A

PROJECT REPORT

ON

**“COMPARISON OF ALGORITHMS FOR TRAVELLING SALESMAN PROBLEM”**

Submitted in the partial fulfillment of the requirement for the award of degree of

BACHELOR OF TECHNOLOGY

In

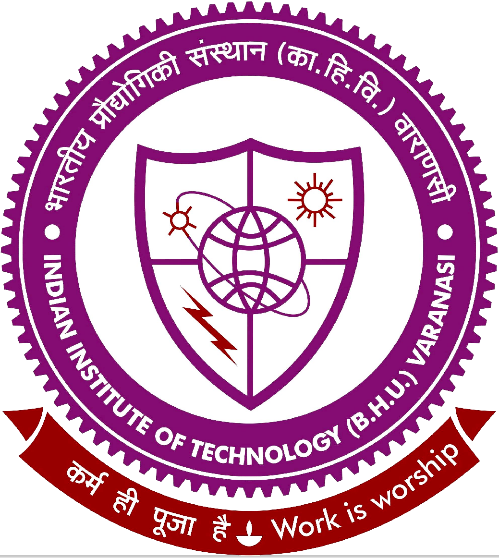
**Mechanical Engineering**

**INDIAN INSTITUTE OF TECHNOLOGY**

**(BANARAS HINDU UNIVERSITY)**

VARANASI-221005

Session: 2016-17



***GUIDED BY***

**Dr. CHERIAN SAMUEL**

***SUBMITTED BY***

**HARIOM BAIRWA (13135044)**

**HARISH CHOUDHARY (13135045)**

**KANHEKAR SHUBHAM BHIMRAO (13135050)**

# **Acknowledgement**

I have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and organizations. I would like to extend my sincere thanks to all of them.

I am highly indebted to Dr. Cherian Samuel, Assistant Professor, Department of Mechanical Engineering, IIT(BHU), for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project.

My thanks and appreciations also go to my colleague in developing the project and people who have willingly helped me out with their abilities.

Thanks

# **Abstract**

The travel salesman problem is a well-known problem which has become a comparison benchmark test for different computational methods. Its solution is computationally difficult, although the problem is easily expressed. A salesperson must make a closed complete tour of a given number of cities. All cities are connected by roads, and each city can be visited only once. The program must solve the optimization problem by minimizing the value represented by total tour length, and by changing the order of the city.

# **Contents**

1. Introduction 4
2. Objective 6
3. Applications 7
4. Why permutation and combination is not feasible 10
5. Different Approaches 11

Greedy algorithm

Dynamic programming

Genetic algorithm

1. Comparisons and result 12
2. References 13

Appendix 1

Appendix 2

# **Introduction**

## Origin:

The traveling salesman problem (TSP) were studied in the 18th century by a mathematician from Ireland named Sir William Rowam Hamilton and by the British mathematician named Thomas Penyngton Kirkman. Detailed discussion about the work of Hamilton & Kirkman can be seen from the book titled Graph. It is believed that the general form of the TSP have been first studied by Kalr Menger in Vienna and Harvard. The problem was later promoted by Hassler, Whitney & Merrill at Princeton.

## Definition:

Given a set of cities and the cost of travel (or distance) between each possible pairs, the TSP, is to find the best possible way of visiting all the cities and returning to the starting point that minimize the travel cost (or travel distance).

## Mathematical formulation:

The distances between any pair of cities are assumed to be known by the salesman. Distance can be replaced by another notion, such as time or money. In the following the term ’cost’ is used to represent any such notion.

This problem, the traveling salesman problem (TSP), is one of the most widely studied problems in combinatorial optimization. The problem is easy to state, but hard to solve. Mathematically, the problem may be stated as follows:

Given a ‘cost matrix’ C = (cij), where cij represents the cost of going from city i to city j, (i, j = 1, ..., n), find a permutation (i1, i2, i3, ..., in) of the integers from 1 through n that minimizes the quantity:

Properties of the cost matrix C are used to classify problems.

• If cij = cji for all i and j, the problem is said to be symmetric; otherwise, it is asymmetric.

• If the triangle inequality holds (cik ≤ cij + cjk for all i, j and k), the problem is said to be metric.

• If cij are Euclidean distances between points in the plane, the problem is said to be Euclidean. A Euclidean problem is, of course, both symmetric and metric.

## Complexity:

Given n is the number of cities to be visited, the total number of possible routes covering all cities can be given as a set of feasible solutions of the TSP and is given as (n-1)!/2.

Classification Broadly, the TSP is classified as symmetric travelling salesman problem (sTSP) and asymmetric travelling salesman problem (aTSP).

