University of West Bohemia
Department of Computer Science and Engineering
Univerzitní 8
306 14 Pilsen
Czech Republic

Algorithms for Manipulation with Large Geometric and Graphic Data

Jiří Skála

Supervisor: Ivana Kolingerová

Technical Report No. DCSE/TR-2009-02
April 2009

Distribution: Public

Abstract

The theme of this work is manipulating large data in the field of computer graphics. Generally, large data appear in many scientific disciplines ranging from weather forecasting to marketing analyses. The computing power of modern computers still increases but so do the demands to process larger and larger data sets. The main memory is in principle insufficient to hold all the data at the same time so techniques are developed to handle the data in pieces. Random access is unacceptable in such cases so special, so called out-of-core, methods are used to process the data.

Data stream algorithms are frequently used for efficient computations on large data. The algorithms are characterised by processing the data as a continuous stream in one or very few linear scans. Streaming algorithms were getting more attention in the last few years, however, they are not much used in computer graphics.

This work first describes the state of the art concerning large data and data streams. An overview of clustering and a Delaunay triangulation follows. Next we propose a solution for manipulation with large geometric data. It is based on a clustering that identifies groups in the data. Each group is then replaced by a representative which reduces the data significantly. A data stream approach is used to cluster really huge data. A hierarchy of clusters is built which is then used by the dynamic hierarchical triangulation. It constructs a triangulation of the clustered data. By switching between clusters and their representatives, the level of detail can be changed in various parts of the data.

Several more improvements are presented. The clustering algorithm was adapted. Both the clustering and the triangulation can use anisotropic metrics if suitable for any specific problem. A concept is presented how to modify the clustering to do space partitioning for ray tracing acceleration.

This work was supported by the following projects:

Czech Science Foundation (GACR), project 201/09/0097

Ministry of Education, Youth and Sports, project 2C 06002 (VIRTUAL)

Ministry of Education, Youth and Sports, project KONTAKT 5/2005-06

Ministry of Education, Youth and Sports, project LC 06008 (CPG)

Contents

Chapter 1

Introduction

In my PhD study I research algorithms for manipulation with large geometric data. I believe it is a prospective field because computer graphics deals with large data in many applications, ranging from rendering high definition photorealistic images for film industry, to visualisation of scientific data from complex measurements and simulations in geography or medicine. More about large data in computer graphics can be found in Chapter 2.

As technology goes forward, the amount of data that is to be processed also increases. We have larger memories, more powerful processors, can acquire and transmit data faster. But we can always produce more data than our computers can reasonably process. Thus new techniques are being developed to handle such large amounts of data. Since the beginning of computers, there are data that by far exceed the amount of available main memory. The processing is therefore done either online without storing all the data, or so called out-of-core which means using a slow (but large) external memory like an array of hard drives or tapes. Random access is extremely inefficient in such situations and even impossible for online streams. This is why data stream algorithms are now being researched extensively.

Data stream algorithms emerged in the last few years starting perhaps in 1970's. A brief history could be found for example in [117]. First streaming applications were concerned with sorting and searching. Nowadays, data stream algorithms are used in many areas for complex analyses of massive data. Many applications emerge from the great expansion of Internet. The traffic is permanently monitored to keep the network running, detect weak points and possible intrusions or abuses. A strong demand comes from marketing that needs statistics on browser clicks or user queries. This leads to another large application of data streams. Financial and banking analyses, stock market monitoring, trend tracking and forecasting. Take the exploration of natural phenomena as the last example. Large data need to be processed in astronomy, meteorological surveys or seismic observations. You can read more about data streams in Chapter 3.

So far, data stream approaches were not much employed in computer graphics. Even though it is an area that does deal with huge data and the streaming approach would be often natural. There are techniques to handle large data in computer graphics. However, not much of them could process extremely large

data that do not fit into the main memory. The streaming approach still seems to be on the edge of interest. Considering that it proved useful in many areas described above, we[1] decided to adopt the data stream approach to master large geometric data.

The target of my research is to develop algorithms that allow to manipulate huge graphics data. We decided to address this task by means of clustering. Our concept is to use clustering to identify groups with similar features in the data. Such groups can be then replaced by a single or a few representatives, thus reducing the amount of data significantly while preserving all important features. This is the major difference from ordinary sampling. Another great advantage over sampling is that we have a cluster associated with each representative. If we store the clusters in an external memory, we can later return arbitrary cluster back to the data. It is thus possible to restore selected parts of the data to the original state for a more precise examination. Another possibility is to process the clusters separately and then aggregate the results for the whole data set.

For really large data the clustering solution alone would not be sufficient. Here comes the data stream approach into account. We use a data stream clustering technique that can process gigantic data using just a small amount of memory. The cluster representatives are eventually clustered over and over until the data are reduced to a manageable size. So the data stream clustering intrinsically creates a hierarchy which is another great feature. We get a hierarchical model of the data so we can later put back clusters at various levels and thus control the level of detail. An extended description of clustering techniques including the data stream solution can be found in Chapter 4.

We are currently working on a dynamic hierarchical triangulation that utilises the cluster hierarchy. So far it is intended for visualisation but it could be used for further scientific computing as well. The program starts with a triangulation of the top level, i.e., all clusters are replaced by a representative. When a cluster is put back, its points are inserted to the triangulation so the detail is increased. Of course further clusters can be inserted up to the limit of available memory. Clusters can be later selectively removed to save memory for other data. Fundamentals of the Delaunay triangulation are discussed in Chapter 5. The dynamic hierarchical triangulation is described in Section 6.3.

Chapter 6 discusses our further contributions to the current state of the art. We made improvements to the clustering algorithm. These are rather details so please refer to Sections 6.1 and 6.2 if interested. In order to extend the potential of the clustering and the triangulation, we integrated the possibility to compute with anisotropic distance measures. We selected namely the elliptical metrics which, though relatively simple, offers a good flexibility. The work is documented in Section 6.4. Based on a positive feedback on the hierarchical clustering, we are currently developing a method that could be used for space partitioning for ray tracing acceleration. Our research so far is documented in Section 6.5.

Chapter 7 concludes this work and sketches our plans for the future work.

[1] Me and my Ph.D. study advisor.

Chapter 2

Large Geometric Data

Large data may be found in many application areas such as databases, sensor networks, network traffic monitoring or market statistics. It is also intensively studied in computer graphics. This chapter gives a general overview of selected methods for large geometric data acquisition and manipulation. Especially for the manipulation, there are many profoundly different approaches. This chapter mentions the fundamental techniques and gives a basic overview of their function. Following chapters concentrate on selected particular methods in detail. It is to be noted that this work focuses on large data in computer graphics, especially data of a geometric character. Other areas such as video processing are not covered in the text.

The term of *large data* has a continually evolving meaning as new technologies are discovered and brought to practise. This applies both to data acquisition and manipulation. The Stanford 3D Scanning Repository [137] offers a good example. The famous Stanford Bunny was scanned in 1994. With its 36 000 vertices it was considered quite a large model. In 1999 several Michelangelo's statues were scanned, including the Atlas with about 250 million vertices.

Today geometric models are often even larger. Let us mention for example terrain models. Today, even the whole world is available in digital form [93, 139]. The digital elevation map has a size of 1.9 GB, the resolution is about 900 m per pixel. Visualisation of large detailed models is required in medicine, for example in The Visible Human Project [141]. The data from the year 2000 contain 58 GB of high resolution images. Further large models can be found in industry, see for example The Walkthru Project [143]. The model of the Double Eagle Tanker consists of 82 million triangles. And we must not forget the film industry and computer games.

2.1 Large data acquisition

A lot of large geometric models come from scanning real-world objects. Today it is most often done using a laser scanner. The technology is called LIDAR (LIght Detection And Ranging). It is an optical scanning technology that emits laser pulses and detects the reflected light. The principle is common with a radar with the difference that LIDAR uses light instead of radio waves. This

way it is possible to capture a full 3D shape of virtually any object. It is even possible to mount such a device on an aeroplane and make a large detailed scan of Earth's surface. Satellite photography is used to record wide areas. Other scanning methods producing large data could be found in medicine, namely the computer tomography and the magnetic resonance. A typical scan could have $256 \times 256 \times 256$ or $512 \times 512 \times 512$ samples.

There are also applications working with synthetic data generated by the computer. These include various simulations, for example the finite element method. Another area is concerned with computing statistics from large data. The data often comes from real-time measuring of network traffic, monitoring market transactions, etc. The input arrives continuously over time and is often too large to be processed completely and exactly, so approximate algorithms must be employed. It is not the common case for geometric data so this problem is not further discussed in this text.

2.2 Large data manipulation

This section summarises different approaches to the manipulation with large geometric data. The list of the methods mentioned here has no ambition of being complete. The most relevant techniques are discussed and their fundamental principles are explained. Nevertheless, there are many other special approaches. Detailed description of every method would be beyond the scope of this document.

2.2.1 Processing the entire data set

In some cases it is necessary to process the entire data set; nothing could be omitted. If the data set is too large, the only possibility is to process it in pieces. There are generally two possible approaches to this task. The first one is to use a parallel and/or distributed computing to distribute the data among several (or several thousands) processing units. More on parallel and distributed processing of geometric data could be found for example in [1, 89]. Another approach also processes the data in pieces but on a single computer. This is the first step to the so called data stream approach. Streaming algorithms are discussed in detail in Section 3.

2.2.2 Reducing the amount of data

There are such situations when it is not necessary to deal with all the data at the same time, so it is possible to consider just a subset of the original data. This is mostly the case of visualisation or performing some local computations on the data. Generally, there are two ways how to reduce the amount of data. The first approach defines a region of interest, takes only that part of the data and discards everything else. It keeps all information about a limited region. Like if you take a single page (or several individual pages) of a large map. This technique is well suitable for both visualisation and local computing. The second approach takes the data as a whole but in a lower resolution. It keeps

the most important information about all regions. Like if you take a map with a lower ratio scale. This technique is particularly suitable for visualisation.

Clipping and culling

Selectively discarding unnecessary data is basically done by *clipping* and *culling*. These techniques differ in the way how particular regions are selected to be discarded or not. See Figure 2.1. Clipping removes those regions that are out of the current scope. For example, when you are virtually overlooking a digital terrain, you do not need to render what is behind your back. Culling removes those parts that are unimportant from the nature of the data. Back to the example with a terrain, when looking at a hill, you cannot see what is behind. This is called occlusion culling. You even cannot see the other side of the hill itself. This is the back face culling.

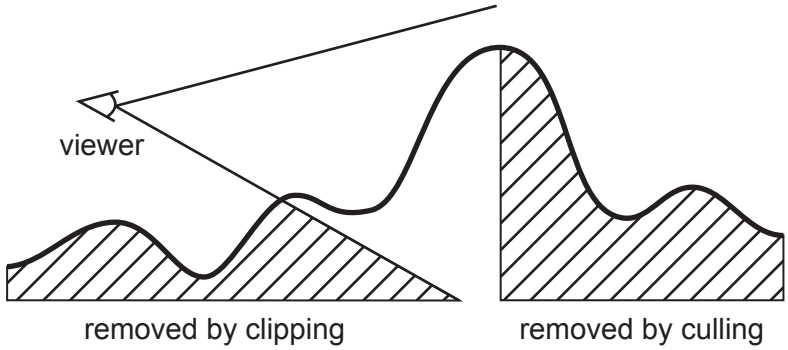


Figure 2.1: Parts of a scene removed by clipping and culling.

Various methods exist to speed up both clipping and culling. Those fundamental include space subdivision – quadtrees [45, 30], octrees [18] and kD-trees [8, 31]. More advanced methods, such as the binary space partitioning [47, 32], are used especially for the culling. A detailed description of such techniques would be beyond the scope of this work.

Simplification

Even after applying clipping and culling the amount of data may be still too large. Simplification is used in such cases. It is a way how to handle all the required data at the cost of reduced quality. This technique is especially useful in visualisation applications.

The simplification involves removing insignificant detail from the model. The fundamental approach by Shroeder et al. [131] uses the technique of *vertex removal*, also referred to as *vertex decimation*. It simplifies the model by successively removing a vertex and patching the resulting hole as illustrated in Figure 2.2. To avoid degeneracies in the model it is necessary to check whether the vertex removal would not change the topology of the model. Preserving the topology is a valuable property for example in medical applications.

Turk [138] proposed an interesting method that could be named mesh resampling. Unlike most other methods, this one generally does not use any of

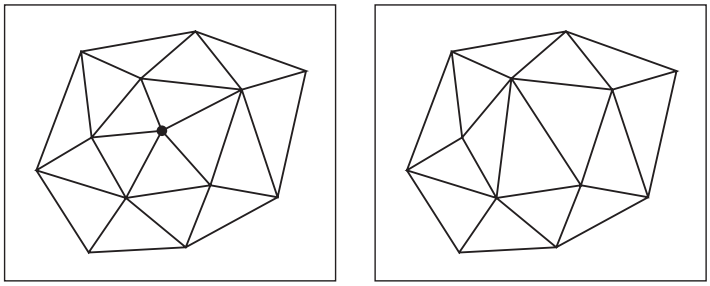


Figure 2.2: Example of a vertex removal. The marked vertex has been removed.

the original vertices. It places random samples on the mesh surface instead. The sampling could be either uniform or with increasing density in the areas of higher curvature. Next a new mesh must be constructed from the samples. Turk uses a very elegant and robust method. First the newly sampled vertices are inserted into the original mesh. This is done by simple triangle subdivision since the new vertices has been sampled from the mesh surface. The original vertices are then successively removed similarly as in decimation. The method guarantees that the topology will be preserved. Turk further presents how to interpolate between the resampled models.

Another method is the *edge contraction* or *edge collapse*. It replaces two incident vertices by a single one. All the edges that were connected to both original vertices are connected to the new vertex. Figure 2.3 shows an example. Edge collapse is an essential part of the algorithm by Hoppe et al. [69]. They

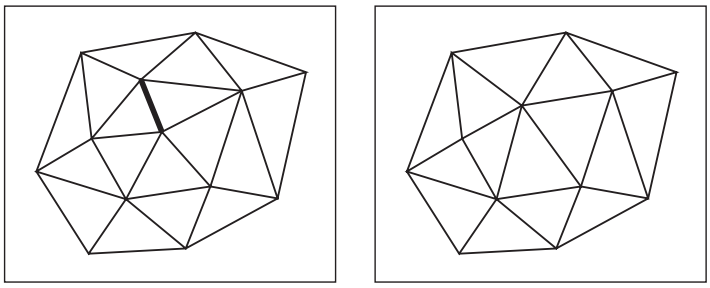


Figure 2.3: Example of an edge contraction. The marked edge has been contracted.

define an energy function that models the competing requirements of compact representation and geometric fidelity to the original mesh. The simplification is then solved as an optimisation problem to minimise the energy function.

