Brynne Hansen and Ife George– CS 312 Section 002
We spent about 15 hours on this project, but we didn't really keep track.

Algorithm (Branch and Bound)

State: A state consists of three things: a partial tour, a lower bound, and a cost matrix.

BSSF: Our quick solution implementation that calculated the initial best solution so far used the farthest insertion algorithm. Given a certain amount of points (cities), this algorithm first found the two farthest points (cities) on our map and figuratively drew a straight path between the two. It then found the point that is farthest from the points that make up that line. This farthest city then joins the path that was made between these two cities. This process is continued until all cities on the map are connected in a tour.

Bounding function: We re-used the reduced cost matrix method for our bounding function.

State expansion strategy: Our branch and bound algorithm creates successors of a state by including all possible edges from the previous city to the remaining cities. Impossible edges are eliminated by reducing the cost matrix; placing infinities where appropriate.

Agenda/Priority function: Our agenda was simply a map or a dictionary that had integers as key that pointed to a priority queue holding state objects. These integer keys represented the number of cities each state had in its tour, and each key mapped to a priority queue containing tours of that size. The priority queues were prioritized by bound. Therefore to focus on achieving a depth first search in our state space to update our BSSF quickly this agenda design choice made it easy to look for the highest value key and then dequeue the priority queue it was linked to.
* We used a third party priority queue that implemented a binary heap. It's also worth mentioning that this source and myself did adhere to the third party requirements explained in the syllabus. You can access this code at the following url: https://bitbucket.org/BlueRaja/high-speed-priority-queue-for-c/wiki/Getting%20Started.

Criterion: If the tour of any given state contains all the cities, then it is a solution. Because there are no duplicates in the tour, once it contains all the cities it is done.

Baseline Algorithm: To compare the optimality of our branch and bound, we used a greedy algorithm to compute a baseline tour. Starting with the first city, edges are added by finding the minimum distance to a non-visited city. That city is then added to the tour and we go on to add the next edge. This process continues until all cities are added.

Results

Branch and Bound

Competition Set						
Problem Specification			**Performance**			
# Cities	Seed	Problem #	Running time (s)	Cost of best tour found	Optimal?	Max. spage usage
10	17	1	.012416s	2610	Yes	10 million
10	17	2	.00726s	3326	Yes	10 million
50	321	1	60s	6016	No	50 million
50	321	2	60s	6423	No	50 million
50	321	3	60s	6160	No	50 million
15	5	1	.7829s	3613	Yes	15 million
20	8	2	60s	3626	No	20 million
30	13	1	60s	4700	No	30 million
35	13	1	60s	4976	No	35 million
40	13	1	60s	5242	No	40 million

One-Time Competition Set						
Problem Specification			**Performance**			
# Cities	Seed	Problem #	Running time (s)	Cost of best tour found	Optimal?	Max. spage usage
40	733	1	60s	5733	No	40 million
40	733	2	60s	4983	No	40 million
40	733	3	60s	5117	No	40 million

Baseline

Competition Set						
Problem Specification			**Performance**			
# Cities	Seed	Problem #	Running time (s)	Cost of best tour found	Optimal?	Max. spage usage
10	17	1	.0069710s	2851	No	100
10	17	2	.0000334s	3370	No	100
50	321	1	.0003051s	6777	No	2500
50	321	2	.0003508s	7389	No	2500
50	321	3	.0003386s	7411	No	2500
15	5	1	.0000418s	4022	No	225
20	8	2	.0000563s	3573	No	400
30	13	1	.0002062s	5241	No	900
35	13	1	.0001955s	5633	No	1225
40	13	1	.0002466s	5849	No	1600

One-Time Competition Set						
Problem Specification			**Performance**			
# Cities	Seed	Problem #	Running time (s)	Cost of best tour found	Optimal?	Max. spage usage
40	733	1	.0001891s	6539	No	1600
40	733	2	.0002031s	5724	No	1600
40	733	3	.0002469s	5522	No	1600

