

# A New Approach to Solve Travelling Salesman Problem



A Thesis submitted to the

Dept. of Computer Science and Engineering, BRAC University in partial fulfilment of the requirements for the Bachelor of Science degree in Computer Engineering

By

Sanjana Sarker

10201013

Supervised by

**Ms. Farzana Rashid**

Co-supervised by

**Ms. Dilruba Showkat**

December 28th

BRAC University, Dhaka

## **ACKNOWLEDGEMENT**

I have incurred many debts of gratitude over the last few days while preparing for this report. First and foremost, I would like to pay my gratitude to the Almighty Allah for giving me the ability to work hard. This is my humble attempt to present gratitude in writing this report. Secondly, I would like to express my gratitude to my Supervisor Ms.Farzana Rashid and my co-supervisor Ms.Dilruba Showkat for their valuable guidance and support. I really appreciate what they have taught me for this thesis; I express my heart-felt gratefulness for that.

Hence, I have taken help from different sources for preparing the report. Without the help, completing the report perfectly would not have been possible. Now here is a petite effort to show my deep gratitude to those helpful persons who have put it up on the texts and websites.

Last but not the least, I would like to thank my fellow friends and others whose names are not mentioned here, but directly or indirectly offered suggestions and guidelines to the completion of my work. I warmly thank them for their kind contribution.

# DECLARATION

## Statement

I hereby proclaim that this thesis is based on the results I found by my hard work. Contents of the work found by other researcher(s) are motioned by references. This thesis has never been previously submitted for any degree neither in whole nor in part.

Signature of Acting Chairperson And Associate Professor:

---

Dr. Md. Khalilur Rhaman

Signature of Supervisor:

---

Ms. Farzana Rashid

Signature of Co-supervisor :

---

Ms. Dilruba Showkat

Signature of Author:

---

Sanjana Sarker

[ Student ID : 10201013 ]

## ***Abstract***

The travelling Salesman Problem is one of the most NP-hard problems. Our research provides a yieldable method for solving the problem using genetic algorithm. To solve TSP we use genetic algorithm, a search algorithm which generates random tours and using crossover technique it gives almost optimized solution for for these kinds of problems. We are introducing a map reduction technique with Genetic Algorithm to create a new approach to solve TSP.

# Table of Content

<b>Content</b>	<b>Page No.</b>
<b>1. Introduction</b>	<b>1</b>
1.1 Objective	1
<b>2. Theory</b>	<b>2</b>
- 2.1 Travelling Salesman Problem	2
- 2.2 Objective and Field of Application	2
- 2.3 Complexity and Methods of solving TSP	4
- 2.4 Genetic Algorithm	5
- 2.4.1 Definition	5
- 2.4.2 Overview of GA	6
- 2.5 Map Reduction	12
<b>3. Solving TSP with GA and Map Reduce:</b>	<b>13</b>
- 3.1 Previous work	13
- 3.2 Our Approach	15
<b>4. New Crossover</b>	<b>17</b>
- 4.1 Theory of Crossover	17
- 4.2 JAVA code for New Crossover	18

<b>5. Methodology</b>	<b>24</b>
- 5.1 Selecting real world problem	25
- 5.1.a Converting Geological Coordinates	26
- 5.2 Selecting Dividing axes from Map Reduction	27
- 5.3 Applying Genetic Algorithm to Each Coordinate	29
- 5.4 Combining the routes from four coordinate	29
<b>6. Findings</b>	<b>55</b>
- 6.1 Result using Crossover	55
- 6.2 Results using Map Reduce, GA and Permutation	57
<b>7. Conclusion and Continuation</b>	<b>59</b>
- 7.1 Future Works	59
- 7.2 Conclusion	59
<b>8. References</b>	<b>60</b>

# Table of Figures

<b>Figure Number</b>	<b>Page No.</b>
Figure 1: Population & result of crossover in GA	8
Figure 2: Selection from population	9
Figure 3: Crossover in Genetic Algorithm	10
Figure 4: Mutation in Genetic Algorithm	11
Figure 5: Map reduction	12
Figure 6: Approach of TSP by Map Reduce	14
Figure 7: Map reduce approach	15
Figure 8: New crossover	18
Figure 9: Methodology Flow Chart	24

Figure 10: Map reduction axes of Burma14	26
Figure 11: Screenshot of Solution using new crossover	55
Figure 12: Graph of Solution using new crossover	56
Figure 13Graph of optimal solution	57
Figure 14: Screenshot of solution using Map Reduce, GA, Combination	58
Figure 15: Graph of solution using Map Reduce, GA & combination	58



# List of Tables

<b>Table Number</b>	<b>Page No.</b>
Table 1: Table 1: Geological co-coordinate of Burma14	25

# 1. Introduction

## *1.1 Objective*

Day by day, our life is getting complex. And to make our life easier we lean on to various electronic devices. But to make them work as we want, we need to address many mathematical problems. Travelling salesman problem is a very old mathematical problem. It models a scenario where a salesman has many cities to visit in shortest possible time. Given the distance among the cities, he must calculate the shortest route.

Researchers have been working with Travelling Salesman problem for over centuries. Many models have been introduced to solve this legendary mathematical problem. In this paper I tried to introduce a new approach to solve the Travelling Salesman Problem.

I combined Genetic Algorithm Along with Map Reduction technique to get a new tactic and experimented to see whether the result is optimized.

## **2. Theory**

### ***2.1 Travelling Salesman Problem:***

The problem is very simple. Provided a list of cities and distance between them, it asks to find the shortest possible path so that, starting from any city, one can visit all the cities once and return to the starting city.

The origin of Travelling Salesman Problem (TSP) dates back to 1759 when the first instance of the travelling salesman problem was from Euler whose problem was to move a knight to every position on a chess board exactly

once.<sup>[1]</sup> Then in 1832 mathematician W.R Hamilton and Thomas Kirkman formulated it. The travelling salesman first gained fame in a book written by German salesman

BF Voigt in 1832 on how to be a successful travelling salesman.<sup>[1]</sup> Though he did not mention TSP by name but suggested that to cover as many locations as possible not visiting any location twice is the key factor of scheduling of a tour. The standard or symmetric travelling salesman problem can be stated mathematically as follows:

Given a weighted graph  $G = (V, E)$ ; where the weight  $c_{ij}$  on the edge between nodes  $i$  and  $j$  is a non-negative value, find the tour of all nodes that has the minimum total cost.

### ***2.2 Objective and Field of Application***

The TSP has numerous applications, such as scheduling or planning, logistics, and the manufacture of microchips. A touch adjusted, it appears as a sub-problem in many fields,

such as DNA sequencing. In these applications, the concept of city represents, for instance, customers, soldering points, or DNA fragments, and the concept distance represents travelling times or cost, or a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows may be imposed.

The vehicle routing problem can be demonstrated as a traveling salesman problem. Here the problem is to find which customers should be attended by which vehicles and the minimum number of vehicles needed to serve each customer. There are different variations of this problem including finding the minimum time to serve all customers. We can address some of

these problems as the TSP.

An application found by Plate, Lowe and Chandrasekaran is overhauling gas turbine engines in aircraft. Nozzle-guide fin assemblies, consisting of nozzle guide fins fixed to the circumference, are located at each turbine stage to ensure uniform gas flow. The placement of the fins in order to minimize fuel consumption can be modeled as a symmetric TSP.<sup>[2]</sup>

The scheduling of jobs on a single machine given the time it takes for each job and the time it takes to prepare the machine for each job is also TSP. We try to minimize the total time to process each job.

### ***2.3 Complexity and Methods of solving TSP:***

At present the only known method guaranteed to optimally solve the Travelling Salesman Problem of any size, is by computing each possible tour and searching for the tour with least cost. Each possible tour is a combination of  $123 \dots n$ , where  $n$  is the number of cities, so therefore the number of tours is  $n!$ . When  $n$  gets large, it becomes impossible to find the cost of every tour in polynomial time.

Obviously we need to find an algorithm that will give us a solution in a shorter amount of time. The travelling salesman problem is NP-hard so there is no known algorithm that will solve it in polynomial time. We will probably have to sacrifice optimality in order to get a good answer in a shorter time.

Many different methods of optimization have been used to try to solve the TSP. Among them

Greedy algorithm, Nearest neighbour algorithm, minimum spanning tree is mentionable.

The Genetic Algorithm (GA) however is preferable to many researchers for its reputation to solve these kinds of problem close to optimally.

## ***2.4 Genetic Algorithm:***

### ***2.4.1 Definition :***

Genetic Algorithms were invented to imitate some of the courses observed in natural evolution. Many people, biologists included, are astonished that life at the level of complexity that we observe could have evolved in the relatively short time suggested by the fossil record. The idea with GA is to use this power of evolution to solve optimization problems. The father of the original Genetic Algorithm was John Holland who invented it in the early 1970's.<sup>[3]</sup>

Genetic Algorithms (GAs) are adaptive heuristic search algorithm based on the evolutionary ideas of natural selection and genetics. As such they represent an intelligent exploitation of a random search used to solve optimization problems. Although randomized, GAs are by no means random, instead they exploit historical information to direct the search into the region of better performance within the search space. The basic techniques of the GAs are designed to simulate processes in natural systems necessary for evolution, specially those follow the principles first laid down by Charles Darwin of "survival of the fittest.". Since in nature, competition among individuals for scanty resources results in the fittest individuals dominating over the weaker ones.<sup>[4]</sup>

### ***2.4.2 Overview of GA:***

GA simulates the survival of the fittest among individuals over consecutive generation for solving a problem. Each generation consists of a population of character strings that are

analogous to the chromosome that we see in our DNA. Each individual represents a point in a search space and a possible solution. The individuals in the population are then made to go through a process of evolution. GAs are based on an analogy with the genetic structure and behaviour of chromosomes within a population of individuals using the following foundations:

- i. Individuals in a population compete for resources and mates.
- ii. Those individuals most successful in each 'competition' will produce more offspring than those individuals that perform poorly.
- iii. Genes from 'good' individuals propagate throughout the population so that two good parents will sometimes produce offspring that are better than either parent.
- iv. Thus each successive generation will become more suited to their environment.