After the edge contraction the question is where to place the new vertex. Simple algorithms use the midpoint between the removed vertices. Advanced techniques try to minimise the error incurred. This leads to the question of error measure. Simplification algorithms need to evaluate the error incurred by removing a particular vertex or contracting an edge, so as to decide which vertices to remove or which edges to contract. The error of removing a single vertex is often measured as a distance of the vertex from an average plane of its neighbours. For edge contraction, the distance between the involved vertices is used. Alternatively, the change in object volume may be measured. The vertices

are placed into a priority queue according to their associated error which is continuously updated as the simplification proceeds. Vertices are simplified in the order of minimal error until the model is reduced to a desired size or until an error threshold is reached.

Garland and Heckbert [50] proposed an analogous method to the edge contraction – the *pair contraction*. It differs in that it can merge any two vertices, being incident or not. This way the topology may change dramatically and even independent objects may be joined together. This yields nice results when a drastic simplification is required. The algorithm uses quadric error metrics to evaluate possible contractions. The error is computed as a quadratic form which allows to compute the right replacement for the two contracted vertices and generally achieves good quality simplifications.

Cohen et al. [22] proposed a technique of *simplification envelopes*. It is a general framework within which various existing simplification algorithms can run. Simplification envelopes are a generalisation of offset surfaces. They allow to generate mesh approximations that are guaranteed not to deviate from the original mesh by more than a pre-specified amount. Precisely speaking, all vertices of the simplified model will be within a distance ϵ from the original and vice versa. Topology is also guaranteed to be preserved. The algorithm surrounds the original mesh with two envelopes and then performs simplification within this volume. The envelopes are constructed by offsetting each vertex of the original mesh in the direction of its normal and in the opposite direction by ϵ. If should any self-intersection occur, the offset ϵ is reduced so as to avoid that. The authors present two methods for computing the envelopes as well as two simplification algorithms that can actually be used within the framework. Simplification envelopes inherently ensure that sharp edges will be preserved.

The simplification envelopes are particularly associated with the Hausdorff distance which is commonly used to measure the difference between the original and the simplified model. Let X and Y be two point sets representing some objects and let d be any metric. The (directed) Hausdorff distance *from* X *to* Y is defined as

$$d_h(X,Y) = \max_{x \in X}\{\min_{y \in Y}\{d(x,y)\}\} \tag{2.1}$$

It is the maximum of distances from any point $x \in X$ to the closest point $y \in Y$. The (symmetric) Hausdorff distance *between* X and Y is then defined as

$$d_H(X,Y) = \max\{d_h(X,Y), d_h(Y,X)\} \tag{2.2}$$

Simply speaking, the Hausdorff distance is the greatest local difference between the two objects.

2.2.3 Large data visualisation

This section describes techniques of large data visualisation. Many of the above mentioned approaches could be used for visualisation as well. But they are rather general and image rendering is just one of their possible applications. The algorithms described in this section were developed primarily for the visualisation purposes.

Level of detail

The idea of using a simpler representation of objects to improve rendering frame rate was first proposed in [21]. Level of detail techniques [102] are now often employed to render complex scenes efficiently. Not all objects present in the scene need to be rendered at full resolution. Distant objects are too small for the fine detail to be visible. Similarly, fast moving objects do not need to be rendered in high detail. Therefore simplified models are used to reduce system load. However, if an object slows down or gets close to the viewer, a more detailed model should be used. So the rendering system must be able to dynamically select a model in the appropriate resolution, thus control the level of detail. Figure 2.4 shows a model at several different resolutions.

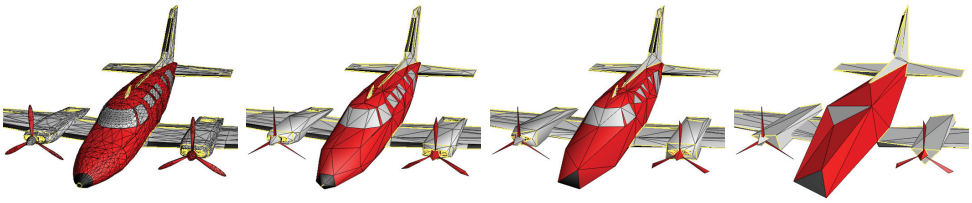


Figure 2.4: Example of a model at four different levels of detail. Images adopted from [67].

In earlier times, lower resolution models were prepared by hand. This had the advantage of perfect quality. The modeller person knows best which details should be preserved and which can be dropped. But the handwork is rather slow and expensive so automatic methods are used nowadays. The handmade models usually had discrete levels of detail. It means that there was the original finest model plus two or three simplified ones. Such models can be easily handled but the so called popping effect may appear when switching between particular levels of detail. It means that the substitution of simplified models could be noticeable for the viewer and often even disturbing. Automatic methods [100, 130] can generate smooth transitions by gradually increasing or decreasing the amount of detail in the model. Sophisticated view-dependent methods [68, 97] can even change the detail depending on the view direction, keeping high detail only in the near parts and on the silhouette where fine details are well visible.

Most simplification algorithms can be used for the level of detail. Among them the technique of *progressive meshes* [67] is especially remarkable. The images in Figure 2.4 come from a progressive mesh. The goal is to find such a mesh M that both accurately represents the original object and has a small number of vertices. This can be expressed as a minimisation of an energy function

$$E(M) = E_{dist}(M) + E_{rep}(M) + E_{spring}(M) \tag{2.3}$$

The E_{dist} term measures the distance of the mesh from the original, the E_{rep} term is a penalisation for the number of vertices, and the E_{spring} term is to regularise the optimisation problem. Please consult [67] for more details.

The algorithm is based on [69] though it uses a modified energy function and performs only edge collapses (no edge splits or edge swaps). The key is that an edge collapse is invertible. The inverse transformation is called a vertex split which adds a new vertex and two faces. A mesh is then stored in a much coarser version together with a sequence of vertex splits. They indicate how to incrementally refine the mesh exactly back into the original mesh. The representation thus defines a continuous sequence of meshes of increasing accuracy, from which approximations of any desired complexity can be retrieved. Moreover, geomorphs can be constructed between any two such meshes. A geomorph is basically a linear interpolation between two models. It makes very smooth transitions between them. Progressive meshes naturally support progressive transmission, they permit selective refinement and could be used as a mesh compression.

Point based rendering

Points as rendering primitives were first discussed in [98]. The point based rendering got an increased popularity later [56, 127, 11]. A comparison of various point based rendering techniques may be found in [129]. As the name suggests the point based rendering uses points or similar very simple primitives to render complex 3D models. Every point is defined by its coordinates and preferably has also a normal vector corresponding to the surface from where it was sampled. Optionally, the point colour may be specified. Unlike conventional triangle meshes, it works with just the points without connectivity. It does not require any information about point adjacency. So there is no need for a demanding and possibly unstable triangle mesh construction. This spares a lot of time as well as memory because there is no need to store the mesh. For large models, the problem is not only to compute the reconstruction. The problem is even to hold the whole triangle mesh in the memory.

The point based rendering displays just individual points. It is therefore necessary to render the points large enough so that they overlap and there are no gaps between them. Otherwise the rendered model would have holes in its surface. If the rendering is correct, points fill the whole surface smoothly as seen in Figure 2.5.

The most common primitives are indeed points. Do not confuse points with pixels. A point is a spot that could be rendered several pixels large. Using programmable graphics, the point size can be adjusted according to the actual projected screen size. Rendering a spot is of course much easier then rasterising a triangle, evaluating the lighting model in all three vertices and interpolating the colour. Sprites[1] are more flexible primitives in that a texture can be mapped on them. The most complicated primitives are quads or triangles, still with no connectivity among them. Textures with alpha channel are mapped onto them so points can be rendered for example as disks oriented according to the surface normal.

Rendering points is simple and allows to display even large models. But it is still profitable to employ acceleration techniques such as an octree [90].

[1] Small 2D images integrated into a 3D scene.

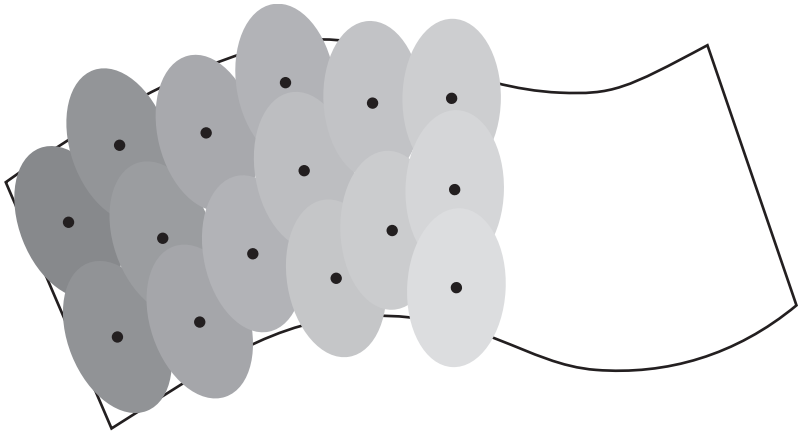


Figure 2.5: The point based rendering. Overlapping points covering a surface.

Additionally, it allows to control the (view dependent) level of detail, do the view-frustum and the back-face culling [120]. The octree is traversed as usual. If a node lies outside of the view frustum, or if a node is determined as back-facing, or if the screen-projected size of a node is smaller than a threshold, then the sub-tree under the node is not taken into account.

Vertex clustering

Rossignac and Borrel [126] proposed a simplification method that uses vertex clustering to render complex scenes. Perhaps a more precise expression would be vertex quantisation. The scene is uniformly divided by a grid. The grid resolution controls the amount of simplification. Vertices falling into one grid cell form a cluster which is then replaced by a single representative vertex. This could be either the centre of mass of the cluster or the most significant vertex.

The significance of vertices is evaluated by their visual importance. Important vertices are those that

1. have a high probability of lying on the object's silhouette from an arbitrary viewing direction

2. bound large faces that should not be affected by the removal of small details

The first criterion may be efficiently estimated by the inverse of the maximum angle between all pairs of incident edges. The second criterion may be estimated by the longest edge incident to the vertex.

The notable property of the algorithm is that it does not simplify objects separately. If two objects are close together so that some of their vertices fall into a single grid cell, the objects will be merged together. Manifold topology is not required and not guaranteed to be preserved.

An example of the method results can be seen in Figure 2.6. The top image shows an object vanishing into a distance with decreasing level of detail. The bottom image shows all the instances at the same size.

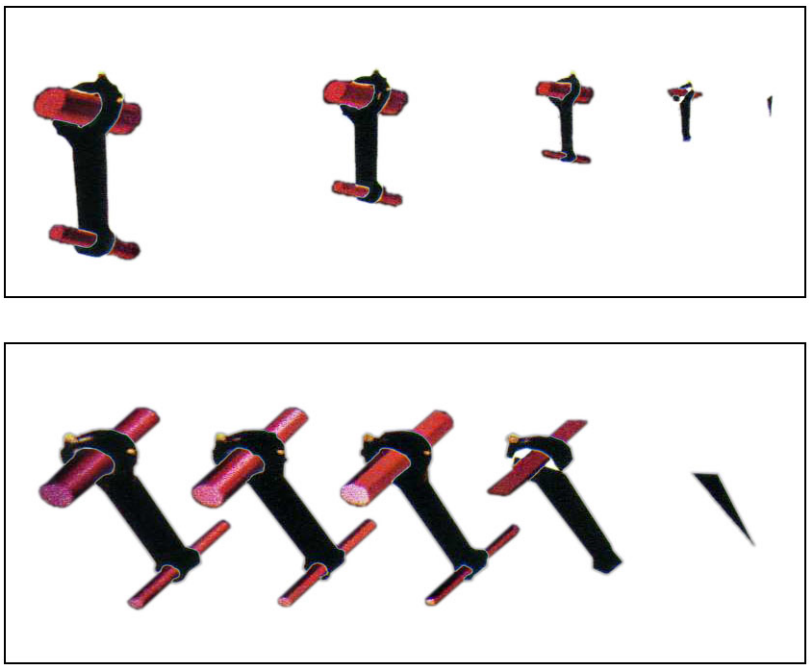


Figure 2.6: Examples of different level of detail obtained by the clustering. Image adopted from [126]; colours adjusted.

Luebke and Erikson [101] proposed a hierarchical dynamic simplification. It is a similar but more advanced approach. It also uses vertex clustering (but can use any other simplification algorithm) to create the so called *vertex tree*. The tree defines how vertices (possibly from different objects) will be merged together to simplify the scene. Each node in the tree contains one or more vertices. The algorithm may collapse all of the vertices within a node into a single representative vertex. Triangles whose corners have been collapsed together are removed. Likewise, a node may be expanded by replacing the representative vertex by the original node's vertices. The triangles that were removed become visible again. The vertex tree is queried dynamically at run time to generate the desired degree of simplification. The algorithm uses a screen space error metric that measures the error in pixels and therefore perfectly reflects the distortion that is actually visible.

Chapter 3

Data Streams

The concept of data streams is versatile and can be used in an extremely wide area of applications. Typical tasks include detecting outliers, extreme events, intrusions, track trends and perform analyses. Data streams are often found in connection with computer networks [85, 94, 146], namely monitoring the traffic or computing statistics of browser clicks and user queries. More applications include astronomical [48], satellite and meteorological surveys, financial observations [48] such as stock exchange and currency trades. An increasing interest emerges also in computer graphics [70], particularly in computational geometry [44, 121, 74, 78, 75].

The following section starts with an informal definition of data stream and an on-the-surface discussion of diverse concepts adopted in different disciplines. Classical model problems are presented. Although simple, they can be inspiring for solving practical tasks. Fundamental techniques frequently used in data stream processing are introduced. The second section focuses on data streams in computer graphics.

3.1 Data stream fundamentals

From a general point of view, data stream is a sequence of data. Every application branch then has different specifications, expectations and limitations posed on the stream. A comprehensive overview can be found in [117]. There are essentially three challenges that may be concerned – to transmit the data to the program, to compute sophisticated functions on large pieces of input in an acceptable time, and to store the presented information long-term. Either of the tasks may be demanded, or all of them.

In most cases the amount of data in the stream is extremely large so it is hard to store it or to compute complex functions by conventional algorithms. Let N be the length of the stream, i.e., the number of distinct pieces of information, which is often supposed to be known. Data stream algorithms are usually allowed to use $\mathcal{O}(N^a)$, $a < 1$, or $\mathcal{O}(\log N)$ memory.

