# CPSC 335 Project 3: Euclidean traveling salesperson problem

Fall 2015

Prof. Doina Bein, CSU Fullerton

dbein@fullerton.edu

## Introduction

In this project you will design, implement, and analyze two algorithms for the same problem, the Euclidean traveling salesperson problem. As you might expect, it is a special case of the classical traveling salesman problem (TSP) where the input is a Euclidean graph.

For this problem, you will design two algorithms, describe the algorithms using clear pseudocode, analyze them mathematically, implement your algorithms in C/C++/Java/Python, measure their performance in running time, compare your experimental results with the efficiency class of your algorithms, and draw conclusions.

## The hypotheses

This experiment will test the following hypotheses:

1. *Exhaustive search algorithms are feasible to implement, and produce correct outputs.*
2. *Approximation algorithms are reasonably fast and produce reasonably good outputs.*

## The Euclidean traveling salesperson problem (ETSP)

Remember that a *coordinate* is a number $x \in \Re$, and in the plane, a *point* is a pair $(x, y) \in \Re^2$. A *Euclidean graph* is a complete, weighted, undirected graph $G = (V, E)$ where $V$ is a set of points $V \subset \Re^2$ and $E$ is the set of all possible edges between distinct points

$$E = \bigcup_{p,q \in V,\, p \neq q} \{ \{p,q\} \},$$

and the weight $w(p,q)$ of an edge between $p$ and $q$ is defined by the Euclidean distance

$$w(\,(x_p, y_p),\ (x_q, y_q)\,) \;=\; \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2}.$$

A Euclidean graph is complete, so may be defined entirely by the points comprising its vertices. There is no need to explicitly store a set of edges, since every pair of distinct vertices is connected, and the weight of such an edge may be computed as needed by evaluating the Euclidean distance function.

The Euclidean traveling salesperson problem is:
*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.

We developed an exhaustive search algorithm that solves the TSP problem on general graphs. Its time complexity is $O(n \cdot n!)$, which is even slower than factorial, so theory predicts that an implementation of this algorithm will be extremely slow.

Dirac's theorem states that:

> *If each vertex of a connected graph with 3 or more vertices is adjacent to at least half of the remaining vertices, then the graph has a Hamilton Circuit. Two vertices are ADJACENT if there is an edge connecting them.*

In Euclidean space, every node is connected to the rest of the nodes, so the ETSP problem will always have a solution.

## The Exhaustive Optimization Algorithm (EOA)

The exhaustive optimization algorithm will list all possible Hamilton cycles (leaving out the exact reversals, if you wish), find the weight of each cycle, and choose the one with the smallest weight.

A Hamiltonian cycle is a Hamiltonian path in which the first vertex is added at the end. Since all vertices are connected in a Euclidean space, once you have a Hamiltonian path one can always obtain the Hamiltonian cycle from it by appending the first vertex.

To generate all possible Hamiltonian paths is equivalent to generating all permutations of vertices. There are several algorithms for generating permutations, including Heap's algorithm presented in class. It is up to you to choose such an algorithm, provided that it will generate all permutations, so it is not a random permutation algorithm, that generate some random permutations.

Steps of the EOA:
1. Initially the best solution has infinite length (or a very large number). One way of calculating this large number is to calculate the farthest pair of vertices. Let us consider A and B to be such vertices
and DIST to be the distance. Then the large number will be $N * DIST$, where N is the total number of vertices.
2. Generate all permutations. For each permutation, calculate the length of the Hamiltonian cycle generated by the permutation and compare it with the best solution found so far. If a shortest cycle is found, store it as the current best solution.
When calculating the length of the Hamiltonian cycle, do not forget to include the edge between the last vertex and the first vertex of the Hamiltonian path.

3. Output the best solution.

EOA always works, given enough time to find the optimal result. But it can only be used for relatively small graphs. For a computer doing 10,000 cycles/second, it would take about 18 seconds to handle 10 vertices, 50 days to handle 15 vertices, 2 years for 16 vertices, 193,000 years for 20 vertices.

Unfortunately, the exhaustive optimization algorithm is the ONLY method known that is guaranteed to produce an optimal solution.

Mathematicians have not been able to prove that another such method exists nor have they been able to prove that one doesn't exist.

This is one of the most important and famous unsolved problems in mathematics.

## Improved Nearest Neighbor Algorithm (INNA)

Although we do not have an efficient algorithm for solving ETSP, we do have several algorithms that produce results that may not be optimal. In this respect, we are willing to give up our requirement for an optimal solution in the interest of time and settle for a "good" solution which may not be optimal. We call this class of algorithms *approximate algorithms*.

For approximate algorithms, we will be interested in a quantity called the relative error.

$$\text{Relative Error} = \frac{\text{Difference between the Solution and the Optimal Solution}}{\text{Optimal Solution}}$$

The relative error tells us how close the approximate solution is to the optimal solution.
The Relative Error is important but one will be able to calculate the Optimal Solution only for simple cases. A complete discussion of evaluating the performance of an approximate algorithm is beyond the scope of this course. But in summary, two important factors are:
(1) worst-case performance and
(2) average performance.

