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

namespace TSP
{

    class ProblemAndSolver
    {

        private class TSPSolution
        {
            /// <summary>
            /// we use the representation [cityB,cityA,cityC]
            /// to mean that cityB is the first city in the solution, cityA is the second, cityC is the third
            /// and the edge from cityC to cityB is the final edge in the path.
            /// You are, of course, free to use a different representation if it would be more convenient or efficient
            /// for your data structure(s) and search algorithm.
            /// </summary>
            public ArrayList
                Route;

            /// <summary>
            /// constructor
            /// </summary>
            /// <param name="iroute">a (hopefully) valid tour</param>
            public TSPSolution(ArrayList iroute)
            {
                Route = new ArrayList(iroute);
            }

            /// <summary>
            /// Compute the cost of the current route.
            /// Note: This does not check that the route is complete.
            /// It assumes that the route passes from the last city back to the first city.
            /// </summary>
            /// <returns></returns>
            public double costOfRoute()
            {
                // go through each edge in the route and add up the cost.
                int x;
                City here;
                double cost = 0D;

                for (x = 0; x < Route.Count - 1; x++)
```

```
            {
                here = Route[x] as City;
                cost += here.costToGetTo(Route[x + 1] as City);
            }

            // go from the last city to the first.
            here = Route[Route.Count - 1] as City;
            cost += here.costToGetTo(Route[0] as City);
            return cost;
        }
    }

    #region Private members

    /// <summary>
    /// Default number of cities (unused -- to set defaults, change the values in 
      the GUI form)
    /// </summary>
    // (This is no longer used -- to set default values, edit the form directly.  
      Open Form1.cs,
    // click on the Problem Size text box, go to the Properties window (lower 
      right corner),
    // and change the "Text" value.)
    private const int DEFAULT_SIZE = 25;

    /// <summary>
    /// Default time limit (unused -- to set defaults, change the values in the 
      GUI form)
    /// </summary>
    // (This is no longer used -- to set default values, edit the form directly.  
      Open Form1.cs,
    // click on the Time text box, go to the Properties window (lower right 
      corner),
    // and change the "Text" value.)
    private const int TIME_LIMIT = 60;        //in seconds

    private const int CITY_ICON_SIZE = 5;


    // For normal and hard modes:
    // hard mode only
    private const double FRACTION_OF_PATHS_TO_REMOVE = 0.20;

    /// <summary>
    /// the cities in the current problem.
    /// </summary>
    private City[] Cities;
    /// <summary>
    /// a route through the current problem, useful as a temporary variable.
    /// </summary>
    private ArrayList Route;
    /// <summary>
```

```
        /// best solution so far.
        /// </summary>
        private TSPSolution bssf;

        /// <summary>
        /// how to color various things.
        /// </summary>
        private Brush cityBrushStartStyle;
        private Brush cityBrushStyle;
        private Pen routePenStyle;


        /// <summary>
        /// keep track of the seed value so that the same sequence of problems can be
        /// regenerated next time the generator is run.
        /// </summary>
        private int _seed;
        /// <summary>
        /// number of cities to include in a problem.
        /// </summary>
        private int _size;

        /// <summary>
        /// Difficulty level
        /// </summary>
        private HardMode.Modes _mode;

        /// <summary>
        /// random number generator.
        /// </summary>
        private Random rnd;

        /// <summary>
        /// time limit in milliseconds for state space search
        /// can be used by any solver method to truncate the search and return the
          BSSF
        /// </summary>
        private int time_limit;
        #endregion

        #region Public members

        /// <summary>
        /// These three constants are used for convenience/clarity in populating and
          accessing the results array that is passed back to the calling Form
        /// </summary>
        public const int COST = 0;
        public const int TIME = 1;
        public const int COUNT = 2;

        public int Size
        {
```

```
            get { return _size; }
        }

        public int Seed
        {
            get { return _seed; }
        }
        #endregion

