Scott Christensen

03/30/16

C.S. 312

Project 5 Design Report

1. My Code:

```
public class HeapImplementation
{
    public State[] allNodes;
    readonly State[] data;
    readonly double[] distances;
    public int count;


    //Creates a new, empty priority queue with the specified capacity.
    public HeapImplementation(int capacity)
    {
        capacity = 10000000;
        allNodes = new State[capacity];
        data = new State[capacity];
        distances = new double[capacity];
        count = 0;
    }

    //Insert
    //Time: O(log v) Because your using a Heap that must reorder itself with
every add.
    //Space: O(1) Because your adding one thing.

    //Adds an item to the queue.  Is position is determined by its priority
relative to the other items in the queue.
    public void Add(State item, double priority)
    {
        if (count == data.Length)
            throw new Exception("Heap capacity exceeded");

        // Add the item to the heap in the end position of the array (i.e. as a
leaf of the tree)
        int position = count++;
        data[position] = item;
        item.QueuePosition = position;
        distances[position] = priority;
        // Move it upward into position, if necessary
        MoveUp(position);
    }

    public void AddToAllNodes(State item)
    {
        allNodes[item.pointsIndex] = item;
    }

    //DeleteMin
```

```
            //Time: O(log v) Because we're using a Heap and it must rebalance itself
after every deletion.
            //Space: O(1) Because we only change one thing.

            //Extracts the item in the queue with the minimal priority value.
            public State ExtractMin()
            {
                allNodes[data[0].pointsIndex].distance =
distances[data[0].QueuePosition];
                State minNode = data[0];
                Swap(0, count - 1);
                count--;
                MoveDown(0);
                return minNode;
            }

            //DecreaseKey
            //Time: O(log v) Because if you change the weight of one of the Nodes, the
Hea needs to rebalance itself accordingly.
            //Space: O(log v) Because you are changing one thing.
            //Reduces the priority of a node already in the queue.
            public void DecreasePriority(int index, double priority)
            {
                int position = allNodes[index].QueuePosition;
                while ((position > 0) && (distances[Parent(position)] > priority))
                {
                    int original_parent_pos = Parent(position);
                    Swap(original_parent_pos, position);
                    position = original_parent_pos;
                }
                distances[position] = priority;
            }

            //Moves the node at the specified position upward, it it violates the Heap
Property.
            //This is the while loop from the HeapInsert procedure in the slides.
            void MoveUp(int position)
            {
                while ((position > 0) && (distances[Parent(position)] >
distances[position]))
                {
                    int original_parent_pos = Parent(position);
                    Swap(position, original_parent_pos);
                    position = original_parent_pos;
                }
            }

            //Moves the node at the specified position down, if it violates the Heap
Property
            void MoveDown(int position)
            {
                int lchild = LeftChild(position);//look at it's left child and get its
value
                int rchild = RightChild(position);//look at it's right child and get its
value
                int largest = 0;
                if ((lchild < count) && (distances[lchild] < distances[position]))
                {
```

```
            largest = lchild;
        }
        else
        {
            largest = position;
        }
        if ((rchild < count) && (distances[rchild] < distances[largest]))
        {
            largest = rchild;
        }
        if (largest != position)
        {
            Swap(position, largest);
            MoveDown(largest);
        }
    }

    //Get the number of items waiting in queue
    public int Count
    {
        get
        {
            return count;
        }
    }

    //Swaps the nodes at the respective positions in the heap
    //Updates the nodes' position properties accordingly.
    void Swap(int position1, int position2)
    {
        State temp = data[position1];
        data[position1] = data[position2];
        data[position2] = temp;
        data[position1].QueuePosition = position1;
        data[position2].QueuePosition = position2;

        double temp2 = distances[position1];
        distances[position1] = distances[position2];
        distances[position2] = temp2;
    }

    //Gives the position of a node's parent, the node's position in the queue.
    static int Parent(int position)
    {
        return (position - 1) / 2;
    }

    //Returns the position of a node's left child, given the node's position.
