OPTIMIZATION OF TOUR BASED PROBLEM USING GENETIC ALGORITHM

REPORT OF PROJECT SUBMITTED FOR PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE OF

BACHELOR OF TECHNOLOGY
In
INFORMATION TECHNOLOGY

By

S.NO	TEAM MEMBERS	ROLL NO	Registration No.
1.	Prakash Dubey	IT/2014/044	141170110144
2.	Jatin Prakash	IT/2014/038	141170110133
3.	Umakant Piyush	IT/2014/026	141170110177

UNDER THE SUPERVISION OF

Dr. Dipankar Majumdar
Assistant Professor, Department of Information Technology
RCC Institute of Information Technology

AT

RCC INSTITUTE OF INFORMATION TECHNOLOGY [Affiliated to West Bengal University of Technology] CANAL SOUTH ROAD, BELIAGHATA, KOLKATA – 700 015

RCC INSTITUTE OF INFORMATION TECHNOLOGY

KOLKATA – 7OOO15, INDIA

CERTIFICATE

The report of the Project titled APPLICATION OF GENETIC ALGORITHM IN OPTIMIZATION OF TSP submitted by (Prakash Dubey Roll No.: IT/2014/044 of B. Tech. (IT) 8 th Semester of 2018, Jatin Prakash Roll No.: IT/2014/038 of B. Tech. (IT) 8 th Semester of 2018, Umakant Piyush Roll No.: IT/2014/026 of B. Tech. (IT) 8 th Semester of 2018) has been prepared under our supervision for the partial fulfillment of the requirements for B Tech (IT) degree in West Bengal University of Technology.

The report is hereby forwarded.

Counter signed by

_____

DR. Dipankar Majumdar
Assistant Professor
Dept. of Information Technology,
RCC Institute of Information Technology
KOLKATA-700015

Dr. Abhijit Das
HOD,Dept. of Information Technology
RCC Institute of Information Technology
KOLKATA-700015

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to Dr. Dipankar Majumdar of Department of INFORMATION TECHNOLOGY, RCCIIT and for extending their valuable times for me to take up this problem as a Project.We are extremely thankful for the keen interest he look in advising us for the reference materials provided for the moral support extended to us.

Last but not the least we would like to express our gratitude to all the teachers of our department who helped us in their own way whenever needed.

Date: 13th June, 2018

Prakash Dubey
Reg. No.: 141170110144 of 2014-15
Roll No.: IT/2014/044
Jatin Prakash
Reg. No.:141170110133 of 2014-15
Roll No.:IT/2014/038
Umakant Piyush
Reg. No.:141170110177 of 2014-15
Roll No.:IT/2014/026
B. Tech (IT) – 8 th Semester, 2016, RCCIIT

RCC INSTITUTE OF INFORMATION TECHNOLOGY

KOLKATA – 7OOO15, INDIA

CERTIFICATE of ACCEPTANCE

The report of the Project titled APPLICATION OF GENETIC ALGORITHM IN SOFTWARE TESTING submitted by(Prakash Dubey Roll No.: IT/2014/044 of B. Tech. (IT) 8 th Semester of 2018,Jatin Prakash Roll No.: IT/2014/038 of B. Tech. (IT) 8 th Semester of 2018,Umakant Piyush Roll No.: IT/2014/026 of B. Tech. (IT) 8 th Semester of 2018),
is hereby recommended to be accepted for the partial fulfillment of the requirements for B Tech (IT) degree in West Bengal. University of Technology

Name of the Examiner	**Signature with Date**
1……..	……..
2……..	……..
3……..	……..
4……..	……..

TABLE OF CONTENTS

Introduction

Optimization:The action of making the best or most effective use of a situation or resource.

In Optimization for tour based problem we have to find the shortest route from available routes.We have one famous problem on tour based that is Traveling Salesman Problem.

The Travelling Salesman problem (TSP) The Travelling Salesman Problem (TSP) is a classic combinatorial optimization problem, which is simple to state but very difficult t o solve. This problem is known to be NP-hard, and cannot be solved exactly in polynomial time. Many exact and heuristic algorithms have been developed in the field of operations research (OR) to solve this problem. The problem is to find the shortest possible tour through a set of n vertices so that each vertex is visited exactly once. The traveling salesman first gained fame in a book written by German salesman BF Voigt in 1832 on how to be a successful traveling salesman. He mentions the TSP, although not by that name, by suggesting that to cover as many locations as possible without visiting any location twice is the most important aspect of the scheduling of a tour. The origins of the TSP in mathematics are not really known -all we know for certain is that it happened around 1931. On the basis of the structure of the cost matrix, the TSPs are classified into two groups – symmetric and asymmetric. The TSP is symmetric if cij = cji, for all i, j and asymmetric otherwise. For an n-city asymmetric TSP, there are (n −)!1 possible solutions, one or more of which gives the minimum cost. For an n-city symmetric TSP, there are (n − 1)!/2 possible solutions along with their reverse cyclic permutations having the same total cost. In either case the number of solutions becomes extremely large for even moderately large n so that an exhaustive search is impracticable.

