COIS 4050H – Assignment #4
Simon Willshire (0491272)

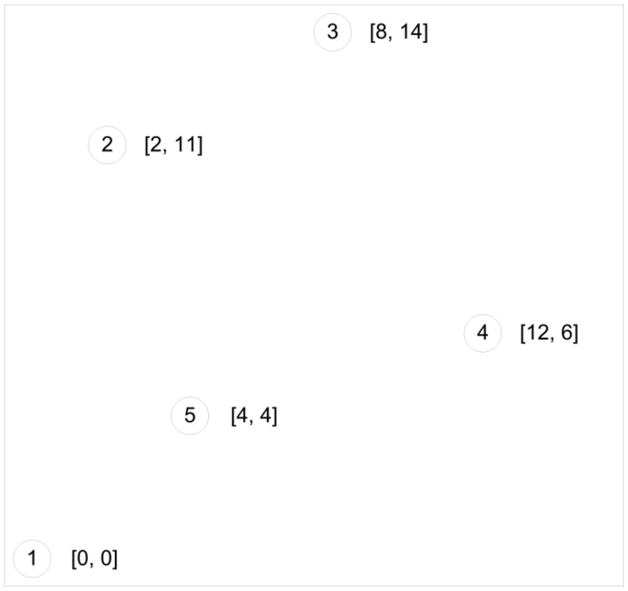
Euclidian Traveling Salesperson Problem (ETSP)

The Comparison of Optimum and Approximate Solutions

1.

For the contents of this assignment, I have used static values for city locations, and **assumed city 1 at [0, 0] is the starting/ending position**. I have chosen to do this so that I can work out the correct solution on paper to prove that the program is indeed correct, rounding errors permitting.

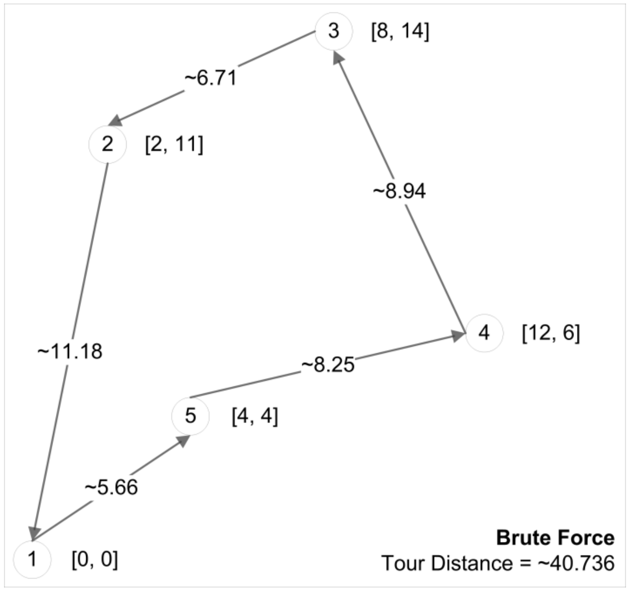
Other considerations:
- I will use Prim's Algorithm for MST
- I will use a non-recursive DFS using a stack
- I will use a brute force solution to show the true optimal solution.



2.1 The Optimal Algorithm

To solve the ETSP, the most obvious answer is to compare all permutations and see which is the most optimal (Brute force solution). This approach is still roughly $O(n!)$, where n represents the number of cities/locations in the problem. So in this case we have $n = 5: 120$ permutations to check. If we know the starting location, we can assume $O((n-1)!)$, so a much smaller 24 permutation.

After rummaging with some paper for a while, I came across the solution located on the right, with a total tour distance of **40.736**.



