World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:7, No:10, 2013

Implementation of Heuristics for Solving Travelling Salesman Problem Using Nearest Neighbour and Minimum Spanning Tree Algorithms

Fatma A. Karkory, Ali A. Abudalmola

***Abstract*—**The travelling salesman problem (TSP) is a combinatorial optimization problem in which the goal is to find the shortest path between different cities that the salesman takes. In other words, the problem deals with finding a route covering all cities so that total distance and execution time is minimized. This paper adopts the nearest neighbor and minimum spanning tree algorithm to solve the well-known travelling salesman problem. The algorithms were implemented using java programming language. The approach is tested on three graphs that making a TSP tour instance of 5-city, 10 – city, and 229–city. The computation results validate the performance of the proposed algorithm.

***Keywords*—**Heuristics, minimum spanning tree algorithm, Nearest Neighbor, Travelling Salesman Problem (TSP).

I. Introduction

WHEN solving problems with computers it quite quickly becomes obvious that some problems are fundamentally harder to solve than others. While for some problems it is possible to design ingenious algorithms that solve the problem efficiently, for others it seems considerably harder, or even impossible to come up with any algorithms at all.

The theory of computational complexity makes it possible to formalize the concepts of "easy" and "hard" problems and the distinction between them. Problems can be formally classified based on their complexity [1], and if a problem belongs to the class of NP-hard or complete problems, we know in advance that there is little hope of finding an efficient and exact algorithm for solving it. Any exact algorithm for such a problem has an execution time exploding for increasing problem sizes, and is often useless for most practical purposes.

The search for alternative algorithms is thus justified, there is a demand for faster algorithms that do not necessarily produce the exact optimal solution, but in most cases provide solutions of sufficient quality. Such methods are called heuristic algorithms or approximation algorithms. One member of the NP-complete class and possibly the most well–known is the Traveling Salesman Problem (TSP). The Traveling Salesman Problem has commanded much attention of mathematicians and computer scientists specifically because it is so easy to describe and so difficult to solve. The problem can be stated as: given a finite number of "cities" along with the cost of travel between each pair of them, find the cheapest way of visiting all of the cities and returning to your starting point. The travel costs are symmetric in the sense that traveling from city X to city Y costs just as much as traveling from Y to X.

As noticed in [1], what makes TSP such an interesting problem is not its direct applicability, since few real problems may actually be described as TSPs, but the fact that TSPs are frequent components of combinatorial optimization problem.

TSP is also an interesting problem because of the wealth of knowledge that has been accumulated over the years.

Because of the hardness of NP-complete problems, no one has found a polynomial-time algorithm for TSP. Therefore much research has concentrated on approximation algorithms whose goal is to find near optimal rather than optimal tours, but it does not mean that it is impossible to solve any large instances of such problems.

II. The Travelling Salesman Problem

A. Formulation

In order to formalize the definition of the traveling salesman problem, several basic computer scientific terms must be defined. The first is that of a graph. This term is used to describe a set of vertices, or nodes, and a set of edges that connect the vertices. A complete graph has an edge between all pairs of vertices. These edges can be directed or undirected and can have weights associated with them. A path between two vertices is a sequence of edges that begins at one vertex and ends at another vertex. A cycle is a path that begins and ends at the same vertex. A Hamiltonian cycle is a cycle covering all nodes in the graph exactly once.

The Traveling Salesman Problem can now be formally defined as follows:

Determine the shortest Hamiltonian Cycle in a complete weighted graph.

B. Solution Algorithms

As it is described on [2], there are three types of approach for solving NP complete problems:

- Devising algorithms for finding exact solution (they will work reasonably fast only for relatively small problem sizes)

Fatma A. Karkory is with the Higher Institute of Refrigeration and Air Conditioning Sokna, Libya (phone: +218913395819; e-mail: f_sokna@ yahoo.com).

Ali A. Abudalmola is with the Higher Institute of Comprehensive professions Aljufra at Sokna, Libya (phone: +218917971489; e-mail: talballah@hotmail.com).

- Devising "sub-optimal" or heuristic algorithms that deliver seemingly or provably good solutions, but which could not proved to be optimal.
- Finding special cases for the problem for which either exact or better heuristic are possible.

In particular here are the Solution algorithms for TSP:

1. Exact Algorithms

The exact algorithms are guaranteed to find an optimal solution but may take an exponential number of iterations.

This means that you have to generate all possible routes and takes the shortest. This becomes impractical as the number of towns, N, increases since the number of possible routes is ! (N-1). The most effective exact algorithms are cutting –plane or facet –finding algorithms [3]. These algorithms are quite complex, and are very demanding of computer power. For example the exact solution for 15,112 Germany cities was determined over a period of 22.6 years on a network of 110 processors located at Rice University and Princeton University [2].

2. Approximation (or Heuristic) Algorithms

These algorithms are much faster and they obtain good solutions, but they do not guarantee the optimal solution. Some of them give solutions that on average differ only by a few percent (2-3%) from the optimal solution [2]. Therefore, if a small deviation from optimum can be accepted, it may be appropriate to use an approximation algorithm.

The heuristic algorithms can be categorized into the following three classes: [3]

- Tour construction algorithms
- Tour improvement algorithms
- Composite algorithms
- Tour Construction Algorithms

In tour construction algorithms, the tour is built from scratch and the cities are added one at a time until a complete tour is found. The best tour construction algorithms usually get within 10-15% of optimally. An Example of a tour constructor algorithm is:

The Nearest Neighbor Algorithm

This is perhaps the simplest and most straightforward TSP heuristic, which is normally fairly close to the optimal route, and it does not take too long to execute (the time complexity is $O(n^2)$, where n is the number of the cities). The key to this algorithm is to always visit the nearest city. Select a starting point, as long as there are cities that have not yet been visited, visit the nearest city that still has not appeared in the tour, finally, return to the first city.