The problem was first formulated in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact algorithms are known, so that some instances with tens of thousands of cities can be solved completely and even problems with millions of cities can be approximated within a small fraction of 1%.The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept city represents, for example, customers, soldering points, or DNA fragments, and the concept distance represents travelling times or cost, or a similarity measure between DNA fragments. The TSP also appears in astronomy, as astronomers observing many sources will want to minimize the time spent moving the telescope between the sources. In many applications, additional constraints such as limited resources or time windows may be imposed.

Problem Definition

The travelling salesman problem consist of a salesman and a set of cities,the salesman has to visit each one of the cities starting from a certain one and returning to the same city.The challenging of the problem is that the traveling salesman wants to minimize the total length of the city.

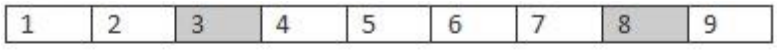
Traveling salesman problem-adjacency representation and path representation.
We consider the path representation for a tour ,which simply lists the level of nodes.

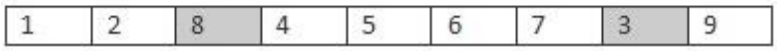
We can solve traveling salesman problem using ***Genetic algorithm***

To ensure the genetic algorithm does indeed meet this requirement special types of mutation and crossover methods are needed.

Firstly, the mutation method should only be capable of shuffling the route, it shouldn't ever add or remove a location from the route, otherwise it would risk creating an invalid solution. One type of mutation method we could use is swap mutation.

With swap mutation two location in the route are selected at random then their positions are simply swapped. For example, if we apply swap mutation to the following list, [1,2,3,4,5] we might end up with, [1,2,5,4,3]. Here, positions 3 and 5 were switched creating a new list with exactly the same values, just a different order. Because swap mutation is only swapping pre-existing values, it will never create a list which has missing or duplicate values when compared to the original, and that's exactly what we want for the traveling salesman problem.



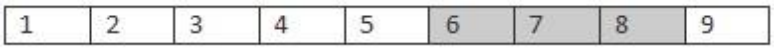


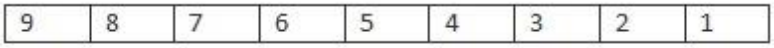
Now we've dealt with the mutation method we need to pick a crossover method which can enforce the same constraint.

One crossover method that's able to produce a valid route is ordered crossover. In this crossover method we select a subset from the first parent, and then add that subset to the offspring. Any missing values are then adding to the offspring from the second parent in order that they are found.

To make this explanation a little clearer consider the following example:

Parents





Offspring

Here a subset of the route is taken from the first parent (6,7,8) and added to the offspring's route. Next, the missing route locations are adding in order from the second parent. The first location in the second parent's route is 9 which isn't in the offspring's route so it's added in the first available position. The next position in the parents route is 8 which is in the offspring's route so it's skipped. This process continues until the offspring has no remaining empty values. If implemented correctly the end result should be a route which contains all of the positions it's parents did with no positions missing or duplicated.

Literature Study

Despite an intensive study by mathematicians, computer scientists, operations researchers, and others, it remains an open question whether or not an efficient general solution exists for the travelling salesman problem . Hence, it is considered as a benchmark problem, and various methods are being applied to it, so as to get better solutions than those already known. Solutions to the travelling salesman problem can be approached using combinatorial optimization techniques. There are two types of solution search methods: First is to use exact algorithms, which can give optimal solution but takes a huge amount of time. Second is to use approximate algorithms, which never guarantee an optimal solution but gives near optimal solution in a reasonable amount of computational time. Few of these methods are described below: The most direct solution would be to use the brute force approach and try all permutations and check which solution is optimal. The computational complexity of this approach is of the order O(N!) for 'N' cities. Because of the ever increasing number of the possible solutions and the combinatorial nature of the travelling salesman problem, it is impractical to use this approach to solve the problems even with high performance computers.

