















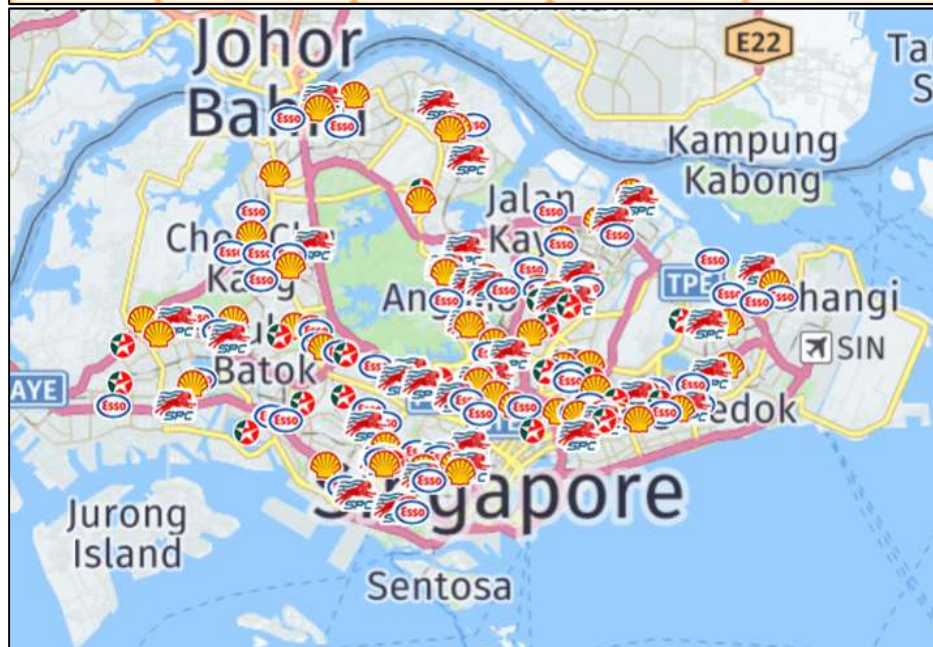


Seedly seedly.sg			 CALTEX	
92-Octane	N/A	N/A	\$2.15	\$2.15
95-Octane	\$2.19	\$2.22	\$2.19	\$2.19
98-Octane	\$2.52	\$2.55	\$2.56	\$2.49
Top Grade	V-Power Nitro+ \$2.73	-	-	-
Diesel	\$1.73	\$1.74	\$1.71	\$1.71
Credit Card Promotions	  	  	  	   



## Petrol Station Route Optimizer

THOR team members  
 IAN TAN ENG KIONG  
 KHOO WEE BENG  
 KOH SOOK BING  
 RANA BHATTACHARJEE  
 TAN YAO TAI TEERAPONG  
 YEO WHYE CHUNG NELSON

# Table of Contents

---

1	EXECUTIVE SUMMARY .....	3
2	PROBLEM OVERVIEW .....	4
2.1	PROBLEM STATEMENT .....	4
2.2	PROJECT OBJECTIVES .....	6
2.3	PRODUCT OVERVIEW .....	6
2.4	REQUIREMENTS & CONSTRAINTS .....	6
3	TECHNICAL DISCUSSION .....	8
3.1	INTRODUCTION .....	8
3.2	TECHNICAL SOLUTIONS .....	9
3.2.1	DATA MINING AND COLLECTION .....	9
3.2.2	CHOICE OF MAP API SERVICES .....	9
3.2.3	RULES ENGINE WITH CLIPS .....	10
3.2.4	ROUTE OPTIMISATION WITH OPTAPLANNER .....	10
3.2.5	BRUTE FORCE SEARCH WITH JAVASCRIPT ("My Route is fixed") .....	18
3.2.6	GENETIC ALGORITHM WITH PYTHON ("No Fixed Order") .....	18
4	SYSTEM ARCHITECTURE .....	28
4.1	PROJECT SCOPE .....	28
4.2	ASSUMPTIONS .....	29
4.3	SYSTEM'S FEATURES .....	29
4.4	LIMITATIONS .....	31
5	CONCLUSION & REFERENCES .....	32
5.1	IMPROVEMENTS AND FUTURE ENHANCEMENTS .....	32
6	BIBLIOGRAPHY .....	33
7	APPENDICES .....	34

# 1 EXECUTIVE SUMMARY

---

How people live, work and spend their money has changed dramatically over the past decade, especially with the advent of today's disruptive technology. The new and disruptive technologies have opened new ways to make and spend money – online, picking up a “gig” (freelance engagement) is as easy as making plans for dinner or finding a date. In other words, we are also leaving in an “Appgeneration”, where apps are changing the world, and we are part of the change. The numerous apps are altering the way that people view and perform work. Often driven by algorithms that align supply with demand, these apps are also paving the road towards an economy in which services are supplied only when needed and reachable to a wider audience.

The result is a system of flexible working that stands in stark contrast to the corporate world's insistence on a five day, nine-to-five work week. Flexibility and autonomy are what draw many to the freelance lifestyle, but such freelancers have limited productivity and organizational tools unlike multinational companies. The demand for freelance couriers' services is expected to grow year on year given the popularity of online shopping and e-commerce. In addition, one key weekly tasks that a freelance courier need to do is refueling of his/her vehicle. This task is also a common task to any driver.

Our project team would like to build an app that helps freelance couriers to improve their productivity as well as locate a preferred petrol station (the petrol station with the best price offer based on choice of credit card) along one of their routes for any driver. The reason we are targeting the logistic industry, is because the need of low cost non urgent delivery services is on the rise, where items can be placed at designated drop off points and delivered to designated collections points for customers to pick up. These drop off points and collections points can be at retails malls or commercial premises. Transportation and manpower expenditure are key cost elements to these companies, and many of these companies leverage freelance couriers for the logistic services. In addition, the app should also benefit the man on the street to locate the optimize route from one destination to another including locating the preferred petrol station.

Using the techniques imparted to us in lectures, our group first set out to build a hybrid reasoning system that incorporates an expert system built using rules engine like Clips which helps in selecting the preferred petrol station together with genetic algorithm written in python to determine optimized routes to any of the app user. The app will display all these through visual maps as well as the suggested segments of the routes.

Our team had an exciting time working on this project and hope to share our insights with everyone. There are truly are a wide array of individual factors to come to a final decision in the switch to a different pricing plan, and we only wish there was more time to work on the scope and scale of the project.

## 2 PROBLEM OVERVIEW

### 2.1 PROBLEM STATEMENT

We've all been there and for many the fuel light can be the bane for a driver especially for those who drive extensively. The question that comes to mind is where the nearest petrol station with the best offer is. Which route shall I take to reach the petrol station or among my numerous destinations that I need to go today. Let's look at the following personas who encounter such issues on a daily basis.



John drives every day and travels extensively. He is also very cost conscious and hope to get the best offer from his credit card given that fuel is a major cost component. As he travels a lot, within a week his car fuel light will light up at least once or twice. Many drivers including John find it annoying when they cannot find the preferred petrol station. Hence, they may choose not to refill until they locate the preferred petrol station which can be many kilometers away. Well, every driver should know some facts about the fuel tank capacity to avoid either damaging the engine or feeling too rushed. Hence, an application or app that can help drivers like John to navigate to their preferred petrol station will certainly be helpful in removing anxiety and risk of empty fuel tank predicament.

In addition of locating the preferred petrol station, identifying the shortest route will certainly be at the back of each drivers mind when they want to locate the preferred petrol station. This app will help the drivers do that. Rather than limiting the app to be a locator of petrol stations, it also serves as a route optimizer.

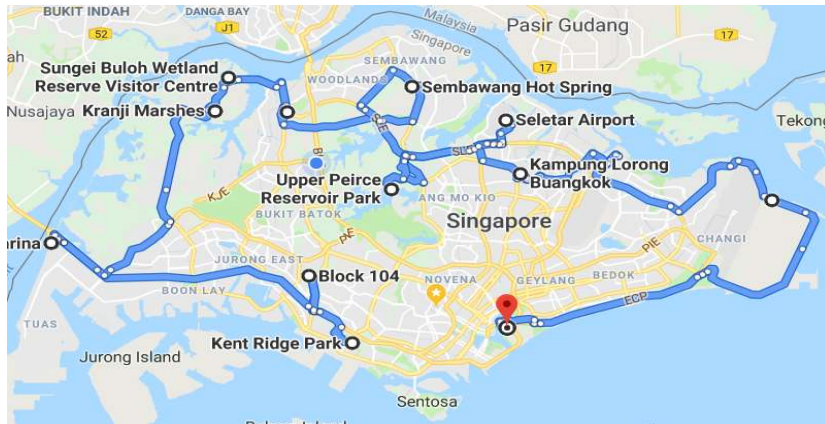
John who is a private hire car driver will certainly benefit in using this app in planning and optimizing his routes, as well as incorporating the preferred petrol station to be one of his key destinations. There are many similar private hired car drivers like John in Singapore. In fact, such app can benefit each vehicle owner in Singapore and there are about 950k vehicles in Singapore, consisting of about 550K cars and station wagons, 22K taxis, 66K private hired cars, 138K motorcycles and scooters and 180K other vehicles such as buses, good vehicles etc. This is only the Singapore market, and the app will also serve other drivers in South East Asia and beyond.



