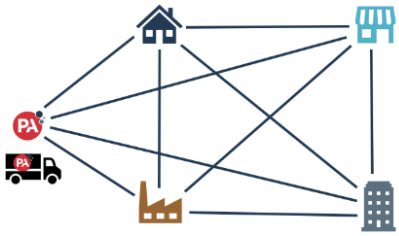
King Abdulaziz University
DEPARTMENT OF COMPUTER SCIENCE
Faculty of Computing and Information Technology
CPCS223 Analysis & Design of Algorithms Project – Winter 2023



# Empirical Analysis of Travel salesman and Branch and Bound Algorithms for finding the shortest path

Section: B2A

Group number: 4

| Student name | ID |
|---|---|
| (Leader) Ebtihaj [redacted] | 2[redacted] |
| Layan [redacted] | 1[redacted] |
| Hanoof [redacted] | 1[redacted] |
| Nsreen Hujjatulaah | [redacted] |

Instructor: Dr. Hajar Alharbi

## Assignments distribution:

**Due to that, we were working on different codes to fulfill the requirements then we shifted to a new code each of us has participates in all aspects of the project.**

| **Part &&student** | **Nsreen** | **Layan** | **Hanoof** | **Ebtihaj** |
|---|---|---|---|---|
| Brute force code | Fixing the code | Searching for appropriate codes | Searching for appropriate codes | Run and test different codes |
| Second code which has been changed many times | Fixing the branch and bound code | | Researching for different codes and finding resources | |
| Report | Pseudocode of 2d algorithm | Introduction - Characteristics of The Input Sample | Pseudocode of 2d algorithm | Pseudocode of 1st algorithm and analysis tools Experiment's purpose |
| Excel gathering data and calculation | All students participate in the two sections | | | |
| Graphs | | | | |
| Design and formatting of the report and tables | 100% | | | |

# Table of contents

# Illustrations:

**Tables:**



**Figures:**

# 1. Introduction

Algorithm analysis is a widely used and important field in computer science that helps determine the efficiency of an algorithm and predict its behavior without requiring it to be implemented on a specific computer. It aids in evaluating an algorithm's performance and comparing it to other algorithms to determine which one has the best performance. To analyze an algorithm, two approaches are provided: mathematical analysis and empirical analysis, both of which have distinguishing characteristics.

Even though the power of mathematical analysis, there is a limitation. Analyzing some problems can be complicated by using mathematical calculations. For this purpose, we need to study these problems based on real-world performance and experiments using empirical analysis.

# 2. Problem Description

In this report, we will discuss The Traveling Salesman Problem (TSP). It is an algorithmic problem that seeks the shortest non-overlapping path through all points in each list while visiting each point exactly once. It is an NP-hard problem, meaning that the time required to solve the problem increases rapidly as the number of points increases.

Therefore, we will empirically analyze the effectiveness of two algorithms (Traveling salesman using Brute Force Algorithm & TSP using Branch and Bound Algorithm) to find a path at the lowest cost to solve the problem of the shipment delivery representative which is our problem in this project.

The Brute Force Algorithm considers all possible permutations of the cities and thus guarantees the shortest route. However, it is very inefficient, as the number of permutations increases exponentially as the number of cities increases. For this reason, it is not suitable for large problems. The steps for solving the TSP using this algorithm are as follows:

(i) Generate the graph for all cities.
(ii) Calculate the distance for all cities.
(iii) Create all possible permutations for all cities.
(iv) Calculate the total distance for each permutation.
(v) Finally, select the permutation with the shortest total distance.

In the Branch and Bound Algorithm, this algorithm divides the problem to be solved into several sub-problems. To solve the TSP using the Branch and Bound algorithm by following this step:

(i) select a start node.
(ii) set the bound to a very large value (say, infinity).
(iii) choose the shortest path between the unvisited and current nodes.
(iv) add the distance to the current distance.
When the current distance is less than the bound, we have successfully completed the algorithm.

# 3. Empirical Analysis of the algorithm

## 3.1. Experiment's purpose:

The main purpose of this experiment is to practice the analysis of an algorithm by applying it on two similar algorithms for finding the shortest tour. Also, to make the decision of what is better relying on the order of growth.

## 3.2. Choice of efficiency metric:

Two basic approaches to measuring the efficiency of a class are basic operation counter and running time. A basic operation counter has an advantage over running time for being stable on different machines, but it is a more complex practice because we need to determine the position of the counter accurately in such a code. However, we choose the running time for its simplicity. Running time is the determination of the amount of time to execute the code of the algorithm.

In order to measure it, there are many tools found such as the time command in UNIX, function clock in C language, and currentTimeMillis() in Java.

We reported the start time after creating the array of distances and the array that points to each city. At the end of the method that has the basic operation we reported the end-time, then we used the formula: execution-time= end-time -start-time.

We used milli seconds as a unit because it is more realistic to measure data.

## 3.3. Characteristics of The Input Sample:

In this study, selecting the input and input instances is just as crucial as selecting the efficiency metric. Because of this, we demonstrate our decision-making process for the input in this section. We chose the appropriate range of 3 to 10 as the input size for the "Traveling salesman" problem because we needed to ensure that it was neither too large nor too little.

We created random numbers for each distance and values array to remove any bias toward particular input instances (only odd and even numbers). In these kinds of tests, applying the same input size to different instances can greatly enhance our analysis. It will demonstrate that our program does not favor particular inputs on particular measurements. For this reason, we used the trial process and ran the same input size five times, with each run producing a different set of random numbers for the array of distances.

There are two inputs needed for the salesman problem. The first input is a random distance array, which consists of random numbers. Each trail has a different range of distances. We utilized a range of distances from 1 to 10 in the first trial, 11 to 20 in the second, and so on. The second input is the start point or node.

## 3.4. Implementation Algorithms and analysis tools:

- **TSP-brute force algorithm:**

**ALGORITHM** tsp_brute_force(Array[0…n][0…n],start node)

//Generate all possible paths then select the optimal path.

//Input: 2-dimensional array of distances between every two cities and start node.

//Output: The optimal path

```
Cities [0..n]              //create array that represents cities
For i ← 0 to n do
        Cities[i] ← i
        Min-cost← +∞
DO
        K← start node
       For i←0 to n do
         K← Cities [i]
         Current cost ← current cost+ array[k][s] // calculate the path between start node and next point
 Current cost ← current cost+ array[k][s] // calculate the path between final node and start point

If current cost <min-cost
      Min-cost←current cost
While findNextPermutation() =true
Repeat
Return min-cost
```

To see the full code, check Appendix A

The following is the implementation of the findNextPermutation() method that returns a Boolean value and used by TSP brute force main function.

```
public static boolean findNextPermutation(ArrayList<Integer> data) {
    // If the given dataset is empty
    // or contains only one element
    // next_permutation is not possible
    if (data.size() <= 1) {
        return false;
    }
    int last = data.size() - 2;
    // find the longest non-increasing suffix and find the pivot
    while (last >= 0) {
        if (data.get(last) < data.get(last + 1)) {
            break;
        }
        last--;
    }
    // If there is no increasing pair there is no higher order permutation
    if (last < 0) {
        return false;
    }
    int nextGreater = data.size() - 1;
    // Find the rightmost successor to the pivot
    for (int i = data.size() - 1; i > last; i--) {
        if (data.get(i) > data.get(last)) {
            nextGreater = i;
            break;
        }
    }
    // Swap the successor and the pivot
    data = swap(data, nextGreater, last);
    // Reverse the suffix
    data = reverse(data, last + 1, data.size() - 1);
    // Return true as the next_permutation is done
    return true;
}
```

*Figure 1: Screenshot code find permutation function used by Tsp of brute force main method*

The following is the implementation of swap and reverse function for generating the next permutation:

```
//================================================================================
// Function to swap the data
// present in the left and right indices

    public static ArrayList<Integer>
        swap(ArrayList<Integer> data, int left, int right) {
        // Swap the data
        int temp = data.get(left);
        data.set(left, data.get(right));
        data.set(right, temp);
        // Return the updated array
        return data;
    }
//================================================================================
// Function to reverse the sub-array
// starting from left to the right
// both inclusive

    public static ArrayList<Integer>
        reverse(ArrayList<Integer> data, int left, int right) {
        // Reverse the sub-array
        while (left < right) {
            int temp = data.get(left);
            data.set(left++, data.get(right));
            data.set(right--, temp);
        }
        // Return the updated array
        return data;
    }
//================================================================================
```

*Figure 2: Screenshot code of reverse and swap functions used by Tsp brute force.*

- **TSP-Using Branch And Bound**

**ALGORITHM** TSP_brunch_and_bound(arry[0..n][0..n],s)

//Generate all possible paths then select the optimal path.

