TSP Branch & Bound

Kevin DeVocht Sec 2

Code:

```
public void solveProblem()
            {
                Route = new ArrayList();
                State baseState = new State(Cities);
                baseState.reduceCosts();
                queue.Push(baseState, baseState.lowerbound);
                DateTime startTime = DateTime.UtcNow;
                DateTime endTime = DateTime.UtcNow.AddSeconds(30);
                while(!queue.isEmpty()){
                    if (endTime > DateTime.UtcNow)
                    {
                        if (bssf == null && !includeQueue.isEmpty())
                        {
                            findBestZero(includeQueue.Pop());
                        }
                        else
                        {
                            if (queue.Count > maxStored)
                            {
                                maxStored = queue.Count;
                            }
                            findBestZero(queue.Pop());
                        }

                    }
                    else
                    {
                        Program.MainForm.tbElapsedTime.Text = " 30";
                        Program.MainForm.Invalidate();
                        break;
                    }
                }
                System.Console.WriteLine("Static Count: " + State.numCreated);
                System.Console.WriteLine("Number Pruned: " + numPruned);
                System.Console.WriteLine("Number of Solutions: " + numSolutions);
                System.Console.WriteLine("Maxed Stored in Queue: " + maxStored);
                State.numCreated = 0;
                Program.MainForm.tbElapsedTime.Text = " "+ (DateTime.UtcNow -
startTime);

                Program.MainForm.tspSolution.Text = " " + numSolutions;
                Program.MainForm.Invalidate();
```

```
        }



        public void findBestZero(State current)
        {
            //this should be the pruning portion and it should only ever prune
if there is already a solution
            if(bssf != null && current.lowerbound > costOfBssf())
            {
                numPruned++;
                return;
            }

            double bestDifference = Double.NegativeInfinity;
            State bestInclude = null;
            State bestExclude = null;
            int bestRow = -1;
            int bestCol = -1;
            for (int row = 0; row < current.costMatrix.GetLength(0); row++)
            {
                for (int col = 0; col < current.costMatrix.GetLength(1);
col++)
                {
                    if (current.costMatrix[row, col] == 0)
                    {
                        State include = new State(current);
                        State exclude = new State(current);
                        include = createIncludeState(include, row,
col);
                        exclude = createExcludeState(exclude, row,
col);
                        if(include.isSolution)
                        {
                            createSolution(include);
                            return;
                        }
                        if (exclude.isSolution)
                        {
                            createSolution(exclude);
                            return;
                        }
                        double diff;
                        if(exclude.lowerbound !=
Double.PositiveInfinity && include.lowerbound != Double.PositiveInfinity)
                        {
                            diff = exclude.lowerbound -
include.lowerbound;
                        }
                        else
                        {
                            diff = Double.PositiveInfinity;
                        }

                        if (diff > bestDifference)
                        {
                            bestDifference = diff;
                            bestInclude = include;
                            bestExclude = exclude;
                            bestRow = row;
                            bestCol = col;
                        }
                    }
                }
            }
            if(bssf == null)
            {
                includeQueue.Push(bestInclude, bestInclude.lowerbound);
            }
            if (bssf != null && bestInclude.lowerbound > costOfBssf())
            {
                numPruned++;
            }
            else
            {
                queue.Push(bestInclude, bestInclude.lowerbound);
            }

            if (bssf != null && bestExclude.lowerbound > costOfBssf())
            {
                numPruned++;
            }
            else
            {
                queue.Push(bestExclude, bestExclude.lowerbound);
            }



        }

        public State createIncludeState(State include, int rowID, int ColID)
        {
            //Set Column to Infinity
            for (int col = 0; col < include.costMatrix.GetLength(1); col++)
            {
                include.costMatrix[rowID, col] = Double.PositiveInfinity;
            }
            //Set Row to Infinity
            for (int row = 0; row < include.costMatrix.GetLength(1); row++)
            {
                include.costMatrix[row, ColID] = Double.PositiveInfinity;
            }
            include.checkForCycles(rowID, ColID);
            include.reduceCosts();


            return include;
        }


        public State createExcludeState(State exclude, int rowID, int ColID)
        {
            exclude.costMatrix[rowID, ColID] = Double.PositiveInfinity;
            exclude.reduceCosts();
            return exclude;
        }

        public void createSolution(State solutionState)
        {
            if (bssf == null || solutionState.lowerbound < costOfBssf())
            {
                if(bssf != null)
                {
                    numSolutions++;
                }
                List<int> keys = new List<int>();
                Route.Clear();
                foreach (int key in solutionState.cycleKeyMap.Keys)
                {
                    keys.Add(key);

                }
                if (keys.Count > 1)
                {
                    throw new Exception("There are more than keys in the
cycleKeyMap");
                }
                int startCity = keys[0];
                Route.Add(Cities[startCity]);
                int nextCity = startCity;
```

```
                    for (int i = 0; i < solutionState.routeMap.Count; i++)
                    {
                        nextCity = solutionState.routeMap[nextCity];
                        Route.Add(Cities[nextCity]);
                    }

                    // call this the best solution so far.  bssf is the route
that will be drawn by the Draw method.
                    bssf = new TSPSolution(Route);
                    // update the cost of the tour.
                    Program.MainForm.tbCostOfTour.Text = " " +
bssf.costOfRoute();
                    // do a refresh.
                    Program.MainForm.Invalidate();
                }
            }


using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace TSP
{
    class State
    {

        public double[,] costMatrix;
        public double lowerbound = 0;
        public double Infinity = Double.PositiveInfinity;
        public bool isSolution;
        public static int numCreated = 0;

        //the key is the city that you are leaving and the value is the city that
you arrive at
        public Dictionary<int, int> routeMap = new Dictionary<int,int>();
        public Dictionary<int, int> cycleKeyMap = new Dictionary<int, int>();
        public Dictionary<int, int> cycleValueMap = new Dictionary<int, int>();


        public State(City[] cities)
        {
            numCreated++;
            int size = (int)cities.Count();
            costMatrix = new double[size, size];
```

```
                    for (int row = 0; row < cities.Count(); row++)
                    {
                        for (int col = 0; col < cities.Count(); col++)
                        {
                            if (row == col)
                            {
                                costMatrix[row, col] = Double.PositiveInfinity;
                            }
                            else
                            {
                                costMatrix[row, col] = 
cities[row].costToGetTo(cities[col]);
                            }

                        }
                    }
                }

                public State(State other)
                {
                    numCreated++;
                    costMatrix = (double[,])other.costMatrix.Clone();
