Facility location problem using genetic algorithm.

Kavin R V, Shashank Singh, Yash Saxena
IISER- Bhopal

Abstract- Since industrial developments, there has been an alarming rise in fire accidents, specifically in highly dense and industrial areas. Due to the lack of well-planned location-allocation, damage caused by these incidents are significant. The facility location problem has become the preferred approach for dealing with such emergency problems. To solve this problem,

the primary approach we use here is the genetic algorithm. We have used this algorithm, on the dataset containing potential demand locations and facility locations for the city of Mumbai, India. We have utilised data provided to predict the potential locations for the fire stations and number of vehicles required in them to handle an emergency situation in the nearby area. We have done this on the basis of the fact that the demand locations can be of types, commercial, residential or marketplace. Depending on the location, the demand for the number of fire stations and rescue vehicles required is used to accurately predict the location of fire stations. We have used Genetic algorithm to find the optimal locations. The results we have obtained are convincing enough to be used for the fire station location planning in Mumbai and various other locations facing such a problem and searching for such a robust solution.

Keywords: Facility Location Planning, Genetic Algorithm, Firestations, Demand Locations, Potential Locations, Hubs

---

1.Introduction:

In India, many deaths are caused due to delays in rescue vehicles reaching the required location.

There is a shortage of emergency fire services in India. There are only 3377 fire stations against the required 8859 fire stations. There is also a 78% shortage of fire fighting equipment or vehicles in fire stations.

The delay in the required help to reach the demand location is also one of the major reasons for the higher number of deaths in India. To add-on, the narrow roads and incompatible routes cause the delay and sometimes act as obstacles for the vehicles to reach the demand

location. This causes the fire officers to carry some equipment manually. This further delays the rescue operations and puts the officers' life at risk.

We have used a genetic algorithm to deal with the problem of allocating a mini station for serving the demand location from a potential distribution point (mini station). Such that any demand point is served by its nearest potential station and is served solely by that particular station, considering constraints like time travel and the compactness of the route.

2. Data Details:

The data focuses on the following four datasets.

2.1 *Demand data file:*

Contains information about the rescue vehicles requirements of various points across three different time intervals 4am - 8 am, 8 am - 12 pm and 12 pm - 16 pm .

2.2 *Potential locations data file:*

Contains information regarding various points that can be a potential location.

2.3 *Shapefiles data set:*

Plotting this data using a 3D visualization tool or some computational geometry method gives an idea of the location and how the points are distributed over a map.

2.4 *Dealing with the data:*

	fire_truck	rescue_tru	water_tank	abulance	Total	Y	X	ID	Fixed
0	1	1	1	1	4	19.005856	72.835206	0	No
1	1	1	1	1	4	19.006121	72.857993	1	No
2	1	1	1	1	4	19.006154	72.860842	2	No
3	1	1	1	1	4	19.003445	72.860876	3	No
4	1	1	1	1	4	19.000177	72.812489	4	No

Initial data

	loc
0	(1210.200545299848, 2.375829022467883)
1	(1378.4562546969855, 0.2625744484497545)
2	(1399.4883845044892, 0.0)
3	(1399.7442291706961, 21.571231371245062)
4	(1042.470849582988, 47.5928528332264)

Data after normalisation

The data for location was normalised

```
[2] array12_16 = np.loadtxt(open("12_16.csv", "r"), delimiter=",", skiprows=1)  #loading the data frame 12-6.

    array12_16[:4, :4]

    array([[1.009e+03, 1.000e+00, 5.200e+01, 7.500e+01],
           [9.270e+02, 2.600e+01, 1.000e+00, 2.000e+01],
           [9.920e+02, 1.790e+02, 1.230e+02, 1.730e+02],
           [9.870e+02, 6.500e+01, 2.900e+01, 1.000e+00]])

[4] array12_16_norm = (array12_16 - array12_16.min())/(array12_16.max() - array12_16.min()) #code used for normalisation of the array row wise 12-6.

[5] array12_16_norm[:4,:4] #peeking into the normalised array.

    array([[0.36896047, 0.        , 0.01866764, 0.02708638],
           [0.33894583, 0.00915081, 0.        , 0.00695461],
           [0.36273792, 0.06515373, 0.04465593, 0.06295754],
           [0.36090776, 0.02342606, 0.0102489 , 0.        ]])

[6] array8_12 = np.loadtxt(open("8_12.csv", "r"), delimiter=",", skiprows=1) #loading the data frame 8-12.

[7] array8_12_norm = (array8_12 - array8_12.min())/(array8_12.max() - array8_12.min()) #code used for normalisation of the array row wise time 8-12.

[8] array8_12_norm[:4,:4] #peeking into the normalised array.

    array([[0.37354651, 0.        , 0.02579942, 0.03415698],
           [0.34375   , 0.01635174, 0.        , 0.01417151],
           [0.36736919, 0.07194767, 0.05159884, 0.06976744],
           [0.36555233, 0.03052326, 0.01744186, 0.        ]])
```