Eric, like John also travels extensively every day. He is a freelance courier who collects items or goods mainly for business clients. His daily travel routes are very varied, and resources such as time and fuel are important resources. As a freelance courier he has limited productivity and organizational app at his disposal unlike large corporations. The number of freelance couriers in Singapore is expected to grow year on year given the popularity of online shopping as well as new services that leverages on logistic services.

The freelance courier must travel between  $N$  locations. The freelance courier will visit all locations from a list, where distances between all the locations are known and each location should be visited just once. In non-urgent deliveries, the order in which the freelance courier takes is something the logistic company need not care about. However, for a freelance courier, he or she

would like to optimize his journey and determine the shortest possible route that he or she visits each location exactly once. Below is a sample route where Eric needs to drive around the island.



Each of those links between the locations has one or more weights (or the cost) attached. The cost describes how "difficult" it is to traverse between locations, and may be given, for example, by the time taken or/and distance travelled. The freelance courier wants to keep both the time travel, as well as the distance as low as possible.

However, for this proof of concept project the cost to be optimized is only the distance. In addition to the route optimization feature, another key feature that Eric will certainly need is to locate the preferred petrol station within this route for refilling of his fuel tank.



Besides the freelance courier user group, this app can also help anyone from the public to locate the preferred petrol station from the optimum route identified based on their start and end destination of a journey. Anyone from the public can also use the app to find the shortest route based on a start and end destination.

Also, if the usage of the app is very encouraging in future, it will certainly entice logistic companies, freelance couriers, retail malls, commercial entities and petrol companies to put up advertisement on this application website. This will certainly be a potential source of income to the developer of this app in future.

## 2.2 PROJECT OBJECTIVES

In this project, the team developed a hybrid reasoning system that decides the preferred petrol station which is determined based on the best discount offer from a selected credit card, and subsequently returns the optimized route based on the list of preselected destinations. All these will be illustrated in a map. The selection of the petrol station is determined using an expert system built using Clips, and together with genetic algorithm and brute force options to determine the optimized route for the respective preselected locations.

This application is developed to help drivers in route planning optimization and at the same time also finding the optimal route to the nearest petrol station with the highest discount based on the credit card they have.

## 2.3 PRODUCT OVERVIEW

The application will return the optimized route of the preselected destination plus the preferred petrol station which is determined by the expert system based on credit card selected by the user. There will also be a feature for the user to decide whether the preselected destinations shall be in a fixed order of travel or based on the system recommend order of destinations, that is categorized as no fixed order.

## 2.4 REQUIREMENTS & CONSTRAINTS

The application has the following inputs and features:

- 1) Choice of credit cards based on brand
- 2) A pull down list of type of credit card based on the selected brand
- 3) Inputs for destinations up to 10 destinations but exclude the petrol station
- 4) Choice of route optimization, my route is fixed or no fixed order
- 5) The first and end origins are fixed regardless of the choice of optimization



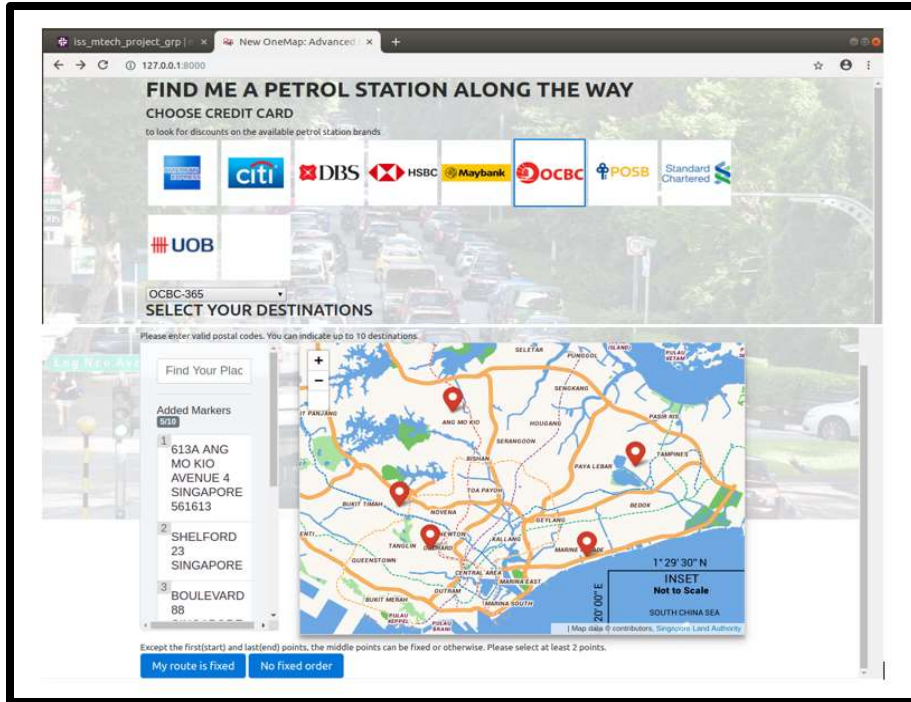


Figure 2a: The user interface screen to enter and select the necessary inputs

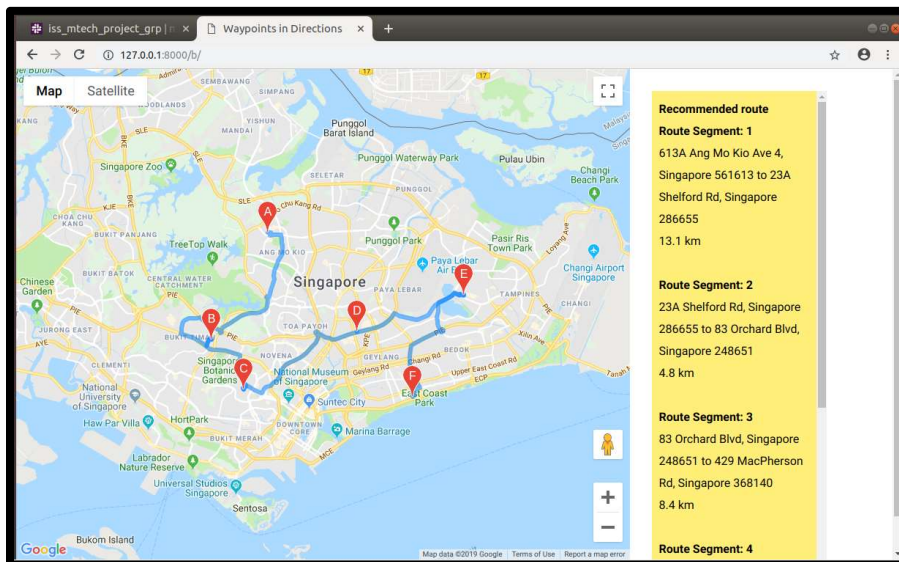


Figure 2b: Output screen of the optimized route

The recommended route will be broken down to route segments. Route segment 1 will depict the route between the origin and the next destination. Depending the number of destinations specified, the number of route segments shall be equal to the number of preselected destinations.

## 3 TECHNICAL DISCUSSION

---

### 3.1 INTRODUCTION

Credit card discount offers with petrol purchase is a common feature in Singapore petrol station landscape. Different brands of petrol stations have different tie up with these credit cards. Given the numerous offers, evaluating each of them can be cumbersome. Hence, an expert system could be used to help identify the preferred petrol station. Clips was used to develop the expert system.

In addition to the selection of the preferred petrol station, there is another challenge to be resolved, which is a variant of the Travelling Salesman Problem (TSP) but with conditions where one of the destination is the preferred petrol station and the route is open ended.

The idea behind TSP is rooted in a rather realistic problem that many people have probably actually encountered. The thesis is simple: a salesman has to travel to every single city in an area, visiting each city only once. Traditionally, he needs to end up in the same city where he starts his journey from but in this project it is an open ended route. The actual “problem” within the traveling salesman problem is finding the most efficient route for the salesman to take. There are obviously many different routes to choose from, but finding the best one—the one that will require the least cost or distance of our salesman—is what we’re solving for.

In this particular application, the selection of the petrol station is first determined using an expert system built using Clips when the user selects a specific bank card. The user then inputs his/her destinations and is subsequently given 2 options. The first option (“No Fixed Order”) optimizes with only the first and last of the preselected destinations fixed i.e. the order of the destinations in between are allowed to permute to find the shortest optimal solution. The second option (“My Route is fixed”), optimizes with the order of all the preselected destinations fixed.

Initially, we investigated the viability of implementing route optimization using the Optaplanner tool, an open-source Java constraint solver engine. Using this tool would greatly simplify the development process of the route optimization module as the underlying algorithms are already provided by Optaplanner, so we need only configure the application to interface with its API. However, significant issues were identified that would hinder the full development and integration process, so we were forced to reject this option. Subsequently, we decided to split the module into two submodules: one for each option.

