# CHAPTER 3

# PROPOSED METHOD

## 3.1 Application of k-means Cclustering on a Data Set

The procedure follows a simple and easy way to classify a given data set through a certain number of clusters (assume k clusters) fixed apriori. The main idea is to define k centers, one for each cluster. These centers should be placed in a cunning way because of different location causes different result. So, the better choice is to place them as much as possible far away from each other. The next step is to take each point belonging to a given data set and associate it to the nearest center. When no point is pending, the first step is completed and an early group age is done. At this point we need to re-calculate k new centroids as barycenter of the clusters resulting from the previous step. After we have these k new centroids, a new binding has to be done between the same data set points and the nearest new center. A loop has been generated. As a result of this loop we may notice that the k centers change their location step by step until no more changes are done or in other words centers do not move any more.

We have taken following eight points (with (x, y) representing locations) group them into three clusters A1(2, 10) A2(2, 5) A3(8, 4) A4(5, 8) A5(7, 5) A6(6, 4) A7(1, 2) A8(4, 9). Initial cluster centers are: A1(2, 10), A4(5, 8) and A7(1, 2). The distance function between two points *a=(x1, y1)* and *b=(x2, y2)*  is defined as: *ρ(a, b) = |x2 – x1| + |y2 – y1|* .

Solution:

Iteration 1

First we list all points in the first column of the table above. The initial cluster centers – means, are (2, 10), (5, 8) and (1, 2) - chosen randomly. Next, we will calculate the distance from the first point (2, 10) to each of the three means, by using the distance function:

**Table 3.1: Distance of points from the initial centers**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | (2, 10) | (5, 8) | (1, 2) |  |
|  | **Point** | **Dist Mean 1** | **Dist Mean 2** | **Dist Mean 3** | **Cluster** |
| A1 | (2, 10) | 0 | 5 | 9 | 1 |
| A2 | (2, 5) | 5 | 6 | 4 | 3 |
| A3 | (8, 4) | 12 | 7 | 9 | 2 |
| A4 | (5, 8) | 5 | 0 | 10 | 2 |
| A5 | (7, 5) | 10 | 5 | 9 | 2 |
| A6 | (6, 4) | 10 | 5 | 7 | 2 |
| A7 | (1, 2) | 9 | 10 | 0 | 3 |
| A8 | (4, 9) | 3 | 2 | 10 | 2 |

After first iteration

Cluster 1 contains the points (2, 10)

Cluster 2 contains the points (8, 4) (5, 8) (7, 5) (6, 4) (4, 9)

Cluster 3 contains the points (2, 5) (1, 2)

Next, we need to re-compute the new cluster centers (means). We do so, by taking the mean of all points in each cluster.

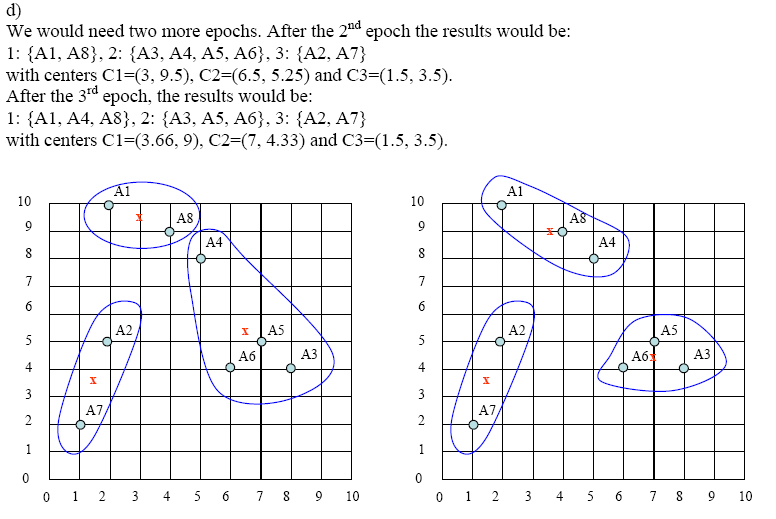
For Cluster 1, we only have one point A1(2, 10), which was the old mean, so the cluster center remains the same.

