CS 325 Analysis of Algorithms

Gregory Oertli
Victor Campa
Tobias Bird

Traveling Salesman Problem

Branch and Bound:

Description: “Branch and Bound is a general method for finding optimal solutions to problems... [with an optimization trick, where it ignores] large parts of the search space by using previous estimates” (RJLipton+KWRegan) The Branch and Bound approach for the Travelling Salesman Problem creates a level order tree and a reduced matrix to calculate the cost of nodes in a tree and the minimum cost path through the tree. Branch and Bound is an optimized version of backtracking, it accomplishes this by it only continues calculating the children nodes of the smallest cost node, instead of calculating every node out to end. The branch and bound approach for the travelling salesman problem is difficult to identify a time complexity for, the best we can find for it is that it would run a worst case like a brute force problem.

Pseudocode:

```
tour[N][N]
TSPbnb(tour[][]) {
        Int path[N]
        Bound = 0

        Set path indexes  to -1;
        Set visited indexes to 0;

        For i < N {
                Int Bound += the first minimum edge + the second minimum edge in the matrix
        }
        Set the first visited tab to 0
        Set the first curr path to 0

        Recursive call to function tspBnBRec() to find node values
}

tspBnBRec(tour[][], bound, weight, level, path) {
        There are 3 checks to make for the final
        1 Check if level is N //when you reach N you have covered, if so check 2
        2 Check if there is a route back to the first vertex, if so set current tour to the weight plus
the weight back to the first vertex and check 3
```

3 Check if the current tour is less than the final tour and if it is set final tour to current tour

```
For i < N {
    if(tour[path[level-1][i] != 0 && i has not been visited) {
        Weight += tour[path[level-1][i]
        If on the first level, then calculate the current bound with the minimum
edge cost
        Else calculate the current bound with the second minimum edge cost plus
the minimum

        If the current bound + the weight is less than the final tour weight, then
            The path at the level gets i
            i becomes true in the visited section
            And the function calls itself recursively with everything the same
        but the level up by one


        The weight goes down by the cost of the node at the previous current
        path and i

        And the visited array gets reset to the current path
    }
}
}
```

Implementation: We attempted to implement this method, however our implementation of it appears to be too slow to solve any graph with more than 25 vertices. We were unable to optimize it at this time so we choose to focus on an implementation that works. Because of this we do not have data for the best tours or the best solution for each test instance.

Sources:
https://www.youtube.com/watch?v=1FEP_sNb62k This youtube video was very helpful in understanding the approach branch and bound takes to solve the travelling salesman problem on a higher level.
https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/ This site was helpful in understanding a simple coded example of the problem as well as a simple recap of the concepts of branch and bound
https://rjlipton.wordpress.com/2012/12/19/branch-and-bound-why-does-it-work/ This site was helpful expressing the difficulties of getting the time complexity for branch and bound as well as a very proffesional description of the branch and bound method in general.

Nearest Neighbours Algorithm:

Description: Nearest Neighbours Algorithm works by calculating the euclidean distance between all the connections to determine which is the closest point to connect. Where a point can only be visited once. This algorithm does not guarantee that all the points will be the most optimal. The position that you start does determine what the result is, meaning that starting at two different points will get two different final paths. On the other hand, If the Traveling Salesman problem were brute force then it would not matter where you start because the results should be optimal (the same) but in doing so it will make the solution extremely slow. I will be creating a modified version of Nearest Neighbours where we will be continuously getting one point at a time rather than checking multiple points at once. The time complexity should be O(n^3). If the file is big enough there is a timer that will break the calculations and find the most optimal choice with what it has. Below is Pseudocode that explains my implementation of my nearest Neighbours Algorithm.

Pseudocode

KNN(pass file count, pass the city number arr, pass the x and y coordinates arr)

Create int array called travaled
Create int array called best_traveled
Create int named successor
Create int array called traveled alloc with size of file count
Create int array called best_traveled alloc with size of file count
Create int array called dis_trav for summing up all the euclidean distances
Create int vector called identification
Create int vector called nearest
Create int vector called total_smallest
Create int called visited

Start a timer (used to not go over the time limit of 300 seconds)

For loop through all starting points of x
    For loop here to reset the city numbers back to being unvisited
    Reset the visited, dis_trav to 0 for the next x point
    Assign the x'th city to the traveled array where x will be mult points
    Assign the city arr to -1
    Loop through the file count for all cities (lines of the file -1) of i
        Loop through the file count for all cities (lines of the file) of b
            Check if the number arr is -1 which represents visited values of i
                If the city is not visited
                    Get euclid distance from last city saved to *all other cities* b
                        Store the distance in nearest vector
                        Store the identification to a vector of number i
                If the city is visited
                    Skip the city since it was already visited