For the first option (“No Fixed Order”), the search space can become extremely massive given the total number of permutations possible. For example, if the user has input a maximum of 10 destinations, the total permutations possible is  $(10-2+1)! \times 176 = 63\text{million}+$  combinations (first and last location fixed + slotting in 1 petrol station out of 176). The wait time taken to solve for the most optimal solution can be too long and hence a genetic algorithm approach has been used to determine the optimized route for this option selection.

For the second option (“My Route is fixed”), the search space is greatly reduced since the order of the preselected destinations are fixed. For example, if the user has input a maximum of 10



destinations, the total permutations possible is only  $(10-1) * 176 = 1,584$ . Hence for this particular option, a brute force approach has been used to find the shortest optimal path.

The hybrid reasoning system of leveraging a rules engine, genetic algorithm and brute force is adopted in resolving this technical project challenge.

## 3.2 TECHNICAL SOLUTIONS

### 3.2.1 DATA MINING AND COLLECTION

For this particular project, 2 sets of data were required. The first was the list of all the credit cards from all the banks in Singapore that provided specific discounts to specific petrol station brands. The second was the address and geolocations of all the petrol stations in Singapore.

To start off, a python script was developed to data mine all the credit card and respective discounts from websites such as SingSaver and MoneySmart. These credit card discount data were stored in a tabular format and subsequently used in the CLIPS decision making process.

Next, the address of all 176 stations were first data mined from the web using a python script. Given that the information on some of these petrol station websites were displayed using Java rather than plain HTML, a significant amount of effort together with some creativity was used to extract the address information of all the petrol stations in Singapore. Another python script was then developed to geocode the entire list of all the petrol stations through the One Map API. These geolocations were then stored in a Javascript array and subsequently used in the calculation of the shortest optimal distance between 2 geocode locations using the “Haversine” Formula.

### 3.2.2 CHOICE OF MAP API SERVICES

Google Map API services was first explored. It was discovered that the Mapping service provided by Google Map is comprehensive, but allows for a few free API calls. Hence an alternative mapping service, OneMap was explored. However unlike Google Map, One Map API does not allow mapping of multiple waypoints on a single map. Hence a combination of both services were adopted. OneMap was used to retrieve addresses and destinations; Google Map was used to plot the optimized route with multiple waypoints. There is a limit to 8 waypoints (not inclusive of start and end points) for Google Map direction services. Hence the project will limit the destination inputs based on this limitation.

### 3.2.3 RULES ENGINE WITH CLIPS

Most credit card with petrol discount will have tied up with all four petrol brands ( SPC, Shell, Esso and Caltex), but with varying discount rate. Considering the huge discount differences between petrol brands (sometimes up to 15%) and all petrol kiosks of each brand are scattered throughout Singapore, we assumed that a typical consumer would want to minimize cost with no preference for a particular petrol brand,

The first step was to filter out the petrol brand providing the highest discount under each credit card. Next we define the rules engine in the expert system (CLIPS) to match the credit card to the petrol brand based on the best discount.

*e.g. User selects Credit card with - AMEX-Krisflyer, the rule engine will automatically provide the petrol brand with highest discount e.g. SPC. which interface with Django function that set the list of geolocations for all SPC stations which will be one of the input factors for the Route optimization algorithms.*

### 3.2.4 ROUTE OPTIMISATION WITH OPTAPLANNER

Optaplanner is a constraint-satisfaction engine used for optimizing planning problems. It is an open-source software implemented in pure Java. It was explored as our first choice to implementing the route optimization functionality due to its lightweight and embeddable design.

To implement this tool for our problem, we approached it as an assignment problem. To illustrate, consider an example where the user has to visit five destinations, excluding the station. That implies there is a total of six stops to be made on this route:

R1 (Start) → R2 → R3 → R4 → R5 → R6 (End)

The Optaplanner implementation would need to assign all five destinations to the route, keeping the assignments for R1 and R6 fixed, as well as one station out of the pool of candidate stations such that the overall distance travelled is at a minimum.

In our implementation, we declare four entity classes: Location, Waypoint, Station and RoutePoint.

```
public abstract class Location {
    protected int id;
    protected Coordinate coord; // A wrapper class containing the lat-lon coordinates

    public double distanceFrom(Location nextLocation) {
        // Calculates distance between this and next Location using the Haversine formula
    }

    // Getters
    ...
}
```

```

public class Waypoint extends Location {
    // Waypoints are the destinations entered by the user which must be visited
    private Boolean isStart; // Indicates whether Location is the start point
    private Boolean isEnd; // Indicates whether Location is the end point

    // Getters
    ...
}

public class Station extends Location {
    private String brand; // e.g. BP, Shell, Esso etc.

    // Getters
    ...
}

@PlanningEntity
public class RoutePoint {
    // Represents one stop on the route
    private int routePosition; // Position of the RoutePoint object on the route
    private Boolean pinned;
    private Location location;

    @PlanningPin
    public Boolean isPinned() {
        return pinned;
    }

    @PlanningVariable(valueRangeProviderRefs = {"locationRange"})
    public Location getLocation() {
        return location;
    }

    public void setLocation(Location location) {
        this.location = location;
    }

    // Other getters
    ...
}

```

Of particular note are the Optaplanner annotations used in the RoutePoint class declaration. @PlanningEntity indicates that Optaplanner may modify properties of objects from this class. In our problem, we want Optaplanner to assign a Location object to each RoutePoint, so the getter method of the location attribute is annotated with @PlanningVariable (valueRangeProviderRefs will be explained later in this section). @PlanningPin is used by Optaplanner to check whether this particular RoutePoint is “pinned” i.e. if the pinned attribute is true, then Optaplanner is not allowed to change the Location already assigned to this RoutePoint object. This allows us to assign the start and end points to the route in advance and ensure that Optaplanner will not change these two assignments.

Next, we declare the `PetrolSolution` class, which is the object holding the initial problem, the working solution during the solving process, and subsequently the final solution when the solving process completes.

```
@PlanningSolution
public class PetrolSolution {
    // Matrix storing distances between each Location
    private HashMap<Integer, HashMap<Integer, Double>> distanceMatrix;
    // List of all Waypoints and Stations
    private ArrayList<Location> locationList;
    // Set of all Waypoints
    private HashSet<Waypoint> waypoints;
    // The route to be planned
    private ArrayList<RoutePoint> route;
    // Optaplanner score object
    private HardSoftScore score;

    public double getDistance(int locId1, int locId2) {
        // Obtains the distance from the distance matrix if it has already been stored
        // Otherwise, the Location.distanceFrom() method is called to calculate the
        // distance between these two Locations (identified by their id attributes)
        // before storing the result into the matrix so that it need not be recalculated
    }

    @PlanningEntityCollectionProperty
    public ArrayList<RoutePoint> getRoute() {
        return route;
    }

    @ValueRangeProvider(id="locationRange")
    @ProblemFactCollectionProperty
    public ArrayList<Location> getLocationList() {
        return locationList;
    }

    @PlanningScore
    public HardSoftScore getScore() {
        return score;
    }
}
```

In the above declaration, `HardSoftScore` is an Optaplanner class containing two score values:

- Hard score: Scoring of hard constraint violations. Hard constraints must be complied with in order for the solution to be valid.
- Soft score: Scoring of soft constraints, which are the parameters to be optimized.

`@ProblemFactCollectionProperty` annotates that `locationList`, the collection of `Locations`, contains the objects to be assigned. `@PlanningEntityCollectionProperty` annotates that `route`, the collection of `RoutePoints`, contains the objects that the aforementioned `Locations` shall be assigned to. The `@ValueRangeProvider` annotation binds `locationList` to the identifier

“locationRange”, matching the value of valueRangeProviderRefs in the RoutePoint class. This indicates to Optaplanner that only Location objects from this locationList may be assigned to the location attribute in RoutePoint objects. @PlanningScore indicates to Optaplanner the getter method for the solution’s score.

The Optaplanner API provides a few different score calculation implementations. For the initial investigation effort, the simplest Easy Java implementation is used via the PetrolEasyScoreCalculator:

```
public class PetrolEasyScoreCalculator implements EasyScoreCalculator<PetrolSolution> {
    public HardSoftScore calculateScore(PetrolSolution solution) {
        int hardScore = 0;
        int softScore = 0;

        // softScore = -(totalRouteDistance)

        // For each unique Waypoint not inside route,
        // hardScore -= 1

        // If numStations in route not equals 1,
        // hardScore -= Math.abs(numStations - 1)

        return HardSoftScore.valueOf(hardScore, softScore);
    }
}
```

In the above implementation, there are two hard constraints:

- All Waypoints must be assigned to the route.
- Exactly one Station is assigned to the route.

Hard score is 0 when all hard constraints are met. When these constraints are violated, hard score will be negative – the more negative the value, the greater the degree to which the constraints are not met.