//Input: 2 dimensional array of distances between each two cities, and start node.

//Output: The optimal path

```
min ← +∞;  first ← +∞;  second ←  +∞
Visetsed ← false
Node ← arry[0..n].length
Curr_path[Node+1]
curr_bound  ←  0;
For i ← 0 to i<= Node+1 do
        Curr_path ← arry[i]<--(-1)

For i ← 0 to i <node do
                For  k ← 0 to k< Node  do //for finding find the minimum edge cost
                    if arry[i][k] < min and  i notequal k then do
                            min = arry[i][k];
                For  j ← 0 to j< Node do     // find the second minimum edge cost
                     if  arry[i][j] <= first then
                             second  ←  first
                             first  ←  arry[i][j]

                     if arry[i][j] <= second and arry [i][j] notequal first then
                             second  ← arry[i][j];

   curr_bound ←  curr_bound+ (min + second)


curr_bound  ← (curr_bound=1)? curr_bound/2 + 1 : curr_bound/2
curr_path[0] = 0;
Visetsed ← true
TSPRec(arry[0..n][0..n], curr_bound, 0, 1, curr_path) //methode used recursive call
```

To see the full code, check Appendix B

The following is the implementation of the TSPRec(arry[0..n][0..n], curr_bound, 0, 1, curr_path) method that void

```
public static void TSPRec(int adj[][], int curr_bound, int curr_weight,
        int level, int curr_path[]) {
    // base case is when we have reached level N which means we have covered all the nodes once
    if (level == N) {
        // check if there is an edge from last vertex in path back to the first vertex
        if (adj[curr_path[level - 1]][curr_path[0]] != 0) {
            // curr_res has the total weight of the solution we got
            int curr_res = curr_weight + adj[curr_path[level - 1]][curr_path[0]];
            // Update final result and final path if current result is better.
            if (curr_res < final_res) {
                copyToFinal(curr_path);
                for (int i = 0; i <= N; i++) {
                    // print the paths
                    System.out.printf("%d ", curr_path[i]);
                    curr_path[N] = curr_path[0];
                }
                System.out.println("");
                final_res = curr_res;
                System.out.println("Cost this path = " + curr_res);
            }
        }
        return;
    }
    // for any other level iterate for all vertices to build the search space tree recursively
    for (int i = 0; i < N; i++) {
        // Consider next vertex if it is not same (diagonal entry in adjacency matrix and not visited already)
        if (adj[curr_path[level - 1]][i] != 0 && visited[i] == false) {
            int temp = curr_bound;
            curr_weight += adj[curr_path[level - 1]][i];
            // different computation of curr_bound for level 2 from the other levels
            if (level == 1) {
                curr_bound -= ((firstMin(adj, curr_path[level - 1]) + firstMin(adj, i)) / 2);
            } else {
                curr_bound -= ((secondMin(adj, curr_path[level - 1]) + firstMin(adj, i)) / 2);
            }
            // curr_bound + curr_weight is the actual lower bound for the node that we have arrived on
            // If current lower bound < final_res, we need to explore the node further
            if (curr_bound + curr_weight < final_res) {
                curr_path[level] = i;
                visited[i] = true;
                // call TSPRec for the next level
                TSPRec(adj, curr_bound, curr_weight, level + 1, curr_path);
            }
            // Else we have to prune the node by resetting
            // all changes to curr_weight and curr_bound
            curr_weight -= adj[curr_path[level - 1]][i];
            curr_bound = temp;
            // Also reset the visited array
            Arrays.fill(visited, false);
            for (int j = 0; j <= level - 1; j++) {
                visited[curr_path[j]] = true;
            }
        }
    }
}
```

*Figure 3: Screenshot codeTspRec function used in branch and bound algorithm*

.

Analysis tools:

- NetBeans IDE 8.2 used to run the java code.
- Microsoft Excel, used to create analysis tables and to represent them in graphs.

## 3.5. Generate a sample of inputs.

For the number of cities, it will be entered by the user, and we have sent it as a parameter in the method called generateDistancesCites().

As for the distances between cities, we've added a method to our program called generateDistancesCites().

The method takes three parameters (n, min, max) are all of type (int). The first parameter for the number of cities. As well as the other parameters min and max to determine the range of the array. The min and max change in each trial, for example, the first trial: 1-10, the second trial 11-20, and so on...

In addition, the method contains an instance of the class Random, i.e., Random rand = new Random(). There are also two loops for generating random distances between cities, we used Rand.nextInt((max - min) + 1). The method returns the two demission arrays (matrices) of the randomly generated distances. The following figure shows the code for this method.

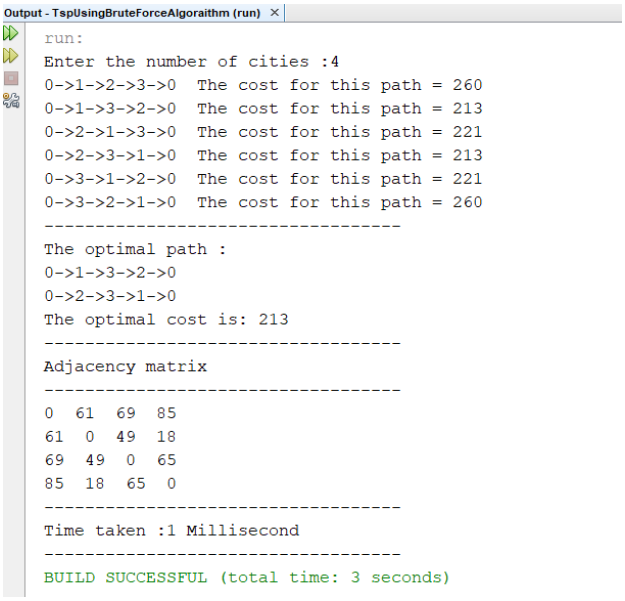
```
//============================================================================
    //method to genrate cites distances randomly

    public static int[][] genratedistancesCites(int n, int min, int max) {
        int distances[][] = new int[n][n];
        Random rand = new Random(100);//genrate randomly
        //for loop to genrate randomly distances to store distances in 2d array
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < i; j++) {
                //genrate cites distances randomly
                int value = Math.abs((min + rand.nextInt((max - min) + 1)));
                distances[i][j] = value;
                distances[j][i] = value;
            }
        }
        return distances;
    }
//============================================================================
```

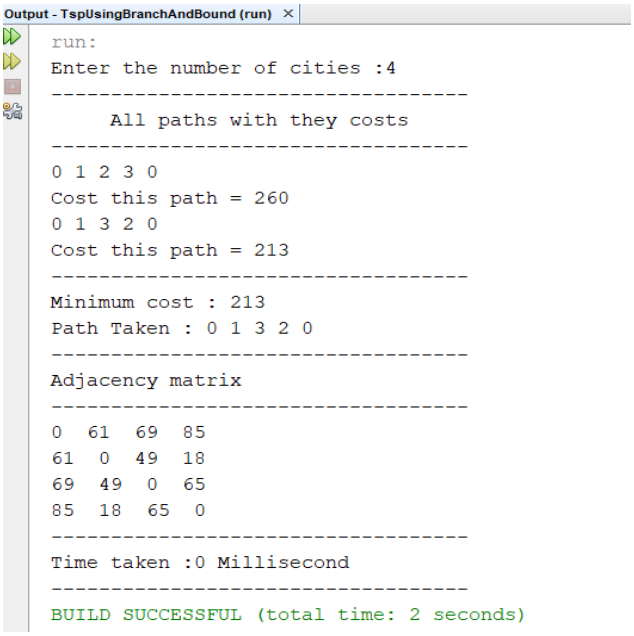
*Figure 4: Screenshot code metod generateDistancesCites().*