The data stream can be stored on hard drives or tapes. This is the offline variant [78]. Processing time is not critical and it is possible, though discouraged, to do multiple passes over the data. Random access is prohibited. Offline

data streams allow to compute complex analysis on them. This is a common case in computer graphics. By contrast, in online data streams [55] data arrive at a very high rate so it is hard and usually impossible to process them exactly. Approximate algorithms must be used. This is a common scenario in computer networks.

3.1.1 Motivation puzzles

This is a list of typical problems being solved in the data stream model. Many real world tasks can be directly mapped to them. The techniques *how* to solve them are discussed in Section 3.1.3. For more details see for example the nice survey [117].

A traditional task is finding missing numbers. Let π be a permutation of $\{1 \dots n\}$ and let π_{-1} be π with one number missing. Paul shows Carole a stream consisting of $\pi_{-1}[i]$ in the increasing order of i; Carole's task is to determine the missing number m. Of course she is not allowed to memorise all the numbers. This is a rare data stream problem that can be solved exactly. Carole maintains a sum of the presented numbers $\sum_{j=1}^{i} \pi_{-1}[j]$. The sum of all the numbers $\{1 \dots n\}$ is easy to compute as $\frac{N(N+1)}{2}$, so the missing number can be computed as

$$m = \frac{N(N+1)}{2} - \sum_{j=1}^{N-1} \pi_{-1}[j] \tag{3.1}$$

Variants exist for more missing numbers [117]. Let π_{-2} be a permutation with two missing numbers m_1 and m_2. Carole now needs to maintain more information. A simple solution would be to maintain a sum and a product of the presented numbers. Carole then gets

$$m_1 + m_2 = \frac{N(N+1)}{2} - \sum_{j=1}^{N-2} \pi_{-2}[j] \tag{3.2}$$

$$m_1 m_2 = n! - \prod_{j=1}^{N-2} \pi_{-2}[j] \tag{3.3}$$

which is a simple system of equations. However, Carole can use far fewer bits by tracking the sum and a sum of squares. The solution is then obtained from the equation system

$$m_1 + m_2 = \frac{N(N+1)}{2} - \sum_{j=1}^{N-2} \pi_{-2}[j] \tag{3.4}$$

$$m_1^2 + m_2^2 = \frac{N(N+1)(2N+1)}{6} - \sum_{j=1}^{N-2} (\pi_{-2}[j])^2 \tag{3.5}$$

This scheme can be extended to k missing numbers.

Further tasks commonly solved in data streams are counting distinct elements and detecting duplicate values [52, 6, 23]. Another problem is to find the majority value [15, 26] which is the value repeated most frequently. Closely

related are the so called iceberg queries [107, 51, 14, 84] looking for items that are repeated more frequently then a specified threshold. A common task is to compute a histogram [17, 58, 53] or quantiles [55].

3.1.2 Data stream models

There are three traditional data stream models [117] that can be applied to a majority of real world problems. Let $a_1, a_2, \ldots$ be the input stream that describes an underlying signal A, suppose $A : [1 \ldots N] \to \mathbf{R}$ for simplicity. Models differ in how the stream describes the signal. In the *time series model* $a_i = A[i]$, i.e., a_i present absolute values of the signal in the increasing order of i. This is suitable for observing some quantity at regular intervals.

In the *cash register model* a_i are increments to $A[j]$. Let A_i be the state of the signal after seeing i items of the stream. Let $a_i = (j, I_i)$, $I_i \geq 0$, then $A_i[j] = A_{i-1}[j] + I_i$. Multiple a_i can increment the same $A[j]$ over time. The cash register model is suitable for instance for web server access monitoring because a single client may access the server multiple times.

The *turnstile model* differs in that a_i are updates to $A[j]$. Let $a_i = (j, U_i)$, $U_i \in \mathbf{R}$, then $A_i[j] = A_{i-1}[j] + U_i$. This model is appropriate for dynamic systems with inserts and deletes, but it is often hard to find good solutions. A modification is the *strict turnstile model* where $\forall i : A_i[j] \geq 0$. The model is suitable for example for a database where it is not possible to delete an item not present.

There are more advanced models for special situations. *Permutation streaming* is a special case of the cash register model in which items do not repeat. The input stream is a permutation of some set and the items arrive in an unordered way. The task may be to estimate various permutation edit distances [27] or to estimate the number of inversions in the permutation [2, 64]. This could be used for example to detect packet retransmissions in a network traffic.

In the case of a *synchronised streaming* [117] a function is to be computed on two signals given as streams, e.g., estimating the edit distance of two strings. *Windowed streaming* [29] is used to emphasise the recent history of the stream. A sliding window of width w is defined and the algorithm focuses only on the data within the window, i.e., the most recent w items from the stream. This is a common practise for example in fraud detection. A variant [28] of the model is to age the input signal and to consider older records with a lower significance. Another variant [24] considers recent items of the stream at a fine granularity while the distant past is aggregated.

3.1.3 Typical data stream techniques

Many data stream algorithms are based on one of several fundamental techniques. *Shedding* is the simplest, often used on very fast and massive online data streams. It just blindly samples the data at a lower frequency. True *sampling*, as it is used in data streams, carefully selects which data to retain and which to drop [17, 55, 52, 107]. The *sliding window* has already been mentioned in Section 3.1.2.

Advanced techniques such as *sticky sampling* and *lossy counting* [107] are used for example to determine how frequently particular elements occur in the stream. Sticky sampling draws a sample set of distinct elements and counts how many times each of them appeared. The sample set is continuously refreshed to drop sporadic elements and to include new ones appearing in the stream. Lossy counting is a bucketing technique, i.e., it divides the stream into pieces (buckets) and processes them one after another. It counts all distinct elements in a bucket. Before proceeding to the next one, all the counters are decremented and elements whose counter reached zero are dropped. The *sketching* technique [53, 15, 23, 25, 26, 73] computes summary information on the stream. It allows various queries to be approximately answered very quickly.

3.2 Data streams in computer graphics

With an increasing size of geometric models, data stream approaches are getting into consideration especially in computational geometry [72]. Isenburg et al. [78] proposed a streaming computation of Delaunay triangulation. Their method is described in detail in Section 5.3.2. This section focuses on two more applications of data streams [74, 121]. The streaming approach is not much used in computer graphics so far. There are several more streaming algorithms proposed by Isenburg et al. [76, 77, 75]. They often rely on the data to be approximately sorted. Otherwise, some external pre-sorting must be done.

3.2.1 Streaming meshes

Geometric models are usually stored in the common format comprising of a list of vertices followed by a list of triangles. There is no ordering required so a triangle may reference vertices from both ends of the vertex list, as well as a vertex may be referenced by triangles from both ends of the triangle list. It is especially notable for instance at the Stanford bunny [137] where both triangles and vertices are heavily scattered. This bad topological coherency is not well suitable for huge gigabyte-sized models since any processing algorithm would need random access to either of the lists. It prohibits the stream processing and makes cashing inefficient.

Isenburg and Lindstrom proposed *streaming meshes* [74] to overcome this problem. The idea is to interleave vertices and triangles so that some algorithm processing the mesh will come to the right vertices when they are needed. Vertices that will not be needed anymore are *finalised*, i.e., marked that they can be freed from memory. An algorithm processing an interleaved mesh can thus hold just the vertices that are actually needed. It uses little memory and does not need random access to the whole mesh.

Isenburg and Lindstrom suggest two variants of streaming meshes. The pre-order format introduces a vertex just before the triangle that references it for the first time. The vertex is finalised as soon as it is referenced by the last incident triangle. The post-order format first introduces triangles. As soon as all triangles incident to a vertex are present, the vertex is introduced and implicitly immediately finalised.

Many mesh generating applications work in such a way that the output can be easily stored in the streaming format. Authors also propose methods how to reorganise existing models.

3.2.2 Stream-processing points

A framework for stream-processing points was proposed by Pajarola [121]. The input is a point cloud sorted along one direction. The algorithm then sweeps the data along that direction and processes them as a stream. Points are sequentially loaded into the memory where *stream operators* (see bellow) are applied to them. Many operations such as normal computation cannot be performed on isolated points. Therefore a *working set* is maintained where points accumulate and their local neighbourhood is available. As soon as a point cannot contribute anymore to any operation, it is released from the working set and written to the output. Eventually, a buffer and a deferred write may be used to preserve the global stream ordering.

Pajarola introduces *stream operators* as functions computed on a single point using only a local neighbourhood. They are applied to the points that have all necessary neighbours present in the working set. Pajarola introduces several elementary operators that are particularly important for processing raw point clouds. Except the elementary I/O operators, they are k nearest neighbours, normal computation, curvature estimation, splat size estimation (for the point based rendering; see Section 2.2.3), and fairing (smoothing). The power of the stream-processing framework consists in that the stream operators can be concatenated and thus a complex computation on the data can be performed in a single pass. The operators are implemented as modules with input and output buffers. A module starts computing when it has enough data in the input buffer. Processed points are passed to the output buffer. This way modules hand over the data efficiently as the computation proceeds.

Chapter 4

Clustering

Over the years, clustering has evolved into a really general concept. It is used in various data analysis [37, 5], data mining [43] (specially mining large databases [118, 149]), information retrieval [124], image segmentation [79], pattern recognition [7] and other scientific fields [81, 82]. It can also be found in non-technical disciplines such as archeology or marketing.

Clustering means, in principle, grouping similar elements together. Depending on the area of application, elements could be anything – from points in 1D space[1] to 3D objects, entire images, documents or data base entries. Each element is described by a characteristic vector. Points are specified by their coordinates. For more complex elements some abstraction is needed to get a suitable representation. This work focuses on clustering of points in 3D. We are going to investigate also more complex objects, surely triangles. The future work is outlined in Section 7.2.

A majority of clustering tasks are NP-hard problems so most algorithms produce only approximate results.

A comprehensive overview of clustering can be found for example in [38, 65, 37, 81, 82, 108]. This chapter will briefly introduce the main concepts of clustering. It will then concentrate on the facility location problem and methods for clustering large data.

4.1 Distance measures

To measure a similarity between elements, it is necessary to determine a distance measure. By far the most common is the Euclidean distance. Given two elements $\mathbf{x}_i = \{x_{i1}, x_{i2}, \dots x_{iD}\}$, $\mathbf{x}_j = \{x_{j1}, x_{j2}, \dots x_{jD}\}$ in D dimensional space, the Euclidean distance is defined as

$$d(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{(\mathbf{x}_i - \mathbf{x}_j)^2} \tag{4.1}$$

It works well for compact spherical clusters. Different measures are used for special applications, e.g., to detect clusters of a complex shape or for pattern

[1] Clustering the depth information for rendering.

matching. The Minkowski metric is a generalisation of the Euclidean distance and is defined as

$$d_p(\mathbf{x}_i, \mathbf{x}_j) = \sqrt[p]{(\mathbf{x}_i - \mathbf{x}_j)^p} \tag{4.2}$$

Anisotropic metrics can be used to find clusters of non-spherical shape. This is discussed in detail in Section 6.4. The Mahalanobis distance [105] can be used if there is a correlation between elements of the characteristic vector. It is suitable for classification of elements into classes. A covariance matrix Σ of each class is computed from known samples. The Mahalanobis distance between an element $\mathbf{x}_i$ and a class with a mean $\mathbf{x}_j$ is then defined as

$$d_M(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{(\mathbf{x}_i - \mathbf{x}_j)^{\mathrm{T}} \Sigma^{-1} (\mathbf{x}_i - \mathbf{x}_j)} \tag{4.3}$$

The Hausdorff distance [116] can be used to measure the distance between sets, e.g., for point set matching. Given two sets $\mathbf{A}$ and $\mathbf{B}$ and an arbitrary metric d, the (oriented) Hausdorff distance *from* $\mathbf{A}$ *to* $\mathbf{B}$ is defined as the maximal distance from an arbitrary point $a \in \mathbf{A}$ to the closest point $b \in \mathbf{B}$. Written in the form of equation

$$d_h(\mathbf{A}, \mathbf{B}) = \max_{a \in \mathbf{A}} \{ \min_{b \in \mathbf{B}} \{ d(a, b) \} \} \tag{4.4}$$

From the properties of minimum and maximum operators follows that the distance is not symmetric. The general Hausdorff distance *between* $\mathbf{A}$ and $\mathbf{B}$ is defined as

$$d_H(\mathbf{A}, \mathbf{B}) = \max\{ d_h(\mathbf{A}, \mathbf{B}), d_h(\mathbf{B}, \mathbf{A}) \} \tag{4.5}$$

Following sections assume the use of the Euclidean distance. It is the most common measure, very simple and works well in many scenarios.

4.2 Clustering approaches

A large amount of clustering techniques exist. This section gives an overview of various different approaches. The most common algorithms are described in more detail. For further information please refer for example to [82, 108].

Clustering algorithms can be divided, according to some particular principle of their function, into two opposed courses. Following paragraphs list several such divisions.

Clustering algorithms can be either *partitional* or *hierarchical*. Partitional approaches divide the data into an exact number of clusters (partitions). Hierarchical algorithms create a hierarchy of small clusters grouped into larger clusters forming a tree structure called dendrogram. The degree of clustering can be than controlled by going up and down in the hierarchy. Figures 4.1 and 4.2 show examples of partitional and hierarchical clustering of the same data. The dashed line in the dendrogram in Figure 4.2(b) shows from which level of the hierarchy the clustering in Figure 4.2(a) was created.

Another possible division is to *agglomerative* and *divisive* or *partitional* algorithms. An agglomerative approach [95, 87, 136] starts with each element in a single cluster. These are then successively merged according to a similarity

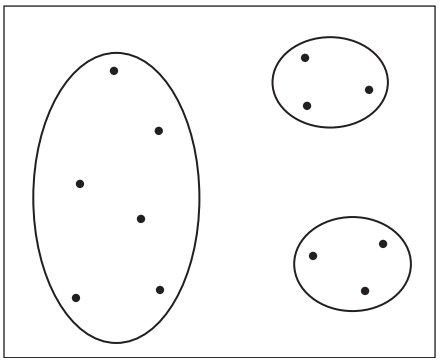
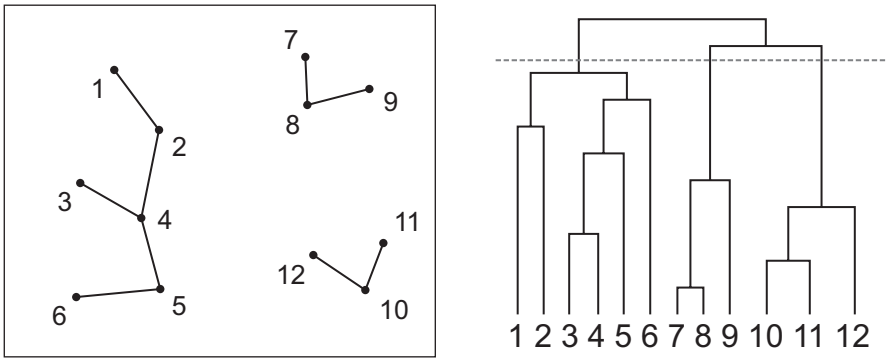


Figure 4.1: An example of a partitional clustering.