There is one soft constraint above:

- The total distance of the entire route.

Optaplanner seeks to maximise the soft score value, hence totalRouteDistance is negated so that shorter distances result in higher scores.

During the solving process, Optaplanner modifies the working solution in steps (i.e. each step represents either one switch of an assigned Location with an unassigned Location, or one swap of Locations between two RoutePoints). The resultant change in score is used as the heuristic to determine if the move is in the right direction. Optaplanner prioritises to meet all hard constraints (i.e. achieve a hardScore of 0) over optimizing soft constraints (i.e. maximise value of softScore). As such, a hardScore:softScore of -1:-100 is considered by Optaplanner to be worse than a score of 0:-500.

The final piece is a XML configuration file defining various important parameters:

```
<?xml version="1.0" encoding="UTF-8"?>
```



```

<solver>
  <!-- Domain model configuration -->
  <scanAnnotatedClasses/>

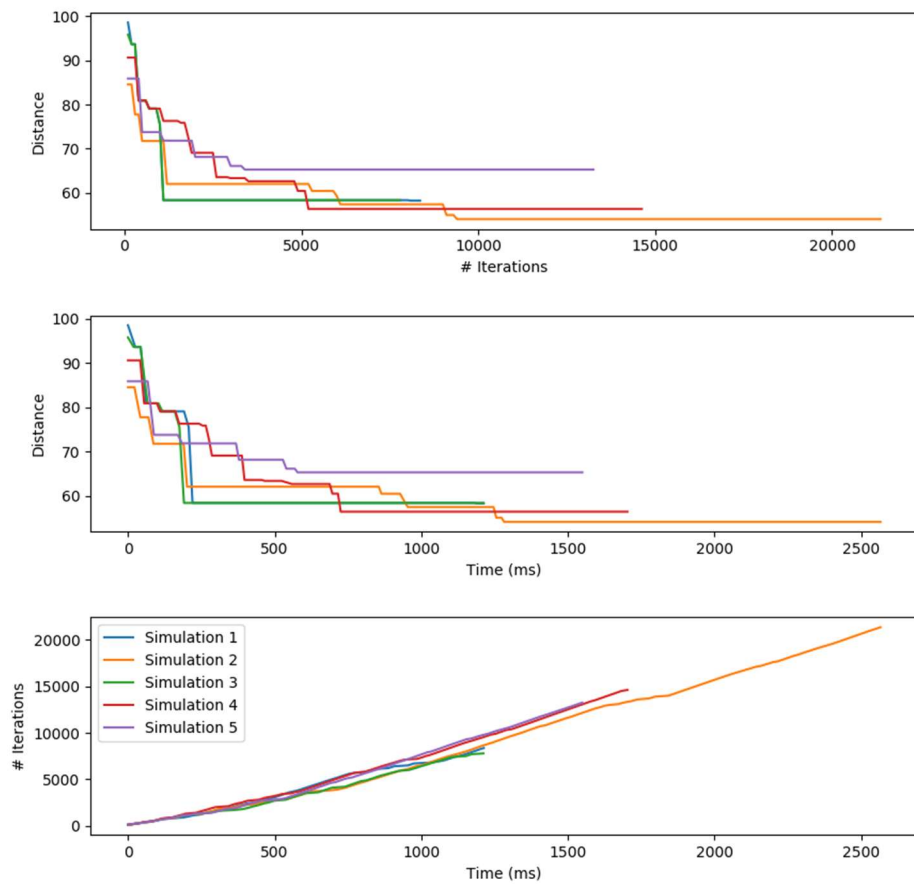
  <!-- Score configuration -->
  <scoreDirectorFactory>
    <easyScoreCalculatorClass>rs_petrol.solvers.PetrolEasyScoreCalculator</easyScoreCalculatorClass>
  </scoreDirectorFactory>

  <!-- Optimisation algorithms configuration -->
  <termination>
    <terminationCompositionStyle>OR</terminationCompositionStyle>
    <secondsSpentLimit>3</secondsSpentLimit>
    <unimprovedMillisecondsSpentLimit>1000</unimprovedMillisecondsSpentLimit>
  </termination>
</solver>

```

The above configuration file informs Optaplanner to parse all the annotations in the code, and identifies the specific score calculation class to use. The termination condition is also defined: Execution will stop when either 3 seconds have elapsed in total, or 1000ms (1 second) has elapsed without any improvement in the score.

To test the performance of the above implementation, various simulations are run on different sets of data, varying between 5-15 Waypoints and 50-200 Stations. First, several simulations are run for a control data set of 10 Waypoints and 50 Stations. The performance of each simulation is visualized in the charts below.



*Figure 3a: Optaplanner performance simulation 1*

As seen in the above chart, Optaplanner makes the most improvement to the route distance within less than 5,000 steps or iterations, and within 1 second. The distance generally plateaus within 10,000 iterations, and usually within 1.5 seconds. The time taken to execute each iteration remains largely constant.

Next, performance is compared between 5, 10, and 15 Waypoints.

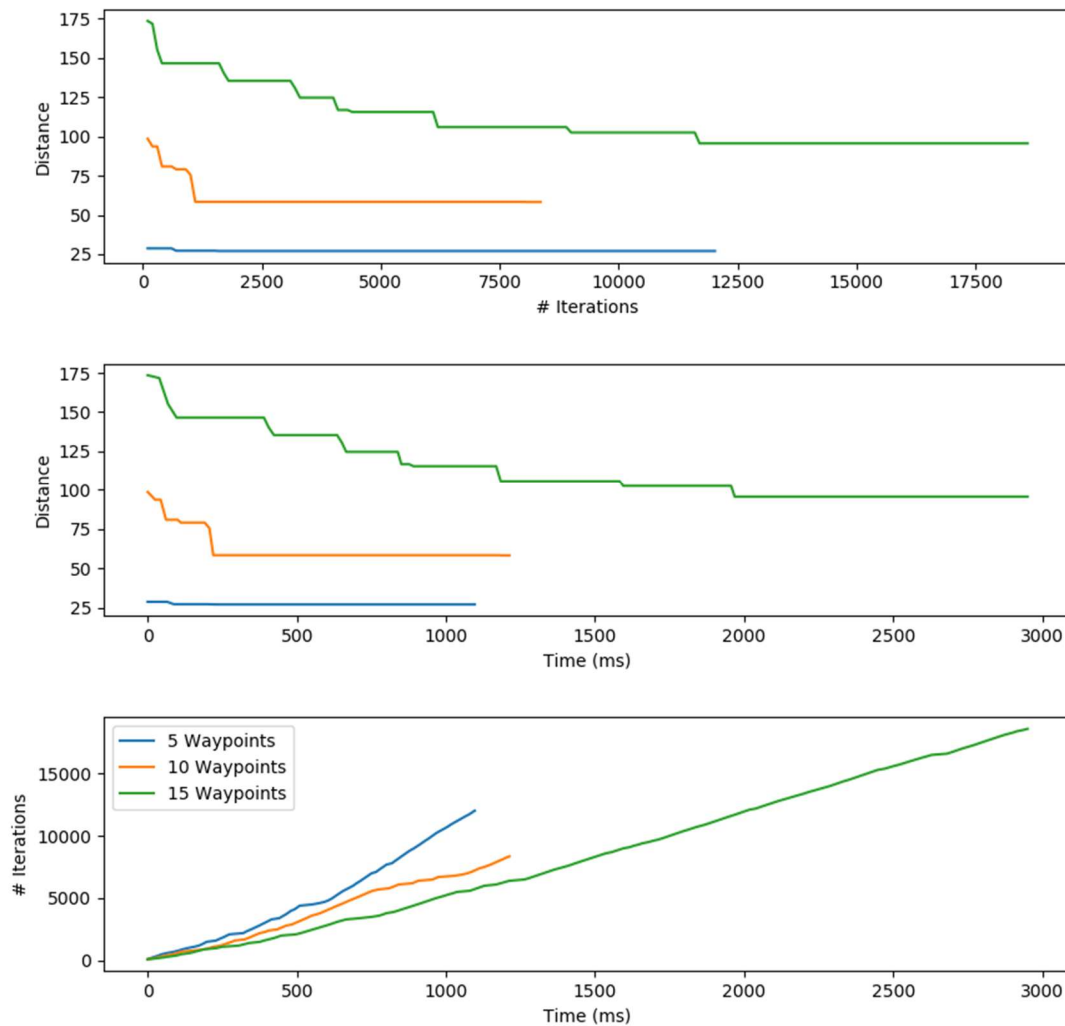
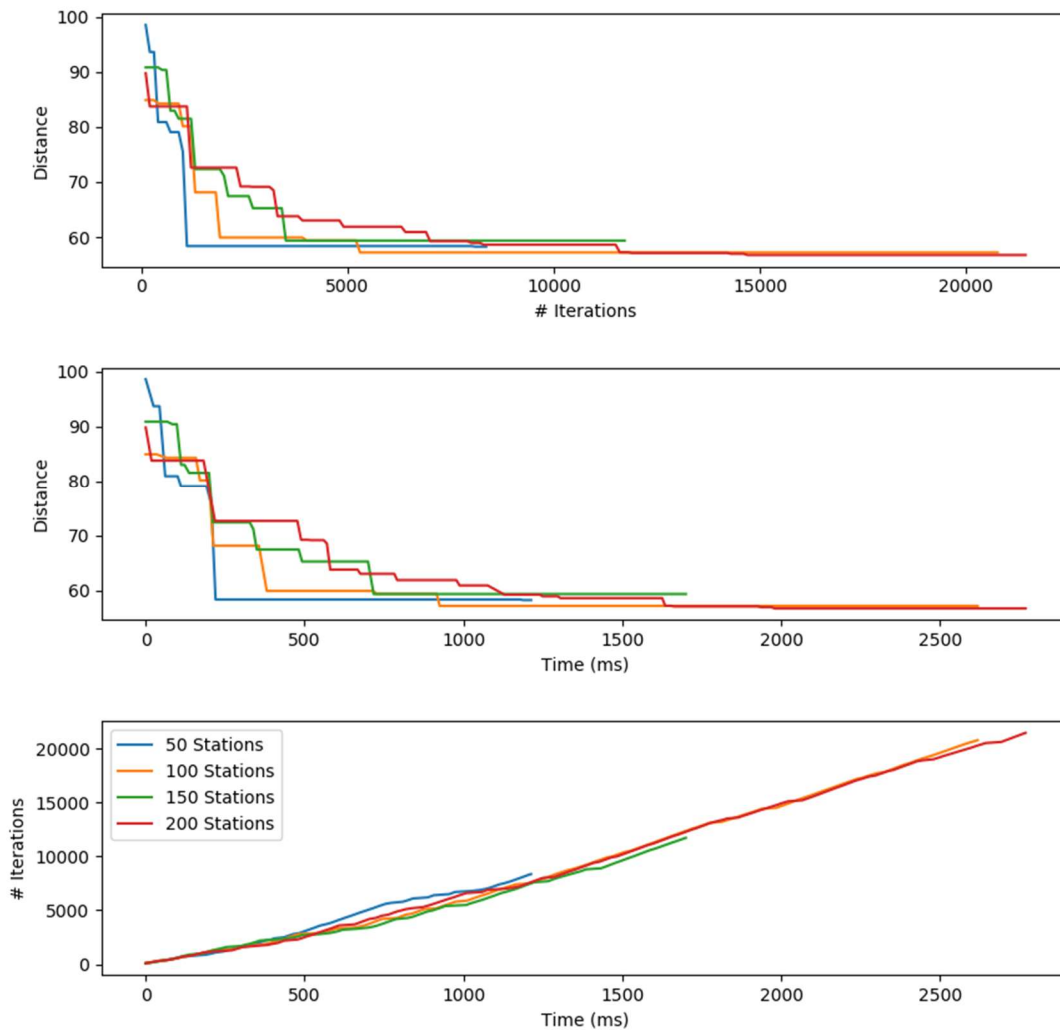


