Some topics on computational geometry
— T.11 —

Summary:

- Steiner trees
- Point location (grid, quad-tree, oct-tree)
- Convex hull, Delaunay triangulation, Voronoi diagram (examples)

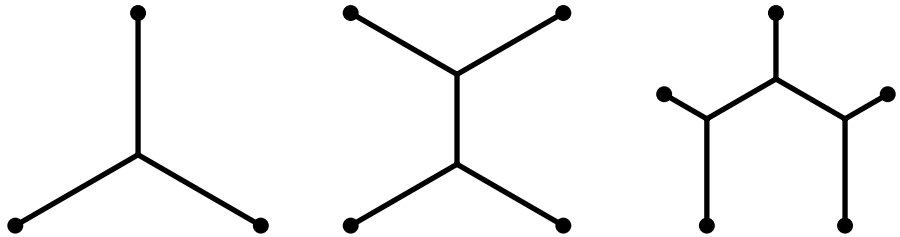
Recommended bibliography for this lecture:

- **Computational Geometry. Algorithms and Applications**, M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, second edition, Springer, 2000.
- **Computational Geometry in C**, Joseph O'Rourke, second edition, Cambridge University Press, 1998.

Steiner trees

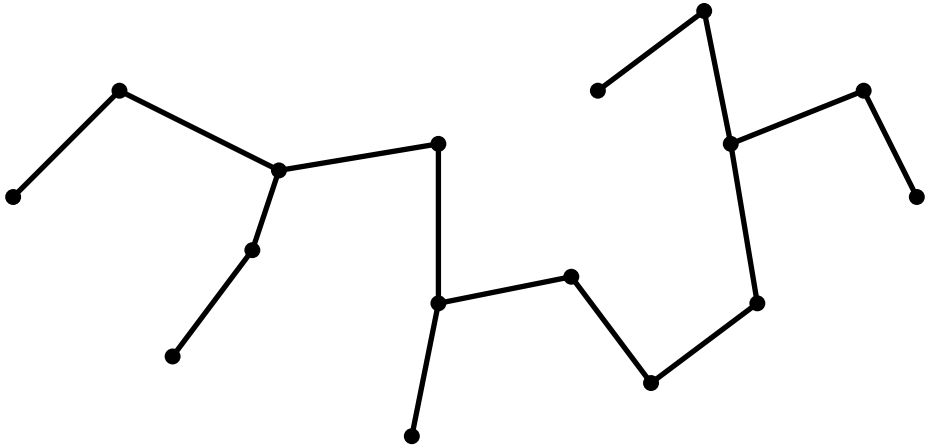
Problem: Given a set of points in the plane, find an interconnection tree with minimal total distance.

This is a very difficult problem (NP-hard).



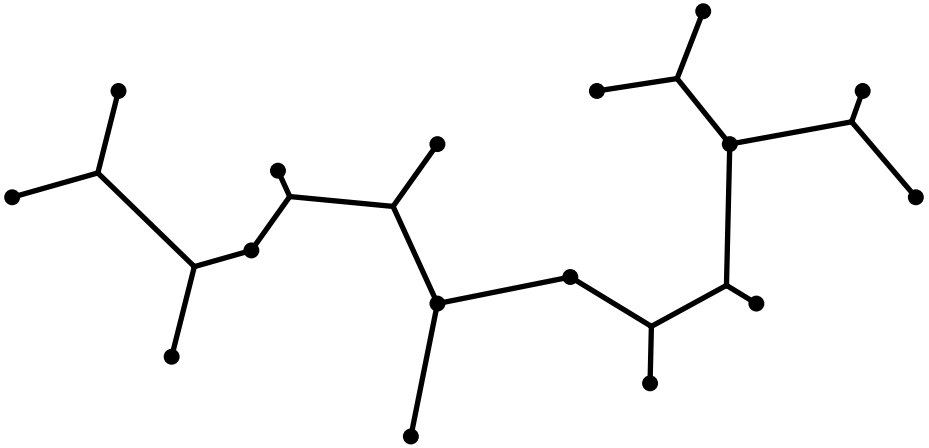
(Can be generalized to more than two dimensions.)

