EE 356 **Assigned: Oct. 4, 2017**
Project 4 **Due: Oct. 16, 2017**

Travelling Salesman Problem using Genetic Programming

The travelling salesman problem (TSP) can be stated as:
A salesman is required to visit n cities where each city must be visited exactly once before he returns to the starting city. If the cost is proportional to the distance between cities, what is the least-cost route the salesman can take.

This problem has been worked on since at least the 1800s with a general form of the problem being worked on at length by mathematicians at Harvard in the 1930s.

A brute force solution would require that we determine the total distance of all possible routes. Say, for example that there are just five cities. If the salesman starts in city 1 then he has four choices for his first trip, three for his second, two for his third, and just one choice for his fourth trip which gets him back to his starting point. To generalize, if there are n cities there are (n-1)! possible paths. If n is small, this problem is solvable in an optimal way but as n grows n! grows very rapidly and for a large number of cities the brute force solution is computationally impossible even for the fastest super computer. Consider that $99! = 9.332 \times 10^{155}$.

We can arrive at a reasonably good approximation to the optimal solution using genetic programming which works like this:

- If we have say 100 cities there are 99 distances we have to calculate for each path. Generate say 1000 random paths, find the total distance of each path, and sort them with the shortest path first. Call these 1000 random paths generation 0.
- For the next generation keep say, the 1% (or 10 in this case) of the paths which are the shortest from the previous generation and generate the next 99% (or 990) by randomly choosing one of the 10, and replacing a random subsection of the path with a random walk through the remaining cities. In pseudocode this looks like the following:

```
Generate and display 100 random cities
Generate 1000 random paths
N = number of generations from user
for i = 1 to N
  {Sort and keep the 10 shortest paths
   for p = 1 to 990
     Choose a random path from one of the 10 shortest
     Choose a random city from this path and delete all
       of the later cities
     Add the deleted cities back to the path by
       choosing them in a random order.
   end
end
```

Write a WPF program to work on the Traveling Salesman problem that has the following features:

- Provides a GUI interface to show the cities and the path through them. Use a small circle for each city and a filled in circle for the first (and last)city.
- Provide controls so that the user can choose the following options:
  - Number of cities
  - Number of paths per generation
  - Percent of paths that are used to generate the next generation.
- Provide a status box that gives the shortest path length.
- Provide a progress bar that shows the programs progress.
- Your program must run on a background worker thread so that the screen remains active while the program is running.

Extras:

1. About/Help box
2. In addition to a progress bar periodically show the cities and the shortest path so the user can see the path converging.
3. Allow the user to choose the start city using a mouse.
4. Do bounds checking on all of the input data.
5. Show the total execution time.

Two classes which *may* be useful for this project at the *City* class and *Path* class shown below. The *City* class has an x and a y location for the city and a Boolean variable called *visited* which is true if the city has been visited by the salesman. This class implements properties X, Y, and Visited to allow access to these variables. The City class also has an overloaded minus sign which allows you to subtract two cities and get the distance between them. The *Path* class has just two private variables which keep track of the path and its length. The path is stored in an `int` array called *p*. Each element of *p* has an index to a city. The path length is stored as a double. Both *p* and the path length are available as properties.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace TravellingSalesmanWPF
   {class City
    {public City()
       {x = 0;
        y = 0;
        visited = false;
       }
     public City(double x1, double y1)
       {x = x1;
        y = y1;
        visited = false;
       }
     private double x;
     private double y;
     bool visited;
     //Properties
     public  double X
       {get
          {return x;
          }
       }
     public double Y
       {get
          {return y;
          }
       }
     public bool Visited
       {get; set;}
     //Public methods
     public static double operator- (City c1, City c2)
       {return (Math.Sqrt((c1.x - c2.x)*(c1.x - c2.x) +
                          (c1.y - c2.y)*(c1.y - c2.y)));
       }
    }
 }
```

Note: `NumCities` must be a public variable giving the total number of cities in the main program, here referred to a MainWindow. You may want to change this to a private variable that is passed in with the constructor.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace TravellingSalesmanWPF
   {class Path
      {public Path()
        {int i;
         for(i=0;i<MainWindow.NumCities;i++)
           p[i] = i;
         pLength = 0;
        }
       private int [] p = new int[MainWindow.NumCities];
       private double pLength;
       //
       public int []P
       {get
         {return p;
         }
        set
         {p = value;
         }
       }
     public double PathLength
       {get
          {return pLength;
          }
        set
          {pLength = value;
          }
       }
    }
 }
```

Note that when you create an array of items in a class such as:

```
c = new City[numCities];                //Create a new city array
```

You are only allocating memory for the array. You are not running the constructor. To run the constructor you must write a loop to invoke the constructor for every item in the array. In the code below the line:

```
c[i] = new City(x, y);                //Instantiate the city
```

instantiates the items in the array. In this code segment rCities creates a random int from the Random class. It forces the cities out of the screen center but allows them to be random. In this case, your solution will most likely be the case where the fewest paths cross the center.

```
{InitializeComponent();
       int i;
       double x, y, xMax, yMax;
       xMax = imgWorld.Width;
       yMax = imgWorld.Height;
       numCities = DEF_NUM_CITIES;            //Set the number of cities as default
       c = new City[numCities];               //Create a new city array
       for(i=0;i<numCities;i++)               //For each city find a random location
           {x = rCities.Next(2, (int)xMax-4);              //  away from the edge by 4 pixels
            y = rCities.Next(2, (int)yMax-4);
            while(((x-xMax/2)*(x-xMax/2) + (y-yMax/2)*(y-yMax/2)) < 15000)
              {x = rCities.Next(2, (int)xMax-4);           // This loop forces x and y
               y = rCities.Next(2, (int)yMax-4);           //  out of the center
              }
           c[i] = new City(x, y);              //Instantiate the city
           randomGenes = false;
          }
       pShortest = new Path();              //Create a new version of the shortest path

     }
```