    static int LeftChild(int position)
    {
        return 2 * position + 1;
    }

    //Returns the position of a node's right child, given the node's position.
    static int RightChild(int position)
    {
        return 2 * position + 2;
    }
```

```
        }

        /// <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];
            Route = new ArrayList();
            int length = Cities.Length;
            //This is the main costs matrix of all cities to all others
            double[,] costs = new double[length, length];
            Stopwatch stopWatch = new Stopwatch();
            //the data structure to store and orginize our states
            HeapImplementation queue = new HeapImplementation(length);
            double upBound = double.PositiveInfinity;

            //This is used to give priority to the level of a state, rather than it's
neighbors. Does so proportional to it's level.
            int priorityFactor = 10 * Cities.Length;
            int statesCreated = 0;
            int statesPurged = 0;
            int maxStoredStates = 0;
            int numBssfUpdates = 0;

            stopWatch.Start();

            /*First, we make the cost matrix by iterating through the Cities 2d array
             * Time Complexity: O(n^2) the size of city matrix.
             * Space Complexity: O(n^2) the size of city matrix.
             */
            for (int row = 0; row < length; row++)
            {
                for (int col = 0; col < length; col++)
                {
                    if (row == col)
                    {
                        costs[row, col] = double.PositiveInfinity;
                    }
                    else
                    {
                        costs[row, col] = Cities[row].costToGetTo(Cities[col]);
                    }
                }
            }


            /*Do greedy alg to get BSSF. Starting from every city, we find the greedy
solution from it and save the best solution.
             * Time Complexity: O(n^3) because we run an n^2 algorithm N times.
             * Space Complexity: O(n^2) because we only need to store the original costs
matrix and follow it.
             */
```

```
            ArrayList bestRoute = new ArrayList();
            for (int city = 0; city < length; city++)
            {
                ArrayList currRoute = new ArrayList();
                currRoute.Add(city);
                int startPosition = city;
                double currGreedyupBound = greedyRecursive(city, city, costs, currRoute,
length, 0);
                if (currGreedyupBound < upBound)
                {
                    upBound = currGreedyupBound;
                    bestRoute.Clear();
                    bestRoute = currRoute;
                }
            }

            int numSolutions = 1;//how many solutions we found. We found one with the
greedy algorithm.
            double elapsedMil = stopWatch.ElapsedMilliseconds;


            /* First we create a starting State for the priority queue.*/
            int[] state1_tour = new int[1];
            state1_tour[0] = 0;
            State state0 = new State(0, 1, costs, 0, state1_tour, length);
            statesCreated++;
            /* Add the first state the the priority queue
             * Time Complexity: O(nlog(n)) because we store the node in a tree structure
and rebalance.
             * Space Complexity: O(n) because we store every city
             */
            queue.AddToAllNodes(state0);
            queue.Add(state0, state0.lowerBound - priorityFactor * state0.level *
state0.level);

            /* Go through the best greedy route and update the bssf object so that we hav
an initial BSSF to compare the children states against for pruning.
             * Time Complexity: O(n) because iterate over every city to get a full cycle.
             * Space Complexity: O(n) because iterate over every city to get a full cycle
and then store it.
             */
            foreach (int i in bestRoute)
            {
                Route.Add(Cities[i]);
            }
            bssf = new TSPSolution(Route);

            //if the greedy solution's upBound it = to the first state's lowerbound, we
know that we have the optimal solution.
            if (upBound == state0.lowerBound)
            {
                Console.WriteLine("States Created: " + statesCreated);
                Console.WriteLine("States Purged: " + statesPurged);
                Console.WriteLine("Max # of Stored States at a given time: " +
maxStoredStates);
                Console.WriteLine("Times BSSF Updated: " + numBssfUpdates);
                Console.WriteLine("Priority Factor: " + priorityFactor);
```

```
                results[COST] = costOfBssf().ToString();
                results[TIME] = stopWatch.Elapsed.ToString();
                results[COUNT] = numSolutions.ToString();
                return results;
            }

            /* The main branch and bound algorithm of our project.
