Chapter1. INTRODUCTION

1.1 Background

The problem of TSP is to find a tour of a given no of cities. A salesman have to visit each and every city exactly once. Salesman have to either ends to a given city or to return back to the starting city such that the length of the tour is minimized.

The problem was first introduced by Euler in 1759 whose problem was to move a knight on a chess board on every positions exactly once.

The travelling salesman problem was first written in a book by German salesman BF Voigt in 1832 on how be a successful salesman . He explains the TSP by not by the proper name TSP but define the problem of a salesman. The most important constraint was that the salesman have to visit each and every city with no repetition. The origin of TSP is not yet certain but it happened in nearly around 1931.

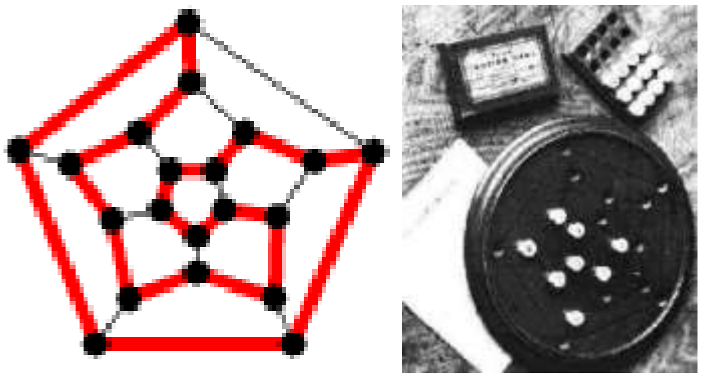


Figure - 1 Pegboard with twenty holes

1.2 Problem statement:

Mathematically TSP can be stated as follows:

Given a weighted graph G = (V,E)

Where the weight Cij on the edge between cities I and j is a non negative value. Find the tour that visit all nodes exactly once with minimum cost.

The problem can be solved by hit and trial method i.e. enumerating each possible solution to traverse all the cities. Each possible solution is a permutation of

1234 ………..n, where n is the no of cities. So the no of possible tours are n!.

When n is large it become impossible to find the path with minimum path length in polynomial time. Many other ways have been suggested to solve TSP which we will discuss in next chapter.

1.3 Solution

1.3.1 Genetic Algorithms

Genetic algorithms (GAs) are search techniques based on principles of natural selection and genetics – a concept taken from medical science (Fraser, 1957; Bremermann, 1958; Holland, 1975). We start with a brief introduction of genetic algorithms and terminology.

GAs encode the decision variables of a search problem into finite-length strings. These strings are candidate solutions to the search problem and are referred to as chromosomes, these alphabets are referred to as genes and the values of these genes are called alleles. For example, in a problem TSP the travelling salesman problem, a chromosome represents a route, and a gene may represent a city in the route followed . In contrast to traditional optimization techniques, GAs work with encoding of parameters, rather than the parameters themselves.

Figure – 2 World TSP consisting of 1,904,711 cities all over the world

Fitness of a chromosome - To evolve good solutions and to implement selection, we need a measure for distinguishing good solutions from bad . The measure could be an objective function that is a mathematical designed model or a computer simulation, or it can be a subjective function where humans selects better solutions over worse ones. THIS fitness measure must determine a candidate solution's fitness, which will subsequently be used by the genetic algorithm to guide the evolution of good solutions.

Another important concept of GAs is the representation of population. Unlike traditional search methods of finding solutions , genetic algorithms are based upon a population of candidate solutions. The population size (no of chromosomes in the population), which is usually a user-specified parameter, is one of the important factors which affect the performance of genetic
algorithms.

For example, small population sizes may lead to premature convergence and may find substandard solutions. On the other hand, large population sizes lead to expenditure of valuable computational efforts.

Once the problem is encoded in a chromosomal form and a fitness function for discriminating good solutions from bad ones has been selected, we can start to evolve solutions to the search problem by using the following steps:

1 Initialization. The initial population of candidate solutions (chromosomes) is usually generated randomly from the search space. However, domain-specific knowledge or other information can be easily incorporated in finding the initial population.

2 Evaluation. In this step the fitness values of the candidate solutions are evaluated by using the fitness function.

3 Selection. Selection allocates more copies of those solutions in to mating pool with higher fitness values and thus imposes the survival-of-the-fittest mechanism. The main idea of selection is to prefer better solutions, and many selection procedures have been proposed by different researchers to accomplish this idea, including roulette-wheel selection, stochastic universal selection, rank based selection and tournament based selection.

4 Recombination. In this step parents have been selected and recombined to generate children. There are many ways of doing this (some of which are discussed in the next chapter in literature review), and competent performance depends on a properly designed recombination mechanism.

5 Mutation. While recombination operates on two or more chromosomes, it locally but randomly modifies a solution. Again, there are many different variations of mutation, but it usually involves one or more changes being made. In other words, mutation performs a random walk in the domain of a candidate solution.

6 Replacement. The new population created by selection, recombination, and mutation replaces the original chromosomes in the parental population. Some replacement techniques like elitist replacement, steady-state replacement methods and generation-wise replacement are used in GAs.

7 Repeat steps 2–6 until a terminating criteria is met.

Procedure GA

```
begin
    Generate N random chromosomes {N is the population size}
    Evaluate tour length produced by each path and store each one
    store best-path-so-far
    repeat
        for each chromosome of the population
            Select two parents using any of the selection methods
            apply crossover operator to produce new offspring
            apply mutation to offspring of the population
            evaluate tour length produced by offspring in the current population
            if offspring is better than weaker parent then it replaces it in population
            if offspring is better than best-path-so-far then it replaces best-path-so-
        far
        endfor
    until stopping criteria satisfied
    print best-path-so-far
end
```

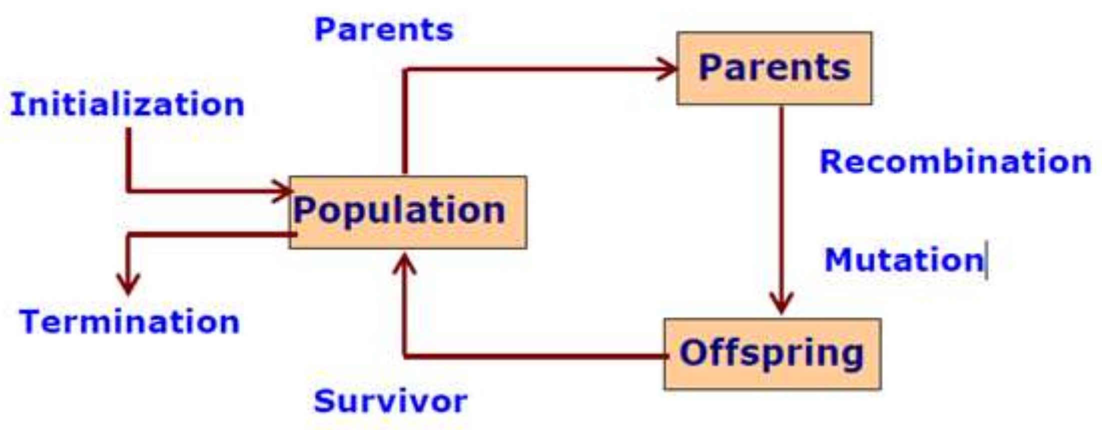


Figure - 1

1.3.2 Basic Genetic Algorithm Operators

Selection Methods. Selection procedures can be mainly classified into two classes as follows.

Fitness Proportionate Selection This includes methods like roulette-wheel selection and stochastic universal selection. In roulette-wheel selection, each chromosome in the population is assigned a roulette wheel slot sized in proportion to its fitness. That is, in the biased roulette wheel, good solutions have a larger slot size than the less fit solutions. The roulette wheel is spun to obtain a reproduction candidate.

Ordinal Selection This includes methods such as tournament selection (Goldberg et al., 1989b), and truncation selection (M¨uhlenbein and Schlierkamp-Voosen, 1993). In tournament selection, s chromosomes are chosen at random (either with or without replacement) and entered into a tournament against each other. The fittest individual in the group of k chromosomes wins the tournament and is selected as the parent. The most widely used value of s is 2. Using this selection scheme, n tournaments are required to choose n individuals. In truncation selection, the top (1/s)th of the individuals get s copies each in the mating pool.

Recombination (Crossover) Operators. After selection, individuals from the mating pool are recombined (or crossed over) to create new, hopefully better, offspring. In the GA literature, many crossover methods have been designed (Goldberg, 1989b; Booker et al., 1997; Spears, 1997) and some of them are described in this section. Many of the recombination operators used in the literature are problem-specific and in this section we will introduce a few generic (problem independent) crossover operators. It should be noted that while for hard search problems, many of the following operators are not scalable, they are very useful as a first option.

Replacement. Once the new offspring solutions are created using crossover and mutation, we need to introduce them into the parental population. There are many ways we can approach this. Bear in mind that the parent chromosomes have already been selected according to their fitness, so we are hoping that the children (which includes parents which did not undergo crossover) are among the fittest in the population and so we would hope that the population will gradually, on average, increase its fitness. Some of the most common replacement techniques are outlined below.

Delete-all This technique deletes all the members of the current population and replaces them with the same number of chromosomes that have just been created. This is probably the most common technique and will be the technique of choice for most people due to its relative ease of implementation. It is also parameter-free, which is not the case for some other methods.

Steady-state This technique deletes n old members and replaces them with n new members. The number to delete and replace, n, at any one time is a parameter to this deletion technique. Another consideration for this technique is deciding which members to delete from the current population. Do you delete the worst individuals, pick them at random or delete the chromosomes that you used as parents? Again, this is a parameter to this technique.

Steady-state-no-duplicates This is the same as the steady-state technique but the algorithm checks that no duplicate chromosomes are added to the population. This adds to the computational overhead but can mean that more of the search space is explored.