For Cluster 2, we have ( (8+5+7+6+4)/5, (4+8+5+4+9)/5 ) = (6, 6)

For Cluster 3, we have ( (2+1)/2, (5+2)/2 ) = (1.5, 3.5)



The initial cluster centers are shown in red dot. The new cluster centers are shown in red x. That was Iteration1 (epoch1). Next, we go to Iteration2 (epoch2), Iteration3, and so on until the means do not change anymore. In Iteration2, we basically repeat the process from Iteration1 this time using the new means we computed.



But due to drawbacks of k-means algorithm that it works with only numerical value we need to have a technique capable of working with both numerical and categorical values. K-prototype solves this purpose by applying data mining and forming clusters based on the both numerical and categorical values .

## 3.1 Clustering mixed attributes using k-prototype algorithm

We are basically using k-prototype algorithm to cluster dataset containing both numeric and categorical values.

K-prototype is an extension of k-mean algorithm. So the basic ideology of k-prototype algorithm is same as k-mean algorithm. The k-means based methods are efficient for processing large data sets, thus very attractive for data mining. The major handicap for them is that they are often limited to numeric data. The reason is that these algorithms optimize a cost function defined on the Euclidean distance measure between data points and means of clusters. Minimizing the cost function by calculating means limits their use to numeric data.

The algorithm is based on the k-means paradigm but removes the numeric data only limitation whilst preserving its efficiency. The algorithm clusters objects with numeric and categorical attributes in a way similar to k-means. Because objects are clustered against k prototypes instead of k means of clusters, we call it the k-prototypes algorithm. We have developed a method to dynamically update the k prototypes in order to maximize the intra cluster similarity of objects. The object similarity measure is derived from both numeric and categorical attributes. When applied to numeric data the algorithm is identical to k-means.

Basic k-prototype algorithm is as follows:

1. Given a dataset X select k initial prototypes one for each cluster.
2. Allocate each object in ***X*** to a cluster whose prototype is the nearest to it.
3. Update the prototype of the cluster after each allocation. After all objects have been allocated to a cluster, retest the similarity of objects against the current prototypes.

If an object is found such that its nearest prototype belongs to another cluster rather than its current one, reallocate the object to that cluster and update the prototypes of both clusters.

1. Repeat (3) until no object has changed clusters after a full cycle test of ***X***.

The algorithm is built upon three processes, *initial prototypes selection*, *initial allocation,* and *re-allocation*. The first process simply randomly selects *k* objects asthe initial prototypes for clusters. The second process initializes different points to cluster.Starting from a set of initial cluster prototypes, this process assigns each object to acluster and updates the cluster prototype accordingly after each assignment.In Figure 2, X[i] represents object *i* and X[i,j] the value of attribute *j* for object *i*.O\_prototypes[] and C\_prototypes[] store the numeric and categorical attribute partsof cluster prototypes respectively. O\_prototypes[i,j] and C\_prototypes[i,j] are two

corresponding numeric and categorical elements of the prototype for cluster *i*. **Distance**() is a square Euclidean distance function. Clustership[] and ClusterCount[] record cluster membership of objects and numbers of objects in clusters. SumInCluster[] sums up numeric values of objects in clusters and is used to update numeric attributes of cluster prototypes. FrequencyInCluster[] records frequencies of different values of categorical attributes in clusters. The function **HighestFreq**() returns the value with the highest frequency.

The reallocation process is similar to the initial allocation process except that after reallocation of an object, prototypes for both the previous and current clusters of the object are updated. Variable moves records the number of objects which have changed clusters in the process.

**3.3 Procedure**

To explain how k-prototype work we have taken a small database of some companies located at different location. The attributes contain the company name, there revenue, Type of company and Location. There are basically 3 categorical attribute and 1 numeric attribute of this dataset. We have taken 7 objects which need to be clustered in 3 of these cluster.

As per the algorithm we randomly choose 3 clusters to be cluster or the prototype.

For our dataset these are the initial prototypes:

After this the first thing to do is to allocate the objects from the dataset to one of these cluster based on their similarity.

For this we find the distance of object to all these 3 clusters and allocate the object to the cluster having the least distance. Here the distance is calculated using two measures:

For calculating distance of numeric attribute following formulae is used:

D(Xi , Qi)= ∑ (xij-qij)2

Here D(Xi , Qi) is the function having arguments

Xi =ith object of the dataset

Qi= ith cluster of the 3 cluster taken.

