INB371 & INN371 – Data Structure and Algorithms

Assignment 2

The Travelling Salesman Problem

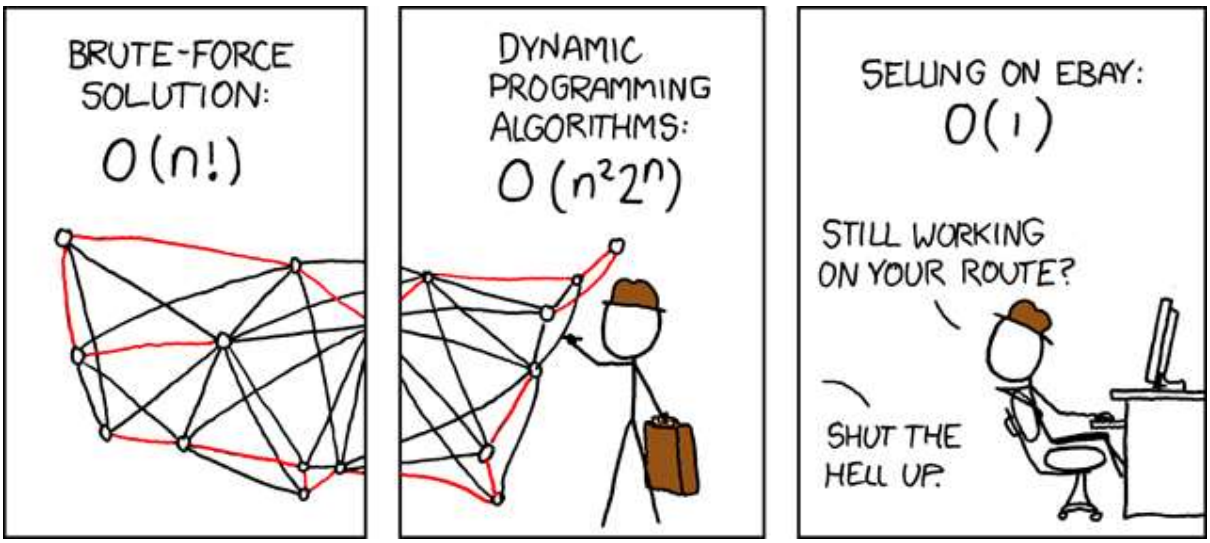

Image from: http://imgs.xkcd.com/comics/travelling_salesman_problem.png

Due Date: 11:59 pm – 4 June, 2013

Weighting: 30%

Individual Assignment

Introduction

The Travelling Salesman Problem (TSP) is one of the most studied problems in computer science.

The aim of the problem is to find the shortest route which visits all members of a collection of locations returning to the starting point. It has many uses in microchip manufacturing, operational planning and operational logistics. Microchip manufacturing starts with many logic gates that are connected to each other. A laser is used to cut the network of logic gates into many individual circuits. The tour finds the optimal order of cutting locations for a laser to move between to minimise the time taken to produce each chip.

An approach to finding the optimal route could be to go from the starting point to the closest point, then to the closest point to this, etc., but in general this does not yield the shortest route.

Since the problem was first studied in the 1930's various algorithms for determining the optimum solution have been developed and the number of locations for which an optimal solution can be found has increased to a record optimal 85,900 city tour. This record was set in 2006.

The problem is designated as an NP-hard problem (non-deterministically polynomial hard – a class of problems in computational complexity theory). There is no known polynomial time algorithm to solve the TSP with an optimal solution, i.e. a minimum distance route.

Note that the TSP is different to finding the shortest route between two locations. The TSP must find the shortest route that travels through all given locations on a map or grid.

The TSP can be modelled as an undirected weighted graph problem, such that the cities are the graph's vertices and the paths between the cities are the edges, with the distance of each path being the weight of the edge.