             * Time Complexity: O(n^2(n-1!)) The defined branch and bound big O worst
case, or brute force.
             * Space Complexity: O(n^2(n-1!))) The defined branch and bound big O worst
case storage, or brute force.
             */
            while (stopWatch.ElapsedMilliseconds < time_limit && queue.Count > 0)
            {
                if (queue.Count > maxStoredStates) maxStoredStates = queue.Count;
                /* Grab the smallest state to examine.
                * Time Complexity: O(n) because iterate over every city to get a full
cycle.
                * Space Complexity: O(n) because iterate over every city to get a full
cycle and then store it.
                */
                State state = queue.ExtractMin();
                //if the lowerbound = the upper bound, its the optimum solution
                if (state.lowerBound == upBound)
                {
                    Console.WriteLine("States Created: " + statesCreated);
                    Console.WriteLine("States Purged: " + statesPurged);
                    Console.WriteLine("Max # of Stored States at a given time: " +
maxStoredStates);
                    Console.WriteLine("Times BSSF Updated: " + numBssfUpdates);
                    Console.WriteLine("Priority Factor: " + priorityFactor);


                    results[COST] = costOfBssf().ToString();
                    results[TIME] = stopWatch.Elapsed.ToString();
                    results[COUNT] = numSolutions.ToString();
                    return results;
                }
                else if(state.lowerBound > upBound)//we dont consider following this path
if it's worse that the upBound
                {
                    statesPurged++;
                }
                else
                {
                    if (state.tour.Length == length + 1)//if it's a leaf..
                    {
                        numSolutions++;
                        if (state.lowerBound < upBound)//..and it's bound is less that
the current best so far, save it and the route.
                        {
                            Route = new ArrayList();

                            /* Go through the best greedy route and update the bssf
object so that we hav an initial BSSF to compare the children states against for pruning.
                            * Time Complexity: O(n) because iterate over every city to
get a full cycle.
```

```
                            * Space Complexity: O(n) because iterate over every city to
get a full cycle and then store it.
                            */
                            for (int i = 0; i < state.tour.Length; i++)
                            {
                                Route.Add(Cities[state.tour[i]]);
                            }
                            bssf = new TSPSolution(Route);
                            numBssfUpdates++;
                            upBound = state.lowerBound;
                        }
                    }
                    else//if it's not a leaf, we must get the children and add em.
                    {
                        /* get children of the currentState and add them to the queue
conditionally if they are not prunable.
                         * Time Complexity: O(n^2) because we go thorugh all of the
children and while also getting tour from each child
                         * Space Complexity: O(n^2) the storeage of all the children and
their tours
                         */
                        for (int col = 0; col < length; col++)
                        {
                            double currCost = costs[state.pointsIndex, col];
                            if (!currCost.Equals(double.PositiveInfinity))
                            {
                                int[] currTour = new int[state.level + 1];
                                //get the tour from this child
                                for (int i = 0; i < state.tour.Length; i++)
                                {
                                    currTour[i] = state.tour[i];
                                }
                                currTour[state.level] = col;

                               /* build a state for each child. */
                                State childState = new State(col, state.level + 1,
state.cost, state.lowerBound, currTour, length);
                                statesCreated++;
                                queue.AddToAllNodes(childState);
                                //we will see if the child is worth adding to the queue
or not.
                                if (childState.lowerBound < upBound)
                                {
                                    /* Add the first state the the priority queue
                                    * Time Complexity: O(nlog(n)) because we store the
node in a tree structure and rebalance.
                                    * Space Complexity: O(n) because we store every city
                                    */