For calculating distance of categorical attribute following formulae is used:

If the value of the attribute of the two object are same the function will return 0 otherwise 1.

Function is ∂ (xij,qij) where xij is the jth attribute of the ith object and qij is the jth attribute of the ith cluster.

Using above formulae we have calculated the distance for each of the object like:

Object 1 to cluster 1: 0+0+0+0=0

Object 1 to cluster 2: 1+1+1+0=3

Object 1 to cluster 3: 1+36+1+0=38

So the object 1 will be assigned to the cluster 1.

Similarly

Object 2 to cluster 1: 1+1+1+0=3

Object 2 to cluster 2: 1+1+1+0=3

Object 2 to cluster 3: 1+100+1+1=103

So the object 2 will be assigned to the cluster 1.

After following these step we get object 3 in cluster 2, object 4 in cluster 2, object 5 in cluster 3, object 6 in cluster 3 and object 7 in cluster 3.

This is the initial allocation step. Now we need to calculate the all the 4 attributes of the cluster.

For numeric attribute we sum the total value of all the object attribute value and divide the sum from number of objects in the cluster.

So for cluster 1 :

**3.2 Implementation**

For Initial allocation:

FOR i = 1 TO NumberOfObjects

Mindistance=Distance(X[i],O\_prototypes[1]) + gamma\*Sigma(X[i],C\_prototypes[1])

FOR j = 1 TO NumberOfClusters

distance=Distance(X[i],O\_prototypes[j])+gamma\*

Sigma(X[i],C\_prototypes[j])

IF (distance < Mindistance)

Mindistance=distance cluster=j

ENDIF

ENDFOR

Clustership[i]=cluster

ClusterCount[cluster] + 1

FOR j=1 TO NumberOfNumericAttributes

SumInCluster[cluster,j]+X[i,j] O\_prototypes[cluster,j]=SumInCluster[cluster,j]/ClusterCount[cluster]

ENDFOR

FOR j=1 TO NumberOfCategoricAttributes

FrequencyInCluster[cluster,j,X[i,j]] + 1 C\_prototypes[cluster,j]=HighestFreq(FrequencyInCluster,cluster,j)

ENDFOR

ENDFOR

For re-allocation:

moves=0

FOR i = 1 TO NumberOfObjects …

(To find the cluster whose prototype is the nearest to object i. Same as Figure 2) …

IF (Clustership[i]<>cluster)

moves+1

oldcluster=Clustership[i]

ClusterCount[cluster] + 1

ClusterCount[oldcluster] - 1

FOR j=1 TO NumberOfNumericAttributes

SumInCluster[cluster,j] + X[i,j] SumInCluster[oldcluster,j] - X[i,j] O\_prototypes[cluster,j]=SumInCluster[cluster,j]/ClusterCount[cluster] O\_prototypes[oldcluster,j]= SumInCluster[oldcluster,j]/ClusterCount[oldcluster]

ENDFOR

FOR j=1 TO NumberOfCategoricAttributes

FrequencyInCluster[cluster,j,X[i,j]] + 1

FrequencyInCluster[oldcluster,j,X[i,j]] - 1

C\_prototypes[cluster,j]=HighestFreq(cluster,j) C\_prototypes[oldcluster,j]=HighestFreq(oldcluster,j)

ENDFOR

ENDIF

ENDFOR

## 3.1

This is basically to solve Travelling Salesman Problem in which we need to find the shortest route among all cities without visiting any city twice. Let take an example of 8 cities named 0, 1, 2, 3, 4, 5, 6 and 7. So, there are 88 (or, 16,777,216) possible combinations, but this PSO algorithm can find the solution computing less than 83 solutions.

The locations of the cities we have taken in our example are (30, 5), (40, 10), (29, 25), (40, 20), (19, 25), (20, 5), (9, 19), (9, 9).This is more like a circle: the solution can be easily checked by seeing a graph and compared the algorithm's solutions with. By performing some calculations, we have found the solution to be about 86.6299, which is the target value of our algorithm.