HELP! WE'RE LOST!
HELP "CAR 54"... AND WIN CASH
54...$1,000 PRIZES
ONE...$10,000 GRAND PRIZE
START FINISH
Help Toody and Muldoon find the shortest round trip route to visit all 33 locations shown on the map.
All you do is draw connecting straight lines from location to location to show the shortest round trip route.
HERE'S THE CORRECT START...
Begin at Chicago, Illinois. From there, lines show correct route as far as Erie, Pennsylvania. Next, do you go to Carlisle, Pennsylvania or Wana, West Virginia? Check the easy instructions on back of this entry blank for details.
© PROCTER & GAMBLE 1962
OFFICIAL RULES ON REVERSE SIDE

Figure – 3 Tsp for visiting major cities of United States

1.4 Thesis outline:-

This dissertation encounters in different part as on starting it deals with slight introduction of what I am going to do. The rest comes in the following sections:-

Section II: -

This section discuss the literature review and result showing different techniques to solve the TSP using GA. In these papers some work has been done to solve the TSP using GA. The main concern of this literature is to provide a study of the role of the cross over operator in the GA process while solving the TSP.

Section III: -

Proposed model will come over here and a new cross over operator has been suggested for solving TSP using GA. The experimental setup and the classes used in the implementation are discussed here in detail.

Section IV: -

This section explain the comparison of results of the proposed method with an existing method.

Section V: -

This section explains the conclusion and the future work .

Section VI: - In this section introduction of tools are given which has been used in this work.

Chapter2. LITERATURE REVIEW

In this thesis an attempt is made to improve the performance of the cross over operator in solving TSP using GA. Different In this chapter we study genes that are ordered lists of objects without repetition. These are useful for classical optimization problems like the famous travelling salesman problem as well as for evolving controls on a greedy algorithm. There is a good deal of mathematical structure that applied to ordered genes, viewed as permutations. A useful subset of this lore appears at the end of the chapter in Section 4.3. We will begin by looking at possible variation operators that operate directly on an ordered list. In later sections we will look at alternate representations.

2.1 Cycle crossover

he cycle crossover produces children that obey the rule “the object in position i of the list is equal to the object at position i from one of the two parents. Since all structure is derived from parental structure, this operator is “sexual”. It is always possible to do crossover of this sort, but since the number of free choices of which parent to take an element from depends on the structure of the permutation, the number of possible children varies wildly.

Algorithm : The CX Operator

```
Input: two ordered lists the same length (parents)
Output: two ordered lists the same length (children)
Details:
        Initialize two empty children.
        While there are empty locations in the children
        At the first empty location, fill that location with values at that
        location from the parents, with order selected randomly.
        While selections are forced
        Make forced selections, filling child locations
End While
End While
```

The forced selections are the key to understanding the CX operator. Examine the following example:

Example CX Example

In this example we build only the first child, the other child simply has all its values at each position equal to those in a parent but not in the first child.

Parents
8 2 6 3 7 1 4 5
6 3 2 8 7 5 1 4
Steps to generate the child path from the parents

```
* * * * * * * * Initial child is empty
6 * * * * * * * Pick from second parent
6 * 2 * * * * * 6 is used, position 3 must be 2
6 3 2 * * * * * 2 is used, position 2 must be 3
6 3 2 8 * * * * 3 is used, position 4 must be 8
Out of forced moves
6 3 2 8 7 * * * Pick from second parent
No forced moves
6 3 2 8 7 1 * * Pick from first parent
6 3 2 8 7 1 4 * 1 is used, position 7 must be 4
6 3 2 8 7 1 4 5 4 is used, position 8 must be 5
Done
8 2 6 3 7 5 1 4 Second child, choices not made before
```

2.2 Ordered crossover

The ordered crossover (OX1) operator is one of the simplest. It preserves the order of some of the objects in the the order they appear in each parent. This operator is sexual in the sense that information from both parents survives into both children.

```
Algorithm  The OX1 Operator
    Input: two ordered lists the same length (parents)
    Output: two ordered lists the same length (children)
    Details:
    Align the two parent lists and pick a position p
    Copy the locations before p from each parent to one of the children
    After p copy into the child the remaining loci in the order they
    appear in the other parent.
```

Return children

OX1 Example

Parents

8 7 6 3 2 1 4 5
6 4 2 8 3 5 1 7

8 7 6 3 * 2 1 4 5
6 4 2 8 * 3 5 1 7

Children

8 7 6 3 4 2 5 1
6 4 2 8 7 3 1 5

The OX2 Operator

The ordered based crossover (OX2) operator is also relatively simple. Like OX1, it preserves the order of some of the objects in the order they appear in each parent. This operator is sexual in the sense that information from both parents survives into both children. It accomplishes this in a somewhat different manner.

```
Algorithm OX2 Operator
Input: two ordered lists the same length (parents)
Output: two ordered lists the same length (children)
Details:
        Align the two parent lists and pick a number of positions p
        Copy locations before an even number of markers from parent to corresponding child.
        Copy locations before an odd number of locations as well, but place
        them in the order they appear in the other parent.
Return children
```

OX2 Example

Parents

8 7 6 3 2 1 4 5
3 4 2 8 6 5 1 7

Parents with position chosen

8 7 * 6 3 * 2 1 * 4 5
3 4 * 2 8 * 6 5 * 1 7

The effect of the crossover is to place the numbers in locations 3, 4, 7, and 8 in the order they appear in the other parent.

Children

8 7 3 6 2 1 4 5
3 4 8 2 6 5 7 1

2.3 VARIATION OPERATORS

2.3.1Partially matched crossover

The partially matched crossover (PMX) operator for ordered genes is not really a crossover operator. Crossover blends genetic material from each parent in the child or children. Instead of doing this, the PMX operator uses a comparison of a part of the parents genome to generate a block of mutations. The PMX operator is a binary variation operator and turns out to be valuable. The sense in which it is not a crossover operator is that in which it fails to resemble any sort of sexual recombination in nature.

Algorithm

```
Input: two ordered lists the same length (parents)

Output: two ordered lists the same length (children)

Details:

    Copy both parents to make the initial version of the children

    Align the two parent lists and pick two positions, a and b

    For all positions p between a and b in the parents

        Take the list items at position p in the children as swap them

    End For

Return children
```

PMX Example

Parents

8 2 6 3 7 1 4 5

6 3 2 8 7 5 1 4

Parents with positions chosen

8 2 6 * 3 7 1 * 4 5

6 3 2 * 8 7 5 * 1 4

So the children are the parents with the following swaps 3 ↔ 8, 7 ↔ 7, and 1 ↔ 5. Swapping 7 with itself, of course,changes nothing so we get:

Children

3 2 6 8 7 5 4 1

6 8 2 3 7 1 5 4

2.4 Comparing some of the crossovers

A 50 city problem with an expected solution of 100 was generated using pseudo random numbers in the range 0 - R, where R is the area of required square region calculated using the method of BHH and Stein's constant . Because, in retrospect, it is obvious that a 50 city problem is far too small to extrapolate anything significant regarding the performance of genetic algorithms and operators to large problems, only a brief summary of the work is included.

Results for a 50 city problem.

Table – 1

Mutation rate %	**Crossover rate %**	**Mean Best Tour of 100000 trials for Ten Replicate runs**		
		CX	**PMX**	**OX**
100	00	317.27	317.27	317.27
90	10	122.46	173.82	130.16
80	20	106.60	106.60	114.82
70	30	108.74	159.37	105.28
60	40	107.92	159.37	103.92
50	50	108.03	160.34	106.66
40	60	110.73	164.41	107.14
30	70	109.44	166.26	107.01

20	80	110.59	150.11	108.37
10	90	109.27	110.72	109.48
0	100	326.15	225.13	144.61

Chapter3. PROPOSED METHODOLOGY FOR COMMON SUBROUTE CROSSOVER

3.1 Common Sub Route Crossover Process :-

This cross over operator generate new child by taking in to consideration of common sub routes in two paths. The key idea is that cities which are close to each other must be visited one after the other thus making sub routes in the TSP problem. If these sub routes are modified by any of the GA operator then it may increase the overall cost of the route and thus generate the child which are less fit. By not disturbing the sequence of the cities in the sub routes we have tried to generate child with high fitness values. Example explaining Common Sub routes crossover is as follows:

Let No of cities = 15

Path p1 : 1-13-12-8-9-10-11-4-5-3-6-7-2-14-15

Path p2 : 1-6-8-9-10-11-13-14-2-12-4-3-5-7-15

It can be seen that in both the parents a common sub route exist which 8-9-10-11 is. The child generated after CSC cross over must have this sub route. The generated child will be

Child c1 : 1-8-9-10-11-2-13-4-5-3-7-2-14-15

ALGORITHM:-

Algorithm: CSC()

// This algorithm take two paths as parents and generate a new path (child-path)

Step 1 : Place last node of the parent on to child-path and go to step 5.

Step 2 : Find all common sub routes in both of the parents. If such common sub route(s) exists then go to step 3 otherwise copy parent 1 in to child-path and exit.