Nearest Insertion

Nearest insertion is quit straightforward .The basics idea of this algorithm is to select the shortest edge, and make a subtour of it, then select a city not in the subtour, having the shortest distance to any one of the cities in the subtour. Find an edge in the subtour such that the cost of inserting the selected city between the edge's cities will minimal. Repeat the selection of the cities until no more cities remain. The time complexity for this algorithm is $O(n^2)$.

Other Examples of tour constructor algorithms include cheapest insertion farthest insertion and random insertion [4].

- Tour Improvement Algorithms

The tour construction heuristic is a greedy approach. The part of the tour, which is already built, remains unchanged during the tour construction process. No attempt is made to change or undo part of the tour that has been built. This is in contrast to the tour improvement heuristic which changes the configuration of the tour during the iterative improvement process until a short tour is found. A simple example of this type of algorithm is the:

2-Opt Algorithm

This algorithm starts from either a random tour or from the tour that resulted from the nearest neighbor heuristic. In this method replace 2 links of the tour with 2 other links in such a way that the new tour length is shorter. Continue in this way until no more improvements are possible.

- Composite Algorithms

Use a construction algorithm to obtain an initial solution and then improve it using an improvement algorithm. An example of such algorithms is:

Simulated Annealing Algorithm

This algorithm has been successfully adapted to give approximate solutions for the TSP. The basics idea of simulated annealing (SA) is from the statistical mechanics and motivated by an analogy of behavior of physical systems in the presents of a heat bath. This algorithm obtains better solution by gradually going from one solution to the next.

3. Special Cases : TSP With Triangle Inequality

In mathematics, the triangle inequality is a statement which states roughly that the distance from A to B to C is never shorter than going directly from A to C. We say that the cost function c satisfies the triangle inequality if for all vertices u, v, w ∈ V,

$$c(u, w) \leq c(u, v) + c(v,w).$$

In other words, the cheapest (shortest) way of going from one city to another is the direct route (straight line) between two cities. In particular, if every city corresponds to a point in Euclidean space, and distance between cities corresponds to Euclidean distance, then the triangle inequality is satisfied (in this paper we consider the problem of TSP with the additional constraint that edge weights satisfy the triangle inequality.) In this method, first compute a minimum spanning tree, whose weight is a lower bound on the length of an optimal travelling salesman tour, and then use the minimum spanning tree to create a tour whose cost is no more than twice that of minimum spanning tree's weight, as long as the cost function satisfies the triangle inequality:

a. Select starting city as root vertex in graph G

b. Find minimum spanning tree of G using Prim's or Kruskal's algorithm
c. Starting from root, traverse the spanning tree using depth first search (DFS).
d. Construct the path according to order of nodes visited in DFS

III. Design

As we mentioned, TSP problem can be view as a graph problem, with nodes for cities and edges for trips. In this stage we decide what classes will be needed to build this application and how those classes are related.

The fundamental decisions about designing a class involve selecting its fields.

In the main class, which is a Graph class, we will associate with each vertex v all the vertices to which v is adjacent. And we will include with each adjacent vertex w the weight of the edge < v, w>, that is we will associate with each vertex v in the Graph object all the vertex – weight Paris of the form <w, weight>. Because the order of the vertex weight pairs (edge) is not important, a LinkedList object is the collection that will be chosen to hold the vertex-weight pairs. Now we will need a collection to associate a given vertex v with the LinkedList object of vertex – weight pairs <w, weight>, where w is adjacent to v and weight is the weight of the edge .The idea of this association is that, given a vertex v, we want to quickly access the associated LinkedList object, as the term "association" suggests we will map each vertex to its LinkedList object, for speed a HashMap object is chosen.

In the HashMap, each key will be a vertex (Vertex class), each value will be a LinkedList object whose elements are objects in an Edge class. The Edge class will contain a Vertex object and a double value, and methods getToVertex () and getWeight (). Because each vertex is associated with its list of neighbors, this representation is referred to as an adjacency list representation.

Fig. 1 shows a graph that represents the cities and the relationship between them (distance), and Fig. 2 shows its representation by Graph class.

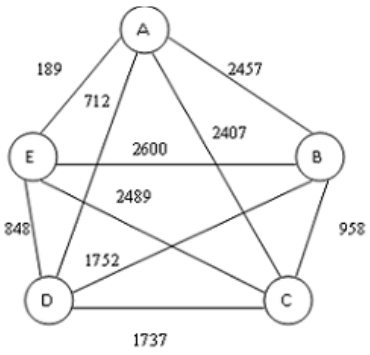

Fig. 1 A completed weighted graph in which the vertices represented cities and the weights represent the distance between these cities

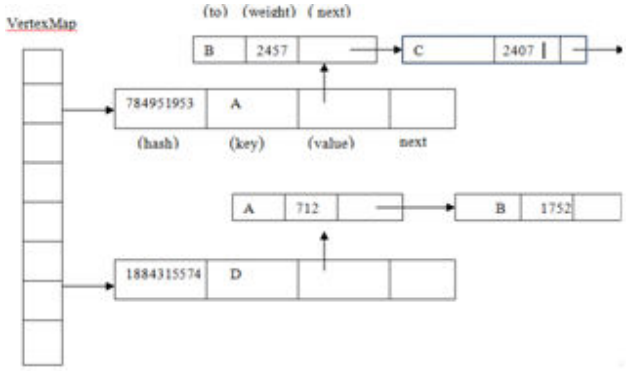

Fig. 2 The internal representation of the graph in Fig. 1 by adjacency list vertex Map

A. Construct Nearest Neighbor Algorithm

We construct this algorithm as a LinkedList object of vertices. The strategy for this constructer is to first visit the start point in the tour (vertex v) and mark it as visited, then store it in the LinkedList collection, now we look for the nearest vertex w adjacent to the start vertex by iterating over v's neighbors using getmin () method, the pair (w,wweight) will be the smallest edge that connected to v, using the getTovertex() method we pick the w vertex and we mark it as visited, we store it in the collection, then loop until the collection has as many vertices as the original graph. Finally we go back to the start vertex v.