Find the minimum element from nearest vector
Get the index from the min element
Increment visited
Store the next city min element that was found as -1
Add the distance from the minimum element to dis_trav
Remove all the vectors in nearest and identification
Do the euclid distance from the first traveled point of x to the last traveled point
Add the distance calculated from the euclid distance to dis_trav
Add the dis_trav into the vector total_smallest

If the total_smallest successor is > total_smallest x
Then the successor failed
For loop the data of traveled array to best travel
Check the timer and see if its >= 255
If it is then we break the x for loop

Store the total_smallest successor into a .tour file
For loop through the best_travel vector to store in a .tour file

Print to the terminal the total time it took and that the .tour file was successfully done

Sources
https://en.wikipedia.org/wiki/Nearest_neighbor_search
https://www.youtube.com/watch?v=b7wW5mCPh5U&t=313s
https://www.youtube.com/watch?v=fFfizorMPuk

Dynamic programming Held-Karp algorithm:

Description: The Held-Karp algorithm solves the TSP using a bottom up method. This is a more efficient algorithm than the normal dynamic programming algorithm. The algorithm starts with the last cities in the array to determine the smallest distance. This process will loop through the children nodes until the starting city is reached with the final distance with the smallest distance traveled. There will end up being n-1 paths at the end for every possible ordering of the paths. Then all of the remaining paths are compared to find the final smallest distance traveled. The worst case runtime for this algorithm is O($n^2 * 2^n$).

Pseudocode:

```
orderArray(){
    If(size = finalSize)
        return array
    for(i < finalSize)
        Add city to orderingArray
```

```
                Remove city from availableCities
                orderArr(orderingArray)
                Return city to availableCities
        }
        Held-Karp(distanceArr){
                orderArr()
                finalDistanceArr[];
                for(i=0, i<n-1){
                        curDistance = 0
                        checkPos = 0
                        for(j=n-2, j>=0){
                                curDistance = curDistance + min(orderArr[j,j+1], orderArr[j+1, 0])
                        }
                        distanceArr[i] = curDistance;
                }
                minDistance = distanceArr[0];
                for(i=0, i<n-1){
                        if (minDistance > distanceArr[i])
                                minDistance = distanceArr[i]
                }
                Print minDistance
        }
```

Sources:
https://www.youtube.com/watch?v=-JjA4BLQyqE
https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm#Algorithm
https://stackoverflow.com/questions/14132877/order-array-in-every-possible-sequence
https://www.codeproject.com/Articles/762581/Held-Karp-algorithm-implementation-in-Csharp

Discussion of Implemented Algorithms:

We chose to implement the Nearest Neighbor algorithm because this greedy algorithm is useful for smaller TSP problems. The Language used for the Nearest Neighbor is C++. In the pseudo code for Nearest Neighbor shown above the data structure follows mainly three loops. The first loop is in charge of the point that it starts. This is done because the initial node affects what the total distance is. The second loop is in charge of getting the smallest value out of all the calculated euclidean distances. The last inner loop is in charge of checking if the city is visited or not; if it isn't, then the euclidean distance is calculated and stored temporarily for the second loop to verify. The efficiency of greedy algorithms have made the Nearest Neighbor algorithm very popular. Since we were allowed to approximate the smallest distance traveled we decided that going through every possible combination of cities visited would take too long to get an exact answer. The Nearest Neighbor has a run time of $O(n^3)$ while a brute force algorithm has

an exponential running time. We decided a potentially worse estimate for the final distance traveled was worth the speed of the algorithm's run time. One of our team mates implemented the Branch and Bound algorithm but unfortunately it turned out to be super inefficient because the algorithm is done in a brute force manner. We tested it out with test-input-1.txt and it takes longer than 5 minutes to calculate. So in conclusion we decided to use the Nearest Neighbor Algorithm for all our test cases.

Best Tours: (5 minutes = 300 seconds)

	Example 1	Example 2	Example 3
Distance	130921	2975	1930997
Time (seconds)	0.04	1.75	295.96

	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7
Distance	5911	8011	14826	19711	27128	39469	61972
Time (s)	0.01	0.09	1.28	10.65	81.41	290.15	290.84