CPSC 335 Project 2: exhaustive search

Spring 2015
Prof. Kevin Wortman, CSU Fullerton
kwortman@fullerton.edu

Introduction

In this project you will implement two algorithms that solve related problems, and compare their performance empirically. The first has a tractable (polynomial) running time, while the second has an intractable (exponential or factorial) running time.

The first problem is the *Euclidean minimum spanning tree* problem, which is a special case of the classical minimum spanning tree (MST) problem, restricted to *Euclidean graphs*. A Euclidean graph is a complete graph where each vertex is a (x, y) point in the Euclidean plane, and an edge's weight is the Euclidean distance between the two points at its ends.

The second problem is the *Euclidean traveling salesperson* problem. As you might expect, it is a special case of the classical traveling salesperson problem (TSP) where the input is a Euclidean graph.

The hypotheses

This experiment will test the following hypotheses:

1. *Exhaustive search algorithms are feasible to implement, and produce correct outputs.*
2. *Algorithms with exponential or factorial running times are extremely slow, probably too slow to be of practical use.*

The problems

Both problems take a Euclidean graph as input. 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 minimum spanning tree problem is:
input: a list P of distinct points representing vertices of a Euclidean graph
output: a list of tuples (p, q) where each (p, q) tuple corresponds to an edge between p and q in a minimum spanning tree of the graph

Recall that we have studied the Prim-Jarník algorithm that solves the MST problem on general graphs. The version we developed in class has time complexity $O(mn)$, though I claimed that it is possible to speed this up to $O(m \log n)$ or even $O(m + n \log n)$ through reduction to data structure operations (we will get back to that soon). All these time complexities are polynomial, so our theory predicts that an implementation of any of these variants will be reasonably fast.

The Euclidean traveling salesperson problem is:
input: a list P of distinct points representing vertices of a Euclidean graph
output: a list of points from P representing a Hamiltonian cycle of minimum total weight for the graph

We developed an exhaustive search algorithm that solved the TSP problem on general graphs. Its time complexity is $O(n^3 \cdot n!)$, which is factorial, so our theory predicts that an implementation of this algorithm will be extremely slow.

What to do

1. Implement an algorithm that solves the Euclidean MST problem. As you can see in the stub code, your algorithm should go in a `euclidean_mst` function that takes a list of `Point` objects and returns a list of `Point` pairs. It will probably be easiest to use the reduction pattern, using an existing MST algorithm to do most of the computation. You may use any of the following MST implementations:

    a. You may convert the pseudocode for the $O(mn)$ -time Prim-Jarník algorithm in our lecture notes into a working implementation.
    b. *Challenge:* You may research the $O(m \log n)$ -time Prim-Jarník, Kruskal, or Borůvka algorithm, and implement it on your own.
    c. You may use the MST algorithm implementation in the NetworkX or PADS libraries. If you go this route, you will have to install and learn about the library yourself.

