Skyler Smith

CS 312 - 002

Nov 19, 2016

TSP Report

Part 1 - Code

```
/// <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];

   //Find a smart starting bssf
   GreedySolver greedy = new GreedySolver(Cities, bssf, results);
   bssf = greedy.Solve();

   BranchAndBoundSolver brancher = new BranchAndBoundSolver(Cities, bssf, results, time_limit);
   bssf = brancher.Solve();

   return results;
}

/// <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];

   GreedySolver greedy = new GreedySolver(Cities, bssf, results);
   bssf = greedy.Solve();

   return results;
}

namespace TSP
{
   public class GreedySolver
   {
      City[] cities;
      ProblemAndSolver.TSPSolution bssf;
      string[] results;

      public GreedySolver(City[] cities, ProblemAndSolver.TSPSolution bssf, string[] results)
      {
         this.cities = cities;
         this.bssf = bssf;
         this.results = results;
      }


      public ProblemAndSolver.TSPSolution Solve()
      {
         ArrayList route = new ArrayList();
         int numUpdates = -1;
         var timer = new Stopwatch();

         timer.Start();
         for (int startCity = 0; startCity < cities.Length; startCity++)
         {
            route.Clear();
            int currentCity = startCity;
            do
            {
               route.Add(cities[currentCity]);
               if (route.Count == cities.Length)
               {
                  //go back to first city
                  double pathCost = cities[currentCity].costToGetTo(cities[startCity]);
                  if (double.IsPositiveInfinity(pathCost))
                  {
                     break;
                  }
                  ProblemAndSolver.TSPSolution solution = new ProblemAndSolver.TSPSolution(route);
                  if (solution.costOfRoute() < costOfBssf())
                  {
                     bssf = solution;
                     Debug.Assert(costOfBssf() == solution.costOfRoute());
                     numUpdates++;
                     break;
                  }
               }
               double shortestRoute = double.PositiveInfinity;
               int nearestCity = -1;
               for (int i = 0; i < cities.Length; i++)
               {
                  double pathCost = cities[currentCity].costToGetTo(cities[i]);
                  if (pathCost < shortestRoute && !route.Contains(cities[i]))
                  {
                     shortestRoute = pathCost;
                     nearestCity = i;
                  }
               }
               if (nearestCity == -1)
               {
                  //unable to find a path out of the current city to a new city
                  break;
               }

               currentCity = nearestCity;

            } while (true);
         }

         timer.Stop();

         results[ProblemAndSolver.COST] = costOfBssf().ToString();                           // load results array
         results[ProblemAndSolver.TIME] = timer.Elapsed.ToString();
         results[ProblemAndSolver.COUNT] = numUpdates.ToString();
         return bssf;
      }


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


namespace TSP
{
   public class BranchAndBoundSolver
   {
      public static double averageDistance = 0;

      City[] cities;
      ProblemAndSolver.TSPSolution bssf;
      string[] results;
      PriorityQueue queue;
      int prunedNodes;
      int nodesCreated;
      int timeLimit;
      int numUpdates;

      public BranchAndBoundSolver(City[] cities, ProblemAndSolver.TSPSolution bssf, string[] results, int timeLimit)
      {
         this.cities = cities;
         this.bssf = bssf;
         this.results = results;
         this.timeLimit = timeLimit;
      }

      public ProblemAndSolver.TSPSolution Solve()
      {
         //Tests show that heap is better for more cities, array is better for less
         queue = new HeapPriorityQueue();
         nodesCreated = 0;
         prunedNodes = 0;
         averageDistance = 0;
         numUpdates = 0;
         var timer = new Stopwatch();
         timer.Start();
         //Build the mother node state
         //Build a starting matrix. time O(n^2) space O(n^2)
         int count = 0;
         double[,] motherMatrix = new double[cities.Length,cities.Length];
         for (int row = 0; row < cities.Length; row++)
         {
            for (int col = 0; col < cities.Length; col++)