                                    queue.Add(childState, childState.lowerBound -
priorityFactor * childState.level * childState.level);
                                }
                                else if (childState.lowerBound > upBound)
                                {
                                    //prune the child, we arn't going down that path
                                    statesPurged++;
                                }
                            }
                        }
```

```
                    }
                }
            }

            stopWatch.Stop();

            Console.WriteLine("States Created: " + statesCreated);
            Console.WriteLine("States Purged: " + statesPurged);
            Console.WriteLine("Max # of Stored States at a given time: " +
maxStoredStates);
            Console.WriteLine("Times BSSF Updated: " + numBssfUpdates);
            Console.WriteLine("Priority Factor: " + priorityFactor);

            results[COST] = costOfBssf().ToString();    // load results into array here,
replacing these dummy values
            results[TIME] = stopWatch.Elapsed.ToString();
            results[COUNT] = numSolutions.ToString();
            return results;
        }

        /* The recursive greedy algorithm. It travels throught the graph and takes the
greedy path at each step.
         * Time Complexity: O(n^2) because we iterate over every city for each city.
         * Space Complexity: O(n^2) because we store the costs matrix.
         */
        private double greedyRecursive(int startPosition, int currRow, double[,] costs,
ArrayList currRoute, int numCities, double routeCost)
        {
            double minTo = double.PositiveInfinity;
            int to = -1;
            //find the best untravelled to city to go to.
            for (int i = 0; i < numCities; i++)
            {
                //if we found a connection, save it.
                if (costs[currRow, i] < minTo && !currRoute.Contains(i))
                {
                    minTo = costs[currRow, i];
                    to = i;
                }
            }

            if (to == -1)
            {
                if (costs[currRow, startPosition] < double.PositiveInfinity &&
currRoute.Count == numCities)
                {
                    routeCost += costs[currRow, startPosition];
                    return routeCost;
                }
                return double.PositiveInfinity;
            }

            currRoute.Add(to);
            routeCost += minTo;
            currRow = to;
            return greedyRecursive(startPosition, currRow, costs, currRoute, numCities,
routeCost);
        }
```

```
////////////////////////////////////////////////////////////////////////////////////
////
        // 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];
            Route = new ArrayList();
            int length = Cities.Length;
            double[,] costs = new double[length, length];
            Stopwatch stopWatch = new Stopwatch();
            HeapImplementation queue = new HeapImplementation(length);
            double upBound = double.PositiveInfinity;

            stopWatch.Start();

            for (int row = 0; row < length; row++)
            {
                for (int col = 0; col < length; col++)
                {
                    if (row == col)
                    {
                        costs[row, col] = double.PositiveInfinity;
                    }
                    else
                    {
                        costs[row, col] = Cities[row].costToGetTo(Cities[col]);
                    }
                }
            }

            //Do greedy alg to get BSSF
            ArrayList bestRoute = new ArrayList();
            for (int city = 0; city < length; city++)
            {
                ArrayList currRoute = new ArrayList();
                currRoute.Add(city);
                int startPosition = city;
                double currGreedyupBound = greedyRecursive(city, city, costs, currRoute,
length, 0);
                if (currGreedyupBound < upBound)
                {
                    upBound = currGreedyupBound;
                    bestRoute.Clear();
                    bestRoute = currRoute;
                }
```

```
            }

            int numSolutions = 1;//how many  solutions we found.
            double elapsedMil = stopWatch.ElapsedMilliseconds;

            foreach (int i in bestRoute)
            {
                Route.Add(Cities[i]);
            }


            stopWatch.Stop();
            bssf = new TSPSolution(Route);
            results[COST] = costOfBssf().ToString();    // load results into array here,
replacing these dummy values
            results[TIME] = stopWatch.Elapsed.ToString(); ;
            results[COUNT] = numSolutions.ToString();

            return results;
        }

        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
    }

    /* The class we use to store a state in our priority queue. Every time we make one,
it refactor's it's own matrix.