Suppose we want to construct the nearest neighbor algorithm for the graph on Fig. 1.

Pick the start vertex in the tour; which is for example A ,mark it as visited ,store it in the LinkedList L. Iterating over A's neighbors <E,189>,<D, 712>,<C,2407>,<B,2457> to get the edge that has the minimum weigh ,which is <E,189> ,store the vertex E in the collection and mark it as visited. During the next iteration on the E's neighbors that has not been in the collection < D, 848>, <C, 2489>, <B, 2600>, the vertex D will be stored in the collection. During the next iteration over D' neighbors <C, 1737>, <B, 1752>, the vertex C will be stored on the collection. Finally on the iteration over C's neighbors, which is, the only one left on the graph that has not been visted, vertex B will be stored on the collection. Going back to the start vertex A we complete the tour of the TSP.

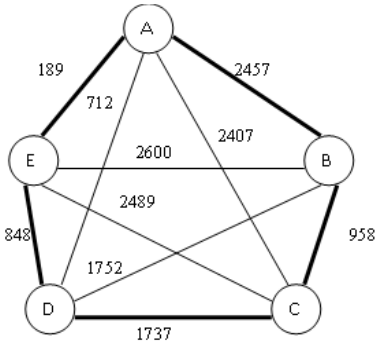

Fig. 3 Applying the nearest neighbor algorithm on Fig. 1

B. Creating Minimum Spanning Tree

In a graph spanning tree is a tree that consists of all the vertices and some of the edges (and their weights) from the graph. A graph may have many spanning trees. A minimum spanning tree is a spanning tree which the sum of all weights is no greater than the sum of all the weights in any other spanning tree.

An algorithm to construct a minimum spanning tree is due to R.C.Prim (1957). The strategy of this algorithm is:

Start with an empty tree T and add the start vertex v in the tour to the tree. For each vertex w such that (v, w) is an edge with weight wweight, save the ordered triple <v, w, wweight > in a collection. Then loop until T has as many vertices as the graph. During each loop iteration, remove from the collection the triple <x, y, yweight>for which yweight is the smallest weight of all triples in the collection; if y is not already in T, add y and the edge (x, y) to T and save in the collection every triple <y, z, zweight>such that z is not already in T and (y, z) is an edge with weight zweight. Because we need to be able to add element that is a triple, to the collection and to remove the triple with lowest weight, priority queue is an appropriate collection to do this task quickly with a heap implementation.

Here the explanation of how the Prim's algorithm (The time complexity is O (V log E+E)) works using the graph in Fig. 1.

Initially, the tree and the PriorityQueue object pq are both empty. Pick A (which is the start city in the tour), add to pq each triple of form <A, w, wweight> where (A, w) is an edge with weight wweight. Fig. 4 shows the contents of T and pq at this point (the triples in pq are shown in increasing order of weight, and the remove Min method returns the triples with the smallest weight).

When the lowest-weighted triple, <A, E, 189> is removed from pq, the vertex E and the edge (A, E) are added to T, and the triples <E, D, 848>, <E, C, 2489> and <E, B, 2600> are added to pq (Fig. 5).

During the next iteration, the triple <A, D, 712> is removed from pq, the vertex D and the edge (A, D) are added to T, and the triples < D, C, 1737> and <D,B,1752> are added to pq (Fig. 6).

During the next iteration, the triple <E, D, 848> is removed from pq, but noting is added to T or pq because D is already in T.

During next iteration the triple (D, C, 1737> is removed from the pq, the vertex C is and the edge (D, C) is added to T. and the triples < C, B, 958> added to pq (Fig. 7).

During the next iteration, the triple <C, B, 958>is removed from the pq, the vertex B and the edge (C, B) is added to T, and noting is added to pq because all of C's edges are already in T and we are done (Fig. 8).

T	Priority Queue
A	(A,E,189)
	(A,D,712)
	(A,C,2407)
	(A,B,2457)

Fig. 4 The content of T and pq during the application of Prim's algorithm to the graph in Fig. 1

T	Priority Queue
AE	(A,D,712)
	(E,D,848)
	(A,C,2407)
	(A,B,2457)
	(E,C,2489)
	(E,B,2600)

Fig. 5 The content of T and pq during the application of Prim's algorithm to the graph in Fig. 1

T	Priority Queue
AE,AD	(E,D,848)
	(D,C,1737)
	(D,B,1752)
	(A,C,2407)
	(A,B,2457)
	(E,C,2489)
	(E,B,2600)

Fig. 6 The content of T and pq during the application of Prim's algorithm to the graph in Fig. 1

T	Priority Queue
AE,AD,DC	(C,B,958)
	(D,B,1752)
	(A,C,2407)
	(A,B,2457)
	(E,C,2489)
	(E,B,2600)

Fig. 7 The content of T and pq during the application of Prim's algorithm to the graph in Fig. 1

T	Priority Queue
AE,AD,DC,CB	(D,B,1752)
	(A,C,2407)
	(A,B,2457)
	(E,C,2489)
	(E,B,2600)

Fig. 8 The content of T and pq during the application of Prim's algorithm to the graph in Fig. 1

C. Depth First Search

The MST gives an upper bound for the minimal tour of the graph. To find a solution for the traveling salesman problem is to traverse the MST. This will be done by using a graph – traversal algorithm; depth first search (DFS). The strategy followed by depth – first Search, as its name implies, to search "deeper" in the sub-graph we obtained from the MST whenever possible; after visiting a given vertex, we visit each not –yet reached vertex in a path that starts at the given vertex. We then back up to the most recently visited vertex that has a not –yet-reached adjacent vertex. Another path is begun starting with that unvisited vertex. With a depth – first search the next vertex to be visited is the most recently reached vertex, to do so the appropriate collection to store the vertices is a Stack.

