Assignment 3: The rectilinear traveling salesperson problem

SangYong Jin, Danny Navarro, Shelley Pham

Improved Nearest Neighbor Algorithm

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

Exhaustive Optimization Algorithm

INPUT: a positive integer n and a list P of n distinct points representing vertices of a rectilinear graph
OUTPUT: a list of n points from P representing a Hamiltonian cycle of minimum total weight for the graph.

Contents

Pseudocode

Improved Nearest Neighbor Algorithm

farthest_point

`for (i = 0 to n - 1 do) {`	`- ((n-1-0)/1 + 1) = n`
`for (j = i to n do) {`	`- ((n-i)/1 + 1) = n+i+1`
`dist1 = abs(P[i].x - P[j].x);`	`-2`
`dist2 = abs(P[i].y - P[j].y);`	`-2`
`dist = dist1 + dist2;`	`-2`
`if (dist > max) { max = dist; A = i; }`	`-1 + Max(2 , 0 )`
`} } endfor`	

S.C = $\sum_{i=0}^{n-1}$ **(6(n+i+1)) = 3** n^2 +4n-1 = O(n^2)

Nearest Point

`for (int i = 0 to n ) {`	`- ((n-1-0)/1 + 1) = n`
`if (Visited[i] == false) {`	`-1 + max(6,3) = 7`
`dist1 = abs(P[i].x - P[A].x);`	`-2`
`dist2 = abs(P[i].y - P[A].y);`	`-2`
`dist = dist1 + dist2;`	`-2`
`if (dist < nearest) {`	`-1 + max(2,0) = 3`
`nearest = dist;`	`-1`
`B = i;`	`-1`
`}`	
`}`	
`}`	
`endfor`	

S.C = `((n-1-0)/1 + 1) + 1 + max(6,3) = 7n = O(n)`

Exhaustive Optimization Algorithm

if (n == 1) {
    for (i = 0; i < sizeA - 1; i++)
    {
        float index = abs(P[A[i]].x - P[A[i + 1]].x) + abs(P[A[i]].y - P[A[i + 1]].y);
        dist = dist + index;
    }
    float index = abs(P[A[0]].x - P[A[sizeA - 1]].x) + abs(P[A[0]].y - P[A[sizeA - 1]].y);
    dist = dist + index;
    if (dist < bestDist) {
        bestDist = dist;
        for (int i = 0; i < sizeA; i++)
        {
            bestSet[i] = A[i];
        }
    }
}
	-1+max(8n+8+2+n , 0)
	=9n + 11
	
	-n *(6+2)
	-6
	-2
	
	-6
	-2
	
	-1+max(1+n,0)
	= 2 + n
	-1
	-n

S.C. (since we are only considering the n==1 if branch, then I am assuming the else part in max(if,else) is 0.

=1 + max([n*(6+2)] + [6+2] + [1 +max(1 + n, 0)])

=1 + max(8n + 8 + 1 + 1 + n) = 1 + max(9n + 10, 0)

=1 + 9n + 10

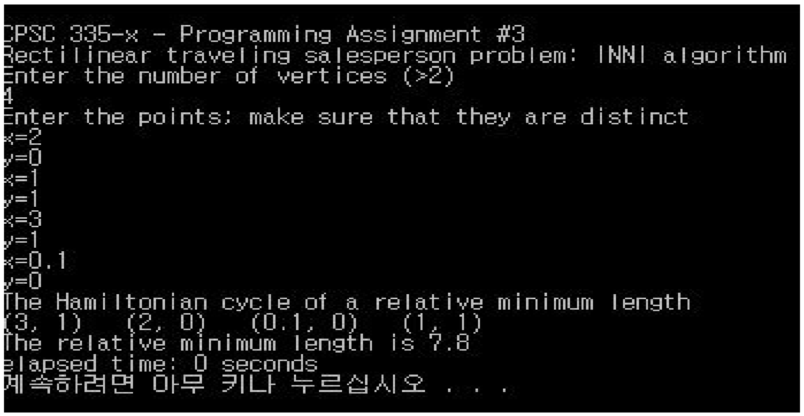
=9n +11

= O(n)