(a) example clustering **(b)** dendrogram

Figure 4.2: An example of a hierarchical clustering with a dendrogram on the right.

measure until a stopping condition is met. The algorithm usually stops when a desired number of clusters has been reached, or when there is such a low similarity between existing clusters that no further can be merged. A divisive approach [104, 5] works vice versa. It starts with all the data in one large cluster. It is then repeatedly split according to a dissimilarity measure. Again the algorithm stops when there is a predetermined number of clusters or when existing clusters are homogeneous enough so that no further splits are necessary.

The clustering can be either *hard* or *fuzzy*. Hard clustering [104] assigns each element into exactly one cluster. Fuzzy set theory was first applied to clustering in [128]. Fuzzy clustering [10, 9] determines a degree of membership in several clusters for each element. It is then possible to get a hard clustering by assigning each element into the cluster with the largest membership value.

Clustering algorithms can be *deterministic* or *stochastic*. Stochastic techniques [104] usually use some random search among all possible solutions. Such algorithms do not always yield an optimal solution but often guarantee a constant factor approximation. A great benefit is that randomised algorithms mostly run rather fast and thus are perfectly suitable for processing large amounts of data.

The clustering method may process the data all at once or work incrementally. If the algorithm works with all the data together it can in principle

produce more precise results. Incremental algorithms [60] can be faster and have much lower memory requirements which is again a great benefit when processing large data. The low memory requirements come from the fact that the algorithm does not need to store all information about the data processed so far. It is only necessary to hold summary information about particular clusters which can be orders of magnitude smaller than all the data contained in them.

The notion of cluster representation was introduced in [38] and was subsequently studied in [36, 114]. There are two major cluster representations – *centroid based* and *sample based*. The centroid based approach stores just the centre of each cluster and possibly the number of elements contained within. This is a very compact representation but completely sufficient for many applications. The sample based approach stores several well chosen representatives for each cluster, e.g., points at the border of the cluster. This requires more memory but also gives more information in particular about the cluster shape and possibly about how elements are distributed within the cluster. Figure 4.3 shows the difference between the centroid based representation (on the left) and the sample based representation (on the right). There is a single cluster in both figures. Crosses show the summary information stored about the cluster.

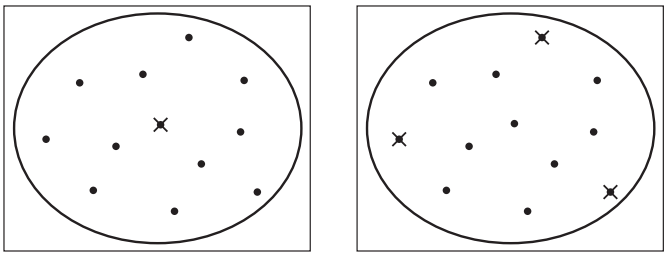


Figure 4.3: The centroid based cluster representation on the left, and the sample based representation on the right.

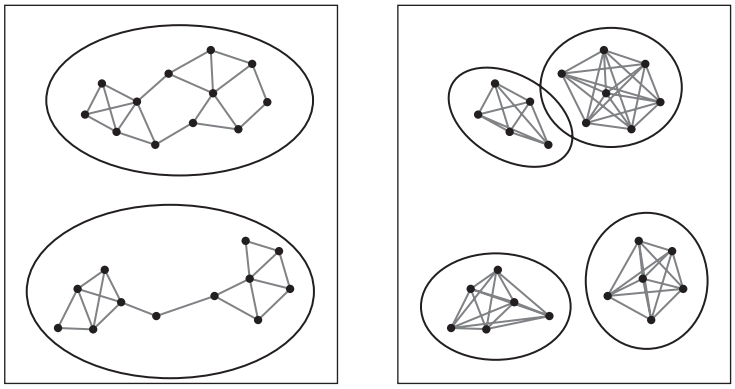
Following sections describe some of the most common clustering algorithms in detail. Namely two examples of the hierarchical agglomerative clustering – the single-link and the complete-link algorithm. Then the famous k-means algorithm and finally the facility location. Except these, a vast amount of other methods exist. Many of them are modifications and/or combinations of the methods described here.

Further, there are different approaches based on genetic algorithms [54], simulated annealing [125], tabu search [3] or artificial neural networks [80]. They have good theoretical properties. Evolutionary approaches (genetic algorithms and simulated annealing) are globalised search techniques. This ensures that they do not remain stuck at some local optimum. All of the four approaches give solutions very close to optimum. Unfortunately, these methods do not perform so well in practise. They are sensitive to the initial setting of learning/control parameters which is impractical. Neural networks are often order-dependent which means that they can produce different results for different input ordering. Perhaps the biggest problem of all the methods is that they are too slow, especially genetic algorithms which also require a lot of memory. They are thus used only for small data sets of about hundreds or thousands of elements.

4.2.1 Single-link and complete-link

Single-link [136] and complete-link [87] are hierarchical agglomerative clustering approaches. They are non-incremental, deterministic and do the hard clustering. They differ in the way they measure the similarity between clusters. The single-link algorithm defines the distance between two clusters as the *minimum* pairwise distance between the elements of the two clusters, i.e., the similarity of the clusters is measured as the similarity of their *most similar* members. By contrast, the complete-link algorithm uses the *maximum* pairwise distance, i.e., the similarity of the clusters is measured as the similarity of their *most dissimilar* members.

The single-link algorithm is more versatile but tends to produce straggly or elongated clusters. This could be unpleasant but in some scenarios it is very useful to detect non-spherical clusters. The complete-link algorithm produces compact clusters. Which algorithm works better depends on the nature of input data. Figure 4.4 shows a comparison of both approaches. The single-link algorithm (a) correctly detects the cluster at the top but merges the two clusters at the bottom. On the contrary, the complete-link algorithm (b) separates the two bottom clusters but also splits the top cluster.



(a) single-link **(b)** complete-link

Figure 4.4: A comparison of the single-link and the complete-link approach.

A few terms of graph theory will be necessary for the following text. A *connected graph* is a graph where there is a path connecting each pair of points. A *connected component* of a graph is a maximal set of connected points such that there is a path connecting each pair. A *clique* in a graph is a set of points that are completely linked together.

The single-link algorithm can be summarised as follows

1. Start with each element in a distinct cluster. Compute distances between all pairs of elements.

2. Take these distances in an ascending order. For each such distance d form a graph where pairs of elements closer than d are connected by an edge. When all the elements form a connected graph, stop.

3. The result is a hierarchy of graphs where an arbitrary similarity level can be selected. The clusters are the *connected components* of the appropriate graph.

The complete-link algorithm basically works the same way. The difference is that the second phase is terminated when all the elements form a single clique. When a graph is selected from the hierarchy, clusters are the *maximal cliques* of the graph.

The time complexity of both algorithms is $\mathcal{O}(N^2 \log N)$, whereas the single-link can be improved to $\mathcal{O}(N^2)$. Nevertheless, both algorithms must compute N^2 distances which is the most demanding part of the computation. More on the complexity analysis can be found in [109].

4.2.2 k-means

The k-means [104] is for sure the best known clustering algorithm. It gained popularity for its simplicity, short running time and low memory requirements. It is a partitional non-incremental stochastic algorithm. It does hard clustering but there is a modification fuzzy c-means [10] that does fuzzy clustering.

The k-means algorithm works as follows

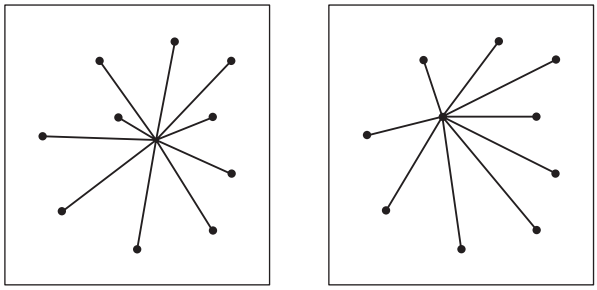
1. Choose k cluster centres either randomly or based on some heuristics.
2. Assign each element to the closest cluster centre.
3. Recompute cluster centres as centroids of particular clusters.
4. While a convergence criterion is not met, go to step 2. Typical convergence criteria are no (or minimal) reassignments between clusters or a low decrease in the squared error (i.e., a sum of squared distances to the cluster centres).

The algorithm has a time complexity of $\mathcal{O}(NkI)$ [110], where I is the number of iterations. Practical experience shows that far less than N iterations are necessary to achieve convergence. The algorithm has some disadvantages. Perhaps the most significant one is that it is necessary to determine the number of clusters k prior to starting the algorithm. It could be solved by running the algorithm several times with different settings and choosing the best result (the minimal squared error). But this considerably increases the running time. Next problem is that for some bad initial configuration the algorithm may converge to a local optimum. There are methods [110] for choosing the initial cluster centres so that the global optimum is reached with high probability.

Another approach is to allow cluster splitting and merging according to some additional criteria. This should solve both the above mentioned problems (the choice of k as well as the initial seed). One example of such an approach is the ISODATA algorithm [5]. The question is then how to balance the splitting/merging criteria.

In addition to the mentioned modifications, there are several more. Namely the k-median algorithm with the restriction that cluster centres can only be

chosen among input data elements. A comparison of the k-means and the k-median can be seen in Figure 4.5. In case of the k-median, the exact geometric centre is shifted to the closest data element. There are other modifications that use different distance measures or cluster representations [110, 10].



(a) k-means **(b)** k-median

Figure 4.5: A comparison of the k-means and the k-median algorithm.

4.2.3 Facility location

The facility location problem is a special clustering task. The general formulation is as follows. Let F be the set of so called *facilities* and C be the set of *clients* (in some literature expressed as D as for *demand nodes*). Every client should be serviced by (connected to) a facility. The problem is to determine which facilities to *open* and which clients should they service. The *facility cost* must be payed for opening a facility. The *service cost* must be payed for connecting clients to facilities (mostly based on the distance). The problem has a direct real life application. Imagine there are some cities that need to be supplied with electricity and there are several potential locations where a power plant could be built. Building a power plant everywhere would be too expensive; as well as connecting all the cities to a single one. It is to be determined where to build a power plant and where to connect particular cities. It is necessary to find such a balance to minimise the overall costs.

Expressing the problem in a mathematical way, the task is to minimise the overall clustering cost Q defined as

$$Q = \sum_{j \in F} fc + \sum_{i \in C} c_{ij} \tag{4.6}$$

where fc is the facility cost, and c_{ij} is the distance between a client $i \in C$ and its facility $j \in F$. Distances are generally considered non-negative, symmetric and satisfying the triangle inequality. It is to be noted that generally there are no restrictions on the set of facilities F. It can be independent of C, a subset of C, or equal to C. There are some specialisations of the facility location problem. Facilities may have different facility costs and may have limited capacities to service just a certain number of clients. These specialisations are not considered in the following text.

To compute an ordinary clustering of a set P, simply set $C = P$ and $F = P$. Unlike the k-means algorithm, there is no need to specify the number of clusters

in advance. It is, however, necessary to choose the facility cost. It determines how the clustering will look like. A high value will produce a clustering with a low number of large clusters. Facilities are expensive so only a few of them will be opened and a lot of clients connected to each of them. On the other hand, a low facility cost will result in many small clusters. Facilities are cheap so a lot of them will be opened and clients distributed among them. Recommendations on how to set the facility cost can be found in Section 6.2 on page 47.

Following sections describe three different approaches to solve the facility location problem. A brief overview of them may be found in [132].

Linear programming rounding

The method was introduced by Shmoys et al. [133] based on the work by Lin and Vitter [99]. It was later extended and improved in [57, 19]. The approach is based on a linear programming relaxation. Let C be the set of clients and F be the set of facilities. We allow $F = C$. Let c_{ij} denote the distance of the client $i \in C$ to the facility $j \in F$. Let $x_{ij} = 1$ if the client i is connected to the facility j; $x_{ij} = 0$ otherwise. Finally, let $y_j = 1$ if the facility j is opened; $y_j = 0$ otherwise. The facility location problem can be then formulated as an integer program

$$\text{minimise} \quad \sum_{i \in C} \sum_{j \in F} c_{ij} x_{ij} + \sum_{j=0}^{M-1} fc \cdot y_j \tag{4.7}$$

subject to

$$\sum_{i \in C} x_{ij} = 1 \qquad \forall j \in F \tag{4.8}$$

$$x_{ij} \leq y_j \qquad \forall i \in C, j \in F \tag{4.9}$$

$$x_{ij} \in \{0; 1\} \qquad \forall i \in C, j \in F \tag{4.10}$$

$$y_j \in \{0; 1\} \qquad \forall j \in F \tag{4.11}$$

whereas 4.8 ensures that each point will be connected to exactly one facility and 4.9 ensures that it will be connected to an open facility. If we let x_{ij} and y_j be any positive real numbers we get a linear relaxation of the integer program. This can be solved in polynomial time. The fractional solution is then rounded to the integer solution while increasing the clustering cost by a small constant factor. The proof may be found in [133].

Primal-dual algorithm

There is a related approach also based on linear programming. It was introduced by Jain and Vazirani [83] and later addressed by Charikar and Guha [16] and Mahdian et al. [106]. The method again starts with an integer program stated by Equations 4.7–4.11 and its linear relaxation. A dual program is then constructed as

$$\text{maximise} \quad \sum_{i \in C} \alpha_i \tag{4.12}$$

subject to

$$\alpha_i - \beta_{ij} \leq c_{ij} \qquad \forall i \in C, j \in F \tag{4.13}$$

$$\sum_{i \in C} \beta_{ij} \leq fc \qquad \forall j \in F \tag{4.14}$$

$$\alpha_i \geq 0 \qquad \forall i \in C \tag{4.15}$$

$$\beta_{ij} \geq 0 \qquad \forall i \in C, j \in F \tag{4.16}$$

The solution of this dual linear program gives the solution to the original problem. There is an intuitive interpretation of the dual variables α_i and β_{ij}. The α_i can be understood as a total contribution of the client i towards opening some facility and connecting the client to it. This can be divided to c_{ij} for connecting i to the facility j and β_{ij} for opening the facility; see Equation 4.13. Equation 4.14 describes how several clients pay for opening a facility.