We will explain the idea behind the depth –first search in relation to the minimum-spanning tree that we get in the last section Fig. 9.

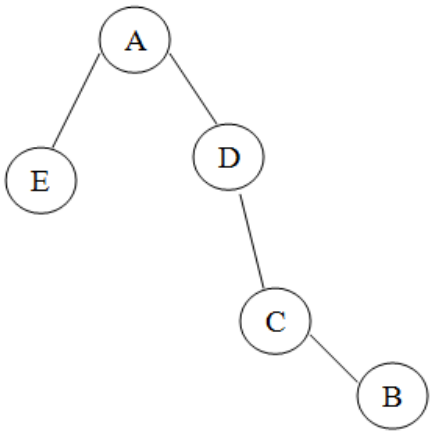

Fig. 9 Minimum spanning tree for the graph on Fig. 1

Pick starting point – in this case, vertex A, visit this vertex push it into a stack so we can remember it and mark it so we will not visit it again. Next we go to any vertex adjacent to A that has not yet been visited so we are at E. At this point we need to do something else because there are no unvisited vertices adjacent to E. So we pop E off the stack, which brings us back to A. The next vertex adjacent to A is D, visit it push it in the stack and mark it as visited. Repeat this rule we visit C and then B and we are done.

Order of traversal in DFS: A → E →D →C→B

To complete the tour for the traveling salesman problem we go back to start point; A.

IV. Class Diagram

The UML class diagram describes the system design that we have developed in terms of the classes and the relationship between them.

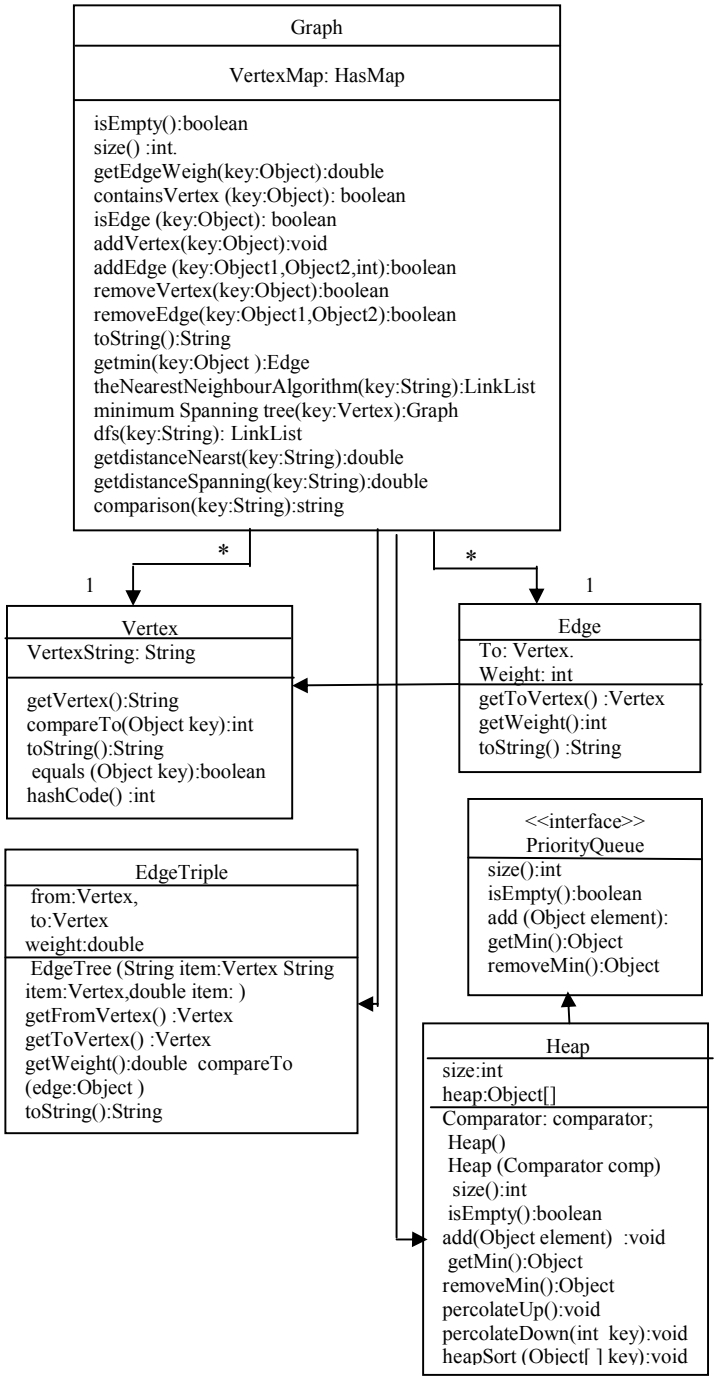

Fig. 10 Class diagram for the system

V. Implementation and Testing

A. Implementation

Implementation involves developing a graph data structure to hold the data; developing two methods to implement the Nearest Neighbor and Minimum Spanning Tree algorithms; and finally developing an interface to allow the user interact with the system.

In the Graph class the main attribute is a vertex Map that involves four one – line method definitions:

1. public Graph()

This method creates an empty Graph

2. public boolean isEmpty()

This method returns true if this Graph is empty; otherwise, it returns false.

3. public int size()

This method returns the number of vertices in the Graph.

4. public boolean containsVertex (Vertex vertex)

This method returns true if the Graph contains vertex; otherwise, it returns false.

5. Adding a vertex to a Graph is straightforward:

```
public boolean addVertex (Vertex vertex) {
    if (vertexMap.containsKey (vertex))
     return false;
    vertexMap.put(vertex,new LinkedList());
    return true;
  } // method advertex
```

In our design of the Graph class, vertices were stored as keys in a HashMap object. Each value field in the HashMap object was an adjacency list of vertex– weight pairs, specifically, a linked list of the neighbors (and edge weights) of the vertex key. So here if the vertex is not already in this Graph, add it to the Graph.Otherwise skip.

