Yazan Halawa
3/30/16
Lab 5-TSP Report

Lab Code:

```
//////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////// State Class ///////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////
    #region StateClass
    /**
    * This class represents the state at each node in the branch and bound algorithm.
    *
    */
    public class State
    {
       private ArrayList path;
       private double lowerBound;
       private double priority;
       private double[,] costMatrix;

       /**
       * Constructor
       */
       public State(ref ArrayList newPath, ref double newLowerBound, ref double[,]
newCostMatrix, int length)
       {
          path = newPath;
          lowerBound = newLowerBound;
          costMatrix = newCostMatrix;
          priority = double.MaxValue;
       }

       /**
       * Functions to Manipulate and return the path
       */
       public ArrayList getPath()
       {
          return path;
       }
       public void addCityToPath(City newCity)
```

```
        {
            path.Add(newCity);
        }

        /**
        * Functions to Manipalte and return the priority
        */
        public double getPriority()
        {
            return priority;
        }
        public void setPriority(double newPriority)
        {
            priority = newPriority;
        }

        /**
        * Functions to set and return the lower bound
        */
        public double getLowerBound()
        {
            return lowerBound;
        }
        public void setLowerBound(double newLowerBound)
        {
            lowerBound = newLowerBound;
        }

        /**
        * Functions to set and return the cost matrix
        */
        public double[,] getCostMatrix()
        {
            return costMatrix;
        }

    }
    #endregion

////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////// Helper Functions ////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////
    #region HelperFunctions
/**
```

```
        * Helper Function to create a key value for a state for use in priority queue
        * Time Complexity: O(1) as it only perform a mathematical division, and if we
        * assume n is the size of the input then those would be constant operations
        * Space Complexity: O(1) as it does not create any extra data structures that depend on the
size of the input.
        */
        double calculateKey(int numofCitiesLeft, double lowerBound)
        {
            // If there are no cities left, just use the lower bound
            if (numofCitiesLeft < 1)
                return lowerBound;
            else
                return lowerBound / (Cities.Length - numofCitiesLeft);
        }
    /**
        * Helper Function to create an initial greedy solution to assign BSSF to in the beginning
        * Time Complexity: O(n^2) because for each city it is iterating over all the cities in the list.
so n is the
        * number of cities, or rather |V|
        * Space Complexity: O(n) as it creates an Array list(Route) of size equal to the number of
cities in the graph where
        * n is the number of cities, or rather |V|
        */
        double createGreedyInitialBSSF()
        {
            // Create variables to track progress
            Route = new ArrayList();
            Route.Add(Cities[0]);
            int currCityIndex = 0;

            // While we haven't added |V| edges to our route
            while (Route.Count < Cities.Length)
            {
                double minValue = double.MaxValue;
                int minIndex = 0;

                // Loop over all the cities and find the one with min cost to get to
                for (int i = 0; i < Cities.Length; i++)
                {
                    // We don't want to be checking ourselves because that will be the minimum and it
won't be a tour
                    if (currCityIndex != i)
                    {
```

```
                // We don't want to add a city that we already added because that forms a cycle
and it won't be a tour
                if (!Route.Contains(Cities[i]))
                {
                    double tempValue = Cities[currCityIndex].costToGetTo(Cities[i]);
                    if (tempValue < minValue)
                    {
                        if (Route.Count == Cities.Length - 1 && Cities[i].costToGetTo(Cities[0]) ==
double.MaxValue)
                        {
                            continue;
                        }
                        minValue = tempValue;
                        minIndex = i;
                    }
                }
            }
        }

        // Add the min edge to the Route by adding the destination city
        currCityIndex = minIndex;
        Route.Add(Cities[currCityIndex]);
    }

    // Once we have a complete tour, we set out BSSF to it as an upper bound for all
solutions to follow
    bssf = new TSPSolution(Route);
    return bssf.costOfRoute();
}
/**
* Helper Function to initially set up a cost matrix at a current state
* Time Complexity: O(n) as it iterates over one row and one column in the matrix and n
would be the length of
* the row/column which is the number of cities in the graph, or rather |V|
* Space Complexity: O(1) as all the data is passed by reference and the function does not
create extra
* data structures that depend on the size of the input.
*/
void setUpMatrix(ref double[,] costMatrix, int indexOfParent, int indexOfChild, ref double
lowerBound)
{
    if (costMatrix[indexOfParent, indexOfChild] != double.MaxValue)
        lowerBound += costMatrix[indexOfParent, indexOfChild];