        #region Constructors
        public ProblemAndSolver()
        {
            this._seed = 1;
            rnd = new Random(1);
            this._size = DEFAULT_SIZE;
            this.time_limit = TIME_LIMIT * 1000;                  // TIME_LIMIT is in 
              seconds, but timer wants it in milliseconds

            this.resetData();
        }

        public ProblemAndSolver(int seed)
        {
            this._seed = seed;
            rnd = new Random(seed);
            this._size = DEFAULT_SIZE;
            this.time_limit = TIME_LIMIT * 1000;                  // TIME_LIMIT is in 
              seconds, but timer wants it in milliseconds

            this.resetData();
        }

        public ProblemAndSolver(int seed, int size)
        {
            this._seed = seed;
            this._size = size;
            rnd = new Random(seed);
            this.time_limit = TIME_LIMIT * 1000;                        // TIME_LIMIT 
              is in seconds, but timer wants it in milliseconds

            this.resetData();
        }
        public ProblemAndSolver(int seed, int size, int time)
        {
            this._seed = seed;
            this._size = size;
            rnd = new Random(seed);
            this.time_limit = time*1000;                        // time is entered in 
              the GUI in seconds, but timer wants it in milliseconds

            this.resetData();
        }
```

```
        #endregion

        #region Private Methods

        /// <summary>
        /// Reset the problem instance.
        /// </summary>
        private void resetData()
        {

            Cities = new City[_size];
            Route = new ArrayList(_size);
            bssf = null;

            if (_mode == HardMode.Modes.Easy)
            {
                for (int i = 0; i < _size; i++)
                    Cities[i] = new City(rnd.NextDouble(), rnd.NextDouble());
            }
            else // Medium and hard
            {
                for (int i = 0; i < _size; i++)
                    Cities[i] = new City(rnd.NextDouble(), rnd.NextDouble(), 
                      rnd.NextDouble() * City.MAX_ELEVATION);
            }

            HardMode mm = new HardMode(this._mode, this.rnd, Cities);
            if (_mode == HardMode.Modes.Hard)
            {
                int edgesToRemove = (int)(_size * FRACTION_OF_PATHS_TO_REMOVE);
                mm.removePaths(edgesToRemove);
            }
            City.setModeManager(mm);

            cityBrushStyle = new SolidBrush(Color.Black);
            cityBrushStartStyle = new SolidBrush(Color.Red);
            routePenStyle = new Pen(Color.Blue,1);
            routePenStyle.DashStyle = System.Drawing.Drawing2D.DashStyle.Solid;
        }

        #endregion

        #region Public Methods

        /// <summary>
        /// make a new problem with the given size.
        /// </summary>
        /// <param name="size">number of cities</param>
        public void GenerateProblem(int size, HardMode.Modes mode)
        {
            this._size = size;
            this._mode = mode;
```

```
            resetData();
        }

        /// <summary>
        /// make a new problem with the given size, now including timelimit paremeter 
          that was added to form.
        /// </summary>
        /// <param name="size">number of cities</param>
        public void GenerateProblem(int size, HardMode.Modes mode, int timelimit)
        {
            this._size = size;
            this._mode = mode;
            this.time_limit = timelimit*1000;                                   //     
              convert seconds to milliseconds
            resetData();
        }

        /// <summary>
        /// return a copy of the cities in this problem.
        /// </summary>
        /// <returns>array of cities</returns>
        public City[] GetCities()
        {
            City[] retCities = new City[Cities.Length];
            Array.Copy(Cities, retCities, Cities.Length);
            return retCities;
        }

        /// <summary>
        /// draw the cities in the problem.  if the bssf member is defined, then
        /// draw that too.
        /// </summary>
        /// <param name="g">where to draw the stuff</param>
        public void Draw(Graphics g)
        {
            float width  = g.VisibleClipBounds.Width-45F;
            float height = g.VisibleClipBounds.Height-45F;
            Font labelFont = new Font("Arial", 10);

            // Draw lines
            if (bssf != null)
            {
                // make a list of points.
                Point[] ps = new Point[bssf.Route.Count];
                int index = 0;
                foreach (City c in bssf.Route)
                {
                    if (index < bssf.Route.Count -1)
                        g.DrawString(" " + index +"("+c.costToGetTo(bssf.Route[index 
                      +1]as City)+")", labelFont, cityBrushStartStyle, new PointF 
                      ((float)c.X * width + 3F, (float)c.Y * height));
                    else
```

