a has function that accepts each individual node and its values as an input. Since the traffic light times for each new node has a vast solution space to select from, it will be very rare to discover identical solutions making the hash function inadequate for this use case. Improvement to the time complexity is outside the scope of this project and can be explored in a later project.

With the results of the experiment, it is clear that dynamically controlled traffic lights are a superior method to reduce traffic trip time and fuel consumption over the fixed traffic light timings. It is arguable that the current traffic light system in counties is far more elaborate than the randomly selected traffic light timings in this model. They may consider many factors that are missing in this model when deciding how to determine traffic light times. But adaptive traffic lights using GA should be able to come to the same conclusion as the engineers if not a better solution as it progresses with trial an error. So all in all, while the improvement may not be as drastic in real life applications as have been in these simulations, theoretically this solution works and now needs to be tested on a real road network.

Commented [SM20]: Is this supposed to be hash?

Commented [SM21]: "and" not "an"

Commented [SM22]: "as they have been"

5 Conclusion

This project had 3 goals to achieve. Realizing a realistic road network simulation, determining what speed a traveler must travel based on the traffic light timings of their route, and consider what timings of traffic lights will optimize travel for all travellers of a network. These goals were largely met but still require further verification that can be the basis of future projects.

The development of the road network map was very rigorously done. Nodes were created for traffic light intersections; edges were created for roads that start from one node and end at another. Node traffic light timings were determined based off studies and data that was available online. The limitation for determining the initial traffic light timings was the missing comprehensive factors that determine traffic light timings without intervention. For example, certain holidays, weather conditions, etc. The limitation of determining speed limit and traffic conditions for the edges was the lack of pre-determined knowledge of the roadway. Other factors such as number of lanes, length of incoming intersection, road history could have been useful in creating a more realistic roadway.

The development of a random traveller's path was very straightforward but had some limitations. For example, since the path is randomly discovered, it may not be the best path for the traveller to take in a realistic scenario. This aspect of path planning is not a focus of this project, but the inclusion of the Shortest Path Finding with Traffic Lights could very well be an excellent project as a standalone. Perhaps testing larger paths would also be of greater interest in future experiments to see a long-term time and fuel saving's for a traveller.

Commented [SM23]: Drop this "a"

The theoretical solution for determining travel speed for a traveller based off traffic light information and using GA was successful. As the experiments prove, there is great potential time

and fuel savings that a traveller can benefit from should this algorithm be used in navigation systems. This solution is ideal for cities where the traffic light information is known but does not change. The only questionable limitation for this model is the determination of delay time based on the different actions a car can take. While this model selected 5 seconds as a reasonable delay for waiting at a red light or taking a left turn, these values may be different on a case-to-case basis. For example, a road that has extreme traffic might require the traveller to wait through 2 or more cycles before getting a chance to cross the intersection. But on average 5 seconds is actually optimistic, so the fact that this solution shows promise for this metric means that there is a chance that on a real-life road network the algorithm may yield even greater benefits.

The random traffic light generator worked as expected. Its only limitation was the determination of cycle time range and green percent for the East to West light. While this project took the typical value range of 30 to 120 seconds, it may be worth to explore traffic light cycles greater than this range. For instance, a road that receives 1 car per hour on the E-W traffic lights may only need to turn green every 200 or more seconds. The minimum percent for a traffic light to be green is 20%. This value should be modified and tested for many scenarios where paths interest for various travelers.

Commented [SM24]: Is this supposed to be "intersect"? "interest" doesn't make sense here

The Adaptive Traffic Lights model provided promising results. Many of the tests conducted revealed either a massive improvement in trip time and or fuel consumption. The main limitation for this model was the time complexity. The problem grows exponentially as the number of generations grow for GA. Since the algorithm conducts GA to determine potential node configurations and then conducts GA for each path that uses the new configuration, this problem grows exponentially with the number of generations used in testing. There are many ways to mitigate this problem. Memorizing solutions so that they do not need to be recalculated,

parallelizing the calculations, using distributive computing, and better hardware. Instead of randomly solving for potentially solutions, it can be beneficial to have a heuristic approach that aids in only searching for solutions that will be better.

