Micah Geertson & Justin Stewart
CPSC 335
Project 1
04/24/2016

Programming Assignment #3

Table of Contents

Exhaustive Optimization Pseudocode/Analysis 2

Nearest Neighbor Pseudocode/Analysis 4

Exhaustive Optimization C++ Code 6

Nearest Neighbor C++ Code 10

Sample Output 14

```
def print_perm(int n, Array A, int sizeA, Array P, Array bestSet, int bestDist)
dist = 0.0
j = 0
if n == 1 do
    for i = 0 to (sizeA – 1) do
        dist += abs(P[A[i]].x – P[A[i+1]].x) + abs(P[A[i]].y – P[A[i+1]].y)

    dist += abs(P[A[0]].x – P[A[sizeA - 1]].x) + abs(P[A[0]].y – P[A[sizeA - 1]].y)
    if dist < bestDist do
        bestDist = dist
        for i = 0 to sizeA do
            bestSet[j] = A[i]
            j++
else
    for i = 0 to (n – 1) do
            call print_perm(n-1, A, sizeA, bestSet, bestDist)
            if n % 2 == 0
                int temp = A[i];
                A[i] = A[n-1];
                A[n-1] = temp;
            else
                int temp = A[0];
                A[0] = A[n-1];
                A[n-1] = temp;

    call print_perm(n – 1, A, sizeA, P, bestSet, bestDist)
```

$1\ Step$

$1\ Step$

$\boldsymbol{No. of\ Steps\ A}$:

$n = sizeA$

$\frac{(n-1-0)}{1} + 1 = n$

$n * 1 = n$

$1\ Step$

= n + 1

$\boldsymbol{No. of\ Steps\ B}$:

$1\ Step$

$n = sizeA$

$\frac{(n-0)}{1} + 1 = (n+1)$

$(n+1) * 2 = 2n + 2$

$1 + \max(2n+3, 0)$

$= 2n + 4 + 1$

$= 2n + 5$

$A + B = 3n + 6$

$\boldsymbol{No. of\ Steps\ C}$:

$1 + \max(3, 3) = 4$

$\frac{n-1-0}{1} + 1 = n$

$n * 4 = 4n$

$\underline{\boldsymbol{n = 1}}$

$T(1) = 3n + 6$

$\underline{\boldsymbol{n > 1}}$

$T(n) = T(n-1) + 3$

$T(n-1) = T(n-2) + 3$

$T(n) = T(n-k) + 3(k)$

$k = n - 1$

$T(n) = T(1) + 3(n-1)$

$T(n) = 3n + 6 + 3n - 3$

$T(n) = 6n + 3$

$T(n) = (6n+3) * 4n$

$T(n) = 24n^2 + 12n$

```
def farthest(int n,  Array P)

max_dist = 0

for i = 0 to (n-1) do

    for j = 0 to n do

            dist = abs(P[i].x – P[j].x) + abs(P[i].y – P[j].y)

            if max_dist < dist

                max_dist = dist

return max_dist
```

Farthest

$1\ Step$

$\boldsymbol{No.of\ Steps\ A}$:

$$\frac{(n-0)}{1}+1=n+1$$

$$(n+1)*(1+\max(1,0)$$

$$=2n+2$$

$\boldsymbol{No.of\ Steps\ B}$:

$$\frac{(n-1-0)}{1}+1=n$$

$$(n)*(2n+2)$$

$$=2n^2+2n$$

$$=2n^2+2n+2$$

```
def main_ex(int n, Array A, int sizeA, Array P, Array bestSet, int bestDist)

Dist = call farthest(n, P)

bestDist = n*Dist


A = new Array [n]

for i = 0 to n do

        A[i] = i


call print_perm(n, A, n, P, bestSet, bestDist)
```

Main

$2n^2+2n+2$

$1\ Step$

$1\ Step$

$$\frac{(n-0)}{1}+1=n+1$$

$24n^2+12n$

$$24n^2+12n+2n^2+2n+2+2+n+1$$

$26n^2+15n+5$

$$\lim_{n\to\infty}\frac{26n^2+15n+5}{n^2}$$

$$\lim_{n\to\infty}\frac{26n^2}{n^2}=26$$

$26\ \geq 0$ and constant.