Notice in the data displayed in the tables above the greedy algorithm did not need a full minute to output the solution it came up with. Actually extra time would be useless because given more time it couldn't produce a more optimal solution. So although it is a lot faster than Branch & Bound it is also a lot less accurate. Notice that our Branch and Bound implementation in most cases used the full minute it was allotted but was able to produce a more optimal solution every time then what the greedy came up with it.

```
class State : PriorityQueueNode

{
       # region static members

    private static ArrayList cities;

    #endregion

       # region member variables

    public ArrayList tour { get; set; }
    private double[,] costMatrix;
    public double bound { get; set; }

    #endregion

       #region Constructors

       public State() { }

       /// <summary> creates the start state from a list of cities </summary>
       /// <param name="iRoute"> the list of cities generated by the solver </param>
       public State(ArrayList iRoute)
       {
              cities = iRoute;

              tour = new ArrayList();
              tour.Add(0);
              bound = 0;

              costMatrix = new double[iRoute.Count, iRoute.Count];
              for (int i = 0; i < iRoute.Count; ++i)
              {
                     City current = iRoute[i] as City;
                     for (int j = i; j < iRoute.Count; ++j)
                     {
                            double currCost = current.costToGetTo(iRoute[j] as City);
                            if (i == j) // on the diagonal
                                   currCost = double.PositiveInfinity;
                            costMatrix[i, j] = currCost;
                            costMatrix[j, i] = costMatrix[i, j];
                     }
              }
              reduceMatrix();
       }

    /// <summary> creates a deep copy of a state </summary>
    /// <param name="s"> the state to copy (the parent state) </param>
    public State(State s)
    {
        this.tour = new ArrayList(s.tour);
        this.bound = s.bound;

        costMatrix = (double[,])s.costMatrix.Clone();
    }

    #endregion
```

```
#region public methods

public int getNumCities()
{
    return cities.Count;
}

/// <summary>
///  returns all the successors of a state
///  adds the next city in the tour based off the previous city
///  duplicate cities and impossible cities are eliminated (by the cost matrix)
/// </summary>
/// <returns> the list of successors </returns>
public List<State> successors()
{
          int lastCity = (int)tour[tour.Count - 1];

          List<State> result = new List<State>();
          for (int i = 0; i < cities.Count; ++i)
          {
                State successor = new State(this);
                if (costMatrix[lastCity, i] != Double.PositiveInfinity)
                {
                      successor.tour.Add(i);
                      successor.bound += costMatrix[lastCity, i];
                      successor.reduceMatrix();
                      result.Add(successor);
                }
          }
    return result;
}

/// <summary> determines if a state is a valid solution </summary>
/// <returns>
///  true if the state is a solution
///  false if the state is not a solution
/// </returns>
   public bool criterion()
   {
          return tour.Count == cities.Count;
   }

/// <summary> returns the list of cities in the order of the tour </summary>
/// <returns> the tour of the current state </returns>
   public ArrayList getTour()
   {
          ArrayList route = new ArrayList();
          for (int i = 0; i < tour.Count; ++i)
          {
                int city = (int)tour[i];
                route.Add(cities[city]);
          }
          return route;
   }

#endregion
```

```
    # region private methods

    /// <summary>
       ///  reduces the costMatrix
       ///  subtracts the minimum from each row and column and adds it to the bound
       /// </summary>
       private void reduceMatrix()
       {
              // knock out row and diagonal
              if (tour.Count > 1)
              {
                     int secondToLast = (int)tour[tour.Count - 2];
                     int last = (int)tour[tour.Count - 1];
                     for (int i = 0; i < cities.Count; ++i)
                            costMatrix[secondToLast, i] = double.PositiveInfinity;
                     for (int i = 0; i < cities.Count; ++i)
                            costMatrix[i, last] = double.PositiveInfinity;
              }

              // reduce each row
              for (int i = 0; i < cities.Count; ++i)
              {
                     double min = double.PositiveInfinity;
                     for (int j = 0; j < cities.Count; ++j)
                     {
                            if (costMatrix[i, j] < min)
                                   min = costMatrix[i, j];
                     }
                     if (min != 0 && min != double.PositiveInfinity)
                     {
                            for (int j = 0; j < cities.Count; ++j)
                            {