# **2. Objective**

For a given set of 40 cities, we have to find optimum solution by applying three algorithms and do comparison between those:

[{"x":116,"y":404},{"x":161,"y":617},{"x":16,"y":97},{"x":430,"y":536},{"x":601,"y":504},{"x":425,"y":461},{"x":114,"y":544},{"x":127,"y":118},{"x":163,"y":357},{"x":704,"y":104},{"x":864,"y":125},{"x":847,"y":523},{"x":742,"y":170},{"x":204,"y":601},{"x":421,"y":377},{"x":808,"y":49},{"x":860,"y":466},{"x":844,"y":294},{"x":147,"y":213},{"x":550,"y":124},{"x":238,"y":313},{"x":57,"y":572},{"x":664,"y":190},{"x":612,"y":644},{"x":456,"y":154},{"x":120,"y":477},{"x":542,"y":313},{"x":620,"y":29},{"x":245,"y":246},{"x":611,"y":578},{"x":627,"y":373},{"x":534,"y":286},{"x":577,"y":545},{"x":539,"y":340},{"x":794,"y":328},{"x":855,"y":139},{"x":700,"y":47},{"x":275,"y":593},{"x":130,"y":196},{"x":863,"y":35}]

# **3. Applications**

## Drilling of printed circuit boards:

A direct application of the TSP is in the drilling problem of printed circuit boards (PCBs). To connect a conductor on one layer with a conductor on another layer, or to position the pins of integrated circuits, holes have to be drilled through the board. The holes may be of different sizes. To drill two holes of different diameters consecutively, the head of the machine has to move to a tool box and change the drilling equipment. This is quite time consuming. Thus it is clear that one has to choose some diameter, drill all holes of the same diameter, change the drill, drill the holes of the next diameter, etc. Thus, this drilling problem can be viewed as a series of TSPs, one for each hole diameter, where the 'cities' are the initial position and the set of all holes that can be drilled with one and the same drill. The 'distance' between two cities is given by the time it takes to move the drilling head from one position to the other. The aim is to minimize the travel time for the machine head.

## X-Ray crystallography:

Analysis of the structure of crystals is an important application of the TSP. Here an X-ray diffractometer is used to obtain information about the structure of crystalline material. To this end a detector measures the intensity of X-ray reflections of the crystal from various positions. Whereas the measurement itself can be accomplished quite fast, there is a considerable overhead in positioning time since up to hundreds of thousand positions have to be realized for some experiments. The time needed to move from one position to the other can be computed very accurately. The result of the experiment does not depend on the sequence in which the measurements at the various positions are taken. However, the total time needed for the experiment depends on the sequence. Therefore, the problem consists of finding a sequence that minimizes the total positioning time. This leads to a traveling salesman problem.

## Computer wiring:

Modules are located on a computer board and a given subset of pins has to be connected. In contrast to the usual case where a Steiner tree connection is desired, here the requirement is that no more than two wires are attached to each pin. Hence we have the problem of finding a shortest Hamiltonian path with unspecified starting and terminating points. A similar situation occurs for the so-called test bus wiring. To test the manufactured board one has to realize a connection which enters the board at some specified point, runs through all the modules, and terminates at some specified point. For each module we also have a specified entering and leaving point for this test wiring. This problem also amounts to solving a Hamiltonian path problem with the difference that the distances are not symmetric and that starting and terminating point are specified.

## The order-picking problem in warehouses:

This problem is associated with material handling in a warehouse. Assume that at a warehouse an order arrives for a certain subset of the items stored in the warehouse. Some vehicle has to collect all items of this order to ship them to the customer. The relation to the TSP is immediately seen. The storage locations of the items correspond to the nodes of the graph. The distance between two nodes is given by the time needed to move the vehicle from one location to the other. The problem of finding a shortest route for the vehicle with minimum pickup time can now be solved as a TSP.

## Vehicle routing:

Suppose that in a city n mail boxes have to be emptied every day within a certain period of time, say 1 hour. The problem is to find the minimum number of trucks to do this and the shortest time to do the collections using this number of trucks. As another example, suppose that n customers require certain amounts of some commodities and a supplier has to satisfy all demands with a fleet of trucks. The problem is to find an assignment of customers to the trucks and a delivery schedule for each truck so that the capacity of each truck is not exceeded and the total travel distance is minimized. Several variations of these two problems, where time and capacity constraints are combined, are common in many real-world applications. This problem is solvable as a TSP if there are no time and capacity constraints and if the number of trucks is fixed (say m). In this case we obtain an m - salesmen problem.

## Mask plotting in PCB production:

For the production of each layer of a printed circuit board, as well as for layers of integrated semiconductor devices, a photographic mask has to be produced. In our case for printed circuit boards this is done by a mechanical plotting device. The plotter moves a lens over a photosensitive coated glass plate. The shutter may be opened or closed to expose specific parts of the plate. There are different apertures available to be able to generate different structures on the board. Two types of structures have to be considered. A line is exposed on the plate by moving the closed shutter to one endpoint of the line, then opening the shutter and moving it to the other endpoint of the line. Then the shutter is closed. A point type structure is generated by moving (with the appropriate aperture) to the position of that point then opening the shutter just to make a short flash, and then closing it again. Exact modeling of the plotter control problem leads to a problem more complicated than the TSP and also more complicated than the rural postman problem.