## 3.6. Program Run

Sample run of brute force travelling salesman:




Sample run of branch and bound algorithm:



## 3.7. Data Collection

For each input size, we execute the two specified algorithms five times (3-10). In each trail, a different array was produced randomly. The time taken is shown in the two tables below, taking into account the average, best, and worst run times for each input size.

| input-size | trail1(1:10) | trail2(11:20) | trail3(21:30) | trail4(31:40) | trail5(41:50) | Best-case | Avarge-case | Worst-case |
|---|---|---|---|---|---|---|---|---|
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 2 | 2 | 2 | 1 | 2 | 1 | 1.8 | 2 |
| 6 | 8 | 13 | 13 | 14 | 7 | 7 | 11 | 14 |
| 7 | 48 | 43 | 42 | 51 | 41 | 41 | 45 | 51 |
| 8 | 285 | 236 | 245 | 251 | 258 | 236 | 255 | 285 |
| 9 | 2377 | 2201 | 2053 | 2064 | 2181 | 2053 | 2175.2 | 2377 |
| 10 | 23965 | 20332 | 19925 | 20894 | 20630 | 19925 | 21149.2 | 23965 |

*Table 1: Brute force (TSP) Runtime*

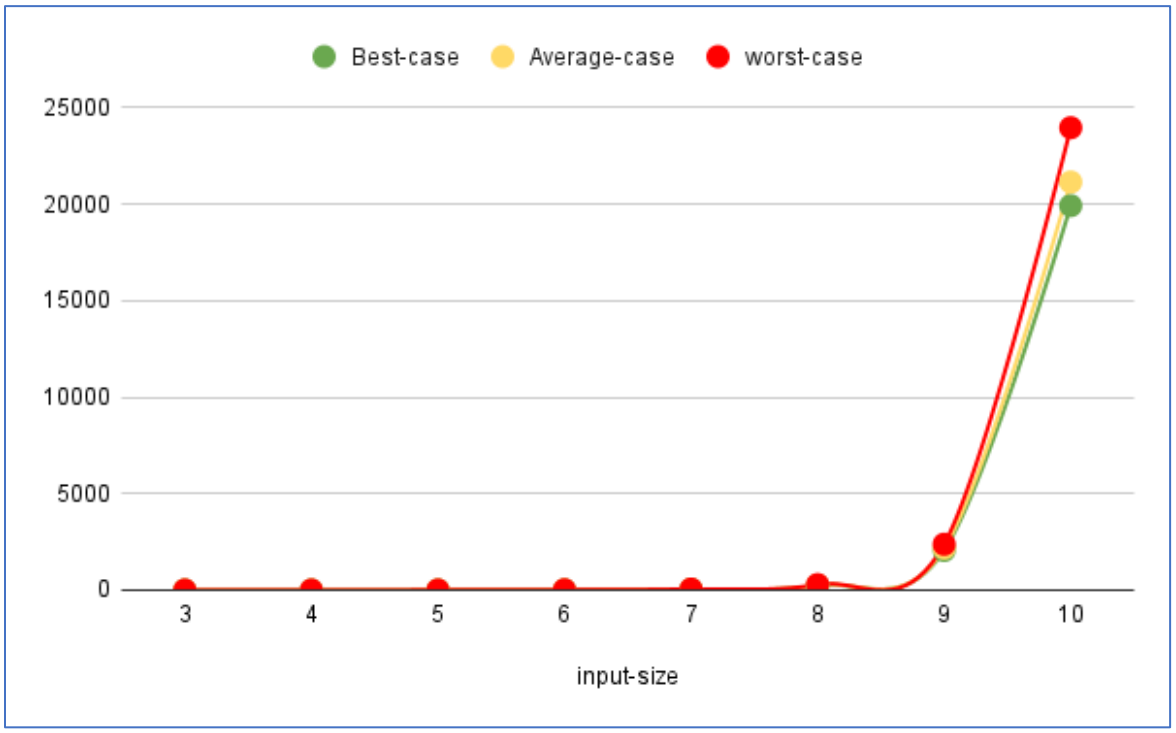



*Figure 5: Brute force (TSP) Graph*

The graph shows that when the inputs were small (3-7), the time was shorter, and once the inputs became (8-10) the time increased exponentially. We conclude from this graph that this algorithm is highly affected by the input.

| Input-size | trail1(1:10) | trail2(11:20) | trail3(21:30) | trail4(31:40) | trail5(41:50) | Best-case | Average-case | Worst-case |
|---|---|---|---|---|---|---|---|---|
| 3 | 6 | 7 | 9 | 7 | 8 | 6 | 7.4 | 9 |
| 4 | 6 | 8 | 9 | 9 | 9 | 6 | 8.2 | 9 |
| 5 | 8 | 9 | 10 | 10 | 11 | 8 | 9.6 | 11 |
| 6 | 9 | 10 | 11 | 13 | 11 | 9 | 10.8 | 13 |
| 7 | 14 | 13 | 15 | 12 | 15 | 12 | 13.8 | 15 |
| 8 | 15 | 15 | 16 | 14 | 13 | 13 | 14.6 | 16 |
| 9 | 15 | 14 | 15 | 15 | 16 | 14 | 15 | 16 |
| 10 | 18 | 19 | 17 | 16 | 17 | 16 | 17.4 | 19 |

*Table 2: Branch And bound (TSP) Runtime*

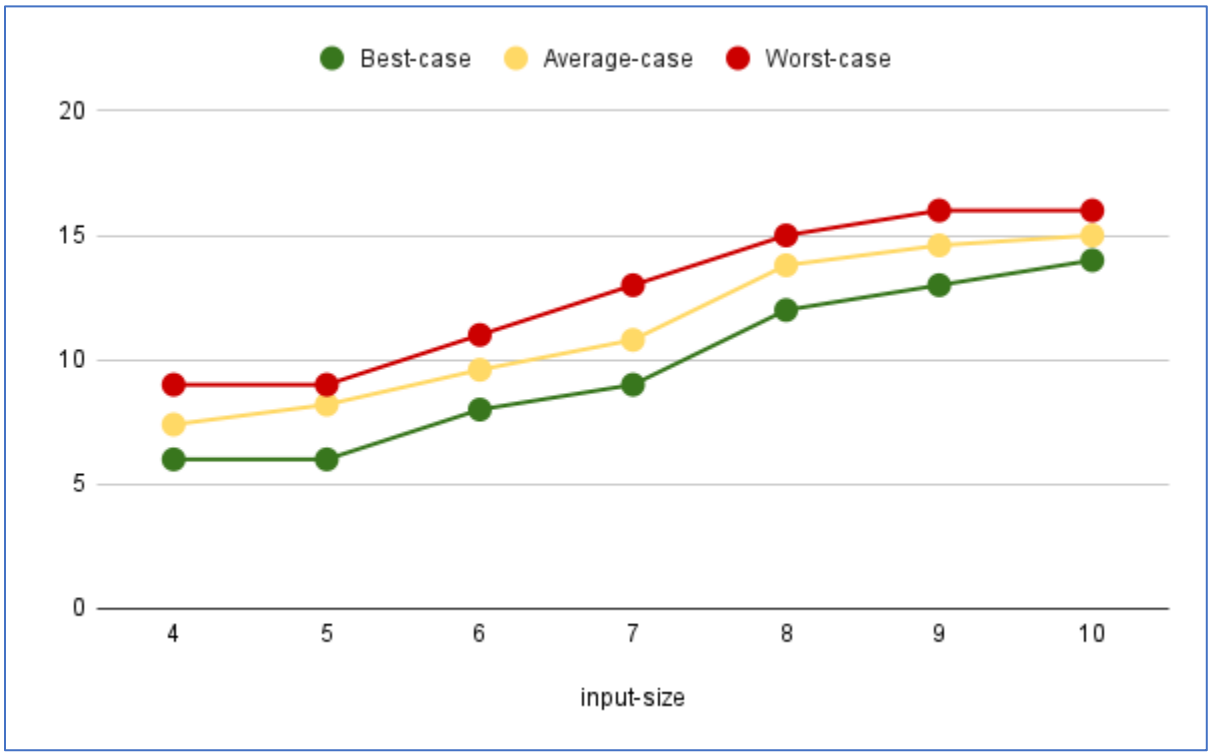



*Figure 6: Branch And bound (TSP) Graph*

The graph shows that when the inputs are small (3-6), or when the inputs become larger (7-10), we notice that the time increases gradually as the inputs increase.