The data for time normalised between 0 and 1

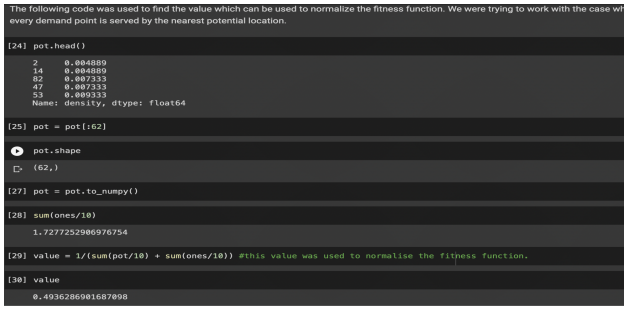
```
[9] array4_8 = np.loadtxt(open("4_8.csv", "r"), delimiter=",", skiprows=1) #loading the data frame 4-8.

    array4_8_norm = (array4_8 - array4_8.min())/(array4_8.max() - array4_8.min()) #code used for normalisation of the array row wise time 4-8.

[11] array4_8_norm[:4,:4] #peeking into the normalised array.

     array([[0.32179386, 0.        , 0.03186467, 0.04091267],
            [0.2895358 , 0.02163651, 0.        , 0.01927616],
            [0.31510622, 0.08182533, 0.05979544, 0.07946499],
            [0.31313926, 0.03697876, 0.02281668, 0.        ]])
```

Both the normalisations were done so that the data goes well with the genetic algorithm as the density used in the objective function lies between 0 and 1.



In this code we tried to find the value that could be used to normalize the fitness function to get a solution for the case when every demand point is served by the nearest location.

3.Algorithm:

3.1.*Objective Function:*

$$\sum_i d_i x_i + \sum_j \sum_i y_{ji} t_{ji} \quad \text{... (1)}$$

$$d_i \in [0,1]$$

, where d denotes density/compactness of location i per hub at i

$$x_i = \begin{cases} 1, & hub\ present\ at\ i \\ 0, & hub\ absent\ at\ i \end{cases}$$

$$y_{ji} = \begin{cases} 1, & location\ at\ j\ is\ served\ by\ hub\ at\ i \\ 0, & otherwise \end{cases}$$

t_{ji} = time taken by hub at i to some location at j

The first term signifies the property of an establishment cost of a hub at that point based on the area's compactness; the second term signifies the cost in terms of travel time for each establishment. The Fitness function is the inverse of the objective

Constraints:

$$\sum_i y_{ij} = 1, \qquad \forall\ j$$

3.2. *Genetic Algorithm:*

Genetic algorithms (GAs) can be defined as meta-heuristics based on the evolutionary process of natural systems[11]. They have been applied to a lot of optimization problems giving us a great result. GAs are a new approach to solving complex problems such as determination of facility layout. GAs became known through the work of John Holland in the 1960s[11]. The GAs contain the elements of the methods of blind searching for the solution and of directed and stochastic searching and thus give compromise between the utilization and searching for solution. At the beginning, they search in the entire search space and afterwards, by means of crossover, they search only in the surrounding of the promising solutions. So GAs employed random, yet directed search for locating the globally optimal solution [12]. The starting point in GA presented in this work was an initial population of solutions (which was randomly generated). The facility layout and its randomly generated chromosome are shown on figure1. This population undergoes a number of transformations designed to improve the solutions provided. Such transformations are made in the main loop of the algorithm, and have three basic stages: selection, crossover, and replacement, as discussed below.