            {
               double cost = cities[row].costToGetTo(cities[col]);
               motherMatrix[row, col] = row == col ? double.PositiveInfinity : cost;
               if (!double.IsPositiveInfinity(cost))
               {
                  averageDistance += cost;
                  count++;
               }
            }
         }
         averageDistance = averageDistance / count;
         //Reduce the matrix & get the bound. time O(n^2)
         double bound = ReduceMatrix(motherMatrix);
         ArrayList route = new ArrayList();
         route.Add(cities[0]);
         //Put them in a new state
         State motherState = new State(motherMatrix, route, bound, 0);
         //Expand the mother node state. time O(n^3) space O(n^3)
         ExpandState(motherState);
         //While the priority queue is not empty. Impossible worst case: O(b^n * log(n!))
         while (!queue.IsEmpty() && timer.ElapsedMilliseconds < timeLimit)
```

```
            {
                //pop off the top state and expand it. time O(n^3) space O(n^3)
                //DeleteMin could potentially be as bad as time O(log(n!))
                ExpandState(queue.DeleteMin());
            }
            timer.Stop();

            //When the timer goes off, update results and return bssf
            results[ProblemAndSolver.COST] = costOfBssf().ToString();
            results[ProblemAndSolver.TIME] = timer.Elapsed.ToString();
            results[ProblemAndSolver.COUNT] = numUpdates.ToString();
            Console.WriteLine("Max Stored States: " + queue.LargestSize());
            Console.WriteLine("States Created: " + nodesCreated);
            Console.WriteLine("States Pruned: " + prunedNodes);
            return bssf;
        }

        /// <summary>
        /// Makes states for all the nodes in the graph that aren't in state's route and stores them in a priority queue
        /// Time O(n^3) Space O(n^3)
        /// </summary>
        /// <param name="state">parentState. The state to expand</param>
        public void ExpandState(State parentState)
        {
            //if the state's bound >= bssf: increment prunedNodes & return
            if (parentState.bound > costOfBssf())
            {
                prunedNodes++;
                return;
            }
            //If all the nodes in the graph are in the route (compare sizes): wrap up the route by traveling to the first node and
checking the result against bssf. Update bssf if needed.
            if (parentState.route.Count >= cities.Count())
            {
                //Time O(n)
                double costToReturn = TravelInMatrix(parentState.matrix, parentState.lastCity, 0);
                if (!double.IsPositiveInfinity(costToReturn))
                {
                    parentState.bound += costToReturn;
                    parentState.route.Add(cities[0]);
                    if (parentState.bound < costOfBssf())
                    {
                        bssf = new ProblemAndSolver.TSPSolution(parentState.route);
                        numUpdates++;
                    }
                }
                return;
            }
            //Else:
            //For each node in the graph that isn't in the route: time O(n^3) space O(n^3)
            for (int i = 0; i < cities.Count(); i++)
            {
                City city = cities[i];
                if (double.IsPositiveInfinity(cities[parentState.lastCity].costToGetTo(city)) || parentState.route.Contains(i))
                {
                    continue;
                }
                //Copy the parent node state
                //Time O(n^2) size O(n^2)
                State childState = parentState.Copy();
                nodesCreated++;
                childState.route.Add(cities[i]);
                childState.lastCity = i;
                //Travel in the matrix to the new node and set the appropriate cells to infinity (TravelInMatrix). time O(n)
                double travelCost = TravelInMatrix(childState.matrix, parentState.lastCity, i);
                if (double.IsPositiveInfinity(travelCost))
                {
                    continue;
                }
                childState.bound += travelCost;
                //Reduce the matrix and update the bound. time O(n^2)
                childState.bound += ReduceMatrix(childState.matrix);