Appendix – Code

MapObjects.py

This class contains all of the data structures used to model and test simulations.

```
class Node:
    #Typical cycle times – 30s to 120s.
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.R_T = [0,0,0,0]
        self.Y_T = [0,0,0,0]
        self.G_T = [0,0,0,0]
        self.Go_T = [0,0,0,0]
        self.Stop_T = [0,0,0,0]


        # these are edges
        self.east= None
        self.west = None
        self.south = None
        self.north = None

        self.cycletime = None
        self.isIsolated = False
```

A node has an x,y coordinate, an array that stores the timing of the lights for each R, Y, and G, and the Go and Stop time for each light. Go time is the sum of Y and G time. A node points to the edges that are attached to it. If this node has no edges, it is isolated.

```
class Edge:
    def __init__(self, node_1, node_2, distance, direction):
        self.node_1 = node_1 # (x,y)
        self.node_2 = node_2 #(x,y)
        self.distance = distance # km
        self.speed = 10 * random.randint(0,5) + 30 # speeds from 30-80 km/h
        self.direction = direction # 0 – s_n or 1 – e_w
```

```
        self.traffic = gaussian() * random.randint(0,1)
```

The edge contains coordinates to the two nodes that it is formed from. The distance is calculated by: `d = self.node_1 - self.node_2`

Where either the x or y coordinates of the nodes are different.

Direction is determined by wheter the x or y coordinates of the nodes are different.

Traffic coefficient is a Gaussian distribution:

```
def gaussian():
    x = -1
    while(x < 0 or x > 1):
        x = random.gauss(0.5,1)
        if(x >= 0 and x <= 1):
            return x
class Path:
    def __init__(self, minSize):
        self.minSize = minSize
        self.nodes = []
        self.edges = []
        self.q_Seq = [] # quick sequence of coordinates travelled for testing purposes
        self.directions = []
        self.motion = []
```

The path data structure contains the minSize which is the size of the path in number of edges traversed. Each node and edge visited is concatenated here. The motion array stores a list of actions such as left turn, right turn, and go straight.

MapGenerator.py

This class creates the map model.

```
def makeNodes(w,h):
    num_nodes =  w*h * 0.5
    nodes = set()
    checkExists  = set ()
    while len(nodes) < num_nodes:
        x = random.randint(0,w_grid)
        y = random.randint(0,h_grid)
        #check and see if it exists in collection, if not then ignore
        if  not ((x,y) in checkExists):
            nodes.add(Node(x,y))
            checkExists.add((x,y))
    return sorted(nodes,key=lambda node: (node.x, node.y))
```

The code above demonstrates how the nodes are created for the map. Given an w * h map, only a few of the nodes will be generated and in this case that is 50% of all the nodes possible.

```
def makeEdges(nodes):
    #using only the co

    x_e = []
    sortedNodes = sorted(nodes,key=lambda node: (node.x, node.y))
    lastNodes = []

    for node in sortedNodes:
        thisX = node.x
        if(len(lastNodes) > 0 ):
            l_node = lastNodes.pop(-1)
            lastX = l_node.x
            if thisX == lastX:
                d =  node.y - l_node.y
                edge = edge = Edge((l_node.x,l_node.y), (node.x,node.y), d, 0 )
                x_e.append(edge)
                l_node.north = edge
                node.south = edge

                lastNodes.append(l_node)
                lastNodes.append(node)
            else:
                lastNodes.append(l_node)
                lastNodes.append(node)
```

```
        else:
            lastNodes.append(node)

    y_e = []
    sortedNodes = sorted(lastNodes,key=lambda node: (node.y, node.x))
    lastNodes = []


    for node in sortedNodes:
        thisY = node.y
        if(len(lastNodes) > 0 ):
            l_node = lastNodes.pop(-1)
            lastY = l_node.y
            if thisY == lastY:
                d =  node.x - l_node.x
                edge = edge = Edge((l_node.x,l_node.y), (node.x,node.y), d, 1 )
                y_e.append(edge)
                l_node.east = edge
                node.west = edge
                lastNodes.append(l_node)
                lastNodes.append(node)
            else:
                #add it back because it doesnt need to be updated
                lastNodes.append(l_node)
                lastNodes.append(node)
        else:
            lastNodes.append(node)

        sortedNodes = sorted(lastNodes,key=lambda node: (node.x, node.y))
    return x_e, y_e, sortedNodes
```

For every node that exists, the code must connect them to create a set of edges. First, all of the nodes are sorted by their X coordinate. Then while their Y coordinates are the same, we can connect those edges to make an edge. Then the nodes are sorted by their Y coordinates and X coordinates are checked.