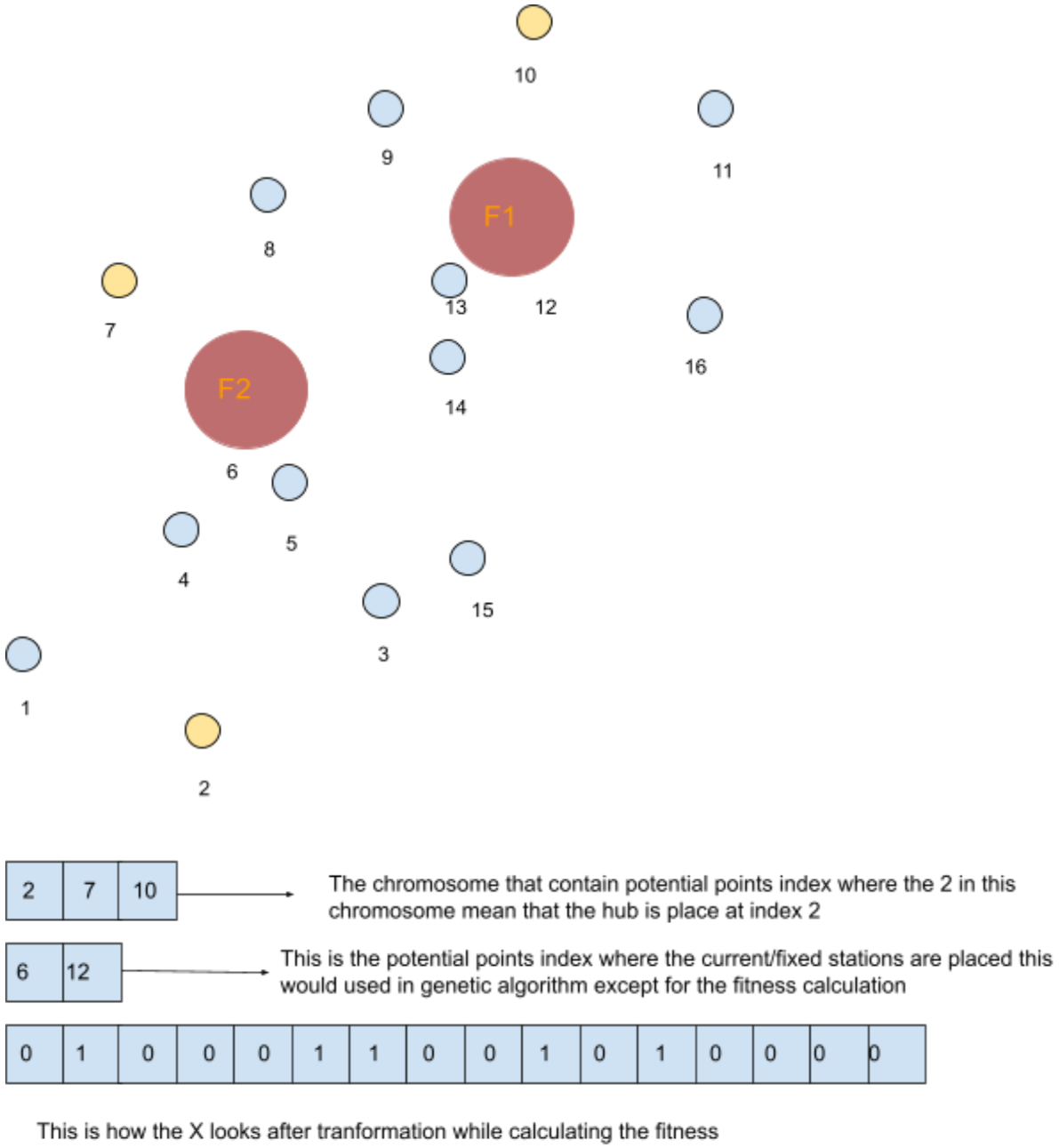


Fig. 1. Type of layout used in calculations and its chromosome representation (actual data is larger)

Each of the selection-transformation cycles that the population undergoes constitutes a generation. Hopefully, after a certain number of generations, the population will have evolved towards the optimum solution to the problem, or at least to a near-best solution[1].