Step 3 : For each common sub-route found repeat following steps

3.1 Take a common sub route

3.2 Place all the cities of the common sub-route in the end part of the child path.

3.3 If it is the last common sub route then go to step 4 otherwise go to step 3.1

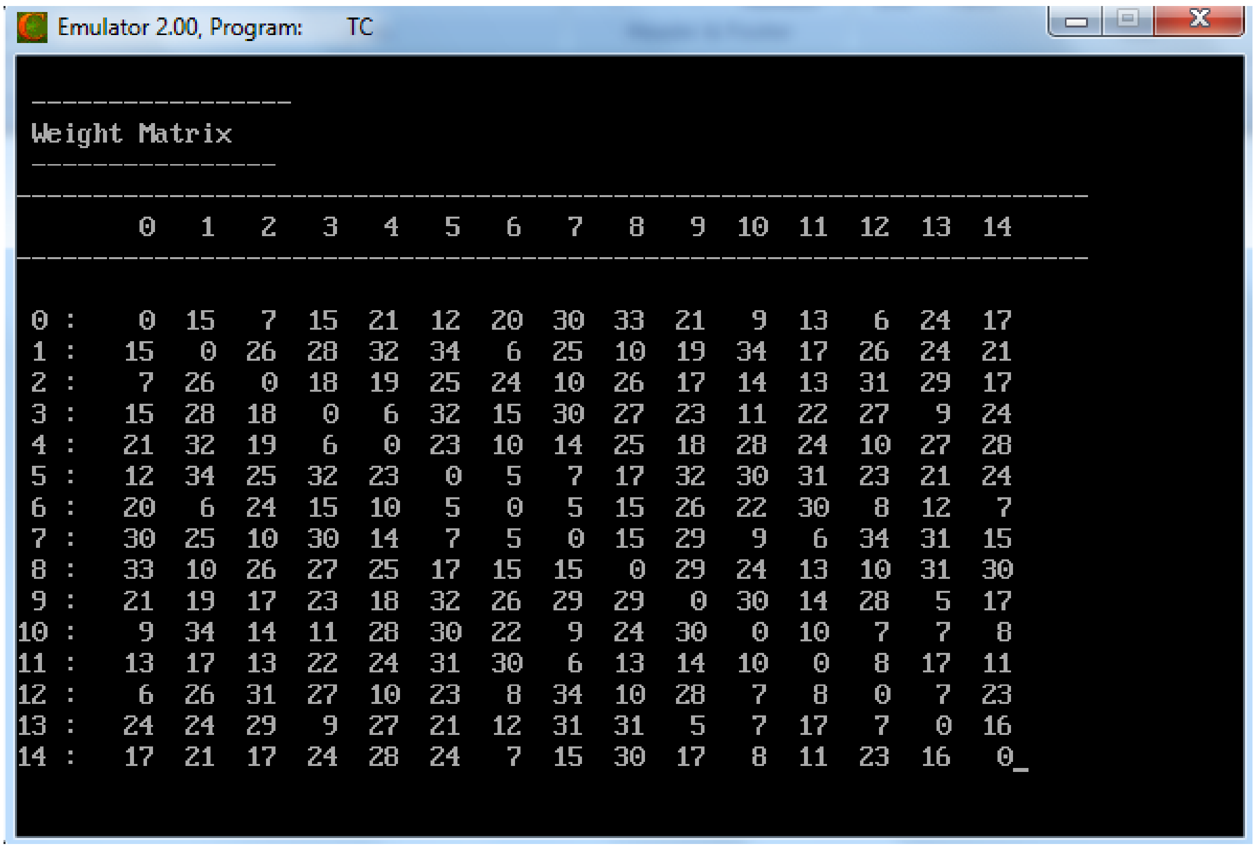
Step 4 : If child-path is not completed then copy remaining part of the child-path from parent-1.

Step 5 : Return chid-path.

3.2 Experimental study:-

3.2.1 EXPERIMENTAL SETUP AND SNAPSHOTS

The algorithm CSC crossover has been implemented in C++ and results have been analyzed. A weight matrix has been created randomly which stores the distance between all the cities with each other. Figure- 3.1 shows the weight matrix generated randomly. Rows and columns of the matrix indicate the cities from city-1 to city-15. Row i stores the distance for city (i+1).



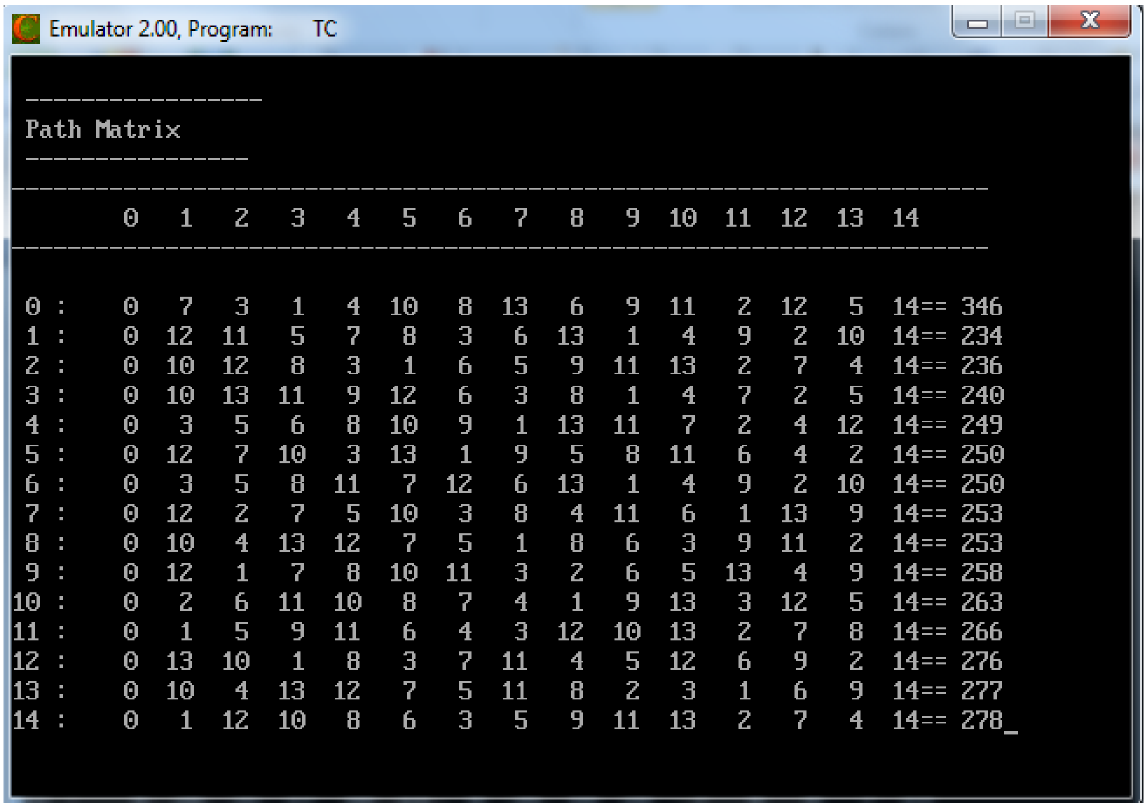
```
 -----------------
 Weight Matrix
 ----------------
----------------------------------------------------------------------
        0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
----------------------------------------------------------------------


 0 :    0  15   7  15  21  12  20  30  33  21   9  13   6  24  17
 1 :   15   0  26  28  32  34   6  25  10  19  34  17  26  24  21
 2 :    7  26   0  18  19  25  24  10  26  17  14  13  31  29  17
 3 :   15  28  18   0   6  32  15  30  27  23  11  22  27   9  24
 4 :   21  32  19   6   0  23  10  14  25  18  28  24  10  27  28
 5 :   12  34  25  32  23   0   5   7  17  32  30  31  23  21  24
 6 :   20   6  24  15  10   5   0   5  15  26  22  30   8  12   7
 7 :   30  25  10  30  14   7   5   0  15  29   9   6  34  31  15
 8 :   33  10  26  27  25  17  15  15   0  29  24  13  10  31  30
 9 :   21  19  17  23  18  32  26  29  29   0  30  14  28   5  17
10 :    9  34  14  11  28  30  22   9  24  30   0  10   7   7   8
11 :   13  17  13  22  24  31  30   6  13  14  10   0   8  17  11
12 :    6  26  31  27  10  23   8  34  10  28   7   8   0   7  23
13 :   24  24  29   9  27  21  12  31  31   5   7  17   7   0  16
14 :   17  21  17  24  28  24   7  15  30  17   8  11  23  16   0_
```

Figure-3.1 Weight Matrix for TSP (Randomly Generated)

A random initial population of 15 path has been generated and cost of these random paths has been calculated.

<u>Assumption</u> : In this implementation it is assumed that cities can be visited in any order. Further route can be started from any city and then visiting all the cities it can be terminated at any city. Figure-3.2 shows the initial population of 15 paths. It also shows the cost of these paths.



```
 -----------------
 Path Matrix
 ----------------
----------------------------------------------------------------------
        0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
----------------------------------------------------------------------

 0 :    0   7   3   1   4  10   8  13   6   9  11   2  12   5  14== 346
 1 :    0  12  11   5   7   8   3   6  13   1   4   9   2  10  14== 234
 2 :    0  10  12   8   3   1   6   5   9  11  13   2   7   4  14== 236
 3 :    0  10  13  11   9  12   6   3   8   1   4   7   2   5  14== 240
 4 :    0   3   5   6   8  10   9   1  13  11   7   2   4  12  14== 249
 5 :    0  12   7  10   3  13   1   9   5   8  11   6   4   2  14== 250
 6 :    0   3   5   8  11   7  12   6  13   1   4   9   2  10  14== 250
 7 :    0  12   2   7   5  10   3   8   4  11   6   1  13   9  14== 253
 8 :    0  10   4  13  12   7   5   1   8   6   3   9  11   2  14== 253
 9 :    0  12   1   7   8  10  11   3   2   6   5  13   4   9  14== 258
10 :    0   2   6  11  10   8   7   4   1   9  13   3  12   5  14== 263
11 :    0   1   5   9  11   6   4   3  12  10  13   2   7   8  14== 266
12 :    0  13  10   1   8   3   7  11   4   5  12   6   9   2  14== 276
13 :    0  10   4  13  12   7   5  11   8   2   3   1   6   9  14== 277
14 :    0   1  12  10   8   6   3   5   9  11  13   2   7   4  14== 278_
```

Figure-3.2 Initial population.

Figure-3.3 shows the CSC cross over. The cross over rate is 40% so 6 out of 15 parents has been selected randomly to participate in meeting process. So path no 6,4,2,0,1 and 11 has been selected to perform crossover. The CSC cross over process generate 3 new children.

Let us discuss the CSC cross over process of path 2 and 0.

