**TSP TIMING PROJECT**

NAME: Akshay Swaminathan

CWID: 10832697

1. **DETAILS OF THE TWO IMPLEMENTATIONS:**

Two different approaches are taken in both the implementations.

**1.1 Nearest Neighbor Approach:**

In this approach, there is a point class which is being defined and with that class different functions are called and executed. In nearest neighbor approach a point P0 is selected, we try to calculate the distance from this point to all the other points. The unvisited point which has the shortest distance from the point that we are already in is the point that the program should traverse. Nearest neighbor heuristic takes in list of coordinates as input and returns the Coordinates which it traverses to in the order it has traversed from the starting point, Total distance travelled and total time take to traverse all these points and return to the starting point.

**CODE:**

 for (int i=0;i<count;i++)

    {

        fin>>x;

        fin>>y;

        coordinate.push\_back(point(x,y));

    }

    clock\_t t1,t2;

    t1=clock();

    while(coordinate.size()>1)

    {

        minimum=999999999;

        start=coordinate[0];

        for(int i=1;i<coordinate.size();i++)

        {

            distance = sqrt((pow((start.getx() - coordinate[i].getx()),2)) + (pow((start.gety() - coordinate[i].gety()),2)));

            if (distance < minimum)

            {

                minimum = distance;

                minimum\_point = coordinate[i];

                minimum\_coord = i;

            }

            else

            {

                continue;

            }

        }

        std::swap(coordinate[0], coordinate[minimum\_coord]);

        route.push\_back(start);

        coordinate.erase(coordinate.begin() + minimum\_coord);

    }

    route.push\_back(coordinate[0]);

    for (int i = 1; i < route.size() ; i++)

    {

        score += sqrt((pow((route[i].getx() - route[i-1].getx()),2)) + (pow((route[i].gety() - route[i-1].gety()),2)));

    }

    score += sqrt((pow((route[0].getx() - route[route.size() - 1].getx()), 2)) +

                  (pow((route[0].gety() - route[route.size() - 1].gety()), 2)));

**1.2 EXHAUSTIVE SEARCH APPROACH:**

Taken a different approach by creating a header file and class with the **point** which is used for performing various tasks. Exhaustive algorithm computes all the permutations with the points that are given as input and gives us the best possible solution with the shortest route. As the name suggests the algorithm takes more time to compute compared to the N-N approach as it is computing all the possibilities. In this I have used the generative permutation to calculate all the permutations for a given set of points.

**CODE:**

**for** (**int** i = 0; i<count; i++)  
{  
  
 fin >> x;  
 fin >> y;  
 coordinate.push\_back(point(x, y));  
}  
  
  
  
min = 9999999.0;  
**if** (count > 2) {  
 **while** (next\_permutation(coordinate.begin() + 1, coordinate.end()))  
 {  
  
 distance = CalcDistance(coordinate[0].getx(), coordinate[0].gety(), coordinate[coordinate.size() - 1].getx(), coordinate[coordinate.size() - 1].gety());  
 **for** (**int** i = 1; i < coordinate.size(); i++)  
 {  
  
 distance += CalcDistance(coordinate[i].getx(), coordinate[i].gety(), coordinate[i - 1].getx(), coordinate[i - 1].gety());  
  
 }  
 **if** (distance < min)  
 {  
  
 minimum = distance;  
 route = coordinate;  
  
 }  
  
  
 score.push\_back(distance);  
  
  
 }  
}  
**else** {  
  
 route = coordinate;  
 minimum = 2 \* CalcDistance(coordinate[0].getx(), coordinate[0].gety(), coordinate[1].getx(), coordinate[1].gety());  
  
}  
  
  
**for** (**int** i = 0; i < route.size(); i++)  
{  
 std::cout << **"Point "** <<i+1<< std::endl;  
 std::cout << route[i].getx()<<**" "**<<route[i].gety() << std:;endl;  
}

1. **WORST CASE COMPLEXITY**

The worst-case complexities is like taking the worst case into consideration and running the code. We can represent that asymptotically θ(n2)

* 1. **Nearest Neighbor Approach:**

In the nearest neighbor approach, the portion which calculates the distance is constant time and which is denoted as θ(1). The below code:

if (distance < minimum)

            {

                minimum = distance;

                minimum\_point = coordinate[i];

                minimum\_coord = i;