Optimal Solutions

A brute force solution works by enumerating all possible routes and keeping track of the minimum of all route distances. If there are four locations that must be visited, A, B, C, and D, the possible routes are:

A-B-C-D-A
A-B-D-C-A
A-C-B-D-A
A-C-D-B-A
A-D-B-C-A
A-D-C-B-A

This is a total of 6 routes for 4 locations. If there were five locations, the number of routes increases to 24.

A-B-C-D-E-A	A-C-B-D-E-A	A-D-B-C-E-A	A-E-B-C-D-A
A-B-C-E-D-A	A-C-B-E-D-A	A-D-B-E-C-A	A-E-B-D-C-A
A-B-D-C-E-A	A-C-D-B-E-A	A-D-C-B-E-A	A-E-C-B-D-A
A-B-D-E-C-A	A-C-D-E-B-A	A-D-C-E-B-A	A-E-C-D-B-A
A-B-E-C-D-A	A-C-E-B-D-A	A-D-E-B-C-A	A-E-D-B-C-A
A-B-E-D-C-A	A-C-E-D-B-A	A-D-E-C-B-A	A-E-D-C-B-A

For six locations, the number of routes is 120. This progression is $(N-1)!$. It can be noticed that the route A-B-C-D-A is essentially the same as the route A-D-C-B-A i.e. the reverse, and that each route enumerated above has a reverse. So for N locations, we can reduce the number of routes to $\frac{(N-1)!}{2}$ but the factorial value dominates the number of routes, so the overall time complexity is still $O(N!)$. Using the brute force approach, a maximum for N is approximately 10 to 12 for a solution in a reasonable amount of time.

Fortunately, algorithms exist which do not rely on the brute force approach. One such approach recognises that parts of the exhaustive list of tours are common between some tours. In the exhaustive list of routes for five cities above, each three city sub-tour e.g. B-C-

D is repeated in tours which would have different distances: A-B-C-D-E-A and A-E-B-C-D-A. As the number of cities increases, the number of repeated sub-tours increases as does the length of repeated sub-tours. One approach to solving the TSP optimally makes use of this insight and reduces the time complexity to $O(N^2 2^N)$ which is still exponential in growth, but better than $O(N!)$ allowing N to be increased to about 20 to 23 for a solution in a reasonable amount of time.

Non-optimal solutions

A number of approaches to solving the TSP have used an approximation technique to deliver non-optimal solutions. These approximation techniques can handle much larger data sets. There are a number of researchers working on discovering a tour of 1,904,711 cities across the world (http://www.tsp.gatech.edu/world/). Lower bounds on the tour are still being sought.

One of the non-optimal solutions was mentioned in the introduction, where the closest location to the current location is used at all way points. This is known as the nearest neighbour approach. Another approach using this technique is known as the repetitive nearest neighbour approach, where the nearest neighbour approach is applied N times starting from each of the N locations.

The approach to be taken in this assignment will result in a tour that is guaranteed to be no more than twice the length of the optimal tour. It sometimes finds the optimal tour but this cannot be guaranteed.

This approach starts by finding the Minimum Spanning Tree for the fully connected graph. Then any vertex is chosen as the starting point for a depth first traversal of the graph. The order of the visited vertices defines the path for the TSP tour and hence its length.

Task

Your program will do the following:

1. Randomly select a number of points on a Cartesian plane with x and y co-ordinates between 0 and 100 inclusive to be the cities for the TSP tour.
2. From the points, create a fully connected graph where the points are the vertices of the graph. A fully connected graph has an edge from each vertex to every other vertex. The edges will be undirected. Each edge weight will be the Euclidean distance (*d*) between the two vertices for the edge:

$$d = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$

3. ***Optional:*** The result for an optimal tour could be calculated.
    a. A recursive brute force solution has a limit of 10 to 12 vertices. *Implementing this solution if you cannot get the approximation solution working will earn some marks.*

b. A dynamic programming solution has a limit of around 20 to 23 vertices. *Implementing this algorithm for comparison with the approximate solution will enable you to earn the top marks for this assignment.*

Algorithms for these two optimal approaches are presented below.