Adding an edge <v1, v2> to the Graph class requires adding the destination vertex and the weight <v2, weight> to the LinkedList object associated with v1.

```
public boolean addEdge (Vertex v1, Vertex  v2, double
weight)
{
   addVertex (v1);
   addVertex (v2);
    Edge e = new Edge (v2, weight);
    ((LinkedList)vertexMap.get (v1)).add (e);
    return true;
}
```

6. Isedge Method

To determine if the Graph object contains a given edge <v1, v2> we iterate over v1's associated LinkedList object searching for a vertex v2. If found the true is returned. Otherwise, false is returned.

```
Public Boolean isEdge(Vertex v1,Vertex v2)
 {
  if(! (vertexMap.containsKey (v1) &&
  vertexMap.containsKey (v2)))
   return false;
   Iterator itr= ((LinkedList)vertexMap.get(v1)).iterator();
while (itr.hasNext())
if(((Edge)itr.next()).getToVertex().equals (v2))
 return true;
 return false; }
```

7. getEdgeWeight Method

To determine the weight of the given edge <v1, v2> we iterate over v1's associated LinkedList object searching for a vertex v2. If found the weight associated with this vertex is returned. Otherwise, -1.0 is returned, which indicates <v1,v2> is not an edge in the Graph object.

```
public double getEdgeWeight (Vertex v1, Vertex v2) {

 if(!(vertexMap.containsKey(v1)&&  vertexMap.containsKey
(v2)))
 return -1;
LinkedList list = (LinkedList)(vertexMap.get (v1));
 Iterator itr = list.iterator();
   while (itr.hasNext()) {
   Edge e = (Edge)(itr.next());
   if (e.getToVertex().equals (v2))
   return e.getWeight();
  } // while
       return -1.0;  // there is no edge <v1, v2>   }
```

8. clearMarks Method

Before we start implementing our two algorithms, for all vertices in the Graph we must set the field **mark** in the vertex class to false to indicate they are not yet visited.

```
  private void clearMarks ()
     {
      Iterator itr= vertexMap.keySet().iterator();
      while (itr.hasNext())
      {
        Vertex vertex = (Vertex)(itr.next());
        vertex.clear();
       }//while       }
```

Now we are ready to implement our algorithms; nearest neighbor algorithm and minimum spanning tree.

9. getmin Method

In this method we iterate over a given vertex's neighbors, pick out the edge that has the minimum weight and that hasn't been visited, if found return it and mark it as visited. We will use this method in the theNearestNeighbourAlgorithm() method to select what vertex we go to next .

```
  public Edge getmin(Vertex  from) {
     Edge current;
     LinkedList l;
     if (!vertexMap.containsKey (from) )
        return null;
     l= ((LinkedList)vertexMap.get (from));
     if (l!=null){
  int index=0;
  Edge min =(Edge)l.get(index);
  while (min.getToVertex().mark == true){
  index++;
  min =(Edge)l.get(index);
  }
```

```
Iterator itr =((LinkedList)vertexMap.get
(from)).iterator();
while (itr.hasNext()){
current = (Edge)itr.next();
if(current.getToVertex().mark == false){
if (current.getWeight()<min.getWeight())
  {
   min=current;     }
}
}
min.getToVertex().mark =true;
return min    ;      }
return null; }
```

10. theNearestNeighbourAlgorithm Method

The start vertex in the tour is a parameter in this method .To calculate the time to perform this method System.currentTimeMillis() will be used at the beginning and the end of this method.

We start by inserting the start vertex v in an empty LinkedList object and then we mark it as visited. Then we iterate over v's neighbors to pick out the smallest edge that is connected to this vertex using getMin() method .If found take the end point (vertex) of this edge by getTovertex() method and add this vertex to the LinkedList object .Loop until LinkedList has as many vertices as the original Graph. Now go back to the start vertex and add it to the end of LinkedList and return the LinkedList object.

```
public LinkedList theNearestNeighbourAlgorithm(String s) {
   vertex start=getv(s);
   long beginTime = System.currentTimeMillis(); //...Get the
start time...
        LinkedList cycle = new LinkedList();
        Edge e;
        Vertex v;
        clearMarks ();
        if (!vertexMap.containsKey (start) )
          return null;
        cycle.add (start);
        start.mark=true;
        v = start;
   while (cycle.size() < size())        {
            e=getmin(v);
            if (e!=null)          {
              v=e.getToVertex();
              cycle.add (v);
             } //if
          } //while
 cycle.add (start);
long endTime=System.currentTimeMillis();//Get the end  time
elapsedTime1=(double)(endTime- beginTime) / (double)1000;
 return cycle;    }
```

11.getMinimumSpanning Tree Method

As we indicated earlier we will save the edge triple < v, w, weight> in a priority queue, where weight is the f the edge connecting the two vertices. The root of the tree is the start vertex or point in the tour. First we iterate over the root's neighbors; for each neighbor w, we add the edge triple <root, w, weight> to the priority queue. Then, until the tree has as many vertices as the original Graph, we remove an edge <x ,y, weight> that has the smallest weight from the priority queue .