CreatePaths.py

This class takes the saved map created by MapGenerator.py and creates a viable route.

```
def createPath(path_min_len, x,y,n):
    try:
        allEdges = x + y
        i = 0
        #1.) checked nodes is none
        checkedNodes = []
        path =  Path(path_min_len) #2. start a new path
        start_node = None
        options = []
        east = None
        west = None
        south = None
        north = None

        start_node =  n[random.randint(0,len(n) - 1)] # select a randome start node

        while (start_node.isIsolated): # while the node discovered has no edges
            start_node =  n[random.randint(0,len(n) - 1)] # select a randome start
node
            checkedNodes.append((start_node.x, start_node.y))

        path.nodes.append(start_node)
        path.directions.append(None) # the first direction travelled is null
        path.motion.append(None) # first node no motion

        j = 0

        while j < path_min_len:
            #3.) we will check each node to see if it has edges
            #4.) if it doesnt we will back track
            #5.) if it does, then we will move to the next node
            #6.) we do not consider reversing our path as an option

                # direction
            # 0 - East
            # 1 - West
            # 2 - South
            # 3 - North
            if len(path.nodes)  == 0: # we backtracked all the way to the begining
                return None
            curNode = path.nodes[-1] # get the last appeneded node
            lastDir = path.directions[-1] # get the last direction travelled
```

```
            exclude  = []
            options = [ curNode.east, curNode.west, curNode.south, curNode.north] #
NOTE HERE east and north are 0 , 1

            if lastDir == 0:
                options[1] == None # dont go west if u came from the east

            elif lastDir == 1:
                options[0] == None

            elif lastDir == 2:
                options[3] == None

            elif lastDir == 3:
                options[2] == None

            # Exlcude the nodes already visited or nodes that are isolated
            if (options[0] != None):
                test = findNode(options[0].node_2,n)
                if  options[0].node_2 in checkedNodes or test.isIsolated:
                    options[0] = None

            if options[1] != None:
                test = findNode(options[1].node_1,n)
                if (options[1].node_1 in checkedNodes) or test.isIsolated:
                    options[1] = None

            if options[2] != None:
                test = findNode(options[2].node_1,n)
                if (options[2].node_1 in checkedNodes) or test.isIsolated:
                    options[2] = None

            if options[3] != None:
                test = findNode(options[3].node_2,n)
                if (options[3].node_2 in checkedNodes) or test.isIsolated:
                    options[3] = None

            for i in range(4):
                if options[i] == None : exclude.append(i)
            if len(exclude) == 4: # time to backtrack
                path.nodes.pop() # remove the last node we found
                if path.directions:
                    path.directions.pop()
                if path.edges:
                    path.edges.pop()
                if path.motion:
```

```
                path.motion.pop()
            j = j - 1
            if j < 0:
                #print("Bad run get new route")
                return None
        else:

            # motion
            # 0 - straight
            # 1 - right
            # 2 - left
            randSide = choice([p for p in range(0,3) if p not in exclude])
            if options[randSide] != None:
                lastDirection  = path.directions[-1]
                lastMotion  = path.motion[-1]
                path.directions.append(randSide)
                if lastMotion == None:
                    path.motion.append(0) # moving straight
                elif lastDirection ==  randSide:
                    path.motion.append(0) # moving straight
                elif lastDirection == 0: #east
                    if randSide == 2:
                        path.motion.append(1) # EN Left
                    if randSide == 3:
                        path.motion.append(2) # ES Right
                elif lastDirection == 1: #west
                    if randSide == 2:
                        path.motion.append(2) # WN
                    if randSide == 3:
                        path.motion.append(1) # WS
                elif lastDirection == 2: #south
                    if randSide == 0:
                        path.motion.append(2) # SE
                    if randSide == 1:
                        path.motion.append(1) # SW
                elif lastDirection == 3: #north
                    if randSide == 0:
                        path.motion.append(1) # NE
                    if randSide == 1:
                        path.motion.append(2) # NW

                # Get the updated edge instead of the pointer edge.
                #newEdge =  findEdge()
                # this_edge  = options[randSide]
                # newEdge =  findEdge(this_edge.node_1, this_edge.node_2,
allEdges)
```

```
                    # if (newEdge == None) :
                    #     print("FATAL ERROR. NO such edge exists")

                    # path.edges.append(newEdge)

                    path.edges.append(options[randSide])
                    newNode = None
                    if randSide == 0 or randSide == 3: # east or north
                        #this means look at the second node of the edge
                        newNodeCor =  options[randSide].node_2

                    elif randSide == 1 or randSide == 2:  # west or south
                        # this means that we look at the first node of the edge to
travel to
                        newNodeCor =  options[randSide].node_1

                    newNode = findNode(newNodeCor,n)
                    if newNode == None: return None
                    path.nodes.append(newNode)
                    checkedNodes.append(newNodeCor)
                    j = j + 1
        return path
    except Exception as e:
        print("error : ", e)
        return None
```

This function creates a path provided a path size and map information as input.