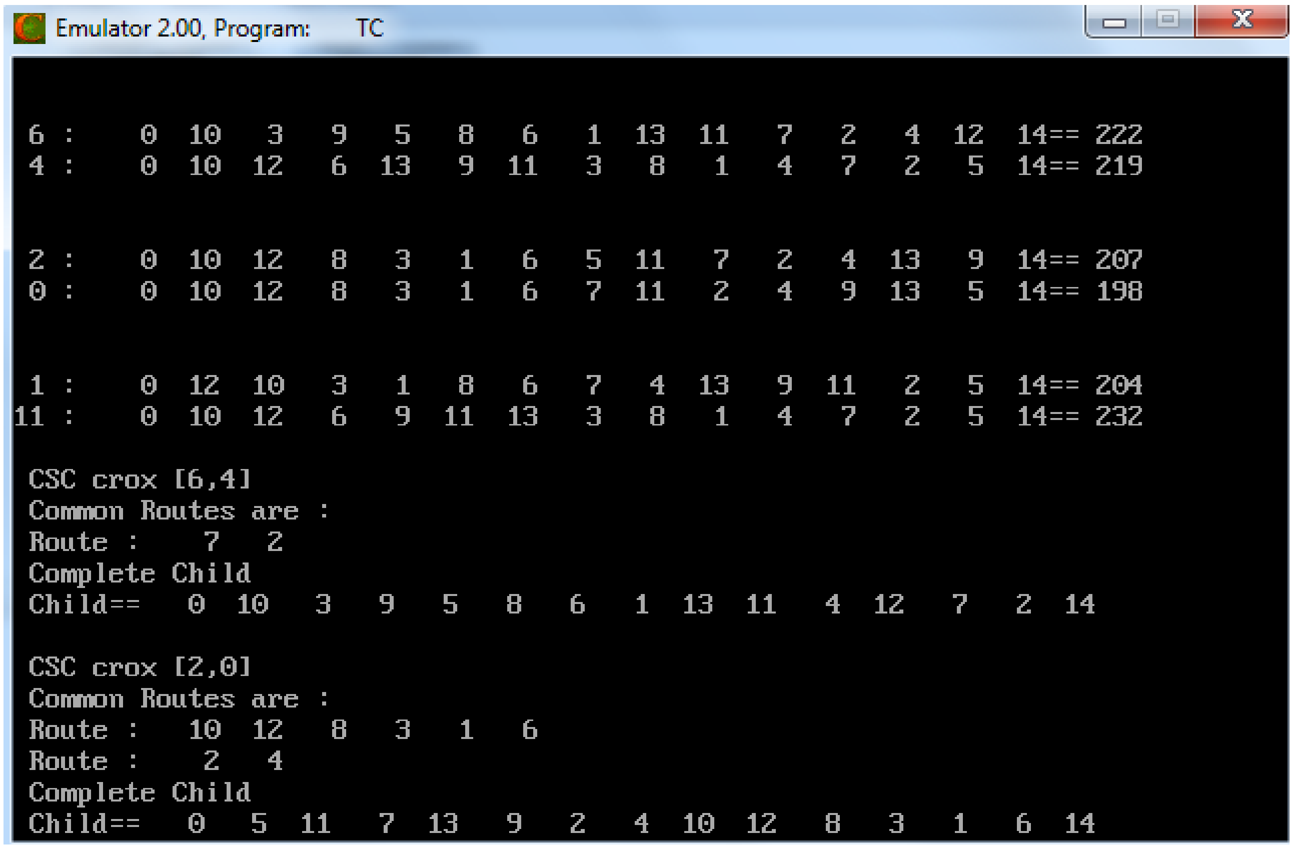
Path 2 : 0-10-12-8-3-1-6-5-11-7-2-4-13-9-14

Path 0 : 0-10-12-8-3-1-6-7-11-2-4-9-13-5-14

It can be observed that these two path contain a two common sub routes having cities 10-12-8-3-1-6 and 2-4 in same order in both the routes. To generate the child this information is used. So first of all this common sub route has been added in the last in the children. After that all cities has been added in the start in the same order in which these cities occur in path 2 i.e. parent one. Hence the process will generate a child which is :

Child: 0-5-11-7-13-9-2-4-10-12-8-3-1-6-14

In the same way other two children are also generated from paths 6,4,1 and 11.



Emulator 2.00, Program: TC

```
 6 :    0  10   3   9   5   8   6   1  13  11   7   2   4  12  14== 222
 4 :    0  10  12   6  13   9  11   3   8   1   4   7   2   5  14== 219


 2 :    0  10  12   8   3   1   6   5  11   7   2   4  13   9  14== 207
 0 :    0  10  12   8   3   1   6   7  11   2   4   9  13   5  14== 198


 1 :    0  12  10   3   1   8   6   7   4  13   9  11   2   5  14== 204
11 :    0  10  12   6   9  11  13   3   8   1   4   7   2   5  14== 232

 CSC crox [6,4]
 Common Routes are :
 Route :    7   2
 Complete Child
 Child==   0  10   3   9   5   8   6   1  13  11   4  12   7   2  14

 CSC crox [2,0]
 Common Routes are :
 Route :   10  12   8   3   1   6
 Route :    2   4
 Complete Child
 Child==   0   5  11   7  13   9   2   4  10  12   8   3   1   6  14
```

Figure-3.3 CSC cross over process to generate new children

3.3 CLASSES USED IN THE IMPLEMENTATION AND THEIR DETAILS

We have used following four classes to implement the proposed work :

1. Weight
2. Random Paths
3. Meeting
4. CSC

The details of the classes used in the implementation are as follows.

1. Class Weight

This class is used to generate weight matrix that will store the distance between all the cities with each other.Members of this class are as follows :

Table 3.1 Details of class Weight

Class Name : Weight		Purpose
Data Members	intwt[21][21]	This matrix will store the distance between cities in a two dimensional array.
Member Functions	Weight()	It is the constructor method and call the method which actually creates the weight matrix at the time of object creation
	void call_methods()	This method call the generate_weight() method.
	void generate_weight()	This method randomly set the distance between the cities in the 2D array.
	void display_column_no()	This method is displaying the column no or the cities no in the output screen.

	void display_weight()	This method display display the 2D array weight matrix on the output screen .

2. ClassRandom_Paths

This class is used to generate path matrix. It will generate a random path matrix of 15 paths and store it in a 2D array path[][].Members of this class are as follows :

Table 3.2 Details of class Random_Paths

Class Name : **Random_Paths**		Purpose
Data Members	Weight w	It is an object of Weight class and stores the 2D array weight matrix used to find the distance between the cities.
	int path[21][21];	It is a 2D array which stores the paths/chromosomes/candidate solutions for solving TSP using Genetic Algorithms.
	intpath_length[21];	This array stores the path distance of individual paths of the path matrix.
Member Functions	void generate_paths()	This method will generate 15 paths randomly
	void display_paths()	This method display the paths on the output screen.
	void multiply_and_display_paths()	This method calculate the path lengths of different paths and display paths with their path lengths.

3. Class Meeting

This class is used to perform classical one point cross over. It will perform selection, classical cross over (OX)[ ] and mutation to find a solution for the TSP. Members of this class are as follows :

Table 3.3 Details of class Meeting

Class Name : **Meeting**		Purpose
Data Members	Random_Pathsrp;	It is an object of class Random_Paths and stores the path matrix and weight matrix.
	float cross_over_rate;	This variable stores the cross over rate i.e. how many paths will participate in the cross over operation.
	intcount_cross_over_paths;	This variable stores how many paths are participating in the cross over process.
	intcross_over_positions[15];	This array stores the path no of the paths in path matrix array which have been selected randomly to participate in cross over operation.
	int point;	It stores the position of cross over point in classical one point crossover.
	intnew_paths[10][15];	This array stores the newly generated paths after cross over.
	intmin_path_length;	This variable stores the minimum path length among all the paths in the current population.
	intposition_of_best_path;	This variable stores the position of the best path (in 2D array) i.e. the path with minimum path length in the current population.
	inttotal_paths_in_this_population;	This variable stores total no of paths in the current population after adding then new distinct children.
Member Functions	Meeting()	It is a constructor function and it sets the value of cross over rate, cross over position and count of paths which will participate in cross over operation.
	void call_methods_2()	This method calls other methods in a sequence to complete the one iteration of

		cross over process.
	void multiply_and_display_paths()	This method find the path lengths of different paths using the weight matrix and then display paths with their path lengths in the output.
	void display_meeting_data()	This method display the cross over rate and cross over position.
	void find_meeting_positions()	This method randomly select some paths which will participate in the cross over operation.
	void print_original_paths()	This method print the original paths which are participating in cross over operation.
	void display_newly_generated_paths_by_meeting()	This method display the newly generated children after cross over operation.
	void do_meeting()	This method actually perform the classical one point cross over operation and generate new children.
	intcheck_new_path_in_already_paths_to_restrict_duplicates(intchild_path_no)	This method check whether or not the child generated is already there in the population. If yes then that child will not be added in the population so that duplicate path does not exist in the population.
	void append_new_paths_and_multiply()	This method add the new children in the previous population.
	void sort_paths()	This method sort all the paths (old + new children) by their path lengths and selects the paths which have minimum path length for the next generation. So the population becomes better and better after every iteration.