There are three 5 main aspects of GA. They are namely

- i. Population
- ii. Fitness calculation
- iii. Selection
- iv. Crossover
- v. Mutation

### ***i.Population:***

A population of individuals is maintained within search space for a GA, each representing a possible solution to a given problem. Each individual is coded as a finite length vector of components, or variables, in terms of some alphabet, usually the binary alphabet {0,1}. To continue the genetic analogy these individuals are likened to chromosomes and the variables are analogous to genes. Thus a chromosome (solution) is composed of several genes (variables). A fitness score is assigned to each solution representing the abilities of an individual to 'compete'. The individual with the optimal (or generally near optimal) fitness score is sought. The GA aims to use selective 'breeding' of the solutions to produce 'offspring' better than the parents by combining information from the chromosomes. In the figure below the parent1 and parent 2 are the population.



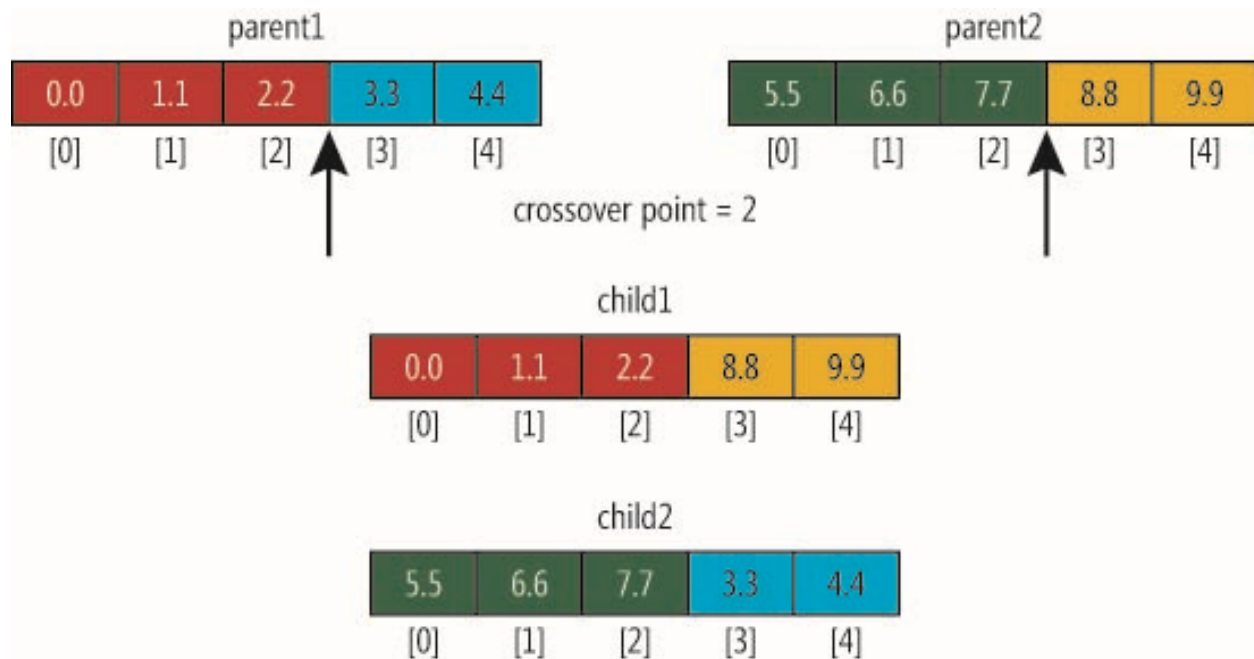


Fig 1 : Population & result of crossover in GA

### ***ii. Fitness Calculation***

Fitness calculation is a way to determine which candidates of the population serves the purpose best. For instance. In case of TSP, the way to calculate fitness is to determine which route in the population cost least. For different problems fitness calculation method changes accordingly.

### ***iii. Selection:***

There can be more than two individuals in population. Then certain number of candidate must be chosen for the crossover. This is called the selection. The selection process is based on the fitness calculation. It gives preference to better individuals, allowing them to pass on their genes to the next generation. The goodness of each individual depends on its fitness.

Fitness may be determined by an objective function or by a subjective judgment.

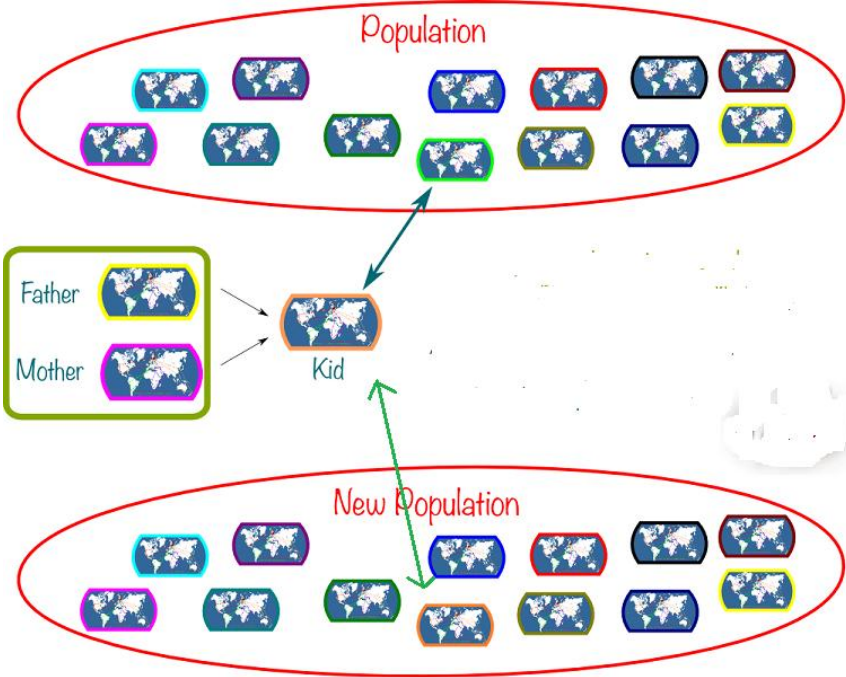


Fig 2 : Selection from population

*iv. Crossover:*

Crossover refers to merging of two or more parent individuals to make new offspring. In crossover a portion of uniform size is selected from both the parents. Then the portion from first parent is added to the portion of second, thus a offspring is born. Similarly second child is produced. Prime distinguished factor of GA from other optimization techniques. Two individuals are chosen from the population using the selection operator. A crossover site along the bit strings is randomly chosen. The values of the two strings are exchanged up to this point

If  $S_1=000000$  and  $s_2=111111$  and the crossover point is 2 then  $S_1'=110000$  and  $s_2'=001111$

the two new offspring created from this mating are put into the next generation of the population. By recombining portions of good individuals, this process is likely to create even better individuals

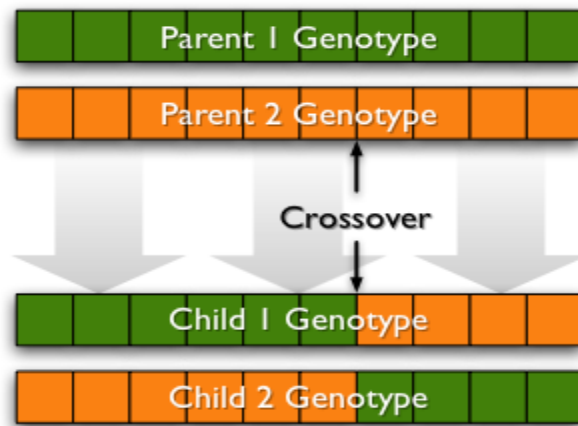


Fig3 : Crossover in Genetic Algorithm

***v . Mutation:***

After Crossover has run several times and yet expected result is not found, then mutation comes into action. Mutation refers to changing each candidate parent at certain point or points. With some low probability, a portion of the new individuals will have some of their bits flipped. Its purpose is to maintain diversity within the population and inhibit premature convergence. Mutation alone induces a random walk through the search space

Mutation and selection (without crossover) create a parallel, noise-tolerant, hill-climbing algorithms .

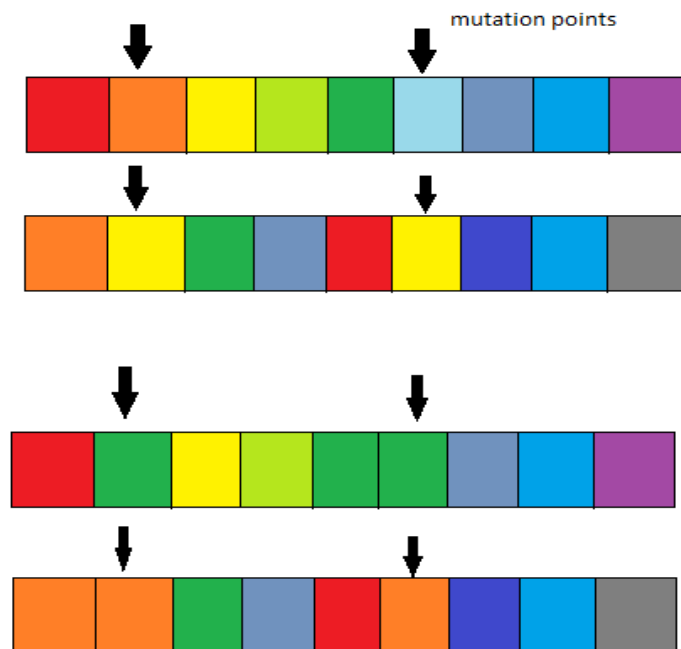


Fig 4 : Mutation in Genetic Algorithm

## 2.5 Map Reduction:

Map Reduce is programming model which has proven to be very effective to run a *query* on big data.