# **4. Why permutation and combination is not feasible**

In modern computers we can even solve non-trivially-sized problems that way. The difficulty is how fast the computation time scales upwards as the number of points increases. The number of combinations you have to investigate is *n*! where *n* is the number of points. That's *n*!=*n*\*(*n*-1)\*(*n*-2)\*...3\*2\*1. Let’s take 10 cities to see how fast it is:   
  
1!=1  
2!=2  
3!=6  
4!=24  
5!=120  
6!=720  
7!=5040  
8!=40320  
9!=362880  
10!=3628800  
  
It barely hit double digits and we're already at well over a million combinations, each of which have to be tried, and each of which themselves will take some multiple of *n* operations to investigate. So it will work just fine with small problems (*n*<10) but poorly with only slightly larger problems (10<=*n*<15) and not at all with large problems and by large, It mean not very large at all, that is 50!=~3.0414\*10^64.

# **5. Different approaches:**

## Greedy algorithm:

This algorithm belongs to the heuristic algorithms category, which searches for the local optima and optimizes the local best solution to ﬁnd the global optima. It begins by sorting all the edges and then selects the edge with the minimum cost. It continues selecting the best next choices given a condition that no loops are formed. The computational complexity of the greedy algorithm is O(N2 log2 (N)) and there is no guarantee that a global optimum solution is found. On the other hand, the greedy algorithm terminates in a reasonable number of steps.

## Dynamic programming:

Dynamic Programming is a method of solving problems by breaking the solution into a set of steps or stages so that the solution of the problem can be viewed from a series of interrelated decisions.

In dynamic programming, a series of optimal decisions are made by using the principle of optimality. The principle of optimality: if the optimal total solution, then the solution to the kth  stage is also optimal. With the principle of optimality it is guaranteed that decision making at some stage is the right decision for the later stages. The essence of dynamic programming is to remove a small part of a problem at every step, and then solve the smaller problems and use the results of the settlement to remedy. The solution is added back to the issue in the next step.

## Genetic algorithm:

Genetic Algorithm (GA) works in a way similar to the nature. A basic GA starts with a randomly generated population of candidate solutions. Some (or all) candidates are then mated to produce offspring and some go through a mutating process. Each candidate has a fitness value telling us how good they are. By selecting the most fit candidates for mating and mutation the overall fitness of the population will increase. Applying GA to the TSP involves implementing a crossover routine, a measure of fitness, and also a mutation routine. A good measure of fitness is the actual length of the solution.

# **6. Comparison and results:**

* With higher no. of iterations than the greedy search algorithm, genetic algorithm is able to find a shorter route.
* The Genetic algorithm does not guarantee to ﬁnds the shortest path, although it approaches it.
* Dynamic programming can be used for a small company. Dynamic programming gives shortest path or minimum length of tour route.