Minimum spanning tree



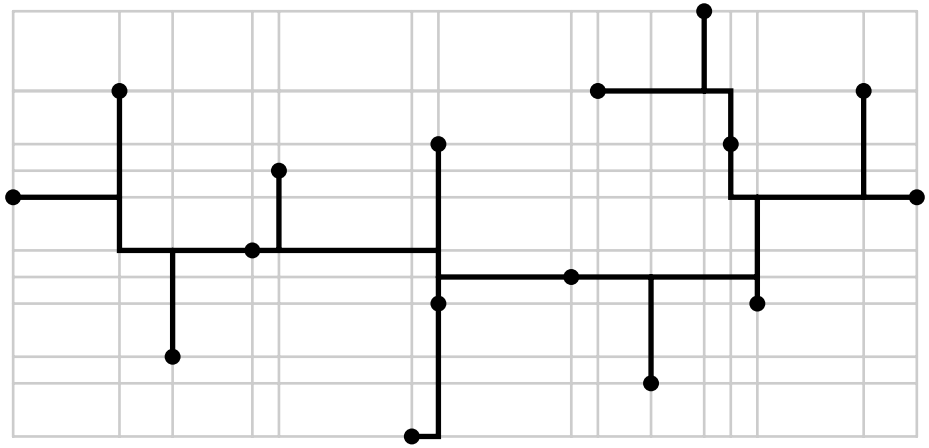
Total length of $\mathbf{788.472}$

Euclidean minimum Steiner tree



Total length of $\mathbf{759.996}$

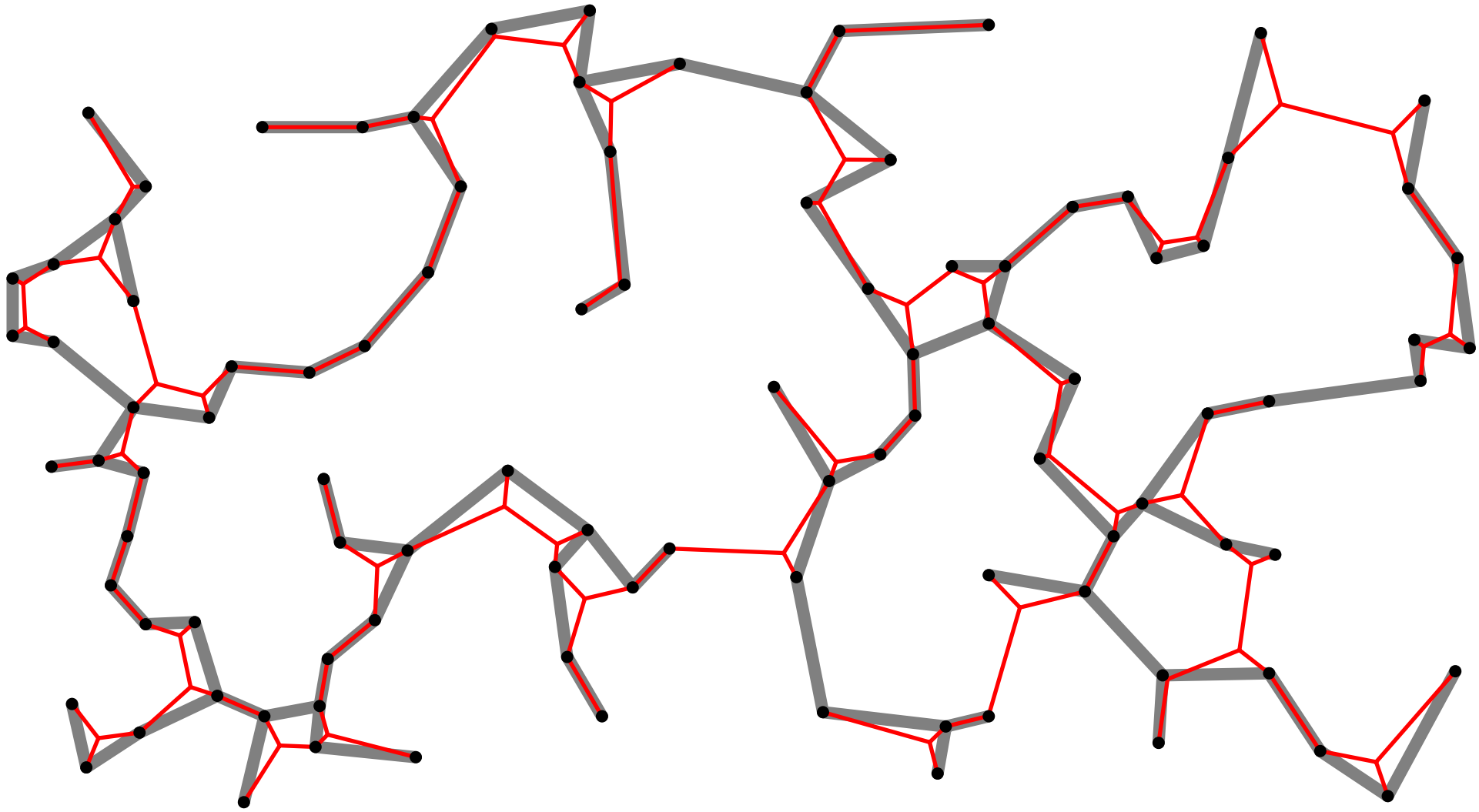
Rectilinear minimum Steiner tree



Total length of $\mathbf{840.000}$

Steiner trees ("large" example, Euclidean minimum Steiner tree)

(Example made with the help of the geosteiner program)