Generally speaking, it works like this:

- The data is **partitioned** across multiple computer nodes.
- A **map** function runs on every partition and returns a result.
- A **reduce** function reduces 2 results into one result. Its continuously run until only a single result remains.<sup>[5]</sup>

The figure bellow shows map reduction on TSP. Given the cities, we can scenario into four possible co-ordinates and solve them separately.

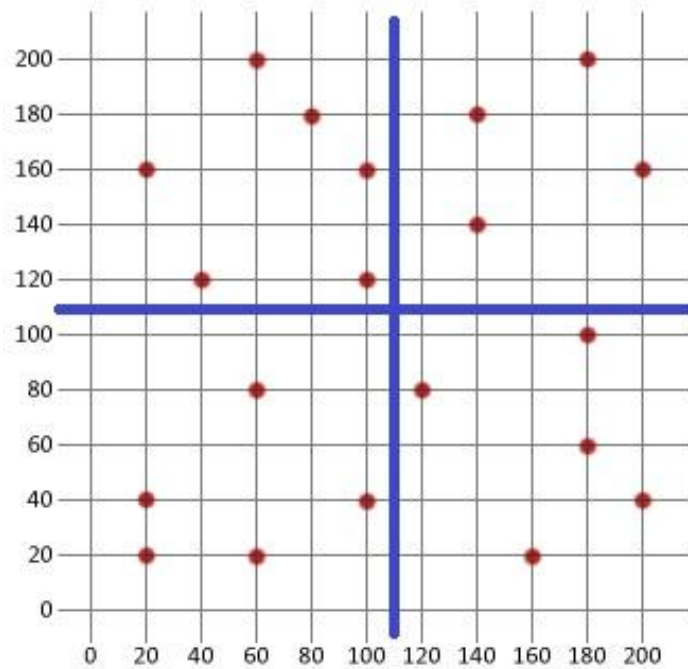


Fig 5: Map reduction

### **3. Solving TSP with GA and Map Reduce:**

#### ***3.1 Previous work:***

Travelling Salesman problem being a complex and famous one, researchers have tried to achieve optimality using numerous methods and algorithms. We have come across many works that has solved TSP with GA. Also new crossover technique has been introduced over time to make result optimal and efficient .

As, TSP is a NP- hard problem, researchers often tried to reduce in complexity by dividing it into pieces of several problems, however, optimality was beyond reach. It is established that Map Reduce cannot solve any planning problem optimally.

Previously researchers at Optaplanner organization have showed that Map Reduce can not be a reasonable approach to TSP. What their approach was-

- Divide the Map of cities into four co-ordinates
- Solve each co-ordinate separately to get the shortest path at each
- Merge the shortest path

# MapReduce on TSP

How does MapReduce and Divide&Conquer behave on NP-hard problems?

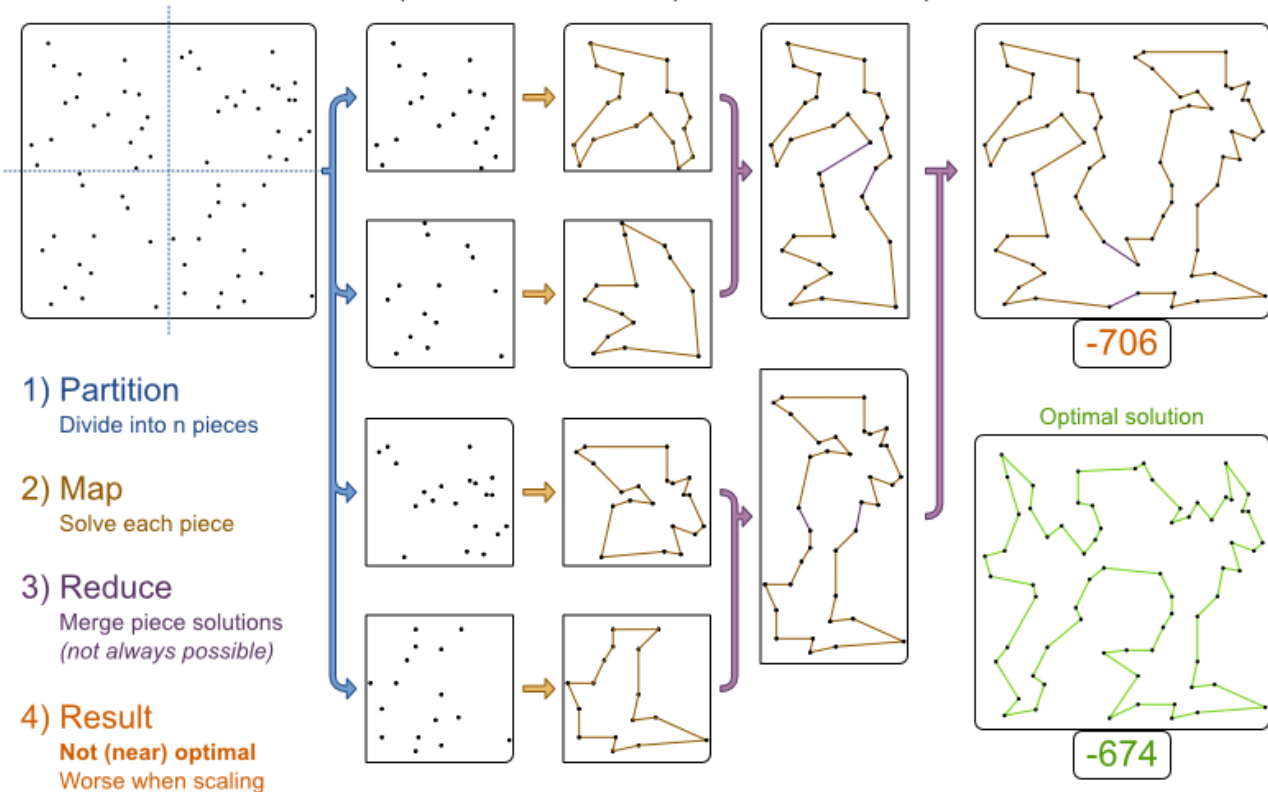


Fig 6: Approach of TSP by Map Reduce

However, the result showed even if the shortest path was obtained at each co-ordinate, and they were merged optimally, the solution was not optimal.

Hence, I came forward to examine if the result shifts to optimality if we combine Genetic Algorithm, Map-Reduce and Combination.

### 3.2 Our Approach:

I noticed, choosing the shortest path in each co-ordinate does not help obtaining optimal total path when using map reduce technique. The reason can be elaborated with an example.

if there were 2 shortest path of 24 and 17 and the cost to merge them is 12, the total cost is  $24+17+12 = 53$ . Fig 7 left

However, there might be two not- optimal path of cost 27 and 20 and their merging cost 5 so in totals cost stands  $27+20+5=52$  fig 7 right

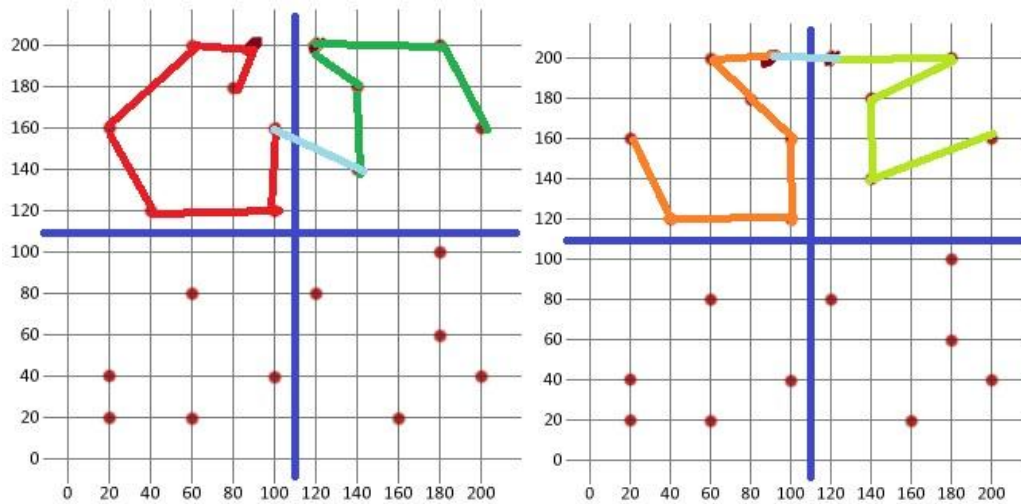


Fig 7: Map reduce approach

First of all, I designed a new crossover technique to increase the efficiency of crossover operator of GA for TSP. Then I divided the map of given city into four equal co-ordinates such that no city rest on the axes. Then GA was run for each four co-ordinates and not only the best



path, but also second or third best paths were recorded. Then we used simple combination to determine the 4 paths of four co-ordinates and merging in between them.