The selection stage consists of sampling the initial population, thereby obtaining a new

population with the same number of individuals as the initial one. This stage aims at improving the quality of the population by favoring those individuals that are more adequate for a particular problem (the quality of an individual is gauged by calculating its fitness, using equation 1, which indicates how good a solution is)[1].

The selection, mutation, and crossover functions were used to create the new generation of solutions. A fitness function evaluates the likelihood of a design to be in the next generation. Each chromosome is put into a mating several times based on the fitness making it more likely to be selected.

Mutation is the process of randomly changing the genes in the chromosome and it prevents GAs from stagnating during the solution process. Crossover is responsible for introducing most new solutions by selecting two parent strings at random from the mating pool and exchanging parts of the strings.

A GA cycle used in this work operates as follows:

1. Generate an initial population consisting of 200-1000 members using a random number generator. The initial population's chromosome size varies from 0, 40 i.e. the total number of hubs.

2. Calculate the fitness f (Eq. (1)) of all population members. And append it to a new list with the matching index.

3. Place each member in a mating pool, more number of times if the fitness value is higher.

4. Randomly select two parents from the mating pool. So the chromosome with the highest fitness value has a high probability to be selected.

5. Now the two parent chromosomes are crossed over with a midpoint randomly chosen.

6. Now for each member in the population each gene is mutated with a low probability. And replaces the previous gen's population.

The starting chromosome in the new iteration isn't randomly generated. It is the chromosome obtained by crossover of two parent chromosomes discussed above. Consider a pair of parent chromosomes (P1, P2) shown below:

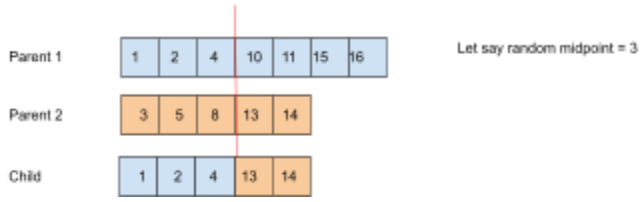


Fig 2. The crossover

The way of crossover implementing in this work was choosing a midpoint randomly based on the parents size and take the first half from the first parent and the second half from the second parent i.e. for random midpoint of 3, from P1(1, 2, 4) is taken an from P2 (13, 14) is taken and the child formed is (1, 2, 4, 13, 14). Which is shown in the above figure. The mutation function is used to replace each gene with a certain low probability .

3.3. *The Codes:*

The following codes were ran using pycharm in a notebook with an ARMx64 architecture Apple M1 with 8 core CPU with 8GB of ram.

1.Population Initialisation:

```
    def gen_population(self):
        a = len(self.potential)
        for _ in range(a, self.pop_size):
            self.population.append(ran_gen(self.fixed))
```

The gen_population method generates loops for the pop_size mentioned and triggers a function called “ran_gen” which randomly generates chromosomes of random size which do not contain any indexes from fixed points.

```
def ran_gen(fixed):
    lst = [x for x in range(0, 432) if x not in fixed]
    lst = random.sample(lst, random.randint(0, 37))
    return np.array(lst)
```

2.Fitness Evaluation:

```
def eval_fitness(self):
    self.i += 1
    self.fitness = []
    t = self.travel_time/10
    d = self.densities/10
    for j, x2 in enumerate(self.population):
        f = self.calc_fitness(x2, d, t)
        self.fitness.append(float(f))
    self.fitness = normalize(self.fitness)
```

The eval_fitness method loops over every population chromosome, then transforms the chromosome to a list of 0’s and 1’s where 1 means that there is a facility at the location in that facility and calculates their fitness.

```
    def calc_fitness(self, x2, d, t):
        x = self.gen_full(x2)
        x = [1 if d in x else 0 for d in range(0, 432)]
        x = np.array(x).reshape(len(x), )
        f = 1 / (sum(x * d) + sum(min_mat(x, t)))
        return f
```

The calc_fitness calculates the value for the fitness. Here the min_mat function finds the y matrix i.e where a particular demand location is served by the hub where the rows are demands which adds to 1 which is as the constraint mentioned in eq (2).

```
def min_mat(a, b):
    return np.where(a*b > 0, a*b, np.inf).min(1)
```

Here after x(i.e a) is multiplied with t(i.e b) by the numpy broadcasting each row in t has dot product with x(i.e. Without summing them up), then the minimum of each row is taken and turned into a numpy array, which is what the function return later it is summed in the fitness function which basically the summing of all elements in y*t matrix. And this fitness value is added to the respective index in the fitness list.