Based on the dual linear program and the interpretation of dual variables, an algorithm can be formulated. The original primal-dual algorithm was proposed in [83]. The following description is, however, based on [106] which is a slightly easier and more intuitive modification. A notion of time is introduced. The algorithm starts at the time 0. Initially, all clients are unconnected, all facilities are closed and $\alpha_i = 0 \; \forall i \in C$. While $C \neq \emptyset$, for every client $i \in C$, increase the parameter α_i simultaneously at a unit rate (say by 1 in a unit time), until one of the following events occurs.

1. For an unconnected client i and an open facility j, the equality $\alpha_i = c_{ij}$ comes true. In this case, connect the client i to the facility j and remove i from C.

2. For a closed facility j, $\sum_{i \in C} \beta_{ij} = \sum_{i \in C} \max(0, \alpha_i - c_{ij}) = fc$. This means that the total contribution of clients is sufficient to open the facility j. In this case, open the facility j and for each unconnected client i for which $\alpha_i \geq c_{ij}$, connect i to j and remove i from C.

If more events occur at the same time, they can be processed in an arbitrary order.

Local search algorithm

From the general point of view, local search technique operates on a graph on the space of all feasible solutions. Two solutions are connected by an edge if one solution can be obtained from the other by a particular type of modification. The local search technique then walks in the graph along nodes with decreasing costs and searches for a local optimum. That is such a node whose cost is not greater than the cost of each of its neighbours. Korupolu et al. [91] analysed clustering techniques based on the local search. One of the first such techniques was proposed by Charikar and Guha [16]. First, a coarse initial solution is generated. It is then iteratively refined by a series of local search improvements. A single local search step can be briefly described as follows. A facility is chosen at random and it is determined whether opening it can improve the solution.

If so, nearby clients are reassigned to the new facility. Facilities with a low number of remaining clients are then closed and their clients are reassigned to the new facility too.

Describing the local search algorithm more precisely, a facility $j \in F$ is selected at random (does not matter whether it is open or closed) and it is determined whether it can improve the current solution: If j is not already open, the facility cost would have to be payed for opening it. Next, some clients may be closer to j then to their current facility. All such clients can be reassigned to j, decreasing the connection cost. After that some facilities may have just a few clients. If those clients would be reassigned somewhere else, the facilities could be closed and their facility costs spared. To limit computational complexity, reassignments are allowed only to the facility j which is being investigated. The reassignments will indeed increase connection costs, but the savings for closing the facilities (sparing their facility costs) could be larger. The possible improvement of the current solution is computed by the so called *gain* function. If $gain(j) > 0$, the facility j is opened (if not already open) and reassignments and closures are performed. The algorithm written in a pseudo-code can be found in Appendix A.1.

Figure 4.6 illustrates one local search improvement. Facilities are shown as big circles. Lines show the assignment of points to facilities. The original situation is on the left. The big grey circle denotes the facility candidate. Gray dashed lines show prepared reassignments. The situation after reassignments and closures is shown on the right. You can see that the candidate facility was actually opened. Some points were reassigned to it and the facility at the bottom was closed.

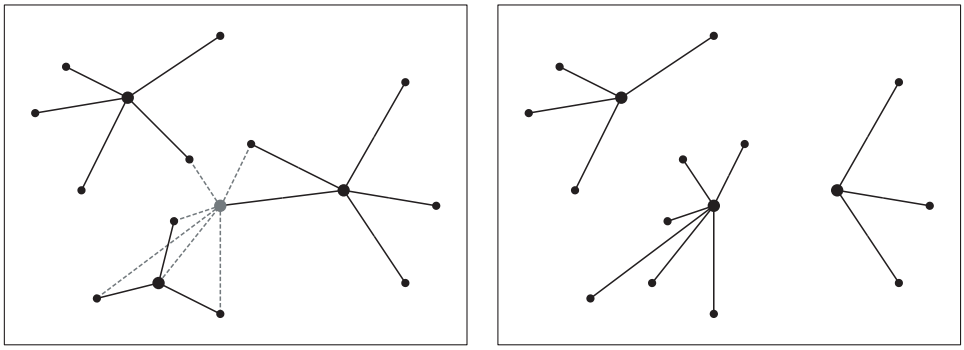


Figure 4.6: Situation before and after one local search step.

In order to obtain a constant-factor approximation, the described local search technique is repeated $N \log N$ times [16], where N is the number of potential facilities. The complete clustering algorithm written in a pseudo-code can be found in Appendix A.2. We believe that the number of iterations could be considerably reduced at the cost of slightly decreased accuracy. This is discussed in Section 6.1, starting on page 46.

An algorithm to create the initial solution is also presented in [16]. It is for the general case when the facility cost can be different for each facility. This text assumes uniform facility costs so a different algorithm proposed by Meyerson

[113] will be described here. It assumes that all input points are potential facilities, i.e., $C = F$, which is quite common in general clustering problems. Points are taken in random order. A facility is always created at the first one. For every other point, the distance d to the closest already open facility is measured. A new facility is opened at the point with probability d/fc (or one if $d > fc$). Otherwise, the point is assigned to the closest already open facility. Figure 4.7 illustrates the process how the initial solution is generated. The image on the left shows the situation after processing the first six points. The image in the middle shows that a new facility has been opened at the seventh point and three more points were processed. It is obvious that the assignment of clients to facilities is not perfect. The image on the right emphasises this by showing a crossing of assignments.

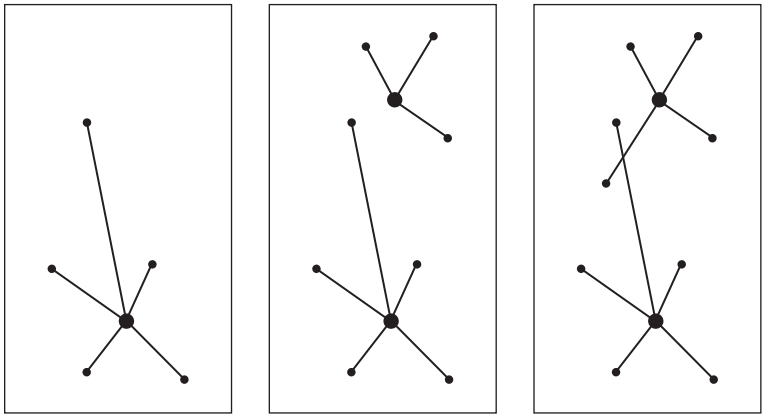


Figure 4.7: The process of generating the initial solution.

4.3 Clustering large data

Methods for clustering large data can be roughly divided into two parts. The first family of methods was developed for large databases focusing mainly on reducing the computational time. The second family is intended for processing real data streams. In this case the amount of data fairly exceeds the available memory so special techniques are required to process the data in smaller manageable pieces.

4.3.1 Methods for large databases

Originally, the problem with large data was not so much concerned with limited memory. Problems appeared earlier than all the memory was full. Many common clustering algorithms have a time complexity of $\mathcal{O}(N^2 \log N)$ which could be too much even when processing some thousands of elements. Several techniques were developed to overcome this issue. Although some of the methods can deal even with data that are larger than the available memory, they are usually not infinitely scalable and therefore are not suitable for true data stream processing. The following sections describe algorithms based on

a minimisation of the sum of squared errors which is the most common case for general clustering. There are special density-based [41] or grid-based [122] methods but these are out of the scope of this work.

Algorithm CLARANS

One of the first approaches to clustering large databases is the CLARA (Clustering LARge Applications) algorithm [86]. Instead of processing the entire data set, the algorithm draws a random sample of the data. The sample is then clustered using an ordinary k-medoid algorithm[2] [86]. Resulting medoids approximate the medoids of the whole data set. To get a better approximation, several samples may be drawn and the one with minimal sum of squared errors (SSQ) will be selected. For even better accuracy, the SSQ is computed among the entire data set, not just the sample.

The algorithm CLARANS (Clustering Large Applications based on RANdomised Search) [118] uses a different approach. Instead of working with just a sample and possibly missing some important data element, it uses the whole data set but limits the effort of the clustering algorithm. More specifically, CLARANS views the problem of clustering as a graph searching problem. Each set of cluster centres represents a graph node. Two nodes are neighbours if they differ in just one centre, i.e., if one set of centres can be obtained from the other by replacing a single centre. The search starts at an arbitrary node, searching for the node with a minimal clustering cost. Thus at every node, neighbours are examined in a random order and the search continues to a neighbour with a lower (not necessarily the lowest) cost. To limit the computational complexity, just a limited number of neighbours is inspected at every node. If none with a lower cost is found, the current node is proclaimed a local minimum. Again, this algorithm can be repeated several times and the best solution selected.

Algorithm BIRCH

The BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) [149] uses a completely different approach. For each cluster the so called *clustering feature* (CF) is defined as a triple (N, LS, SS), where N is the number of elements in the cluster, LS is the (linear) sum of elements and SS is the square sum. Only this summary information is held for each cluster, instead of all the elements. This is highly efficient and fully sufficient for all further algorithm decisions. Clustering features are organised in the CF tree. Each leaf node contains a list of $[CF_i]$ entries, where CF_i are cluster features of particular clusters. A leaf node is a cluster made up of all the subclusters represented by its entries. The size of all clusters (entries) in a leaf node is limited by a threshold value T. A non-leaf node contains a list of $[CF_i, child_i]$ entries, where $child_i$ is a pointer to a child node and CF_i is the cluster feature of this child. So a non-leaf node is also a cluster made up of all the child subclusters.

[2] Also known as the k-median algorithm. A medoid is similar to a mean, but a medoid always belongs to the data set.

The BIRCH is an incremental algorithm so it dynamically builds the CF tree as data arrive. Each new element finds its way through the tree into the appropriate cluster. If the cluster can absorb the element, i.e., the threshold T would not be exceeded, the element is added to the cluster. If the cluster would exceed the threshold, a new cluster (leaf entry) is created and the new element is placed there. If any node would have too many entries, the node is split and its entries are redistributed. In either case (absorption, new cluster creation, split) appropriate cluster features are updated. Updates are propagated up the tree which is no problem because cluster features are additive.

With the restrictions on the maximal cluster size and the maximal number of entries in a node, the algorithm can process very large data in a constant memory. The CF tree compresses data into compact summaries while maintaining the finest granularity that can be achieved given the available memory. Entries in the leaf nodes need not be perfect clusters so it is best to pass them to some ordinary clustering algorithm.

Algorithm CURE

CURE (Clustering Using REpresentatives) [61] is a hierarchical agglomerative clustering algorithm. It uses a compromise between all-point and centroid-based cluster representation. A constant number of well scattered samples is chosen in each cluster. These samples capture the shape and extent of the cluster which allows CURE to capture even non-spherical clusters. The samples are then shrunk towards the centre of the cluster by a fraction α and used as representatives of the cluster. The algorithm then proceeds as common hierarchical agglomerative clustering. It merges clusters with the closest pair of representatives.

CURE is less sensitive to outlier points. If an outlier would be selected for the representative, it would be shifted a large distance due to the shrinking, thus reducing any distortive effect. The kinds of clusters identified by the algorithm can be tuned by the parameter α. For $\alpha = 1$ the algorithm reduces to a centroid-based method like the complete link. For $\alpha = 0$ it becomes similar to the all-points approach like the single link. See Section 4.2.1 for a reference on single link and complete link algorithms.

To handle large data sets that cannot fit into the main memory, CURE uses random sampling to reduce the size of the input. If this is still not enough, the algorithm first partitions the random sample and partially clusters the data in each partition. The pre-clustered data is then clustered again in a second pass to generate final clusters. This also improves running time since it is faster to process several smaller pieces than a single large one.

4.3.2 Methods for data streams

Algorithms for clustering data streams can be divided into three groups. The first approach is based on the divide and conquer strategy. The data set is divided into several blocks which are solved separately. One or more representatives are then selected from each cluster and these are further clustered to

get the result. This technique can be extended to more levels if the data set is still too large.

The second approach is an incremental clustering. A cluster is created for the first data element. Then following elements are considered one after another. Each one is either assigned to one of the existing clusters or to a new cluster. This is done based on some similarity measure. This approach is used by the leader clustering algorithm [65], for a more recent application see [113]. The major advantage of incremental clustering is that the algorithm does not need to store all the data in the memory. So the space requirements are small. Algorithms are typically non-iterative so their running time is also low. The problem with incremental approaches is that the algorithms are mostly order-dependent. This means that the result of the clustering depends on the order in which the data is presented to the algorithm.

The last approach represents parallel algorithms. This area is out of scope of this work.

Divide and conquer data stream clustering

This paragraph will describe a method based on the divide and conquer strategy. The algorithm was proposed by Guha et al. [60] and later addressed in [119, 59]. It is easier to first explain the algorithm that can process smaller data streams. It partitions the stream into blocks which are processed separately. Each block is clustered by an ordinary clustering algorithm. The method by Guha et al. prefers the local search algorithm for its linear memory requirements. Resulting intermediate cluster centres are weighted by the number of elements assigned to them. The algorithm keeps only these centres and discards all other data from the blocks. The intermediate centres are then clustered again in order to get the final clustering. The clustering is done with respect to centre weights so that centres of big clusters have a larger importance.

If the data stream is too large, the intermediate cluster centres may not fit in the memory. This can be easily solved by allowing multiple passes. The intermediate centres can be stored on an external memory and then processed in the same way as the original data stream. Weights for resulting centres are then computed as a sum of weights of all assigned intermediate centres. By recursively using the algorithm, it is possible to process a data stream of virtually any size. It is now straightforward to extend the technique so that no external memory is required. The algorithm proceeds with clustering blocks of the data stream and stores the intermediate centres in another block at a higher level. When this block is full, it is clustered again and resulting centres are stored at the next level.

Given a block size m, the algorithm maintains at most m intermediate centres at every level. As soon as there are m centres at any level, they are clustered again and passed one level higher. At the end, the final result is found at the top level after clustering all the intermediate centres. Figure 4.8 illustrates the divide and conquer clustering. Blocks in the data stream are delimited by bold lines. Black dots indicate cluster centres in particular blocks.

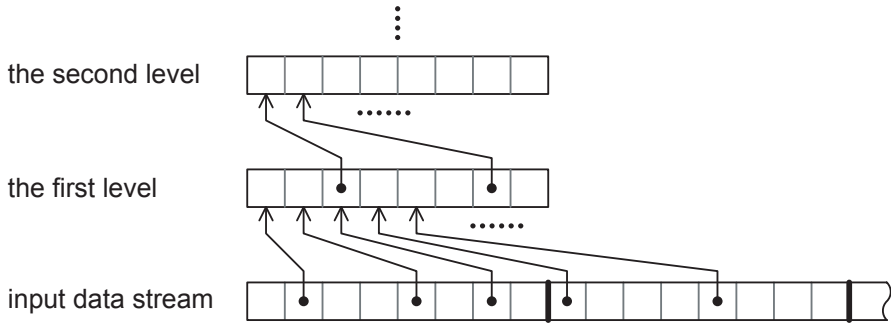


Figure 4.8: The divide and conquer clustering.