If y is not already in tree, we add y and edge <x ,y> to tree and for each neighbor z of y ,if z is not already in tree we add the edge triple <y ,z, weight> to the priority queue

```
public Graph getMinimumSpanningTree(Vertex root) {
        Graph tree = new Graph();
         PriorityQueue pq = new Heap();
         Edge e;
         EdgeTriple edge;
         Vertex w,x,y,z;
         Iterator itr,itr1;
         double weight;
         if (isEmpty())
            return null;
        tree.addVertex (root);
         itr= ((LinkedList)vertexMap.get (root)).iterator();
         while (itr.hasNext()) {
            e = (Edge)itr.next();
            w = e.getToVertex();
            weight = e.getWeight();
            edge = new EdgeTriple (root, w, weight);
            pq.add (edge);
         } // adding root's edges to pq
         while (tree.size() < size()) {
            edge = (EdgeTriple)pq.removeMin();
            x = edge.getFromVertex();
            y = edge.getToVertex();
            weight = edge.getWeight();
            if (!tree.containsVertex (y)) {
               tree.addVertex (y);
               tree.addEdge (x, y, weight);
               itr1 = ((LinkedList)vertexMap.get (y)).iterator();
               while (itr1.hasNext()) {
                  e = (Edge)itr1.next();
                  z = e.getToVertex();
                  if (!tree.containsVertex (z)) {
                     weight = e.getWeight();
                     edge = new EdgeTriple (y, z, weight);
                     pq.add (edge);
                  } // z not already in tree
               } // iterating over y's neighbors
            } // y not already in tree }
           return tree;
      } // method getMinimumSpanningTree
```

12.DFS Method

The key for the DFS is being able to find the vertices that are unvisited and adjacent to a specified vertex. By iterating over the vertex's neighbors, pick out the adjacent vertex and then check whether this vertex is unvisited. If so you've found what you want – the next vertex to visit. We put the code for

this process in the getAdjUnvisitedVertex () method. In this method we make the subGraph a HashMap object that holds the Minimum Spanning Tree for the Graph from specified vertex (starting point). Then iterate over this vertex's neighbors and return the adjacent vertex that has not been visited.

```
public Vertex getAdjUnvisitedVertex(Vertex v)
{
HashMap
 subGraph=getMinimumSpanningTree(v).vertexMap;
LinkedList edgeList = (LinkedList)subGraph.get (v);
Iterator itr = edgeList.iterator();
while (itr.hasNext()) {
   Edge e = (Edge)itr.next();
   Vertex to = e.getToVertex();
    if(!to.mark)
     return to;
     }
           return null;
      }// end getAdjUnvisitedVertex
 Stack stack = new Stack();
Vertex current;
 LinkedList path= new LinkedList();
clearMarks ();
 stack = new Stack();
stack.push (start);    // begin at vertex start
start.mark=true;       // mark it
path.add(start);    // add it to the path
while( !stack.isEmpty() )      // until stack empty,
   {
    // get an unvisited vertex adjacent to stack top
    Vertex v = getAdjUnvisitedVertex( (Vertex)stack.peek() );
    if(v == null)                  // if no such vertex,
    stack.pop();
    else                        // if it exists,
    {
     v.mark = true;  // mark it
     stack.push(v); // push it
   path.add(v); //add it to the path
    }
   }  // end while
path.add(start); //add the start node to complete the path   long
endTime = System.currentTimeMillis(); //...Get the end
          time...
elapsedTime2=(double)(endTime- beginTime) / (double)1000;
          return path;
   }
```

Now we are ready for the dfs () method of the minimum spanning tree. In this method we use a Stack object, with push, peek, and pop methods.

Push the start vertex in the stack and mark it as visited and in the same time store it in a LinkedList (to make the tour).

We examine the vertex at the top of the Stack, using peek (), and try to find unvisited neighbor of this vertex using getAdjUnvisitedVertex (). If it doesn't find one pops the stack .If it finds such a vertex visit it and pushes it onto the stack and stores it in the LinkedList. Then loop until the stack is empty. Now back to the start vertex to complete the tour.

```
public LinkedList dfs(String d)  // depth-first search
{
long beginTime = System.currentTimeMillis(); //...Get the
start time...
Vertex start=getv(d);
   }
```

To calculate the total length of the tour by nearest neighbor and minimum spanning tree algorithms we use similar methods. In both of them we call the collection that contains the specified algorithm. To store the total length of the tour we declare a variable of type double and we loop over the elements in that collection. During the loop and for each pair of these elements we check whether they form an edge on the Graph class, using is Edge () method .If so calculate the sum of the weight of these edges using getEdgeWeight () method.

```
   public  double getdistanceNearst(String p)  {
         LinkedList s=theNearestNeighbourAlgorithm(p);
         double d=0;
         int index=0;
         String n="";
         while (index<s.size()-1){
            Vertex vx =(Vertex)s.get(index);
            Vertex w =(Vertex)s.get(index+1);
            if (isEdge(vx,w)){
                d+=getEdgeWeight(vx,w);
                index++;
            }
         }
         return d;
         }
   public double getdistanceSpanning(String p)
      {
         LinkedList s=dfs(p);
         double distance=0;
         int index=0;
         while (index<s.size()-1)
     {
            Vertex vx =(Vertex)s.get(index);
            Vertex w =(Vertex)s.get(index+1);
            if (isEdge(vx,w)){
             distance+=getEdgeWeight(vx,w);
index++;
      }
                }
         return distance;
      }
```

13.Comparison Method

Finally this method is used to make a comparison on the performance of our two algorithms, in terms of the total length of the tour and the time elapsed.

```
public String comparison (String p)
   {
    String com="";
    double distance1=0;
     double distance2=0;
    distance1=getdistanceNearst(p);
    distance2=getdistance(p);
    if (distance1<distance2)
 com="The  Nearest Neighbour is better  than Minmum Spanning Tree by Factor: "+(distance2-distance1);
    else if(distance1>distance2)
 com="Minmum  Spanning  Tree  is  better  than  Nearest Neighbourby Factor: "+(distance1-distance2);
    else com="They are Equal";
    return com;
    }
```

B. User Interface

The main part for implementing any application is to implement its Human – computer Interface efficiently. A GUI, that is simple and convenient to use, results in an efficient and user – friendly application. The class TSP contains all feature of GUI, which builds the front end for the user to interact with this application. Use the following swing components:

1- Panels: these are swing objects and they are used in our system to frame the various other swing objects together.
2- Labels: these are used to describe various functions in the system.
3- Command Buttons: these buttons are not toggled on and off, but instead act as "push" buttons. When user presses the enter key, or click on a button that has the focus an event is fired that can be caught by an action listener associated with the button this listener performs the action associated with the command.
4- JTextArea is used as display area to show the result of queries.
5- ComboBoxes: a combo box is a special text field with a drop down list .the text field displays the currently selected from the list. This list appears when the user clicks the down arrow displayed in the text field.