GeneticAlgorithm.py

This class contains two classes. One for GA provided a path as an input (class `GeneticAlgorithm)`, and one for GA provided a list of nodes and paths as input (class `GeneticAlgorithmnAdaptive`).

```
def getFuel(self, time):
        # input time in seconds
        return 1.89/3600 * time
```

This is the price of the fuel calculated to be wasted by the idle time

```
def f(self, x):
        # this is the function
        t_total = 0
        f_total = 0

        penalty = 5
        leftDelay = 5

        # node 1 is starting node.
        # if the light is red we must wait for green
        if self.G_Offset[0] > 0:
            # this is a green offset
            # wait time =  time stop -  offset
            wait_time  = self.R_T[0] - self.G_Offset[0] + penalty
            f_total += self.getFuel(wait_time)
            t_total += wait_time
    # else
        # otherwise proceed to the algorithmn as usual
        for i in range(0, len(self.D) ):
            if x[i] == 0:
                x[i] = 0.1
            t_t = (self.D[i]/ x[i] * 1 * 60 * 60) #time taken on this edge

            if (i + 1 ) != len(self.D):
                motion_side = self.motion[i+1]
                if motion_side == 1:
                    # motion is right
                    # no delay, concat travel time and move to the next edge
                    t_total += t_t
                    # f_total += self.getFuel(t_t)
                else:
                    # check to see if green or if red
```

```
                # if green no delay move forward
                # if red then wait until completion of time and then add a penalty
                # This is not the last edge to travel
                offset  = self.G_Offset[i+1]
                cur_GT  = self.G_T[i+1]
                cur_TR  = self.R_T[i+1]
                cur_C = self.C[i+1]
                if cur_GT == 0: # this means we have a light free intersection
                    #as there is no alternative direction at this node
                    t_total += t_t

                elif offset <= 0: # we are at green t = 0
                    #extra delay for motion_side == 2 (left)

                    newTime = t_total + t_t
                    n = newTime // cur_C
                    lhs = n * cur_C +  offset
                    rhs = n * cur_C + offset + cur_GT

                    if newTime > rhs:
                        delay = cur_C * (n+1) + offset - newTime
                        f_total += self.getFuel(delay + penalty)
                        t_total = cur_C * (n+1) + offset + penalty

                    elif newTime <= lhs:
                        delay = lhs - newTime
                        f_total += self.getFuel(delay + penalty)
                        t_total = lhs + delay

                    else:
                        # we got to a green light no delay no extra milage
                        # extra milage for arriving at green light and not at end
of red light
                        if motion_side == 2:
                            t_total += t_t
                            t_total += leftDelay
                            f_total += self.getFuel(leftDelay)

                elif offset > 0 : # we are at red t = 0

                    newTime = t_total + t_t
                    n = newTime // cur_C
                    lhs = -1 * offset + cur_TR + n * cur_C
                    rhs = -1 * offset + (n+1) * cur_C

                    if newTime <= lhs:
```

```
                        reachTime  = lhs
                        delay = reachTime - newTime
                        f_total += self.getFuel(delay + penalty)
                        t_total += reachTime + penalty

                    elif newTime > rhs:
                        newLhs  = -1 * offset + cur_TR + (n+1) * cur_C
                        delay =  newLhs - newTime
                        f_total += self.getFuel(delay + penalty)
                        t_total = newLhs + delay
                    else:
                        # no delay no extra milage
                        t_total += t_t
            else: # this is the last edge to travel. No need to check anything just
                t_total += t_t
        return t_total, f_total
```

This is a massive function that determines the total time and total fuel consumption of a path. In summary this class accepts a list of traversed traffic lights and edges and checks their traffic light times, distance to calculate the amount of time taken to travel and the amount of time that is taken waiting in an idle state. Then using the idle time, it determines how much fuel is wasted and its cost to the traveller.

```
def russianRoulette(self):
        surviors = []

        # keep the best two based off relative fitness including the base solution

        surviors.append(copy.deepcopy(self.X[0]))
        surviors.append(copy.deepcopy(self.X[1]))

        for i in range(self.population - 2):
            # lets keep it significant by 5 decimal places since there are many
possible ties at
            # 4 decimal places
            # we should also not keep 1 as a possibly randomly generated number.
            # from experimentation the sum of the surviors is arround
0.9999999999999996

            check = round(random.uniform(0, 0.99999999),8)
```

```
            concat  = 0
            for j in range(len(self.X)):
                concat += self.X[j][4]
                if concat >= check:
                    surviors.append(copy.deepcopy(self.X[j]))
                    break
        self.X = copy.deepcopy(surviors)
```