                //If the bound is lower than bssf's:
                if (childState.bound < costOfBssf())
                {
                    //add the state to the priority queue. time O(logn)
                    queue.Insert(childState);
                }
                else
                {
                    prunedNodes++;
                }
            }

        }

        /// <summary>
        /// Reduces the given matrix. Time O(n^2) Space O(1)
        /// </summary>
        /// <returns>The additional cost associated with reducing the matrix</returns>
        /// <param name="matrix">Matrix to reduce.</param>
        public double ReduceMatrix(double[,] matrix)
        {
            double cost = 0;
            //reduce each row. time O(n^2)
            for (int row = 0; row < cities.Count(); row++)
            {
                //find the smallest cost in the row
                double smallestCost = double.PositiveInfinity;
                for (int col = 0; col < cities.Count(); col++)
                {
                    if (matrix[row, col] < smallestCost)
                    {
                        smallestCost = matrix[row, col];
                    }
                }
                if (smallestCost <= 0.01 || double.IsPositiveInfinity(smallestCost))
                {
```

```
                continue;
            }
            cost += smallestCost;
            //reduce the row
            for (int col = 0; col < cities.Count(); col++)
            {
                matrix[row, col] -= smallestCost;
            }
        }
        //reduce each column. time O(n^2)
        for (int col = 0; col < cities.Count(); col++)
        {
            //find the smallest cost in the column
            double smallestCost = double.PositiveInfinity;
            for (int row = 0; row < cities.Count(); row++)
            {
                if (matrix[row, col] < smallestCost)
                {
                    smallestCost = matrix[row, col];
                }
            }
            if (smallestCost <= 0.01 || double.IsPositiveInfinity(smallestCost))
            {
                continue;
            }
            cost += smallestCost;
            //reduce the column
            for (int row = 0; row < cities.Count(); row++)
            {
                matrix[row, col] -= smallestCost;
            }
        }

        return cost;
    }

    /// <summary>
    /// Sets the fromNode row, toNode column, and cell (toNode, fromNode) values to infinity if the travel is possible
    /// Time O(n) Space O(1)
    /// </summary>
    /// <param name="matrix">Matrix to mutate after travelling.</param>
    /// <param name="fromNode">From node.</param>
    /// <param name="toNode">To node.</param>
    /// <returns>The cost required to travel from node to node.</returns>
    public double TravelInMatrix(double[,] matrix, int fromNode, int toNode)
    {
        double cost = matrix[fromNode, toNode];
        if (double.IsPositiveInfinity(cost))
        {
            return double.PositiveInfinity;
        }
        matrix[toNode, fromNode] = double.PositiveInfinity;
        for (int i = 0; i < cities.Count(); i++)
        {
            matrix[fromNode, i] = double.PositiveInfinity;
            matrix[i, toNode] = double.PositiveInfinity;
        }

        return cost;
    }

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

public class State
{
    //(the states have a route property, so they know where they are in the tree without needing pointers)
    public double[,] matrix;
    public ArrayList route;
    public double bound;
    public int lastCity;

    //Only needed in heap implementation of PQ
    public int queueIndex;

    public State(double[,] matrix, ArrayList route, double bound, int lastCity)
    {
        this.matrix = matrix;
        this.route = route;
        this.bound = bound;
        this.lastCity = lastCity;
    }

    public State Copy()
    {
        return new State((double[,])matrix.Clone(), (ArrayList)route.Clone(), bound, lastCity);
    }

    public bool PrioritizeOver(State state)
    {
        //Weight the difference between route lengths by the average distance of an edge.
        int sizeDiff = state.route.Count - route.Count;
        double padding = sizeDiff * BranchAndBoundSolver.averageDistance;
        return padding + bound <= state.bound;
    }
}