3.Selection

```
    def nat_sel(self):
        self.mat_pool = []
        for i, x in enumerate(self.population):
            f = self.fitness[i]
            n = round(f * 100)
            for j in range(n):
                self.mat_pool.append(x)
```

In nat_sel method mat_pool is emptied and looped over the entire population and adds the chromosomes to the mat_pool for the respective fitness times 100.

4.Crossover And Mutation:

```
def new_pop(self):
    """
    Creates new list of population
    :return:
    """
    for i, x in enumerate(self.population):
        p1 = random.choice(self.mat_pool)
        p2 = random.choice(self.mat_pool)
        c1 = self.crossover(p1, p2)
        self.population[i] = c1
    self.eval_fitness()
```

In the new_pop method we loop over the population and two parents are randomly chosen and are crossed over as we discussed above.

```
def crossover(self, p1, p2):
    p1 = p1.tolist()
    p2 = p2.tolist()
    mid_point = min(random.randint(0, len(p1)), random.randint(0, len(p2)))
    child = p1[: mid_point] + p2[mid_point:]
    child = self.mutate(child)
    return np.array(child)
```

Then we call over the mutate method which loops over the child's gene and each gene is changed to a different value with a low probability mut_rate.

```
def mutate(self, chrom):
    lst = [x for x in range(0, 432) if x not in chrom and x not in self.fixe
    for i, gene in enumerate(chrom):
        if random.random() < self.mutation_rate:
            chrom[i] = random.choice(lst)

    return chrom
```

The resultant child is returned as a numpy array which then replaces a chromosome in the population. And the whole process is repeated.

4. Results:

The Genetic algorithm ran as expected the average fitness value of the population increased over loops. It was done with a max facility size of 50 including 22 fixed stations and a population size of 200. The finding for the 8-12 dataset is provided below. It is over a loop of 1000.

Here we can see that we started with around 0.22 for average fit with reaching to 0.34 in the 128th loop and 0.34 on the 400th loop subsequently reaching to 0.36 in the 1000th loop.

Same trend can be seen with best fitness and average time as well. But for max time, i.e. the point where the facilities are farest from, seems to increase this could be mainly due to the density factor that is included in the objective function. And the other reason could be that the algorithm tries to find ones with some location having very small time values ignoring the other ones.

We were able to run the file on all the time period data sets of 4-8, 8-12 and 12-16, and the findings for each of them is discussed below.

Looking at the final findings(Fig. 4) for the different time stamps we observe that fitness value for data set on time 4_8 seems to have very high compared to other ones this is mainly due

to reason that, during this time the road traffics are very less so most places can be reached in less time.