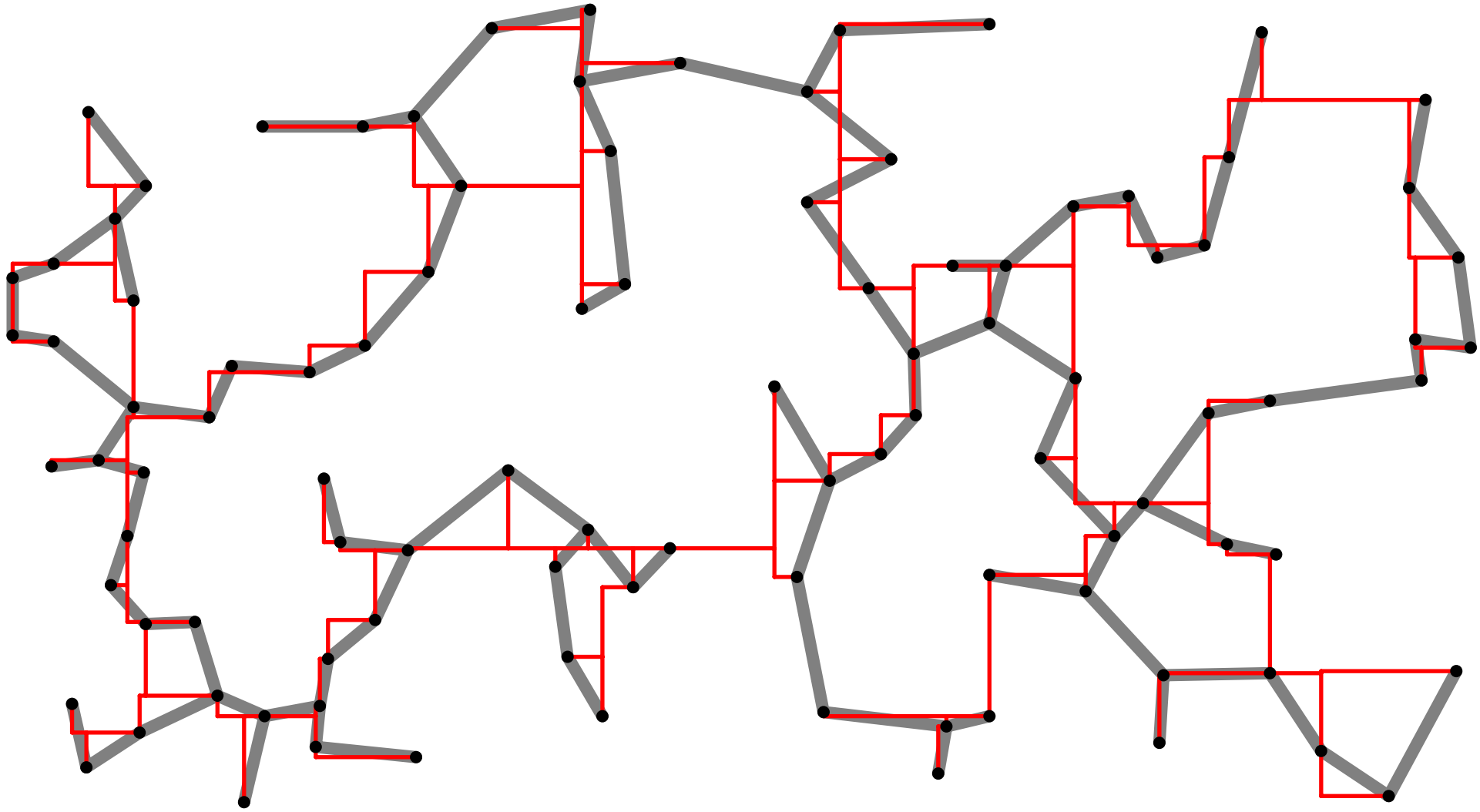


Minimum spanning tree total length of $\mathbf{3843.871}$

Euclidean minimum Steiner tree total length of $\mathbf{3717.030}$

Steiner trees ("large" example, Rectilinear minimum Steiner tree)

(Example made with the help of the geosteiner program)