                                   if (costMatrix[i,j] != double.PositiveInfinity) //
don't reduce the infinities
                                          costMatrix[i, j] -= min;
                            }
                            bound += min;
                     }

              }
              // reduce each column
              for (int j = 0; j < cities.Count; ++j)
              {
                     double min = double.PositiveInfinity;
                     for (int i = 0; i < cities.Count; ++i)
                     {
                            if (costMatrix[i, j] < min)
                                   min = costMatrix[i, j];
                     }
                     if (min != 0 && min != double.PositiveInfinity)
                     {
                            for (int i = 0; i < cities.Count; ++i)
                            {
                                   if (costMatrix[i, j] != double.PositiveInfinity) //
don't reduce the infinities
                                          costMatrix[i, j] -= min;
                            }
                            bound += min;
```

```
                    }

                }
            // remove impossible edges
                if (tour.Count > 1)
                {
                    int lastCity = (int)tour[tour.Count - 1];
                for (int j = 0; j < tour.Count - 1; ++j)
                    {
                        costMatrix[lastCity, j] = double.PositiveInfinity;
                    }
                }
        }

    #endregion
}

class Agenda
{
    Dictionary<int, HeapPriorityQueue<State>> statespace = new Dictionary<int,
HeapPriorityQueue<State>>();

    public Agenda() {}

    public void clear()
    {
        statespace.Clear();
    }

    public bool empty()
    {
        if(statespace.Count==0)
        {
            return true;
        }
        return false;
    }

    public void add (State s)
    {
        int key = s.tour.Count;

        if(statespace.ContainsKey(key))
        {
            statespace[key].Enqueue(s, s.bound);
        }
        else
        {
            HeapPriorityQueue<State> entry = new HeapPriorityQueue<State>(1000000);
            entry.Enqueue(s, s.bound);
            statespace.Add(key, entry);
        }
    }

    public State remove_first(int depth)
    {
        try
```

```
{
    State result = statespace[depth].Dequeue();
    if (statespace[depth].Count == 0)
    {
        statespace.Remove(depth);
    }
    return result;
}
catch (KeyNotFoundException e)
{
    int[] levels = new int[statespace.Count];
    statespace.Keys.CopyTo(levels, 0);
    int champ;

    try
    {
        champ = levels[levels.Length - 1];
    }
    catch (ArgumentOutOfRangeException e2)
    {
        champ = -1;
    }

    //Then return it
    if(champ != -1)
    {
        State result=statespace[champ].Dequeue();
        if (statespace[champ].Count==0)
        {
            statespace.Remove(champ);
        }
        return result;
    }
    return null;
}
//Copy keys of map
//int[] levels = new int[statespace.Count];
//statespace.Keys.CopyTo(levels, 0);

////Find the highest key
//int champ;
//try
//{
//    champ = levels[levels.Length-1];
//}
//catch (ArgumentOutOfRangeException e)
//{
//    champ = -1;
//}
//for (int i = 0; i<levels.Length; i++)
//{
//    if(levels[i]>champ)
//    {
//        champ = levels[i];
//    }
//}

//Then return it
```

```
    //if(champ != -1)
    //{
    //    State result=statespace[champ].Dequeue();
    //    if (statespace[champ].Count==0)
    //    {
    //        statespace.Remove(champ);
    //    }
    //    return result;
    //}
    //return null;
}

public State first(int depth)
{
    try
    {
        State result = statespace[depth].First;
        return result;
    }
    catch (KeyNotFoundException e)
    {
        int[] levels = new int[statespace.Count];
        statespace.Keys.CopyTo(levels, 0);
        int champ;

        try
        {
            champ = levels[levels.Length - 1];
        }
        catch (ArgumentOutOfRangeException e2)
        {
            champ = -1;
        }

        //Then return it
        if (champ != -1)
        {
            return statespace[champ].First;
        }
        return null;
    }
    ////Copy keys of map
    //int[] levels = new int[statespace.Count];
    //statespace.Keys.CopyTo(levels, 0);

    ////Find the highest key
    //int champ;
    //try
    //{
    //    champ = levels[levels.Length-1];
    //}
    //catch (ArgumentOutOfRangeException e)
    //{
    //    champ = -1;
    //}
    ////for (int i = 0; i < levels.Length; i++)
    ////{
    ////    if (levels[i] > champ)
```

```
        ////    {
        ////        champ = levels[i];
        ////    }
        ////}

        ////THen return it
        //if (champ != -1)
        //{
        //    return statespace[champ].First;
        //}
        //return null;
    }
}

/// <summary>
///  solve the problem.  This is the entry point for the solver when the run button is
clicked
///  right now it just picks a simple solution.
/// </summary>
public void solveProblem()
{
    Route = new ArrayList();
    Stopwatch timer = new Stopwatch();