To simplify this example, starting or ending point doesn’t matter, moreover the direction of the tour travels also doesn’t matter. For example, a solution that looks like 23456701 is valid because it travels forward from 2 and around to 1. The solution 65432107 is equally valid because it just goes backward from 6 to 7.

The most important part of the PSO algorithm is the calculation of the velocity vector as it is this velocity vector which updates the particle position. So, we first added the global worst to the global variables. The velocity score is then calculated using this global worst.

In this example, only three variables can be experimented with:

1. COUNT\_OF\_PARTICLE - number of particles in the test.
2. MAX\_VELOCITY - maximum change allowed for velocity.
3. EPOCHS\_MAX – maximum number of iterations.

**Figure 3.1: Coordinates of Cities Location**

### 3.1.1 How it works?

Every particle has some position (x, y) and a velocity (vx, vy) with which it is moving through. Every particle has some tendency to move in direction same as before.

Every particle possesses some acceleration due to change in their velocity which depends upon two things:

1. Each particle will try to move towards the best location that it has found personally (also known as personal best-pBest).
2. Each particle will try to move toward the best location that any particle has found (also known as global best-gBest).

The strength with which the particles are attracted in each of these directions depends upon the parameters like pBest and gBest. PSO is used to either maximize or minimize the fitness function by updating the position in every loop. All paths from source to destination are considered as particles.

### 3.1.2 Procedure

For solving Travelling Salesman Problem using Particle Swarm Optimization ,we first choose some random paths(particles). In the example we are going to discuss we have taken 4 randomly generated paths(particle).We choose these paths by randomizing the initial path value( (0,1,2,3,4,5,6,7) for all the four particles).The only thing need to keep in mind while choosing these paths is that the path should be such that it should connect all points(cities) and not repeating any point(city) twice.

So

Initialize:

1. Path 10 1 2 3 4 5 6 7
2. Path 20 1 2 3 4 5 6 7
3. Path 30 1 2 3 4 5 6 7
4. Path 40 1 2 3 4 5 6 7

Now randomly rearrange each path(particle) 10 times. For rearranging these particles we select two cities for any path keeping in mind the two cities are different and swap them. We do this 10 times to increase the randomness. After randomly rearranging we get the following paths.

1. Path 12 4 3 6 5 7 0 1
2. Path 21 3 5 2 6 0 7 4
3. Path 31 3 5 2 6 0 7 4
4. Path 46 2 5 3 7 4 1 0

We calculate the distances of all these four paths(particles).

The distance comes out to be:

1. Path 1116.18432
2. Path 2180.60110
3. Path 3182.07424
4. Path 4185.11207

Now we will iterate until the maximum number of iterations has been reached or the target has been found. The maximum iteration count we have taken in this example is 10000. The target value taken by us is 86.63. If the result distance of any path comes out to be less than or equal to 86.63 we stop the iteration.

After every iteration the path value of every path changes due to change in velocity introduced. So we keep in account the personal best of each path (particle) .The personal best of each particle will be the best value (the minimum distance ) the particle has encountered after n iteration.

We also need to keep track of the global best of all the particle. Global best means the best value (minimum distance) achieved so far.

The next step we are going to do is to bubble sort the paths (particles) according to their personal best scores i.e from best to worst.

We now update the particle by changing the path (particle) value.We do this with the help a very important factor, velocity. We calculate the velocity by using the formulae

Value=(MAX\_VELOCITY\*(personal best of path( particle) )/worst result;

Here MAX\_VELOCITY is the maximum velocity allowed. We have taken MAX\_VELOCITY as 4.

Here worst result is the worst pBest among all paths.

This value is the indication of the no of changes to be allowed to all of the four paths so as to update them.

The values comes out to be:

1. Path 12
2. Path 22
3. Path 33
4. Path 43

Now we will again rearrange the four path according to the no of changes allowed.

After rearranging the paths comes out to be

1. Path 12 4 3 6 5 7 0 1 116.18436
2. Path 21 5 7 4 0 6 3 2 162.00802
3. Path 35 4 1 3 2 0 7 6 117.88545
4. Path 43 6 2 4 7 5 0 1 172.94305

Now these are the results after first iteration. We check whether the target value is achieved or not .We also take in account the maximum iterations.