    // Make sure to set all costs coming from the currState to infinity
```

```
            for (int column = 0; column < Cities.Length; column++)
            {
                costMatrix[indexOfParent, column] = double.MaxValue;
            }
            // Make sure to set all costs coming into the child State to infinity
            for (int row = 0; row < Cities.Length; row++)
            {
                costMatrix[row, indexOfChild] = double.MaxValue;
            }
            // Make sure to set the cost of going from child state back to parent to infinity as we
don't want cycles
            costMatrix[indexOfChild, indexOfParent] = double.MaxValue;
        }
        /**
        * Helper function to reduce a cost matrix and calculate the lower bound of the
corresponding state
        * Time Complexity: O(n^2) because it iterates over all the cells in an nxn matrix 2 times (so
really it is
        * O(4n^2) but the constant is ommitted.
        * Space Complexity: O(1) as the matrix is passed by reference and so the function does not
create
        * any data structures that depend on the size of the input
        */
        double reduceMatrix(ref double[,] costMatrix)
        {
            double lowerBound = 0;
            // Loop through the rows, find the min value for each, and subtract it from every other
cell value
            for (int row = 0; row < Cities.Length; row++)
            {
                // Find The Minimum value in the row
                double minVal = double.MaxValue;
                for (int column = 0; column < Cities.Length; column++)
                {
                    if (costMatrix[row, column] < minVal)
                    {
                        minVal = costMatrix[row, column];
                    }
                }

                // Subtract the min value from each cell if the min value is not infinity
                if (minVal != 0 && minVal != double.MaxValue)
                {
                    lowerBound += minVal;
```

```
            for (int column = 0; column < Cities.Length; column++)
            {
                if (costMatrix[row, column] != double.MaxValue)
                    costMatrix[row, column] -= minVal;
            }
        }
    }

    // Loop through the columns, find the minvalue for each and subtract it from every other
cell value
    for (int column = 0; column < Cities.Length; column++)
    {
        // Find The Minimum value in the row
        double minVal = double.MaxValue;
        for (int row = 0; row < Cities.Length; row++)
        {
            if (costMatrix[row, column] < minVal)
            {
                minVal = costMatrix[row, column];
            }
        }
        // Subtract the min value from each cell if the min value is not infinity
        if (minVal != 0 && minVal != double.MaxValue)
        {
            lowerBound += minVal;
            for (int row = 0; row < Cities.Length; row++)
            {
                if (costMatrix[row, column] != double.MaxValue)
                    costMatrix[row, column] -= minVal;
            }
        }
    }

    return lowerBound;
}
/**
* Helper function that will create the initial State starting at the first City in the list.
* Time Complexity: summing up the time complexities of the parts of this function as
explained in the code:
* n^2 + 1 + n^2 = O(n^2) because constants are ommitted
* Space Complexity: summing up the space complexities of the parts of this function as
explained in the code:
* n^2 + 1 + 1 = O(n^2) because constants are ommitted.
*/
```

```
        State createInitialState()
        {
            /* First Create the initial cost matrix based on the costs to get from each city to the other
*/
            /* Loop through all the matrix cells, if we are at the diagonal then set the cost to infinity,
else set it to the
              cost to get from city at rows(i) to city at columns(j) */
            /* This part takes O(n^2) time and O(n^2) space*/
            double[,] initialCostMatrix = new double[Cities.Length, Cities.Length];
            for (int i = 0; i < Cities.Length; i++)
            {
                for (int j = 0; j < Cities.Length; j++)
                {
                    if (i == j)
                        initialCostMatrix[i, j] = double.MaxValue;
                    else
                        initialCostMatrix[i, j] = Cities[i].costToGetTo(Cities[j]);
                }
            }
            /* Second the path will be simply the starting node */
            /* This part takes O(1) time and space */
            ArrayList path = new ArrayList();
            path.Add(Cities[0]);

            /* Third Calculate the lower Bound */
            /* This part takes O(n^2) time and O(1) space */
            double lowerBound = reduceMatrix(ref initialCostMatrix);
            return new State(ref path, ref lowerBound, ref initialCostMatrix, Cities.Length);
        }
        #endregion

////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////// BB Algorithm ///////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////
        #region MainBBAlgorithm
        /// <summary>
        /// performs a Branch and Bound search of the state space of partial tours
        /// stops when time limit expires and uses BSSF as solution
        /// Time Complexity: O((n^2)*(2^n) as that is the most dominant factor in the code, and it
is a result
        /// of the loop, for more details scroll to the comment above the loop in the function.
        /// Space Complexity: O((n^2)*(2^n) as that is the most dominant factor in the code, and it
is a result