## **4.New Crossover:**

In Travelling Salesman Problem we need To visit each city only once. In general crossover there are chances of repetition of cities in a tour. So I modifies the Crossover and introduced a new type of crossover for travelling salesman problem.

### ***4.1 Theory of Crossover:***

In this Crossover I calculated the average cost of the each parent tour and selected the minimum between them. I did this because the parent tour in which the average distance cost is less, probability is more that, the distance between the cities in that tour is going to be less.

Then, I compared the distance between two cities and if that is less than the average cost, then I considered them for child tour.

If distance between two cities is greater then the average cost, then they were not selected for the tour.

Then we moved through the second parent and checked which city does not exist in the child tour, if fount any, they were added to the child tour.

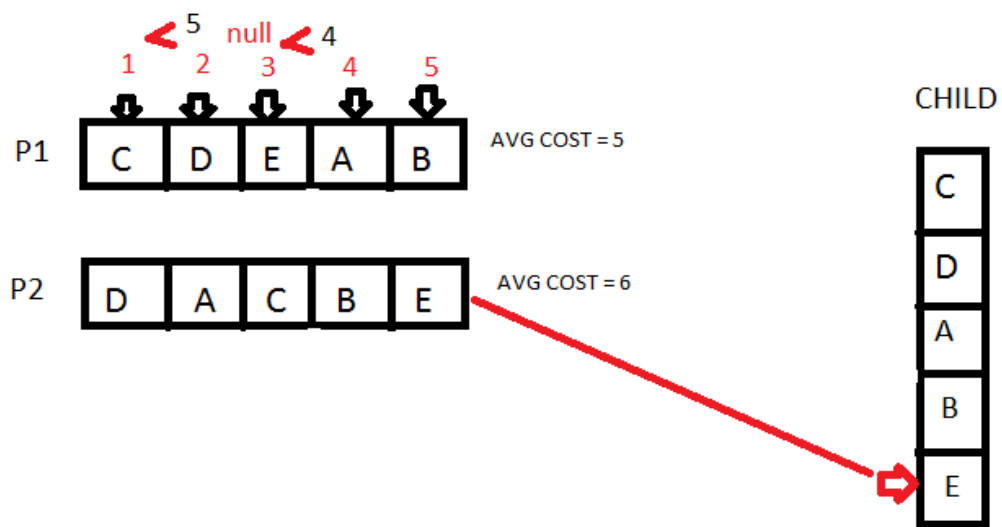


Fig 8: New crossover

#### 4.2 JAVA code for New Crossover:

```
private static City[] crossover(City[] p1, City[] p2) {

    List<City> child = new ArrayList<City>();

    double totalCost = 0;
```

```
for(int i=0; i<p1.length; i++) {

    City u = p1[i]; //getCity(p1[i]);

    City v;

    if(i+1 < p1.length) {

        v = p1[i+1]; //getCity(p1[i+1]);

    } else {

        v = p1[0]; //

    }

    totalCost += u.distanceTo(v);

}

double avgCost = totalCost / p1.length;

totalCost = 0;
```

```
for(int i=0; i<p2.length; i++) {  
  
    City u = p2[i]; //getCity(p2[i]);  
  
    City v;  
  
    if(i+1 < p2.length) {  
  
        v = p2[i+1]; //getCity(p2[i+1]);  
  
    } else {  
  
        v = p2[0]; // getCity(p2[0]);  
  
    }  
  
    totalCost += u.distanceTo(v);  
  
}
```

```
avgCost = Math.min(totalCost/p2.length, avgCost);
```

```
List<City> maxLengthSubString = new ArrayList<City>();
```

```
List<City> tmpSubString = new ArrayList<City>();

for(int i=1; i<p1.length; i++) {

    City u = p1[i-1]; //getCity(p1[i-1]);

    City v = p1[i]; //getCity(p1[i]);

    double cost = u.distanceTo(v);

    if(cost < avgCost) {

        if(tmpSubString.size() == 0)

            tmpSubString.add(p1[i-1]);

            tmpSubString.add(p1[i]);

        } else {

//            child.addAll(tmpSubString);

            if(maxLengthSubString.size() < tmpSubString.size()) {
```

```
        maxLengthSubString = tmpSubString;

        tmpSubString = new ArrayList<City>();

    }

    tmpSubString.clear();

}

}

child.addAll(maxLengthSubString);

for(int i=0; i<p2.length; i++) {

    if(!child.contains(p2[i])) {

        child.add(p2[i]);

    }

}
```

```
City[] childArray = new City[child.size()];
```

```
for(int i=0; i<child.size(); i++)
```

```
    childArray[i] = child.get(i);
```

```
return childArray;
```

```
}
```



## 5.Methodology

This paper is divided into two parts. First I developed a new crossover technique. And in second part I used this crossover, map reduce and combination to address TSP.

Our methodology of Map Reduce comprises of –

- Selecting real world data
- Selecting Dividing axes for map reduce
- Applying genetic Algorithm to each divided co-ordinates
- Storing best 3 paths in each coordinates
- Using combination to get the least cost route

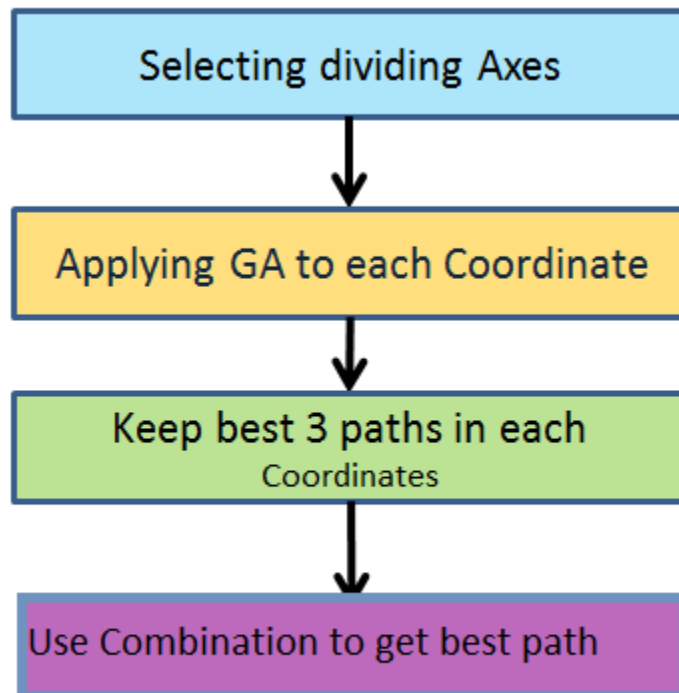


Fig 9 : Methodology Flow Chart

### ***5.1 Selecting real world problems:***

For my paper, we chose real world data for cities. I used Data called Burma14 Which consists of geological co-ordinates of 14 cities of current Myanmar. Co-ordinates are provided Below, to make identification easy we denoted the city by Alphabets as the original names were unavailable.

City	Latitude	Longitude
A	16.47	96.10
B	16.47	94.44
C	20.09	92.54
D	22.39	93.37
E	25.23	97.24
F	22.00	96.05
G	17.20	96.29
H	16.30	97.38
I	14.05	98.12
J	16.53	97.38
K	21.52	95.59
L	19.41	97.13
M	20.09	94.55
N		

Table 1: Geological co-coordinate of Burma14

### ***5.1.a Converting Geological Coordinates:***

Geological coordinates represents a places position based on earths equator and meridian.

The parameters are as follows-

**i**Latitude:

Latitude (shown as a horizontal line) is the angular distance, in degrees, minutes, and seconds of a point north or south of the Equator. Lines of latitude are often referred to as parallels.

**ii** Longitude:

Longitude (shown as a vertical line) is the angular distance, in degrees, minutes, and seconds, of a point east or west of the Prime (Greenwich) Meridian. Lines of longitude are often referred to as meridians.

**iii** Minutes and Seconds :

For precision purposes, degrees of longitude and latitude have been divided into minutes (') and seconds ("). There are 60 minutes in each degree. Each minute is divided into 60 seconds. Seconds can be further divided into tenths, hundredths, or even thousandths.

**iv** Haversine equation :

Obtaining Distance between to real world Co-ordinate is different from getting distance between to 2-dimensional coordinate. I used Haversine formual to calculate Distance between to cities. The pseudo code is given below.

$$a = \sin^2(\Delta\phi/2) + \cos \phi_1 \cdot \cos \phi_2 \cdot \sin^2(\Delta\lambda/2)$$

$$c = 2 \cdot \text{atan2}(\sqrt{a}, \sqrt{1-a})$$

$$d = R \cdot c$$

*$\phi$  is latitude,  $\lambda$  is longitude, R is earth's radius (mean radius = 6,371km);  
where note that angles need to be in radians to pass to trig functions!*

$$\text{distance\_lonititude} = \text{longtittude2} - \text{longtittude1}$$

$$\text{distance\_lattitude} = \text{lattitude2} - \text{lattitude1}$$

$$a = (\sin(\text{distance\_lattitude} / 2))^2 + \cos(\text{lattitude1}) * \cos(\text{lattitude2}) * (\sin(\text{distance\_lonititude} / 2))^2$$

$$c = 2 * \text{atan2}(\text{sqrt}(a), \text{sqrt}(1-a))$$

$$d = R * c \text{ (where R is the radius of the Earth)}$$

## ***5.2 Selecting dividing axes for map reduce:***

To Decide Where to Place the axes to divide the city map into four coordinate we plotted the cities of Burma14 on graph and selected the exes carefully so that no cities rest on any

of the axes. The new horizontal axis intersects Y axis at 95.25 and new vertical axis intersects X axis at 19.00.

