TSP Project 5 Report

Anne Thomas

1. [20] Include your well-commented code.

   See end of document

2. [10] Explain the time and space complexity of your algorithm by showing and summing up the complexity of each subsection of your code.

   The overall worst case time complexity of the algorithm is roughly O(n^2 *(n-1!)); reducing the matrix at each iteration is roughly O(n^2) while, if every state of the tree were generated, this would be done a fraction of (n-1)! times. The space complexity is similar: if all nodes were created, there would be a fraction of (n-1)! nodes with a dominating n^2 of space each (the matrices). However, even in the worst case not all of these would be stored at the same time, as nodes will be removed as new ones are created. However, a lot more nodes are created than are removed at the same time. In reality, there are also a lot of states pruned or never added to the queue, reducing both the time and space complexity.

3. [10] Describe the data structures you use to represent the states.

   I created an object called stateNode which includes the reduced cost matrix, the lower cost bound based on the matrix, an ArrayList with the indexes of the cities in the partial route so far, an ArrayList with the remaining cities, and the distance down the state tree (depth), and the key for the priority queue, which is the lower bound divided by the depth.

4. [5] Describe the priority queue data structure you use and how it works.

   I used a min heap, which sorts by a combination of the depth and lower cost bound of each node as mentioned above. By dividing by the depth, a node with a higher cost but further down the tree is likely given priority over lower cost nodes higher in the tree to ensure leaf nodes and solutions are reached in a timely way. This also allows nodes to sort by lower cost on the same level, while also allowing the possibility of extremely low bounds taking priority over tree depth.

5. [5] Describe your approach for the initial BSSF.

   The initial BSSF comes from a random valid tour.

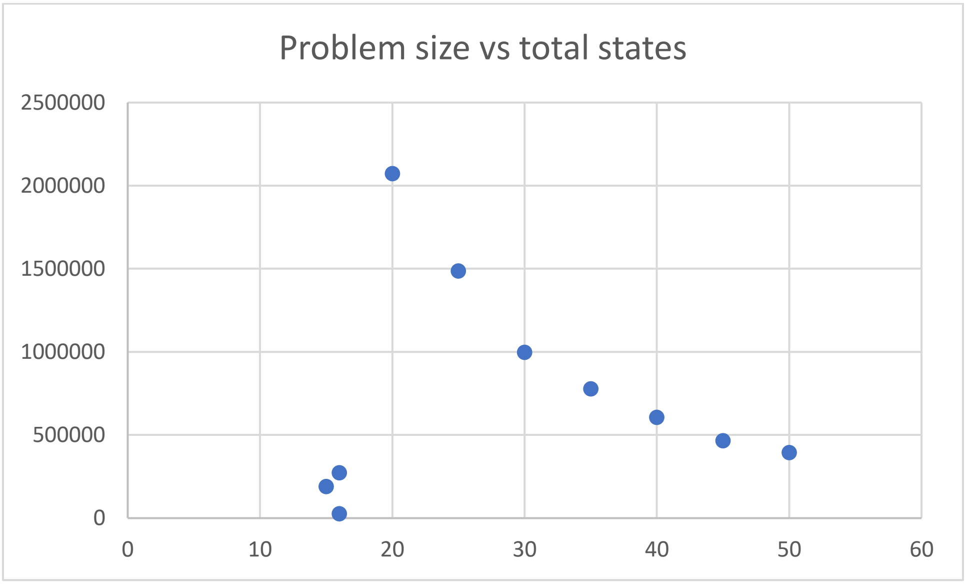
6. [30] Include a table containing the following columns.

# 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	3.45	*2430	100	5	190254	34696
16	902	0.5	*3247	106	12	26157	3024
16	34	5.6	*3152	106	1	273703	31877
20	45	57	*3952	180	23	2072972	210726
25	24	60	4971	958	10	1487215	197428
30	765	60	5455	1468	8	998256	226831
35	234	60	5217	2388	13	778645	101432
40	123	60	5468	1206	2	606285	79208
45	234	60	6386	13772	6	465354	71861

50	543	60	6212	4487	10	395055	73754

7. [10] Discuss the results in the table and why you think the numbers are what they are, including how time complexity and pruned states vary with problem size.