The number of levels l required to process the entire data stream can be computed as

$$l = \frac{\log(N/m)}{\log(m/k)} \tag{4.17}$$

where N is the number of elements in the whole data stream, m is the block size and k is the average number of clusters in each block. Cited papers present the equation without any further explanation. This paragraph shows how it can be derived. At the zero level, the data stream has N elements which are divided into N/m blocks. These blocks will be clustered into $N/m \cdot k$ intermediate cluster centres, which will be divided into $N/m \cdot k/m$ first level blocks. The situation repeats at higher levels until resulting l level intermediate centres fit into a single block. This can be expressed as

$$N/m \cdot k/m \cdot k/m \cdot \ldots \cdot k/m = 1 \tag{4.18}$$

where k/m repeats l times. After a simple rearrangement

$$N/m = m/k \cdot m/k \cdot \ldots \cdot m/k = (m/k)^l \tag{4.19}$$

Taking a logarithm of Equation 4.19 we get

$$l = \log_{m/k}(N/m) = \frac{\log(N/m)}{\log(m/k)} \tag{4.20}$$

Chapter 5

Delaunay Triangulation

The Delaunay triangulation [33] is well known in computational geometry. This chapter recapitulates fundamental properties and methods of construction. Further details may be found for example in [123, 103, 88]. The rest of this chapter then focuses on methods for building hierarchies in the triangulation and methods suitable for large data.

Let's start with a formal definition of a general triangulation. A triangulation $T(S)$ of a set of points S is a partitioning of space into simplices (triangles in 2D, tetrahedra in 3D) having the following properties

1. vertices of every simplex are a subset of S
2. intersection of arbitrary two simplices is either empty, a common vertex, or a common edge (or a common face)
3. the set of simplices $T(S)$ is maximal, i.e., it is not possible to add any other simplex without violating one of the previous conditions

From the last property follows that the boundary of a triangulation is the convex hull of S. The Delaunay triangulation is a triangulation where the circumcircle of any simplex is an empty circle, i.e., it does not contain any other point of S. A circumsphere is used in 3D and a circumscribed hypersphere in any higher dimension. The Delaunay triangulation is so popular because it has some nice properties. It maximises the minimum inner angle of the triangles in 2D. In other words, the Delaunay triangulation produces triangles most close to the equiangular triangle. This property is important because it ensures a low number of narrow triangles that could cause numerical problems in later processing. In higher dimensions it minimises the maximal containment sphere of simplices. In d-dimensional space if no $d+2$ points lie on a common d-sphere and no $k+2$ points, $k < d$, lie in a common k-dimensional subspace, then the Delaunay triangulation is unique.

5.1 Constructing the Delaunay triangulation

There are generally six approaches to construct the Delaunay triangulation. They are described in the following sections. Alternatively, the Delaunay tri-

angulation can be directly obtained from a Voronoi diagram [142, 123]. This is used very rarely, for instance in [71].

5.1.1 Local improvements

The technique based on *local improvements* [96] first creates an arbitrary triangulation. It then checks the empty circumcircle criterion for all neighbouring triangles and swaps their common edge if the criterion is violated. Given two neighbouring triangles forming a quad, an *edge swap* is a local modification such that the edge is replaced by the other quad diagonal. In Figure 5.1 on the left, the grey triangle does not fulfil the Delaunay property – its circumcircle contains another vertex. So the bold edge is swapped. Resulting triangles are shown on the right.

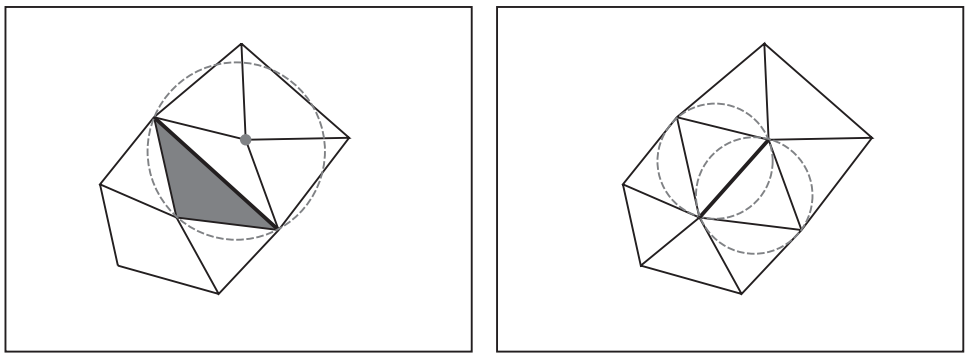


Figure 5.1: Example of an edge swap.

The algorithm of local improvements is guaranteed to converge only in 2D. Obviously, it is not suitable for large data processing because the whole triangulation must be held in memory. It also does not allow to have any hierarchy in the triangulation. The time complexity is governed by the number of swaps after the construction of the initial triangulation. In 2D, it is $\mathcal{O}(N^2)$ in the worst case and $\mathcal{O}(N)$ expected.

5.1.2 Incremental construction

Quite a different approach is the *incremental construction*. The fundamental algorithm [111] starts with an arbitrary point of S and its closest neighbour. The edge between these two points forms the base for the triangulation. Then for every outer edge AB of the current triangulation, the algorithm finds such a point C for which the triangle ABC has an empty circumcircle[1]. The triangle ABC is added to the triangulation. This repeats until the whole set S has been triangulated. The progress of incremental construction is illustrated in Figure 5.2. The algorithm has the worst case time complexity of $\mathcal{O}(N^3)$ unless some efficient data structure is used, e.g., a grid as in [20].

[1] Alternatively, a triangle with the smallest circumcircle may be found. It must inherently be empty, because otherwise another triangle with a smaller circumcircle would exist.

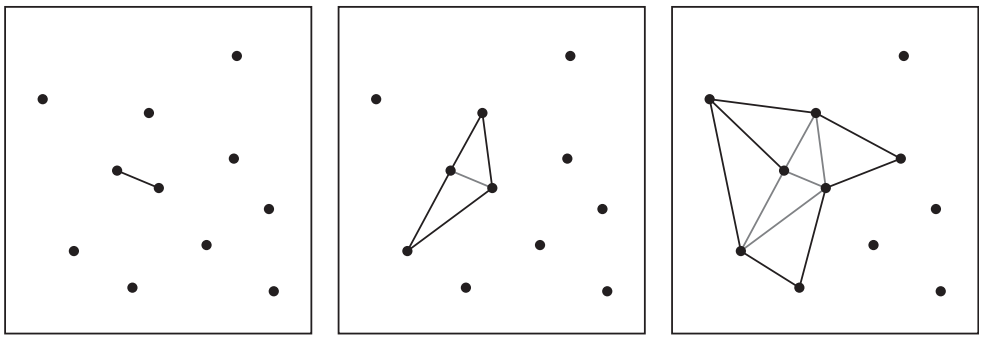


Figure 5.2: The process of incremental construction.

5.1.3 Sweeping construction

A *sweeping algorithm* for the construction of the Delaunay triangulation was presented by Fortune [46]. It is rather complicated but the time complexity is $\mathcal{O}(N \log N)$ in the worst case. An advantage of incremental construction is that once a triangle has been created, it is never changed. This allows to process even large amounts of data. If the algorithm proceeds wisely, the already triangulated parts can be put away, leaving memory for further data. This technique was proposed by Isenburg et al. [78] and is described in Section 5.3.

5.1.4 Incremental insertion

Perhaps the most popular approach is the *incremental insertion*. The algorithm starts with an artificial super-triangle enclosing all the input points. These are then successively inserted into the current triangulation, often in random order.

There are two methods of the insertion. The first one [63, 103] locates the triangle that contains the point being inserted. The triangle is subdivided into three new triangles formed by two vertices of the original triangle and the inserted vertex. The triangulation is then "legalised" by edge swaps as necessary to satisfy the Delaunay criterion. The insertion with swaps is shown in Figure 5.3. These swaps can affect the whole triangulation in the worst case. The time complexity of the incremental insertion is then $\mathcal{O}(N^2)$. But this happens very rarely, in some special cases designed on purpose to demonstrate this effect. The expected time complexity is $\mathcal{O}(N \log N)$ if a good point location algorithm is used (see Section 5.2).

Another method of insertion is known as the Bowyer-Watson algorithm [144] with a worst case time complexity of $\mathcal{O}\left(N^{\frac{2d-1}{d}}\right)$. It first finds and deletes all the triangles whose circumcircle contains the point being inserted. The resulting hole is then re-triangulated by creating edges from the hole perimeter to the new point. New triangles fulfil the Delaunay property and no swaps are necessary. The insertion with a re-triangulation is shown in Figure 5.4.

The algorithm of incremental insertion is relatively simple and has reasonable numerical stability. When a point is to be inserted, the algorithm needs to locate it inside the triangulation. Several methods exist for this. Section 5.2 describes them in more detail. Incremental insertion does not need to have

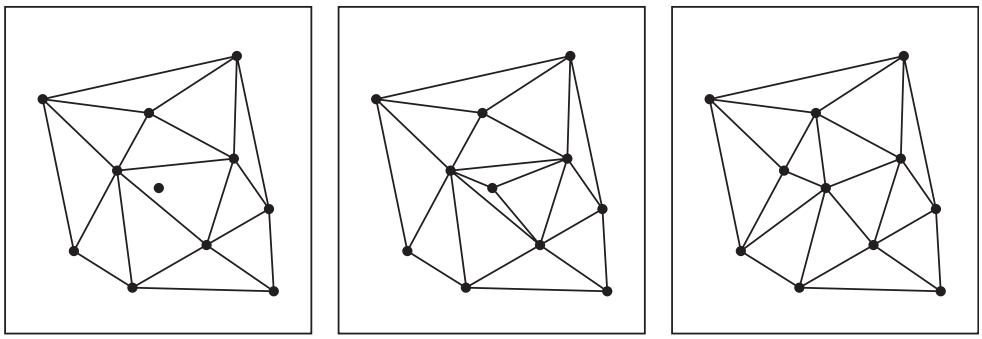


Figure 5.3: Example of point insertion with swaps.

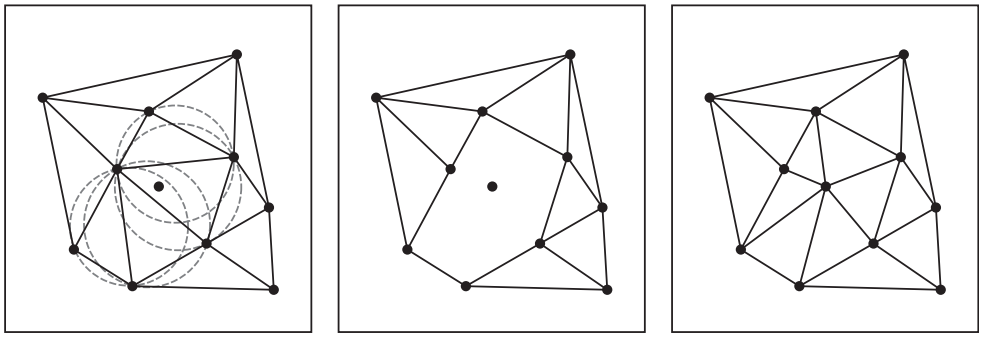


Figure 5.4: Example of point insertion with re-triangulation.

all the data at the beginning. It is only necessary to know the bounding box in order to construct the initial super-triangle. All the above properties make incremental insertion well suitable for processing large data sets.

5.1.5 Divide & conquer

The *divide & conquer* strategy is a well known approach and is often used for large data. The most influential works are [62] with the worst case complexity of $\mathcal{O}(N \log N)$, [39] with the expected running time of $\mathcal{O}(N \log \log N)$, and the algorithm DeWall [20] whose complexity was empirically showed to be sub-quadratic.

Generally, a divide & conquer algorithm recursively divides the data until the pieces are small enough to be easily triangulated for example by the incremental construction. Pieces are processed separately. Results are then merged together to get the complete triangulation. The merging phase is perhaps the most difficult. The pieces must be sewed together by constructing triangles between them. Moreover, some edges already present in particular triangulations may require to be swapped. An example is shown in Figure 5.5.

5.1.6 Higher dimension embedding

The *higher dimension embedding* [12] is a completely different approach. The input points in dimension E^d are projected on a surface of a paraboloid con-

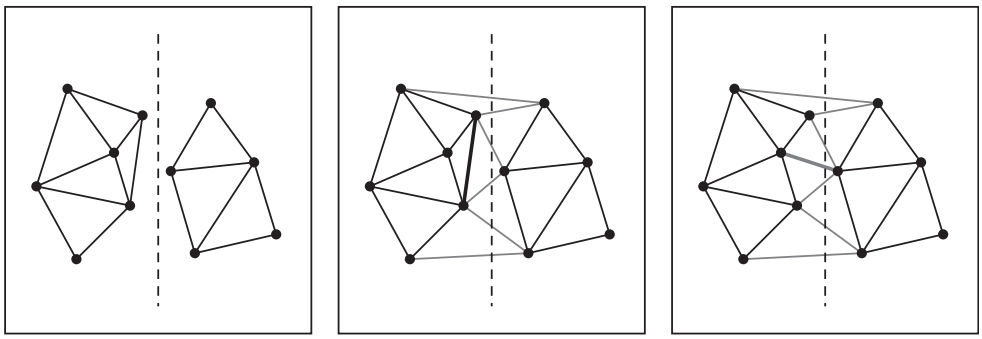


Figure 5.5: The merging phase of the divide and conquer method.

structed in dimension E^{d+1}. The projection of 2D point P into 3D point P' on the paraboloid can be written as

$$P[x, y] \mapsto P'[x, y, x^2 + y^2] \tag{5.1}$$

A convex hull of the projected points is then constructed. It is proved that projecting the convex hull back to E^d yields the Delaunay triangulation.

The time complexity of higher dimension embedding is determined by the complexity of convex hull construction. The popular gift wrapping algorithm [13] has a complexity of $\mathcal{O}(N^{\lfloor \frac{d+1}{2} \rfloor + 1})$, which is $\mathcal{O}(N^2)$ in 2D and $\mathcal{O}(N^3)$ in 3D. The higher dimension embedding can be used to partition the input data into pieces so that the borders between them are guaranteed to be edges of the Delaunay triangulation. This allows to process large data in a divide and conquer fashion with virtually no merging step [88].

5.2 Point location strategies

Sometimes it is necessary to locate a point inside a triangulation, i.e., to find the triangle that contains the given point. This is in particular essential for the incremental insertion algorithm that needs to locate every point being inserted. Traditional point location techniques are the directed acyclic graph and the walk. They are described in detail in the following sections. Further methods include, e.g., a uniform grid [42, 20] or a skip list [147].