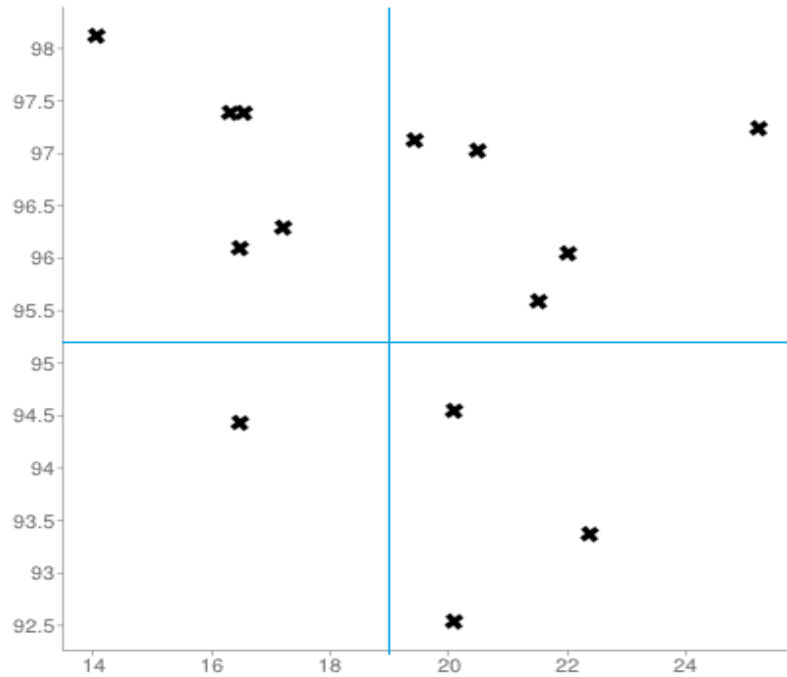


Fig 10 : Map reduction axes of Burma14

I Applied Genetic Algorithm To each Coordinate and saved best three paths instead of one. For the crossover I used a new crossover that I am going to introduce later in this paper.

In total I got 12 routes for four coordinates.

### ***5.3 Applying Genetic Algorithm to Each Coordinate:***

I Applied Genetic Algorithm To each Coordinate and saved best three paths instead of one. For the crossover I used a new crossover that I am going to introduce later in this paper.

In total I got 12 routes for four coordinates.

### ***5.4 combining the routes from four coordinate:***

- Using rules of mathematical combination I combined the 4 routes, one from each coordinate and four merging distance. So from 12 possible routes from each coordinate and 36 merging distances we had to chose four routes and four merging distances. That makes 495 combinations of cities and 58905 merging distance.

## Class City

```
public class City {  
    String cityname = "";  
    double x;  
    double y;  
  
    // Constructs a randomly placed city  
    /* public City(){  
        this.x = (Math.random()*200);  
        this.y = (Math.random()*200);  
    }*/  
  
    // Constructs a city at chosen x, y location  
    public City(String name, double x, double y){  
        cityname = name;  
        this.x = x;  
        this.y = y;  
    }  
  
    // Gets city's x coordinate  
    public double getX(){  
        return this.x;  
    }  
  
    // Gets city's y coordinate
```

```

public double getY(){
    return this.y;
}

//

public String getcityname(){
    return this.cityname;
}

// Gets the distance to given city
public double distanceTo(City city){

    double userLat = getX();

    double venueLat = city.getX();

    double userLng = getY();

    double venueLng = city.getY();

    //double xDistance = Math.abs(getX() - city.getX());

    //double yDistance = Math.abs(getY() - city.getY());

    //double distance = Math.sqrt( (xDistance*xDistance) + (yDistance*yDistance) );

    double latDistance = Math.toRadians(userLat - venueLat);

    double lngDistance = Math.toRadians(userLng - venueLng);

    double a = Math.sin(latDistance / 2) * Math.sin(latDistance / 2)
+ Math.cos(Math.toRadians(userLat)) * Math.cos(Math.toRadians(venueLat))

```



```

* Math.sin(lngDistance / 2) * Math.sin(lngDistance / 2);

double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));

return (int)(AVERAGE_RADIUS_OF_EARTH * c);

}

public final static double AVERAGE_RADIUS_OF_EARTH = 6371;

/*public int calculateDistance(double userLat, double userLng ) {

double venueLat =

double latDistance = Math.toRadians(userLat - venueLat);

double lngDistance = Math.toRadians(userLng - venueLng);

double a = Math.sin(latDistance / 2) * Math.sin(latDistance / 2)

+ Math.cos(Math.toRadians(userLat)) * Math.cos(Math.toRadians(venueLat))

* Math.sin(lngDistance / 2) * Math.sin(lngDistance / 2);

double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));

return (int) (Math.round(AVERAGE_RADIUS_OF_EARTH * c));

}*/

@Override

public String toString(){

return getcityname()+" "+getX()+" "+getY();

}

```

}

## Class GA

```
public class GA {

    /* GA parameters */

    private static final double mutationRate = 0.015;

    private static final int tournamentSize = 5;

    private static final boolean elitism = true;

    // Evolves a population over one generation

    public static Population evolvePopulation(Population pop) {

        Population newPopulation = new Population(pop.populationSize(), false);

        // Keep our best individual if elitism is enabled

        int elitismOffset = 0;

        if (elitism) {

            Tour [] temp = pop.getFittest();

            newPopulation.saveTour(0, temp[0]);

            elitismOffset = 1;

        }

        // Crossover population

        // Loop over the new population's size and create individuals from

        // Current population

        for (int i = elitismOffset; i < newPopulation.populationSize(); i++) {
```

```

// Select parents
Tour parent1 = tournamentSelection(pop);
Tour parent2 = tournamentSelection(pop);

// Crossover parents
Tour child = crossover(parent1, parent2);

// Add child to new population
newPopulation.saveTour(i, child);
}

// Mutate the new population a bit to add some new genetic material
for (int i = elitismOffset; i < newPopulation.populationSize(); i++) {
    mutate(newPopulation.getTour(i));
}

return newPopulation;
}

// Applies crossover to a set of parents and creates offspring
public static Tour crossover(Tour parent1, Tour parent2) {
    // Create new child tour
    Tour child = new Tour();

    // Get start and end sub tour positions for parent1's tour
    int startPos = (int) (Math.random() * parent1.tourSize());

```

```

int endPos = (int) (Math.random() * parent1.tourSize());

// Loop and add the sub tour from parent1 to our child
for (int i = 0; i < child.tourSize(); i++) {

    // If our start position is less than the end position
    if (startPos < endPos && i > startPos && i < endPos) {

        child.setCity(i, parent1.getCity(i));

    } // If our start position is larger
    else if (startPos > endPos) {

        if (!(i < startPos && i > endPos)) {

            child.setCity(i, parent1.getCity(i));

        }

    }

}

// Loop through parent2's city tour
for (int i = 0; i < parent2.tourSize(); i++) {

    // If child doesn't have the city add it
    if (!child.containsCity(parent2.getCity(i))) {

        // Loop to find a spare position in the child's tour
        for (int ii = 0; ii < child.tourSize(); ii++) {

            // Spare position found, add city
            if (child.getCity(ii) == null) {

                child.setCity(ii, parent2.getCity(i));

            }

        }

    }

}

```

```

        break;
    }
}
}
}
return child;
}

// Mutate a tour using swap mutation
private static void mutate(Tour tour) {
    // Loop through tour cities
    for(int tourPos1=0; tourPos1 < tour.tourSize(); tourPos1++){
        // Apply mutation rate
        if(Math.random() < mutationRate){
            // Get a second random position in the tour
            int tourPos2 = (int) (tour.tourSize() * Math.random());

            // Get the cities at target position in tour
            City city1 = tour.getCity(tourPos1);
            City city2 = tour.getCity(tourPos2);

            // Swap them around
            tour.setCity(tourPos2, city1);
            tour.setCity(tourPos1, city2);
        }
    }
}

```

```

    }
}

// Selects candidate tour for crossover
private static Tour tournamentSelection(Population pop) {
    // Create a tournament population
    Population tournament = new Population(tournamentSize, false);
    // For each place in the tournament get a random candidate tour and
    // add it
    for (int i = 0; i < tournamentSize; i++) {
        int randomId = (int) (Math.random() * pop.populationSize());
        tournament.saveTour(i, pop.getTour(randomId));
    }
    // Get the fittest tour
    Tour [] temp = tournament.getFittest();
    Tour fittest = temp[0];
    return fittest;
}
}

```

## Class Population

```
public class Population {

    // Holds population of tours
    Tour[] tours;

    // Construct a population
    public Population(int populationSize, boolean initialise) {
        tours = new Tour[populationSize];

        // If we need to initialise a population of tours do so
        if (initialise) {
            // Loop and create individuals
            for (int i = 0; i < populationSize(); i++) {
                Tour newTour = new Tour();
                newTour.generateIndividual();
                saveTour(i, newTour);
            }
        }
    }

    // Saves a tour
    public void saveTour(int index, Tour tour) {
        tours[index] = tour;
    }
}
```



```

}

// Gets a tour from population
public Tour getTour(int index) {
    return tours[index];
}

// Gets the best tour in the population
public Tour[] getFittest() {
    Tour [] fittest = new Tour[3];
    fittest[0] = tours[0];
    fittest[1] = tours[0];
    fittest[2] = tours[0];
    // Loop through individuals to find fittest
    for (int i = 1; i < populationSize(); i++) {
        if (fittest[0].getFitness() < getTour(i).getFitness()) {
            fittest[2] = fittest[1];
            fittest[1] = fittest[0];
            fittest[0] = getTour(i);
        }
    }
    System.out.println("1"+fittest[0]);
    System.out.println("2"+fittest[1]);
    System.out.println("3"+fittest[2]);
    System.out.println(" ");
}