    // baseline
    //timer.Start();
    //bssf = new TSPSolution(baseLine());
    //timer.Stop();

    Agenda agenda = new Agenda();

    //bssf
    ArrayList sol = quickSolution();
    bssf = new TSPSolution(sol);
    double bssfCost = costOfBssf();
    //init_state
    State initial = new State(sol);

    agenda.add(initial);

    timer.Start();
    int depth = 1; // tour starts at 1 city
    while (!agenda.empty() && timer.Elapsed < timeLimit && bssfCost !=
agenda.first(depth).bound)
    {
        State s = agenda.first(depth); //initial; // change to agenda.first()
        agenda.remove_first(depth);
        if (s.bound < bssfCost)
        {
            List<State> children = s.successors();
            foreach (State child in children)
            {
                if (!(timer.Elapsed < timeLimit))
                {
                    timer.Stop();
                    break;
                }
                if (child.bound < bssfCost)
```

```
                {
                    if (child.criterion())
                    {
                        Console.Write("updated BSSF: " + costOfBssf());
                        bssf = new TSPSolution(child.getTour());
                        bssfCost = costOfBssf();
                    }
                    else
                        agenda.add(child);
                }
            }
        }
        depth++;
    }
    timer.Stop();

    // update the cost of the tour.
    Program.MainForm.tbCostOfTour.Text = " " + bssf.costOfRoute();
    // print out the time elapsed
    Program.MainForm.tbElapsedTime.Text = timer.Elapsed.ToString();
    // do a refresh.
    Program.MainForm.Invalidate();

    // print to clipboard
    Clipboard.SetText(bssf.ToString());
    // System.Console.Write(bssf.ToString());
}
#endregion

public ArrayList quickSolution()
{
    ArrayList quick = new ArrayList();

    City first = Cities[0];
    double champ = 0;
    City other = null;

    for (int x = 1; x < Cities.Length; x++)
    {
        double dist = first.costToGetTo(Cities[x]);
        if(dist > champ)
        {
            champ = dist;
            other = Cities[x];
        }
    }

    quick.Add(Cities[0]);
    quick.Add(other);

    while (quick.Count != Cities.Length)
    {
        City insert = null;
        double winner = 0;

        for (int i = 1; i < Cities.Length; i++)
        {
            City r = Cities[i];
```

```
            if (!quick.Contains(r))
            {
                double loser = Double.PositiveInfinity;
                for (int j = 0; j < quick.Count; j++)
                {
                    double path = r.costToGetTo((City)quick[j]);
                    if (path < loser)
                    {
                        loser = path;
                    }
                }
                if (winner < loser)
                {
                    winner = loser;
                    insert = r;
                }
            }
        }

        double minIntersect = Double.PositiveInfinity;
        int frontindex = -1;
        int behindindex = -1;

        for (int k=0; k < quick.Count; k++)
        {
            City begin = (City)quick[k % quick.Count];
            City last = (City)quick[(k + 1) % quick.Count];
            double cir = begin.costToGetTo(insert);
            double crj = insert.costToGetTo(last);
            double cij = begin.costToGetTo(last);
            double total = cir + crj - cij;
            if(total < minIntersect)
            {
                minIntersect = total;
                frontindex = k;
                behindindex = k + 1;
            }
        }
        quick.Insert(behindindex, insert);
    }
    return quick;
}

public ArrayList baseLine()
{
    // greedy algorithm
    // create cost matrix
    double[,] costMatrix = new double[Cities.Length,Cities.Length];
    for (int i = 0; i < Cities.Length; ++i)
    {
        City current = Cities[i];
        for (int j = i; j < Cities.Length; ++j)
        {
            double currCost = current.costToGetTo(Cities[j]);
            if (i == j) // on the diagonal
                currCost = double.PositiveInfinity;
            costMatrix[i, j] = currCost;
            costMatrix[j, i] = costMatrix[i, j];
```

```
        }
    }

    ArrayList greedySolution = new ArrayList();
    City first = Cities[0];
    greedySolution.Add(first); // add the first city
    City previous = first;
    int index = 0;
    do
    {
        // mark out previous city in costMatrix
        for (int i = 0; i < Cities.Length; ++i)
            costMatrix[i, index] = double.MaxValue;

        // find the min distance in the row
        double min = double.MaxValue;
        int winner = -1;
        for (int i = 0; i < Cities.Length; ++i)
        {
            if (costMatrix[index,i] < min)
            {
                min = costMatrix[index, i];
                winner = i;
            }
        }
        City nextInRoute = Cities[winner];
        greedySolution.Add(nextInRoute);

        // update index
        index = winner;
        previous = Cities[winner];
    }
    while (greedySolution.Count != Cities.Length);

    return greedySolution;
}
```