        /// of the loop, for more details scroll to the comment above the loop in the function.
```

```
        /// </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];

            // Helper variables
            /* This part of the code takes O(1) space and time as we are just initializing some data */
            int numOfCitiesLeft = Cities.Length;
            int numOfSolutions = 0;
            int numOfStatesCreated = 0;
            int numOfStatesNotExpanded = 0;

            // Initialize the time variable to stop after the time limit, which is defaulted to 60 seconds
            /* This part of the code takes O(1) space and time as we are just initializing some data */
            DateTime start = DateTime.Now;
            DateTime end = start.AddSeconds(time_limit/1000);

            // Create the initial root State and set its priority to its lower bound as we don't have any extra info at this point
            /* This part of the code takes O(n^2) space and time as explained above */
            State initialState = createInitialState();
            numOfStatesCreated++;
            initialState.setPriority(calculateKey(numOfCitiesLeft - 1, initialState.getLowerBound()));

            // Create the initial BSSF Greedily
            /* This part of the code takes O(n^2) time and O(n) space as explained above */
            double BSSFBOUND = createGreedyInitialBSSF();

            // Create the queue and add the initial state to it, then subtract the number of cities left
            /* This part of the code takes O(1) time since we are just creating a data structure and
            O(1,000,000) space which is just a constant so O(1) space as well*/
            PriorityQueueHeap queue = new PriorityQueueHeap();
            queue.makeQueue(Cities.Length);
            queue.insert(initialState);

            // Branch and Bound until the queue is empty, we have exceeded the time limit, or we found the optimal solution
            /* This loop will have a iterate n! times without pruning. However, since we are pruing and removing a lot of states, it just iterates 2^n times approximately with expanding and pruning for each state, then for each state it
```

```
        does O(n^2) work by reducing the matrix, so over all O((n^2)*(2^n)) time and space as
well as it creates a nxn
        matrix for each state*/
        while (!queue.isEmpty() && DateTime.Now < end && queue.getMinLB() != BSSFBOUND)
        {
            // Grab the next state in the queue
            State currState = queue.deleteMin();

            // check if lower bound is less than the BSSF, else prune it
            if (currState.getLowerBound() < BSSFBOUND)
            {
                // Branch and create the child states
                for (int i = 0; i < Cities.Length; i++)
                {
                    // First check that we haven't exceeded the time limit
                    if (DateTime.Now >= end)
                        break;

                    // Make sure we are only checking cities that we haven't checked already
                    if (currState.getPath().Contains(Cities[i]))
                        continue;

                    // Create the State
                    double[,] oldCostMatrix = currState.getCostMatrix();
                    double[,] newCostMatrix = new double[Cities.Length, Cities.Length];
                    // Copy the old array in the new one to modify the new without affecting the old
                    for (int k = 0; k < Cities.Length; k++)
                    {
                        for (int l = 0; l < Cities.Length; l++)
                        {
                            newCostMatrix[k, l] = oldCostMatrix[k, l];
                        }
                    }
                    City lastCityinCurrState = (City)currState.getPath()[currState.getPath().Count-1];
                    double oldLB = currState.getLowerBound();
                    setUpMatrix(ref newCostMatrix, Array.IndexOf(Cities, lastCityinCurrState), i, ref
oldLB);
                    double newLB = oldLB + reduceMatrix(ref newCostMatrix);
                    ArrayList oldPath = currState.getPath();
                    ArrayList newPath = new ArrayList();
                    foreach (City c in oldPath)
                    {
                        newPath.Add(c);
                    }
```

```
                newPath.Add(Cities[i]);
                State childState = new State(ref newPath, ref newLB, ref newCostMatrix,
Cities.Length);
                numOfStatesCreated++;

                // Prune States larger than the BSSF
                if (childState.getLowerBound() < BSSFBOUND)
                {
                   City firstCity = (City)childState.getPath()[0];
                   City lastCity = (City)childState.getPath()[childState.getPath().Count-1];
                   double costToLoopBack = lastCity.costToGetTo(firstCity);

                   // If we found a solution and it goes back from last city to first city
                   if (childState.getPath().Count == Cities.Length && costToLoopBack !=
double.MaxValue)
                   {
                      childState.setLowerBound(childState.getLowerBound() + costToLoopBack);
                      bssf = new TSPSolution(childState.getPath());
                      BSSFBOUND = bssf.costOfRoute();
                      numOfSolutions++;
                      numOfStatesNotExpanded++; // this state is not expanded because it is not
put on the queue
                   }
                   else
                   {
                      // Set the priority for the state and add the new state to the queue