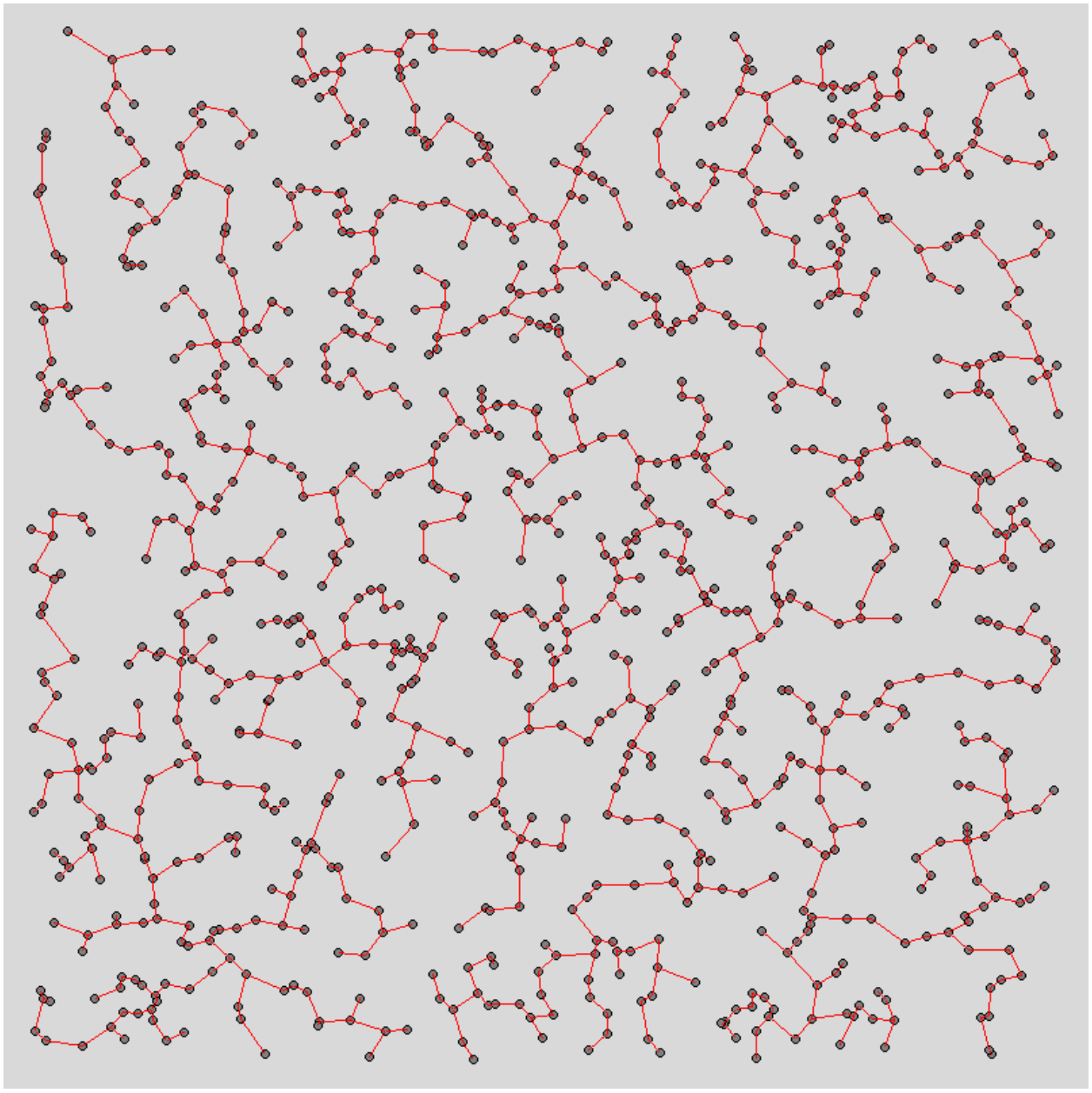
2. Adapt our exhaustive search algorithm for the general TSP problem to work for the Euclidean MST problem. This will involve computing the total weight of an ordered cycle through points. Remember that the total cost of a cycle includes the "wrap-around" edge from the last vertex back to the first; it is easy to overlook that detail. Your code will probably need to generate permutations of the input point list. You may either implement the pseudocode in our lecture notes, or use the `permutations` function in Python's itertools module.
3. For each of your two algorithms:
    a. Analyze your implementation code mathematically and prove its efficiency class. If you invoke a MST algorithm in a library, you will need to ascertain the time efficiency of that algorithm.
    b. Gather empirical timing data by running many trials over increasing values of n.
    c. Draw a scatter plot and fit line for your timing data.
    d. Conclude whether the empirical data is consistent or inconsistent with your mathematical analysis.

I have provided you with a Python 3 source file, `project2_stub.py`, that you can use as the basis for your code.