The whole process is repeated till we reach our destination of finding the shortest path connecting all cities.

So we again bubble sort the paths (particles) and update the velocity. The updated velocity comes out to be:

1. Path 12
2. Path 22
3. Path 32
4. Path 42

Updating the paths (particles), the paths (particles) comes out to be

1. Path 12 3 4 7 6 5 0 1 110.23192
2. Path 22 4 3 6 5 7 0 1 116.18436
3. Path 32 4 1 3 5 7 0 6 114.15432
4. Path 46 7 4 5 3 9 1 2 138.37124

Again for the example we have taken the target value is not reached after the 2nd iteration. So again updating the path by applying changes to path with the velocity. After iterating 800 times we get the result in the 801 iteration.

The updated velocity comes out to be:

1. Path 12
2. Path 23
3. Path 32
4. Path 42

The updated path will be:

1. Path 12 3 4 7 6 5 0 1 110.23192
2. Path 23 4 5 7 6 2 0 1 131.82403
3. Path 37 0 1 2 3 4 5 6  86.62995
4. Path 42 4 3 6 5 7 0 1 116.18436

Here we get the personal best of the path (particle) less than the target, we will terminate and finally get the shortest route connecting all points (cities).

The shortest path for out example in

Route 7 0 1 2 3 4 5 6

The distance for the shortest path is 86.6299.

We analyze the algorithm by changing the count of the no of paths and the velocity\_max i.e. the maximum allowable change.

We get the following result.

**Case 1: COUNT\_OF\_PARTICLE = 10, MAX\_VELOCITY = 4, EPOCHS\_MAX = 10000.**

Route 2 3 4 6 5 7 0 1  99.9458

Route 2 7 3 4 6 5 0 1  132.2345

Route 6 4 7 3 5 0 1 2  161.0345

Route 4 7 0 5 2 6 1 3  178.4356

Route 3 6 1 5 0 4 7 2  194.8090

Route 4 6 0 7 2 1 5 3  148.6579

Route 7 0 1 2 3 4 5 6  86.4536

Route 5 2 3 6 7 0 1 4  139.6543

Route7 2 3 6 0 4 5 1  171.7685

Route 2 3 0 4 6 1 5 7  179.8768

Changes for particle 1 2

Changes for particle 2 1

Changes for particle 3 4

Changes for particle 4 3

Changes for particle 5 3

Changes for particle 6 2

Changes for particle 7 4

Changes for particle 8 1

Changes for particle 9 4

epoch number: 87

Target reached.

Shortest Route: 7, 0, 1, 2, 3, 4, 5, 6, Distance: 86.4536

**Case 2:COUNT\_OF\_PARTICLE = 10, MAX\_VELOCITY = 8, EPOCHS\_MAX = 10000.**

Route: 0, 7, 5, 6, 4, 3, 2, 1, Distance: 99.6786

Route: 1, 0, 7, 6, 5, 4, 3, 2, Distance: 86.6345

Route: 6, 4, 1, 5, 2, 3, 0, 7, Distance: 161.5426

Route: 5, 2, 3, 0, 4, 1, 6, 7, Distance: 172.4235

Route: 1, 6, 4, 0, 2, 3, 5, 7, Distance: 162.5345

Route: 0, 1, 3, 2, 7, 5, 6, 4, Distance: 136.65456

Route: 0, 3, 6, 5, 1, 2, 7, 4, Distance: 165.6578

Route: 6, 5, 4, 7, 0, 3, 1, 2, Distance: 133.6764

Route: 7, 3, 1, 2, 4, 6, 5, 0, Distance: 136.4567

Route: 6, 5, 7, 3, 1, 4, 2, 0, Distance: 155.4568

Changes for particle 1: 5

Changes for particle 2: 3

Changes for particle 3: 6

Changes for particle 4: 8

Changes for particle 5: 2

Changes for particle 6: 5

Changes for particle 7: 7

Changes for particle 8: 6

Changes for particle 9: 8

epoch number: 223

Target reached.