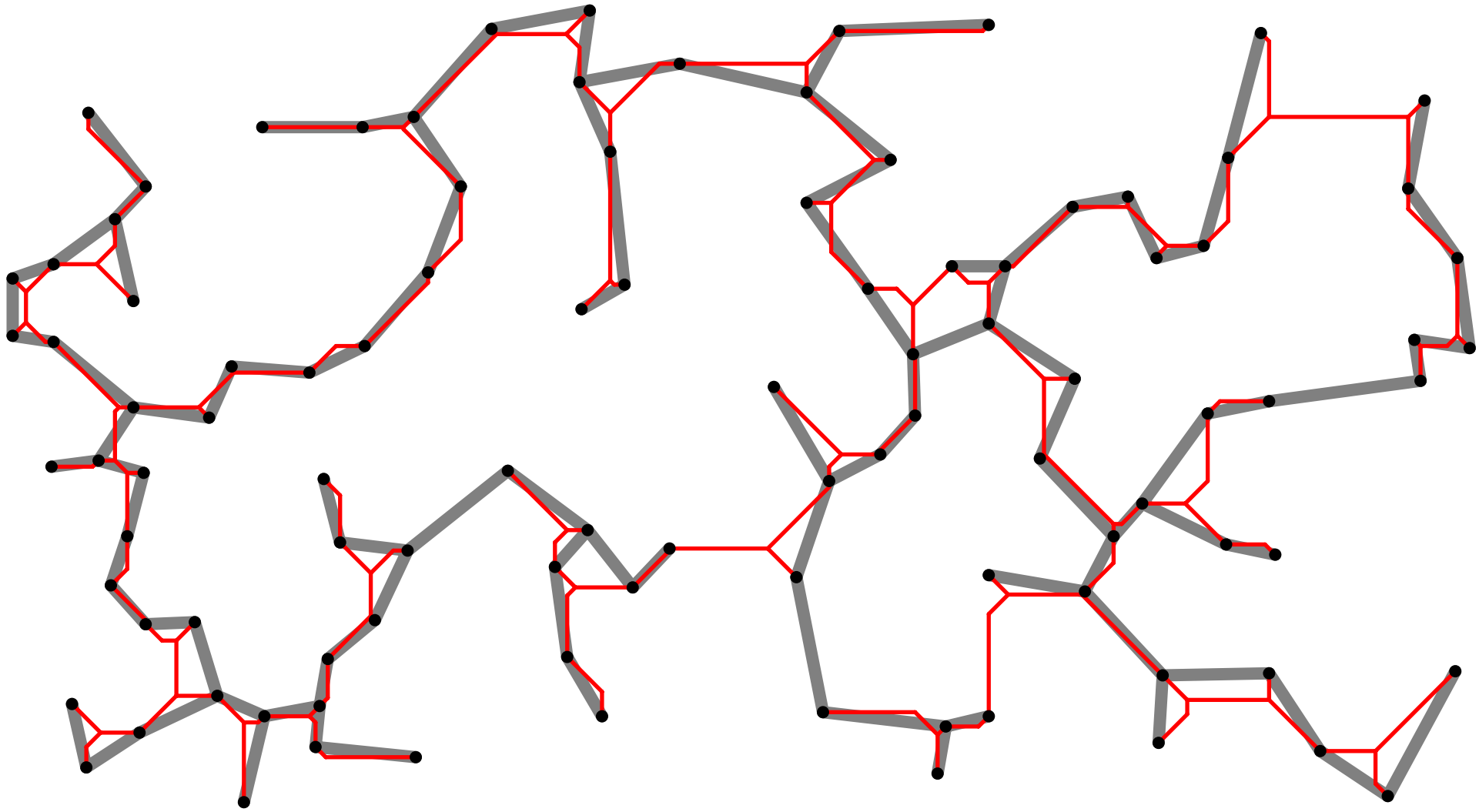


Minimum spanning tree total length of $\mathbf{3843.871}$

Rectilinear minimum Steiner tree total length of $\mathbf{4233.000}$

Steiner trees ("large" example, Octilinear minimum Steiner tree)

(Example made with the help of the geosteiner program)



Minimum spanning tree total length of $\mathbf{3843.871}$

Octilinear minimum Steiner tree total length of $\mathbf{3847.034}$

Point location (part 1, problem formulation)

Problems: Given a set S of n points on a plane (or in 3-dimensional space) and another point p:

- find the point of S closer to p;
- find the k points of S closer to p; or
- find all points of S whose distance to p is smaller that d.

The obvious algorithm, trying all points of S, solves the first and third problems in $O(n)$. By sorting all distances to p the answer to the second problem can be obtained in $O(n \log n)$, irrespective of the value of k.

Is it possible to do better? After all, in the one-dimensional case a binary search solves the first problem in $O(\log n)$ (this, of source, assumes that the points are already sorted). The answer is yes. The following slides present some simple techniques that can be used to speed up point location problems.

This problem appears in many applications:

- visualization of the part of a map near a given location,
- location of the restaurant nearer to a given location,
- in a game, discovery is a given projectile is close enough to a target,
- and so on.

Point location (part 2, subdivision of space using a regular grid)