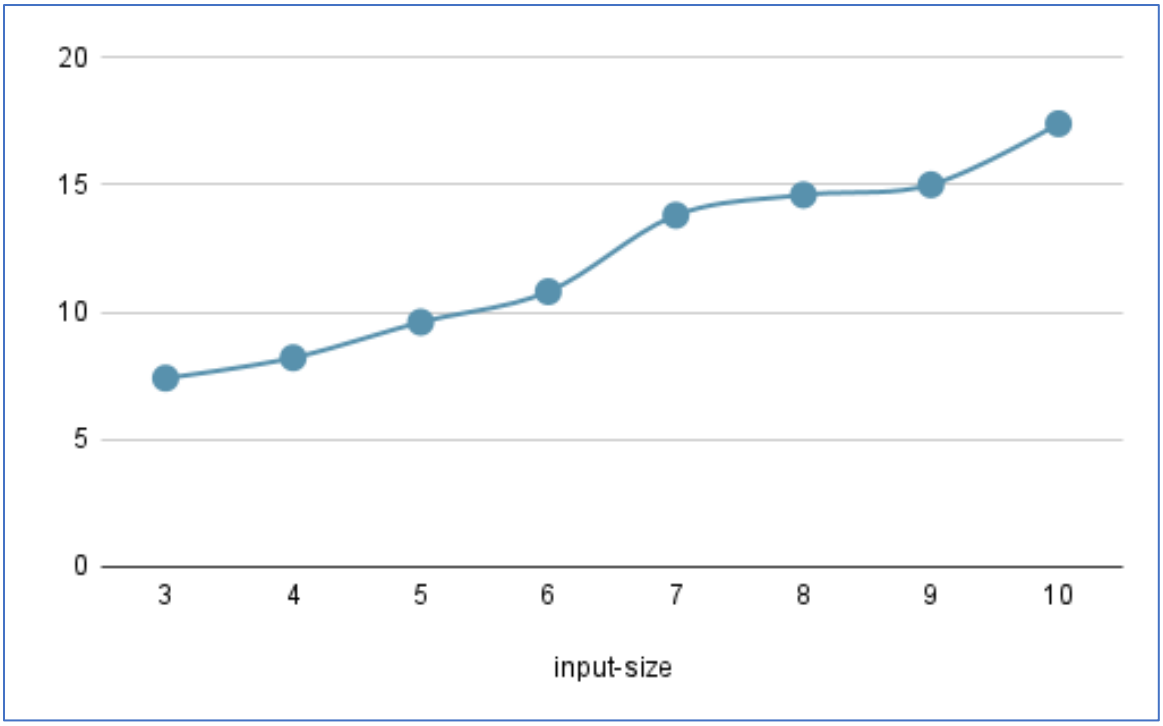



*Figure 7: Branch and bound (TSP) Average Graph*

We display these statistics to represent each algorithm. The input size is represented by the X-axis, while the physical computation time is shown by the Y-axis.

## 3.8. Data analysis and order of growth

### 3.8.1. Class Efficiency

| Input-size (n) | 1 | log(n) | n | n*log(n) | n^2 | n^3 | 2^n | n! |
|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 1,584962501 | 3,00 | 4,754887502 | 9 | 27 | 8 | 6 |
| 4 | 1 | 2 | 4,00 | 8 | 16 | 64 | 16 | 24 |
| 5 | 1 | 2,321928095 | 5,00 | 11,60964047 | 25 | 125 | 32 | 120 |
| 6 | 1 | 2,584962501 | 6,00 | 15,509775 | 36 | 216 | 64 | 720 |
| 7 | 1 | 2,807354922 | 7,00 | 19,65148445 | 49 | 343 | 128 | 5040 |
| 8 | 1 | 3 | 8,00 | 24 | 64 | 512 | 256 | 40320 |
| 9 | 1 | 3,169925001 | 9,00 | 28,52932501 | 81 | 729 | 512 | 362880 |
| 10 | 1 | 3,321928095 | 10,00 | 33,21928095 | 100 | 1000 | 1024 | 3628800 |

*Table 3: Class Efficiency*

### 3.8.2. Brute force

| Input-size (n) | Best | 1 | log(n) | n | n*log(n) | n^2 | n^3 | 2^n | n! |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 0 | 1 | 2,512106129 | 9 | 22,60895516 | 81 | 729 | 64 | 36 |
| 4 | 0 | 1 | 4 | 16 | 64 | 256 | 4096 | 256 | 576 |
| 5 | 1 | 0 | 1,747493888 | 16 | 112,564471 | 576 | 15376 | 961 | 14161 |
| 6 | 7 | 36 | 19,49255612 | 1 | 72,41627062 | 841 | 43681 | 3249 | 508369 |
| 7 | 41 | 1600 | 1458,678138 | 1156 | 455,759116 | 64 | 91204 | 7569 | 24990001 |
| 8 | 236 | 55225 | 54289 | 51984 | 44944 | 29584 | 76176 | 400 | 1606727056 |
| 9 | 2053 | 4210704 | 4201803,336 | 4177936 | 4098481,514 | 3888784 | 1752976 | 2374681 | 130196123929 |
| 10 | 19925 | 396965776 | 396873257,2 | 396607225 | 395682940,2 | 393030625 | 358155625 | 357247801 | 13023978765625 |
| SUM | | 401233343 | 401130836 | 400838343 | 399827093 | 396950811 | 360139863 | 359634981 | 13155807129753 |

*Table 4: Brute force Best case approximate order of growth*

| Input-size (n) | Average-case | 1 | log(n) | n | n*log(n) | n^2 | n^3 | 2^n | n! |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 0 | 1 | 2,512106129 | 9 | 22,60895516 | 81 | 729 | 64 | 36 |
| 4 | 0 | 1 | 4 | 16 | 64 | 256 | 4096 | 256 | 576 |
| 5 | 1,8 | 0,64 | 0,2724089362 | 10,24 | 96,22904624 | 538,24 | 15178,24 | 912,04 | 13971,24 |
| 6 | 11 | 100 | 70,81285611 | 25 | 20,33807059 | 625 | 42025 | 2809 | 502681 |
| 7 | 45 | 1936 | 1780,219299 | 1444 | 642,5472404 | 16 | 88804 | 6889 | 24950025 |
| 8 | 255 | 64516 | 63504 | 61009 | 53361 | 36481 | 66049 | 1 | 1605204225 |
| 9 | 2175,2 | 4727145,64 | 4717714,647 | 4692422,44 | 4608194,987 | 4385673,64 | 2091494,44 | 2766234,24 | 130107952743 |
| 10 | 21149,2 | 447246363,2 | 447148159,4 | 446865776,6 | 445884641,7 | 443068820,6 | 405990260,6 | 405023675 | 13015144294741 |
| SUM | | 452040063,5 | 451931235,9 | 451620712,3 | 450547043,4 | 447492491,5 | 408298636,3 | 407800840,3 | 13146882918998 |

*Table 5: Brute force Average case approximate order of growth*

| Input-size (n) | Worst-case | 1 | log(n) | n | n*log(n) | n^2 | n^3 | 2^n | n! |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 0 | 1 | 2,512106129 | 9 | 22,60895516 | 81 | 729 | 64 | 36 |
| 4 | 0 | 1 | 4 | 16 | 64 | 256 | 4096 | 256 | 576 |
| 5 | 2 | 1 | 0,1036376983 | 9 | 92,34519005 | 529 | 15129 | 900 | 13924 |
| 6 | 14 | 169 | 130,3030811 | 64 | 2,279420564 | 484 | 40804 | 2500 | 498436 |
| 7 | 51 | 2500 | 2322,53104 | 1936 | 982,7294269 | 4 | 85264 | 5929 | 24890121 |
| 8 | 285 | 80656 | 79524 | 76729 | 68121 | 48841 | 51529 | 841 | 1602801225 |
| 9 | 2377 | 5645376 | 5635069,225 | 5607424 | 5515314,511 | 5271616 | 2715904 | 3478225 | 129962413009 |
| 10 | 23965 | 574273296 | 574162016 | 573842025 | 572730128,4 | 569538225 | 527391225 | 526289481 | 12994835377225 |
| SUM | | 580002000 | 579879068,7 | 579528212 | 578314727,9 | 574860036 | 530304680 | 529778196 | 13126425994552 |

*Table 6: Brute force Worst case approximate order of growth*

According to the above tables, brute force algorithm in all best, worst and average case has order of growth of 2^n which has the lowest error ratio .

