Anthony Constantino
21 November 2015
Project 5 Report

Traveling Salesman

Time and Space complexity

The time complexity of the insertions and deletions of the Priority Queue are of time complexity O(n) because of its tree like structure so only a specific branch is traversed and modified with each call. The Space complexity is size O(n), containing one instance of each state, but is most likely smaller than n because not each state will be created as some of the branches will be pruned as the program is running, avoiding that the priority queue gets too large.

The State Data Structure

For the states I created an anonymous private sub class called State. Within this state class there were several data members to keep track of various parts of the state.

```
private class State
{
    //Definition of state
    public ArrayList pathSoFar;
    public int currentIndex;
    public ArrayList childrenIndexes;
    public double[,] costArray;
    public double currentCost;
    public City[] cities;
    public double lowerBound;
    public int depthIntoSolution;
    public bool solutionFound;
```

The list pathsofar is used to know what the current list of cities looks like, using the array to create this list and the currentCost of the path. The cost array is used to keep track of the costs from one city to the next, and in the reducedCostMatrix function inside this class is used to

update the array with the reduced cost matrix which is then used with each state in the program (ProblemAndSolver.cs lines 563 to 613). The program also implements a generate children states function that uses the list of children to create the children states with their reduced cost matrices and then stores them in the current state object. This way there is a tree like structure where the states can be followed from the parent down through the children states.

```
public ArrayList generateChildrenStates()
{
    ArrayList children = new ArrayList();
    //create child state for all possibilities
    foreach (int i in childrenIndexes)
    {
        ArrayList newChildren = (ArrayList)childrenIndexes.Clone();
        newChildren.Remove(i);

        // update the path based on this child
        ArrayList newPath = (ArrayList)pathSoFar.Clone();
        newPath.Add(currentIndex);
        double cost = cities[currentIndex].costToGetTo(cities[i]);

        //reduced matrix for new state
        double[,] newCost = (double[,])costArray.Clone();
        for (int j = 0; j <= newCost.GetUpperBound(0); j++)
            newCost[j, currentIndex] = double.PositiveInfinity;

        children.Add(new State(lowerBound + costArray[currentIndex, i],
            newChildren, newCost, newPath,
            currentCost + cost, i, cities, depthIntoSolution + 1));
    }
    return children;
}
```

Priority Queue

The way I implanted the priority queue, it stores State objects and gives them priority based on their current cost value which is a double. After a state is added or removed from the queue, the queue then calls a function that will heapify the queue based on the current cost of each state. This way the states are stored in a fashion where the next state in the queue that has a lowest cost is stored at the top of the priority queue, and will be the next state effectively expanded to be looked at.