C. Testing

This section deals with testing the system and the stages involved in performing the test. There are two main stages in testing this system, Unit Testing and Integration Testing.

1. Unit Testing

This stage is concerned with testing the individual components (classes or methods) of system in isolation. Most of the problems that we faced, was in the Graph class .In terms of what is the appropriate data structure that can be used to represent edges in the form of adjacency list; which is usually represented by LinkedList object and how can we map each vertex to its LinkedList. We tried several data structure such as ArrayList, Vector and others. And finally we came up with the HashMap data structure for speed.

2. Integration Testing

This stage tests the complete application. It is done by applying the application on three test graphs (in our case) that represent the instance of the TSP of size; 5-city, 10-city, and 29-city. The programmer is sure that the system will perform as expected and the required functionality is there. The users are now given the chance to test the application as the following:

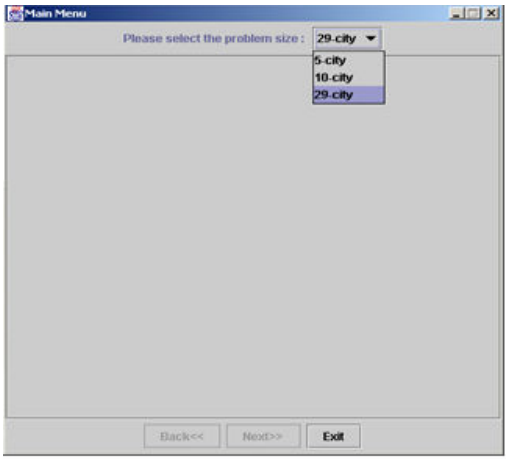

Fig. 11 Choosing the problem size

This is the first screen the user will see .It allow the user to choose what problem size would he like to implement and apply the two algorithms upon them. Comobox Model does this task. Then a list of the city in the tour will be displayed on the screen using JtextAera.

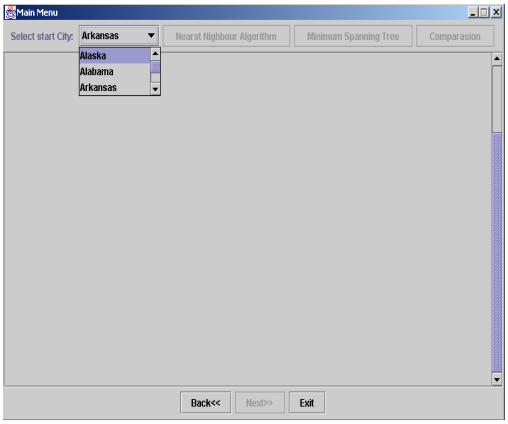

Fig. 12 The selecting of the start city in the tour

The first component in this screen is to select the start city in tour. Now you ready to perform any of two algorithms by clicking in the specific button for each algorithm. In the both case the tour solved by the algorithm will be displayed.

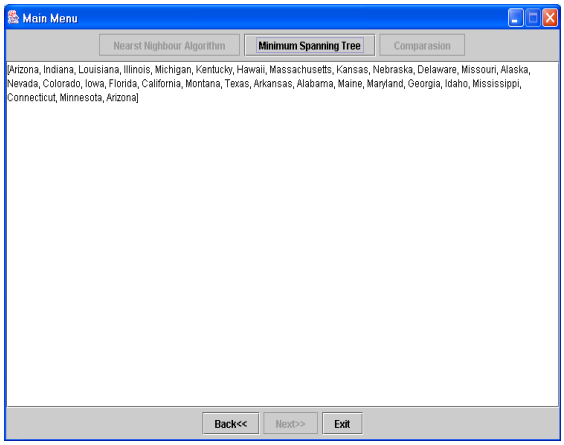

Fig. 13 The tour solved by Nearest Neighbor Algorithm

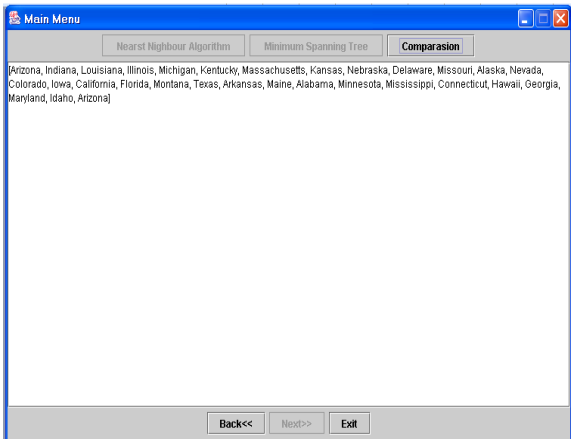

Fig. 14 The tour solved by Minimum Spanning Tree

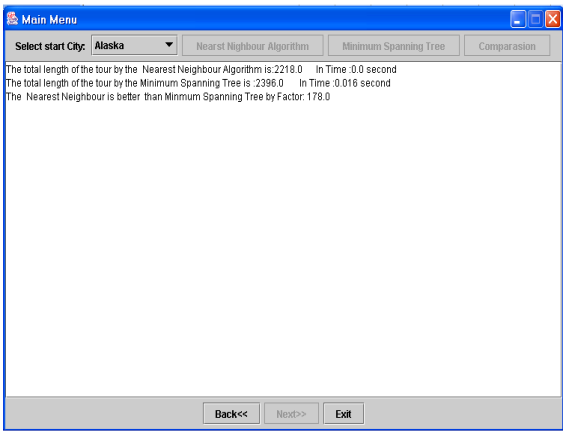

Fig. 15 The comparison of the performance for the Nearest Neighbor Algorithm and Minimum Spanning Tree

This screen displays the comparison of the performance for the both algorithms, in terms of the total length of the tour, and the time elapsed.