2.2 Optimal Implementation in C#

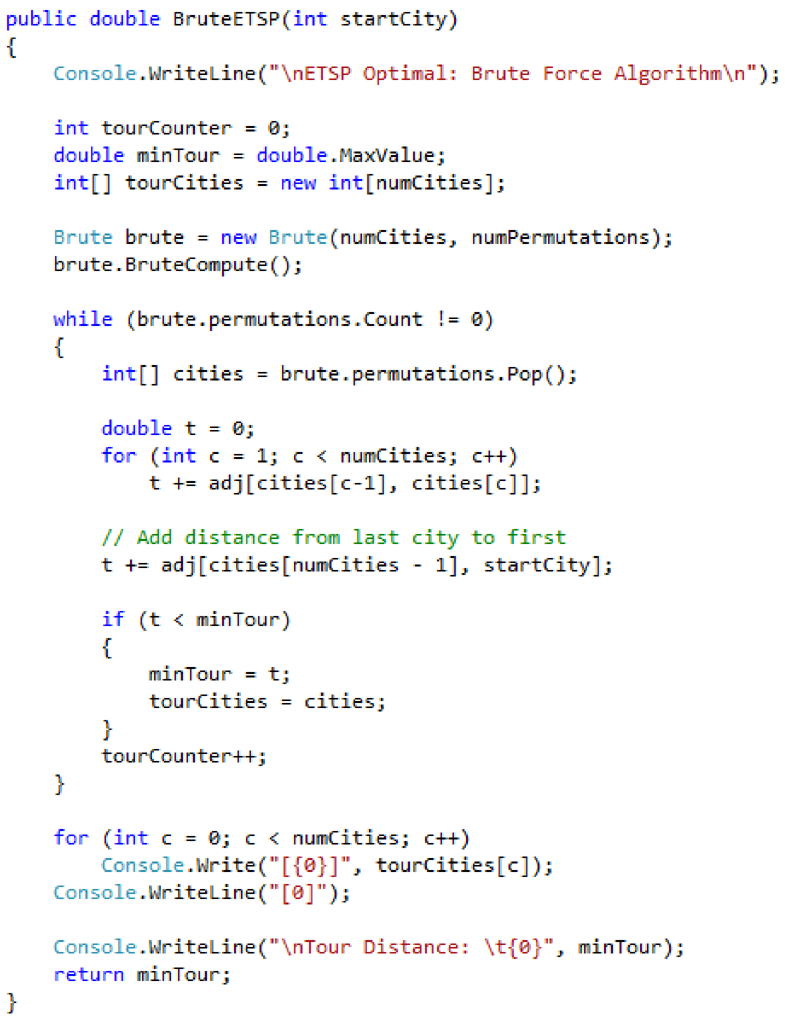
To implement the brute force approach, I needed to enumerate all of the permutations of the cities. Also note: permutations must begin and end with the starting position, 0. This means that we only need to find combinations ranging from 1 to 4 and insert them between 0s.

Permutations: {0, w, x, y, x, 0}

After all permutations have been found, we sum the distance between each of the cities to find a total tour length. We then find the minimum tour length to find the optimum solution.

The code segment on the right is the function which is called from the main ETSP class to return the optimum length using the brute force algorithm.

See the next page for the full Brute force class code.



```
public double BruteETSP(int startCity)
{
    Console.WriteLine("\nETSP Optimal: Brute Force Algorithm\n");

    int tourCounter = 0;
    double minTour = double.MaxValue;
    int[] tourCities = new int[numCities];

    Brute brute = new Brute(numCities, numPermutations);
    brute.BruteCompute();

    while (brute.permutations.Count != 0)
    {
        int[] cities = brute.permutations.Pop();

        double t = 0;
        for (int c = 1; c < numCities; c++)
            t += adj[cities[c-1], cities[c]];

        // Add distance from last city to first
        t += adj[cities[numCities - 1], startCity];

        if (t < minTour)
        {
            minTour = t;
            tourCities = cities;
        }
        tourCounter++;
    }

    for (int c = 0; c < numCities; c++)
        Console.Write("[{0}]", tourCities[c]);
    Console.WriteLine("[0]");

    Console.WriteLine("\nTour Distance: \t{0}", minTour);
    return minTour;
}
```

```
public class Brute
{
    public Brute(int nCities, int nPerms)
    {
        permutations = new Stack<int[]>();
        numCities = nCities;
        numPerms = nPerms;
    }

    public void BruteCompute()
    {
        int[] c = new int[numCities - 1];
        for (int i = 1; i < numCities; i++)
            c[i - 1] = i;

        while (NextPermutation(c)) { }
    }

    // Using Knuth's Lexicographic Algorithm,
    // Reference: http://42studios.com/2013/07/lexicographic-permutations/
    private bool NextPermutation(int[] numList)
    {
        var largestIndex = -1;
        for (var i = numList.Length - 2; i >= 0; i--)
        {
            if (numList[i] < numList[i + 1])
            {
                largestIndex = i;
                break;
            }
        }

        if (largestIndex < 0) return false;

        var largestIndex2 = -1;
        for (var i = numList.Length - 1; i >= 0; i--)
        {
            if (numList[largestIndex] < numList[i])
            {
                largestIndex2 = i;
                break;
            }
        }

        var tmp = numList[largestIndex];
        numList[largestIndex] = numList[largestIndex2];
        numList[largestIndex2] = tmp;

        for (int i = largestIndex + 1, j = numList.Length - 1; i < j; i++, j--)
        {
            tmp = numList[i];
            numList[i] = numList[j];
            numList[j] = tmp;
        }

        int[] n = new int[numCities];
        for (int p = 1; p < numCities; p++)
            n[p] = numList[p - 1];

        permutations.Push(n);

        return true;
    }

    private int numCities;
    private int numPerms;
    public Stack<int[]> permutations;
}
```