Figure 3b: Optaplanner performance simulation 2

When the route consists of 5 or 10 Waypoints, a near-optimal solution can be found within less than 2,000 iterations and less than 300 ms. However, as the route increases in length to 15 Waypoints, performance decreases, as Optaplanner continues to discover better solutions up until the total time limit of 3 seconds is reached.

Finally, performance is compared between 50, 100, 150 and 200 Stations.



*Figure 3c: Optaplanner performance simulation 3*

The above results show that as the size of the Station dataset increases, the time taken to achieve a near-optimal solution increases as expected, at a slightly higher than proportionate rate. Nonetheless, improvements to the distance still plateau within less than 10,000 iterations and approximately up to 1 second.

The above results show that, for the size of data our application can expect to process, solving times to reach a near-optimal solution can generally be achieved within 1-1.5 seconds, well within our application's time constraints. Thus far, Optaplanner appears to be an excellent choice for implementing our route optimization functionality.

On further analysis, however, our team encountered significant difficulty in integrating Optaplanner with the rest of our application's modules. Additionally, more work needs to be done to implement other constraints into the score calculation, such as brand filtering etc. After assessing the time and effort required, based on our team's relevant skillsets and proficiencies, we concluded that there is a risk of being unable to complete the Optaplanner's final implementation within the development time available to us. There, we had rejected the use of Optaplanner to implement our route optimization functionality.

### 3.2.5 BRUTE FORCE SEARCH WITH JAVASCRIPT ("My Route is fixed")

In this particular option, the order of all the preselected destinations are fixed. The Javascript first receives the geolocation of all preselected destinations. Together with the previously stored Javascript array of petrol station geolocations, the script iterates through every single possible combination to find the shortest optimal path using the "Haversine" Formula. Because of the much reduced search space for this particular approach, the application is able to find the most optimal solution within a short time even with 10 preselected destinations  $(n-1) * 176 = 1,584$ . Where  $n$  is the number of preselected destinations by the user.

*e.g. With 3 preselected destinations A, B, C by the user, these are the possible outcomes. Where S is the petrol Station, A-S-B-C, A-B-S-C with 176 possible selections of S.  $(3-1) * 176 = 352$*

With the optimal solution derived from this brute force iteration, the geolocations of all the destinations + 1 Petrol station is then passed on to Google Map API and subsequently mapped out.

### 3.2.6 GENETIC ALGORITHM WITH PYTHON ("No Fixed Order")

The Genetic Algorithm(GA) is explored as one of the means to solve the optimization problem for finding the shortest route from the starting point, covering all the intermediate points as required by the user, finding the nearest petrol station on the route and terminating at the end point.

The requirements are similar but not identical to the Travelling Salesman Problem (TSP), with the following differences:-

1. TSP covers all the points in the data set with no particular order, however here we have a specific starting point and a specific end point
2. All the points in TSP are reached once and the path is closed as the starting point and end point are the same. Here the two points are different and therefore the final path is an open line and not closed.
3. The final path will have only one petrol station and not all the petrol stations in the available set.



With the above requirements in mind, we have developed a GA to solve the problem and modified a few parts of the generalized TSP solution. The code was developed in Python.

## Definitions

The following is the definition of the terms and concepts used in the GA

- **Gene:** an intermediate or starting or end point defined in (x,y) co-ordinates for latitude and longitude.
- **Individual (“chromosome”):** one of the routes satisfying the requirements :-
  - Fixed start point
  - Reach all the intermediate points en-route exactly once
  - Include one of the possible petrol station locations en-route as an intermediate point
  - Fixed end point
- **Population:** a collection of possible routes (i.e., collection of individuals) from the above
- **Parents:** two routes that are combined to create a new route
- **Mating pool:** a collection of parents that are used to create our next population (thus creating the next generation of routes)
- **Fitness:** a function that tells us how good each route is. Here it is the shortest distance.
- **Mutation:** introduce variation in our population by randomly swapping two intermediate points in the route
- **Elitism:** carry the best individuals into the next generation

## Basic Classes Used

The loc class create and handle the waypoints, including those for the starting point, end point, intermediate points and the petrol stations. These (x, y) coordinates for the latitude and the longitude of the location. Within the Loc class, we add a distance calculation using the haversine formula to account for geolocations. We will create a dummy node connecting the start point and the end point and will set the distance from dummy node connecting these two points to 0.

```
class Loc:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, loc):
        if((self.x==startx)and(self.y==starty)and(loc.x==10000)):
            dist=0
            return dist
```

```

if((loc.x==startx)and(loc.y==starty)and(self.x==10000)):
    dist=0
    return dist
if((self.x==endx)and(self.y==endy)and(loc.y==10000)):
    dist=0
    return dist
if((loc.x==endx)and(loc.y==endy)and(self.y==10000)):
    dist=0
    return dist
    radlat1 = math.pi * self.x/180
radlat2 = math.pi * loc.x/180
theta = self.y-loc.y
radtheta = math.pi * theta/180
dist = math.sin(radlat1) * math.sin(radlat2) + math.cos(radlat1) * math.cos(radlat2) * math.cos(radtheta)
if (dist > 1):
    dist = 1

dist = math.acos(dist)
dist = dist * 180/math.pi
dist = dist * 60 * 1.1515
dist = dist * 1.609344
if((self.x==startx)and(self.y==starty)):
    if (loc.x==10000):
        dist=0
    if ((loc.x==startx)and(loc.y==starty)):
        if (self.x==10000):
            dist=0
    if((self.x==endx)and(self.y==endy)):
        if (loc.y==10000):
            dist=0
    if ((loc.x==endx)and(loc.y==endy)):
        if (self.y==10000):
            dist=0

return dist

def __repr__(self):
    return "(" + str(self.x) + "," + str(self.y) + ")"

```

The Fitness class calculates the fitness score of the route. In our case, the fitness score is the inverse of the route distance. We want to minimize route distance, so a larger fitness score is better.

```

class Fitness:
    def __init__(self, route):
        self.route = route
        self.distance = 0
        self.fitness= 0.0

    def routeDistance(self):
        if self.distance ==0:
            pathDistance = 0
            for i in range(0, len(self.route)):

```

```

        fromLoc = self.route[i]
        toLoc = None
        if i + 1 < len(self.route):
            toLoc = self.route[i + 1]
        else:
            toLoc = self.route[0]
        pathDistance += fromLoc.distance(toLoc)
        self.distance = pathDistance
        return self.distance

    def routeFitness(self):
        if self.fitness == 0:
            self.fitness = 1 / float(self.routeDistance())
        return self.fitness

```

### Treatment of starting and endpoints

In order to customise the TSP problem to one having a fixed starting and ending point, we tried the following 2 approaches:-

1. Fixed the start point and the end point and performed the TSP iterations for all the intermediate points
2. Created a dummy node connecting the starting and the ending point with 0 distance for each of the two legs and ran the normal TSP across the entire set.