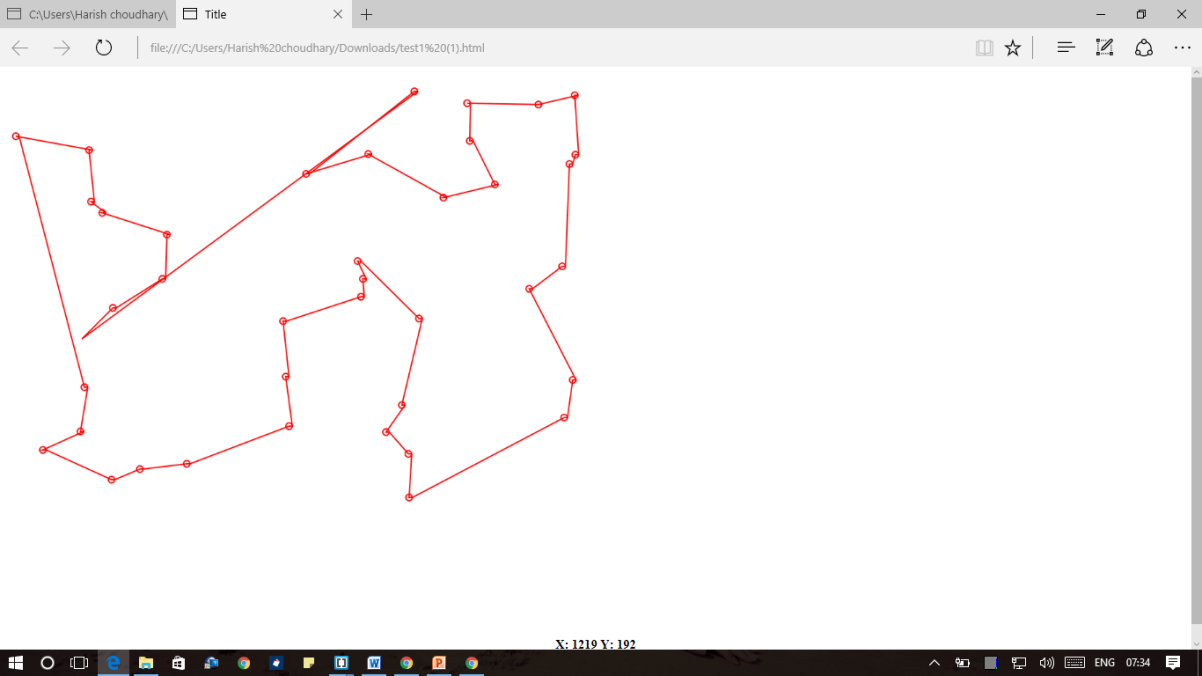


Figure :Result for 40 cities using greedy algorithm

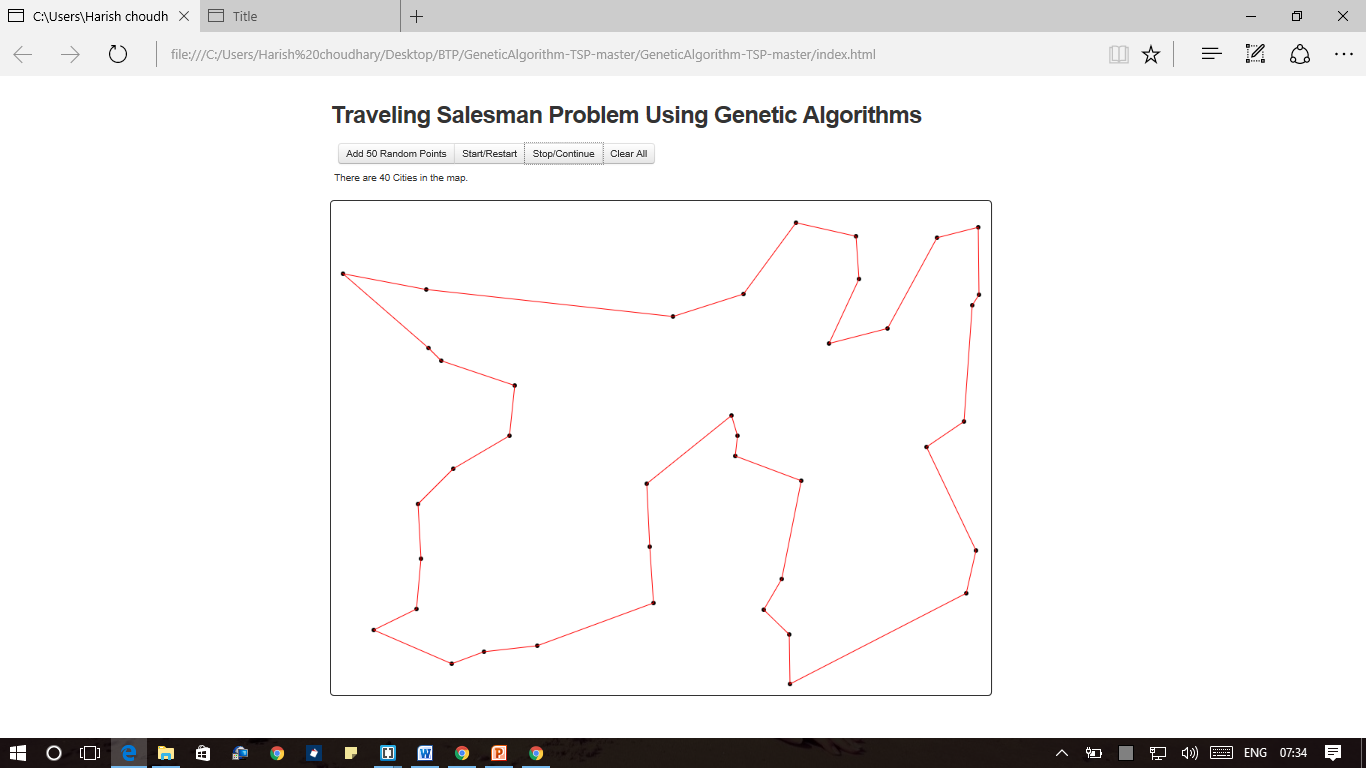


Figure : Result for 40 cities using genetic algorithm

# **7. References**

1. The Traveling Salesman Problem, Michael Jünger, Gerhard Reinelt , Giovanni RinaMi, M.O. Ball et al., Eds., Handbooks in OR & MS, VoL 7, page no. 225-330.

2. A random-key genetic algorithm for the generalized traveling salesman problem, Lawrence V. Snyder, Mark S. Daskin, European Journal of Operational Research 174 (2006) 38–53

# **APPENDIX 1**

## DATA POINTS:

40 points taken for the problem are as:

{"x":116,"y":404},{"x":161,"y":617},{"x":16,"y":97},{"x":430,"y":536},{"x":601,"y":504},{"x":425,"y":461},{"x":114,"y":544},{"x":127,"y":118},{"x":163,"y":357},{"x":704,"y":104},{"x":864,"y":125},{"x":847,"y":523},{"x":742,"y":170},{"x":204,"y":601},{"x":421,"y":377},{"x":808,"y":49},{"x":860,"y":466},{"x":844,"y":294},{"x":147,"y":213},{"x":550,"y":124},{"x":238,"y":313},{"x":57,"y":572},{"x":664,"y":190},{"x":612,"y":644},{"x":456,"y":154},{"x":120,"y":477},{"x":542,"y":313},{"x":620,"y":29},{"x":245,"y":246},{"x":611,"y":578},{"x":627,"y":373},{"x":534,"y":286},{"x":577,"y":545},{"x":539,"y":340},{"x":794,"y":328},{"x":855,"y":139},{"x":700,"y":47},{"x":275,"y":593},{"x":130,"y":196},{"x":863,"y":35}