George Dantzig, Ray Fulkerson and Selmer Johnson illustrated the power of cutting - plane method for solving the travelling salesman problem in 1954. This method was introduced when no algorithms were available to solve integer linear programs. They solved an instance with 49 cities, an impressive size at that time. Problem-specific methods are needed to find the cuts used by this method. More discussion of cutting-plane algorithms . In 1960s, the branch-andbound method was proposed by A. H. Land and A. G. Doig for discrete programming. It controls the searching process through an effective restrictive boundary so that it can search for the optimal solution branch from the space state tree instantly. The key point of this algorithm is the choice of the restrictive boundary. Different restrictive boundaries may form different branch-and-bound algorithms. In the late 1970s and 1980, Grtschel, Padberg, Rinaldi and others managed to apply the algorithm to travelling salesman problem and exactly solve instances with up to 2392 cities, using cutting planes and branch-and-bound method. However, the branch-and-bound algorithm alone is not feasible enough for solving the large-scale problem.

Michael Held and Richard Karp developed the Held-Karp bound which provides a near-optimal lower bound on the cost of solutions to the travelling salesman problem. For a travelling salesman problem with 'N' cities, their method guarantees that it is proportional to N22 N. For any large value of 'N' the Held-Karp guarantee is much less than (N-1)!. Held-Karp bound is actually a solution to the linear programming relaxation of the integer formulation of the travelling salesman problem. A Held-Karp lower bound averages about 0.8% below the optimal tour length. More details about the Held-Karp bound technique can be found in, and a good survey on this technique is given by Gerhard Woeginger.

Algorithm

A genetic algorithm (GA) is great for finding solutions to complex search problems. They're often used in fields such as engineering to create incredibly high quality products thanks to their ability to search a through a huge combination of parameters to find the best match. For example, they can search through different combinations of materials and designs to find the perfect combination of both which could result in a stronger, lighter and overall, better final product. They can also be used to design computer algorithms, to schedule tasks, and to solve other optimization problems. Genetic algorithms are based on the process of evolution by natural selection which has been observed in nature. They essentially replicate the way in which life uses evolution to find solutions to real world problems. Surprisingly although genetic algorithms can be used to find solutions to incredibly complicated problems, they are themselves pretty simple to use and understand.

How they work

As we now know they're based on the process of natural selection, this means they take the fundamental properties of natural selection and apply them to whatever problem it is we're trying to solve.

The basic process for a genetic algorithm is:

1. Initialization - Create an initial population. This population is usually randomly generated and can be any desired size, from only a few individuals to thousands.
2. Evaluation - Each member of the population is then evaluated and we calculate a 'fitness' for that individual. The fitness value is calculated by how well it fits with our desired requirements. These requirements could be simple, 'faster algorithms are better', or more complex, 'stronger materials are better but they shouldn't be too heavy'.
3. Selection - We want to be constantly improving our populations overall fitness. Selection helps us to do this by discarding the bad designs and only keeping the best individuals in the population. There are a few different selection methods but the basic idea is the same, make it more likely that fitter individuals will be selected for our next generation.
4. Crossover - During crossover we create new individuals by combining aspects of our selected individuals. We can think of this as mimicking how sex works in

nature. The hope is that by combining certain traits from two or more individuals we will create an even 'fitter' offspring which will inherit the best traits from each of it's parents.

5. Mutation - We need to add a little bit randomness into our populations' genetics otherwise every combination of solutions we can create would be in our initial population. Mutation typically works by making very small changes at random to an individuals genome.
6. And repeat! - Now we have our next generation we can start again from step two until we reach a termination condition.

Termination

There are a few reasons why you would want to terminate your genetic algorithm from continuing it's search for a solution. The most likely reason is that your algorithm has found a solution which is good enough and meets a predefined minimum criteria. Offer reasons for terminating could be constraints such as time or money.

Problem Discussion

Genetic Algorithm is an optimization technique. It is an evolutionary algorithm which generate solution to problem inspired by natural evolution, In this, randomly population are selected and that inherit to new population called offstring. Fitness of population is evaluated to generate offstring. After evaluation of fitness value ,perform mutation and crossover to generate offstring. This process perform in intration, until optimal solution is not found. It Consists Three Operation 2.1.1.Selection Selection in which analyze chromosome or population by fitness function to generate offstring whether they are survive or not and reproduce in nature. There are various selection methods- 1.Random Selection 2.Roulette Wheel Selection 3.Tournament Selection 2.1.2.Crossover It is technique to generate new chromosome, define pair of chromosome and swapping in sequence of bits and generate new population. For instance[3], Parent 1: aAbBcC Parent 2: 123456 Child: aAb45 (one possibility out of many) 2.1.3. Mutation Mutaion is performed in bits and mutate every bit of chromosome every bit of chromosome to generate offstring. For instance, Parent 1: aAbBcC Parent 2: 123456 Child: aAbZ56 (changing 4 to Z).