Number of states and running time made a significant jump between 16 and 20, while number of states generated and pruned actually declined with size after that. This is likely because the time limit is a bottleneck from size 20 on, and because there are more cities to perform operations on, fewer states are generated in that amount of time. Total number of states pruned was roughly proportional to total number of states generated. In these relationships, however, there are outliers, indicating that factors such as the initial bssf or the complexity of the non-metric paths between cities could affect the time, outcome of the states, and pruning. Number of solutions generated is completely uncorrelated with problem size, again probably depending on how easily paths are found between the cities. Max number of states on the priority queue also has a weak relationship with the problem size, which likely reflects the method of key determining as well.



code

```
public class BBSolver
        {
            private ProblemAndSolver parent;
            private PriorityQueue pq;
            private int solCount;
            private int maxQ;
            private int totalStates;
            private int prunedStates;
            private Stopwatch stopwatch;
            public BBSolver(ProblemAndSolver parent)
            {
                this.parent = parent;
                solCount = 0;
                maxQ = 0;
                totalStates = 0;
                prunedStates = 0;
                stopwatch = new Stopwatch();
              }

            public class stateNode : IComparable<stateNode>
            {
                protected int depth; //depth in the solution tree
                protected double key; //will be calcuated from cost and depth
                private ArrayList route; //partial route of cities identified by index in
Cities array list
                private ArrayList remaining;
                private CostMatrix cost_mat; //includes both matrix and total cost
                private double cost_bound; //total cost of redcued matrix

                public stateNode(int depth, ArrayList route, ArrayList remaining,
CostMatrix cost_mat)
                {
                    this.depth = depth;
                    this.route = route;
                    this.remaining = remaining;
                    this.cost_mat = cost_mat;
                    cost_bound = cost_mat.getCost();
                    key = cost_bound / depth;
                }

                public int CompareTo(stateNode n)
                {

                    if (this.key < n.key) return -1;
                    else if (this.key == n.key) return 0;
                    else return 1;
                }
                public double getKey()
                {
                    return key;
                }
                public ArrayList getRoute()
                {
                    ArrayList rou = new ArrayList();
                    rou.AddRange(route);
                    /* for(int i = 0; i < route.Count; i++)
                    {
                        rou.Add((int)route[i]);
                    }
                    */
                    return rou;
                }
                public ArrayList getRemaining()
                {
                    ArrayList rem = new ArrayList(remaining.Count);
                    rem.AddRange(remaining);
                    return rem;
```

```
    }
    public CostMatrix getMatrix()
    {
        return cost_mat;
    }
    public double getBound()
    {
        return cost_bound;
    }
    public int getDepth()
    {
        return depth;
    }
}


public class CostMatrix
{
    private double[,] matrix;
    private double tot_cost;
    public CostMatrix(double[,] matrix, double tot_cost)
    {
        this.matrix = matrix;
        this.tot_cost = tot_cost;
    }
    public double[,] getMatrix()
    {
        double[,] mat = new double[matrix.GetLength(0), matrix.GetLength(1)];
        for (int i = 0; i < matrix.GetLength(0); i++)
        {
            for (int j = 0; j < matrix.GetLength(0); j++)
            {
                mat[i, j] = matrix[i, j];
            }
        }
        return mat;
    }
    public double getCost()
    {
        return tot_cost;
    }
    public override string ToString()
    {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < matrix.GetLength(0); i++)
        {
            string row = "";
            for (int j = 0; j < matrix.GetLength(1); j++)
            {
                row += matrix[i, j].ToString() + " ";
            }
            sb.Append(row + "\n");
        }
        return sb.ToString();
    }

}
```

```
            //reduceMatrix: O(4n^2) = O(n^2)
            public CostMatrix reduceMatrix(CostMatrix cost_matrix, int from, int to)
            {
                double[,] cost = cost_matrix.getMatrix();
                double totCost = cost_matrix.getCost();
                int[] zerocols = new int[cost.GetLength(1)];
                //for each row, find min; subtract min from every cell; O(n*2n)
                for (int i = 0; i < cost.GetLength(0); i++)
                {
                    double min = double.PositiveInfinity;
                    //find min
                    for (int j = 0; j < cost.GetLength(1); j++)
                    {
                        if (cost[i, j] < min)
                        {
                            min = cost[i, j];
                        }
                    }
                    //if row wasn't a cancelled out row from previous reductions
                    if (min != double.PositiveInfinity)
                    {
                        //add row's cost to total cost
                        totCost += min;
                        //subtract min from each cell
                        for (int j = 0; j < cost.GetLength(1); j++)
                        {
                            if (cost[i, j] != double.PositiveInfinity)
                            {
                                cost[i, j] -= min;
                            }
                            //if there is a zero, mark the column
                            if (cost[i, j] == 0)
                            {
                                zerocols[j] = 1;
                            }
                        }
                    }
                }
                //for every column, if it does not have a zero, find and subtract min ;
O(n*2n)

                for (int j = 0; j < cost.GetLength(1); j++)
                {
                    if (zerocols[j] != 1)
                    {
                        double min2 = double.PositiveInfinity;
                        for (int k = 0; k < cost.GetLength(0); k++)
                        {
                            if (cost[k, j] < min2)