```

```
    }  
  }  
  //System.out.println("3"+fittest[0]);  
  //System.out.println("4"+fittest[1]);  
  return fittest;  
}  
  
// Gets population size  
public int populationSize() {  
  return tours.length;  
}  
}
```

### **Class Tour**

```
import java.util.ArrayList;  
import java.util.Collections;  
  
public class Tour{  
  
  // Holds our tour of cities  
  private ArrayList tour = new ArrayList<City>();
```

```

// Cache

private double fitness = 0;

private int distance = 0;

// Constructs a blank tour
public Tour(){
    for (int i = 0; i < TourManager.numberOfCities(); i++) {
        tour.add(null);
    }
}

public Tour(ArrayList tour){
    this.tour = tour;
}

// Creates a random individual
public void generateIndividual() {
    // Loop through all our destination cities and add them to our tour
    for (int cityIndex = 0; cityIndex < TourManager.numberOfCities(); cityIndex++) {
        setCity(cityIndex, TourManager.getCity(cityIndex));
    }
    // Randomly reorder the tour
    Collections.shuffle(tour);
}

```

```
// Gets a city from the tour
public City getCity(int tourPosition) {
    return (City)tour.get(tourPosition);
}

// Sets a city in a certain position within a tour
public void setCity(int tourPosition, City city) {
    tour.set(tourPosition, city);
    // If the tours been altered we need to reset the fitness and distance
    fitness = 0;
    distance = 0;
}

// Gets the tours fitness
public double getFitness() {
    if (fitness == 0) {
        fitness = (double)getDistance();
    }
    return fitness;
}

// Gets the total distance of the tour
public int getDistance(){
```

```
if (distance == 0) {  
    int tourDistance = 0;  
    // Loop through our tour's cities  
    for (int cityIndex=0; cityIndex < tourSize(); cityIndex++) {  
        // Get city we're travelling from  
        City fromCity = getCity(cityIndex);  
        // City we're travelling to  
        City destinationCity;  
        // Check we're not on our tour's last city, if we are set our  
        // tour's final destination city to our starting city  
        if(cityIndex+1 < tourSize()){  
            destinationCity = getCity(cityIndex+1);  
        }  
        else{  
            destinationCity = getCity(0);  
        }  
        // Get the distance between the two cities  
        tourDistance += fromCity.distanceTo(destinationCity);  
    }  
    distance = tourDistance;  
}  
return distance;  
}
```

```
// Get number of cities on our tour

public int tourSize() {
    return tour.size();
}

// Check if the tour contains a city

public boolean containsCity(City city){
    return tour.contains(city);
}

@Override

public String toString() {
    String geneString = "|";
    for (int i = 0; i < tourSize(); i++) {
        geneString += getCity(i)+"|";
    }
    return geneString;
}
}
```

### **Class TourManager**

```
import java.util.ArrayList;

public class TourManager {

    // Holds our cities
    public static ArrayList destinationCities = new ArrayList<City>();

    public static ArrayList quarter1 = new ArrayList<City>();
    public static ArrayList quarter2 = new ArrayList<City>();
    public static ArrayList quarter3 = new ArrayList<City>();
    public static ArrayList quarter4 = new ArrayList<City>();

    // Adds a destination city
    public static void addCity(City city) {
        if (city.x >= 19 && city.y <= 95.25) {
            //quarter 3
            quarter3.add(city);
        } else if (city.x < 19 && city.y > 95.25) {
            //quarter 1
            quarter1.add(city);
        } else if (city.x < 19 && city.y < 95.25) {
            //quarter 4
            quarter4.add(city);
        }
    }
}
```

```
    } else if (city.x > 19 && city.y > 95.25) {  
        //quarter 2  
        quarter2.add(city);  
    }  
  
    destinationCities.add(city);  
}  
  
// Get a city  
public static City getCity(int index){  
    return (City)destinationCities.get(index);  
}  
  
// Get the number of destination cities  
public static int numberOfCities(){  
    return destinationCities.size();  
}  
}
```



## Class TSP\_GA

```
public class TSP_GA {

    public static void main(String[] args) {

        // TODO Auto-generated method stub

        // Create and add our cities

        City city = new City("A",16.47, 96.10);
        TourManager.addCity(city);

        City city2 = new City("B",16.47, 94.44);
        TourManager.addCity(city2);

        City city3 = new City("C",20.09, 92.54);
        TourManager.addCity(city3);

        City city4 = new City("D",22.39, 93.37);
        TourManager.addCity(city4);

        City city5 = new City("E",25.23, 97.24);
        TourManager.addCity(city5);

        City city6 = new City("F",22.00, 96.05);
        TourManager.addCity(city6);

        City city7 = new City("G",20.47, 97.02);
        TourManager.addCity(city7);

        City city8 = new City("H",17.20, 96.29);
        TourManager.addCity(city8);

        City city9 = new City("I",16.30, 97.38);
        TourManager.addCity(city9);

    }

}
```

```
City city10 = new City("J",14.05, 98.12);
TourManager.addCity(city10);

City city11 = new City("k",16.53, 97.38);
TourManager.addCity(city11);

City city12 = new City("L",21.52, 95.59);
TourManager.addCity(city12);

City city13 = new City("M",19.41, 97.13);
TourManager.addCity(city13);

City city14 = new City("N",20.09, 94.55);
TourManager.addCity(city14);

Tour [] q1,q2,q3,q4;

TourManager.destinationCities = TourManager.quarter1;

// Initialize population
Population pop = new Population(10, true);
Tour [] temp = pop.getFittest();
System.out.println("Initial distance: " + temp[0].getDistance());

// Evolve population for 100 generations
pop = GA.evolvePopulation(pop);
for (int i = 0; i < 10; i++) {
    pop = GA.evolvePopulation(pop);
}
```

```
}

// Print final results
System.out.println("Finished");

temp = pop.getFittest();
q1 = temp;
System.out.println("Final distance: " + temp[0].getDistance());
System.out.println("Solution:");
System.out.println(pop.getFittest());

TourManager.destinationCities = TourManager.quarter2;

// Initialize population
pop = new Population(10, true);
temp = pop.getFittest();
System.out.println("Initial distance: " + temp[0].getDistance());

// Evolve population for 100 generations
pop = GA.evolvePopulation(pop);
for (int i = 0; i < 10; i++) {
    pop = GA.evolvePopulation(pop);
}
```

```
// Print final results

System.out.println("Finished");

temp = pop.getFittest();

q2 = temp;

System.out.println("Final distance: " + temp[0].getDistance());

System.out.println("Solution:");

System.out.println(pop.getFittest());

TourManager.destinationCities = TourManager.quarter3;

// Initialize population

pop = new Population(10, true);

temp = pop.getFittest();

System.out.println("Initial distance: " + temp[0].getDistance());

// Evolve population for 100 generations

pop = GA.evolvePopulation(pop);

for (int i = 0; i < 10; i++) {

    pop = GA.evolvePopulation(pop);

}

// Print final results

System.out.println("Finished");

temp = pop.getFittest();
```

```
q3 = temp;

System.out.println("Final distance: " + temp[0].getDistance());

System.out.println("Solution:");

System.out.println(pop.getFittest());

TourManager.destinationCities = TourManager.quarter4;

// Initialize population

pop = new Population(10, true);

temp = pop.getFittest();

q4 = temp;

System.out.println("Initial distance: " + temp[0].getDistance());

// Evolve population for 100 generations

pop = GA.evolvePopulation(pop);

for (int i = 0; i < 10; i++) {

    pop = GA.evolvePopulation(pop);

}

// Print final results

System.out.println("Finished");

temp = pop.getFittest();

q4 = temp;

System.out.println("Final distance: " + temp[0].getDistance());
```

```

System.out.println("Solution:");

System.out.println(pop.getFittest());

//System.out.println(q1[0].getDistance() + q1[0].getCity(0).distanceTo(q2[0].getCity(0)) +
q2[0].getDistance() + q2[0].getCity(0).distanceTo(q3[0].getCity(0)) + q3[0].getDistance() +
q1[0].getCity(0).distanceTo(q2[0].getCity(0)) + );

double min = Double.MAX_VALUE;

for (int i=0; i < 3; i++) {
    for (int j=0; j < 3; j++) {
        for (int k=0; k < 3; k++) {
            for (int l=0; l < 3; l++) {

                double d1 = q1[i].getDistance() +
q1[i].getCity(0).distanceTo(q2[j].getCity(0)) + q2[j].getDistance() + q2[j].getCity(q2[j].tourSize()-
1).distanceTo(q3[k].getCity(0)) + q3[k].getDistance() + q3[k].getCity(q3[k].tourSize()-
1).distanceTo(q4[l].getCity(0)) + q4[l].getDistance();

                double d2 = q1[i].getDistance() + q1[i].getCity(q1[i].tourSize()-
1).distanceTo(q2[j].getCity(0)) + q2[j].getDistance() + q2[j].getCity(q2[j].tourSize()-
1).distanceTo(q3[k].getCity(0)) + q3[k].getDistance() + q3[k].getCity(q3[k].tourSize()-
1).distanceTo(q4[l].getCity(0)) + q4[l].getDistance();

                double d3 = q1[i].getDistance() + q1[i].getCity(q1[i].tourSize()-
1).distanceTo(q2[j].getCity(q2[j].tourSize()-1)) + q2[j].getDistance() +
q2[j].getCity(0).distanceTo(q3[k].getCity(0)) + q3[k].getDistance() +
q3[k].getCity(q3[k].tourSize()-1).distanceTo(q4[l].getCity(0)) + q4[l].getDistance();

                double d4 = q1[i].getDistance() + q1[i].getCity(q1[i].tourSize()-
1).distanceTo(q2[j].getCity(q2[j].tourSize()-1)) + q2[j].getDistance() +
q2[j].getCity(0).distanceTo(q3[k].getCity(q3[k].tourSize()-1)) + q3[k].getDistance() +
q3[k].getCity(0).distanceTo(q4[l].getCity(0)) + q4[l].getDistance();