Implementation

Steps Of Algorithms

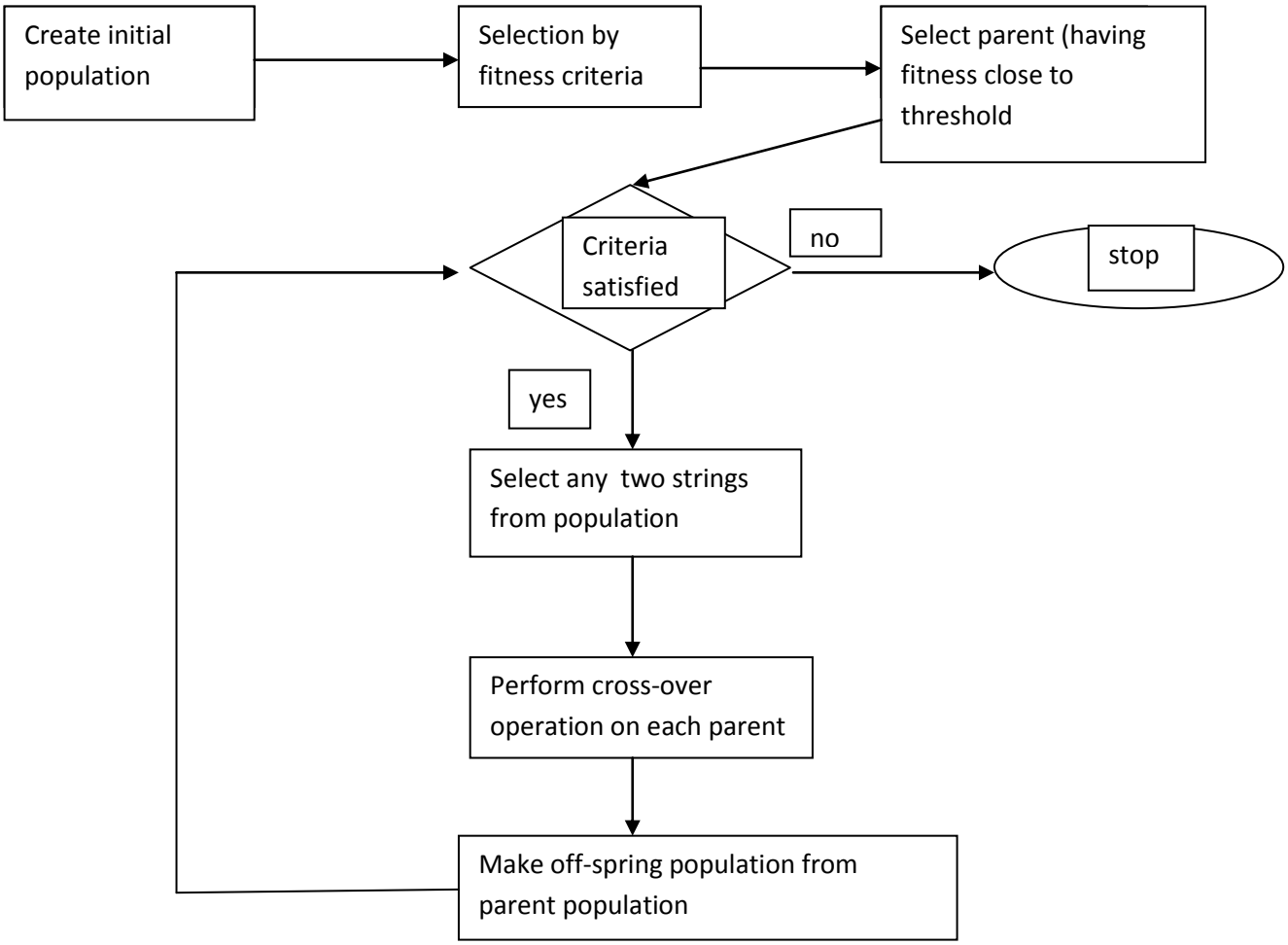
1.Randomly create the initial population of individual string of the given TSP problem and create a matrix representation of the cost of the path between two cities.

2.Assign the fitness to each chromosome in the population using fitness criteria measure. F(x) = 1/x where, x represents the total cost of the string. The selection criteria depends upon the value of string if it is close to some threshold value.

3. Create new offspring population from two existing chromosomes in the parent population by applying crossover operator.

4.Mutate the resultant off-springs if required. NOTE: After the crossover off spring population has the fitness value higher than the parents.