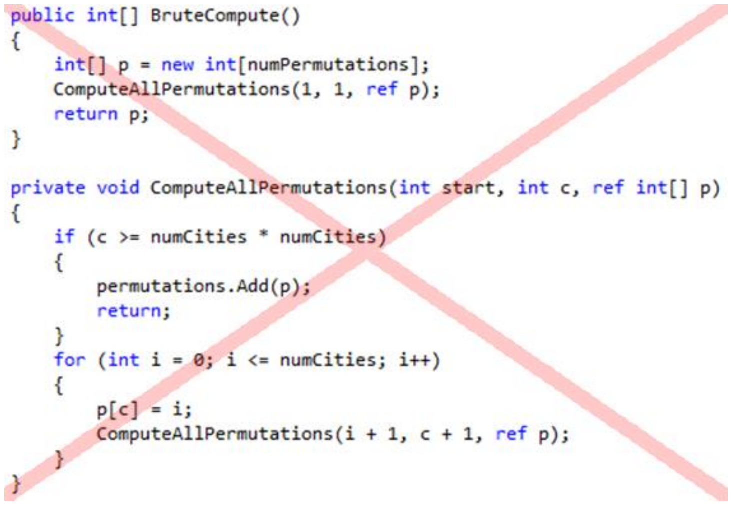
Note: I used Knuth's Lexicographic Permutation algorithm to generate the next permutation given a series of numbers.

BruteCompute() runs *NextPermutation*() until there are no more permutations to generate.

NextPermutation() uses Knuth's Algorithm to generate the next permutation in the series, and then adds it to the permutation stack.

2.2 Testing & Output

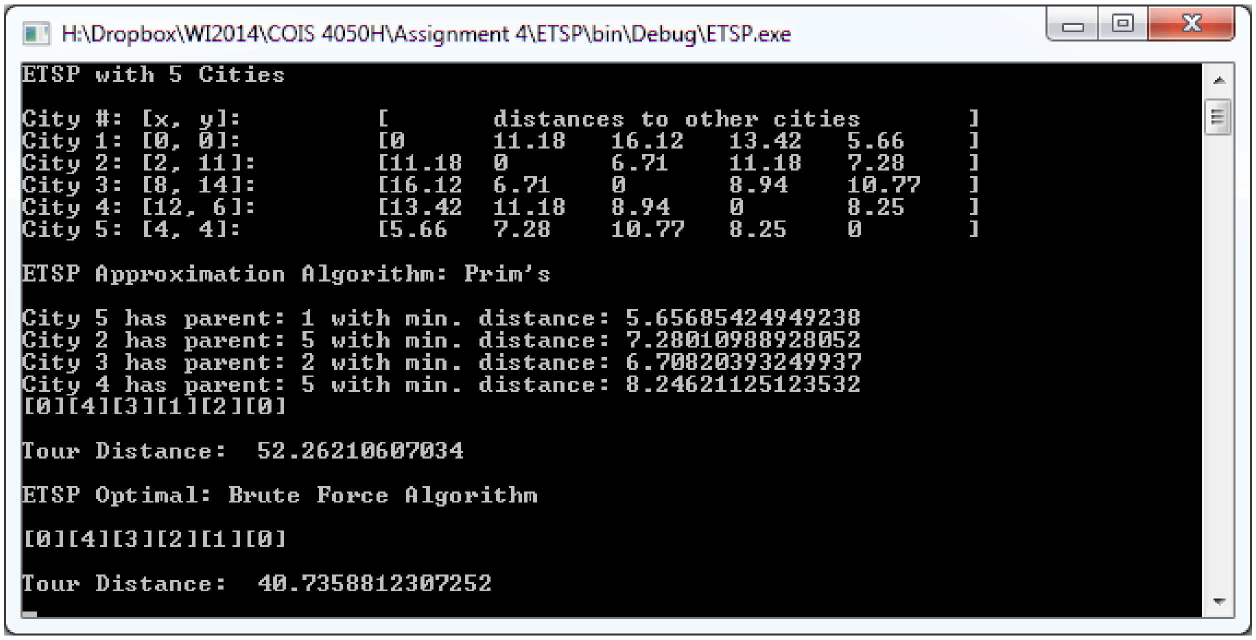
Before discovering Knuth’s algorithm as a permutation solution, I had attempted to program a set of for loops, one for each parent, and throwing the permutations of parent arrays at my DFS. *calculateTourDistance(…)* function. This produced some results, but also produced results of 0 distances, where cities were their own parents, and not linking to the starting city. I then attempted to use a recursive permutation stepping function, but I was not pleased with the overhead. **(See right)**



```
public int[] BruteCompute()
{
    int[] p = new int[numPermutations];
    ComputeAllPermutations(1, 1, ref p);
    return p;
}

private void ComputeAllPermutations(int start, int c, ref int[] p)
{
    if (c >= numCities * numCities)
    {
        permutations.Add(p);
        return;
    }
    for (int i = 0; i <= numCities; i++)
    {
        p[c] = i;
        ComputeAllPermutations(i + 1, c + 1, ref p);
    }
}
```