# **APPENDIX 2**

## Greedy algorithm:

INPUT: 40 POINTS FROM APPENDIX 1

OUTPUT: A GRAPH WITH MINIMUM DISTANCE PATH

var coordinates\_Copy= [{"x":116,"y":404},{"x":161,"y":617},{"x":16,"y":97},{"x":430,"y":536},{"x":601,"y":504},{"x":425,"y":461},{"x":114,"y":544},{"x":127,"y":118},{"x":163,"y":357},{"x":704,"y":104},{"x":864,"y":125},{"x":847,"y":523},{"x":742,"y":170},{"x":204,"y":601},{"x":421,"y":377},{"x":808,"y":49},{"x":860,"y":466},{"x":844,"y":294},{"x":147,"y":213},{"x":550,"y":124},{"x":238,"y":313},{"x":57,"y":572},{"x":664,"y":190},{"x":612,"y":644},{"x":456,"y":154},{"x":120,"y":477},{"x":542,"y":313},{"x":620,"y":29},{"x":245,"y":246},{"x":611,"y":578},{"x":627,"y":373},{"x":534,"y":286},{"x":577,"y":545},{"x":539,"y":340},{"x":794,"y":328},{"x":855,"y":139},{"x":700,"y":47},{"x":275,"y":593},{"x":130,"y":196},{"x":863,"y":35}];

var coordinates= [{"x":116,"y":404},{"x":161,"y":617},{"x":16,"y":97},{"x":430,"y":536},{"x":601,"y":504},{"x":425,"y":461},{"x":114,"y":544},{"x":127,"y":118},{"x":163,"y":357},{"x":704,"y":104},{"x":864,"y":125},{"x":847,"y":523},{"x":742,"y":170},{"x":204,"y":601},{"x":421,"y":377},{"x":808,"y":49},{"x":860,"y":466},{"x":844,"y":294},{"x":147,"y":213},{"x":550,"y":124},{"x":238,"y":313},{"x":57,"y":572},{"x":664,"y":190},{"x":612,"y":644},{"x":456,"y":154},{"x":120,"y":477},{"x":542,"y":313},{"x":620,"y":29},{"x":245,"y":246},{"x":611,"y":578},{"x":627,"y":373},{"x":534,"y":286},{"x":577,"y":545},{"x":539,"y":340},{"x":794,"y":328},{"x":855,"y":139},{"x":700,"y":47},{"x":275,"y":593},{"x":130,"y":196},{"x":863,"y":35}];

var startPoint=coordinates[0];

delete coordinates[0];

function nextPoint(startArray,coordinatesArray) {

var orderedArray=[];

for(var i=0;i<coordinates.length;i++){

if(coordinatesArray[i]===undefined){

}else {

var distance=(startArray.x-coordinatesArray[i].x)\*(startArray.x-coordinatesArray[i].x)+(startArray.y-coordinatesArray[i].y)\*(startArray.y-coordinatesArray[i].y);

orderedArray[i]={dist:distance,order:coordinatesArray[i].n}

}

}

orderedArray.sort(compareDist);

return orderedArray[0].order;

}

function compareDist(objA, objB) {

if (objA.dist > objB.dist) {

return 1;

} else if (objA.dist === objB.dist) {

return 0;

} else {

return -1;

}

}

var order=[];

console.log(nextPoint(startPoint,coordinates));

order.push(nextPoint(startPoint,coordinates));

for(var i=0;i<coordinates.length-2;i++){

startPoint=coordinates\_Copy[nextPoint(startPoint,coordinates)];

delete coordinates[nextPoint(startPoint,coordinates)];

console.log(nextPoint(startPoint,coordinates));

order.push(nextPoint(startPoint,coordinates));

}

window.onload=function () {

var w=window.innerWidth-50;

var h=(window.innerHeight-50);

var paper=document.getElementById("paper");

paper.onmousemove=showCoords;

paper.setAttribute("width",w);

paper.setAttribute("height",h);

var c=paper.getContext("2d");

c.fillStyle="black";

c.fillRect(0,0,paper.width,paper.height);

c.strokeStyle="white";

c.lineWidth=2;

c.beginPath();

c.moveTo(coordinates\_Copy[0].x,coordinates\_Copy[0].y);

for(var i=0;i<coordinates\_Copy.length-1;i++){

console.log(coordinates\_Copy[order[i]]);

c.lineTo(coordinates\_Copy[order[i]].x,coordinates\_Copy[order[i]].y);

}

c.lineTo(coordinates\_Copy[0].x,coordinates\_Copy[0].y);

c.stroke();