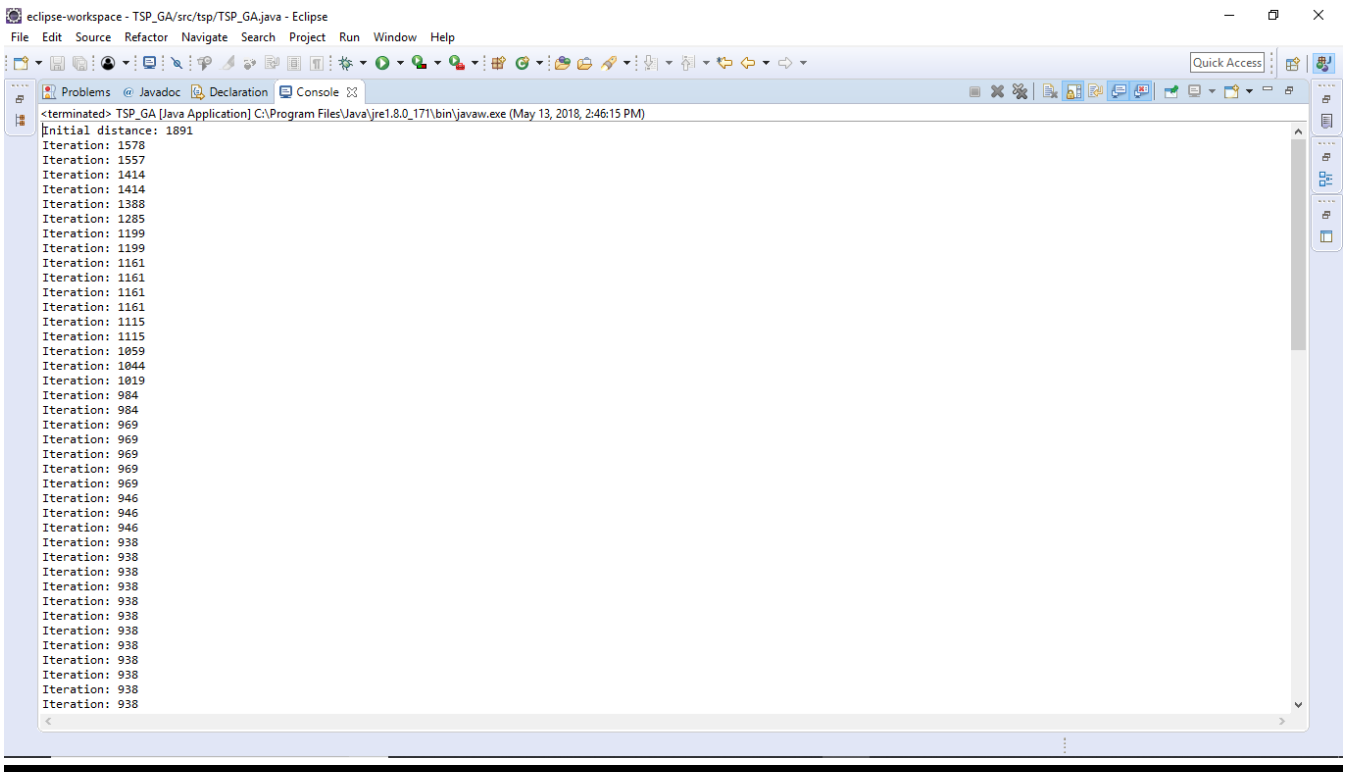
5.Repeat step 3 and 4 until we get an optimal solution to the problem.