                      numOfCitiesLeft = Cities.Length - childState.getPath().Count;
                      childState.setPriority(calculateKey(numOfCitiesLeft,
childState.getLowerBound()));
                      queue.insert(childState);
                   }
                }
                else
                {
                   numOfStatesNotExpanded++; // States that are pruned are not expanded
                }
             }
          }
          currState = null;
       }
       numOfStatesNotExpanded += queue.getSize(); // if the code terminated before queue is
empty, then those states never got expanded
       Console.WriteLine("Number of states generated: " + numOfStatesCreated);
       Console.WriteLine("Number of states not Expanded: " + numOfStatesNotExpanded);
```

```
            end = DateTime.Now;
            TimeSpan diff = end - start;
            double seconds = diff.TotalSeconds;
            results[COST] = System.Convert.ToString(bssf.costOfRoute());    // load results into array
here, replacing these dummy values
            results[TIME] = System.Convert.ToString(seconds);
            results[COUNT] = System.Convert.ToString(numOfSolutions);

            return results;
        }
        #endregion

////////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////// Priority Queue /////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////
        #region PriorityQueue
        public sealed class PriorityQueueHeap
        {
            private int capacity;
            private int count;
            private State[] states;
            public PriorityQueueHeap()
            {
            }

            /**
            * This functions returns whether the queue is empty or not. Time and Space = O(1) as it
only involves an int comparison
            */
            public bool isEmpty()
            {
                return count == 0;
            }

            /**
            * This function returns the number of items in the queue
            */
            public int getSize()
            {
                return count;
            }
            /**
            * This function returns the lower bound of the first item in the queue
            */
```

```
        public double getMinLB()
        {
           return states[1].getLowerBound();
        }

        /**
        * This method creates an array to implement the queue. Time and Space Complexities
are both O(1) as the queue always has the same max capacity of 1,000,000 which is a constant
        */
        public void makeQueue(int numOfNodes)
        {
           states = new State[1000000];
           capacity = numOfNodes;
           count = 0;
        }

        /**
        * This method returns the index of the element with the minimum value and removes it
from the queue.
        * Time Complexity: O(log(|V|)) because removing a node is constant time as we have its
position in
        * the queue, then to readjust the heap we just bubble up the min value which takes as
long as
        * the depth of the tree which is log(|V|), where |V| is the number of nodes
        * Space Complexity: O(1) because we don't create any extra variables that vary with the
size of the input.
        */
        public State deleteMin()
        {
           // grab the node with min value which will be at the root
           State minValue = states[1];
           states[1] = states[count];
           count--;
           // fix the heap
           int indexIterator = 1;
           while (indexIterator <= count)
           {
              // grab left child
              int smallerElementIndex = 2 * indexIterator;

              // if child does not exist, break
              if (smallerElementIndex > count)
                 break;
```

```
            // if right child exists and is of smaller value, pick it
            if (smallerElementIndex + 1 <= count && states[smallerElementIndex +
1].getPriority()
                                        < states[smallerElementIndex].getPriority())
            {
               smallerElementIndex++;
            }

            if (states[indexIterator].getPriority() > states[smallerElementIndex].getPriority())
            {
               // set the node's value to that of its smaller child
               State temp = states[smallerElementIndex];
               states[smallerElementIndex] = states[indexIterator];
               states[indexIterator] = temp;
            }

            indexIterator = smallerElementIndex;
         }
         // return the min value
         return minValue;
      }

      /**
      * This method updates the nodes in the queue after inserting a new node
      * Time Complexity: O(log(|V|)) as reording the heap works by bubbling up the min value
to the top
      * which takes as long as the depth of the tree which is log|V|.
      * Space Complexity: O(1) as it does not create any extra variables that vary with the size
of the input.
      */
      public void insert(State newState)
      {
         // update the count
         count++;
         states[count] = newState;

         // as long as its parent has a larger value and have not hit the root
         int indexIterator = count;
         while (indexIterator > 1 && states[indexIterator / 2].getPriority() >
states[indexIterator].getPriority())
         {
            // swap the two nodes
            State temp = states[indexIterator / 2];
            states[indexIterator / 2] = states[indexIterator];
```

```
            states[indexIterator] = temp;

            indexIterator /= 2;
        }
    }
}
#endregion
```