Create the population

Next, we create the population(first generation). To do so, we need have created a function that produces routes that satisfy our conditions. To create an individual, we randomly select the order in which we visit each city:

```

def createRoute(locList):
    route = random.sample(locList, len(locList))
    return route

```

This produces one individual, but we want a full population, so in our next function, we loop through the createRoute function until we have as many routes as we want for our population.

```

def initialPopulation(popSize, locList):
    population = []
    for i in range(0, popSize):
        population.append(createRoute(locList))
    return population

```

### Determine fitness

To simulate our “survival of the fittest”, we make use of Fitness to rank each individual in the population. Our output will be an ordered list with the route IDs and each associated fitness score. Each of the routes has the starting point and the ending point next to each other, with the intermediate dummy node.

```
def rankRoutes(population):
    fitnessResults = {}
    for i in range(0, len(population)):
        fitnessResults[i] = Fitness(population[i]).routeFitness()
    return sorted(fitnessResults.items(), key = operator.itemgetter(1), reverse = True)
```

### Select the mating pool

We have used “Fitness proportionate selection” method to select the parents that will be chosen to create the next generation. The fitness of each individual relative to the population is used to assign a probability of selection. With elitism, the best performing individuals from the population will automatically carry over to the next generation, ensuring that the most successful individuals persist.

We have therefore created the mating pool in two steps. First, we’ have used the output from rankRoutes to determine which routes to select in our selection function. Then we have compared a randomly drawn number to these weights to select our mating pool. We’ll also want to hold on to our best routes, so we introduce elitism. Ultimately, the selection function returns a list of route IDs, which we can use to create the mating pool in the matingPool function.

```
def selection(popRanked, eliteSize):
    selectionResults = []
    df = pd.DataFrame(np.array(popRanked), columns=["Index", "Fitness"])
    df['cum_sum'] = df.Fitness.cumsum()
    df['cum_perc'] = 100*df.cum_sum/df.Fitness.sum()
    for i in range(0, eliteSize):
        selectionResults.append(popRanked[i][0])
    for i in range(0, len(popRanked) - eliteSize):
        pick = 100*random.random()
        for i in range(0, len(popRanked)):
            if pick <= df.iat[i,3]:
                selectionResults.append(popRanked[i][0])
                break
    return selectionResults
```

Now that we have the IDs of the routes that make up our mating pool from the selection function, we have created the mating pool by simply extracting the selected individuals from our population.

```
def matingPool(population, selectionResults):
    matingpool = []
    for i in range(0, len(selectionResults)):
        index = selectionResults[i]
        matingpool.append(population[index])
    return matingpool
```

## Breed

With our mating pool created, we create the next generation in a process called **crossover** (or “breeding”). The TSP needs to include all locations exactly one time. To abide by this rule, we can use a special breeding function called **ordered crossover**. In ordered crossover, we randomly select a subset of the first parent string and then fill the remainder of the route with the genes from the second parent in the order in which they appear, without duplicating any genes in the selected subset from the first parent.

```
def breed(parent1, parent2):
    child = []
    childP1 = []
    childP2 = []

    geneA = int(random.random() * len(parent1))
    geneB = int(random.random() * len(parent1))

    startGene = min(geneA, geneB)
    endGene = max(geneA, geneB)

    for i in range(startGene, endGene):
        childP1.append(parent1[i])

    childP2 = [item for item in parent2 if item not in childP1]

    child = childP1 + childP2
    return child
```

Next, we have generalized this to create our offspring population.

```
def breedPopulation(matingpool, eliteSize):
    children = []
    length = len(matingpool) - eliteSize
    pool = random.sample(matingpool, len(matingpool))

    for i in range(0, eliteSize):
        children.append(matingpool[i])

    for i in range(0, length):
        child = breed(pool[i], pool[len(matingpool)-i-1])
        children.append(child)
    return children
```

## Mutate

Mutation is to avoid local convergence by introducing novel routes that will allow us to explore other parts of the solution space. Similar to crossover, the TSP has a special consideration when it comes to mutation. To abide by the rules, we cannot drop the intermediate points. Therefore we'll use **swap mutation**. This means that, with specified low probability, two cities will swap places in our route. We'll do this for one individual in our mutate function:



```
def mutate(individual, mutationRate):
    for swapped in range(len(individual)):
        if(random.random() < mutationRate):
            swapWith = int(random.random() * len(individual))

            loc1 = individual[swapped]
            loc2 = individual[swapWith]

            individual[swapped] = loc2
            individual[swapWith] = loc1
    return individual
```

Next, we have extended the mutate function to run through the new population.

```
def mutatePopulation(population, mutationRate):
    mutatedPop = []

    for ind in range(0, len(population)):
        mutatedInd = mutate(population[ind], mutationRate)
        mutatedPop.append(mutatedInd)
    return mutatedPop
```

Now we repeat the following steps.

- Rank the routes in the current generation using rankRoutes.
- Determine the potential parents by running the selection function, which allows us to create the mating pool using the matingPool function.
- Create new generation using the breedPopulation function and then applying mutation using the mutatePopulation function.

```
def nextGeneration(currentGen, eliteSize, mutationRate):
    popRanked = rankRoutes(currentGen)
    selectionResults = selection(popRanked, eliteSize)
    matingpool = matingPool(currentGen, selectionResults)
    children = breedPopulation(matingpool, eliteSize)
    nextGeneration = mutatePopulation(children, mutationRate)
    return nextGeneration
```

We need a function to clean the final list by removing the temporary link node and arranging the route from the start point to the end point. This is done by the clean\_list function.

We finally run the algorithm for all the intermediate points, including the starting point and the ending point. There is a set of possible petrol stations, so we run this for each petrol station, one at the time and keep track of the 3 shortest paths for each, which are finally compared to provide the top 3 results.

```

final_list=[]
curlist=[]
for i in range(0, len(Petrolcor)):
    curlist=locList[:]
    curlist.append(Loc(x=Petrolcor[i][0], y=Petrolcor[i][1]))

    popSize=100
    eliteSize=20
    mutationRate=0.01
    generations=500
    population=curlist
    pop = initialPopulation(popSize, population)

    for i in range(0, generations):
        pop = nextGeneration(pop, eliteSize, mutationRate)

    curlist=[]
    Shortest_distance1=1 / rankRoutes(pop)[0][1]
    bestRouteIndex1 = rankRoutes(pop)[0][0]
    bestRoute1 = pop[bestRouteIndex1]
    Shortest_distance2=1 / rankRoutes(pop)[1][1]
    bestRouteIndex2 = rankRoutes(pop)[1][0]
    bestRoute2 = pop[bestRouteIndex2]
    Shortest_distance3=1 / rankRoutes(pop)[2][1]
    bestRouteIndex3 = rankRoutes(pop)[2][0]
    bestRoute3 = pop[bestRouteIndex3]

    final_list.append((Shortest_distance1, bestRoute1))

    final_list.append((Shortest_distance2, bestRoute2))
    final_list.append((Shortest_distance3, bestRoute3))

```

## Simulation results with live coordinates

We ran the GA with the following sample location coordinates

```

Loccor=[[1.431767,103.709027],[1.306836,103.901068],[1.266119,103.802384]
,[1.309067,103.915058], [1.285065,103.849206], [1.290508,103.768218],
[1.351713,103.870426]]

#Startloc is the coordinates of the starting point
Startloc=[1.363019,103.970202]

# Endloc is the coordinates of the end point
Endloc=[1.305892,103.786977]

#Petrolcor is the coordinates of the petrol stations
Petrolcor=[[1.328276,103.813597],[1.44244103727,103.776200994],[1.365289,103.839673],[1.36449,103.84592],
[1.378994,103.836168],[1.3266884782029,103.84788192046],[1.3278458229585,103.93984413714],
[1.28408372439,103.818506849],[1.318023,103.831998],[1.383397,103.772767],[1.3185147308667,103.9087617346
1],