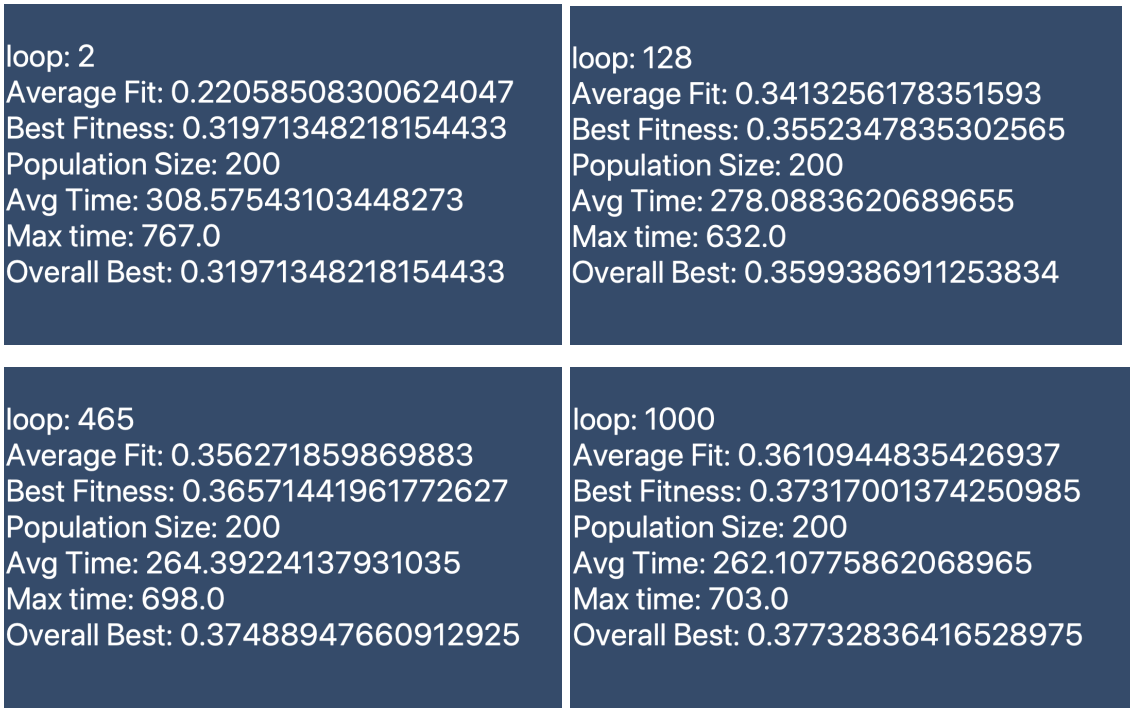
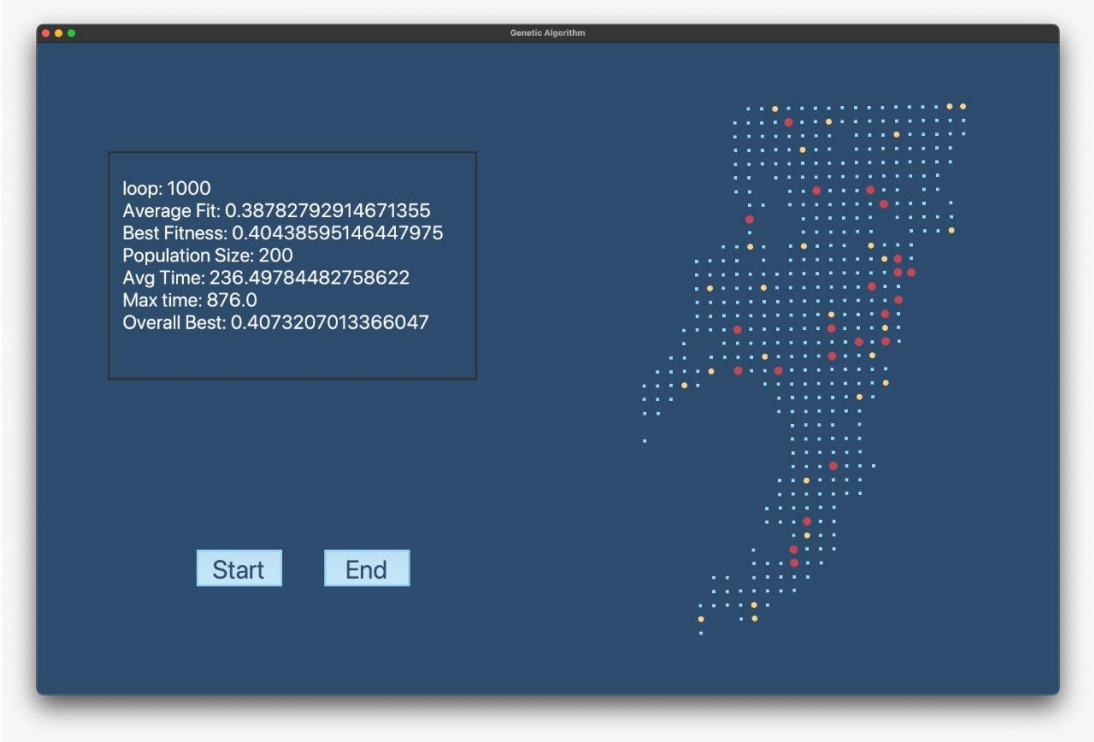