}

function showCoords(eventObj) {

var coords=document.getElementById("coords");

var x=eventObj.clientX-8;

var y=eventObj.clientY-7;

coords.innerHTML="X: "+x+" "+"Y: "+y;

}

## Genetic algorithm:

INPUT: 40 POINTS FROM APPENDIX 1

OUTPUT: A GRAPH WITH MINIMUM DISTANCE PATH

**utils.js** :

Array.prototype.clone = function() { return this.slice(0); }

Array.prototype.shuffle = function() {

for(var j, x, i = this.length-1; i; j = randomNumber(i), x = this[--i], this[i] = this[j], this[j] = x);

return this;

};

Array.prototype.indexOf = function (value) {

for(var i=0; i<this.length; i++) {

if(this[i] === value) {

return i;

}

}

}

Array.prototype.deleteByValue = function (value) {

var pos = this.indexOf(value);

this.splice(pos, 1);

}

Array.prototype.next = function (index) {

if(index === this.length-1) {

return this[0];

} else {

return this[index+1];

}

}

Array.prototype.previous = function (index) {

if(index === 0) {

return this[this.length-1];

} else {

return this[index-1];

}

}

Array.prototype.swap = function (x, y) {

if(x>this.length || y>this.length || x === y) {return}

var tem = this[x];

this[x] = this[y];

this[y] = tem;

}

Array.prototype.roll = function () {

var rand = randomNumber(this.length);

var tem = [];

for(var i = rand; i<this.length; i++) {

tem.push(this[i]);

}

for(var i = 0; i<rand; i++) {

tem.push(this[i]);

}

return tem;

}

Array.prototype.reject = function (array) {

return $.map(this,function (ele) {

return $.inArray(ele, array) < 0 ? ele : null;

})

}

function intersect(x, y) {

return $.map(x, function (xi) {

return $.inArray(xi, y) < 0 ? null : xi;

})

}

function Point(x, y) {

this.x = x;

this.y = y;

}

function randomPoint() {

var randomx = randomNumber(WIDTH);

var randomy = randomNumber(HEIGHT);

var randomPoint = new Point(randomx, randomy);

return randomPoint;

}

function randomNumber(boundary) {

return parseInt(Math.random() \* boundary);

//return Math.floor(Math.random() \* boundary);

}

function distance(p1, p2) {

return euclidean(p1.x-p2.x, p1.y-p2.y);

}

function euclidean(dx, dy) {

return Math.sqrt(dx\*dx + dy\*dy);

}

**algorithm.js** :

function GAInitialize() {

countDistances();

for(var i=0; i<POPULATION\_SIZE; i++) {

population.push(randomIndivial(points.length));

}

setBestValue();

}

function GANextGeneration() {

currentGeneration++;

selection();

crossover();

mutation();

setBestValue();

}

function tribulate() {

//for(var i=0; i<POPULATION\_SIZE; i++) {

for(var i=population.length>>1; i<POPULATION\_SIZE; i++) {

population[i] = randomIndivial(points.length);

}

}

function selection() {

var parents = new Array();

var initnum = 4;

parents.push(population[currentBest.bestPosition]);

parents.push(doMutate(best.clone()));

parents.push(pushMutate(best.clone()));

parents.push(best.clone());

setRoulette();

for(var i=initnum; i<POPULATION\_SIZE; i++) {

parents.push(population[wheelOut(Math.random())]);

}

population = parents;

}

function crossover() {

var queue = new Array();

for(var i=0; i<POPULATION\_SIZE; i++) {

if( Math.random() < CROSSOVER\_PROBABILITY ) {

queue.push(i);

}

}

queue.shuffle();

for(var i=0, j=queue.length-1; i<j; i+=2) {

doCrossover(queue[i], queue[i+1]);

}

}

function doCrossover(x, y) {

child1 = getChild('next', x, y);

child2 = getChild('previous', x, y);

population[x] = child1;

population[y] = child2;

}

function getChild(fun, x, y) {

solution = new Array();

var px = population[x].clone();

var py = population[y].clone();

var dx,dy;

var c = px[randomNumber(px.length)];

solution.push(c);

while(px.length > 1) {

dx = px[fun](px.indexOf(c));

dy = py[fun](py.indexOf(c));

px.deleteByValue(c);

py.deleteByValue(c);

c = dis[c][dx] < dis[c][dy] ? dx : dy;

solution.push(c);

}

return solution;

}

function mutation() {

for(var i=0; i<POPULATION\_SIZE; i++) {

if(Math.random() < MUTATION\_PROBABILITY) {

if(Math.random() > 0.5) {

population[i] = pushMutate(population[i]);

} else {

population[i] = doMutate(population[i]);

}

i--;

}

}

}

function preciseMutate(orseq) {

var seq = orseq.clone();

if(Math.random() > 0.5){

seq.reverse();

}

var bestv = evaluate(seq);

for(var i=0; i<(seq.length>>1); i++) {

for(var j=i+2; j<seq.length-1; j++) {

var new\_seq = swap\_seq(seq, i,i+1,j,j+1);

var v = evaluate(new\_seq);

if(v < bestv) {bestv = v, seq = new\_seq; };