5.2.1 Directed acyclic graph (DAG)

Several approaches exist for point location. There is an algorithm based on a directed acyclic graph (DAG) [103]. The DAG is a data structure that tracks the history of construction of the Delaunay triangulation. It is a tree shaped graph (a tree with additional edges) whose nodes correspond to triangles. The following text uses the terms *root* and *leaf*. Although this is not absolutely correct in a general graph, the meaning of the terms will be obvious and appropriate nodes are easy to identify as if the graph would be a real tree.

So the root represents the initial super-triangle. Inner nodes stand for triangles that were present in some previous version of the triangulation (recall

the incremental insertion algorithm). Leaves correspond to triangles that make up the current triangulation. Each non-leaf node has children that correspond to the triangles created either by a subdivision of the node's triangle, or by an edge swap. Figure 5.6 shows how the DAG changes during a point insertion. The point is located inside the triangle $T2$ which is subdivided, so the node $T2$ in the DAG gets three new children $T4$, $T5$, $T6$. Then the edge between the triangles $T1$ and $T4$ needs to be swapped, so both the node $T1$ and $T4$ get two new children $T7$ and $T8$.

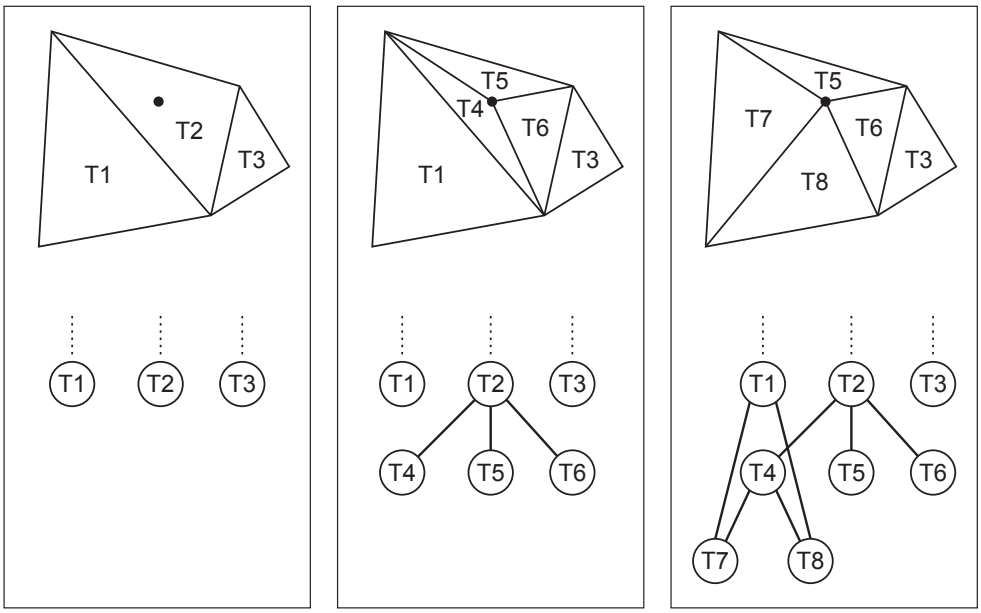


Figure 5.6: Changes in the DAG when a point is inserted into the triangulation.

When a point p needs to be located, the algorithm starts from the DAG "root". It inspects its children and finds the node whose corresponding triangle contains p. The search then continues through that node until a leaf node is reached. Point p is then located inside the triangle associated with the leaf. The point location with DAG achieves the optimal time complexity of $\mathcal{O}(\log N)$ in the expected case. In the worst case, when the DAG "tree" is extremely imbalanced, the complexity is $\mathcal{O}(N)$. A disadvantage is that the DAG consumes a lot of memory which makes it inappropriate for processing any large data. Namely $\mathcal{O}(N^2)$ memory is required in the worst case, though, the worst case is very unlikely to occur.

5.2.2 Walk in a triangulation

A different approach for the point location is a walk in the triangulation. The algorithm starts at an arbitrary spot in the triangulation and traverses triangles from neighbour to neighbour until it reaches the triangle containing the query point. There are several methods how the walk may proceed [35]. The last point inserted into the triangulation is often selected as the starting point for

the next walk. To be consistent with [35], this text designates the starting point q and the query point p.

The *straight walk* proceeds along the line qp. The algorithm starts with an arbitrary triangle incident to q and turns around q until it reaches the triangle intersected by the line qp. Then for each following visited triangle the line qp goes out of the triangle through the edge e. If p lies on the near side of e, the current triangle contains p. Otherwise, the walk proceeds to the neighbour across e. The new vertex of the neighbour is located with respect to the line qp. This determines through which edge of the neighbour the line qp goes out. The straight walk is simple unless it has to deal with degenerate cases. Also numerical stability can be a problem.

The *orthogonal walk* is different in that it first goes along a horizontal line from $q = (q_x, q_y)$ to (p_x, q_y), and than along a vertical line from (p_x, q_y) to $p = (p_x, p_y)$. The advantage of this technique is that while walking only in axis aligned directions, it does not need to evaluate expensive orientation tests. Simple greater than/less than comparisons are enough to decide which way to walk. Only at the end of both the horizontal a vertical passes, a few orientation test may be necessary to decide precisely.

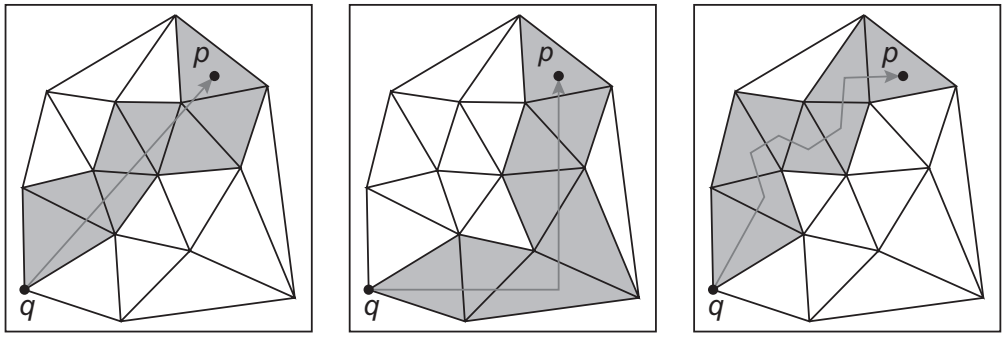
The *visibility walk* in its fundamental version starts from any triangle incident to q. Then for each visited triangle it tests the first edge e whether the line supporting e separates the triangle from p. If so, the walk proceeds to the neighbour across e. Otherwise, the next edge is tested. If all three tests fail, the current triangle contains p.

It is necessary to define some edge ordering for this algorithm. But a much more serious problem is that for a non-delaunay triangulation, this walk may fall into an infinite loop. So it cannot be used for example in a constrained triangulation. Fortunately, there is a simple modification – the *stochastic walk* – that overcomes this issue. The only modification is that triangle edges are tested in a random order. This is proved to ensure that the walk will reach p in a finite number of steps.

There is one more improvement called the *remembering stochastic walk*. It remembers the edge it came through to the current triangle. This edge is then excluded from the tests, since it has been already tested in the previous step and it is worthless to do it again. This may spare one orientation test in some triangles.

Figure 5.7 shows a comparison of the walking strategies – (a) the straight walk, (b) the orthogonal walk, (c) the stochastic visibility walk. The grey path shows the triangles visited by each particular walk. In this example, all strategies visit the same number of triangles. This is not a rule in a general case.

The worst case complexity of the straight walk and the orthogonal walk is $\mathcal{O}(N)$ per point location. The stochastic walk can have exponential length. There is an example for every algorithm that demonstrates the worst case [35]. But in practise the expected complexity of all the mentioned walk algorithms is $\mathcal{O}\left(\sqrt[d]{N}\right)$, where d is the dimension. This is nice, although worse than the optimal $\mathcal{O}(\log N)$ complexity of the DAG. The most important is that walking



(a) straight walk **(b)** orthogonal walk **(c)** visibility walk

Figure 5.7: Comparison of the walking strategies.

algorithms need a constant (and very little) extra memory, so they are perfectly suitable for processing large data.

Devillers [34] proposed a hierarchical approach. The lowest level holds the complete triangulation T_0. Each higher level contains a triangulation of a small sample of the level below. The walk starts at the highest level k and locates vertex $v_k \in T_k$ that is closest to the query point p. The walk then continues at the levels below (note that also $v_k \in T_{k-1}$) until p is located at the lowest level. This scheme guarantees $\mathcal{O}(\log N)$ location time in all cases.

5.3 Streaming Delaunay triangulation

Recently, Isenburg et al. [78] introduced a method for computing Delaunay triangulation of very large data sets using extremely small amount of memory compared to the size of input data. The method builds on a non-surprising observation that most real-world data have a spatial coherence. If the data are presented in a form of a stream, points lying geometrically close together are also located close together in the stream.

The proposed method exploits this property to introduce *finalisation tags* into the stream. The data set is divided into regions. When all points from a particular region appeared in the stream, the region is declared finalised and a finalisation tag for that region is injected into the stream. The Delaunay triangulator then uses these tags to identify areas where no more points will arrive and so the triangulation will not change. Such finalised parts of triangulation may be sent to output, freeing memory for further data.

5.3.1 Inserting finalisation tags into the stream

The finalisation algorithm processes the stream in three passes. In the first pass, it finds the bounding box of the data. A regular grid is then laid over the data partitioning it into rectangular cells. The second pass counts the number of points in each cell. These statistics are then used in the third pass. This time, the algorithm decrements the counters. When a cell's counter reaches

zero, all points from the cell have arrived and the finalisation tag for that cell is inserted into the stream.

The spatial coherence can be further increased during the third pass. The algorithm buffers all the points in each cell until the cell's counter reaches zero. All points are then output at the same time followed by the finalisation tag. The increased coherence is well worth the additional memory demands and still requires far less work than fully sorting all the data.

Many algorithms, such as the incremental insertion, may experience a significant drop of performance if input data arrive in an undesired order. To avoid this it is best to process input points in a random order. So ultimate spatial coherence is not always the best. Therefore the finalisation algorithm samples several points from the stream and promotes them to earlier spots in the stream. This is done locally within each cell and also globally among all cells.

On the local level the algorithm uses the BRIO [4] which was designed exactly for this purpose. In the third pass when a cell is finalised and points released to the output stream, the algorithm moves a sample of randomly selected points to the front of the chunk.

The global sampling starts in the second phase. The algorithm builds a quadtree whose leaves are the grid cells, and stores one point from each quadrant at each level of the tree. During the third pass it moves these sampled points to early spots in the stream. Points are not moved right to the beginning because for very large data this would destroy the spatial coherence. Instead of releasing all the points at once, they are inserted into the stream in a lazy fashion. When a cell is finalised, the sample points of all its ancestors and their immediate children in the quadtree are released before the points of the finalised cell.

5.3.2 Streaming triangulation

The triangulation algorithm uses the common method of incremental insertion. Conventional programs output triangles after processing all the data. The speciality of the streaming algorithm is that it outputs a triangle whenever it determines that the triangle is final, i.e., its circumcircle has no intersection with any unfinalised cell. Such a triangle is for sure in the Delaunay triangulation since no point arriving in the future can be inside its circumcircle. A triangle that is not final is called active.

In addition to the triangulation itself, the algorithm keeps a quadtree that remembers which cells have been finalised. When a point arrives in the stream, it is inserted into the triangulation. When a finalisation tag arrives, the algorithm notes the finalised cell, determines which triangles become final, writes them to the output, and frees their memory. This dramatically reduces the amount of memory used by the program. Figure 5.8 shows the streaming Delaunay triangulation in progress. The points in the white part have been processed and their triangles sent to output. The grey triangles are active. There are a few representative circumcircles that intersect unfinalised cells. The figure shows a situation when points are being inserted into the cell in the middle.

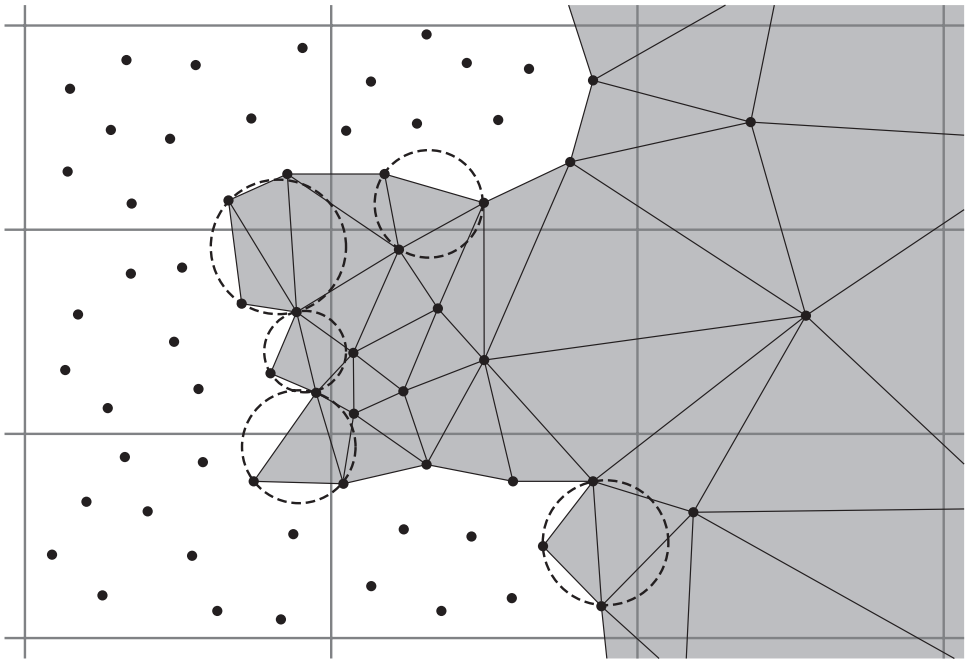


Figure 5.8: The streaming Delaunay triangulation in progress. The image is inspired by [78].

The algorithm uses a straight walk to locate a point inside the triangulation. As can be seen from Figure 5.8, the location may fail because it tries to walk through a final triangle that is no longer in memory. In such a case the walk is restarted from a different starting point. Each quadtree leaf maintains some triangles whose circumcircles intersect the leaf's quadrant. So the algorithm finds the quadrant containing the point to locate and restarts the walk from one of the triangles on the quadrant's list. If the walk fails again, the algorithm tries other triangles on the list and then triangles from lists of neighbouring quadrants. If no walk succeeds, the algorithm resorts to an exhaustive search through all active triangles. Isenburg et al. claim that this happens for fewer than 0.001% of points.