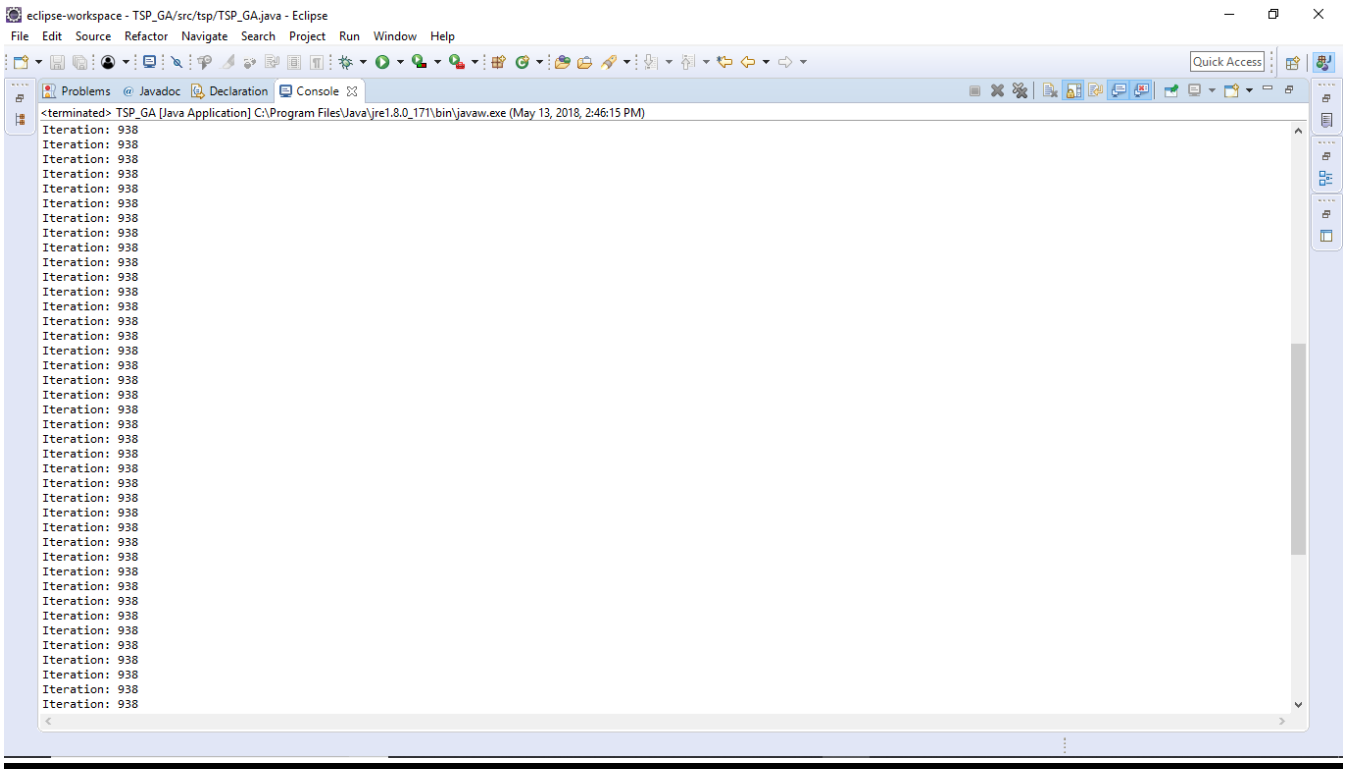


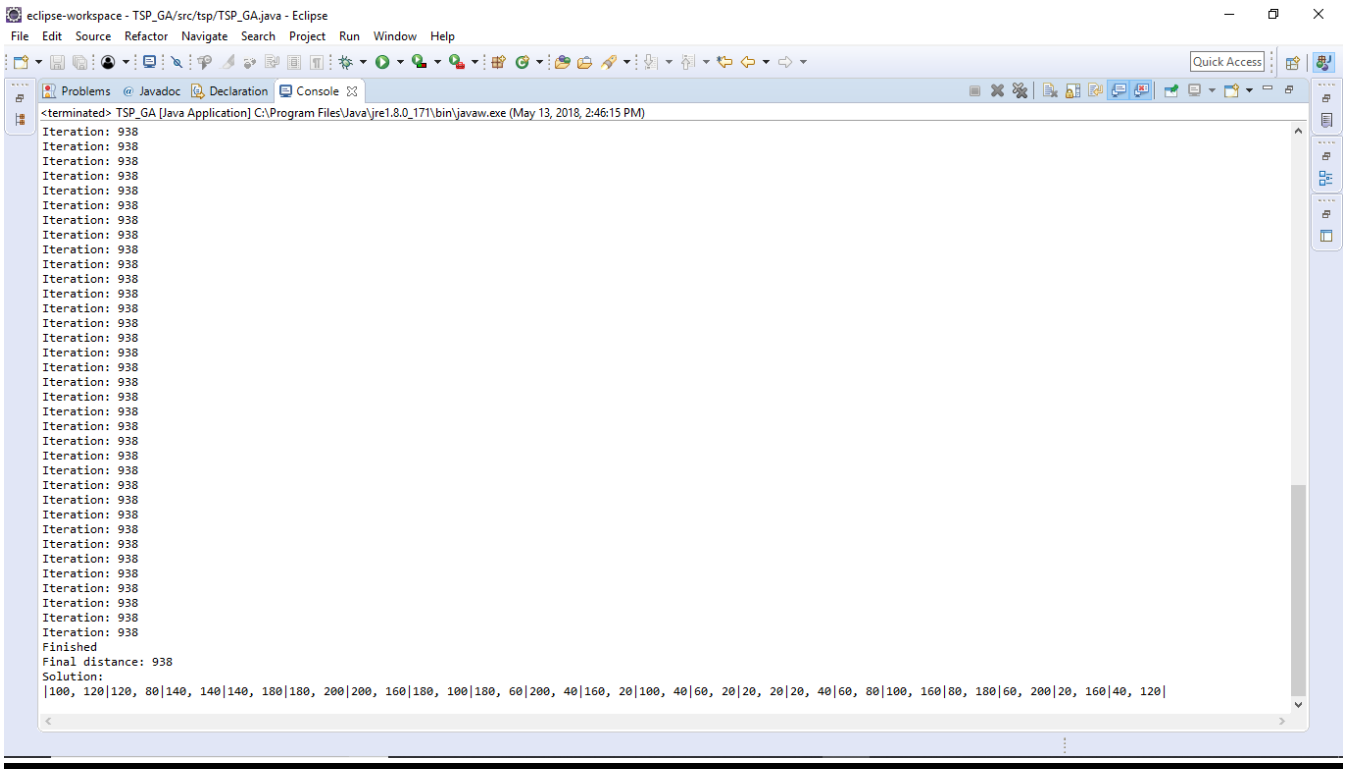
for any optimization problem, one has to think a way for encoding solutions as feasible chromosomes so that the crossovers of feasible chromosomes result in feasible chromosomes. The techniques for encoding solutions vary by problem and, involve a certain amount of art. For the TSP, solution is typically represented by chromosome of length as the number of nodes in the problem. Each gene of a chromosome takes a label of node such that no node can appear twice in the same chromosome. There are mainly two representation methods for representing tour of the TSP – adjacency representation and path representation. W e consider the