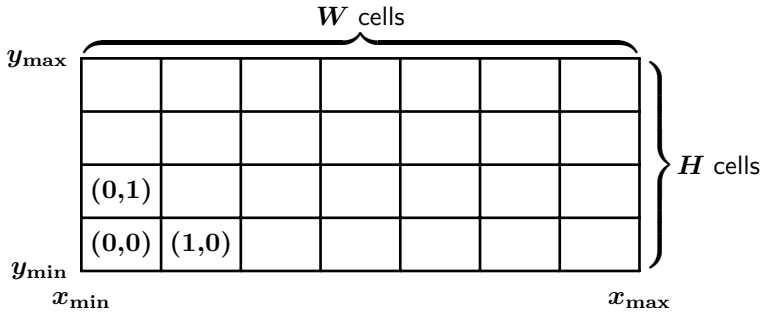
One way to speed up point location queries consists of subdividing the plane into a regular grid. Each grid cell keeps a linked list of the points that belong to that cell. For example, suppose that it is known that the x coordinates of the points satisfy $x_{\min} \leqslant x < x_{\max}$, and that the y coordinates satisfy $y_{\min} \leqslant y < y_{\max}$ (note the strict inequality in the upper bounds). Furthermore, suppose that our grid has W cells in the x direction and H cells in the y direction. The width of a cell will then be $w = (x_{\max} - x_{\min})/W$, and its height will be $y = (x_{\max} - y_{\min})/H$. A point with coordinates (x, y) will then belong to the cell with coordinates

$$\left(\left\lfloor \frac{x - x_{\min}}{w} \right\rfloor, \left\lfloor \frac{y - y_{\min}}{h} \right\rfloor\right).$$

Locating the nearest point would then involve looking at the list of points of the cell where the query point lies, and at the lists of nearby cells.

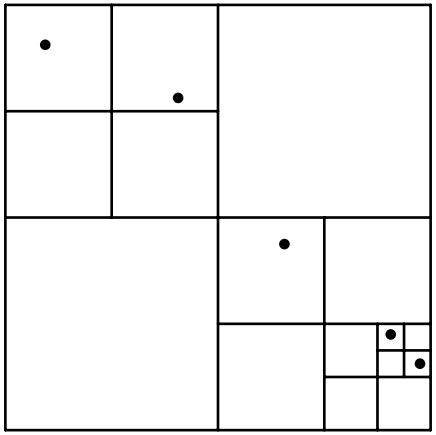
Assuming that the points are well distributed and that $wh \approx n$, finding the nearest point takes, on average and like a hash table, $O(1)$ time.

In dynamic situations, where the coordinates of the points change over time (but stay within the bounds given above), maintaining this data structure is easy: when the cell coordinates of a point change, one deletes the point from one linked list and one inserts it in another linked list. (Doubly-linked lists are better for this!)



Point location (part 3, subdivision of space using a quad-tree or oct-tree)

Another possibility consists of subdividing the space into smaller and smaller regions, in a tree-like manner, as illustrated in the following figure for the two-dimensional case.



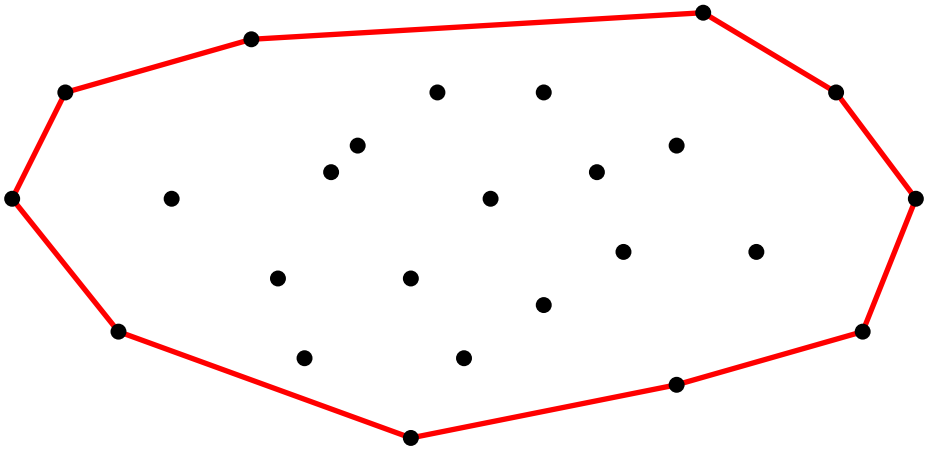
At each tree level one distributes the points that belong to a given tree branch (a rectangular region) into four subbranches (2D) or into eight subbranches (3D). The subdivision process stops when there is only one point left, in which case no further subdivision is necessary to disambiguate the points.