4. Class CSC

This class is used to perform common sub-routes cross over. The methods and functions of this class are as follows:

Table 3.4 Details of class CSC

Class Name : **CSC**		Purpose
Data Members	Meeting m	
	intcsc_path[20][21];	This matrix stores the paths to perform CSC cross over. This array will be initialized after performing 20 iterations of classical one point cross over.
	intcsc_path_length[20];	This array stores the path lengths of the paths.
	intcsc_child[6][15];	This array stores thenewly generated paths after cross over operation.
	intcsc_single_child[15];	This 1D array stores the one child to assist in children generation process.
	intcsc_child_count;	This variable stores the no of children generated by CSC process.
	intdistinct_child_count	Out of three how many children are distinct i.e. not already present in the population.
	intdistinct_child_poistions[10]	This 1D array stores the positions of the distinct children in among the new children.
	intcsc_child_paths_length[6]	This array stores the path lengths of the children generated by CSC.
	intcsc_child_good_bad_status[6]	This array is a flag array and stores 0 if child is really distinct and not present in the population otherwise store 1.
	intappended_population_count;	This variable stores how many children have been appended in the population after CSC process.
Member	void remove_invalid_path()	This method checks the paths in the

Functions		population and delete a path if it is not a valid path in any way.
	void call_methods()	This method call the other methods in a sequence to perform CSC.
	void csc_sort_paths()	sort all 18 to select best 15 paths
	void display_appended_csc_population()	This method display the appended population i.e. the original population plus the children added to the population.
	void append_new_csc_paths()	This method adds the new children in the population.
	void find_childs_path_length()	This method calculates the path lengths of the new children using weight matrix.
	void find_new_distinct_childs()	This method finds that whether a child is a new distinct child or a duplicate child.
	void init_child_to_zero()	This method initialize the one path to all it'spositions = - and then common sub-routes in the this path to generate a new child.
	void display_child()	This method display a new child generated.
	void do_csc_cross_over()	This method actually perform the CSC cross over and generate new children.
	void diplay_new_csc_childs()	This method display newly generated children after CSC process.
	void generate_csc_path()	This method find the path length of the new children and display these children.
	void copy_weight_matrix_and_csc_path_matrix_to_file()	This method copy the current population and weight matrix to a file so that these files can be used for further analyses.
	void	This method add the new children in to

	copy_csc_paths_in_to_random_paths()	current population.
	void display_csc_paths()	This method displays the paths in the current population.
	void find_csc_paths()	This method calls the sort method and select the best paths to generate the next population of 15 paths.

Chapter4. RESULT ANALYSIS

Given TSP problem has been implemented using classical one point crossover and for proposed CSC crossover. Experimental results show that the proposed CSC crossover is better as compared to classical one point crossover. The same TSP problem is implemented and CSC crossover will found a path having less distance to traverse all cities as compared to classical one point crossover. Following table shows a comparative study in terms of number of iterations and the path length of the best path and the worst path in the population in that iteration. The program has been run for 500 iterations and results has been analyzed after every 100 iterations.

Table- 4.1 Comparative results for 500 iterations

No of Iterations	**Classical Cross Over**		**CSC Cross Over**	
No of Iterations Repeated	**Path Length of Smallest Path (Best Path)**	**Path Length of Largest Path**	**Path Length of Smallest Path**	**Path Length of Largest Path**
10	191	241	195	229
20	184	225	191	217
30	166	215	191	211
40	166	210	191	208
50	166	200	184	204
100	166	198	173	191
200	166	198	168	181
300	166	198	154	180
400	166	198	150	175

500	166	198	150	175

The program has been run for 500 iterations and results has been analyzed after every 100 iterations.

It is justified from the results of Table - 4.1 that new CSC crossover will found an optimal solution with path length of 154 in only 300 iterations whereas classical one point crossover is unable to find it even after 500 iterations or more. So our new CSC crossover is better than the classical one point crossover.

Chapter5. PUBLICATION FROM THESIS

Chapter6. CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this paper we have proposed a new crossover operator named common sub route cross over (CSC) for a genetic algorithm for the Travelling Salesman Problem (TSP). It is concluded from the experimental results that this new cross over improve the solution in terms of no of iterations required as compared to classical one point or two point cross over. Further there is a future scope to solve some well known TSP problems such as bayg29, kroA100 etc and compare the performance of the algorithm.

6.2 Future directions

In this work I have tried to propose a new cross over and it has been justified that it's results are better. Further the proposed method have to be applied on some standard TSP problems such as bayg29, kroA100 etc. In future this work can also be tested for different values of rate of cross over. Different parent selection techniques such as Rank Selection, Roulette Wheel Selection, Tournament Selection can also be applied and results can be tested.