4. Calculate an approximation of the distance for the TSP tour of the randomly generated cities using the Minimum Spanning Tree/Depth First Search algorithm presented below.
5. Output the results of the optimal and/or approximation of the tour.
6. The program must also be able to read test files from the command line (as for Assignment 1).

Design Decisions

The following design decisions may aid in your implementation. These guidelines do not have to be followed.

The solution that produces the above output was built by creating the following classes:

- **`Random`** – produces random integers
- **`Point`** – a two-dimensional representation of a Cartesian point with an x co-ordinate and a y co-ordinate
- **`Vertex`** – encapsulates a vertex identifier and an adjacency list for a vertex in a graph
- **`Edge`** – encapsulates information about a weighted undirected edge in a graph
- **`EdgeComparer`** – responsible for providing an ordering for Edge objects
- **`Graph`** – encapsulates a graph representation (adjacency matrix or adjacency list) along with functionality to determine optimal TSP tours, approximate TSP tours, the minimum spanning tree for a graph and to perform a depth first traversal of a graph.
- **`DisjointSet`** – A data structure used to perform Union-Find operations as required in Kruskal's Minimum Spanning Tree algorithm

The `Random` Class

This class is based on the exercises included in Workshops 3 and 4. To ensure that different random values are generated on successive runs, the **`Randomise()`** method should be called by the constructor.

The `Point` Class

This class encapsulates the *x* and *y* co-ordinates of a point in a Cartesian plane and has functionality to determine the distance between two instances of **`Point`** objects.

`Point(int, int)`	Constructor that sets the x and y co-ordinates for the **`Point`** object
`~Point()`	Destructor
`double DistanceTo(Point*)`	Returns the Euclidean distance between this **`Point`** and the **`Point*`** parameter to the function
`string ToString()`	Produces a string representation of this **`Point`**

The `Vertex` Class

This class encapsulates the **`identifier`** of a **`Vertex`** object and a collection of **`Vertex`** objects which are adjacent to this **`Vertex`**. The collection is only used to store the adjacent vertices for the Minimum Spanning Tree of the **`Graph`**.

`Vertex(int)`	Constructor which sets the vertex **`identifier`** (useful for indexing into collections of vertices)
`~Vertex()`	Destructor
`int GetId()`	Accessor for this vertex' **`identifier`**
`void AddAdjacency(Vertex*)`	Adds a pointer to a **`Vertex`** to this **`Vertex`**' adjacency list
`vector<Vertex*> GetAdjacencies()`	Returns a collection of pointer of **`Vertex`**, being the vertices adjacent to this **`Vertex`**
`string ToString()`	Returns a string representation of this **`Vertex`**. For testing purposes.

The `Edge` Class

This class encapsulates pointers to the source and destination **`Vertex`** objects and the **`weight`** of the **`Edge`**. An alternative would be to encapsulate **`Vertex`** identifiers (**`int`**s) rather than pointers to **`Vertex`**.

`Edge(Vertex*, Vertex*, double)`	Constructor which sets the **`source`** vertex, the **`destination`** vertex and the weight for this **`Edge`**
`~Edge()`	Destructor
`Vertex* GetSource()`	Returns a pointer to the **`source`** vertex
`Vertex* GetDestination()`	Returns a pointer to the **`destination`** vertex
`double GetWeight()`	Returns the weight of this **`Edge`**
`string ToString()`	Returns a string representation of this **`Edge`**. For testing purposes.

The `EdgeComparer` Class

This comparer is used to sort **`Edge`** objects in a collection which will be used by Kruskal's Minimum Spanning Tree algorithm.

`bool operator() (Edge*, Edge*)`	This function returns **`true`** if and only if the weight of the first **`Edge`** is less than the weight of the second **`Edge`**

The `Graph` Class

This class encapsulates a weighted undirected graph. Because the calculations require a complete graph, edge weights are best stored in an adjacency matrix (two dimensional array or vector of vectors of weights.

The edges required for calculating Kruskal's Minimum Spanning Tree need to be in sorted order. They can be stored in a vector and then sorted, or stored in a priority queue based on a minimum heap. In either case, the edges will require a comparer class to sort them.