     * Time Complexity: O(n^2) because we iterate over each cell in the given parent
costs matrix.
     * Space Complexity: O(n^2) because we store an editted costs matrix.
     */
    public class State
    {
        public int pointsIndex, QueuePosition, level;
        public double distance, lowerBound;
        public double[,] cost;
        public int[] tour;
        public State(int pot, int level, double[,] parentMatrix, double parentLowerbound,
int[] tour, int numCities)
        {
            this.level = level;
            this.tour = tour;
            this.pointsIndex = pot;
            this.QueuePosition = -1;
            this.distance = 0;
```

```
            //if it is the first state ever, the lower bound is 0 till we calculate it
from the matrix. Should everntually be the greedy upBound
            if (level == 1) this.lowerBound = 0;
            else//it's = the the parent's upper Bound + the cost it takes to get to this
city + future changes to the matrix.
            {
                this.lowerBound = parentLowerbound + parentMatrix[tour[tour.Length - 2],
pot];
            }

            this.cost = new double[numCities, numCities];
            /* Here we make an exact copy of the parent matrix of the parent. We do it
this way to avoid pointer editting.
             * Time Complexity: O(n^2) because we iterate over each cell in the given
parent costs matrix.
             * Space Complexity: O(n^2) because we store an editted costs matrix.
             */
            for (int row = 0; row < numCities; row++)
            {
                for (int col = 0; col < numCities; col++)
                {
                    this.cost[row, col] = parentMatrix[row, col];
                }
            }

            if (level > 1)
            {
                int cameFrom = tour[tour.Length - 2];
                /* Change the row to infinity.
                 * Time Complexity: O(n). We iterate over the row.
                 */
                for (int col = 0; col < numCities; col++)
                {
                    this.cost[cameFrom, col] = double.PositiveInfinity;
                }
                /* Change the column to infinity.
                 * Time Complexity: O(n). We iterate over the row.
                 */
                for (int row = 0; row < numCities; row++)
                {
                    this.cost[row, pot] = double.PositiveInfinity;
                }
                //change the reflexive to infinity.
                this.cost[pot, cameFrom] = double.PositiveInfinity;
            }

            /* Now we go through the matrix row by row and save the differneces in the
row's cells.
             * Time Complexity: O(n^2). We iterate over the row and the column.
             * Space Complexity: O(n^2). We store the costs matrix.
             */
            for (int row = 0; row < numCities; row++)
            {
                double rowMin = double.PositiveInfinity;
                for (int col = 0; col < numCities; col++)
                {
                    double currCost = this.cost[row, col];
                    if (!currCost.Equals(double.PositiveInfinity) && currCost < rowMin)
```

```
                    {
                        rowMin = currCost;
                    }
                }
                for (int col = 0; col < numCities; col++)
                {
                    if (!this.cost[row, col].Equals(double.PositiveInfinity))
this.cost[row, col] = cost[row, col] - rowMin;
                }
                if (rowMin < double.PositiveInfinity)
                {
                    this.lowerBound += rowMin;
                }
            }

            /* Now we go through the matrix column by column and save the differneces in
the column's cells if we didn't have a 0 in the column.
            * Time Complexity: O(n^2). We iterate over the row and the column.
            * Space Complexity:  O(n^2). We store the entire costs matrix.
            */
            for (int col = 0; col < numCities; col++)
            {
                bool isZero = false;
                double colMin = double.PositiveInfinity;
                for (int row = 0; row < numCities; row++)
                {
                    double currCost = cost[row, col];
                    if (currCost == 0)
                    {
                        isZero = true;
                        break;
                    }
                    else if (currCost < colMin) colMin = currCost;
                }
                if (!isZero)
                {
                    for (int row = 0; row < numCities; row++)
                    {
                        if (!this.cost[row, col].Equals(double.PositiveInfinity))
this.cost[row, col] = this.cost[row, col] - colMin;
                    }
                    if (colMin < double.PositiveInfinity)
                    {
                        this.lowerBound += colMin;
                    }
                }
            }
        }
    }
}
```