Initial BSSF

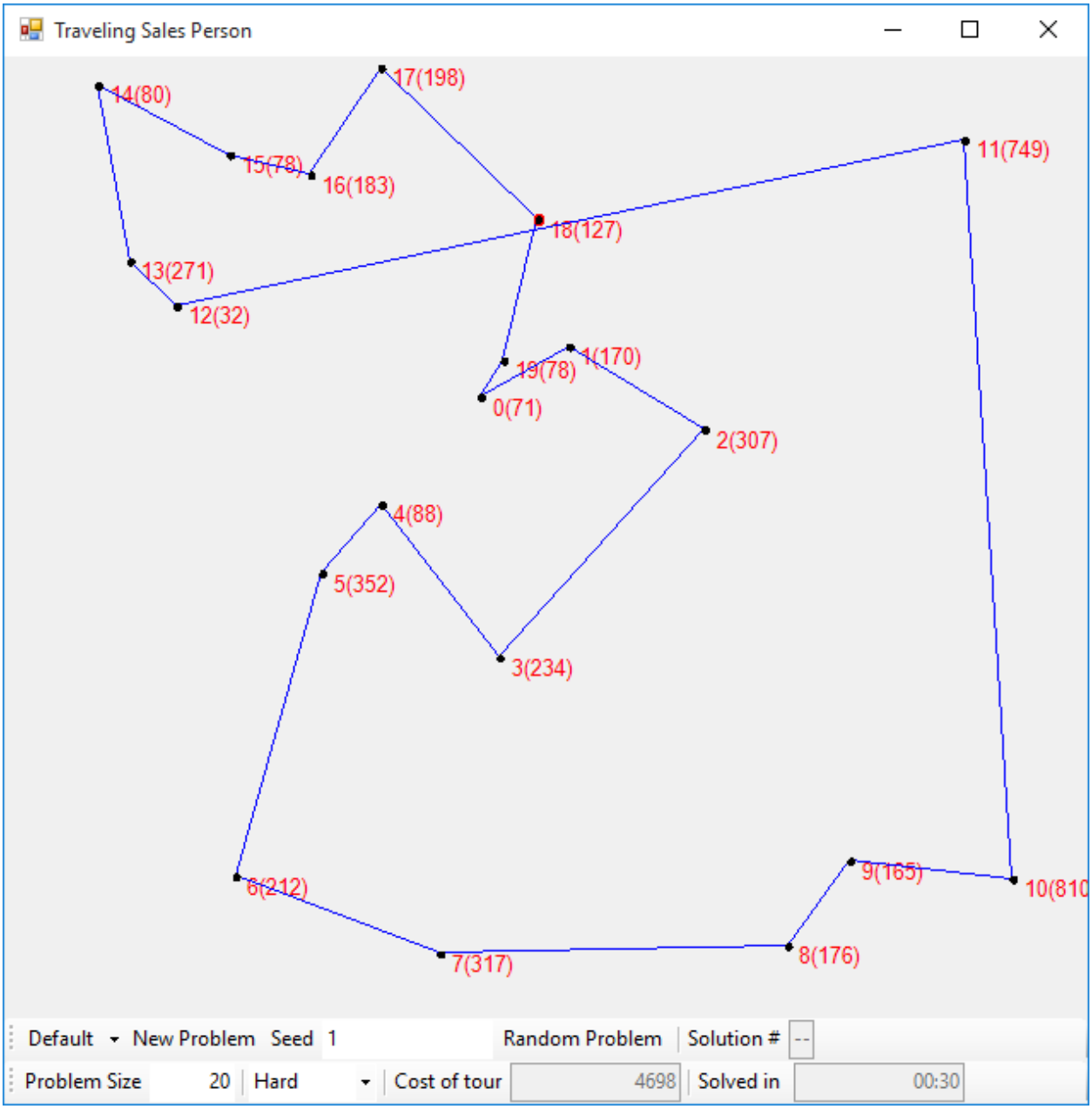
To calculate the BSSF initially the program first stores the costs to reach each of the cities, and makes sure that each of the cities is account for. Then with the costs to reach each of the cities, the cheapest cost paths between any two cities is stored. After these values are stored there is a chance that by swapping out some of the paths there can be a better BSSF created, so there is a loop entered that swaps any two, to see if by swapping we can find a BSSF, and if there is, then that BSSF is chosen and is proceeds to determine a better BSSF.

Results

# Cities	Seed	Running Time (Sec.)	Cost of best tour found	Max # of Stored states at a given time	# of BSSF updates	Total # of States Created	Total # of States Pruned
15	20	9*	3121	24588	0	682393	657805
16	902	<0*	3135	550	1	25098	24548
16	355	19*	3015	39782	2	1217972	1178190
20	1	30	4698	440555	2	639747	199192
20	155	30	3941	494607	1	693488	198881
30	20	30	3959	327281	3	377235	49954
45	35	30	5967	178945	2	181328	10383
18	655	30	3472	642832	2	1028656	385824
12	366	<0*	3515	373	1	7817	7444
14	500	4*	3560	13193	0	341309	328116

With these results it is seen that as the complexity increases with the number cities in the problem, the longer it takes to solve the problem. After increasing to more than 16 cities the time it took to solve the problem usually capped out. Also, as the numbers got significantly larger, the number of states that were pruned in the allotted time got smaller, showing the need of much more time to go through the states to reach a “more” optimal solution. On two of them it seems as though, based on the way the algorithm was implemented must have been lined up perfectly to

solve the problem in less than a second. With the initial BSSF it seems that if it is updated more than once initially the worse off the problem is going to be.



Example of a hard solution.