                    lowerbound = other.lowerbound;
                    isSolution = other.isSolution;
                    routeMap = new Dictionary<int, int>(other.routeMap);
                    cycleKeyMap = new Dictionary<int, int>(other.cycleKeyMap);
                    cycleValueMap = new Dictionary<int, int>(other.cycleValueMap);
                }

                public void reduceCosts()
                {
                    //Check all rows
                    int lowestX = 0;
                    int lowestY = 0;
                    double lowestValue = 0;
                    for (int row = 0; row < costMatrix.GetLength(0); row++)
                    {
                        //This is here because if this row is in the solution then 
skip it
                        if (!routeMap.ContainsKey(row))
                        {
                            lowestValue = costMatrix[row, 0];
                            lowestX = row;
                            lowestY = 0;
                            for (int col = 0; col < costMatrix.GetLength(1); 
col++)
                            {
                                //This is here because if this column is in the
```

```
solution then skip it
                        if (!routeMap.ContainsValue(col))
                        {
                            if (costMatrix[row, col] < lowestValue)
                            {
                                lowestValue = costMatrix[row, 
col];
                                lowestY = col;
                            }
                        }
                    }
                    //TODO:  This portion of the code might still need 
some help when things start to have lots of Infinity
                    if (costMatrix[lowestX, lowestY] != Infinity)
                    {
                        lowerbound += costMatrix[lowestX, lowestY];
                    }
                    for (int col = 0; col < costMatrix.GetLength(1); 
col++)
                    {
                        if (costMatrix[lowestX, lowestY] != Infinity)
                        {
                            costMatrix[row, col] = costMatrix[row, 
col] - lowestValue;
                        }
                    }
                }
            }

            //Check all columns
            for (int col = 0; col < costMatrix.GetLength(1); col++)
            {
                //This is here because if this column is in the solution 
then skip it
                if (!routeMap.ContainsValue(col))
                {
                    lowestValue = costMatrix[0, col];
                    lowestX = 0;
                    lowestY = col;
                    for (int row = 0; row < costMatrix.GetLength(0); 
row++)
                    {
                        //This is here because if this row is in the 
solution then skip it
                        if (!routeMap.ContainsKey(row))
                        {
                            if (costMatrix[row, col] < lowestValue)
                            {
                                lowestValue = costMatrix[row,
col];
                                lowestX = row;
                            }
                        }
                    }
                    if (costMatrix[lowestX, lowestY] != Infinity)
                    {
                        lowerbound += costMatrix[lowestX, lowestY];
                    }
                    for (int row = 0; row < costMatrix.GetLength(1);
row++)
                    {
                        if (costMatrix[lowestX, lowestY] != Infinity)
                        {
                            costMatrix[row, col] = costMatrix[row,
col] - lowestValue;
                        }

                    }
                }
            }
        }

        public void printOutCostMatrix()
        {
            for (int row = 0; row < costMatrix.GetLength(0); row++)
            {
                for (int col = 0; col < costMatrix.GetLength(1); col++)
                {
                    if (costMatrix[row, col] == Double.PositiveInfinity)
                    {
                        System.Console.Write("INF " + " ");
                    }
                    else
                    {
                        if (costMatrix[row, col] < 10)
                        {
                            System.Console.Write(costMatrix[row,
col] + "    ");
                        }
                        else if (costMatrix[row, col] < 100)
                        {
                            System.Console.Write(costMatrix[row,
col] + "   ");
                        }
                        else if (costMatrix[row, col] < 1000)
                        {
```

```
                                        System.Console.Write(costMatrix[row, 
col] + "  ");
                                    }
                                    else
                                    {
                                        System.Console.Write(costMatrix[row, 
col] + " ");
                                    }
                                }
                            }
                            System.Console.Write("\n");
                        }
                }

                public void checkForCycles(int rowID, int ColID)
                {
                        routeMap.Add(rowID, ColID);
                        //cycleMap.Add(rowID, ColID);//Probably move this so that we don't 
have to just remove it again if there is a merge

                        costMatrix[ColID, rowID] = Double.PositiveInfinity;
                        //crosscheck
                        bool merging = true;
                        int cycleKey = rowID;
                        int cycleValue = ColID;

                        while (merging)
                        {
                            if (cycleKeyMap.ContainsValue(cycleKey))
                            {
                                //set the new cycleKey using the old one
                                cycleKey = cycleValueMap[cycleKey];
                                //Remove the old entry from the cycleValueMap
                                cycleValueMap.Remove(cycleKeyMap[cycleKey]);
                                //Remove the old entry from the cycleKeyMap
                                cycleKeyMap.Remove(cycleKey);

                                //At this point we have merged to nodes together 
along a common path.
                                //And we have a new node to check and see if we can 
merge even further
                            }
                            else if (cycleKeyMap.ContainsKey(cycleValue))
                            {
                                cycleValueMap.Remove(cycleKeyMap[cycleValue]);
                                int keyKey = cycleValue;
                                cycleValue = cycleKeyMap[cycleValue];
                                cycleKeyMap.Remove(keyKey);
                        }
                        else
                        {
                                merging = false;
                        }
                }
                //Now add in the newly merged key value pair, and it's inverse into
the cycleKeyMap and cycleValueMap respectivly
                //You also need invalidate the inverse in the cost matrix
                cycleKeyMap.Add(cycleKey, cycleValue);
                cycleValueMap.Add(cycleValue, cycleKey);
                costMatrix[cycleValue, cycleKey] = Infinity;
                if(routeMap.Count == costMatrix.GetLength(0)-1)
                {
                        isSolution = true;
                }
            }
        }
}

    class PriorityQueue
    {
        private int count;
        SortedDictionary<double, Queue> queue;

        public PriorityQueue()
        {
            this.count = 0;
            this.queue = new SortedDictionary<double, Queue>();
        }

        public int Count
        {
            get { return count; }
        }

        public bool isEmpty()
        {
            return count == 0;
        }

        public State Pop()
        {
            var key = queue.First();
            Queue q = key.Value;
            State state = (State)q.Dequeue();
            if (q.Count == 0)
            {
                queue.Remove(key.Key);
            }
            count--;
            return state;
        }

        public void Push(State state, double lowerbound)
        {
                    if (!queue.ContainsKey(lowerbound))
            {
                           queue.Add(lowerbound, new Queue());
            }
                    queue[lowerbound].Enqueue(state);
            count++;
        }
    }
```