- Cworst (n) = Caverage (n) = Cbest(n) ∈ Θ (2^n).

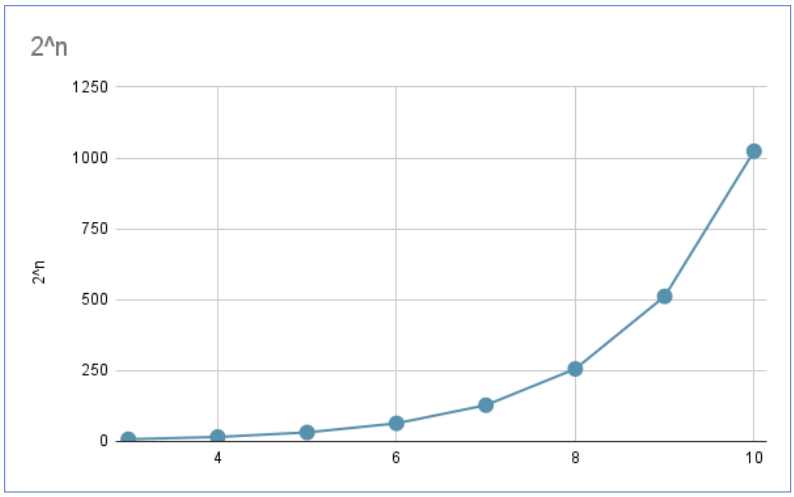



*Figure 8: Θ (2^n) Graph*

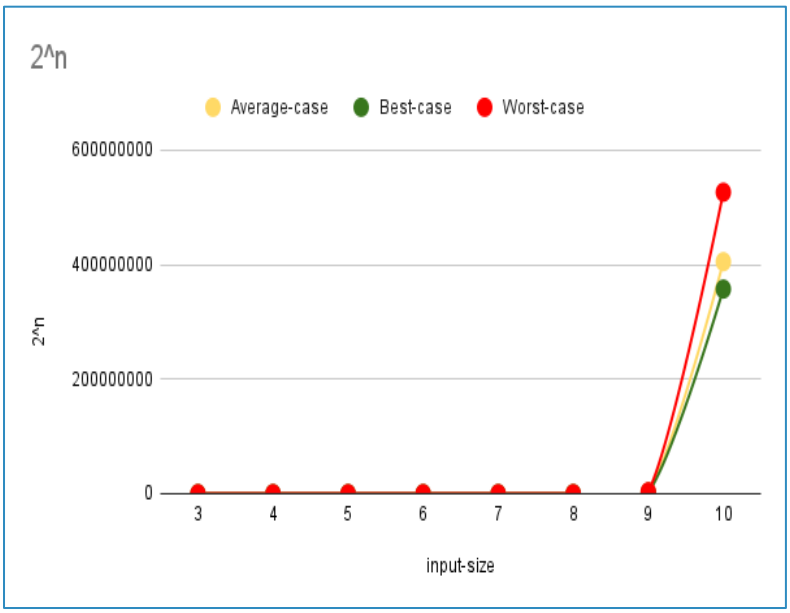



*Figure 9: Θ (2^n) Brute force Graph*

### 3.8.3. Branch and bound

| Input-size (n) | Best | 1 | log(n) | n | n*log(n) | n^2 | n^3 | 2^n | n! |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 6 | 25 | 19,49255612 | 9 | 1,550305132 | 9 | 441 | 4 | 0 |
| 4 | 6 | 25 | 16 | 9 | 4 | 100 | 3364 | 100 | 324 |
| 5 | 8 | 49 | 32,24050056 | 16 | 13,02950435 | 289 | 13689 | 576 | 12544 |
| 6 | 9 | 64 | 41,15270612 | 16 | 42,37717061 | 729 | 42849 | 3025 | 505521 |
| 7 | 12 | 121 | 84,50472353 | 36 | 58,54521436 | 1369 | 109561 | 13456 | 25280784 |
| 8 | 13 | 144 | 100 | 36 | 121 | 2601 | 249001 | 59049 | 1624654249 |
| 9 | 14 | 169 | 117,2905245 | 36 | 211,1012853 | 4489 | 511225 | 248004 | 131671733956 |
| 10 | 16 | 225 | 160,7335072 | 49 | 296,5036364 | 7056 | 968256 | 1016064 | 13168073318656 |
| SUM | | 822 | 571,414518 | 207 | 748,1071162 | 16642 | 1898386 | 1340278 | 13301395506034 |

*Table 7: Branch and bound Best case approximate order of growth*

| Input-size (n) | Average-case | 1 | log(n) | n | n*log(n) | n^2 | n^3 | 2^n | n! |
|---|---|---|---|---|---|---|---|---|---|
| 3,00 | 7,4 | 40,96 | 33,81466112 | 19,36 | 6,996620126 | 2,56 | 384,16 | 0,36 | 1,96 |
| 4,00 | 8,2 | 51,84 | 38,44 | 17,64 | 0,04 | 60,84 | 3113,64 | 60,84 | 249,64 |
| 5,00 | 9,6 | 73,96 | 52,97033066 | 21,16 | 4,038654836 | 237,16 | 13317,16 | 501,76 | 12188,16 |
| 6,00 | 10,8 | 96,04 | 67,48684111 | 23,04 | 22,18198059 | 635,04 | 42107,04 | 2830,24 | 502964,64 |
| 7,00 | 13,8 | 163,84 | 120,8382458 | 46,24 | 34,23987032 | 1239,04 | 108372,64 | 13041,64 | 25262686,44 |
| 8,00 | 14,6 | 184,96 | 134,56 | 43,56 | 88,36 | 2440,36 | 247406,76 | 58273,96 | 1624525269 |
| 9,00 | 15 | 196 | 139,9506745 | 36 | 183,0426353 | 4356 | 509796 | 247009 | 131671008225 |
| 10,00 | 17,4 | 268,96 | 198,1921086 | 54,76 | 250,2496497 | 6822,76 | 965502,76 | 1013243,56 | 13168063158063 |
| SUM | | 1076,56 | 786,2528617 | 261,76 | 589,1494109 | 15793,76 | 1890000,16 | 1334961,36 | 13301384469648 |

*Table 8:Branch and bound Average case approximate order of growth*

| Input-size (n) | Worst-case | 1 | log(n) | n | n*log(n) | n^2 | n^3 | 2^n | n! |
|---|---|---|---|---|---|---|---|---|---|
| 3,00 | 9 | 64 | 54,98278112 | 36 | 18,02098012 | 0 | 324 | 1 | 9 |
| 4,00 | 9 | 64 | 49 | 25 | 1 | 49 | 3025 | 49 | 225 |
| 5,00 | 11 | 100 | 75,30893199 | 36 | 0,3716615081 | 196 | 12996 | 441 | 11881 |
| 6,00 | 13 | 144 | 108,4730061 | 49 | 6,298970572 | 529 | 41209 | 2601 | 499849 |
| 7,00 | 15 | 196 | 148,660594 | 64 | 21,63630763 | 1156 | 107584 | 12769 | 25250625 |
| 8,00 | 16 | 225 | 169 | 64 | 64 | 2304 | 246016 | 57600 | 1624412416 |
| 9,00 | 16 | 256 | 164,6108245 | 49 | 156,9839853 | 4225 | 508369 | 246016 | 131670282496 |
| 10,00 | 19 | 361 | 245,8019387 | 81 | 202,1879507 | 6561 | 962361 | 1010025 | 13168051545961 |
| SUM | | 1410 | 1015,838076 | 404 | 470,4998558 | 15020 | 1881884 | 1329502 | 13301372003462 |

*Table 9: Branch and bound Worst case approximate order of growth*

According to the above tables, branch and bound algorithm in the best and average cases follows order of growth of linear function. However, in the worst case it showed (nlogn) order of growth.