D. Results

As we mentioned in the last section we applied the application on three graphs that making a TSP tour instance size of 5-city, 10 –city, and 29 –city. Table I, II, III show results of the performance of the two algorithms:

1.Problem size: 5 – city

TABLE I
COMPARISON OF TSP ALGORITHM

Start point	Nearest neighbor algorithm		Minimum spanning tree	
	Tour length	Time/second	Tour length	Time/second
New York	6189.0	0	6189.0	0
Los Angeles	6196.0	0	6196.0	0
Seattle	6100.0	0	6100.0	0
Chicago	6100.0	0	6100.0	0
Boston	6196.0	0	6196.0	0

Results show that in all of the cases the performance of the Nearest Neighbor Algorithm and Minimum Spanning Tree are equally in terms of the total length of the tour and the time.

2.Problem size: 10- city:

TABLE II
COMPARISON OF TSP ALGORITHM

Start point	Nearest neighbor algorithm		Minimum spanning tree	
	Tour length	Time	Tour length	Time/second
Dublin	2610.0	0	2858.0	0.01
Cork	2577.0	0	2653.0	0.01
Galway	2805.0	0	2858.0	0.01
Limerick	2570.0	0	2653.0	0.01
Bray	2471.0	0	2606.0	0.01
Drogheda	2610.0	0	2482.0	0.01
Dundalk	2709.0	0	2808.0	0.01
Strabane	2683.0	0	2540.0	0.01
Westport	2723.0	0	2531.0	0.01
killarney	3267.0	0	2531.0	0.011

The results table show that the “Minimum Spanning Tree” approach is worse than “Nearest Neighbour Algorithm” both in time elapsed and in the total length of the tour, except for four cities; Drogheda, Strabane, Westport, killarney.

3.Problem size: 29- city

TABLE III
COMPARISON OF TSP ALGORITHM

Start point	Nearest neighbor algorithm		Minimum spanning tree	
	Tour length	Time/second	Tour length	Time/second
Alaska	2218.0	0	2396.0	0.09
Alabama	2338.0	0	2512.0	0.09
Arkansas	2382.0	0	2509.0	0.1
Arizona	2281.0	0	2527.0	0.08
California	2460.0	0	2594.0	0.09
Colorado	2224.0	0	2594.0	0.09
Connecticut	2431.0	0	2451.0	0.09
Delaware	2200.0	0	2380.0	0.09
Florida	2379.0	0	2600.0	0.09
Georgia	2367.0	0	2507.0	0.09
Hawaii	2353.0	0	2519.0	0.09
Iowa	2468.0	0	2595.0	0.09
Idaho	2305.0	0	2507.0	0.09
Illinois	2266.0	0	2352.0	0.09
Indiana	2306.0	0	2519.0	0.1
Kansas	2139.0	0	2536.0	0.09
Kentucky	2266.0	0	2352.0	0.09
Louisiana	2355.0	0	2441.0	0.09
Massachusetts	2375.0	0	2530.0	0.09
Maryland	2300.0	0	2507.0	0.09
Maine	2367.0	0	2512.0	0.09
Michigan	2313.0	0	2399.0	0.09
Missouri	2534.0	0	2396.0	0.09
Mississippi	2348.0	0	2451.0	0.081
Montana	2454.0	0	2551.0	0.09
Nebraska	2553.0	0	2334.0	0.1
Nevada	2392.0	0	2557.0	0.1
Texas	2382.0	0	2509.0	0.1

Results show that in most of the cases the performance of the Nearest Neighbor Algorithm is better than and Minimum Spanning Tree ,both in the total length of the tour and the time elapsed, except for three cities; Minnesota, Missouri, Nebraska.

VIII. CONCLUSIONS

This paper implements two heuristic algorithms, namely the Nearest Neighbor Algorithm and Minimum spanning Tree for solving the traveling salesman problem. We have compared the performance of these algorithms. Results show that for small size of cities, the two algorithms are performing equally. With the increase in the number of the cities, the performance of Nearest Neighbor Algorithm seems to be better than Minimum spanning Tree in calculating the length of the tour. But the time complexity of Nearest Neighbor Algorithm is always lower than Minimum spanning Tree.

REFERENCES

[1] E. L Lawler, J. K. Lenstra, A. H. G.Rinnooy Kan, and D.B.Shmoys, editors. "The Traveling Salesman Problem", Wiley, 1985
[2] http://en.wikipedia.org/wiki/Traveling_salesman_problem
[3] Keld Helsgaun ,"An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic", Department of Computer Science, Roskilde University,DK-4000 Roskilde, Denmark
[4] Asia – Pasific Journal of Operational Research 18(2001) 77-87,Sim Kim LAU and Li-Yen SHUE, "solving traveling salesman problems with an intelligent search approach" .

Fatma A. Karkory received her M.Sc. in Computing Science from Griffith College Dublin, in 2004 and Bachelor in Computing Science from University of sabha in 1993. She is a lecturer at the Higher Institute of Refrigeration and Air Conditioning Sokna, Libya, and Lecturer collaborator in the Higher Institute of Comprehensive professions Aljufra at Sokna,libya .

Ali A. Abudalmola received his M.Sc. in Computing Science From Griffith College Dublin in 2004 and high diploma from the higher centre for general professions in Musrata in 1996 he is a lecturer at the Higher Institute of Comprehensive professions Aljufra at Sokna, Libya, and Lecturer collaborator at the Higher Institute of Refrigeration and Air Conditioning Sokna, Libya.