Output

Improved Nearest Neighbor Algorithm

Example 1



Example 2

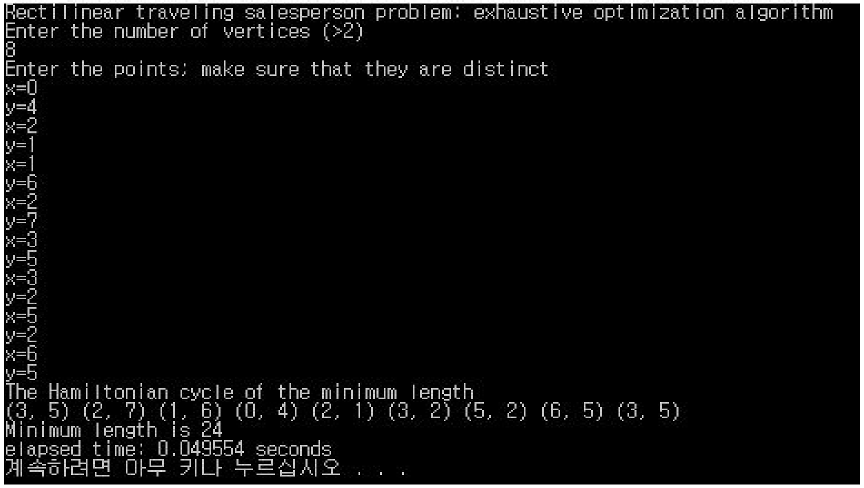
```
Rectilinear traveling salesperson problem: INNI algorithm
Enter the number of vertices (>2)
8
Enter the points; make sure that they are distinct
x=0
y=4
x=2
y=1
x=1
y=6
x=2
y=7
x=3
y=5
x=3
y=2
x=5
y=2
x=6
y=5
The Hamiltonian cycle of a relative minimum length
(2, 1)   (3, 2)   (5, 2)   (6, 5)   (3, 5)   (1, 6)   (2, 7)   (0, 4)
The relative minimum length is 26
elapsed time: 0 seconds
계속하려면 아무 키나 누르십시오 . . .
```

Exhaustive Optimization Algorithm

Example 1

```
CPSC 335-x - Programming Assignment #3
Rectilinear 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
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: 0 seconds
계속하려면 아무 키나 누르십시오 . . .
```

Example 2



Code

#Improved Nearest Neighbor Algorithm

```
#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
// YOU NEED TO IMPLEMENT THIS FUNCTION

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

int nearest(int, point2D*, int, bool*);
// function to calculate the nearest unvisited neighboring point
// YOU NEED TO IMPLEMENT THIS FUNCTION

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 << "Rectilinear 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);

// 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;
```

```
	system("pause");
	return EXIT_SUCCESS;
}

int farthest_point(int n, point2D *P)
// function to calculate the furthest distance between any two 2D points
// YOU NEED TO IMPLEMENT THIS FUNCTION
{
	float max = 0, dist = 0, dist1 = 0, dist2 = 0;
	int A = 0;
	for (int i = 0; i < n - 1; i++) {
		for (int j = i; j < n; j++) {
			dist1 = abs(P[i].x - P[j].x);
			dist2 = abs(P[i].y - P[j].y);
			dist = dist1 + dist2;
			if (dist > max) {
				max = dist;
				A = i;
			}
		}
	}
	return A;
}



int nearest(int n, point2D *P, int A, bool *Visited)
// function to calculate the nearest unvisited neighboring point
// YOU NEED TO IMPLEMENT THIS FUNCTION
{
	float dist = 0, dist1 = 0, dist2 = 0, nearest = numeric_limits<float>::max();

	int B = 0;
	for (int i = 0; i < n; i++) {
```

```
            if (Visited[i] == false) {
                  dist1 = abs(P[i].x - P[A].x);
                  dist2 = abs(P[i].y - P[A].y);
                  dist = dist1 + dist2;
                  if (dist < nearest) {
                              nearest = dist;
                              B = i;
                  }
            }
      }
      return B;
}


void print_cycle(int n, point2D *P, int *seq)
// YOU NEED TO IMPLEMENT THIS FUNCTION
{
      for (int i = 0; i < n; i++) {
            cout << "(" << P[seq[i]].x << ", " << P[seq[i]].y << ")    ";
      }
      cout << endl;
}
```

#Exhaustive Optimization Algorithm

```
#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
// YOU NEED TO IMPLEMENT THIS FUNCTION

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 << "Rectilinear 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();

    // 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
// YOU NEED TO IMPLEMENT THIS FUNCTION
{
      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;
	float dist = 0;

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

		}
		float index = abs(P[A[0]].x - P[A[sizeA - 1]].x) + abs(P[A[0]].y - P[A[sizeA - 1]].y);
		dist = dist + index;
		if (dist < bestDist) {
			bestDist = dist;
			for (int i = 0; i < sizeA; i++)

			{
					bestSet[i] = A[i];

			}
		}
	}
	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);
    }
}
```