Caverage (n) = Cbest(n) ∈ Θ (n).

Cworst(n) ∈ Θ (n logn).

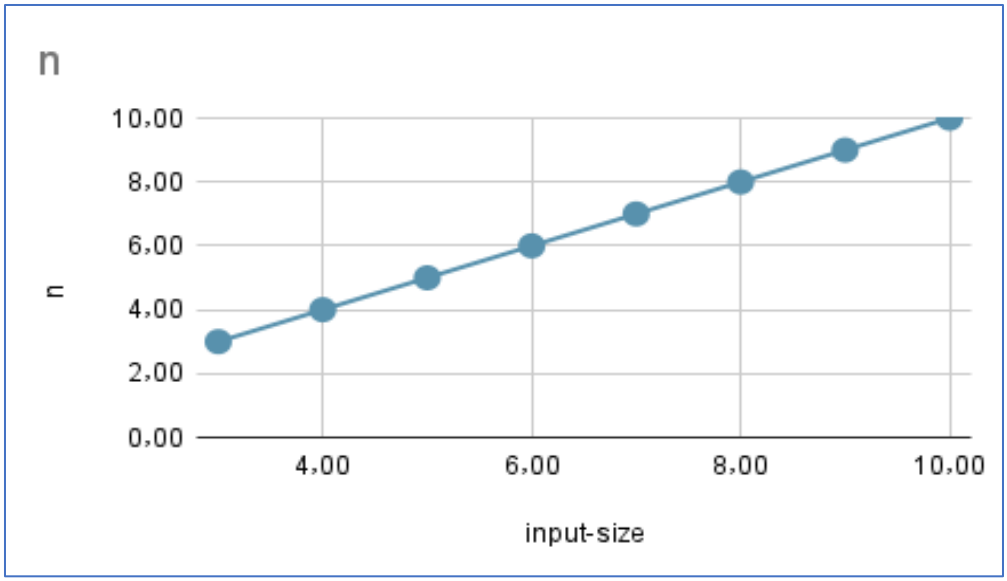



*Figure 10: Θ (n) Graph*

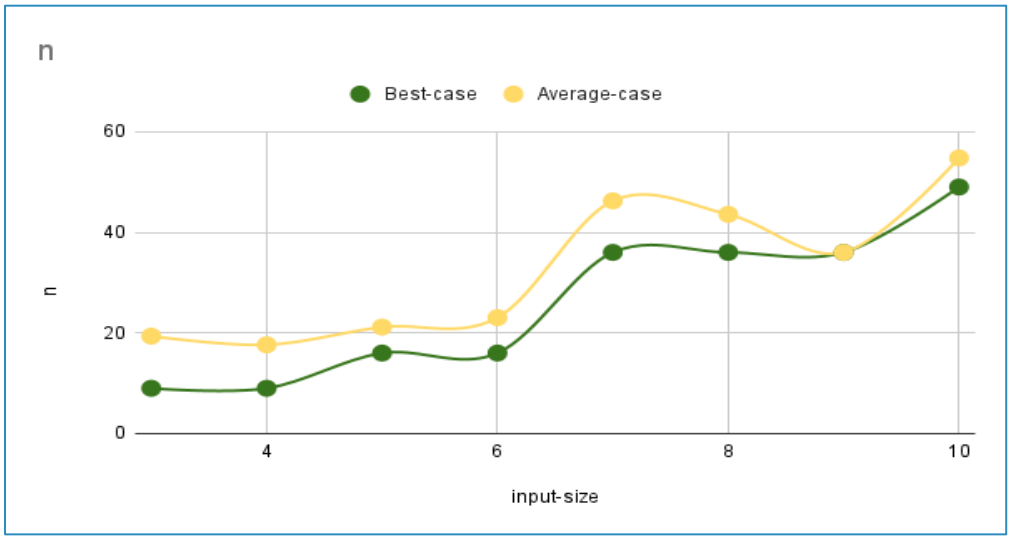



*Figure 11: Θ (n) Branch and bound in Best case & Average case Graph*

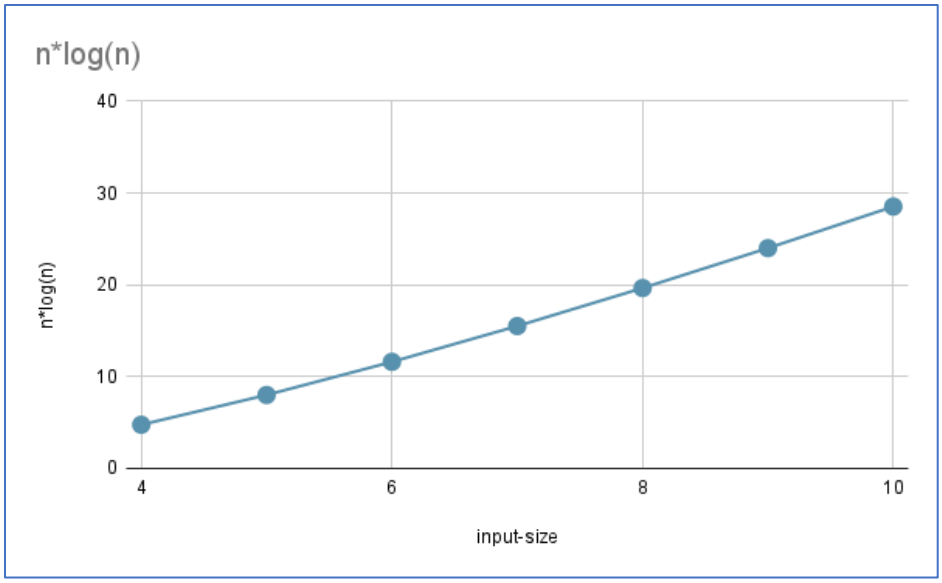



*Figure 12: Θ (n logn) Graph.*

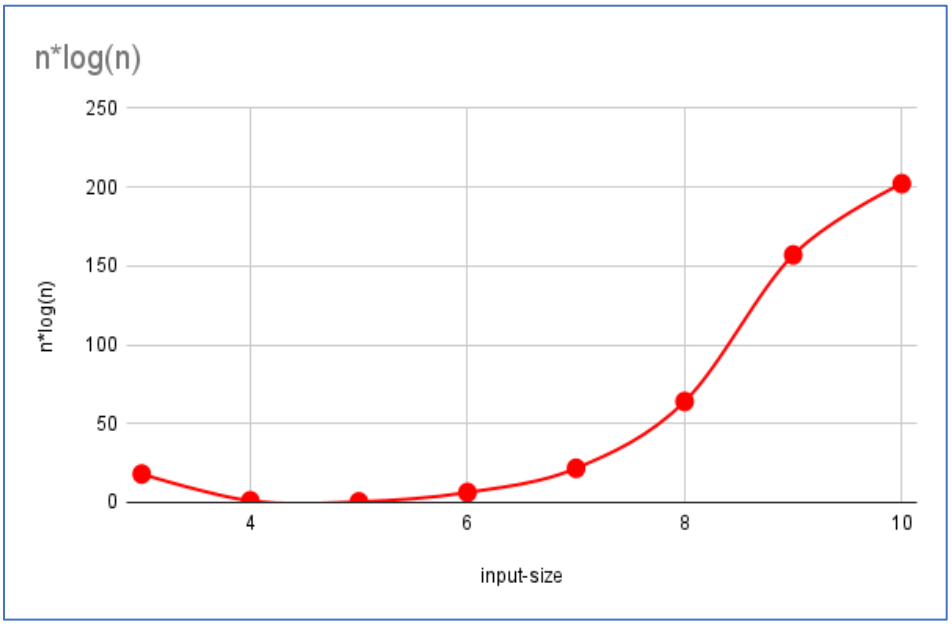



*Figure 13: Θ (n logn) Branch and bound in Worst case Graph.*

## 3.9. Comparing the result with the theoretical analysis

### 3.9.1. Brute force algorithm:

The brute force algorithm of TSP should follow (n-1)! order of growth because its main aim is to generate all possible paths (all permutations). However, in our case, it obtained exponential order of growth (2^n) which was affected by the small input size and the number of trials. In addition, we noticed that the time varies from one device to another, due to the different performance and speed of each device. If we want to try this algorithm for larger inputs, it would require high-performance computers to test. So, we concluded that this algorithm is suitable for small inputs and not effective for large problems.