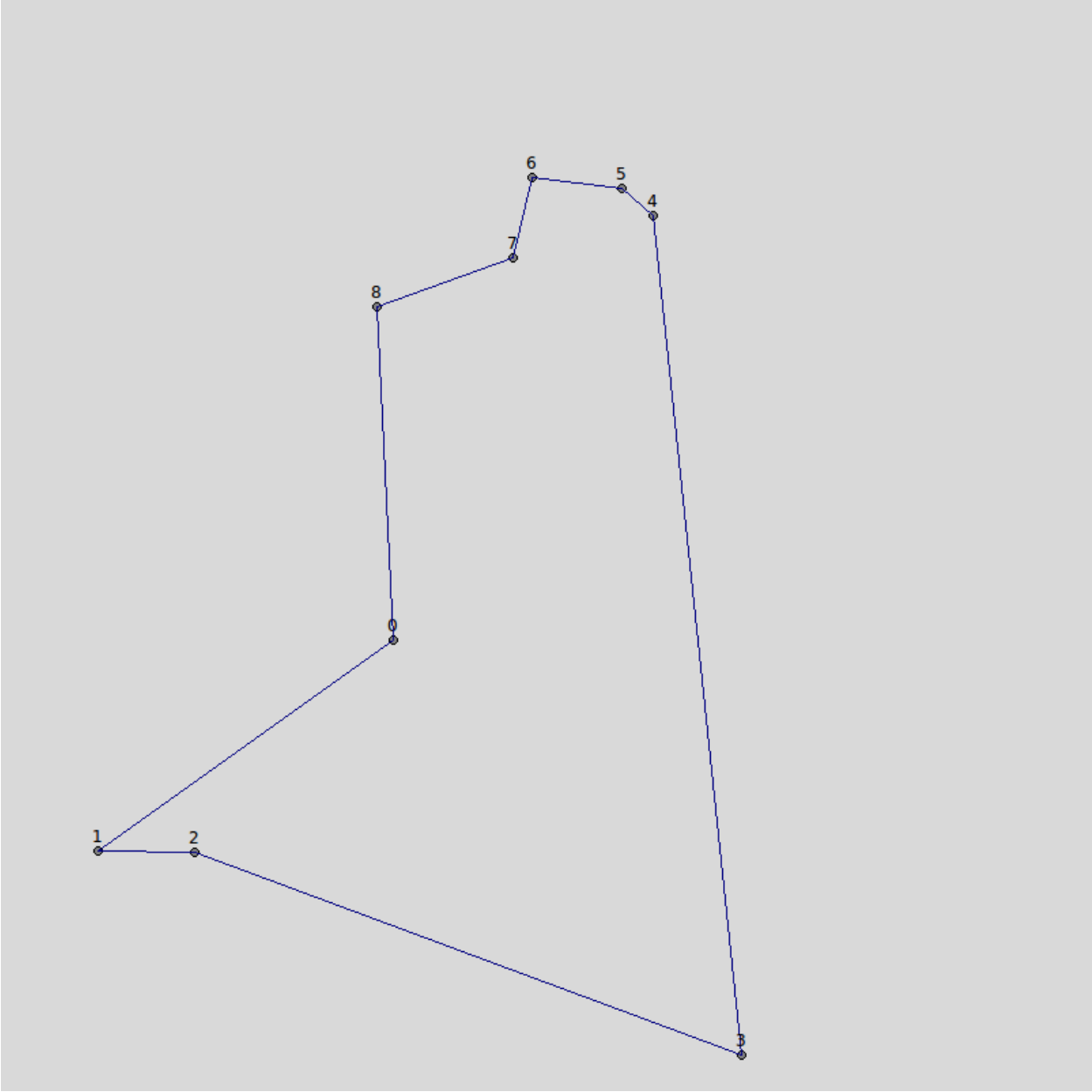
Sample output

The following is the output produced by a MST implementation for $n = 1,000$. The MST edges are drawn in red.

```
generating n=1000 points...
minimum spanning tree...
elapsed time = 2.89853378800035 seconds
```



The following is the output produced by a TSP implementation for $n = 9$. The TSP edges are drawn in navy blue, and the vertices are labeled with their order in the cycle.



Deliverables

Produce a written project report ***in PDF format***. Your report should include the following:

1. Your name(s)[1], CSUF-supplied email address(es), and an indication that the submission is for project 2.
2. Two scatter plots as described above.

[1] Recall that, as stated on the syllabus, you may work in a group of up to three students.

3. Screenshots and standard output from your implementations for two runs of each of your two algorithm implementations (so a total of four sets of output).
4. Answers to the following questions, using complete sentences.
    a. Is there a noticeable difference in the performance of the two algorithms? Which is faster, and by how much? Does this surprise you?
    b. Are your empirical analyses consistent with your mathematical analyses? Justify your answer.
    c. Is this evidence *consistent* or *inconsistent* with hypothesis 1? Justify your answer.
    d. Is this evidence *consistent* or *inconsistent* with hypothesis 2? Justify your answer.
5. Your complete Python source code.

Your document ***must be uploaded to TITANium as a single PDF file.***

Deadline

The project deadline is Thursday, 4/23, 11:55 pm. Late submissions will not be accepted.

License