```
                        g.DrawString(" " + index +"("+c.costToGetTo(bssf.Route[0]as
                      City)+")", labelFont, cityBrushStartStyle, new PointF((float)c.X
                       * width + 3F, (float)c.Y * height));
                    ps[index++] = new Point((int)(c.X * width) + CITY_ICON_SIZE / 2,
                      (int)(c.Y * height) + CITY_ICON_SIZE / 2);
                }

                if (ps.Length > 0)
                {
                    g.DrawLines(routePenStyle, ps);
                    g.FillEllipse(cityBrushStartStyle, (float)Cities[0].X * width - 1,
                      (float)Cities[0].Y * height - 1, CITY_ICON_SIZE + 2,
                      CITY_ICON_SIZE + 2);
                }

                // draw the last line.
                g.DrawLine(routePenStyle, ps[0], ps[ps.Length - 1]);
            }

            // Draw city dots
            foreach (City c in Cities)
            {
                g.FillEllipse(cityBrushStyle, (float)c.X * width, (float)c.Y * height,
                   CITY_ICON_SIZE, CITY_ICON_SIZE);
            }

        }

        /// <summary>
        ///  return the cost of the best solution so far.
        /// </summary>
        /// <returns></returns>
        public double costOfBssf ()
        {
            if (bssf != null)
                return (bssf.costOfRoute());
            else
                return -1D;
        }

        /// <summary>
        /// This is the entry point for the default solver
        /// which just finds a valid random tour
        /// </summary>
        /// <returns>results array for GUI that contains three ints: cost of solution,
           time spent to find solution, number of solutions found during search (not
          counting initial BSSF estimate)</returns>
        public string[] defaultSolveProblem()
        {
            int i, swap, temp, count=0;
            string[] results = new string[3];
            int[] perm = new int[Cities.Length];
```

```
            Route = new ArrayList();
            Random rnd = new Random();
            Stopwatch timer = new Stopwatch();

            timer.Start();

            do
            {
                for (i = 0; i < perm.Length; i++)                                 // ⮐
                  create a random permutation template
                    perm[i] = i;
                for (i = 0; i < perm.Length; i++)
                {
                    swap = i;
                    while (swap == i)
                        swap = rnd.Next(0, Cities.Length);
                    temp = perm[i];
                    perm[i] = perm[swap];
                    perm[swap] = temp;
                }
                Route.Clear();
                for (i = 0; i < Cities.Length; i++)                            // Now ⮐
                  build the route using the random permutation
                {
                    Route.Add(Cities[perm[i]]);
                }
                bssf = new TSPSolution(Route);
                count++;
            } while (costOfBssf() == double.PositiveInfinity);                // until⮐
               a valid route is found
            timer.Stop();

            results[COST] = costOfBssf().ToString();                          // load ⮐
              results array
            results[TIME] = timer.Elapsed.ToString();
            results[COUNT] = count.ToString();

            return results;
        }



        int BBcount = -1;
        int BBstatesCreated = 0;
        int BBstatesPruned = 0;
        int BBcurrentStates = 0;
        int BBmaxStates = 0;
        int BSSFupdates = -1;
        /// <summary>
        /// performs a Branch and Bound search of the state space of partial tours
        /// stops when time limit expires and uses BSSF as solution
        /// </summary>
```

```
        /// <returns>results array for GUI that contains three ints: cost of solution, 
            time spent to find solution, number of solutions found during search (not 
          counting initial BSSF estimate)</returns>
        public string[] bBSolveProblem()
        {
            string[] results = new string[3];

            // TODO: Add your implementation for a branch and bound solver here.
            Stopwatch timer = new Stopwatch();
            timer.Start();
            // Alrighty. We start with an overall bound
            // Then we branch
            // For each branch, it gets bounded
            // Which bound is least?
            // And then more branches.