I had originally popped the permutation stack directly into the passing parameter of the **DFS**.*calculateTourDistance*(…) then realised I do not need to generate a parent array, as I already have a full route established. Having the permutation popped into the parent array (I was not thinking…) produced some very odd results. The save all seemed to be Knuth’s, and here are the results using the **Brute** class on the previous page:

The only issue that I would fix time permitting, is to process the permutations tour total within the *NextPermutation*() call, otherwise we have to wait for *every* permutation to be added to a stack before processing can proceed! (At 12 cities, the program had allocated 3.4GB of memory towards the process for all permutations…)

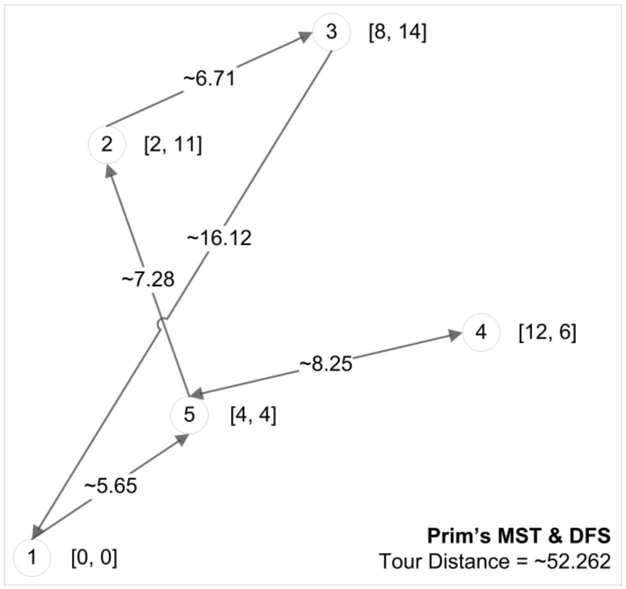


H:\Dropbox\WI2014\COIS 4050H\Assignment 4\ETSP\bin\Debug\ETSP.exe

```
ETSP with 5 Cities

City #: [x, y]:          [       distances to other cities       ]
City 1: [0, 0]:          [0      11.18   16.12   13.42   5.66    ]
City 2: [2, 11]:         [11.18  0       6.71    11.18   7.28    ]
City 3: [8, 14]:         [16.12  6.71    0       8.94    10.77   ]
City 4: [12, 6]:         [13.42  11.18   8.94    0       8.25    ]
City 5: [4, 4]:          [5.66   7.28    10.77   8.25    0       ]

ETSP Approximation Algorithm: Prim's

City 5 has parent: 1 with min. distance: 5.65685424949238
City 2 has parent: 5 with min. distance: 7.28010988928052
City 3 has parent: 2 with min. distance: 6.70820393249937
City 4 has parent: 5 with min. distance: 8.24621125123532
[0][4][3][1][2][0]

Tour Distance:  52.26210607034

ETSP Optimal: Brute Force Algorithm

[0][4][3][2][1][0]

Tour Distance:  40.7358812307252
```