$$26n^2+15n+5\in\ O(n^2)$$

```
def nearest(int n, Array P, int A , Array Visited)
nearest = 0
dist = 100.0
temp = 0.0
for i = 0 to n do
    if NOT Visited[i] do
        temp = abs(P[i].x – P[A].x) + abs(P[i].y – P[A].y)
        if temp < dist do
            dist = temp
            nearest = i
return nearest
```

$1\ Step$

$1\ Step$

$1\ Step$

$\boldsymbol{No.of\ Steps\ A}$:

$1 + \max(2, 0) = 3$

$1\ Step$

$\boldsymbol{No.of\ Steps\ B}$:

$1 + \max(4, 0) = 5$

$\boldsymbol{No.of\ Steps\ C}$:

$\frac{n-0}{1} + 1 = n + 1$

$(n + 1) * 5 = 5n + 5$

$1\ Step$

$5n + 5 + 4 = 5n + 9$

```
def farthest_point(int n, Array P)
max_dist = 0.0
for i = 0 to n-1 do
    for j = 0 to n do
        dist = abs(P[i].x – P[j].x) + abs(P[i].y – P[j].y)
        if max_dist < dist do
            max_dist = dist
            point = i
return point
```

$1\ Step$

$\boldsymbol{No.of\ Steps\ A}$:

$1 + \max(2, 0) = 3$

$1\ Step$

$\boldsymbol{No.of\ Steps\ B}$:

$\frac{n-0}{1} + 1 = n + 1$

$(n + 1) * 4 = 4n + 4$

$\boldsymbol{No.of\ Steps\ C}$:

$\frac{n-1-0}{1} + 1 = n$

$(n) * (4n + 4)$

$= 4n^2 + 4n$

$1\ Step$

$= 4n^2 + 4n + 1$

```
def main_ex(int n, Array P, int A , Array Visited)
M = new int Array [n]
for i = 0 to n do
    M[i] = i

Visited = new Bool Array [n]
for i = 0 to n do
    Visited[i] = true
A = farthest_point(n, P)
i = 0
M[i] = A
for i = 1 to n do
    B = nearest(n, P, A, Visited)
    A = B
    M[I] = A;
    Visited[A] = true
dist = 0
for i = 0 to n-1 do
    dist += abs(P[M[i]].x – P[M[i+1]].x) + abs(P[M[i]].y – P[M[i+1]].y)
dist += abs(P[M[0]].x – P[M[n-1]].x) + abs(P[M[0]].y – P[M[n-1]].y)
```

$1\ Step$

$$\frac{n-0}{1}+1=n+1$$

$1\ Step$

$$\frac{n-0}{1}+1=n+1$$

$$4n^2+4n+1$$

$1\ Step$

$1\ Step$

$$\frac{n-1}{1}+1=n$$

$$n*(5n+9+3)$$

$$(5n^2+12n)$$

$1\ Step$

$$\frac{n-1-0}{1}+1=n*1$$
$$=n$$

$1\ Step$

$$1+(n+1)+1+(n+1)$$

$$=2n+4$$

$$2n+4+4n^2+4n+1$$

$$=\ 4n^2+6n+5$$

$$4n^2+6n+5+4+(5n^2+12n)$$

$$=\ 9n^2+18n+9+n$$

$$=\ 9n^2+19n+9$$

$$\lim_{n\to\infty}\frac{9n^2+19n+9}{n^2}$$

$$\lim_{n\to\infty}\frac{9n^2}{n^2}=9$$

$9\ \geq 0$ and constant.