path representation for a tour, which simply lists the label of nodes. For example, let {1, 2, 3, 4, 5} be the labels of nodes in a 5 node instance, then a tour {1 3 4 2 5 1} may be represented as (1, 3, 4, 2, 5) Fitness function The GAs are used for maximization problem. For the maximization problem the fitness function is same as the objective function. But, for minimization problem, one way of defining a „fitness function" F (x) = 1/ f (x) where f (x) is the objective function. Since, TSP is a minimization problem; we consider this fitness function, where f(x) calculates cost (or value) of the tour represented by a chromosome. Selection Process: In selection process, chromosomes are copied into next generation with a probability associated with their fitness value. By assigning to next generation a higher portion of the highly fit chromosomes, reproduction mimics the Darwinian survival-of-the-fittest in the natural world. In this paper we are using Elitism method for selection. Elitism is name of method, which first copies the best chromosome (or a few best chromosomes) to new population. The rest is done in classical way. Elitism can very rapidly increase performance of GA, because it prevents losing the best found solution

Output:




eclipse-workspace - TSP_GA/src/tsp/TSP_GA.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Problems Javadoc Declaration Console
<terminated> TSP_GA [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (May 13, 2018, 2:46:15 PM)
Iteration: 938


eclipse-workspace - TSP_GA/src/tsp/TSP_GA.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Problems Javadoc Declaration Console
<terminated> TSP_GA [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (May 13, 2018, 2:46:15 PM)
Iteration: 938
Finished
Final distance: 938
Solution:
|100, 120|120, 80|140, 140|140, 180|180, 200|200, 160|180, 100|180, 60|200, 40|160, 20|100, 40|60, 20|20, 20|20, 40|60, 80|100, 160|80, 180|60, 200|20, 160|40, 120|

Conclusion/Future Scope Of Work

Genetic algorithms are often used for optimization problems in which the evolution of a population is a search for a satisfactory solution given a set of constraints. We have reported preliminary results from an experiment comparing random test data generation with a new approach using genetic search. In this paper, we have demonstrated that it is possible to apply Genetic Algorithm techniques for finding the optimal paths for Travelling salesman problem. The Genetic Algorithms also outperforms the exhaustive search and local search techniques.

Genetic algorithm appear to find good solutions for the Travelling Salesman Problem,however it depends very much on the way the problem is encoded and which crossover and mutation methods are used .We have proposed a new crossover operator named for a genetic algorithm for the Traveling Salesman Problem (TSP). . Among all the operators, experimental results show that our proposed crossover operator (SCX) is better in terms of quality of solutions and cost as well as solution times. That is why, we used a local search technique to improve the solution quality. Also, we set here highest probability of crossover to show the exact nature of crossover operator. Mutation with lowest probability is applied wherever required only. We presented a comparative study among Greedy approach, Dynamic programming and Genetic Algorithm for solving TSP.It is very difficult to say that what moderate sized instance is unsolvable exactly by our crossover operator. So an incorporation of good local search technique to the algorithm may solve exactly the other instances, which is under our investigation and is categorized under future work .

Appendix(program code)

To start, let's create a class that can encode the cities.