3.1 The Approximation Algorithm

Using Prim's Algorithm to create an approximate solution to the ETSP, we obtain a minimum spanning tree. This MST can be traversed using a Depth First Search (DFS) algorithm to obtain an approximate total tour distance (Hamiltonian circuit). After traversing the tree and encountering a leaf node, the tour must add an additional distance to backtrack to its parent. This occurs in our example between 4 and 3. Once the tree has been traversed using DFS, we need to add the distance from the last traversed node, to the start position: see 2 to 0.



```
  class PrimsMST
  {
      public PrimsMST(int nCities)
      {
          size = nCities;
          parent = new int[size];
          distance = new double[size];
          included = new bool[size];
          adjacency = new double[size, size];
      }

      public void setAdjacency(double[,] adj) { adjacency = adj; }
      public int[] getParents() { return parent; }

      public void calculateMST(int index)
      {
          // Include root vertex
          included[index] = true;
          for (int i = 1; i < size; i++)
          {
              distance[i] = adjacency[0, i];
              included[i] = false;

    for (int j = 0; j < size - 1; j++)  // For each edge
    {
        int k = 0;
        double min = double.MaxValue;

        for (int i = 0; i < size; i++)
        {
            if (!included[i] && distance[i] < min)
            {
                min = distance[i];
                k = i;
            }
        }

        included[k] = true;
        for (int i = 0; i < size; i++)
        {
            if (!included[i] && (adjacency[k, i] < distance[i]))
            {
                distance[i] = adjacency[k, i];
                parent[i] = k;
            }
        }
    }
}
```

3.2 Approx. Implementation in C#

The approximate solution follows the one outlined in 3.1. Minimal distances of each cities are calculated and compared to produce a MST using Prim's Algorithm, then the tree was traversed and totalled to produce a tour distance of roughly **52.3** to traverse all points and return to the starting position.

Optimal Tour: 1, 5, 4, **3, 2**, 1.
Approximation Tour: 1, 5, 4, **2, 3**, 1.

The next page shows the DFS code used to traverse the MST created from Prim's Algorithm.

```
public static class DFS
{
    public static double calculateTourDistance(int start, City[] Cities, int[] parent, double[,] adj)
    {
        Stack<City> stack = new Stack<City>();

        bool hasChild = false;      // Keep track of when to backtrack increment
        double tourDistance = 0;    // Keep a running total

        City v = Cities[start];     // Root vertex at start city
        City last = v;              // Reference to return
        stack.Push(v);              // Push in the root node

        while (stack.Count > 0)
        {
            tourDistance += adj[parent[v.id], v.id];    // Every node popped, must increment distance
            v = stack.Pop();

            if (!v.visited)
            {
                // Go through each city, if they have children,
                // add them to traverse into,
                v.visited = true;
                hasChild = false;
                for (int c = 1; c < Cities.Length; c++)
                {
                    if (c != v.id)
                    {
                        if (parent[c] == v.id)
                        {
                            stack.Push(Cities[c]);
                            hasChild = true;
                        }
                    }
                }

                // If the city has no children (leaf), add the backtrack distance
                if (!hasChild)
                    tourDistance += adj[parent[v.id], v.id];
            }
            last = v;
        }

        // Add the distance from the deepest node back to the start node (Circuit)
        tourDistance += adj[start, last.id];

        return tourDistance;
    }
}
```