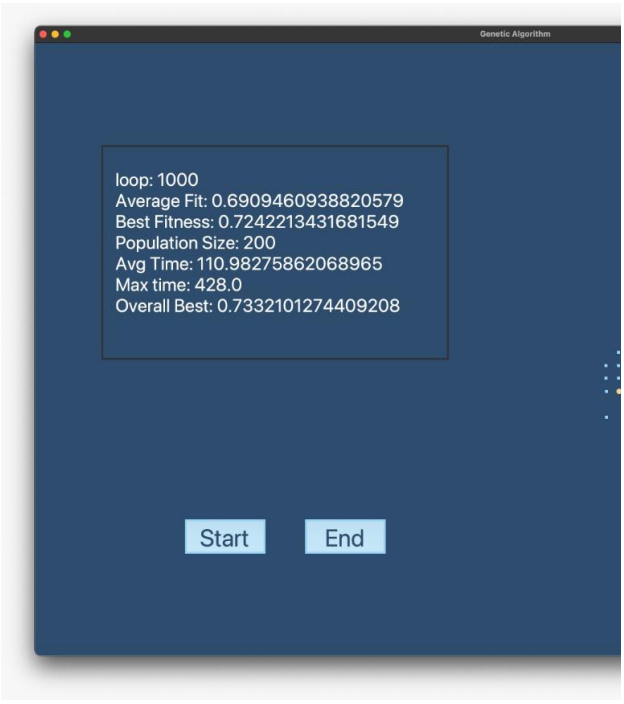
Fig 3. Findings over course of 1000 loop



Distribution for the time 12pm to 16 pm.

low value of average time and high value of best fitness is due to the fact that at this time the traffic is low and so the algorithm can more effectively carry out the allocation while keeping the average time low.

Next on the right we have distribution for the time period 8 am to 12 pm.



Distribution for the time period 4am to 8am. The

From the three images it is clear that our algorithm can deliver as per the changing needs of the situation (here in this case time period) and hence is fit to cater the needs of dynamic allocation.

5. Contribution:

- The dynamic allocation approach is an extension of the static one as it considers the changes over multiple periods and the cost of rearranging the layout as per the requirements. Also genetic algorithm can cater to the needs of dynamic approach without taking much help from dynamic programming.
- We have provided a dashboard for easy visualization of the algorithm. The dashboard allows the user to control the loop and get a visualization of the distribution as per their requirement.
- Our algorithm is better adapted to serve the needs of dynamic allocation.

6. Conclusion:

- We have been able to find out potential locations, assign dynamic hubs for each and were able to visualise which fixed location will serve them.
- The allocation is such that the hub may serve multiple demand points but each demand point is served only by a single hub.
- Allotting the number of vehicles that a hub can get from a particular station near that hub such that it could satisfy the demand of each point.
- Complexity of each genetic cycle is very high. We could try to improve it to improve the speed of the program.
- Can find better value to normalize the fitness. Because it would give us better results.
- We could also try to build it under additional constraints of vehicle availability.

7. Acknowledgements

We would like to thank our instructor for this course Dr Vaibhav Kumar, Assistant Professor, Data Science and Engineering, IISER- Bhopal for providing us with the sufficient data and sources required to complete this project. Also, for providing us with the necessary support and mentorship whenever required.

8. References:

1. Holland H.J., "Adaptation in Natural and Artificial Systems", 1975 (University of Michigan Press: Ann Arbor).
2. Heng L., Love P.E.D, "Genetic Search for solving Construction Site-Level Unequal-Area Facility Layout Problems", Automation in Construction, 2000, 9, 217-226.
3. I Mihajlovic , Z Zivkovic, N Strbac, D Zivkovic, A Jovanovic, " Using Genetic Algorithms To Resolve Facility Layout Problem", Serbian Journal of Management 2 (1) (2007) 35- 46
4. Chawis Boonmee, Mikiharu Arimura, Takumi Asada, " Facility location optimization model for emergency humanitarian logistics", International Journal of Disaster Risk Reduction
5. K .F. Man, K.S. Tang, S. Kwong, "Genetic Algorithm: concepts and applications", IEEE Transactions on Industrial Engineering, (Oct 1996).
6. M.T. Hajiaghayi, M. Mahdian, V.S. Mirrokni, "The facility location problem with general cost function", Wiley International Journal (2003)