### 3.9.2. Branch and bound algorithm:

The worst-case complexity of branch and bound should remain the same as the brute force. However, in practice it performs better depending on the different instances of the TSP distances matrix. The performance also depends on the other functions used in pruning the nodes.

Due to the small number of nodes in our sample, the order of growth showed us that it follows a linear efficiency class in the best and average cases, while in the worst case it follows (n logn).

## 4. Conclusion

To sum up, after the experiment's execution, analyzed it empirically, selection of the proper efficiency metric, and implementing the two algorithms for finding the shortest path, we conclude that the branch and bound algorithm is more efficient than brute force.

The efficiency class of the branch and bound was θ(n) linear in the best and average case, yet in the worst case was (nlogn). Whereas the brute force algorithm followed an exponential order of growth in all cases according to the empirical analysis that we obtained.

## 5. References


- *Branch and Bound Algorithm*. (n.d.). GeeksforGeeks. https://www.geeksforgeeks.org/branch-and-bound-algorithm/
- *Traveling Salesman Problem (TSP) Implementation*. (2017, November 11). GeeksforGeeks. https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/
- *Solving The Travelling Salesman Problem For Deliveries | Routific*. (n.d.). Blog.routific.com. https://blog.routific.com/blog/travelling-salesman-problem
- *AlgoraithmTSP*. (n.d.). Google Docs. Retrieved February 11, 2023, from https://docs.google.com/spreadsheets/d/1IiFQ8sCWBrxkntzXBRNeWCf-mUBQevabfcRRdtGV87M/edit?usp=sharing

# 6. Appendix A

- Brute force code

```java
/*
================================
CPCS-223-Project Coding
================================
Tsp Using Brute Force Algoraithm
================================
The code is from the next web
Traveling Salesman Problem (TSP) Implementation. (2017, November 11). GeeksforGeeks.
https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/
================================
--------Group Member---------
--------Section B2A ---------
(Leader) Ebtihaj Alnaqeeb 2011859
(Member) Nsreen Hujjatullah Asadullah 2010271
(Member) Hanoof mohammed 1909632
(Member) Layan Algarni 1907936
 */
package tspusingbruteforcealgoraithm;

import java.util.ArrayList;
import java.util.Random;
import java.util.Scanner;

public class TspUsingBruteForceAlgoraithm {

    /**
     * @param args the command line arguments
     */
    //============================================================================
    public static void main(String[] args) {
        int[][] arrayDistances = genratedistancesCites(n, 41, 50);
        int s = 0;// start node
        long startTime, endTime;
        startTime = System.nanoTime(); // start time
        System.out.println("\nThe optimal cost is: " + travllingSalesmanProblem(arrayDistances, s));
        endTime = System.nanoTime(); // end time
        printADJ(arrayDistances); // call method print Adjacency matrix

        System.out.println("-----------------------------------");
        System.out.println("Time taken :" + ((endTime - startTime) / 1000000) + " Millisecond");// print the time
        System.out.println("-----------------------------------");
    }
//============================================================================
    //method to genrate cites distances randomly

    public static int[][] genratedistancesCites(int n, int min, int max) {
        int distances[][] = new int[n][n];
        Random rand = new Random(100);//genrate randomly
        //for loop to genrate randomly distances to store distances in 2d array
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < i; j++) {
                //genrate cites distances randomly
                int value = Math.abs((min + rand.nextInt((max - min) + 1)));
                distances[i][j] = value;
                distances[j][i] = value;
            }
        }
        return distances;
    }
//============================================================================
    // Define a scanner to ask the user to enter the number of cities
    static Scanner input = new Scanner(System.in);

    static {
        System.out.print("Enter the number of cities :");// Ask the user to enter the number of cities
    }
    static int n = input.nextInt(); // Store the number of cities we have requested from the user in a variable n
    static int V = n;
//============================================================================
```

```
// implementation of traveling
// Salesman Problem

static int travllingSalesmanProblem(int graph[][], int s) {
    // store all vertex apart
    // from source vertex
    ArrayList<Integer> vertex = new ArrayList<Integer>();
    for (int i = 0; i < V; i++) {
        if (i != s) {
            vertex.add(i);
        }
    }
    // store minimum weight
    int min_path = Integer.MAX_VALUE;
    // array to save the value of optimal path
    int[] g = new int[V];
    int[] g1 = new int[V];
    int[] d = new int[V];
    int[] d1 = new int[V];
    do {
        // store current Path weight(cost)
        int current_pathweight = 0;
        // compute current path weight
        int k = s; //start point
        System.out.print(s + "->");
        for (int i = 0; i < vertex.size(); i++) { //use the 1d array to cal all possible paths distances
            //here we calculate the path between start node and next point
            current_pathweight += graph[k][vertex.get(i)];//use 2d array to find the distance btw the start and other cities
            k = vertex.get(i);
            System.out.print(k + "->");//this print the node
            g[i]= k;// save value of k to array
            d[i]=k; // save value of k to array

        }
        //here we calculate the path between start node and final point
        current_pathweight += graph[k][s];
       //here in the start we print s which we should end with
       System.out.println(s + "  The cost for this path = " + current_pathweight);
       //save the optimal path nodes
       if (current_pathweight < min_path) {
           // update minimum
           min_path = Math.min(min_path, current_pathweight);
           //save the optimal path nodes
           System.arraycopy(g, s, g1, s, vertex.size() - s);
       }else if (current_pathweight == min_path){
           System.arraycopy(d, s, d1, s, vertex.size() - s);
       }

    } while (findNextPermutation(vertex));

    // print the optimal path
    System.out.println("-----------------------------------");
    System.out.println("The optimal path : ");
    System.out.print(s + "->");
    for (int i = s; i < vertex.size(); i++)//use the 1d array to cal all possible paths distances
    {
        System.out.print(g1[i] + "->");
    }
    System.out.print(s);
    System.out.println("");
    System.out.print(s + "->");
    for (int i = s; i < vertex.size(); i++)//use the 1d array to cal all possible paths distances
    {
        System.out.print(d1[i] + "->");
    }
    System.out.print(s);

    return min_path; // return the cost of min_path
}
```

```java
//============================================================================
// Function to swap the data
// present in the left and right indices

    public static ArrayList<Integer>
        swap(ArrayList<Integer> data, int left, int right) {
        // Swap the data
        int temp = data.get(left);
        data.set(left, data.get(right));
        data.set(right, temp);
        // Return the updated array
        return data;
    }
//============================================================================
// Function to reverse the sub-array
// starting from left to the right
// both inclusive

    public static ArrayList<Integer>
        reverse(ArrayList<Integer> data, int left, int right) {
        // Reverse the sub-array
        while (left < right) {
            int temp = data.get(left);
            data.set(left++, data.get(right));
            data.set(right--, temp);
        }
        // Return the updated array
        return data;
    }
//============================================================================
// Function to find the next permutation
// of the given integer array
    public static boolean findNextPermutation(ArrayList<Integer> data) {
        // If the given dataset is empty
        // or contains only one element
        // next_permutation is not possible
        if (data.size() <= 1) {
            return false;
        }
        int last = data.size() - 2;
        // find the longest non-increasing suffix and find the pivot
        while (last >= 0) {
            if (data.get(last) < data.get(last + 1)) {
                break;
            }
            last--;
        }
        // If there is no increasing pair there is no higher order permutation
        if (last < 0) {
            return false;
        }
        int nextGreater = data.size() - 1;
        // Find the rightmost successor to the pivot
        for (int i = data.size() - 1; i > last; i--) {
            if (data.get(i) > data.get(last)) {
                nextGreater = i;
                break;
            }
        }
        // Swap the successor and the pivot
        data = swap(data, nextGreater, last);
        // Reverse the suffix
        data = reverse(data, last + 1, data.size() - 1);
        // Return true as the next_permutation is done
        return true;
    }
```

```java
//================================================================================
    // method to print Adjacency matrix

    public static void printADJ(int[][] adj) {
        System.out.println("-----------------------------------");
        System.out.println("Adjacency matrix ");
        System.out.println("-----------------------------------");
        for (int i = 0; i < adj.length; i++) {
            for (int j = 0; j < adj[i].length; j++) {
                System.out.print(adj[i][j] + "    ");
            }
            System.out.println();
        }
    }

}
```