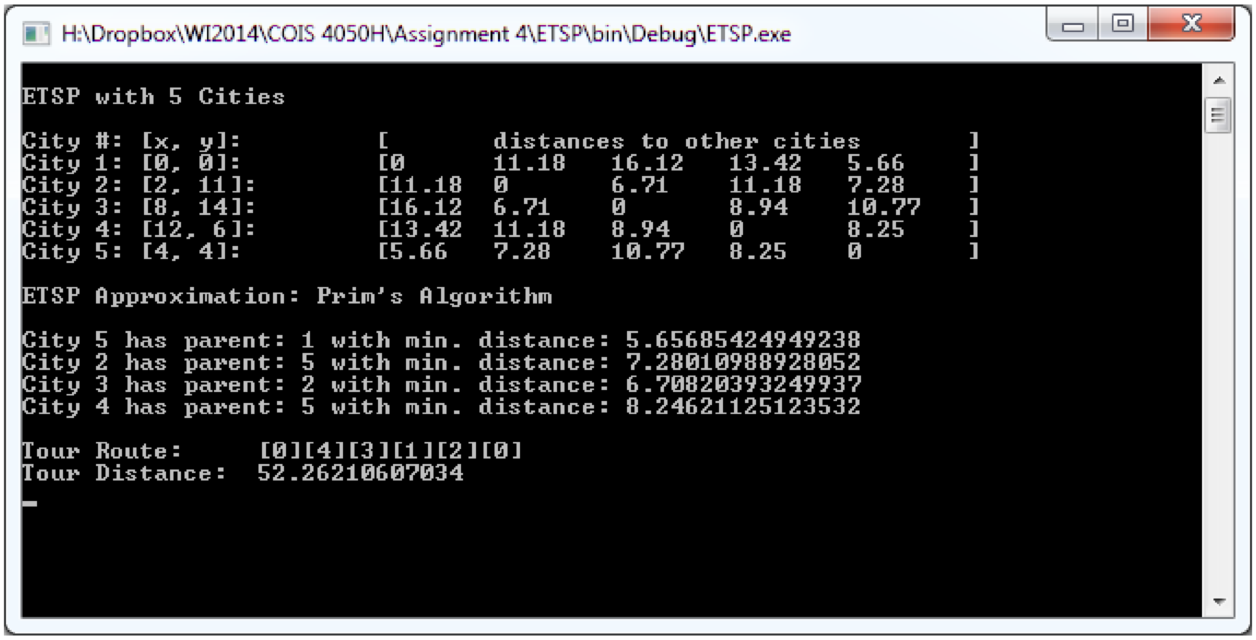
Shared Code

```
public struct City
{
    public City(int cid, int xv, int yv) { id = cid;  x = xv; y = yv; visited = false; }
    public int id;
    public int x;
    public int y;
    public bool visited;
}

class ETSP
{
    double[,] adj;
    const int numCities = 5;

    public static City[] Cities =
    {
        new City(0, 0, 0),
        new City(1, 2, 11),
        new City(2, 8, 14),
        new City(3, 12, 6),
        new City(4, 4, 4)
    };

    public ETSP(int startCity)
    {
        // Create and calculate distances our adjacency matrix from city locations,
        adj = new double[numCities, numCities];

        for (int i = 0; i < numCities; i++)
            for (int j = 0; j < numCities; j++)
                if (i == j)
                    adj[i, j] = 0;
                else
                    adj[i, j] = getDistance(Cities[i], Cities[j]);

        ApproxETSP(startCity);
    }

    public double ApproxETSP(int startCity)
    {
        PrimsMST mst = new PrimsMST(numCities);
        mst.setAdjacency(adj);
        mst.calculateMST(startCity);

        return DFS.calculateTourDistance(startCity, Cities, mst.getParents(), adj);
    }

    public double getDistance(City c1, City c2)
    {
        return Math.Sqrt(Math.Pow((c1.x - c2.x), 2) + Math.Pow((c1.y - c2.y), 2));
    }
}

public class A4
{
    public static void Main()
    {
        new ETSP(0);
        Console.ReadKey();
    }
}
```

3.2 Testing & Output Approximation Algorithm

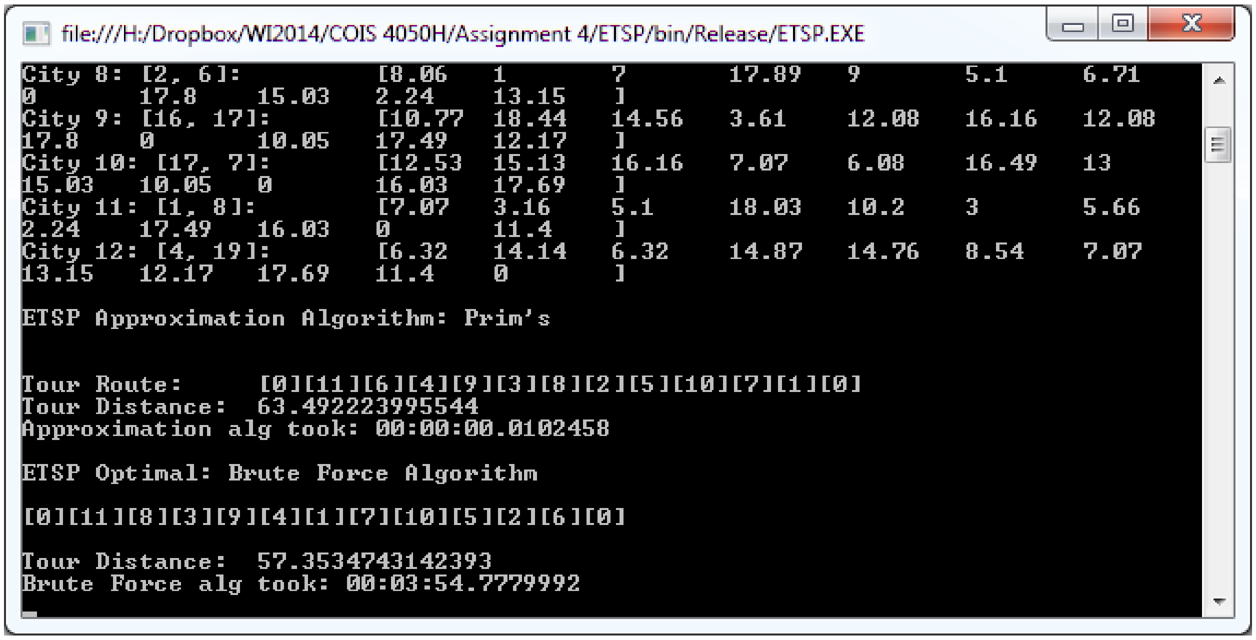
I was able to for the most part successfully run Prim's and DFS search without too much hassle (Class notes with pseudo code were extremely clear this time around). Here is the output of the approximation algorithm. Each city is identified, and for each a minimum distance to the next city was printed.