Shortest Route: 1, 0, 7, 6, 5, 4, 3, 2, Distance: 86.6325

**Increasing the velocity has very less effect.**

### 3.1.3 Program Module for PSO applied on TSP

**Module for applying basic PSO Algorithm:**

private static void mainalgo()

{

Particle aParticle = null;

int epoch = 0;

boolean done = false;

initialization();

while(!done)

{

// two conditions which can end this loop are:

// the number of maximum epochs or iterations allowed has been reached, or,

// the Target is reached.

if(epoch < MAX\_EPOCHS){

for(int i = 0; i < PARTICLE\_COUNT; i++)

{

aParticle = particles.get(i);

System.out.print("Route: ");

for(int j = 0; j < CountOf\_City; j++)

{

System.out.print(aParticle.data\_Item(j) + ", ");

}

TotalDistance(i);

if(aParticle.pBest() <= TARGET){

done = true;

}

}

Sort(); // sorting of particles according to their pBest scores, best to worst.

newvel\_ocity();

update\_particles();

System.out.println("epoch number: " + epoch);

epoch++;

}

else{

done = true;

}

}

return;

}

**Module for initializing particles with random values(choosing random path):**

private static void initialization()

{

for(int i = 0; i < PARTICLE\_COUNT; i++)

{

Particle newParticle = new Particle();

for(int j = 0; j < CountOf\_City; j++)

{

newParticle.data\_Item(j, j);

}

particles.add(newParticle);

for(int j = 0; j < 10; j++)

{

randomly\_Arrange(particles.indexOf(newParticle));

}

TotalDistance(particles.indexOf(newParticle));

}

return;

}

**Module to get velocity:**

private static void newvel\_ocity()

{

double worstResults = 0;

double vValue = 0.0;

// after sorting, worst element will be the last element in list.

worstResults = particles.get(PARTICLE\_COUNT - 1).pBest();

for(int i = 0; i < PARTICLE\_COUNT; i++)

{

vValue = (V\_MAX \* particles.get(i).pBest()) / worstResults;

if(vValue > V\_MAX){

particles.get(i).vel\_ocity(V\_MAX);

}else if(vValue < 0.0){

particles.get(i).vel\_ocity(0.0);

}else{

particles.get(i).vel\_ocity(vValue);

}

}

return;

}

**Program module to update particle (generating new path ):**

private static void update\_particles()

{

// Best value of the particle is at index 0, so start from the second best.

for(int i = 1; i < PARTICLE\_COUNT; i++)

{

// The particle wil need more changes, when the vel\_ocity of the particle is higher.

int changes = (int)Math.floor(Math.abs(particles.get(i).vel\_ocity()));

System.out.println("Changes for particle " + i + ": " + changes);

for(int j = 0; j < changes; j++){

if(new Random().nextBoolean()){

randomly\_Arrange(i);

}

}

// Update the pBest value.

getTotalDistance(i);

}

return;

}

**Program module to get the best solution:**

private static void printoptimalSolution()

{

if(particles.get(0).pBest() <= TARGET){

// Print it.

System.out.println("Target reached.");

}else{

System.out.println("Target not reached");

}

System.out.print("Shortest Route: ");

for(int j = 0; j < CountOf\_City; j++)

{

System.out.print(particles.get(0).data\_Item(j) + ", ");

}

return;

}

**Program module to get total distance of a path (particle):**

private static void TotalDistance(final int index)

{

Particle thisParticle = null;

thisParticle = particles.get(index);

thisParticle.pBest(0.0);

for(int i = 0; i < CountOf\_City; i++)

{

if(i == CountOf\_City - 1){

thisParticle.pBest(thisParticle.pBest() + getDistance(thisParticle.data\_Item(CountOf\_City - 1), thisParticle.data\_Item(0))); // Full trip.