The Roulette Selection takes the 2 best solutions and stores them into the survivors list. Then for each of the remaining population a random solution is selected based on its fitness score.

```
def crossOver(self):
            # cross over chromosomes
            # exclude the 2 fittest solutions
            # crossover produces 2 childern who switch their chromosome values at a
pivot point

            index_swap = []
            cloneList  = copy.deepcopy(self.X)
            index_A = None
            index_B = None

            for i in range(2,self.population):

                check = round(random.uniform(0,1),2) #TODO: fix for more nodes

                if check <= self.pc: #this chromosome needs to be crossed over
                    index_swap.append(i)

            if(len(index_swap) <= 1):
                # only one chromosome to cross over
                #hence do nothing and let the solutions remain the same
                return

            for j in range(len(index_swap)):
                if j == (len(index_swap)-1): # last element must cross over with the
first

                    index_A = index_swap[j]
                    index_B = index_swap[0]
                else:
                    index_A = index_swap[j]
```

```
                    index_B = index_swap[j+1]

                crossOverPoint = int(round(random.uniform(1,len(self.X[0][0]) - 1),0))

                for k in range(len(self.X[0][0])):
                    if k >= crossOverPoint:
                        cloneList[index_A][0][k] = cloneList[index_B][0][k]
            self.X = copy.deepcopy(cloneList)
```

For the crossover, a cloneList is made from the output from the Roulette.

```
def mutation(self):
        # for each chromosome (excluding the 2 best fit pair) run a randome number
generator
        # if the number is less than the ratio then randomly select a speed value.
        # do this for each speed value in a solution X
        # return the children
        clone = copy.deepcopy(self.X)
        for i in range(2,self.population):
            for j in range(len(self.X[0][0])):
                check = round(random.uniform(0,1),2)
                if check <= self.pm:

                    if self.S[j] > 30:
                         clone[i][0][j] =  random.randint(30, round(self.S[j], 0) )
                    elif self.S[j] <= 0: # there was an error in this path. go at a
steady 0.1 km/h
                        clone[i][0][j] = 0.1
                    elif self.S[j] < 1:
                        clone[i][0][j] = round( random.uniform(0.1, self.S[j]) , 1 )
                    else:
                        clone[i][0][j] = round( random.uniform(0.1, self.S[j]) , 1 )
        self.X = copy.deepcopy(clone)
```

This is how the mutation is done.

The only major difference between

class GeneticAlgorithm (GA) and class GeneticAlgorithmnAdaptive (GAA)

is that for each of their individual methods, GA focus on solving for the best speed travel for a path, and GAA focuses on the best configuration of node traffic light timings.

Commented [SM25]: "focuses" not "focus"

Bibliography

[1] “PSW signal Timing2 - Civil and construction engineering,” *oregonstate.edu*. [Online]. Available: https://cce.oregonstate.edu/sites/cce.oregonstate.edu/files/pw_sigtime.pdf. [Accessed: 23-Apr-2022].

[2] “Law document english view,” *Ontario.ca*, 19-Nov-2018. [Online]. Available: https://www.ontario.ca/laws/statute/90h08#BK231. [Accessed: 23-Apr-2022].

[3] H. Y. Alhammadi and J. A. Romagnoli, “Pareto optimality,” *Pareto Optimality - an overview | ScienceDirect Topics*, 2004. [Online]. Available: https://www.sciencedirect.com/topics/engineering/pareto-optimality. [Accessed: 23-Apr-2022].

[4] “Average salary in Canada – average Canadian salary,” *Living in Canada*. [Online]. Available: https://www.livingin-canada.com/work-salaries-wages-canada.html. [Accessed: 23-Apr-2022].

[5] “Best gas prices & local gas stations in Ontario,” *GasBuddy*. [Online]. Available: https://www.gasbuddy.com/gasprices/ontario/. [Accessed: 23-Apr-2022].

[6] “Idling: Why It’s a Problem and What You Can Do,” *scdhec*. [Online]. Available: https://any.run/report/580531d137700448208e00f42b5625e9906d0ab66a13fa8fe56cdc0275dd8189/6628771b-a08b-4319-87c1-e76df47712ce. [Accessed: 23-Apr-2022].

[7] S. John, “The 5 most fuel-efficient cars on the road today - and the 4 least fuel-efficient ones,” *Business Insider*, 02-Dec-2019. [Online]. Available: https://www.businessinsider.com/most-fuel-efficient-cars-vehicles-best-gas-mileage-2019-11#:~:text=The%20average%20car%20sold%20in,miles%20per%20gallon%20(MPG). [Accessed: 23-Apr-2022].