Chapter7. TOOLS TO BE USED

Compiler Used :	Turbo C++ 3.0
Language Used :	C++
Operating System :	Windows XP

APPENDIX

APPENDIX-1 SOURCE CODE OF THE IMPLEMENTATION

```
// Solving TSP by using CSC Cross-over operator in Genetic Algorithms
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
class Weight
{

        public:
        int wt[21][21];
        Weight()
        {
                clrscr();
                call_methods();
        }
        void call_methods()
        {
                 generate_weight();
                // display_weight();
        }
        void generate_weight()
        {
                int i,j,k,num;
                for(i=0;i<=14;i++)
                {
                        for(j=0;j<=14;j++)
                        {
                                num = (rand()%30)+5;
                                if(i<j)
                                {
                                        wt[i][j] = num;
                                }
                                if(i==j)
                                {
                                        wt[i][j]=0;
                                }
                                if(i>j)
                                {
                                        wt[i][j] = wt[j][i];
                                }
                        }
                }

        }// end generate_weight()
        void display_column_no()
        {

                int i;
                printf("\n");
                for(i=1;i<=70;i++)
                {
                        printf("-");
                }
                printf("\n     ");
                for(i=0;i<=14;i++)
                {
                        printf("%4d",i);
                }
                printf("\n");
                for(i=1;i<=70;i++)
                {
                        printf("-");
                }
                printf("\n");
        }
        void display_weight()
        {
                  printf("\n -----------------");
                  printf("\n Weight Matrix");
                  printf("\n ----------------");

                int i,j,k,num;
                display_column_no();
                for(i=0;i<=14;i++)
                {
                        printf("\n%2d : ",i);
                        for(j=0;j<=14;j++)
                        {
                                printf("%4d",wt[i][j]);
                        }
                }
        }// end display_weight
};
class Random_Paths
{
        public:
        Weight w;// Containership of class Weight
        int path[21][21];
        int path_length[21];
        Random_Paths()
        {

                call_methods();
        }
        void call_methods()
        {
                generate_paths();
//              multiply_and_display_paths();
                /*
                  printf("\n -------------");
                  printf("\n Path Matrix");
                  printf("\n -------------");
                */

          //            display_paths();

        }
        void generate_paths() // this method will generate 15 paths randomly
        {
                int i,j,k,num,pos,temp;
                // generate 1 to 15 cities in each path in continue manner
                for(i=0;i<=14;i++)
                {

                        for(j=0;j<=14;j++)
                        {
                                        path[i][j] = j;
                        }
                }
                // Flush to generate diff paths
                for(i=0;i<=14;i++)
                {
                        //pos = i;
                        //temp = path[i][pos];

                        //path[i][pos] = path[i][0];
                        //path[i][0] = temp;
                        for(j=0;j<=50;j++)// flush positions 50 times
                        {
                                                        int pos1 = (random(100)%13)+1;
                                        int pos2 =  (random(100)%13)+1;
                                        temp = path[i][pos1];
                                        path[i][pos1] = path[i][pos2];
                                        path[i][pos2] = temp;
                        }

                }

        }// end method

        void display_paths()
        {
                  printf("\n -----------------");
                  printf("\n Path Matrix");
                  printf("\n ----------------");

                int i,j;
                w.display_column_no();
                for(i=0;i<=14;i++)
                {
                        printf("\n%2d : ",i);
                        for(j=0;j<=14;j++)
                        {
                                printf("%4d",path[i][j]);
                        }
                        //if( (i%4) == 0)
                        //
                        ////getch();
                }
        }
        void multiply_and_display_paths()
        {
                  printf("\n -----------------");
                  printf("\n Path Matrix");
                  printf("\n ----------------");

                int i,j,node1,node2,length,total=0;
                ////// Initialise path lenths to 0
                for(i=0;i<=14;i++)
```

```
                {
                        path_length[i] =0;
                }
                /////////////////////////////////
                for(i=0;i<=14;i++)
                {
                        total=0;
                        for(j=1;j<=14;j++)
                        {
                                node1 = path[i][j-1];
                                node2 = path[i][j];
                                length =w.wt[node1][node2];
                                /*
                                //printf("\n Node 1 = %d",node1);
                                //printf("\n Node 2 = %d",node2);
                                if(i==4)
                                printf("\t Length = %d",length);
                                */
                                total+=length;
                        }
                        //printf("\n i = %d total  = %d",i,total);
                        path_length[i] = total;
                }// end for i
                //////// Displaying Paths With Lengths ///////////////
                w.display_column_no();
                for(i=0;i<=14;i++)
                {
                        printf("\n%2d : ",i);
                        for(j=0;j<=14;j++)
                        {
                                printf("%4d",path[i][j]);
                        }
                        printf("==%4d",path_length[i]);
                }
                ////////////////////////////////////
        }

};
class Meeting
{
        public:
        Random_Paths rp;
        float cross_over_rate;
        int count_cross_over_paths;
        int cross_over_positions[15];
        int point;
        int new_paths[10][15];
        int path_length[21];
        int min_path_length;
        int position_of_best_path;
        Meeting()
        {

                cross_over_rate = 0.40;
                count_cross_over_paths = 15*cross_over_rate;
                point = 7; // fix meeting point
                //call_methods();
        }
```

```
void call_methods_2()
{
            clrscr();
            //printf("\n ----------- CSC Weight Matrix --------------");
            rp.w.display_weight();
  //        gertch();clrscr();
       //   printf("\n ----------- CSC Path Matrix --------------");
              rp.multiply_and_display_paths();
   //         //getch();clrscr();
              printf("Press Enter To See Meeting ....");
              clrscr();
              printf("\n -----------------");
              printf("\n Meeting Started");
              printf("\n ----------------");
              display_meeting_data();
              find_meeting_positions();
              printf("\n -----------------");
              printf("\n Original Paths");
              printf("\n ----------------");
              print_original_paths();
              do_meeting();
              printf("\n -----------------");
              printf("\n New Paths");
              printf("\n ----------------");
              //display_newly_generated_paths_by_meeting();

              //// //getch();
              clrscr();
              printf("\n Appended total 21 paths");
              printf("\n ----------------");

              append_new_paths_and_multiply();
              //// //getch();
              clrscr();
              printf("\n Sorted total 21 paths");
              printf("\n ----------------");
              // sort_paths();
}

void multiply_and_display_paths()
{

            int i,j,node1,node2,length,total=0;
            int min=10000,max=0;
            float avg;
            int total_all_paths=0;
            ////// Initialise path lenths to 0
            for(i=0;i<=14;i++)
            {
                        path_length[i] =0;
            }
            /////////////////////////////////
            for(i=0;i<=14;i++)
            {
                        total=0;
                        for(j=1;j<=14;j++)
                        {
                                    node1 = rp.path[i][j-1];
```

```
				node2 = rp.path[i][j];
				length = rp.w.wt[node1][node2];
				/*
				//printf("\n Node 1 = %d",node1);
				//printf("\n Node 2 = %d",node2);
				if(i==4)
				printf("\t Length = %d",length);
				*/
				total+=length;
			}
			//printf("\n i = %d total  = %d",i,total);
			path_length[i] = total;
			total_all_paths+=total;
			if(total < min)
			{
				min = total;
				min_path_length = min;
				position_of_best_path=i;
			}
			if(total > max)
			{
				max = total;
			}
		}// end for i
		//////// Displaying Paths With Lengths ///////////////
		rp.w.display_column_no();
		for(i=0;i<=14;i++)
		{
			printf("\n%2d : ",i);
			for(j=0;j<=14;j++)
			{
				printf("%4d",rp.path[i][j]);
			}
			// come back to node 1
			printf("==%4d",path_length[i]);
		}
		avg = (float)total_all_paths/15;
		printf("\n\n Min = %d max = %d",min,max);
		printf("\n\n Avg Path Length = %4.2f",avg);
		//printf("\n Position = %d",position_of_best_path);
		///////////////////////////////////
}

void display_meeting_data()
{
		cross_over_rate=0.40;
		printf("\n\n Cross Over rate = %2.2f%",cross_over_rate);
		count_cross_over_paths=6;
		printf("\n\n No of paths crossed over =%d",count_cross_over_paths);
		printf("\n\n Meeting Point = %d",point);

}
void find_meeting_positions()
{
		int i,j,num;
		int n=count_cross_over_paths;
		int repeat;
		for(i=0;i<count_cross_over_paths;)// Select 6 Distinct paths
```

```
			{
				num = rand()%15;
				repeat=0;
				//check for repeated number generated
				if(i>0)//Not the first number
				{
					for(j=0;j<=i-1;j++)
					{
						if(num == cross_over_positions[j])
						{
					 //			printf("\n Repeated No Found = %d",num);
					//			// //getch();
							repeat=1;
							break;
						}
					}
				}
				if(repeat==0)
				{
					//printf("\n New No Found = %d",num);
					cross_over_positions[i] = num;
					i++;
				}
				//printf("\n Total Positions Selected for CSC = %d",i+1);
				////getch();
			 //	// //getch();
			}
			printf("\n Meeting Paths Are : ");
			for(i=0;i<count_cross_over_paths;i++)
			{
				num = rand()%15;
				printf("\t %3d ",cross_over_positions[i]);
				////getch();
			}
		}
		void print_original_paths()
		{
			int i,j,k,pos;
//			printf("\n Meeting Pairs Are :");
			 rp.w.display_column_no();
			for(k=0;k<count_cross_over_paths;k++)
			{
				if(k%2 == 0)printf("\n\n"); // to print pair wise

				i=cross_over_positions[k];
				printf("\n%2d : ",i);
				for(j=0;j<=14;j++)
				{
					printf("%4d", rp.path[i][j]);
				}
				printf("==%4d", path_length[i]);
			}

		}
		void display_newly_generated_paths_by_meeting()
		{
			int i,j;
```

```
//			printf("\n Newly Generated Paths Are :");
			 rp.w.display_column_no();
			for(i=0;i<count_cross_over_paths;i++)
			{
				printf("\n%2d : ",cross_over_positions[i]);
				for(j=0;j<=14;j++)
				{
					printf("%4d",new_paths[i][j]);
				}
			}
		}
		void do_meeting()
		{
			// one point meeting
			int i,pos,j,k,num1,num2,swap_point;

			int path1,path2;
			for(i=0;i<count_cross_over_paths;i++)
			{
				path1 = cross_over_positions[i];
				path2 = cross_over_positions[i+1];
				i++;
//				printf("\n path1 = %d",path1);
//				printf("\n path2 = %d",path2);
//				printf("\n");
			}
			/////////////////// Initialize new paths////////
			for(i=0;i<count_cross_over_paths;i++)
			{
				pos =cross_over_positions[i];
				for(j=0;j<=14;j++)
				{
					if(i%2 ==0)
					{
						if(j<point)
						{
							new_paths[i][j] =  rp.path[pos][j];
						}
						else
						{
							new_paths[i][j] = 0;
						}
					}
					else
					{
						if(j<point)
						{
							new_paths[i][j] = 0;
							//new_paths[i][j] =  rp.path[pos][j];
						}
						else
						{
							new_paths[i][j] =  rp.path[pos][j];
							//new_paths[i][j] = 0;
						}
					}
				}
			}
			//////////////////Performing One Point Meeting
			for(k=0;k<count_cross_over_paths;k++)
			{
//				swap_point= point;
				if(k%2 == 0)
				{
					path1 = cross_over_positions[k];
					path2 = cross_over_positions[k+1];
				}
				else
				{
					path1 = cross_over_positions[k];
					path2 = cross_over_positions[k-1];
				}
				if(k%2 == 0)
				{// To change paths after meeting point
					swap_point= point;
					for(j=0;j<=14;j++)
					{
						num1 =  rp.path[path2][j];
						for(pos=point;pos<=14;pos++)
						{
							num2 =  rp.path[path1][pos];
//							printf("\n %d is compared with %d",num1,num2);
//