}else{

thisParticle.pBest(thisParticle.pBest() + getDistance(thisParticle.data\_Item(i), thisParticle.data\_Item(i + 1)));

}

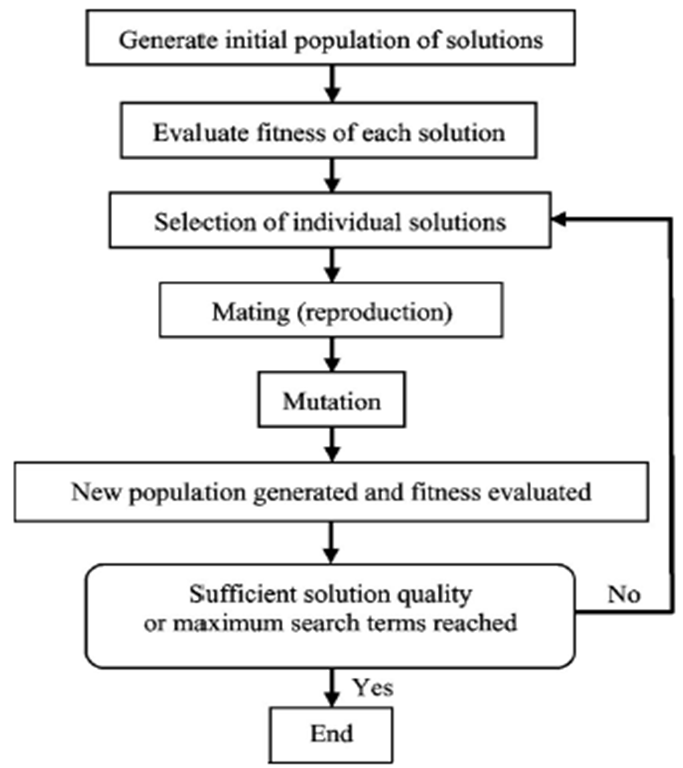
}

return;

}

## 3.2 Genetic Algorithm for Solving TSP

Genetic algorithm is a search heuristic used in the field of artificial intelligence that explains the process of natural evolution. Genetic Algorithm belongs to a class of evolutionary algorithm.GA is applied on a set of chromosomes which are encoded as a population of chromosomes, then a fitness function is used for the evaluation of the fitness of each chromosome, after that offspring is generated through the process of selecting a set of chromosomes known as selection, then selected chromosomes undergoes crossover to create an offspring and finally mutation. After the termination of GA, an optimal solution is found. If the optimal solution is not found depending upon the threshold value, the algorithm is run again with new population. Flowchart for proposed GA is described below-



**Figure 3.2: Flowchart of Genetic Algorithm**

### 3.2.1 Implementation of Proposed Algorithm

We have taken 15 cities and the coordinates of those cities are shown in Fig 3.3:

**Figure 3.3: Coordinates of the 15 Cities for Solving TSP Using GA**

The distance-matrix created for the distance between different cities is symmetric i.e. if we moves from city 1 to city 4 than the distance will be same if we move from city 4 to city 1. Due to this half of the entries from the matrix are omitted.

**Table 3.1: Distance Matrix of 15 Cities**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| CITY | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1 | 0 | 29 | 82 | 46 | 68 | 52 | 72 | 42 | 51 | 55 | 29 | 74 | 23 | 72 | 46 |
| 2 |  | 0 | 55 | 46 | 42 | 43 | 43 | 23 | 23 | 31 | 41 | 51 | 11 | 52 | 21 |
| 3 |  |  | 0 | 68 | 46 | 55 | 23 | 43 | 41 | 29 | 79 | 21 | 64 | 31 | 51 |
| 4 |  |  |  | 0 | 82 | 15 | 72 | 31 | 62 | 42 | 21 | 55 | 51 | 43 | 64 |
| 5 |  |  |  |  | 0 | 74 | 23 | 52 | 21 | 46 | 82 | 58 | 46 | 65 | 23 |
| 6 |  |  |  |  |  | 0 | 61 | 23 | 55 | 31 | 33 | 37 | 51 | 29 | 59 |
| 7 |  |  |  |  |  |  | 0 | 42 | 23 | 31 | 77 | 37 | 51 | 46 | 33 |
| 8 |  |  |  |  |  |  |  | 0 | 33 | 15 | 37 | 33 | 33 | 31 | 37 |
| 9 |  |  |  |  |  |  |  |  | 0 | 29 | 62 | 46 | 29 | 51 | 11 |
| 10 |  |  |  |  |  |  |  |  |  | 0 | 51 | 21 | 41 | 23 | 37 |
| 11 |  |  |  |  |  |  |  |  |  |  | 0 | 65 | 42 | 59 | 61 |
| 12 |  |  |  |  |  |  |  |  |  |  |  | 0 | 61 | 11 | 55 |
| 13 |  |  |  |  |  |  |  |  |  |  |  |  | 0 | 62 | 23 |
| 14 |  |  |  |  |  |  |  |  |  |  |  |  |  | 0 | 59 |
| 15 |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 0 |

### 3.2.2 Initial Population

A unique random number generator function is used to create initial population of chromosomes. The table 3.2 shows the initial population of chromosomes. The initial population comprises of ten chromosomes. Each chromosome is made up of genes used to represent the number assigned to a city, where each chromosome denotes the sequence in which cities have to be traversed.