City.java

```java
/*
* City.java
* Models a city
*/

package tsp;

public class City {
    int x;
    int y;

    // Constructs a randomly placed city
    public City(){
        this.x = (int)(Math.random()*200);
        this.y = (int)(Math.random()*200);
    }

    // Constructs a city at chosen x, y location
    public City(int x, int y){
        this.x = x;
        this.y = y;
    }

    // Gets city's x coordinate
    public int getX(){
        return this.x;
    }

    // Gets city's y coordinate
    public int getY(){
        return this.y;
    }

    // Gets the distance to given city
    public double distanceTo(City city){
        int xDistance = Math.abs(getX() - city.getX());
        int yDistance = Math.abs(getY() - city.getY());
        double distance = Math.sqrt( (xDistance*xDistance) + (yDistance*yDistance) );

        return distance;
    }

    @Override
    public String toString(){
        return getX()+", "+getY();
    }
}
```

Now we can create a class that holds all of our destination cities for our tour

TourManager.java

```
/*
* TourManager.java
* Holds the cities of a tour
*/

package tsp;

import java.util.ArrayList;

public class TourManager {

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

    // Adds a destination city
    public static void addCity(City 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();
    }
}
```

Next we need a class that can encode our routes, these are generally referred to as tours so we'll stick to the convention.

Tour.java

```
/*
* Tour.java
* Stores a candidate tour
*/

package tsp;

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 = 1/(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;
    }
}
```

We also need to create a class that can hold a population of candidate tours

Population.java

```
/*
* Population.java
* Manages a population of candidate tours
*/

package tsp;

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 = tours[0];
        // Loop through individuals to find fittest
        for (int i = 1; i < populationSize(); i++) {
            if (fittest.getFitness() <= getTour(i).getFitness()) {
                fittest = getTour(i);
            }
        }
        return fittest;
    }

    // Gets population size
    public int populationSize() {
        return tours.length;
    }
}
```

Next, the let's create a GA class which will handle the working of the genetic algorithm and evolve our population of solutions.

GA.java

```
/*
* GA.java
* Manages algorithms for evolving population
*/

package tsp;

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) {
            newPopulation.saveTour(0, pop.getFittest());
            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 fittest = tournament.getFittest();
        return fittest;
    }
}
```

Now we can create our main method, add our cities and evolve a route for our travelling salesman problem.

TSP_GA.java

```
/*
* TSP_GA.java
* Create a tour and evolve a solution
*/

package tsp;

public class TSP_GA {

    public static void main(String[] args) {
```

```
        // Create and add our cities
        City city = new City(60, 200);
        TourManager.addCity(city);
        City city2 = new City(180, 200);
        TourManager.addCity(city2);
        City city3 = new City(80, 180);
        TourManager.addCity(city3);
        City city4 = new City(140, 180);
        TourManager.addCity(city4);
        City city5 = new City(20, 160);
        TourManager.addCity(city5);
        City city6 = new City(100, 160);
        TourManager.addCity(city6);
        City city7 = new City(200, 160);
        TourManager.addCity(city7);
        City city8 = new City(140, 140);
        TourManager.addCity(city8);
        City city9 = new City(40, 120);
        TourManager.addCity(city9);
        City city10 = new City(100, 120);
        TourManager.addCity(city10);
        City city11 = new City(180, 100);
        TourManager.addCity(city11);
        City city12 = new City(60, 80);
        TourManager.addCity(city12);
        City city13 = new City(120, 80);
        TourManager.addCity(city13);
        City city14 = new City(180, 60);
        TourManager.addCity(city14);
        City city15 = new City(20, 40);
        TourManager.addCity(city15);
        City city16 = new City(100, 40);
        TourManager.addCity(city16);
        City city17 = new City(200, 40);
        TourManager.addCity(city17);
        City city18 = new City(20, 20);
        TourManager.addCity(city18);
        City city19 = new City(60, 20);
        TourManager.addCity(city19);
        City city20 = new City(160, 20);
        TourManager.addCity(city20);

        // Initialize population
        Population pop = new Population(50, true);
        System.out.println("Initial distance: " + pop.getFittest().getDistance());

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

        // Print final results
        System.out.println("Finished");
        System.out.println("Final distance: " + pop.getFittest().getDistance());
        System.out.println("Solution:");
        System.out.println(pop.getFittest());
    }
}
```

Reference

1. Melanie Mitchell,Santa Fe Institute,1399 Hyde Park Road,Santa Fe, NM 87501 email: mm@santafe.edu
2. Babamir, F.S., Babamir, S.M.: A GA based-method to Generate Test Data of Program Paths. In: 15th National Computer Conference CSICC, Tehran, IRAN (2010)
3. Babamir, S.M., Babamir, F.S.: A Genetic-based Algorithm to Optimum Generation of Data for Program Paths Testing. In: 14th National Computer Conference CSICC, Tehran, IRAN (2009)
4. Goldberg, D.E.: Genetic Algorithm in a Search Optimization and Machine Learning. Addison Wesley, Reading (1989)