The approximation algorithm returned a distance of **52.262.**



H:\Dropbox\WI2014\COIS 4050H\Assignment 4\ETSP\bin\Debug\ETSP.exe

```
ETSP with 5 Cities

City #: [x, y]:         [       distances to other cities       ]
City 1: [0, 0]:         [0      11.18   16.12   13.42   5.66    ]
City 2: [2, 11]:        [11.18  0       6.71    11.18   7.28    ]
City 3: [8, 14]:        [16.12  6.71    0       8.94    10.77   ]
City 4: [12, 6]:        [13.42  11.18   8.94    0       8.25    ]
City 5: [4, 4]:         [5.66   7.28    10.77   8.25    0       ]

ETSP Approximation: Prim's Algorithm

City 5 has parent: 1 with min. distance: 5.65685424949238
City 2 has parent: 5 with min. distance: 7.28010988928052
City 3 has parent: 2 with min. distance: 6.70820393249937
City 4 has parent: 5 with min. distance: 8.24621125123532

Tour Route:     [0][4][3][1][2][0]
Tour Distance:  52.26210607034
_
```



file:///H:/Dropbox/WI2014/COIS 4050H/Assignment 4/ETSP/bin/Release/ETSP.EXE

```
City 8: [2, 6]:         [8.06   1       7       17.89   9       5.1     6.71
0       17.8    15.03   2.24    13.15   ]
City 9: [16, 17]:       [10.77  18.44   14.56   3.61    12.08   16.16   12.08
17.8    0       10.05   17.49   12.17   ]
City 10: [17, 7]:       [12.53  15.13   16.16   7.07    6.08    16.49   13
15.03   10.05   0       16.03   17.69   ]
City 11: [1, 8]:        [7.07   3.16    5.1     18.03   10.2    3       5.66
2.24    17.49   16.03   0       11.4    ]
City 12: [4, 19]:       [6.32   14.14   6.32    14.87   14.76   8.54    7.07
13.15   12.17   17.69   11.4    0       ]

ETSP Approximation Algorithm: Prim's


Tour Route:     [0][11][6][4][9][3][8][2][5][10][7][1][0]
Tour Distance:  63.492223995544
Approximation alg took: 00:00:00.0102458

ETSP Optimal: Brute Force Algorithm

[0][11][8][3][9][4][1][7][10][5][2][6][0]

Tour Distance:  57.3534743142393
Brute Force alg took: 00:03:54.7779992
_
```