}

}

return seq;

}

function preciseMutate1(orseq) {

var seq = orseq.clone();

var bestv = evaluate(seq);

for(var i=0; i<seq.length-1; i++) {

var new\_seq = seq.clone();

new\_seq.swap(i, i+1);

var v = evaluate(new\_seq);

if(v < bestv) {bestv = v, seq = new\_seq; };

}

return seq;

}

function swap\_seq(seq, p0, p1, q0, q1) {

var seq1 = seq.slice(0, p0);

var seq2 = seq.slice(p1+1, q1);

seq2.push(seq[p0]);

seq2.push(seq[p1]);

var seq3 = seq.slice(q1, seq.length);

return seq1.concat(seq2).concat(seq3);

}

function doMutate(seq) {

mutationTimes++;

do {

m = randomNumber(seq.length - 2);

n = randomNumber(seq.length);

} while (m>=n)

for(var i=0, j=(n-m+1)>>1; i<j; i++) {

seq.swap(m+i, n-i);

}

return seq;

}

function pushMutate(seq) {

mutationTimes++;

var m,n;

do {

m = randomNumber(seq.length>>1);

n = randomNumber(seq.length);

} while (m>=n)

var s1 = seq.slice(0,m);

var s2 = seq.slice(m,n)

var s3 = seq.slice(n,seq.length);

return s2.concat(s1).concat(s3).clone();

}

function setBestValue() {

for(var i=0; i<population.length; i++) {

values[i] = evaluate(population[i]);

}

currentBest = getCurrentBest();

if(bestValue === undefined || bestValue > currentBest.bestValue) {

best = population[currentBest.bestPosition].clone();

bestValue = currentBest.bestValue;

UNCHANGED\_GENS = 0;

} else {

UNCHANGED\_GENS += 1;

}

}

function getCurrentBest() {

var bestP = 0,

currentBestValue = values[0];

for(var i=1; i<population.length; i++) {

if(values[i] < currentBestValue) {

currentBestValue = values[i];

bestP = i;

}

}

return {

bestPosition : bestP

, bestValue : currentBestValue

}

}

function setRoulette() {

for(var i=0; i<values.length; i++) { fitnessValues[i] = 1.0/values[i]; }

var sum = 0;

for(var i=0; i<fitnessValues.length; i++) { sum += fitnessValues[i]; }

for(var i=0; i<roulette.length; i++) { roulette[i] = fitnessValues[i]/sum; }

for(var i=1; i<roulette.length; i++) { roulette[i] += roulette[i-1]; }

}

function wheelOut(rand) {

var i;

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

if( rand <= roulette[i] ) {

return i;

}

}

}

function randomIndivial(n) {

var a = [];

for(var i=0; i<n; i++) {

a.push(i);

}

return a.shuffle();

}

function evaluate(indivial) {

var sum = dis[indivial[0]][indivial[indivial.length - 1]];

for(var i=1; i<indivial.length; i++) {

sum += dis[indivial[i]][indivial[i-1]];

}

return sum;

}

function countDistances() {

var length = points.length;

dis = new Array(length);

for(var i=0; i<length; i++) {

dis[i] = new Array(length);

for(var j=0; j<length; j++) {

dis[i][j] = ~~distance(points[i], points[j]);

}

}

}

**main.js** :

var canvas, ctx; var WIDTH, HEIGHT; var points = []; var running; var canvasMinX, canvasMinY;

var doPreciseMutate; var POPULATION\_SIZE; var ELITE\_RATE; var CROSSOVER\_PROBABILITY;

var MUTATION\_PROBABILITY; var OX\_CROSSOVER\_RATE; var UNCHANGED\_GENS;

var mutationTimes; var dis;

var bestValue, best;

var currentGeneration; var currentBest;

var population; var values;

var fitnessValues; var roulette;

$(function() {

init();

initData();

points = data200;

$('#addRandom\_btn').click(function() {

addRandomPoints(50);

$('#status').text("");

running = false;

});

$('#start\_btn').click(function() {

if(points.length >= 3) {

initData();

GAInitialize();

running = true;

} else {

alert("add some more points to the map!");

}

});

$('#clear\_btn').click(function() {

running === false;

initData();

points = new Array();

});

$('#stop\_btn').click(function() {

if(running === false && currentGeneration !== 0){

if(best.length !== points.length) {

initData();

GAInitialize();

}

running = true;

} else {

running = false;

}

});

});

function init() {

ctx = $('#canvas')[0].getContext("2d");

WIDTH = $('#canvas').width();

HEIGHT = $('#canvas').height();

setInterval(draw, 10);

init\_mouse();

}

function init\_mouse() {

$("canvas").click(function(evt) {

if(!running) {

canvasMinX = $("#canvas").offset().left;

canvasMinY = $("#canvas").offset().top;

$('#status').text("");

x = evt.pageX - canvasMinX;

y = evt.pageY - canvasMinY;

points.push(new Point(x, y));

}

});

}

function initData() {

running = false;

POPULATION\_SIZE = 30;