                            {
                                min2 = cost[k, j];
                            }
                        }
                        //if it wasn't a cancelled out column
                        if (min2 != double.PositiveInfinity)
                        {
                            totCost += min2;
                            for (int k = 0; k < cost.GetLength(0); k++)
                            {
                                if (cost[k, j] != double.PositiveInfinity)
                                {
                                    cost[k, j] -= min2;
                                }
                            }
                        }
                    }
                }
                if(from != -1)
                {
                    totCost += cost[from, to];
                    cost[from, to] = double.PositiveInfinity;
                    //set from row to infinity
                    for (int f = 0; f < cost.GetLength(1); f++)
                    {
                        cost[from, f] = double.PositiveInfinity;
                    }
                    //set to column to infinty
                    for (int t = 0; t < cost.GetLength(0); t++)
                    {
                        cost[t, to] = double.PositiveInfinity;
                    }
                }
                CostMatrix new_cost_matrix = new CostMatrix(cost, totCost);
                return new_cost_matrix;
            }

            /// <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[] runBB()
            {
                string[] results = new string[3];


                //calculate cost matrix; O(n^2)
                double[,] cost = new double[parent.Cities.Length, parent.Cities.Length];

                for (int i = 0; i < parent.Cities.Length; i++)
                {
                    for (int j = 0; j < parent.Cities.Length; j++)
                    {
                        if (i == j)
                        {
                            cost[i, j] = double.PositiveInfinity;
                        }
                        else
                        {
                            cost[i, j] = parent.Cities[i].costToGetTo(parent.Cities[j]);
                        }
                    }
                }
                CostMatrix init_matrix = new CostMatrix(cost, 0);

                parent.defaultSolveProblem(); //set bssf

                ArrayList remainingCities = new ArrayList(parent.Cities.Length);
                for(int i = 1; i < parent.Cities.Length; i++)
                {
                    remainingCities.Add(i);
                }


                CostMatrix start_matrix = reduceMatrix(init_matrix, -1, -1);

              //create heap priority queue
                var pq = new C5.IntervalHeap<stateNode>();
                pq.Add(new stateNode(1, start_city, remainingCities, start_matrix));
//O(log n)
                totalStates++;

                stopwatch.Start();
                //run branch and bound algorithm: while there are statenodes on the
priority queue, deletemin and expand the node
                //for each unvisited city; update bssf when a leaf is reached; check each
node against bssf before expanding.
                //worst case O(n^2*n!)
                while(pq.Count() > 0 && stopwatch.Elapsed.CompareTo(new TimeSpan(0, 0,
60)) <= 0)
                {
                    stateNode node = pq.DeleteMin(); //O(log n)
                    double upperlimit = parent.GetBSSF().costOfRoute();

                    if (node.getBound() < upperlimit)
                    {
                        //upperlimit = parent.GetBSSF().costOfRoute();
                        ArrayList remaining = node.getRemaining();
                        int depth = node.getRoute().Count;
                        if (remaining.Count == 0)
                        {
                            if (node.getBound() < upperlimit)
                            {
                              //hit a leaf: check if it's better than bssf and update
                                parent.updateBSSF(node.getRoute());
                                solCount++;
                            }
                        }
                        else
                        {
                            //expand nodes
                            for (int i = 0; i < remaining.Count; i++)
                            {
                                //add to route
                                ArrayList oldroute = node.getRoute();
                                ArrayList newroute = node.getRoute();
                                newroute.Add(remaining[i]);
                                int to = (int)remaining[i];
                                //remove city from remaining
                                ArrayList newRem = (ArrayList)remaining.Clone();
                                newRem.RemoveRange(i, 1);
```

```
                                        //grab last city in parent node's route
                                        int from = (int)oldroute[oldroute.Count - 1];

                                        //reduce cost matrix and create child node
                                        CostMatrix costmat = reduceMatrix(node.getMatrix(), from,
to); //O(n^2)
                                        stateNode newNode = new stateNode(depth + 1, newroute,
newRem, costmat);
                                        totalStates++;
                                        if (newNode.getBound() < upperlimit)
                                        {
                                            pq.Add(newNode);
                                            if (pq.Count > maxQ)
                                            {
                                                maxQ = pq.Count;
                                            }
                                            else prunedStates++;
                                        }
                                    }
                                }
                            }
                            else prunedStates++;
                        }
                        stopwatch.Stop();
                        TimeSpan el_time = stopwatch.Elapsed;

                        results[COST] = parent.costOfBssf().ToString();
                        results[TIME] = el_time.ToString();
                        results[COUNT] = solCount.ToString();
                        Console.WriteLine("max Q length: " + maxQ.ToString());
                        Console.WriteLine("total states created: " + totalStates.ToString());
                        Console.WriteLine("states pruned: " + prunedStates.ToString());


                        return results;
                    }
                }
```