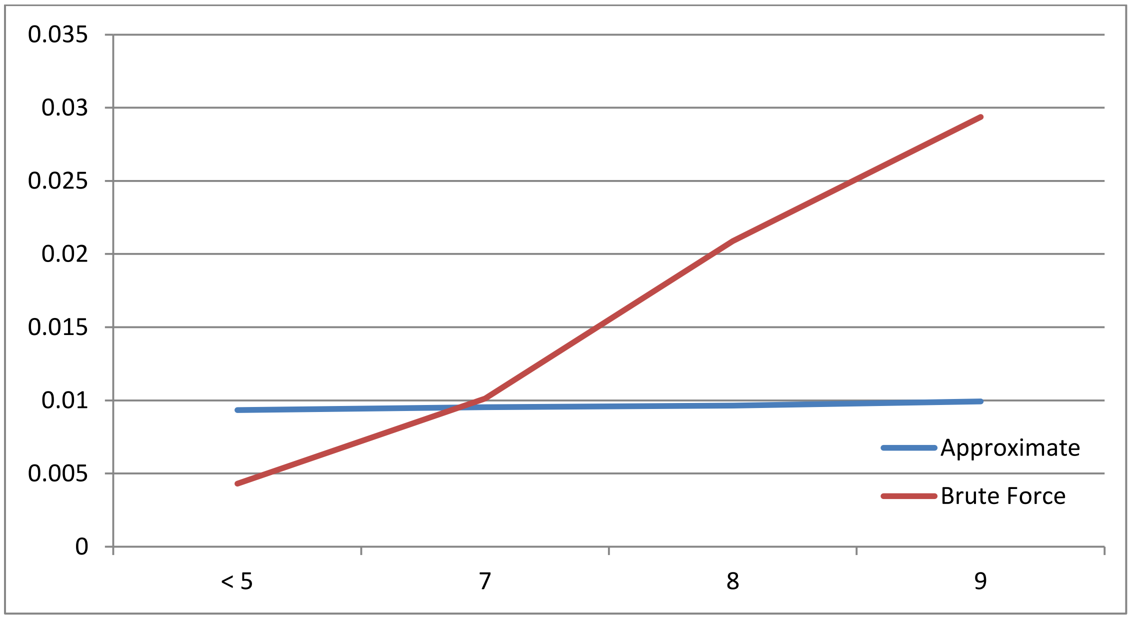
4.1 Average Approximation Ratio

Approximation	Optimal
52.262	40.735

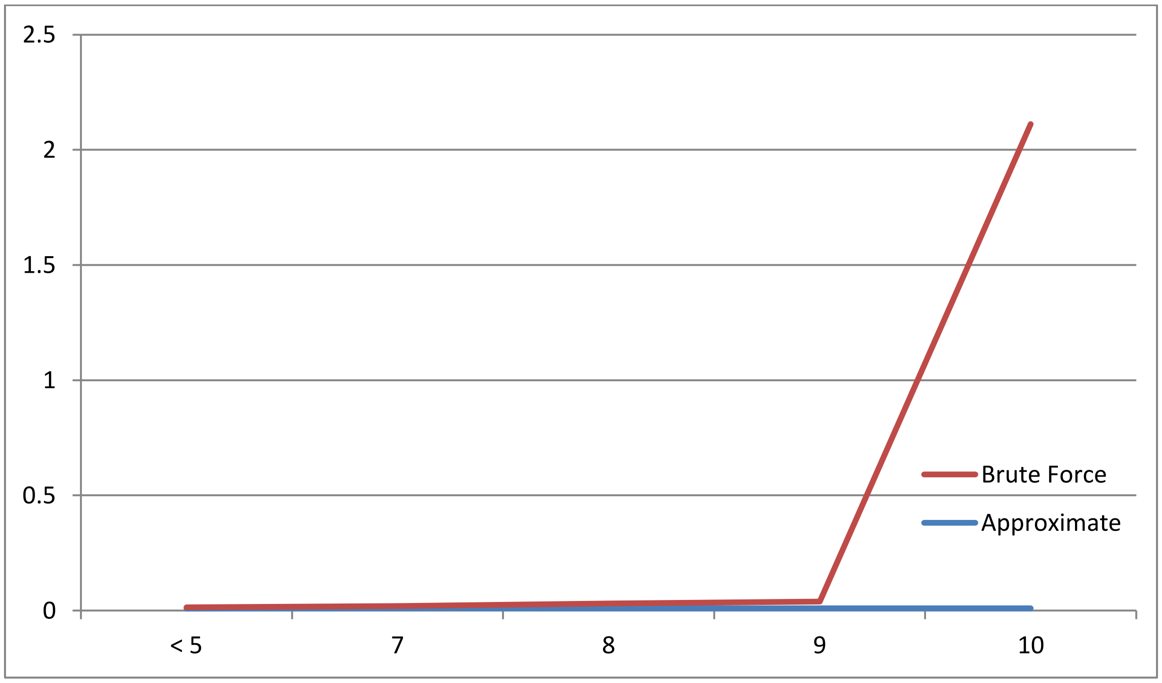
Ratio is *expected* to be under 2 times to optimal solution.

Approximation Ratio $= \frac{A}{O} \cong 1.282978$

4.2 Single Threaded Algorithm Performance



When there are under 9 cities, interestingly brute force is on par or faster than the Approximation.



When there are over 9 cities, we are not able to compare to the approximation algorithm anymore.