Steps of the INNA:
1. Calculate the farthest pair of vertices. Let us consider A and B to be such vertices.
2. Start at a vertex A.
3. Let us consider a generic node V which is the current node. Starting from node V, travel to the vertex that you haven't been to yet but is the closest to V. If there is a tie, pick randomly the next vertex.
4. Continue until you travel to all vertices
5. Travel back to your starting vertex A.
6. Output the solution.

The resulting path is a Hamilton Cycle.

## What to do

1. Write clear pseudocode for each algorithm.

2. Analyze your pseudocode mathematically and prove its efficiency class using Big-Oh notation. (You need to compute the total number of steps of the algorithm.)

3. Type these notes (electronically or on paper) and submit it as a PDF report.

4. Implement your algorithm in C/C++/Java/Python. You may use the templates provided at the end of this file.

5. Compile and execute the program.

6. Create a file with the output of the program for an input value and submit it together with the program. Note, the output can be redirected to a file (for easy printing). For example, the following command line will create an output file in Linux-based operating system called `a1out.txt` by re-directing the output from the screen (display) to the file a1out.txt:

```
K:\cpscs335> a.out > a3out.txt
```

## Sample outputs Exhaustive Optimization Algorithm:

Example #1:

```
K:\202> ast3a
CPSC 335-x – Programming Assignment #3:
The 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 6.65958
elapsed time: 8e-06 seconds
```

Example #2:

```
K:\202> ast3a
CPSC 335-x – Programming Assignment #3:
The 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
(3,2) (5,2) (6,5) (3,5) (2,7) (1,6) (0,4) (2,1) (3,2)
Minimum length is 19.0684
elapsed time: 0.008589 seconds
```

## Sample outputs for Improved Nearest Neighbor Algorithm:

Example #1:

```
K:\202> ast3b
CPSC 335-x – Programming Assignment #3:
The Euclidean traveling salesperson problem: improved nearest neighbor 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
(3,1) (2,0) (1,1) (0.1,0) (3,1)
The relative minimum length is 7.24136
```

```
elapsed time: 6e-06 seconds
```

<u>Example #2:</u>

```
K:\202> ast3b
CPSC 335-x – Programming Assignment #3:
The Euclidean traveling salesperson problem: improved nearest neighbor 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
(0,4) (1,6) (2,7) (3,5) (3,2) (2,1) (5,2) (6,5) (0,4)
The relative minimum length is 22.7079
elapsed time: 2.2e-05 seconds
```

## C/C++ program for generating the set of all permutations in the range 0..n-1

```
// Generate all the permutations with values in the range 0..n-1
// INPUT: a positive integer n
// OUTPUT: all the permutations
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;
void generate_perm(int n, int *A, int sizeA) {
  int i;
  if (n == 1) {
    cout << endl;
    cout << "Permutation: ";
    for(i=0; i< sizeA; i++)      cout << A[i] << " " ;
    cout << endl;
  }
 else {
    for(i = 0 ; i< n-1; i++) {
      generate_perm(n - 1, A, sizeA);
      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;
        }
    }
    generate_perm(n - 1, A, sizeA);
  }
}
int main() {
  int i, n;
  int *A;
  cout << "Enter the range of the values to be permuted (n > 0)" << endl;
  cin >> n;
  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;
  generate_perm(n,A, n);
  delete [] A;
}
```

## Template for a C/C++ program doing Exhaustive Optimization algorithm:

```
// Assignment 3: Euclidean traveling salesperson problem: exhaustive optimization algorithm
// XX YY ( YOU NEED TO COMPLETE YOUR NAME )
// 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>
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();

// 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-1;j++) {
      dist = (P[i].x - P[j].x)*(P[i].x - P[j].x) + (P[i].y - P[j].y)*(P[i].y - P[j].y);
      if (max_dist < dist)
        max_dist = dist;
    }
  return sqrt(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;

  if (n == 1) {
    // we obtain a permutation so we compare it against the current shortest
    // Hamiltonian cycle
    // YOU NEED TO COMPLETE THIS PART
  }
  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);
  }
}

void print_cycle(int, point2D*, int*)
{
// YOU NEED TO IMPLEMENT THIS FUNCTION
}
```

## Template for a C/C++ program doing Improved Nearest Neighbor Algorithm:

```
// Assignment 3: Euclidean traveling salesperson problem: improved nearest neighbor algorithm
// XX YY ( YOU NEED TO COMPLETE YOUR NAME )
// 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>
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: 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 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 += sqrt((P[M[i]].x - P[M[i+1]].x)*(P[M[i]].x - P[M[i+1]].x) +
                 (P[M[i]].y - P[M[i+1]].y)*(P[M[i]].y - P[M[i+1]].y));
  dist += sqrt((P[M[0]].x - P[M[n-1]].x)*(P[M[0]].x - P[M[n-1]].x) +
               (P[M[0]].y - P[M[n-1]].y)*(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;
  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
}

int nearest(int n, point2D *P, int A, bool *Visited)
// function to calculate the nearest unvisited neighboring point
{
  // YOU NEED TO IMPLEMENT THIS FUNCTION
}
```