$$9n^2+19n+9\in\ O(n^2)$$

```
// Assignment 3: Euclidean traveling salesperson problem: exhaustive optimization algorithm
/****************************************

* Name: Micah Geertson & Justin Stewart *
* CPSC 335-01 13115                     *
* Date: 04/20/2016                      *
*****************************************/
// A special case of the classical traveling salesman problem (TSP) where the input is a
Euclidean graph
// INPUT: a positive integer n and a list P of n distinct points representing vertices of a
Euclidean graph
// OUTPUT: a list of n points from P representing a Hamiltonian cycle of minimum total weight
for the graph.

#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <string>
#include <chrono>
#include <cmath>
using namespace std;

struct point2D {
    float x; // x coordinate
    float y; // y coordinate
};

void print_cycle(int, point2D*, int*);
// function to print a cyclic sequence of 2D points in 2D plane, given the
// number of elements and the actual sequence stored as an array of 2D points

float farthest(int, point2D*);
// function to calculate the furthest distance between any two 2D points

void print_perm(int, int *, int, point2D*, int *, float &);
// function to generate the permutation of indices of the list of points

int main() {
    point2D *P;
    int *bestSet, *A;
    int i, n;
    float bestDist, Dist;

    // display the header
    cout << endl << "CPSC 335-x - Programming Assignment #3" << endl;
    cout << "Euclidean traveling salesperson problem: exhaustive optimization algorithm" << endl;
    cout << "Enter the number of vertices (>2) " << endl;

    // read the number of elements
    cin >> n;

    // if less than 3 vertices then terminate the program
    if (n <3)
        return 0;
```

```
// allocate space for the sequence of 2D points
P = new point2D[n];

// read the sequence of distinct points
cout << "Enter the points; make sure that they are distinct" << endl;
for( i=0; i < n; i++) {
    cout << "x=";
    cin >> P[i].x;
    cout << "y=";
    cin >> P[i].y;
}

// allocate space for the best set representing the indices of the points
bestSet = new int[n];
// set the best set to be the list of indices, starting at 0
for(i=0; i<n; i++)
    bestSet[i]=i;

// Start the chronograph to time the execution of the algorithm
auto start = chrono::high_resolution_clock::now();

// calculate the farthest pair of vertices
Dist = farthest(n,P);
bestDist = n*Dist;
// populate the starting array for the permutation algorithm
A = new int[n];
// populate the array A with the values in the range 0 .. n-1
for(i=0; i<n; i++)
    A[i] = i;

// calculate the Hamiltonian cycle of minimum weight
print_perm(n, A, n, P, bestSet, bestDist);

// End the chronograph to time the loop
auto end = chrono::high_resolution_clock::now();
cout << "Input: n\n";
cout << "n=" << n << endl;
// after shuffling them
cout << "The Hamiltonian cycle of the minimum length " << endl;
print_cycle(n, P, bestSet);
cout << "Minimum length is " << bestDist << endl;

// print the elapsed time in seconds and fractions of seconds
int microseconds =
chrono::duration_cast<chrono::microseconds>(end - start).count();
double seconds = microseconds / 1E6;
cout << "elapsed time: " << seconds << " seconds" << endl;

// de-allocate the dynamic memory space
delete [] P;
delete [] A;
delete [] bestSet;
return EXIT_SUCCESS;
```

```
}

void print_cycle(int n, point2D *P, int *seq)
// function to print a sequence of 2D points in 2D plane, given the number of elements and the
actual
// sequence stored as an array of 2D points
// n is the number of points
// seq is a permutation over the set of indices
// P is the array of coordinates
{
    int i;

    for(i=0; i< n; i++)
        cout << "(" << P[seq[i]].x << "," << P[seq[i]].y << ") ";
    cout << "(" << P[seq[0]].x << "," << P[seq[0]].y << ") ";
    cout << endl;
}

float farthest(int n, point2D *P)
// function to calculate the furthest distance between any two 2D points
{
    float max_dist = 0;
    int i, j;
    float dist;

    for(i=0; i < n-1; i++)
        for(j=0; j < n;j++) {
            dist = abs(P[i].x - P[j].x) + abs(P[i].y - P[j].y);
            if (max_dist < dist)
                max_dist = dist;
        }
    return max_dist;

}

void print_perm(int n, int *A, int sizeA, point2D *P, int *bestSet, float &bestDist)
// function to generate the permutation of indices of the list of points
{
    int i;
    int j = 0;
    float dist = 0.0;

    if (n == 1) {
    // we obtain a permutation so we compare it against the current shortest
    // Hamiltonian cycle
    // YOU NEED TO COMPLETE THIS PART
        for(i = 0; i < sizeA - 1; i++) {
            dist += abs(P[A[i]].x - P[A[i+1]].x) + abs(P[A[i]].y - P[A[i+1]].y);
        }

        dist += abs(P[A[0]].x - P[A[sizeA-1]].x) + abs(P[A[0]].y - P[A[sizeA-1]].y);

        if (dist < bestDist) {
            bestDist = dist;
```

```
            for (i = 0; i < sizeA; i++) {
                bestSet[j] = A[i];
                j++;
            }
        }
    }
    else {
        for(i = 0 ; i< n-1; i++) {
            print_perm(n - 1, A, sizeA, P, bestSet, bestDist);
            if (n%2 == 0) {
                // swap(A[i], A[n-1])
                int temp = A[i];
                A[i] = A[n-1];