ELITE\_RATE = 0.3;

CROSSOVER\_PROBABILITY = 0.9;

MUTATION\_PROBABILITY = 0.01;

//OX\_CROSSOVER\_RATE = 0.05;

UNCHANGED\_GENS = 0;

mutationTimes = 0;

doPreciseMutate = true;

bestValue = undefined;

best = [];

currentGeneration = 0;

currentBest;

population = []; //new Array(POPULATION\_SIZE);

values = new Array(POPULATION\_SIZE);

fitnessValues = new Array(POPULATION\_SIZE);

roulette = new Array(POPULATION\_SIZE);

}

function addRandomPoints(number) {

running = false;

for(var i = 0; i<number; i++) {

points.push(randomPoint());

}

}

function drawCircle(point) {

ctx.fillStyle = '#000';

ctx.beginPath();

ctx.arc(point.x, point.y, 3, 0, Math.PI\*2, true);

ctx.closePath();

ctx.fill();

}

function drawLines(array) {

ctx.strokeStyle = '#f00';

ctx.lineWidth = 1;

ctx.beginPath();

ctx.moveTo(points[array[0]].x, points[array[0]].y);

for(var i=1; i<array.length; i++) {

ctx.lineTo( points[array[i]].x, points[array[i]].y )

}

ctx.lineTo(points[array[0]].x, points[array[0]].y);

ctx.stroke();

ctx.closePath();

}

function draw() {

if(running) {

GANextGeneration();

$('#status').text("There are " + points.length + " cities in the map, "

+"the " + currentGeneration + "th generation with "

+ mutationTimes + " times of mutation. best value: "

+ ~~(bestValue));

} else {

$('#status').text("There are " + points.length + " Cities in the map. ")

}

clearCanvas();

if (points.length > 0) {

for(var i=0; i<points.length; i++) {

drawCircle(points[i]);

}

if(best.length === points.length) {

drawLines(best);

}

}

}

function clearCanvas() {

ctx.clearRect(0, 0, WIDTH, HEIGHT);

}

## Dynamic programming:

INPUT: A DISTANCE MATRIX FORMED BY GIVEN POINTS

OUTPUT:AN ARRAY WITH PREDECESSOR OF CITIES

import java.util.Scanner;

public class BellmanFord

{

private int distances[];

private int numberofvertices;

public static final int MAX\_VALUE = 999;

public BellmanFord(int numberofvertices)

{

this.numberofvertices = numberofvertices;

distances = new int[numberofvertices + 1];

}

public void BellmanFordEvaluation(int source, int adjacencymatrix[][])

{

for (int node = 1; node <= numberofvertices; node++)

{

distances[node] = MAX\_VALUE;

}

distances[source] = 0;

for (int node = 1; node <= numberofvertices - 1; node++)

{

for (int sourcenode = 1; sourcenode <= numberofvertices; sourcenode++)

{

for (int destinationnode = 1; destinationnode <= numberofvertices; destinationnode++)

{

if (adjacencymatrix[sourcenode][destinationnode] != MAX\_VALUE)

{

if (distances[destinationnode] > distances[sourcenode]

+ adjacencymatrix[sourcenode][destinationnode])

distances[destinationnode] = distances[sourcenode]

+ adjacencymatrix[sourcenode][destinationnode];

}

}

}

}

for (int sourcenode = 1; sourcenode <= numberofvertices; sourcenode++)

{

for (int destinationnode = 1; destinationnode <= numberofvertices; destinationnode++)

{

if (adjacencymatrix[sourcenode][destinationnode] != MAX\_VALUE)

{

if (distances[destinationnode] > distances[sourcenode]

+ adjacencymatrix[sourcenode][destinationnode])

System.out.println("The Graph contains negative egde cycle");

}

}

}

for (int vertex = 1; vertex <= numberofvertices; vertex++)

{

System.out.println("distance of source " + source + " to "

+ vertex + " is " + distances[vertex]);

}

}

public static void main(String... arg)

{

int numberofvertices = 0;

int source;

Scanner scanner = new Scanner(System.in);

System.out.println("Enter the number of vertices");

numberofvertices = scanner.nextInt();

int adjacencymatrix[][] = new int[numberofvertices + 1][numberofvertices + 1];

System.out.println("Enter the adjacency matrix");

for (int sourcenode = 1; sourcenode <= numberofvertices; sourcenode++)

{

for (int destinationnode = 1; destinationnode <= numberofvertices; destinationnode++)

{

adjacencymatrix[sourcenode][destinationnode] = scanner.nextInt();

if (sourcenode == destinationnode)

{

adjacencymatrix[sourcenode][destinationnode] = 0;

continue;

}

if (adjacencymatrix[sourcenode][destinationnode] == 0)

{

adjacencymatrix[sourcenode][destinationnode] = MAX\_VALUE;

}

}

}

System.out.println("Enter the source vertex");

source = scanner.nextInt();

BellmanFord bellmanford = new BellmanFord(numberofvertices);

bellmanford.BellmanFordEvaluation(source, adjacencymatrix);

scanner.close();

}

}