							if(num1 == num2)
							{
								new_paths[k][swap_point] = num1;
								swap_point++;
//								printf("\n Swapped");
//								printf("\n Swap_point = %d ",swap_point);
							}
						}
					}// end for j
				}// end if
				else
				{// To change paths before meeting point
					swap_point= 0;
					for(j=0;j<=14;j++)
					{
						num1 =  rp.path[path2][j];
//						for(pos=point;pos<=14;pos++)
						for(pos=0;pos<point;pos++)
						{
							num2 =  rp.path[path1][pos];
//							printf("\n %d is compared with %d",num1,num2);
//
							if(num1 == num2)
							{
								new_paths[k][swap_point] = num1;
								swap_point++;
//								printf("\n Swapped");
//
//						printf("\n Swap_point = %d ",swap_point);
							}
						}
					}// end for j
				}// end if-else
//				printf("\n First generated ");
//				break;
				// check validity of kth and k-1th path path
			}// end loop k to entertain all three pairs
			//////////////////////////////////
			///////////////// Copying new in to original
			//Altered Code from here
			/*

			for(i=0;i<count_cross_over_paths;i++)
			{
				pos =cross_over_positions[i];
				for(j=0;j<=14;j++)
				{
					rp.path[pos][j] = new_paths[i][j];
				}
			}
			///////////////////////////////////////// */

			// To Here

		}// end do_meeting

		void append_new_paths_and_multiply()
		{
			int pos=15,i,j;//to 20
			for(i=0;i<count_cross_over_paths;i++)
			{
				//pos =cross_over_positions[i];
				for(j=0;j<=14;j++)
				{
					rp.path[pos][j] = new_paths[i][j];
				}
				pos++;
			}

		//	printf("\n -----------------");
		//	printf("\n Path Matrix");
		//	printf("\n ----------------");

			int node1,node2,length,total=0;
			int min=10000,max=0;
			float avg;
			int total_all_paths=0;
			////// Initialise path lenths to 0
			for(i=0;i<=21;i++)
			{
				path_length[i] =0;
			}
			///////////////////////////////
			for(i=0;i<=20;i++)
			{
				total=0;
				for(j=1;j<=14;j++)
				{
					node1 = rp.path[i][j-1];
					node2 = rp.path[i][j];
				length = rp.w.wt[node1][node2];
				total+=length;
			}
			//printf("\n i = %d total  = %d",i,total);
			path_length[i] = total;
			total_all_paths+=total;
			if(total < min)
			{
				min = total;
				min_path_length = min;
				position_of_best_path=i;

			}
			if(total > max)
			{
				max = total;
			}
		}// end for i

		//////// Displaying Paths With Lengths ///////////////
		rp.w.display_column_no();
		for(i=0;i<=20;i++)
		{
			printf("\n%2d : ",i);
			for(j=0;j<=14;j++)
			{
				printf("%4d",rp.path[i][j]);
			}
			printf("==%4d",path_length[i]);
		}
		avg = (float)total_all_paths/15;
		printf("\n\n Min = %d max = %d",min,max);
		printf("\n\n Avg Path Length = %4.2f",avg);
		printf("\n Best Path Position = %d",position_of_best_path);
		////////////////////////////////////


	}//end append
	void sort_paths() //sort all 21 to select best 15 paths
	{
		int temp_length,temp_path[14];
		int temp,
		smallest,i,j,pos;
		// Selection Sort
		for(i=0;i<21;i++)
		{
			smallest = path_length[i];
			pos=i;
			for(j=i;j<21;j++)
			{
				if(smallest > path_length[j])
				{
					smallest = path_length[j];
					pos=j;

				}
			}
//			printf("\n Smallest %d = %d, pos = %d",i,smallest,pos);
```

```
				// Replacing path and path length
				temp = path_length[i];
				path_length[i] = path_length[pos];
				path_length[pos] = temp;
				//replacing path
				for(int k=0;k<=14;k++)
				{
					temp = rp.path[i][k];
					rp.path[i][k] = rp.path[pos][k];
					rp.path[pos][k] = temp;
				}
				/////////////////////////////////
			}
			//////// Displaying Paths With Lengths ///////////////
			rp.w.display_column_no();
			for(i=0;i<=20;i++)
			{
				if(i==15)
				{
					printf("\n\n");
				}
				printf("\n%2d : ",i);
				for(j=0;j<=14;j++)
				{
					printf("%4d",rp.path[i][j]);
				}
				printf("==%4d",path_length[i]);
			}
			/////////////////////////////////////

		}// end sorting

};
class CSC
{
		public :
		Meeting m;// containership
		int csc_path[20][21];
		int csc_path_length[20];
		int csc_child[6][15];
		int csc_single_child[15];
		int csc_child_count;
		int distinct_child_count;// Out of three how many are new
		int distinct_child_poistions[10];// but only three child
		int csc_child_paths_length[6];//  but only three child
		int csc_child_good_bad_status[6]; // store 0 if child is really ditinct
								// otherwise store 1
		int appended_population_count;

		CSC()
		{
		}
		void remove_invalid_path()
		{
			int i,j,pos;
			for(i=0;i<=appended_population_count;i++)
			{
				for(j=0;j<=14;j++)
```

```
                        {
                                if(m.rp.path[i][j] < 0 || m.rp.path[i][j] > 14)
                                {
                                        // invalid path
                                        // copy last path in it
                                        int k;
                                        for(k=0;k<=14;k++)
                                        {
                                                m.rp.path[i][k]=m.rp.path[10][k];
                                        }
                                        // also interchange 10th position with ith position to restrict
duplicacy
                                        int temp = m.rp.path[i][i];
                                        m.rp.path[i][i] = m.rp.path[i][10];
                                        m.rp.path[i][10] = temp;

                                        goto next_path2;
                                }
                        }
                        next_path2:
                }

        }
        void call_methods()
        {
                remove_invalid_path();
                appended_population_count=14;
                csc_sort_paths();
                clrscr();
                printf("\n Press Enter To See Meeting ....");
                // //getch();
                clrscr();
                printf("\n -----------------");
                printf("\n Meeting Started");
                printf("\n ----------------");
//              m.count_cross_over_paths = 2;
                m.display_meeting_data();
                m.find_meeting_positions();
                printf("\n -----------------");
                printf("\n Original Paths");
                printf("\n ----------------");
                m.print_original_paths();
                //getch();
                do_csc_cross_over();
                //getch();
                diplay_new_csc_childs();
                //getch();
                clrscr();
                printf("\n -----------------");
                printf("\n CSC Cross Over Final Results");
                printf("\n ----------------");
                diplay_new_csc_childs();
                find_childs_path_length();
                //getch();
                m.multiply_and_display_paths();
                //getch();
                find_new_distinct_childs();
```

```
			clrscr();
			append_new_csc_paths();
			//getch();
			clrscr();
			printf("\n ----------------------");
			printf("\n Appended All Paths");
			printf("\n ----------------------");
			display_appended_csc_population();
			//getch();
			csc_sort_paths();
			//getch();
		}
		void csc_sort_paths() //sort all 21 to select best 15 paths
		{
			int temp_length,temp_path[14];
			int temp,
			smallest,i,j,pos;
			// Selection Sort
			for(i=0;i<=appended_population_count;i++)
			{
				smallest = m.path_length[i];
				pos=i;
				for(j=i;j<=appended_population_count;j++)
				{
					if(smallest > m.path_length[j])
					{
						smallest = m.path_length[j];
						pos=j;

					}
				}
//				printf("\n Smallest %d = %d, pos = %d",i,smallest,pos);
				// Replacing path and path length
				temp = m.path_length[i];
				m.path_length[i] = m.path_length[pos];
				m.path_length[pos] = temp;
				//replacing path
				for(int k=0;k<=14;k++)
				{
					temp = m.rp.path[i][k];
					m.rp.path[i][k] = m.rp.path[pos][k];
					m.rp.path[pos][k] = temp;
				}
				//////////////////////////////////
			}
			/////// Displaying Paths With Lengths //////////////
			m.rp.w.display_column_no();
			for(i=0;i<=appended_population_count;i++)
			{
				if(i==15)
				{
					printf("\n\n");
				}
				printf("\n%2d : ",i);
				for(j=0;j<=14;j++)
				{
					printf("%4d",m.rp.path[i][j]);
			}
			printf("==%4d",m.path_length[i]);
		}
		////////////////////////////////////

	}// end sorting

	void display_appended_csc_population()
	{
		int i,j,node1,node2,length,total=0;
		//////// Displaying Paths With Lengths ///////////////
		m.rp.w.display_column_no();
		for(i=0;i<=appended_population_count;i++)
		{
			printf("\n%2d : ",i);
			for(j=0;j<=14;j++)
			{
				printf("%4d",m.rp.path[i][j]);
			}
			// come back to node 1
			printf("==%4d",m.path_length[i]);
		}
	}


	void append_new_csc_paths()
	{
		int pos=15,i,j;//to 20
		appended_population_count=14;
		for(i=0;i<=csc_child_count;i++)
		{

			// csc_child_paths_length[i]
			if(csc_child_good_bad_status[i]==0)
			{

				//pos =cross_over_positions[i];
				for(j=0;j<=14;j++)
				{
					// restrictinbg to add child if there is any -1 entry
					if(csc_child[i][j] == -1)
					{
						// child is not valid
						goto next_child;

					}
					m.rp.path[pos][j] = csc_child[i][j];
				}
				if(csc_child_paths_length[i] <0)
				{
					goto next_child;
				}
				m.path_length[pos] =csc_child_paths_length[i];
				printf("\n path No %d appended in main array ",i);
				appended_population_count++;
				pos++;
			}
			next_child:
		}
}
void find_childs_path_length()
{

		int i,j,node1,node2,length,total=0;
		////// Initialise path lenths to 0
		for(i=0;i<=csc_child_count;i++)
		{
				csc_child_paths_length[i] =0;
		}
		//////////////////////////////////
		for(i=0;i<=csc_child_count;i++)
		{
				total=0;
				for(j=1;j<=14;j++)
				{
						node1 = csc_child[i][j-1];
						node2 = csc_child[i][j];
						length = m.rp.w.wt[node1][node2];
						/*
						//printf("\n Node 1 = %d",node1);
						//printf("\n Node 2 = %d",node2);
						if(i==4)
						printf("\t Length = %d",length);
						*/
						total+=length;
				}
				//printf("\n i = %d total  = %d",i,total);
				csc_child_paths_length[i] = total;
		}// end for i
}
void find_new_distinct_childs()
{
		int i,j,ii,jj,kk,k;
		for(i=0;i<=csc_child_count;i++)
		{
				csc_child_good_bad_status[i]=0;
		}
		int node1,node2;
		//int path;
		distinct_child_count=-1;
		for(i=0;i<=csc_child_count;i++)
		{// check all three child
				// path=i;
				int found=0;
				for(j=0;j<=14;j++)
				{// compare child with all 15 path
						for(k=0;k<=14;k++)
						{// compare every city of ith child
						 // ith jth path
								node1 = csc_child[i][k];
								node2 = m.rp.path[j][k];
								if(node1!=node2)
								{
										break;
								}
						}
```

```
                if(k==15)// all 15 city matches
                {
                        printf("\n child path %3d matches with path %3d ",i,j);
                        csc_child_good_bad_status[i] = 1;
                        found=1;
                        break;
                }
        }
        if(found == 0)
        {
                printf("\n child path %3d is not found in original population ",i);
                //distinct_child_count++;
                //distinct_child_poistions[distinct_child_count++]=i;
        }
}// end for to check child in original population
////////////////To check one child with other two
for(i=0;i<=csc_child_count;i++)
{// check all three child
        // path=i;
        int found=0;
        for(j=0;j<=csc_child_count;j++)
        {// compare child with other two
          if(i!=j)
          {
                for(k=0;k<=14;k++)
                {// compare every city of ith child
                        node1 = csc_child[i][k];
                        node2 = csc_child[j][k];
                        if(node1!=node2)
                        {
                                break;
                        }
                }
                if(k==15)// all 15 city matches
                {
                        printf("\n child path %3d matches with child path %3d ",i,j);
                        csc_child_good_bad_status[i] = 1;
                        found=1;
                        break;
                }
          }// i!=j
        }
        if((found == 0) && (i!=j))
        {
                printf("\n child path %3d is distinct in all childs ",i);
                //distinct_child_count++;
                //distinct_child_poistions[distinct_child_count++]=i;
        }
}// end for to check child in 2 other childs
// check if child contain all the 15 city nos
for(i=0;i<=csc_child_count;i++)
{
        if(csc_child_good_bad_status[i] == 0)
        {
                // status 0 means till yet it is good child
                for(k=0;k<=14;k++)
                {
                        int k2;
```

```
                                        node1 = k;// search every k
                                        for(k2=0;k2<=14;k2++)
                                        {
                                        // find every city in ith child
                                                node2 = csc_child[i][k2];
                                                if(node2<0)
                                                {
                                                        break;
                                                }
                                                if(node1 ==node2)
                                                {
                                                        // city k exist
                                                        goto next_city;