```

```
[1.305726,103.795075],[1.323107,103.819387],[1.2884418507336,103.83815013614],[1.368597016813,103.8843379
2135],

[1.3146267758284,103.7146202756],[1.371972,103.828816],[1.3423035906473,103.73729182853],[1.297636,103.83
8181],

[1.3318422790676,103.88041106101],[1.3020768186352,103.88435589325],[1.2769678624928,103.79068634598],

[1.3708755393005,103.96031413222],[1.400355,103.909162],[1.2913261708666,103.80089438566],[1.293097,103.8
25994],[1.43999072142,103.825246311],

[1.391207,103.898967],[1.41003,103.90099],[1.348636,103.938567],[1.308243296161,103.89477060979],[1.2717507
6934,103.807202523],

[1.3247173754161,103.84163668491],[1.3357392758618,103.85558287997],[1.313866,103.930981],[1.35836174913
09,103.88286422548],[1.348829,103.837778],
[1.357567,103.87521],[1.4174586657123,103.83783562788]]
```

The simulation results with the top 3 routes are as hereunder:-

```
The length of rank 1 is 71.5035100245
The route of rank : 1 is
xcor is 1.363019 and ycor is 103.970202
xcor is 1.309067 and ycor is 103.915058
xcor is 1.306836 and ycor is 103.901068
xcor is 1.351713 and ycor is 103.870426
xcor is 1.285065 and ycor is 103.849206
xcor is 1.266119 and ycor is 103.802384
xcor is 1.290508 and ycor is 103.768218
xcor is 1.431767 and ycor is 103.709027
xcor is 1.328276 and ycor is 103.813597
xcor is 1.305892 and ycor is 103.786977
The length of rank 2 is 71.5035100245
The route of rank : 2 is
xcor is 1.363019 and ycor is 103.970202
xcor is 1.309067 and ycor is 103.915058
xcor is 1.306836 and ycor is 103.901068
xcor is 1.351713 and ycor is 103.870426
xcor is 1.285065 and ycor is 103.849206
xcor is 1.266119 and ycor is 103.802384
xcor is 1.290508 and ycor is 103.768218
xcor is 1.431767 and ycor is 103.709027
xcor is 1.328276 and ycor is 103.813597
xcor is 1.305892 and ycor is 103.786977
The length of rank 3 is 71.5035100245
The route of rank : 3 is
xcor is 1.363019 and ycor is 103.970202
xcor is 1.309067 and ycor is 103.915058
xcor is 1.306836 and ycor is 103.901068
xcor is 1.351713 and ycor is 103.870426
xcor is 1.285065 and ycor is 103.849206
xcor is 1.266119 and ycor is 103.802384
xcor is 1.290508 and ycor is 103.768218
xcor is 1.431767 and ycor is 103.709027
xcor is 1.328276 and ycor is 103.813597
xcor is 1.305892 and ycor is 103.786977
```

### Identification of Parameters for optimization

In order to identify the optimum parameters for running the GA program, the program was run on the sample set of parameters for different combinations of the number of generations and population size. The results are as follows and the optimization was done for the run-time and quality of results.

Population Size	Generation Size	Run time	Shortest Route	Route * time
30	30	3.868	68.145	263.579
30	40	4.406	70.443	310.367
40	30	5.144	67.957	349.579
30	50	5.868	68.145	399.843
40	40	6.311	67.779	427.773
50	30	6.830	67.779	462.921
30	70	7.760	67.765	525.872
40	50	8.856	67.779	600.262
50	40	9.002	67.765	610.044
40	70	11.300	67.765	765.754
30	100	11.394	67.779	772.238
50	50	11.453	67.779	776.299
70	30	11.531	67.765	781.426
70	40	15.720	67.765	1065.251
50	70	15.807	67.765	1071.134
40	100	16.408	67.765	1111.901
70	50	19.462	67.765	1318.859
100	30	21.809	67.765	1477.918
30	200	22.346	67.788	1514.791
50	100	22.936	67.779	1554.560
70	70	27.836	67.765	1886.289
100	40	28.935	67.765	1960.798
40	200	32.725	67.765	2217.638
100	50	36.518	67.765	2474.652
70	100	40.079	67.765	2715.982
50	200	45.753	67.779	3101.076
100	70	52.745	67.765	3574.290
100	100	74.755	67.765	5065.762
200	30	79.547	67.765	5390.479
70	200	80.402	67.765	5448.446
200	40	106.925	67.765	7245.754
200	50	134.594	67.765	9120.794
100	200	150.071	67.765	10169.585
200	70	205.059	67.765	13895.855
200	100	275.635	67.765	18678.374
200	200	559.031	67.765	37882.727

## 4 SYSTEM ARCHITECTURE

### 4.1 PROJECT SCOPE

The hybrid reasoning system is implemented using CLIPS together with genetic algorithm built using python, and the overall system is brought to life using Python programming language (Django) in the form of a web-based graphical user interface which users can easily interact with.

Figure 3, the system architecture diagram, illustrates how the application in the front-end has been interfaced with the back-end rule-based system, lightweight database and rules based system and python genetic algorithm.

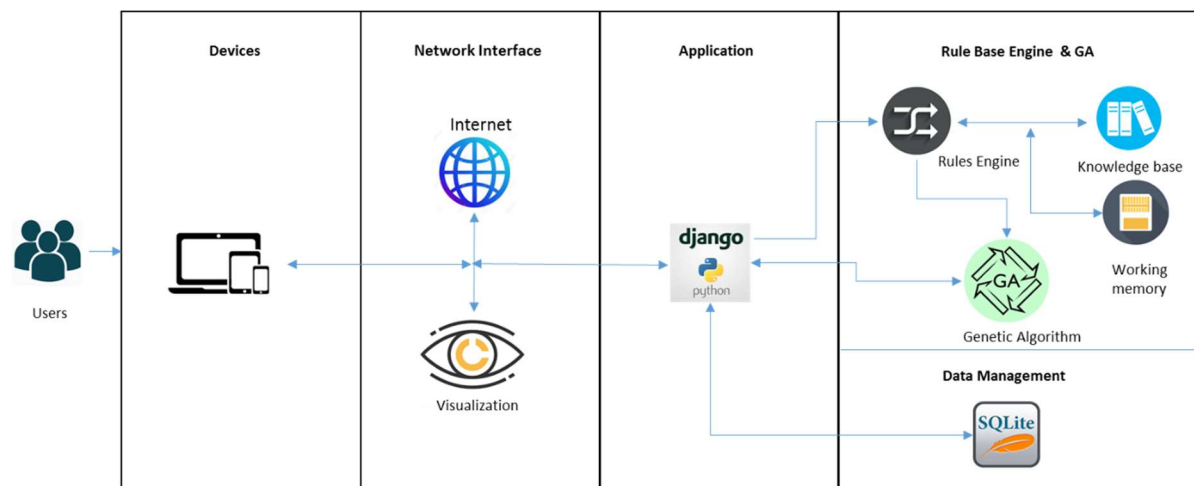


Figure 4a: Hybrid Reasoning System - Petrol Station Route Optimizer

Its scope is limited by the data available to-date in terms of the coordinates of the petrol stations, discounts of petrol offered by the respective credit cards at that time, and google api returns. The Clips expert system recommends a preferred petrol station that is determined from the preselected credit card. This petrol station and preselected destinations are inputs to the genetic algorithm which will eventually return the optimized route.



## 4.2 ASSUMPTIONS

The following data and apis were used for the development of this hybrid reasoning system:

- (i) Locations of the respective brand of petrol stations
- (ii) Discounts offers of each credit card based on the petrol stations
- (iii) One Map and Google Map APIs

The above resources are deemed to be current based on the team's collation efforts in April 2019.

## 4.3 SYSTEM'S FEATURES

Despite the reduced scope and assumptions, the team has gone through an in-depth thought process to implement significant features in the application which can substantially add value to potential users. The application is built on a web-based application with the consumers in mind. Anyone can access it easily via as a personal computer, mobile phones or tablets via the internet.