                A[n-1]=temp;
            }
            else {
                // swap(A[0], A[n-1])
                int temp = A[0];
                A[0] = A[n-1];
                A[n-1]=temp;
            }
        }
        print_perm(n - 1, A, sizeA, P, bestSet, bestDist);
    }
}
```

```
// Assignment 3: Euclidean traveling salesperson problem: improved nearest neighbor algorithm
/****************************************

* Name: Micah Geertson & Justin Stewart *
* CPSC 335-01 13115                     *
* Date: 04/20/2016                      *
****************************************/
// A special case of the classical traveling salesman problem (TSP) where the input is a
Euclidean graph
// INPUT: a positive integer n and a list P of n distinct points representing vertices of a
Euclidean graph
// OUTPUT: a list of n points from P representing a Hamiltonian cycle of relatively minimum
total weight
// for the graph.

#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <string>
#include <chrono>
#include <cmath>
using namespace std;

struct point2D {
    float x; // x coordinate
    float y; // y coordinate
};

void print_cycle(int, point2D*, int*);
// function to print a cyclic sequence of 2D points in 2D plane, given the
// number of elements and the actual sequence stored as an array of 2D points
// SAME AS IN THE PREVIOUS PROGRAM

int farthest_point(int, point2D*);
// function to return the index of a point that is furthest apart from some other point

int nearest(int, point2D*, int, bool*);
// function to calculate the nearest unvisited neighboring point

int main() {
    point2D *P;
    int *M;
    bool *Visited;
    int i, n;
    float dist;
    int A, B;

    // display the header
    cout << endl << "CPSC 335-x - Programming Assignment #3" << endl;
    cout << "Euclidean traveling salesperson problem: INNI algorithm" << endl;
    cout << "Enter the number of vertices (>2) " << endl;

    // read the number of elements
    cin >> n;
```

```
// if less than 3 vertices then terminate the program
if (n <3)
    return 0;

// allocate space for the sequence of 2D points
P = new point2D[n];

// read the sequence of distinct points
cout << "Enter the points; make sure that they are distinct" << endl;
for( i=0; i < n; i++) {
    cout << "x=";
    cin >> P[i].x;
    cout << "y=";
    cin >> P[i].y;
}

// allocate space for the INNA set of indices of the points
M = new int[n];
// set the best set to be the list of indices, starting at 0
for( i=0 ; i<n ; i++)
    M[i]=i;

// Start the chronograph to time the execution of the algorithm
auto start = chrono::high_resolution_clock::now();

// allocate space for the Visited array of Boolean values
Visited = new bool[n];
// set it all to False
for(i=0; i<n; i++)
    Visited[i] = false;

// calculate the starting vertex A
A = farthest_point(n,P);
// add it to the path
i=0;
M[i]= A;

// set it as visited
Visited[A] = true;

for(i=1; i<n; i++) {
    // calculate the nearest unvisited neighbor from node A
    B = nearest(n, P, A, Visited);
    // node B becomes the new node A
    A = B;
    // add it to the path
    M[i] = A;
    Visited[A]=true;
}

// calculate the length of the Hamiltonian cycle
dist = 0;
for (i=0; i < n-1; i++)
```

```
        dist += abs(P[M[i]].x - P[M[i+1]].x) + abs(P[M[i]].y - P[M[i+1]].y);
    dist += abs(P[M[0]].x - P[M[n-1]].x) + abs(P[M[0]].y - P[M[n-1]].y);

    cout << "Input: n\n";
    cout << "n=" << n << endl;

    // End the chronograph to time the loop
    auto end = chrono::high_resolution_clock::now();

    // after shuffling them
    cout << "The Hamiltonian cycle of a relative minimum length " << endl;
    print_cycle(n, P, M);
    cout << "The relative minimum length is " << dist << endl;

    // print the elapsed time in seconds and fractions of seconds
    int microseconds =
    chrono::duration_cast<chrono::microseconds>(end - start).count();
    double seconds = microseconds / 1E6;
    cout << "elapsed time: " << seconds << " seconds" << endl;

    // de-allocate the dynamic memory space
    delete [] P;
    delete [] M;
    return EXIT_SUCCESS;
}

int farthest_point(int n, point2D *P)
// function to calculate the furthest distance between any two 2D points
{
    float max_dist = 0.0;
    int i, j;
    float dist;
    int point;

    for(i=0; i < n-1; i++)
        for(j=0; j < n; j++) {
            dist = abs(P[i].x - P[j].x) + abs(P[i].y - P[j].y);
            if (max_dist < dist) {
                max_dist = dist;
                point = i;

            }
        }

    return point;
}

int nearest(int n, point2D *P, int A, bool *Visited)
// function to calculate the nearest unvisited neighboring point
{
    int i = 0;
    int nearest = 0;
    float dist = 100.0;
    float temp = 0.0;
```

```
    for (i = 0; i < n; i++) {
        if (!Visited[i]) {
            temp = abs(P[i].x - P[A].x) + abs(P[i].y - P[A].y);
            if (temp < dist) {
                dist = temp;
                nearest = i;
            }
        }
    }
    return nearest;
}

void print_cycle(int n, point2D *P, int *seq)
{
    int i;

    for(i=0; i< n; i++)
        cout << "(" << P[seq[i]].x << "," << P[seq[i]].y << ") ";
    cout << "(" << P[seq[0]].x << "," << P[seq[0]].y << ") ";
    cout << endl;
}
```