            // Make our data matrix
            double[,] data = new double[Cities.Length,Cities.Length];

            // populate the data matrix
            for (int row = 0; row < data.GetLength(0); row++) {
                for (int col = 0; col < data.GetLength(1); col++) {
                    // For cities to/from themselves, the cost is now infinity.
                    data[row, col] = (row == col) ? double.PositiveInfinity : Cities 
                      [row].costToGetTo(Cities[col]);
                }
            }

            // Alrighty. We start with an overall bound
            // We get this by getting the minimum leaving path for every city and 
              adding them up

            double base_bound = row_and_column_reduce(data);

            int[] path = new int[Cities.Length];
            for (int i= 0; i< path.Length; i++) {
                path[i] = -1;
            }
            path[0] = 2;    // We start at node 2. Always. Seems to work better than 
              other methods [shrugs]

            LinkedListNode priorityQueue = new LinkedListNode(base_bound, data, path, 
              1);

            // Once the thing at the top of our priority queue is a solved path, we 
              have finished
            while (priorityQueue.path_size < Cities.Length && 
              timer.ElapsedMilliseconds <= time_limit) {
            // 1 we branch on the best node so far
            //  for each branch, we bound it (make a new node and add to queue)
            //  the parent node gets removed from the queue, and then we can 
              continue :)
```

```
            LinkedListNode parent = priorityQueue;
                priorityQueue = removeNode(parent, priorityQueue);

                // How many branches to make? Easy. 1 fewer than our current path 
                  length.
                for (int i = 0; i<Cities.Length; i++) {
                    // If we already have the thing in our path, it means we can't go 
                      to it. fail
                    if (parent.path.Contains(i)) {
                        continue;
                    }
                    // branches. woo.
                    // So now we are branching (last thing in path)->new thing
                    int last = parent.path[parent.path_size - 1];

                    double[,] data_branch = (double[,])parent.data.Clone();   // This 
                      cast is retarded.

                    // Get the bound for this node
                    double bound_branch = parent.bound + data_branch[last, i];


                    // Black out everything now.
                    // Black out the row
                    for (int row = 0; row < data_branch.GetLength(0); row++) {
                        data_branch[row, i] = Double.PositiveInfinity;
                    }
                    // Black out the column
                    for (int col = 0; col < data_branch.GetLength(1); col++) {
                        data_branch[last, col] = Double.PositiveInfinity;
                    }
                    // AND black out the backwards path too.
                    data_branch[i, last] = Double.PositiveInfinity;



                    // Now to row+column reduce.
                    bound_branch += row_and_column_reduce(data_branch);

                    // Make a new path and size for our branch and record the next 
                      move
                    int[] branch_path = (int[]) parent.path.Clone();
                    int branch_path_size = parent.path_size + 1;
                    branch_path[branch_path_size-1] = i;

                    // If we are at the end, then add the last city on there.
                    if(branch_path_size == Cities.Length - 1) {
                        int last_city = 0;
                        while (branch_path.Contains(last_city)) {
                            last_city++;
                        }
```

```
                    branch_path_size++;
                    branch_path[branch_path_size - 1] = last_city;
                    bound_branch += data_branch[branch_path[branch_path_size - 2],
                    branch_path[branch_path_size - 1]];  // way too gnarly. :/
                    // This adds the last item possible onto the path and to the
                    bound. Which makes it a complete path and the path_size ==
                    Cities.length.
                    // If it is the best one, next time around it will be returned
                    as the correct answer.
                }

                // In any case, time to make a new node and put it on the queue.
                LinkedListNode branch_node = new LinkedListNode(bound_branch,
                  data_branch, branch_path, branch_path_size);
                priorityQueue = addNode(branch_node, priorityQueue);
                // Gotta record the max amount of states- might be the current
                  number
                BBmaxStates = (BBcurrentStates > BBmaxStates) ? BBcurrentStates :
                  BBmaxStates;
            }
        }

        //priorityQueue now has the node with our winning thing! Or we timed out


        timer.Stop();   // DONE CALCULATING!