```

```
public interface PriorityQueue
{
   void Insert(State node);
   State DeleteMin();
   //void DecreaseKey(State node);
   bool IsEmpty();
   int LargestSize();
}


public class HeapPriorityQueue : PriorityQueue
{
   List<State> queue;
   int maxNumNodes;

   public HeapPriorityQueue()
   {
      queue = new List<State>();
      maxNumNodes = 0;
   }

   /// <summary>
   /// Finds and returns the node with the smallest distance value. time: O(log|V|)
   /// </summary>
   /// <returns>The node with the smallest distance value </returns>
   public State DeleteMin()
   {
      // return the root node and settle the heap
      State root = queue.First();
      root.queueIndex = -1;
      State lastNode = queue.Last();
      lastNode.queueIndex = 0;
      queue[0] = lastNode;
      queue.RemoveAt(queue.Count - 1);
      if (queue.Count > 0)
      {
         //time: O(log|V|)
         SiftDown(lastNode);
      }
      return root;
   }

   /// <summary>
   /// Insert the specified node. time: O(log|V|)
   /// </summary>
   /// <param name="node">Node.</param>
   public void Insert(State node)
   {
      //Add the node to the bottom of the heap and bubble up
      node.queueIndex = queue.Count;
      queue.Add(node);
      BubbleUp(node);
      if (queue.Count > maxNumNodes)
      {
         maxNumNodes = queue.Count;
      }
   }

   public bool IsEmpty()
   {
      return queue.Count == 0;
   }

   /// <summary>
   /// Bubbles up the given node. O(log|V|)
   /// </summary>
   /// <param name="node">Node to bubble up.</param>
   private void BubbleUp(State node)
   {
      if (node.queueIndex == 0)
      {
         return;
      }
      State parent = queue[(int)Math.Ceiling(node.queueIndex / (decimal)2.0) - 1];
      int position = node.queueIndex;
      //Switch the node with its parent until its parent is smaller that it.
      //In the worst case, the node goes from the bottom to the top.
      //This is a binary heap, so there can be at most log|V| switches. time: O(log|V|)
      while (position != 0 && node.PrioritizeOver(parent))
      {
         queue[position] = parent;

         int childPosition = position;
         position = parent.queueIndex;
         node.queueIndex = position;
         parent.queueIndex = childPosition;

         if (position == 0)
         {
            break;
         }

         parent = queue[(int)Math.Ceiling(position / (decimal)2.0) - 1];
      }
      queue[position] = node;
   }

   /// <summary>
   /// Sifts down the given node. time: O(log|V|)
   /// </summary>
   /// <param name="node">Node to sift down.</param>
   private void SiftDown(State node)
   {
      State minChild = MinChild(node);
      int position = node.queueIndex;
      //Switch the node with its smallest child until its smallest child is bigger that it.
      //In the worst case, the node goes from the top to the bottom.
```

```
            //This is a binary heap, so there can be at most log|V| switches. time: O(log|V|)
            while (minChild != null && minChild.PrioritizeOver(node))
            {
                queue[position] = minChild;

                int parentPosition = position;
                position = minChild.queueIndex;
                node.queueIndex = position;
                minChild.queueIndex = parentPosition;

                minChild = MinChild(node);
            }
            queue[position] = node;
        }

        /// <summary>
        /// Returns the smaller of the two children of the given node, if it has any children. time: O(1)
        /// </summary>
        /// <returns>The parent node.</returns>
        /// <param name="node">Node.</param>
        private State MinChild(State node)
        {
            if (1 + node.queueIndex * 2 >= queue.Count)
            {
                //No children
                return null;
            }

            State child1 = queue[node.queueIndex * 2 + 1];
            if (node.queueIndex * 2 + 2 >= queue.Count)
            {
                return child1;
            }

            State child2 = queue[node.queueIndex * 2 + 2];

            if (child2.PrioritizeOver(child1))
            {
                return child2;
            }
            else
            {
                return child1;
            }
        }

        public int LargestSize()
        {
            return maxNumNodes;
        }
    }

}
```