Sample Output

TSP Exhaustive Optimization Algorithm

```
me@tla-ubuntu-gnome: ~/Desktop
File  Edit  View  Search  Terminal  Help
me@tla-ubuntu-gnome:~/Desktop$ ./exhaustive

CPSC 335-x - Programming Assignment #3
Euclidean traveling salesperson problem: exhaustive optimization algorithm
Enter the number of vertices (>2)
4
Enter the points; make sure that they are distinct
x=2
y=0
x=1
y=1
x=3
y=1
x=0.1
y=0
Input: n
n=4
The Hamiltonian cycle of the minimum length
(2,0) (3,1) (1,1) (0.1,0) (2,0)
Minimum length is 7.8
elapsed time: 3e-06 seconds
me@tla-ubuntu-gnome:~/Desktop$
```

```
me@tla-ubuntu-gnome: ~/Desktop
File  Edit  View  Search  Terminal  Help

me@tla-ubuntu-gnome:~/Desktop$ ./exhaustive

CPSC 335-x - Programming Assignment #3
Euclidean traveling salesperson problem: exhaustive optimization algorithm
Enter the number of vertices (>2)
8
Enter the points; make sure that they are distinct
x=0
y=4
x=2
y=1
x=1
y=6
x=2
y=7
x=3
y=5
x=3
y=2
x=5
y=2
x=6
y=5
Input: n
n=8
The Hamiltonian cycle of the minimum length
(3,5) (2,7) (1,6) (0,4) (2,1) (3,2) (5,2) (6,5) (3,5)
Minimum length is 24
elapsed time: 0.003611 seconds
me@tla-ubuntu-gnome:~/Desktop$
```

TSP Nearest Neighbor Algorithm

```
me@tla-ubuntu-gnome: ~/Desktop
File  Edit  View  Search  Terminal  Help
me@tla-ubuntu-gnome:~/Desktop$ ./nearestNeighbor

CPSC 335-x - Programming Assignment #3
Euclidean traveling salesperson problem: INNI algorithm
Enter the number of vertices (>2)
4
Enter the points; make sure that they are distinct
x=2
y=0
x=1
y=1
x=3
y=1
x=0.1
y=0
Input: n
n=4
The Hamiltonian cycle of a relative minimum length
(3,1) (2,0) (0.1,0) (1,1) (3,1)
The relative minimum length is 7.8
elapsed time: 1.9e-05 seconds
me@tla-ubuntu-gnome:~/Desktop$
```

```
me@tla-ubuntu-gnome: ~/Desktop
File  Edit  View  Search  Terminal  Help

me@tla-ubuntu-gnome:~/Desktop$ ./nearestNeighbor

CPSC 335-x - Programming Assignment #3
Euclidean traveling salesperson problem: INNI algorithm
Enter the number of vertices (>2)
8
Enter the points; make sure that they are distinct
x=0
y=4
x=2
y=1
x=1
y=6
x=2
y=7
x=3
y=5
x=3
y=2
x=5
y=2
x=6
y=5
Input: n
n=8
The Hamiltonian cycle of a relative minimum length
(2,1) (3,2) (5,2) (6,5) (3,5) (1,6) (2,7) (0,4) (2,1)
The relative minimum length is 26
elapsed time: 2e-05 seconds
me@tla-ubuntu-gnome:~/Desktop$
```