2. Time and Space Complexity:

n = number of cities.

a) Greedy Algorithm for initial BSSF:

- Do greedy algorithm starting with the first city and save the best one.

- Time Complexity O(n^3): because I run an n^2 algorithm of iterating over every city and its connections N times.
- Space Complexity O(n^2): because I store the whole cost matrix.

b) Branch and Bound While Loop:
   - While loop that stops once the time is up or once I have been through the whole branch and bound states tree. Inside here, there are some heap operations, such as insert or extract min, which take O(n *log(n)).

     - Time Complexity O(n^2 * (n-1)!): Because for the worst case, it is creating all of the states for every state when finding our solution. Creating a state takes n^2 time as it changes its individual reduced costs matrix.
     - Space Complexity O(n^2 * (n-1)!): Because for the worst case, it is creating all of the states for every state and every state has its own n*n reduced costs matrix.

c) Conclusion:
   - Entire TSP Algorithm:
     - Time Complexity: O(n^3 + n^2 * (n-1)!) = O(n^2 * (n-1)!) after you drop constants.
     - Space Complexity: O(n^2 + n^2 * (n-1)!) = O(n^2 * (n-1)!) after you drop constants.

3. State Data Structure:

To represent each state that represent the branch and bound tree, I used a class called State and put it into a Heap. Each State when created was given information about what path the branch and bound algorithm had followed up to that point(int point index, int level, double[,] parent cost matrix, parentLowerbound, int[] tour, int numCities). The reduced cost matrix was made using the parent's cost matrix by reducing the rows and columns according to what city it was coming from and what city it is going to. This state also includes the level in the tree, its index in the total cities array, and it's priority in the heap. Priority is based off of its lower bound and level.

4. Priority Queue Data Structure:

To represent a priority queue, I used a binary min heap class called Heap Implementation. In this lab, I only used the Add() and the extractMin() functions. These are all O(n*log(n)) operations because I needed to rebalance the tree with each incoming or outgoing node. When adding, I made a state's priority, which was = (10 * Number of Cities) * State.Level^2; Thus, the lower a state went in the recursion, the more precedence it had so as to promote finding new BSSF solutions and pruning instead of bushing out.

5. Initial BSSF Solution:

To represent our initial bssf solution, I used a greedy algorithm. It recursively went through the graph following the shortest path to unvisited cities until it had either visited all cities or failed. If an answer was found, I would return it. I used recursion to account for hard mode because some cities have infinity edges I did this algorithm starting from every city and saved the smallest solution out of all of them. This value was the initial bssf upper bound.

6. Table of Results:

# 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
10	11	0.08	*2147	19	1	2476	2186
12	12	0.3	*3152	42	2	5596	5076
14	13	1.01	*3043	46	2	19837	18281
16	111	0.1	*3362	57	1	804	704
18	15	26.87	*4033	113	7	305892	287738
20	16	60	3922	168	4	587622	556346
25	222	60	4488	211	3	391262	374575
30	18	60	4968	413	1	285998	275665
40	19	60	6578	740	0	153829	149179
50	20	60	6213	1197	0	97875	94687

7. Discussion of Result Table:

Upon examining this table, the resulting data reflects the Big O run time that I had originally thought it would during the design phase. The run time and the number of states stored during that time generally follows a larger than n^2 relationship. The other data, such as number of states created and number of states pruned is also affected by this, but is more dependent the particular seed being used. In other cases, some of the data is completely dependent on the seed, such as the cost of best tour and seed number.

The particular seed used, as was mentioned, can also affect the time it takes to find definite solutions. This can create outliers in our data. In some cases, the branch and bound algorithm could happen to start off by finding a bound that matches the one found by the greedy solution very quickly. In other cases, the algorithm may happen to have its bound updated with every single state path and resulting in a definite solution that takes longer to compute than average.

In conclusion, the data best represents the Big O that was calculated for this entire algorithm, O(n^2 * (n-1)!).