```

```
                double d5 = q1[i].getDistance() + q1[i].getCity(q1[i].tourSize()-
1).distanceTo(q2[j].getCity(q2[j].tourSize()-1)) + q2[j].getDistance() +
q2[j].getCity(0).distanceTo(q3[k].getCity(q3[k].tourSize()-1)) + q3[k].getDistance() +
q3[k].getCity(0).distanceTo(q4[l].getCity(q4[l].tourSize()-1)) + q4[l].getDistance();

                min = Math.min(min, Math.min(Math.min(Math.min(Math.min(d1,
d2),d3),d4),d5));

                }

                }

        }

    }

    System.out.println(min);

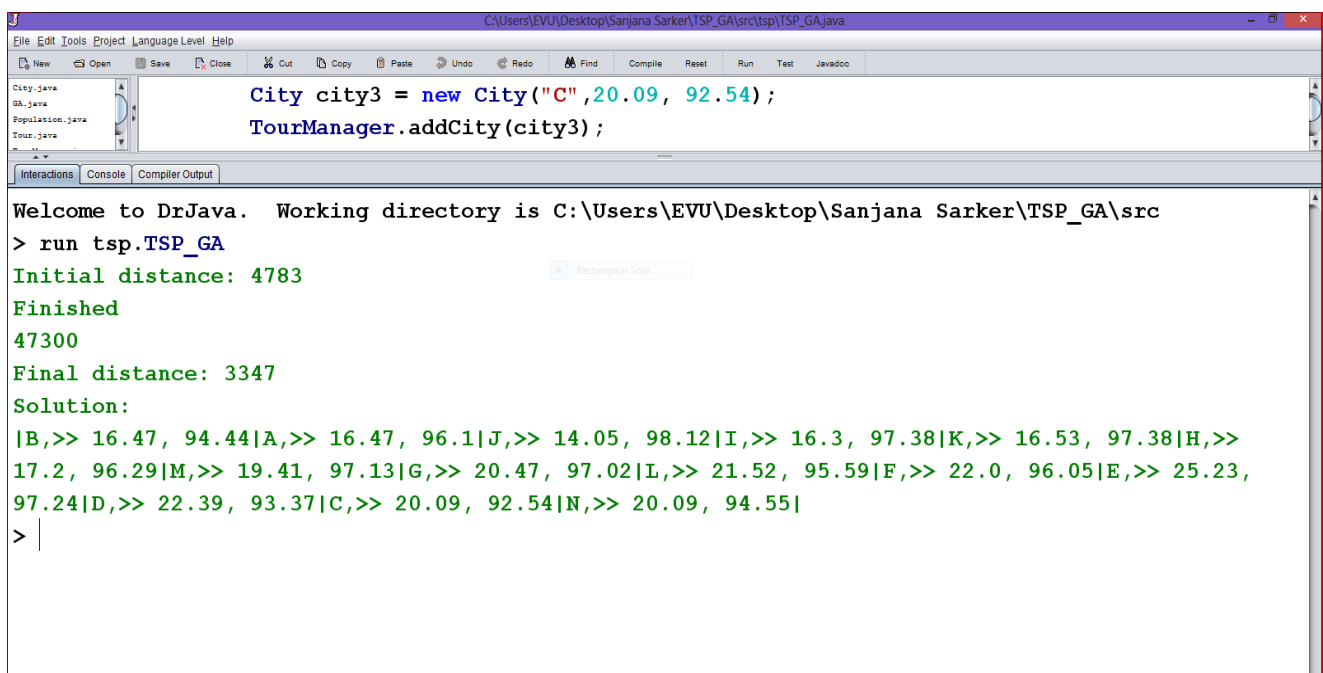
}

}
```

## 6.Findings:

### 6.1 Result using Crossover:

The result we found using the new proposed crossover is optimal. We found the cost to be 3347 km. Which is actually 24 km more than the expected, but is due to conversion that I had to do to calculate the distance between two geographical coordinates. However, I compared the graph between the optimal solution of burma14 to the graph of solution I got using the new crossover, Both the graphs are same.



```
City.java
GA.java
Population.java
Tour.java

City city3 = new City("C",20.09, 92.54);
TourManager.addCity(city3);

Welcome to DrJava. Working directory is C:\Users\EVU\Desktop\Sanjana Sarker\TSP_GA\src
> run tsp.TSP_GA
Initial distance: 4783
Finished
47300
Final distance: 3347
Solution:
|B,>> 16.47, 94.44|A,>> 16.47, 96.1|J,>> 14.05, 98.12|I,>> 16.3, 97.38|K,>> 16.53, 97.38|H,>>
17.2, 96.29|M,>> 19.41, 97.13|G,>> 20.47, 97.02|L,>> 21.52, 95.59|F,>> 22.0, 96.05|E,>> 25.23,
97.24|D,>> 22.39, 93.37|C,>> 20.09, 92.54|N,>> 20.09, 94.55|
> |
```

Fig 11 : Screenshot of Solution using new crossover



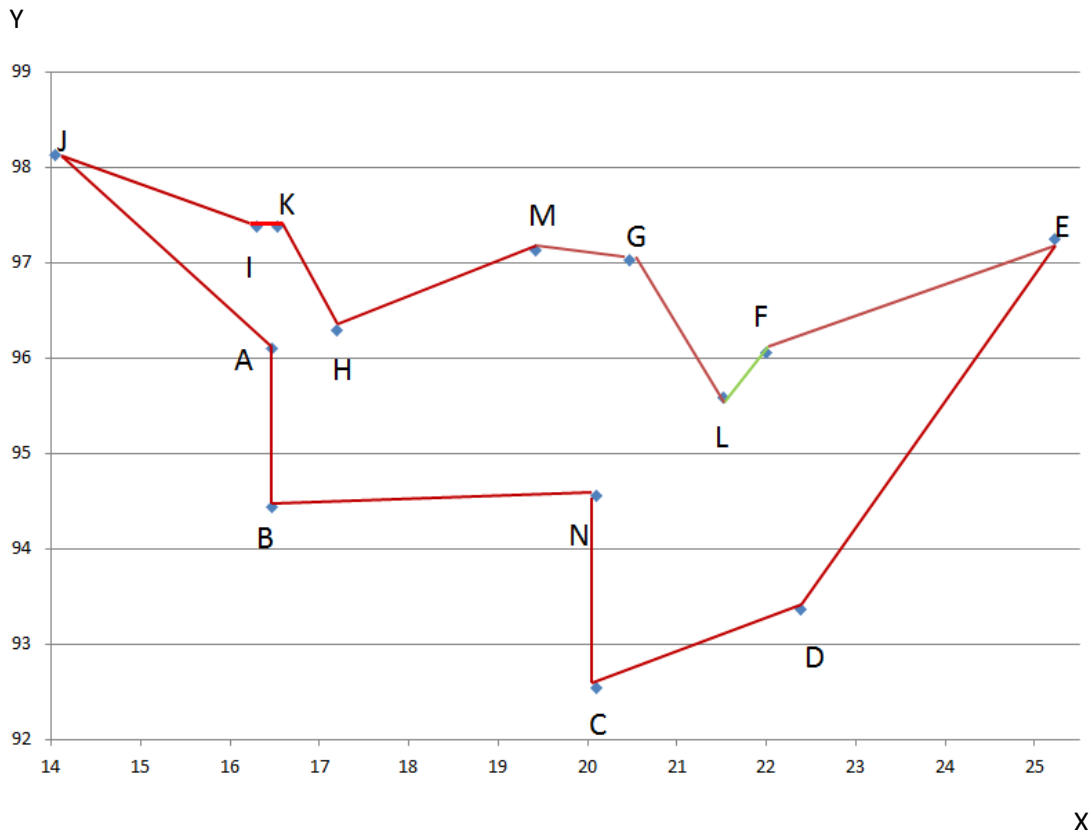


Fig 12 : Graph of Solution using new crossover

Above, in figure 11, the graph of solution of Burma 14 using new crossover is given and below, in figure 12, the graph of best known solution of Burma 14 is give. Comparing this two graphs, it is clear that both the graphs are same, i.e. both the route are same. That concludes the solution I got is an optimal one.