State:
For the State class I used a two dimensional array of doubles for the cost matrix. This help in cycling through the cost matrix when reducing the cost. I also used a dictionary to store my route, and to check for cycles. I had to use another dictionary to help me look up the keys of the cycle checking dictionary.

Priority Queue:
The Priority Queue uses a Sorted Dictionary to maintain the priority. The lower bound of the state is used as the key and so, those states with the lowest lower bounds are at the beginning of the Sorted Dictionary. I then have a pop function that pulls off the first value in the Sorted dictionary and returns it. I also use an int to store how many items are still in the queue.

Initial BSSF:
As long as bssf is null, I use a special priority queue that I only push the includes onto. This is an attempt to dig deep by trying to get a solution fast. The reason why this is fast is because it adds the best path every time (greedy), so generally this will lead to a decent solution. There were a few times though where there was no solution by just including the best zero every time.

Results table:

#Cities	Seed	Running Time (sec.)	Cost of best tour found	Max # of Stored states	# of BSSF updates	Total # of States Created	Total # of States Pruned
15	20	1.94	2430	4373	1	115571	4593
16	902	0.41	3223	720	1	24145	962
17	150	0.35	3853	705	1	18889	734
18	111	3.58	3415	5141	1	161643	5148
19	222	13.66	3968	17309	1	552319	17751
20	333	30	4022	22245	0	808293	1698
30	555	30	4746	6072	0	362429	0
40	777	30	5834	2278	0	193447	0
50	654	30	7221	1057	0	121905	0

Results Analysis:
It is very interesting to see how just going from 19 to 20 states results in a huge increase of time and number of states created. This could just be random based off of the seed used, but it is pretty consistent that anything over 19 cities took more than 30 seconds. It is also interesting to know that after about 30 I no longer pruned anything. I am not sure why this would be the case, other than, the solution I initially found was not very good and every other branch that I checked after that was actually better.

Extra Credit: see Initial BSSF section