In two dimensions, the data structure implementing this idea is called a **quad-tree**. In three dimensions it is called an **oct-tree**.

Finding the nearest point in a quad-tree or oct-tree is not as simple as it is was using a regular space subdivision. Also, the data structure has trouble dealing with coincident points. It, however, adapts itself nicely to an arbitrary distribution of points (without using an excessive amount of memory and using a number of tree levels that is usually not too large).

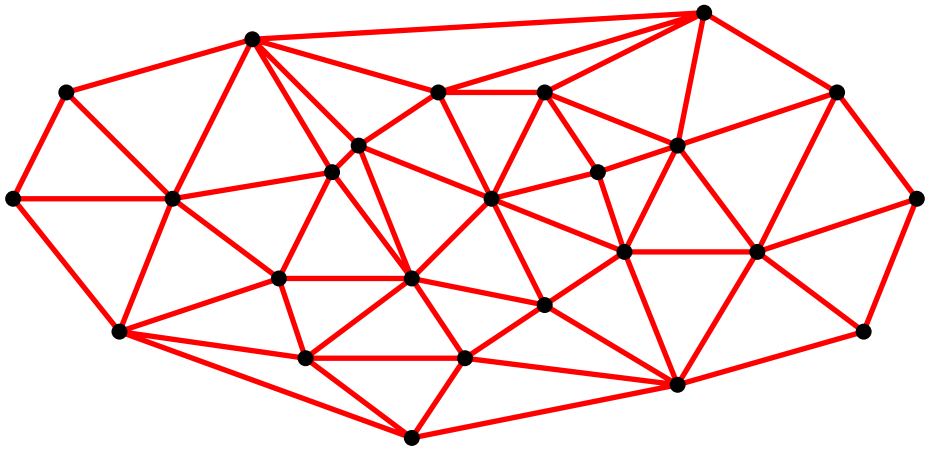
Examples (convex hull, Delaunay triangulation, Voronoi diagram)

Convex hull



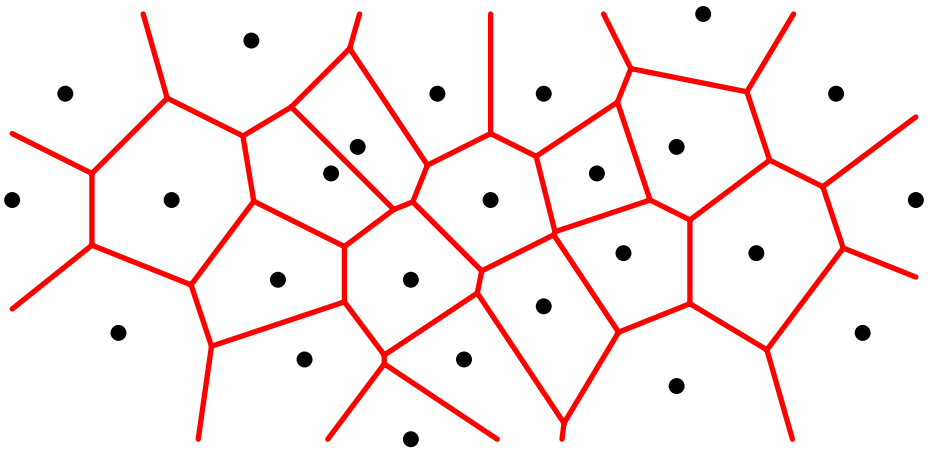
Can be computed in $O(n \log n)$ time

Delaunay triangulation



Can be computed in $O(n \log n)$ time

Voronoi diagram (points)



Can be computed in $O(n \log n)$ time

Voronoi diagram (points and line segments)