The algorithm will result in storing only the edges in the minimum spanning tree. There will be |V| - 1 such edges, so these edges will be best stored in an adjacency list.

Public Functionality	
`Graph(int)`	Constructor which sets the number of vertices in this **`Graph`**
`~Graph()`	Destructor
`void AddVertex(Vertex*)`	Adds pointer to **`Vertex`** to the adjacency list for this **`Graph`**. *This function will be called by the driver program for initialisation purposes. The adjacency list will be used to store the edges in the Minimum Spanning Tree for this graph as calculated by the MinimumSpanningTree function.*
`Vertex* GetVertex(int)`	Accessor returns a pointer to the **`Vertex`** with the identifier/index in the parameter
`void AddEdge(Edge*)`	Adds pointer to **`Edge`** to the edge list for this **`Graph`**. *The AddEdge function will store the information for the edge in the underlying adjacency matrix AND also in a private Edge list that will be used by the MinimumSpanningTree function.*
`double OptimalTSP()`	Returns the length of the optimal TSP tour. *If NUM_CITIES > 11 the TSPDP function should be used. If NUM_CITIES > 22, this function should not be called.*
`double ApproximateTSP()`	Returns the length of the approximate TSP tour calculated using the given algorithm *(Find Minimum Spanning Tree and tour in visit order of the Depth First Search on the Minimum Spanning Tree.*

Private Functionality	
`double TSPBruteForce(int, bool*)`	Recursive brute force solution. Maximum size ~ 11.
`double TSPDP(int, int)`	Top down Dynamic Programming (DP) with memorisation solution. Maximum size ~20.
`void MinimumSpanningTree()`	Uses Kruskal's algorithm to find the Minimum Spanning Tree (MST) for this Graph. Stores the edges of the MST in the adjacency list of each Vertex
`double DepthFirstSearch()`	Returns the approximation for the TSP tour taken by traversing the vertices in depth first order of the sub-graph formed from the MST

The `DisjointSet` Class

This data structure is used for very efficient look-up (Find) to determine which set an element is in. It also provides a very efficient way to join (Union) two sets together.

It is based on arrays/vectors.

`DisjointSet(int)`	Constructor which sets the **`size`** of this **`DisjointSet`**
`~DisjointSet()`	Destructor
`int Find(int)`	Returns the **`index`** of the parent set of the element in the parameter
`void Union(int, int)`	Creates the union of two disjoint sets whose **`index`**es are passed as parameters
`bool SameComponent(int, int)`	Returns **`true`** if the two **`index`**es passed as parameters are in the same set

The Brute Force Optimal Algorithm

The Brute Force Optimal algorithm has a limit of about 10 to 12 vertices. It computes every combination of tour through all vertices finding the minimum. It relies on recursion and backtracking to find the solution.

The status of vertices that have been visited is kept in an array of bool. This array gets copied for each recursive call to indicate progress. The tour starts from vertex 0.

The edge weights for the graph are stored in an adjacency matrix named **weights**

Algorithm

```
create array of bool named visited with size = numVertices
initialise visited to false
set first element of visited to true
result ← TSPBruteForce(0, visited)

TSPBruteForce(current, visited):
   if all elements in visited are true
      return distance from current to 0

   make a copy of visited
   set minDistance to INFINITY
   for adjacent from 0 to numVertices-1
       if current != adjacent AND adjacent not visited
          set adjacent in copy of visited to true
          dist ← distance from current to adjacent +
                      TSPBruteForce(adjacent, copy of visited)
          minDistance ← minimum(minDistance, dist)
          set adjacent in copy of visited to false
       end if
   end for
   return minDistance
end
```

The Dynamic Programming Optimal Algorithm

This algorithm uses a similar approach to the Brute Force algorithm but uses a problem solving approach know as ***Dynamic Programming*** to store optimal solutions to sub-problems of the larger problem being solved. The given algorithm uses recursion and a top down approach. The technique of storing optimal solutions to sub-problems is known as ***memoisation***.