# 7. Appendix B

- Branch and bound code

```
/*
==============================
CPCS-223-Project Coding
==============================
Tsp Using Branch And Bound
==============================
The code is from the next web
Traveling Salesman Problem using Branch And Bound. (2016, October 13). GeeksforGeeks.
https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/
==============================
--------Group Member---------
--------Section B2A ---------
(Leader) Ebtihaj Alnaqeeb 2011859
(Member) Nsreen Hujjatullah Asadullah 2010271
(Member) Hanoof mohammed 1909632
(Member) Layan Algarni 1907936
 */
package tspusingbranchandbound;

import java.util.Arrays;
import java.util.Random;
import java.util.Scanner;

public class TspUsingBranchAndBound {

    /**
     * @param args the command line arguments
     */
    //===========================================================================
    public static void main(String[] args) {
        int[][] adj = genratedistancesCites(c, 10, 100);
        System.out.println("     All paths with they costs     ");
        System.out.println("-----------------------------------");
        long startTime, endTime;
        startTime = System.nanoTime(); // start time
        TSP(adj);
        endTime = System.nanoTime(); // end time

        System.out.println("-----------------------------------");
        System.out.printf("Minimum cost : %d\n", final_res); // print the minimum cost
        System.out.printf("Path Taken : "); // print the path taken
        for (int i = 0; i <= N; i++) {
            System.out.printf("%d ", final_path[i]);
        }
        endTime = System.nanoTime(); // end time
        printADJ(adj);// call method print Adjacency matrix
        System.out.println("-----------------------------------");
        System.out.println("Time taken :" + ((endTime - startTime) / 1000000) + " Millisecond");// print the time
        System.out.println("-----------------------------------");
    }
//===========================================================================
    //method to genrate cites distances randomly

    public static int[][] genratedistancesCites(int n, int min, int max) {
        int distances[][] = new int[n][n];
        Random rand = new Random(100); //genrate randomly
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < i; j++) {
                //genrate cites distances randomly
                int value = Math.abs((min + rand.nextInt((max - min) + 1)));
                distances[i][j] = value;
                distances[j][i] = value;
            }
        }
        return distances;
    }
//===========================================================================
    // Define a scanner to ask the user to enter the number of cities
    static Scanner input = new Scanner(System.in);

    static {
        System.out.print("Enter the number of cities :");// Ask the user to enter the number of cities
    }
    static int c = input.nextInt(); // Store the number of cities we have requested from the user in a variable c
```

```
    static {
        System.out.println("-----------------------------------");
    }
    static int N = c;
    // final_path[] stores the final solution ie, the
    // path of the salesman.
    static int final_path[] = new int[N + 1];
    // visited[] keeps track of the already visited nodes
    // in a particular path
    static boolean visited[] = new boolean[N];
    // Stores the final minimum weight of shortest tour.
    static int final_res = Integer.MAX_VALUE;
//================================================================================
    // Function to copy temporary solution to
    // the final solution

    static void copyToFinal(int curr_path[]) {
        for (int i = 0; i < N; i++) {
            final_path[i] = curr_path[i];
        }
        final_path[N] = curr_path[0];
    }
//================================================================================
    // Function to find the minimum edge cost
    // having an end at the vertex i

    static int firstMin(int adj[][], int i) {
        int min = Integer.MAX_VALUE;
        for (int k = 0; k < N; k++) {
            if (adj[i][k] < min && i != k) {
                min = adj[i][k];
            }
        }
        return min;
    }
//================================================================================
    // function to find the second minimum edge cost
    // having an end at the vertex i

    public static int secondMin(int adj[][], int i) {
        int first = Integer.MAX_VALUE, second = Integer.MAX_VALUE;
        for (int j = 0; j < N; j++) {
            if (i == j) {
                continue;
            }
            if (adj[i][j] <= first) {
                second = first;
                first = adj[i][j];
            } else if (adj[i][j] <= second && adj[i][j] != first) {
                second = adj[i][j];
            }
        }
        return second;
    }
//================================================================================
    // function that takes as arguments:
    // curr_bound -> lower bound of the root node
    // curr_weight-> stores the weight of the path so far
    // level-> current level while moving in the search
    //          space tree
    // curr_path[] -> where the solution is being stored which
    //             would later be copied to final_path[]

    public static void TSPRec(int adj[][], int curr_bound, int curr_weight,
            int level, int curr_path[]) {
        // base case is when we have reached level N which means we have covered all the nodes once
        if (level == N) {
            // check if there is an edge from last vertex in path back to the first vertex
            if (adj[curr_path[level - 1]][curr_path[0]] != 0) {
                // curr_res has the total weight of the solution we got
                int curr_res = curr_weight + adj[curr_path[level - 1]][curr_path[0]];
                // Update final result and final path if current result is better.
```

```
            if (curr_res < final_res) {
                copyToFinal(curr_path);
                for (int i = 0; i <= N; i++) {
                    // print the paths
                    System.out.printf("%d ", curr_path[i]);
                    curr_path[N] = curr_path[0];
                }
                System.out.println("");
                final_res = curr_res;
                System.out.println("Cost this path = " + curr_res);
            }
        }
        return;
    }
    // for any other level iterate for all vertices to build the search space tree recursively
    for (int i = 0; i < N; i++) {
        // Consider next vertex if it is not same (diagonal entry in adjacency matrix and not visited already)
        if (adj[curr_path[level - 1]][i] != 0 && visited[i] == false) {
            int temp = curr_bound;
            curr_weight += adj[curr_path[level - 1]][i];
            // different computation of curr_bound for level 2 from the other levels
            if (level == 1) {
                curr_bound -= ((firstMin(adj, curr_path[level - 1]) + firstMin(adj, i)) / 2);
            } else {
                curr_bound -= ((secondMin(adj, curr_path[level - 1]) + firstMin(adj, i)) / 2);
            }
            // curr_bound + curr_weight is the actual lower bound for the node that we have arrived on
            // If current lower bound < final_res, we need to explore the node further
            if (curr_bound + curr_weight < final_res) {
                curr_path[level] = i;
                visited[i] = true;
                // call TSPRec for the next level
                TSPRec(adj, curr_bound, curr_weight, level + 1, curr_path);
            }
            // Else we have to prune the node by resetting
            // all changes to curr_weight and curr_bound
            curr_weight -= adj[curr_path[level - 1]][i];
            curr_bound = temp;
            // Also reset the visited array
            Arrays.fill(visited, false);
            for (int j = 0; j <= level - 1; j++) {
                visited[curr_path[j]] = true;
            }
        }
    }
}
//=============================================================================
    // This function sets up final_path[]

    public static void TSP(int adj[][]) {
        int curr_path[] = new int[N + 1];
        // Calculate initial lower bound for the root node
        // using the formula 1/2 * (sum of first min + second min) for all edges.
        // Also initialize the curr_path and visited array
        int curr_bound = 0;
        Arrays.fill(curr_path, -1);
        Arrays.fill(visited, false);
        // Compute initial bound
        for (int i = 0; i < N; i++) {
            curr_bound += (firstMin(adj, i) + secondMin(adj, i));
        }
        // Rounding off the lower bound to an integer
        curr_bound = (curr_bound == 1) ? curr_bound / 2 + 1 : curr_bound / 2;
        // We start at vertex 1 so the first vertex
        // in curr_path[] is 0
        visited[0] = true;
        curr_path[0] = 0;
        // Call to TSPRec for curr_weight equal to
        // 0 and level 1
        TSPRec(adj, curr_bound, 0, 1, curr_path);
    }
```

```java
//============================================================================
    // method to print Adjacency matrix

    public static void printADJ(int[][] adj) {
        System.out.println("\n-----------------------------------");
        System.out.println("Adjacency matrix ");
        System.out.println("-----------------------------------");
        for (int i = 0; i < adj.length; i++) {
            for (int j = 0; j < adj[i].length; j++) {
                System.out.print(adj[i][j] + "    ");
            }
            System.out.println();
        }
    }
}
```