        // Now time to count the "pruned" stuff that just stayed on the end
        LinkedListNode temp = priorityQueue.next;
        while (temp != null)
        {
            BBstatesPruned++;
            temp = temp.next;
        }

        // For the writeup
        Console.WriteLine("states created: " + BBstatesCreated);
        Console.WriteLine("states pruned: " + BBstatesPruned);
        Console.WriteLine("max states " + BBmaxStates);
        Console.WriteLine("bssf updates " + BSSFupdates);

        // If we never found a solution, say so
        if (BBcount == -1) {
            //results[COST] = double.PositiveInfinity.ToString();           //
              load results array
            //results[TIME] = timer.Elapsed.ToString();
            //results[COUNT] = BBcount.ToString();
            //return results;
            return defaultSolveProblem();
        }
```

```
            // If we have a solution, update the BSSF. Probably not necessary
            if (priorityQueue.path_size == Cities.Length)
            {
                Route = new ArrayList();
                for (int i = 0; i < priorityQueue.path_size; i++)
                {
                    Route.Add(Cities[priorityQueue.path[i]]);
                }
                bssf = new TSPSolution(Route);
            }
            results[COST] = costOfBssf().ToString();                          // load 
              results array
            results[TIME] = timer.Elapsed.ToString(); ;
            results[COUNT] = BBcount.ToString();

            return results;
        }

        // What this does is row-reduce and column-reduce the received matrix.
        // It returns what to add to the bound.
        // O(n)
        private double row_and_column_reduce(double[,] data) {
            double bound = 0;


            // Time for row reducing
            for (int row = 0; row < data.GetLength(0); row++) {
                double smallest_in_row = Double.PositiveInfinity;

                for (int col = 0; col < data.GetLength(1); col++) {
                    double current = data[row,col];

                    // if the new one is smallest, record it.
                    smallest_in_row = (current < smallest_in_row) ? current : 
                      smallest_in_row ;
                }   // Now we have the smallest in the row.

                // I hope this is not necessary?
                if (smallest_in_row != 0 && smallest_in_row != 
                  Double.PositiveInfinity) {
                    for (int col = 0; col < data.GetLength(1); col++) {
                        data[row, col] -= smallest_in_row;
                    }
                    bound += smallest_in_row;
                }
            }


            // Now time for column reducing
            for (int col = 0; col < data.GetLength(1); col++) {
                double smallest_in_col = Double.PositiveInfinity;
```

```
                for (int row = 0; row < data.GetLength(0); row++) {
                    double current = data[row, col];

                    smallest_in_col = (current < smallest_in_col) ? current :
                      smallest_in_col;
                }

                if (smallest_in_col != 0 && smallest_in_col !=
                  Double.PositiveInfinity) {
                    for (int row = 0; row < data.GetLength(0); row++) {
                        data[row, col] -= smallest_in_col;
                    }
                    bound += smallest_in_col;
                }
            }

            return bound;
        }

        // This is for our priority Queue. Linked lists are basically awesome. Always.
        private class LinkedListNode {
            public LinkedListNode previous;
            public LinkedListNode next;
            public double[,] data;
            public double bound;
            public int[] path;
            public int path_size;
            public LinkedListNode(double b, double[,] d, int[]t, int ps) {
                data = d;
                bound = b;
                path = t;
                path_size = ps;
            }
        }

        // removes to_remove from the list it is a part of
        // O(1)
        private LinkedListNode removeNode(LinkedListNode to_remove, LinkedListNode
          remove_from) {
            BBcurrentStates--;
            if (to_remove.previous == null && to_remove.next == null) {
                remove_from = null;
            }

            if (to_remove.previous != null) {
                to_remove.previous.next = to_remove.next;
            } else {
                remove_from = to_remove.next;
            }

            if (to_remove.next != null) {
                to_remove.next.previous = to_remove.previous;
```

```
            }
            return remove_from;
        }

        // Adds a node to our sorted list O(n)
        private LinkedListNode addNode(LinkedListNode to_add, LinkedListNode 
          priorityQueue) {
            BBstatesCreated++;  // we are adding one; it is created
            BBcurrentStates++;  // and current state is incremented
            if (bssf != null && to_add.path_size > bssf.costOfRoute()) {
                // greater than the best path we have found so far.
                // stop wasting our time
                BBstatesPruned++;       // not adding a node counts as pruning
                BBcurrentStates--;      // when we prune the current states goes down
                return priorityQueue;
            }

            // If we have a full solution, need to check to see if it is the best so 
              far and record it
            if (to_add.path_size == Cities.Length) {
                // we have ourselves a full path
                if (bssf == null || to_add.path_size <= bssf.costOfRoute()) {