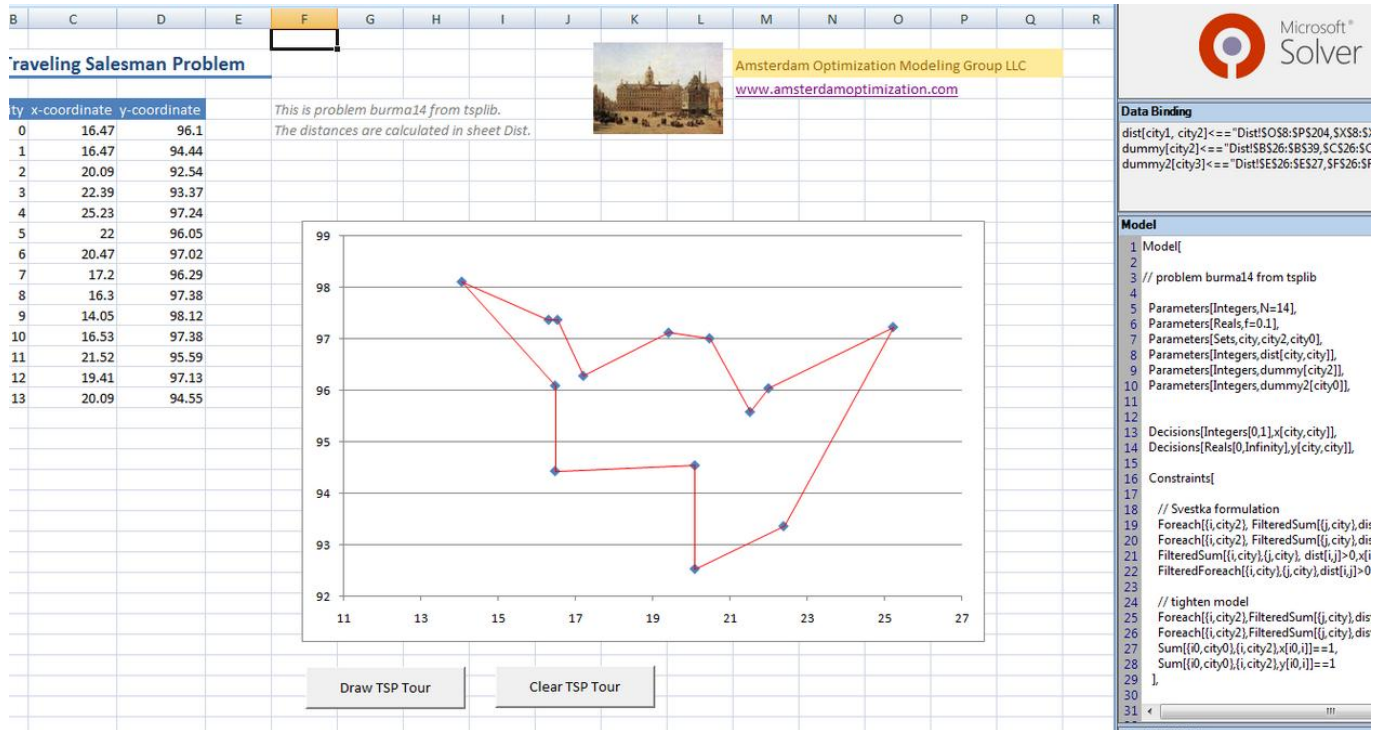


Fig 13: Graph of optimal solution

## 6.2 Results using Map Reduce, GA and Permutation:

Most of the research that used divide and conquer method to solve TSP could not achieve optimal solution. However, As, this time, not only the optimal solution, but also some other reasonably low cost path was considered, the yielded result was close to optimal.

Here, the best result we got is 3422 km, And after studying the graph, I found that only routes between 2 cities we altered from the optimal solution graph.

```

C:\Users\EVU\Desktop\Sanjana Sarker\TSP_GA\src\tsp_GA.java
File Edit Tools Project Language Level Help
New Open Save Close Cut Copy Paste Undo Redo Find Compile Reset Run Test Javadoc
City.java
GA.java
Population.java
Tous.java
TousManager.java

// Evolve population for 100 generations
pop = GA.evolvePopulation(pop);

Interactions Console Compiler Output
Welcome to DrJava. Working directory is C:\Users\EVU\Desktop\Sanjana Sarker\TSP_GA\src
> run tsp.TSP_GA
Initial distance: 5345
Finished
1653
Final distance: 3422
Solution:
|J,>> 14.05, 98.12|I,>> 16.3, 97.38|K,>> 16.53, 97.38|H,>> 17.2, 96.29|M,>> 19.41, 97.13|G,>>
20.47, 97.02|F,>> 22.0, 96.05|L,>> 21.52, 95.59|E,>> 25.23, 97.24|D,>> 22.39, 93.37|C,>> 20.09,
92.54|N,>> 20.09, 94.55|B,>> 16.47, 94.44|A,>> 16.47, 96.1|
> |

```

Fig 13: Screenshot of solution using Map Reduce, GA, Combination

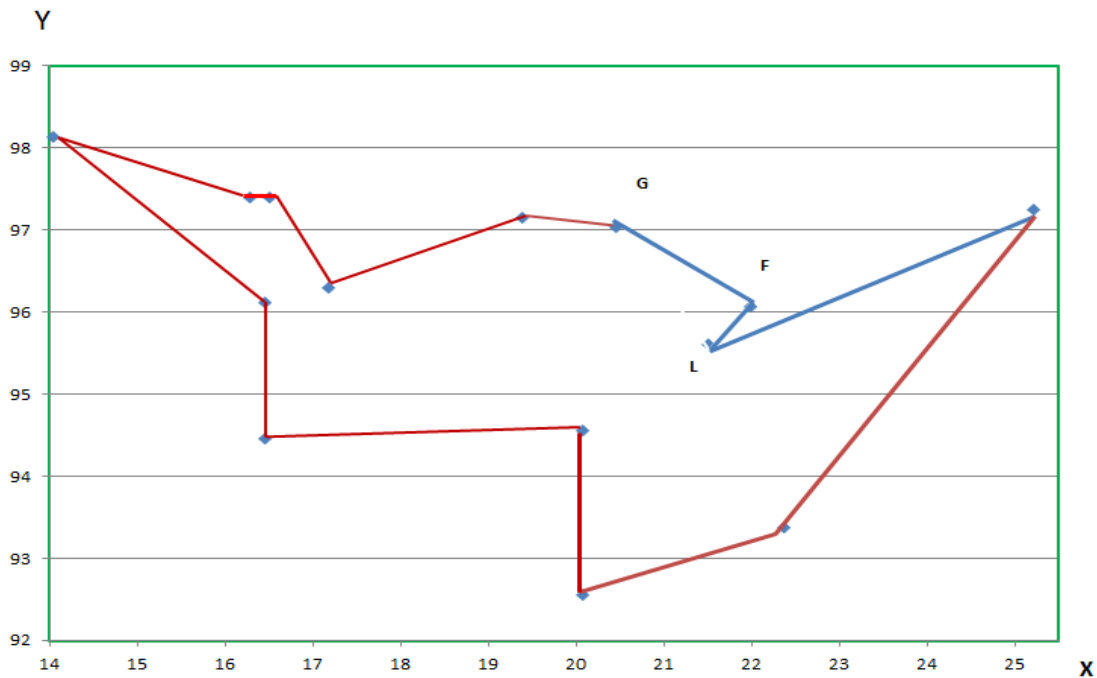


Fig 14: Graph of solution using Map Reduce, GA & combination

## **7. Conclusion and Continuation**

### ***7.1 Future Works:***

- The data I used consists of only 14 cities. So how my new approach will perform is yet to be studied.
- Due to time constraint, and less complex data of Burma 14, I did not consider the Merging between the routes of diagonal coordinates. I expect the result to be better if done so.

### ***7.2 Conclusion:***

My objective was to find a new crossover technique for Genetic Algorithm to solve TSP which I did successfully. My new crossover performed exceptionally well and yielded optimized solution.

My second objective was to Find a new approach to Solve TSP. I combined Map Reduce, GA and Combination to create a new approach. The solution we found was close to optimal. I expect the approach to perform better if improvement is done.

## 8. REFERENCES

- [1] Zbigniew Michalewicz. Genetic Algorithms + Data Structures = Evolution Programs. Springer-Verlag, 2nd edition, 1994.
- [2] David E. Goldberg. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, 1989.
- [3] [http://en.wikipedia.org/wiki/Genetic\\_algorithm](http://en.wikipedia.org/wiki/Genetic_algorithm)
- [4] [http://www.doc.ic.ac.uk/~nd/surprise\\_96/journal/vol1/hmw/article1.html](http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol1/hmw/article1.html)
- [5] <http://amsterdamoptimization.com/pic/tspmip.png>
- [6] Kylie Brant, Aurther Benjamin. Genetic Algorithm and the Travelling Salesman Problem. December 2000.
- [7] Siamak Sarmady. An Investigation on Genetic Algorithm Parameters. Univiersiti Sains Malayasia
- [8] Kusum Deep, Hadush Mebrahtu. Combined Mutation Operators of Genetic Algorithm for the Travelling Salesman problem. International Journal of Combinatorial Optimization Problems and Informatics, Vol. 2, No.3, Sep-Dec 2011, pp. 1-23, ISSN: 2007-1558.
- [9] Dr.Sabry M. Abdel-Moetty , Asmaa O. Heakil . Enhanced Traveling Salesman Problem Solving using Genetic Algorithm Technique with modified Sequential Constructive Crossover Operator . IJCSNS International Journal of Computer Science and Network Security, VOL.12 No.6, June 2012
- [10] R.SIVARAJ . Solving Travelling Salesman Problem using Clustering Genetic Algorithm . R.Sivaraj et al. / International Journal on Computer Science and Engineering (IJCSE).
- [11] Adewole Philip , Akinwale Adio Taofiki , Otunbanowo Kehinde , A Genetic Algorithm for Solving Travelling Salesman Problem . (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 2, No.1, January 2011