Part 2 - Time and Space Complexity

1. Build the starting matrix. Time $O(n^2)$ Space $O(n^2)$
2. Reduce the starting matrix. Time $O(n^2)$
    1. Reduce each row. Time O(n^2)
    2. Reduce each column. Time O(n^2)
3. Expand the mother node state. Time $O(n^3)$ Space $O(n^3)$
4. While the priority queue is not empty (Worst case Time $O(n^3b^n+\log(n!))$, with $b > 1$ Space $O(n^3)$):
    1. Pop off the top state. Time $O(\log(n!))$
    2. Expand the state. Time (n^3) Space $O(n^3)$
        1. For each node not in the route (Time $O(n^3)$ Space $O(n^3)$):
            1. Copy the parent node state. Time $O(n^2)$ Space $O(n^2)$

    2. Travel in the matrix to the new node. Time O(n)
        1. Set the appropriate 2n cells to infinity. Time O(n)
    3. Reduce the matrix. Time O(n^2)
        1. Reduce each row. Time O(n^2)
        2. Reduce each column. Time O(n^2)
    4. Add the state to the priority queue. Time O(logn)

Altogether Time: $O(n^2 + n^2 + n^3 + n^3b^n+\log(n!))$ = Time $O(n^3b^n+\log(n!))$

Altogether Space: $O(n^2 + n^3 + n^3) = O(n^3)$

Part 3 - Data Structures - States

To represent the state, I created a class with a double[,] matrix, a route of cities, a bound, and a lastCity (used for making new states). State objects can make a deep copy of themselves and determine whether or not they should be prioritized over other states in a priority queue. They also know their own index in the queue.

Part 4 - Data Structures - Priority Queue

I used a binary heap priority queue. It works by keeping the highest priority item at the front of the queue by rearranging items whenever something is added or removed. To determine which of two states should be prioritized higher, I use a function of the bound and size of the route in the state. I use the average edge length in this calculation, so that a state's priority is essentially route size * average length + bound.

Part 5 - Initial BSSF

I use a greedy algorithm for my initial BSSF. I simply find the best greedy route from each possible start city and use its total length as my first BSSF.

Part 6 - Results

# Cities	Seed	Running Time (sec)	Cost of Best tour found (* = optimal)	Max states stored	BSSF Updates	States Created	States Pruned
15	20	0.5955795	2430*	68	1	119399	42507
16	902	0.1697498	3223*	85	2	21938	9963
17	11	0.1069993	3326*	113	6	16771	5601
18	107	13.4103556	3815*	131	1	1899010	785297
19	13	6.1263906	3539*	114	2	792425	331914
20	17	60.0000935	3929	164	9	7539092	3040193
25	53	60.0000636	4410	292	1	10358926	18596252
30	71	60.0001515	4807	465	0	7591523	1518501
40	1012	60.0004372	5037	735	2	5597803	841455
50	64	60.0082690	6926	1429	0	5711575	395086

Part 7 - Discussion

Running Time. On the smaller sets of cities (<20), the running time was fairly unpredictable, probably because the chance of getting lucky and quickly finding the best route, or a very good route, isn't too bad. In many few-city cases, the greedy algorithm gets a result very similar to the best one. So, sometimes, the set is such that B&B can prune a lot of possibilities very quickly, while other times it's not so lucky; thus the patternless running times. On larger numbers of cities (>=20), however, the chance of finding one of the best routes quickly isn't very good. With 20 cities, there are $19! \approx 1.22\text{x}10^{17}$ route possibilities, and the greedy algorithm is less likely to get a good route. I would expect running time to steadily increase with more cities.

Cost of Best Route. With more cities, we can only expect that the cost of the best route will increase. In general, the results followed that pattern.

Max States Stored. The number of states stored follows well with the cost of the best route (in fact, a logarithmic function can accurately predict the max stored states given the best cost of the route found). This is probably because with a longer bssf, states are pruned less often. We can see that as the length of the longest route increases, the ratio of states pruned to states created decreases.

BSSF Updates. With fewer cities, the number of BSSF updates is erratic, and with more cities, the number of updates is predictably small. I think that the large numbers of cities produce few BSSF updates because B&B doesn't have much time to find solutions better than Greedy. In some cases, B&B might not reach a complete route in 60 seconds. Because of this, it is difficult to say anything conclusive about the pattern of BSSF updates. I suspect, however, that given enough time, graphs with more cities would yield more BSSF updates.

States Created & States Pruned. The first, obvious trend is that the states created and pruned both increase toward 25 cities, then start to decrease with more cities. I think they decrease with large numbers of cities because of the massive overhead of having lots of cities: each state takes longer to make, add to the queue, and pop off the queue. Because of this, the program can't create as many states in 60 seconds with lots of cities as it could with fewer cities. Another trend is that with more cities, the number of states pruned per state created decreases. I think this is because, as mentioned in my discussion on Max States, a bigger BSSF makes pruning states harder.