                                                }
                                        }
                                        csc_child_good_bad_status[i] = 1;
                                        goto next_child;
                                        next_city:
                                }

                        }
                        next_child:
                }


                ////////////////////////////////////////////////////
                printf("\n Valid New CSC Childs are : ");
                for(i=0;i<=csc_child_count;i++)
                {
                        if(csc_child_good_bad_status[i] == 0)
                        {
                                printf("\n Child No %d",i);
                        }
                }
        }
        void init_child_to_zero()
        {
                int j;
                for(j=0;j<=14;j++)
                {
                        csc_single_child[j]=-1;
                }

        }
        void display_child()
        {
                int i,j;
                //m.rp.w.display_column_no();
                //printf("\n%2d : ",i);
                printf("\n Child==");
                for(j=0;j<=14;j++)
                {
                        printf("%4d",csc_single_child[j]);
                }
        }
        void do_csc_cross_over()
        {
                int i,ii,j,path1,path2,ptr,k,path;
```

```
		int route[14],route_length,child_end_side_pointer=15;
		int child_start_side_pointer=-1;
		int sub_routes[10][15],sub_routes_lengths[10];
		int sub_routes_count,sub_routes_pointer;
		csc_child_count=-1;// (if three child then value = 2)
		for(ii=0;ii<=5/*count_cross_over_paths*/;ii+=2)
		{
			///////////////////////
			init_child_to_zero();
			// VER-SAME-FIRST-LAST
			//child_end_side_pointer=15;
			child_end_side_pointer=14;
			///////////////////////

			path1 = m.cross_over_positions[ii];
			path2 = m.cross_over_positions[ii+1];
			printf("\n\n CSC crox [%d,%d]",path1,path2);
			printf("\n Common Routes are :");
			for(i=0;i<14;i++)
			{
				//route_length=-1;
				//printf("\n Pres Enter ");
				//// //getch();
				ptr=i;
				//first-last-same
				//for(j=0;j<=13;j++)
				for(j=1;j<=13;j++)
				{// finding sub route
					route_length=-1;
					int city_from_path_1 = m.rp.path[path1][ptr];
					int city_from_path_2 = m.rp.path[path2][j];
					if(city_from_path_1 == city_from_path_2)
					{
						//first-last-same
						//for(k=j;k<=14&&ptr<=14;k++,ptr++)
						for(k=j;k<=13&&ptr<=13;k++,ptr++)
						{
							if( (k==j) && (m.rp.path[path1][ptr+1] ==
m.rp.path[path2][k+1]) )
							{
							  //printf("\n Travering node no %d city %d
",i,m.rp.path[path1][ptr]);
							  //printf("First -
%3d",m.rp.path[path1][ptr]);
								//printf("\n
%3d",m.rp.path[path1][ptr]);
								route[++route_length]
=m.rp.path[path1][ptr];
							}
							if(m.rp.path[path1][ptr+1] ==
m.rp.path[path2][k+1])
							{
								i++;// forward ptr

	//printf("%3d",m.rp.path[path1][ptr+1]);
								route[++route_length]
=m.rp.path[path1][ptr+1];
							}
```

```
                                                else
                                                {
                                                        break;
                                                }
                                        }
                                        if(route_length!=-1)// common sub route found
                                        {
                                                int jj;
                                                /////Displaying The Route////////////
                                                // printf("\n Route Found Length %d ==
",route_length+1);

                                                printf("\n Route : ");
                                                for(jj=0;jj<=route_length;jj++)
                                                {
                                                        printf("%4d",route[jj]);
                                                }
                                                ///////////////////////////////////////////////
                                                ////// copy route to child array (right jutified)
                                                for(jj=route_length;jj>=0;jj--)
                                                {
                                                        csc_single_child[--
child_end_side_pointer] = route[jj];
                                                }
                                                //display_child();
                                                //// //getch();


                                                ///////////////////////////////////////////////
                                        }//if route found
                                }//picking the node from path2
                        }//searching the whole path 2
                }// for every node of child 1
                //////// Complete the child from path one
                //////// starting poition which are not in common path
                child_start_side_pointer=-1;
                //
                //for(int jjj=0;jjj<=14;jjj++)
                for(int jjj=0;jjj<=13;jjj++)
                {
                        int node = m.rp.path[path1][jjj];
                        //// search node in child
                        int found=0;
                        for(int kkk = child_end_side_pointer;kkk<=14;kkk++)
                        {
                                if(node == csc_single_child[kkk])
                                {
                                        found=1;
                                        break;
                                }
                        }
                        if(found==0)//node not in child
                        {
                                csc_single_child[++child_start_side_pointer]=node;
                                //display_child();
                                //// //getch();

                        }
                }
```

```
		csc_single_child[14] = 14;
		//////////////// Displaying the ne child
		printf("\n Complete Child");
		display_child();
		//getch();

		///////////////////////////////////////
		///copying child in to csc_childs 2d array
		// ver-same-first-last
		// check validity of a chuild
		// first=0 and last=14
		// all cities from 1-14 must be present
		int valid_flag = 1;
		for(int i=1;i<=13;i++)
		{
			if((csc_single_child[0] != 0) || (csc_single_child[14] != 14))
			{
				valid_flag = 0;
				break;
			}
			// ind city i in csc_single_child[]
			int found = 0;
			// chck cities from 2 to 13 in child
			for(int j=1;j<=13;j++)
			{
				if(csc_single_child[j] ==i)
				{
					found = 1;
					break;
				}
			}
			if(found == 0)
			{
			   valid_flag = 0;
			   break;
			}
		}
		// check validity  ///////////////////////////////

		if(valid_flag == 1 )
		{
			csc_child_count++;
			for(jjj=0;jjj<=14;jjj++)
			{
				csc_child[csc_child_count][jjj] = csc_single_child[jjj];
			}
		}
		/////////////////////////////////////////

	}//for  child pairs
	/*
	/////// Diplaying New csc childs////
	printf("\n All New Child Are :");
	for(int
	jjj=0;jjj<=csc_child_count;jjj++)
	{
		printf("\n Child %2d==",jjj);
		for(int kkk=0;kkk<=14;kkk++)
```

```
                {
                        printf("%3d",csc_child[jjj][kkk]);
                }
        }
        //////////////////////////////////
        */

}// end method do csc crox over
void diplay_new_csc_childs()
{
        find_childs_path_length();
        printf("\n New CSC Childs Are : \n");
        m.rp.w.display_column_no();
        //for(i=0;i<=14;i++)
        for(int jjj=0;jjj<=csc_child_count;jjj++)
        {
                printf("\n%2d : ",jjj);
                for(int kkk=0;kkk<=14;kkk++)
                {
                        printf("%4d",csc_child[jjj][kkk]);
                }
                printf("== %4d",csc_child_paths_length[jjj]);
        }
}




void generate_csc_path()
{ // Importnt module that copy 15 min path length path from 2d array
  // in to csc population
        find_csc_paths();
        clrscr();
        //printf("\n -----------------");
        //printf("\n CSC Paths");
        //printf("\n ----------------");
        //display_csc_paths();
        copy_csc_paths_in_to_random_paths();//rp object
        m.multiply_and_display_paths();
        clrscr();
        copy_weight_matrix_and_csc_path_matrix_to_file();
        printf("\n -----------------");
        printf("\n CSC Weight Matrix ");
        printf("\n -----------------");
        m.rp.w.display_weight();//diplay_weight();
        printf("\n -----------------");
        printf("\n CSC Path Matrix ");
        printf("\n -----------------");
        m.multiply_and_display_paths();
}
void copy_weight_matrix_and_csc_path_matrix_to_file()
{
        int i,j;
        char file1[] = "weight.cpp",file2[] = "cscpaths.cpp";
        FILE *fp1,*fp2;
        fp1 = fopen(file1,"w");
        fp2 = fopen(file2,"w");
        // copy  weight
        for(i=0;i<=14;i++)
        {
                for(j=0;j<=14;j++)
                {
                        fprintf(fp1,"%3d",m.rp.w.wt[i][j]);
                }
                fprintf(fp1,"%c",'\n');
        }
        // copy  weight
        for(i=0;i<=14;i++)
        {
                for(j=0;j<=14;j++)
                {
                        fprintf(fp2,"%3d",m.rp.path[i][j]);
                }
                fprintf(fp2,"%6d",m.path_length[i]);
                fprintf(fp2,"%c",'\n');
        }

}

void copy_csc_paths_in_to_random_paths()//rp object
{
        int i,j;
        for(i=0;i<=14;i++)
        {
                for(j=0;j<=14;j++)
                {
                        m.rp.path[i][j] = csc_path[i][j];
                }
        }
}
void display_csc_paths()
{
        int i,j;
        //////// Displaying CSC paths ////////////////
        m.rp.w.display_column_no();
        for(i=0;i<=14;i++)
        {
                printf("\n%2d : ",i);
                for(j=0;j<=14;j++)
                {
                        printf("%4d",csc_path[i][j]);
                }
                printf("==%4d",csc_path_length[i]);
        }

}
void find_csc_paths()
{
        int path_no;
        int path_length;
        //m.rp.w.display_weight();
        m.multiply_and_display_paths();
        for(int i=0;i<=14;i++)
```

```
{
        //printf("\n\n Press Enter ");
        //// //getch();
        //clrscr();
        m.call_methods_2();
        m.sort_paths();
        //printf("\n\n Press Enter ");
        //// //getch();
        //clrscr();
        {//block find new distinct csc path
                path_no=0;
                int distinct=0,x2;
                int found=0;
                //do not check first path
                while(distinct != 1 && i !=0)
                {// this loop will give a distinct path
                 // and set it's loc in path_no
                        path_length = m.path_length[path_no];
                        found=0;
                        for(x2=0;x2<=i-1;x2++)//linear search
                        {
                            if(csc_path_length[x2] == path_length)
                            {
                                        //not distinct path
                                        found=1;
                                        break;
                            }
                    }
                    if(found == 0)
                    {
                                distinct = 1;
                                break;
                    }
                    path_no++;
                }
        }//end block

        {//block
                // copy path path_no (new distinct csc path)
                // int csc_path array
                // and copy path 21 in path path_no
                int x1;
                for(x1=0;x1<=14;x1++)
                {
                        csc_path[i][x1] = m.rp.path[path_no][x1];
                }
                for(x1=0;x1<=14;x1++)
                {
                        m.rp.path[path_no][x1] = m.rp.path[20][x1];
                }
                csc_path_length[i] = m.path_length[path_no];
                int tmp1 = m.path_length[path_no];
                m.path_length[path_no] = m.path_length[20];
                m.path_length[20] = tmp1;
                //printf("\n path[%d]=%d",i,tmp1);
                //// //getch();
        }//end block
        m.rp.w.display_weight();
```

```
                        m.multiply_and_display_paths();
                }//end for
        }/// end find csc paths
};
void main()
{
        clrscr();
        int i,j,k;
        CSC csc;
        printf("\n Press enter to start 20 iterations of classical cross over");
        //getch();
        csc.generate_csc_path();
        for(i=1;i<=500;i++) // running two iterations only
        {
                csc.call_methods();
                if(i<=

                                                                        50)
                {
                        if(i%10 == 0)
                        {
                                printf("\n   CSC Iteration no = %d",i);
                                getch();
                        }
                }
                else
                {
                        if(i%100 == 0)
                        {
                                printf("\n   CSC Iteration no = %d",i);
                                getch();
                        }
                }

        }
        //getch();
}
```