In addition, the interface is designed to provide users with the best browsing experience. For instance, the display window is dynamic and adjusts to the type of devices which the users are using. Drop down list were also deployed to improve the ease of users' inputs. Input checks are also in place to validate the information keyed in by the user. The key features are as follows:

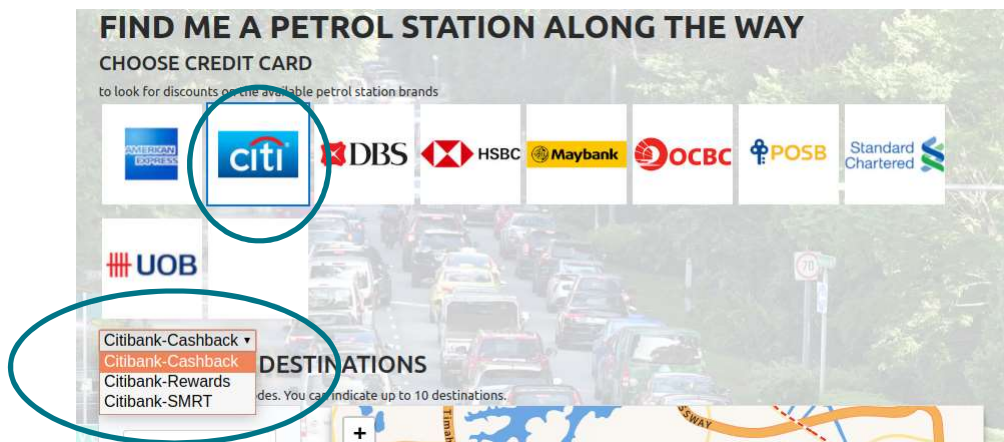


Figure 4b: Credit card icons for the user to select as well as a drop-down list of card types

### SELECT YOUR DESTINATIONS

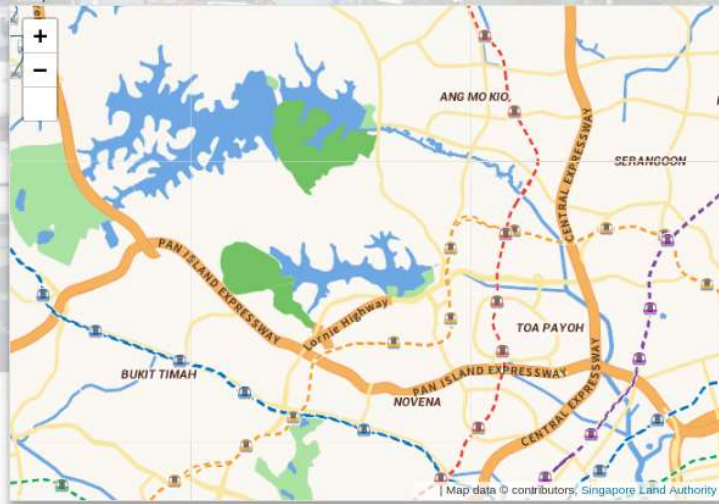
Please enter valid postal codes. You can indicate up to 10 destinations.

Find Your Place

No markers added

Type a location in the search bar and select one from the dropdown list

Marker-to-marker routing can be enabled when there is more than 1 marker present



Except the first(start) and last(end) points, the middle points can be fixed or otherwise. Please select at least 2 points.

My route is fixed

No fixed order

Figure 4c: Select destinations by entering the “Find Your Place” field

Users will just need to key in the addresses of destination at the fields one at the time. A pop-up dialog box will ask the user to confirm the destination before the location icon are placed into the map.

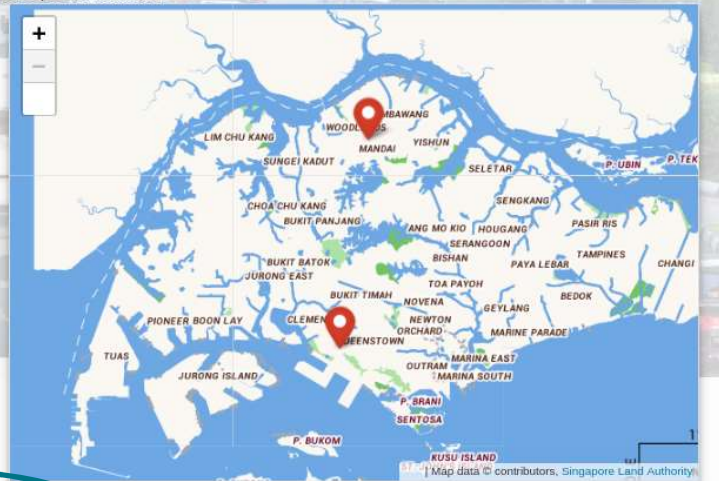
### SELECT YOUR DESTINATIONS

Please enter valid postal codes. You can indicate up to 10 destinations.

Added Markers 2/10

1  
21 LOWER KENT RIDGE ROAD  
NATIONAL UNIVERSITY OF SINGAPORE (LT21)  
SINGAPORE 119077

2  
574 WOODLANDS DRIVE 16  
WOODLANDS GLEN  
SINGAPORE 700011



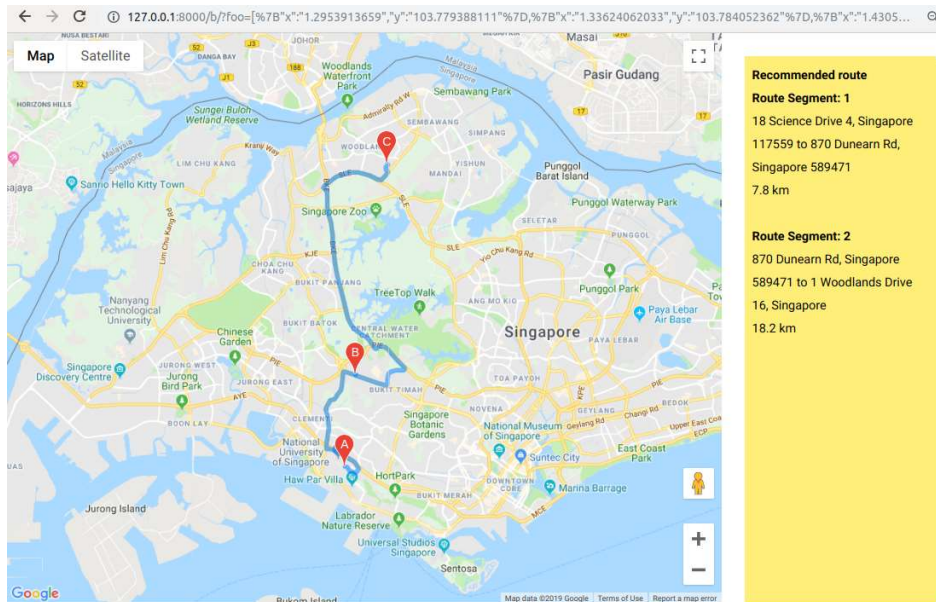
Except the first(start) and last(end) points, the middle points can be fixed or otherwise. Please select at least 2 points.

My route is fixed

No fixed order

Figure 4d: Select either the My route is fixed or No fixed order button

Ten markers or destinations are being provided to the users to enter. Minimum, 2 destinations must be provided before the option of “My route is fixed” and “No fixed order” optimization button can function appropriately. An error will occur if this is not complied. The recommended route will be displayed on google map, and the respective route segments are displayed on the right side of the map. Refer to *Figure 4d* for details.



*Figure 4d: Recommended Routes with respective route segments*

Icon A and C are the designated locations entered by the user. Icon B is the recommended petrol station.

## 4.4 LIMITATIONS

The entire hybrid reasoning system is recommending the optimized route with the preferred petrol station. However, the system is developed with a proof concept basis, and thus have limitations. Some of the key limitations are the processing time of the “No Fixed Order” solution which leverages on the genetic algorithm to determine the optimized path. Further parameter tuning may be required to improve the performance. The solution heavily relies on one map and google api services especially in terms of mapping and locations data. The solution has yet to consider other factors such as traffic that may impact the preferred recommended route. Also, the distances input into the genetic algorithm are not actual route distances but straight-line distances, i.e. Euclidean distances.

## 5 CONCLUSION & REFERENCES

---

### 5.1 IMPROVEMENTS AND FUTURE ENHANCEMENTS

Given that the application is a proof of concept project, the team had brainstormed and are looking forward to incorporating the following enhancements subject to availability of resources:

- 1) Provide an option to select brand of petrol station rather than credit card type
- 2) Provide option for optimized route without petrol stations
- 3) Labelled petrol station icon on the map
- 4) Enhance processing time for the non-fixed order route
- 5) Voice enabled user interface to improve user experience for driver on the road
- 6) Implementation of the route optimization module using Optaplanner for improved efficiency, scalability, output quality, and simplicity of development.

## 6 BIBLIOGRAPHY

---

- 1) Evolution of a salesman: A complete genetic algorithm tutorial for Python  
<https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35>
- 2) Was Darwin a Great Computer Scientist? How evolution taught us the “genetic algorithm” <https://blog.sicara.com/getting-started-genetic-algorithms-python-tutorial-81ffa1dd72f9>
- 3) What TSP variant doesn't return to start point?  
<https://cs.stackexchange.com/questions/43549/what-tsp-variant-doesnt-return-to-start-point>
- 4) New One Map  
<https://www.onemap.sg/home/>
- 5) Google Map Platform  
<http://bit.do/ePYQY>
- 6) Clips: A tool for building expert systems  
<http://www.clipsrules.net/>
- 7) Background image of Singapore roads  
<https://www.straitstimes.com/singapore/meet-hajjah-and-boon-tat-10-singapore-roads-named-after-someones-grandmother-or>
- 8) Best Apps for Finding The Nearest Gas Station Offline  
[https://www.youtube.com/watch?v=Qx99p-t\\_VZs](https://www.youtube.com/watch?v=Qx99p-t_VZs)

## 7 APPENDICES

---

- 1) \Miscellaneous\RS Optaplanner.7z  
3.2.4 Optaplanner codes
- 2) \Miscellaneous\rs\_python\_ga.7z  
Second version of GA in python with added functionalities