**Table 3.2: Chromosome Path value**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Chromosome1 | 1 | 4 | 13 | 3 | 8 | 2 | 5 | 15 | 7 | 10 | 14 | 12 | 6 | 9 | 11 |
| Chromosome2 | 3 | 14 | 13 | 2 | 9 | 10 | 5 | 7 | 1 | 15 | 6 | 12 | 8 | 11 | 4 |
| Chromosome3 | 1 | 15 | 3 | 7 | 14 | 11 | 9 | 2 | 13 | 5 | 12 | 4 | 8 | 10 | 6 |
| Chromosome4 | 4 | 12 | 14 | 13 | 5 | 9 | 11 | 8 | 1 | 3 | 10 | 2 | 6 | 7 | 15 |
| Chromosome5 | 11 | 2 | 9 | 5 | 13 | 14 | 3 | 12 | 8 | 1 | 15 | 6 | 4 | 10 | 7 |
| Chromosome6 | 3 | 10 | 7 | 13 | 11 | 2 | 9 | 4 | 15 | 12 | 6 | 5 | 14 | 1 | 8 |
| Chromosome7 | 11 | 5 | 2 | 9 | 15 | 13 | 7 | 8 | 4 | 1 | 3 | 12 | 6 | 10 | 14 |
| Chromosome8 | 3 | 4 | 13 | 14 | 11 | 7 | 10 | 2 | 8 | 15 | 1 | 5 | 9 | 12 | 6 |
| Chromosome9 | 10 | 11 | 7 | 8 | 15 | 1 | 5 | 9 | 12 | 4 | 14 | 6 | 2 | 13 | 3 |
| Chromosome10 | 10 | 11 | 4 | 7 | 12 | 1 | 6 | 3 | 9 | 5 | 15 | 14 | 13 | 8 | 2 |

### 3.2.3 Fitness Value

Fitness function is used to check for the extent of goodness of a chromosome it means that it gives an idea about how good the chromosome is? The length of a chromosome is used as basic criteria for a good chromosome. Chromosomes are created by performing some calculations. Each chromosome is created and then its fitness function is calculated.

### 3.2.4 Selection

Chromosome with smallest fitness value is selected in the Selection method. We have used the roulette wheel selection method.

### 3.2.5 Roulette Wheel Selection Method

Roulette wheel selection method is used for selecting chromosomes from the given population of chromosomes because best chromosomes should be selected to generate new offspring. Parents are selected according to their fitness. The chromosomes are placed on a wheel according to their fitness value. A marble is thrown to select the chromosome and a chromosome with best fitness value will be selected more number of times.

### 3.2.6 Crossover

Crossover is used to generate offspring from the existing parents. Different crossover techniques are used to generate new offspring like 1-point crossover in which an offspring is generated by interchanging gene at one location within two parents. Another method used for crossover is 2-point crossover in which randomly two positions in the chromosomes are chosen and then replace the gene with each other in both chromosomes.

### 3.2.7 Mutation

Mutation is further applied to form a new generation. Different techniques are used for performing mutation like swap adjacent mutation in which adjacent genes a chromosome are interchanged to generate a new chromosome. We applied interchange mutation in which it randomly selects two genes in chromosome and interchange them to generate a new

chromosome. Mutation is applied because there is more probability of finding better chromosome.

### 3.2.8 Termination and Result

The process is terminated as soon as the best tour is found. The tour after completing the number of iterations the best tour will be obtained and the process will be terminated. In figure 4.5 the tour with minimum distance is shown. The minimum distance comes out to be 291 for the problem of 15 cities.

000000000000