Time and Space Complexity Analysis

The time and space complexity analysis is demonstrated in the comments above the functions in the code. Please scroll up for the analysis.

State Data Structures

I created a State class that holds the NxN matrix of city costs, an N array that holds the partial path of nodes currently in the state, a double to represent the lower priority, and a double to represent the priority key for a state for use later in the queue. The explanation for the key is associated with the section on the Queue. N represents the number of cities in the problem.

The Priority Queue

The Queue is very similar to the queue we had in lab 3. It is a heap implementation consisting of an inner array that holds the items, then inserting and deleting would be adding the item to either the beginning of the array and bubbling the max down, or inserting to the bottom of the array and bubbling the min value up. That way it would be a min heap where the item with the minimum key (highest priority) is first in the array. I modified it to work with this lab like the following: First of all, I deleted the decrease key function and the second array of index pointers as we won't need to update priorities later and so we don't need to track positions of items in the queue. Then I changed the main array to hold States instead of integers. Once that was done, all I needed was some way of specifying the priorities of the items in the queue as we can't use distances like we did in lab 4. So what I did is to create a function (CalculateKey) that takes in the lower bound (computed by reducing the array of the parent), and the number of states left not added to the partial path of the state. The reason I did this is we can't just use the lower bound because it is important to find full solutions so we can update the BSSF and

prune states, and be much faster. So The key had to be a combination of the lower bound and the tree depth somehow. Thus, I had the priority key be the lowerbound/depth(depth = citites.length – numCititesLeft). That way the lower the value of that key, the higher the priority of the state in the queue. Since our depth is at the bottom of the quotient, the higher it is, the lower the key is, which means the farther we are from a leaf node or a full solution, the less priority it has. Thus, states with lower bound/less number of cities left (meaning closer to a leaf node), are the states we visit first, which makes sense for our purposes. This is all documented in the code above.

Initial BSSF Approach

The approach was to start at the first city in the list, then greedily add each next minimum edge to our Route. So basically for each node we check the whole list of cities left and pick the closest one each time, resulting in time and space of O(n^2). Naturally, for each city, I had to make sure I don't add an edge to the same city, or add an edge to a city already in the route which would create a cycle and it won't be a complete tour. Then the BSSFBound would be the cost of that final route.

Empirical Data

# Cities	Seed	Running Time (sec.)	Cost of best tour found (*= optimal)	Max # of stored states at a given time	# of BSSF updates	Total # of states created	Total # of states pruned
15	20	0.203	2430*	62	1	33849	28776
16	902	0.094	3223*	233	73	12774	10514
20	166	14.7	4053*	156	7	1371272	1229962
25	207	0.068	5219*	329	1	3842	3065
30	30	29.65	5628*	1009	2	1868403	1595128
35	99	60	5796	15853	1	2586038	2143250
40	177	60	6151	4707	0	2053803	1781640
45	201	60	6503	3016	0	1952132	1749771
47	133	60	6638	14310	0	1669050	1409966
50	2	60	6153	1180	4	1514041	1388350

Analysis of Empirical table

It makes sense that as the size of the input gets larger, that there is factorially more states, and so it needs more time to find a good solution. This is because the time complexity is O ($n^2 * 2^n$) and so with bigger input, the complexity scales up really bad, due to that exponential term. Naturally because we limit the time to 30 seconds, it just returns the best solution it found. It might have found a couple, or it might have just not had enough time to find any and so it settles for the initial greedy solution (0 BSSF updates). We also notice that as the number of states generated increases, the time to solve a problem increases, while as the number of states pruned increases, the time decreases. Now naturally because number of states generated, and pruned both increase together, they kind of counteract each other. Now there are some exceptions where even though the size of the problem got larger, the time to solve it got smaller, and that depends on the generated costs between the cities which depends on the seed. We also notice that as the size of the problem gets bigger, generally, there are more states to prune, so that number increases as well.