            }

            else

            {

                continue;

            }

        }

        std::swap(coordinate[0], coordinate[minimum\_coord]);

        route.push\_back(start);

        coordinate.erase(coordinate.begin() + minimum\_coord);

will maximum take θ(n) time to execute. The maximum complex portion is the where it finds the nearest neighbor.

for(int i=1;i<coordinate.size();i++)

        {

            distance = sqrt((pow((start.getx() - coordinate[i].getx()),2)) + (pow((start.gety() - coordinate[i].gety()),2)));

}

The “for” loop executes until all the points are done and the distance between the point and the where it is right now and the unvisited points are calculated until it reaches the starting point. So this may take N summation iterations to complete which can be given by n(n+1)/2 which is denoted as θ(n2), the worst case complexity.

* 1. **Exhaustive Algorithm Approach:**

So in this exhaustive algorithm approach is basically a combination of computing all the permutations and then calculating the distance and each and every time for this. The worst case scenario for the exhaustive algorithm is θ(n\*n!).

As defined above

**if** (distance < min)  
 {  
  
 min = distance;  
 route = coord;  
  
 }  
  
  
 score.push\_back(distance);  
  
  
 }  
}  
**else** {  
  
 route = coord;  
 min = 2 \* CalcDistance(coord[0].getx(), coord[0].gety(), coord[1].getx(), coord[1].gety());  
  
}

This above code is a constant time which gives us θ(1).The code below to calculate the permutation will take utmost n! ways, if n is the number of coordinates. So the worst case complexity for this would be θ(n!).

**while** (next\_permutation(coord.begin() + 1, coord.end()))

As the N-N approach it calculates the distance after each and every permutation until it has returned to the staring point which take θ(n) time to execute. So, combining all these time we can determine the worst case complexity of the Exhaustive approach as θ(n\*n!).

1. **INPUT GENERATION/ EXPERIMENTAL TESTING**

**3.1 Nearest Neighbor**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| VALUE OF N | FIRST EXECUTION  (micro seconds) | SECOND EXECUTION micro seconds) | THIRD EXECUTION micro seconds) | AVERAGE  (micro seconds) |
| 1000 | 5415 | 5476 | 4987 | 5292.66 |
| 3400 | 21403 | 20987 | 21765 | 21385 |
| 4600 | 46017 | 45973 | 46641 | 46210.33 |
| 7500 | 125974 | 125412 | 130308 | 127231.33 |

**3.2 Exhaustive:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| VALUE OF N | FIRST EXECUTION  (micro seconds) | SECOND EXECUTION micro seconds) | THIRD EXECUTION micro seconds) | AVERAGE  (micro seconds) |
| 2 | 4 | 4 | 4 | 4 |
| 4 | 16 | 22 | 15 | 17.667 |
| 6 | 434 | 347 | 347 | 376 |
| 8 | 34782 | 29143 | 28208 | 30711 |

1. **Match Theory to practice:**
   1. **Nearest Neighbor**

The results we have obtained above in the table for the nearest neighbor is within the worst case complexity that we have already determined θ(n2). The values that we obtain is below or within the range of this

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| VALUE OF N | FIRST EXECUTION  (micro seconds) | SECOND EXECUTION micro seconds) | THIRD EXECUTION micro seconds) | AVERAGE  (micro seconds) | θ(n2) |
| 1000 | 5415 | 5476 | 4987 | 5292.66 | 100000 |
| 3400 | 21403 | 20987 | 21765 | 21385 | 1156000 |
| 4600 | 46017 | 45973 | 46641 | 46210.33 | 2116000 |
| 7500 | 125974 | 125412 | 130308 | 127231.33 | 5625000 |

**4.5 Exhaustive**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| VALUE OF N | FIRST EXECUTION  (micro seconds) | SECOND EXECUTION micro seconds) | THIRD EXECUTION micro seconds) | AVERAGE  (micro seconds) | θ(n\*n!) |
| 2 | 4 | 4 | 4 | 4 | 4 |
| 4 | 16 | 22 | 15 | 17.667 | 96 |
| 6 | 434 | 347 | 347 | 376 | 4320 |
| 8 | 34782 | 29143 | 28208 | 30711 | 322560 |

So from both these approaches we have listed the worst time complexity theoretically which determines that the average result or any result we obtain will be either equal to or below this value.

**5.DEMO**