This approach runs much faster than the brute force solution but does so at the cost of the space used for storing the sub-problem solutions. It has a limit of approximately 20 to 23 vertices depending on amount of memory and architecture.

To store details of which cities have been visited, a **Bit Set** represented as an integer, called a **bitmask,** is used instead of an array of **`bool`**. If we have **`N`** cities, we use a binary integer of length **`N`** to indicate the visited cities. If bit *i* is '1' (on) we say that city *i* is in the visited set. If bit *i* is '0' (off), the city has not yet been visited. For example, if:

`bitmask =` 19_{10} `=` 10011_2

this implies that cities 0, 1, and 4 are in the visited set. Note that indexes start from the right (the least significant bit).

To check if bit *i* is on or off we can use bitwise manipulation operations. The **`&`** operator does a bitwise **`and`** operation between two values. The **`|`** operator does a bitwise **`or`** operation between two values. The **`<<`** operator is a bitshift left operator. It moves the bits in a value a given number of positions to the left. The result of $\mathbf{00001_2}$ **`<< 2 is`** $\mathbf{00100_2}$. This operation essentially multiplies the initial value by 2.

If the result of

```
bitmask & (1 << i)
```

is 0, the i^{th} bit is off (not in the visited set), if it is 1, it is on (in the visited set).

To set bit *i*, we can use

```
bitmask = bitmask | (1 << i)
```

Optimal sub-problem solutions must be stored in a memoisation table. The size of this table has to be **`N x (1 << N)`**. The data in the table should be set to an initial value indicating that the particular memoisation cell has not been set e.g. **`-1.`**

Algorithm

```
Create memoisation table [N][(1 << N)]
Set all values in table to UNSET

result ← TSPDP(0, 1)

TSPDP(current, bitmask):
   if table[current][bitmask] is not UNSET
      return table[current][bitmask]

   if bitmask == ((1 << N) - 1)
      return distance from current to 0

   minDistance ← INFINTY
   for adjacent from 0 to N-1
      if current != adjacent AND
                    ((bitmask & (1 << adjacent)) == 0)
         dist ← distance from current to adjacent +
                 TSPDP(adjacent, (bitmask | (1 << adjacent)))
         minDistance ← min(minDistance, dist)
      endif
   endfor
   table[current][bitmask] = minDistance
   return minDistance
end
```

The Approximation Algorithm

This algorithm relies on two well-known graph algorithms (Minimum Spanning Tree and Depth First Traversal) to provide an approximation to the TSP.

Algorithm

```
Calculate the Minimum Spanning Tree (MST) for the graph
Traverse the MST from the first vertex depth first
   Calculate distance in visit order


DFS:
   dist ← 0
   initialise visited array to false
   create empty stack
   push vertex 0 onto stack
   mark vertex 0 visited

   let current vertex ← NULL
   let previous vertex ← NULL

   while stack not empty
      current ← pop from stack
      if previous not NULL
         dist ← dist + distance from previous to current
      end if
      for vertices adjacent to current
         if adjacent vertex not visited
            push adjacent vertex onto stack
            mark adjacent vertex visited
         end if
      end for
      previous ← current
   end while

   dist ← dist + distance from current to vertex 0
   return dist
end
```

Late Penalties

You must submit your assignment by the due date or a late penalty will apply as follows:

- One day late, 10% of the mark awarded is deducted
- Two days late, 20% of the mark awarded is deducted
- Three to four days late, 30% of the mark awarded is deducted
- Five to seven days late, 40% of the mark awarded is deducted
- Eight to ten days late, 50% of the mark awarded is deducted
- Greater than ten days late, 100% of the mark awarded is deducted

Academic Integrity

You must submit only your own work for the assignment. Your attention is drawn to QUT's rules on academic integrity and plagiarism (*Manual of Policies and Procedures,* Section C/5.3 *Academic Integrity* and Section E/2.1 *Student Code of Conduct*).