                    Route = new ArrayList();
                    for (int i = 0; i < to_add.path_size; i++){
                        Route.Add(Cities[to_add.path[i]]);
                    }
                    bssf = new TSPSolution(Route);
                    BBcount++;
                    BSSFupdates++;
                }
            }

            // If the list we are adding in is null, then we simply add it.
            if (priorityQueue == null) {
                priorityQueue = to_add;
                return priorityQueue;
            }

            // Special case for adding at the beginning
            if (to_add.bound <= priorityQueue.bound) {
                to_add.next = priorityQueue;
                priorityQueue.previous = to_add;
                priorityQueue = to_add;
            }
            else {


                // Simply go until we find the spot in the list.
                LinkedListNode temp = priorityQueue;
                while (temp.next != null && to_add.bound > temp.next.bound)
                {
```

```
                temp = temp.next;
            }

            // Now we have the good place in the list.
            // Time to insert it!
            if (temp.next != null)
            {
                temp.next.previous = to_add;
            }
            to_add.previous = temp;
            to_add.next = temp.next;
            temp.next = to_add;
        }

        // ------------------------------------------------------------------------------
        // SOME AGGRESSIVE PRUNING HERE
        // If our thing we are adding is a solution, delete ALL nodes with worse bounds.
        if (to_add.path_size == Cities.Length) {
            LinkedListNode temp2 = to_add.next;
            while (temp2 != null) {
                BBstatesPruned++;
                BBcurrentStates--;
                temp2 = temp2.next;
            }
            to_add.next = null;                     // It *is* this easy, no?
        }
        // Now we trust the garbage collector
        // ------------------------------------------------------------------------------

        return priorityQueue;
    }

    /////////////////////////////////////////////////////////////////////////////////////////////
    // These additional solver methods will be implemented as part of the group project.
    ////////////////////////////////////////////////////////////////////////////////////////////

    /// <summary>
    /// finds the greedy tour starting from each city and keeps the best (valid) one
    /// </summary>
    /// <returns>results array for GUI that contains three ints: cost of solution, time spent to find solution, number of solutions found during search (not counting initial BSSF estimate)</returns>
    public string[] greedySolveProblem()
```

```
        {
            string[] results = new string[3];

            // TODO: Add your implementation for a greedy solver here.
            Stopwatch timer = new Stopwatch();
            timer.Start();
            Route = new ArrayList();


            greedySolveSub();
            // At this point, we have all the cities.
            // We have a problem though. What if the last city does not go back to the
               first one???
            while (((City) Route[Route.Count-1]).costToGetTo((City) Route[0]) ==
              Double.PositiveInfinity) {
                greedySolveSub();
            }

            bssf = new TSPSolution(Route);
            timer.Stop();

            results[COST] = costOfBssf().ToString();    // load results into array
              here, replacing these dummy values
            results[TIME] = timer.Elapsed.ToString();
            results[COUNT] = "1";

            return results;
        }

        private void greedySolveSub() {
            // So because adding and removing from hash set are O(1), this feels
              faster.
            HashSet<int> citiesLeft = new HashSet<int>();
            for (int i = 0; i < Cities.Length; i++)
            {
                citiesLeft.Add(i);
            }

            City currentCity = Cities[0];
            while (citiesLeft.Count > 0)
            {
                double lowest = Double.PositiveInfinity;
                int lowestc = 0;
                foreach (int c in citiesLeft)
                {
                    if (currentCity.costToGetTo(Cities[c]) < lowest)
                    {
                        lowest = currentCity.costToGetTo(Cities[c]);
                        lowestc = c;
                    }
                }
                Route.Add(Cities[lowestc]);
```

```
                citiesLeft.Remove(lowestc);
            }
        }



        public string[] fancySolveProblem()
        {
            string[] results = new string[3];

            // TODO: Add your implementation for your advanced solver here.

            results[COST] = "not implemented";    // load results into array here, 
              replacing these dummy values
            results[TIME] = "-1";
            results[COUNT] = "-1";

            return results;
        }
        #endregion
    }

}
```