When the algorithm reads a finalisation tag, it needs to check which active triangles become final. It first checks whether the circumcircle of a triangle is completely inside the cell that was just finalised. This simple test marks many triangles as final. If this test fails, a circle-rectangle intersection is computed. The quadtree hierarchy is used to early reject triangles that are still active. If a triangle's circumcircle intersects an unfinalised cell, it would be wasteful to test the triangle again before that cell is finalised. So the triangle is added into the cell's list and ignored until the cell is finalised. When the proper finalisation tag arrives, the triangle is checked again. Tests continue in the quadtree from the just finalised cell where the triangle was listed.

Chapter 6

Contributions

This chapter presents the contributions we made to the current state of the art of clustering and large geometric data processing. Namely we have improved the facility location algorithm and adapted the data stream clustering to facility location. Further, we made modifications to the clustering as well as to the Delaunay triangulation so that the algorithms can work with an anisotropic metric based on [140]. We believe that it could bring a new interesting potential especially to the area of clustering. Anisotropic metrics allow to create non-spherical clusters that fit a wider variety of shapes.

Currently the greatest deal of work is focused on the hierarchical triangulation. We would like to use it to provide complex geometric models at various levels of detail in different parts. Another problem we work on is adapting the clustering to the Euclidean matching. The task is to group points into pairs so that the clustering could be used for ray tracing acceleration. Both the hierarchical triangulation and the Euclidean matching is work in progress, nevertheless, we have already results to present.

6.1 Speeding up the facility location

This is an improvement to the Local search clustering algorithm [16] described in Section 4.2.3, starting on page 28. It is the facility location approach that iteratively refines an initial solution by local improvements. Each local search step involves computing a *gain function* that determines whether the step is beneficial and actually improves the solution.

Evaluating the gain function takes $\mathcal{O}(N)$ time. It is not bad, however, $\mathcal{O}(N \log N)$ local search iterations are necessary to get a constant-factor approximation to the optimal clustering. This makes the speed of gain evaluation rather significant. We may speed up the computation by limiting the number of points that need to be inspected. Given a facility cost *fc*, any point can be connected to a facility that is at most *fc* far away. Otherwise it would be cheaper to open a new facility at that point. This holds true in the initial solution as well as during the iterative local improvements.

Let us define the *influence area* of facility f to be the circle centred at f with a radius *fc*. From the above reasoning follows that all the points connected

to f must lie within its influence area. So to compute the gain for some facility candidate, it is sufficient to inspect just those points whose distance is at most fc, that is the points which fall into the facility candidate's influence area. When we take all facilities in a $2 \cdot fc$ radius around the facility candidate, and examine all points connected to those facilities, we can be sure that we inspected all the points that may be relevant.

An example situation is illustrated in Figure 6.1. Facilities are shown as dots with their influence area delimited by a dotted circle. The facility candidate is denoted black. We need to inspect all the points that may lie within its influence area IA. Such points can only belong to the facilities whose influence area overlaps with IA. Such facilities (shown as diamonds) lie within the dashed circle with a radius $2 \cdot fc$.

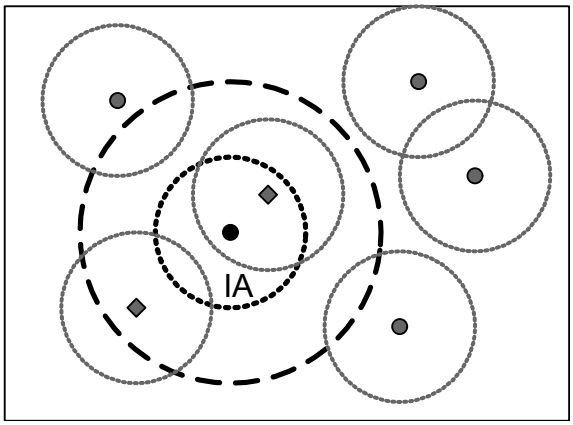


Figure 6.1: Finding points that may lie within the facility candidate's influence area.

The proposed idea works for points without weights. The situation is a bit more complicated for weighted points because distances are multiplied by point weights. Thus a point with a small weight can be connected to a distant facility, while a point with a big weight should be connected to a nearby facility. Nevertheless, the above idea can easily be adapted to this situation. For each facility we find the point with the minimal weight w_{min}. We then set the influence area radius to fc/w_{min} which is the farthest distance where the minimal weighted point can lie. To compute the gain it is then necessary to inspect all the points assigned to facilities lying at most $2 \cdot fc/w_{min}$ away from the facility candidate. Note that w_{min} is generally different for each facility so now it is not possible to take all facilities in some radius. The distance must be checked for every one separately.

Except the above method there is another possibility to speed up the computation greatly at the cost of decreased accuracy. It is proved [16] that $\mathcal{O}(N \log N)$ local search iterations are necessary for a constant factor approximation. We have made experiments with the number of iterations and it turned out that it can be reduced significantly without major impact on the result. Only about $0.1N$ iterations were necessary for uniformly distributed data. Data with obvious clusters required even less iterations. Figure 6.2 shows a comparison of clustering the same data with $N \log N$ and $0.1N$ iterations re-

spectively. Black dots indicate points assigned to a different facility than to the closest one. It can be used as an approximate measure of error, but just for the current set of facilities. No black dots mean an optimal assignment to *currently open* facilities. More detailed experiments are documented in our paper [134].

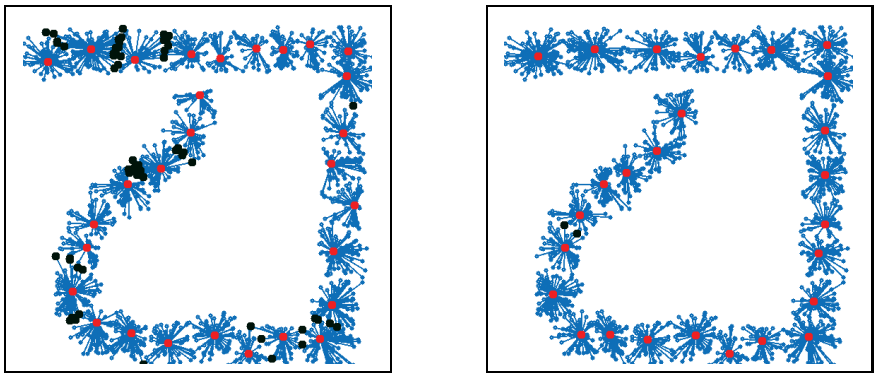


Figure 6.2: Comparison of full and reduced number of iterations.

6.2 From the k-median to the facility location

The data stream clustering method [60] described in Section 4.3.2 on page 33 solves the k-median problem. It computes the facility location repeatedly and using a binary search it finds such a facility cost that yields exactly k clusters.

The k-median algorithm has the property that it clusters data into exactly k clusters. The k must be specified in advance. Although this property may be favourable in some applications, generally we have no idea about the number of clusters in the data. It is unpleasant and inefficient to guess this value by a trial-and-error. From this point of view the facility location approach seems considerably more convenient. It finds a suitable number of clusters automatically with the possibility to control this by the facility cost parameter.

Based on the above consideration we decided to modify the original method to facility location, i.e., to compute the clustering just once without searching for k. Solving the facility location itself seems to be just a part of the original method, but it is not that simple. Instead of a precise number of clusters we must set the facility cost. This appears to be the same problem. But if we find a good default value that yields a natural clustering, the algorithm will then run on any data without user having to tune the parameter every time.

The facility cost is a counterbalance to point distances. To keep consistent results for various data, we need a small cost for clustering points for instance in a unit square, and a high one for points in a $[0; 10^6]$ range. It follows that the facility cost must be derived from the range of point coordinates. Based on our experiments we suggest setting the facility cost equal to the diagonal of data bounding box. It produces good results (i.e., a clustering that seems natural to a human observer) in most scenarios. If necessary the facility cost can be multiplied by a constant to make the clustering either stronger or more

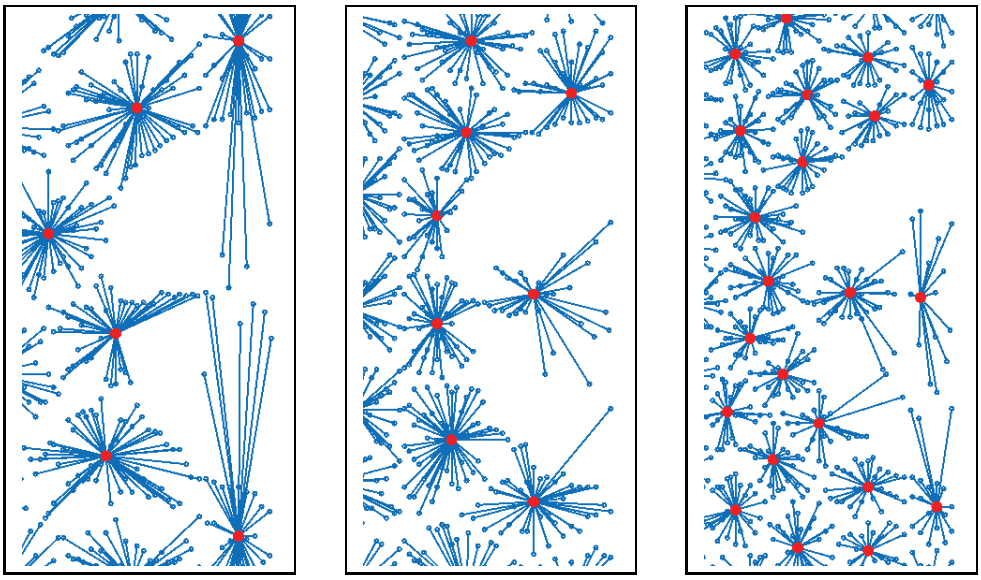


Figure 6.3: Comparison of clustering using a facility cost of $2fc$, fc and $0.5fc$ respectively. Images cropped.

moderate and thus adapt the algorithm to special needs. Figure 6.3 illustrates the effect of scaling the facility cost.

Another problem we had to face was clustering points at higher levels. Points have weights so all distances are multiplied by some (possibly large) numbers. If we perform the clustering as usual, a facility would be opened at almost every point because point weights make them several times farther from each other. A solution might be to increase the facility cost but point weights will then grow higher with increasing level and we may encounter numerical problems. Instead of scaling the facility cost we decided to introduce weight normalisation.

Points at level zero have a unit weight, causing no problems. It would be nice to keep weights around one also at higher levels. To achieve this we simply divide all weights by their average. The average of new weights will be one, exactly as we wanted. It is important to do the normalisation of all the points in a block at the same time. This means not earlier then the block is full. One would think of normalising the weights right after clustering a lower level block (when passing points to the higher level). But this is wrong. Generally, each block may have a different number of clusters so the average weight may also vary. Thus points from different blocks would not be normalised equally.

After processing the whole data stream, the highest level may contain points that lie close together. These are similar facilities from different lower level blocks. Such points obviously make up one big cluster so it might be desirable to group them together. This can be done by simply clustering them once more. But the standard procedure of weight normalisation would cluster too much resulting in just several huge clusters. It is therefore advisable to omit the normalisation and leave weights higher. The clustering then just groups nearby points together and leaves the others as they are.

6.3 Dynamic hierarchical triangulation

The dynamic hierarchical triangulation is the core of our current research. The triangulation uses the point hierarchy created by the data stream clustering algorithm which is described in Section 4.3.2 starting on page 33. Initially a triangulation of the highest level is constructed. Each point at the highest level represents a cluster of points at a lower level. It is then possible to *expand* any of the clusters, insert all its points into the triangulation and thus locally increase the level of detail. The expansion can indeed continue down to the lowest level. Nevertheless, we can expand only as many clusters as fit into the memory. It is therefore possible to *collapse* clusters that are no longer interesting and thus free memory for other data.

Figure 6.4 shows a 2D triangulation of the Lucy model originally consisting of about 10 million vertices. Level 2 of the hierarchy contains just 95 vertices. The right hand is expanded to level 1 and fingers down to level 0 which is the full resolution. Triangulations at particular levels are rendered separately in different colours for visual clearness. Of course our system can also work with all the points in a single triangulation. The frame on the left shows the whole model, frames on the right are closeups of the hand and fingers.

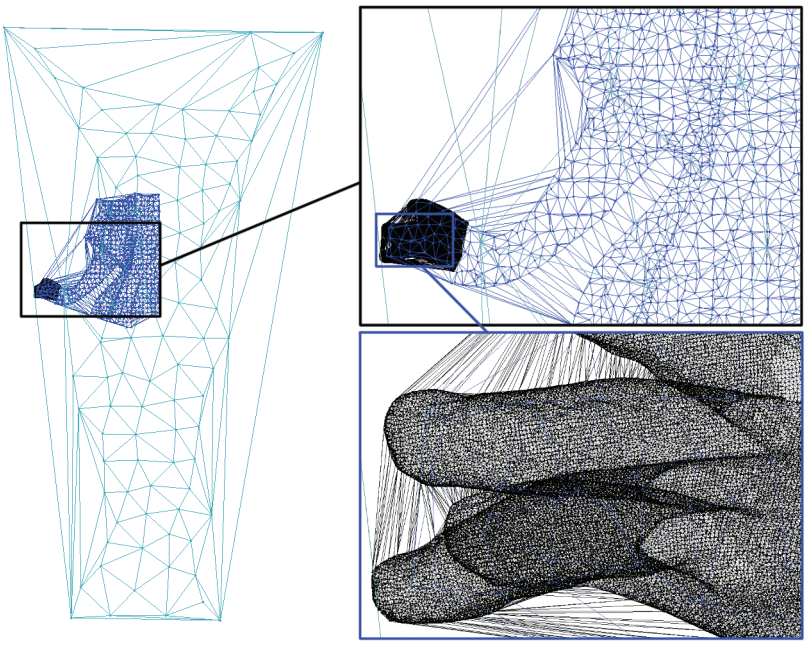


Figure 6.4: Example of the Lucy model with the right hand in a higher resolution and fingers in the full resolution.

An example of clustering a digital elevation map of the whole world can be found in Appendix B.

Now on how the dynamic hierarchical triangulation works. It is to be noted that so far the solution is only 2D. The first thing to describe is the format in which the hierarchy of clusters is stored on a hard disk. Each clustering level is saved in a separate binary file. This is no problem because even million-sized

geometric models can be clustered using only a few levels. So the result will be just several files with a convenient access to particular levels.

Vertices of each cluster are stored in a continuous block as illustrated in Figure 6.5 in the bottom row where you can see clusters distinguished by different shades of grey. Each cluster centre is stored at a higher level along with an address and a size of the block containing points of the cluster. This is seen in the top row in the figure. Using this structure the triangulation program can easily load and expand any particular cluster on demand.

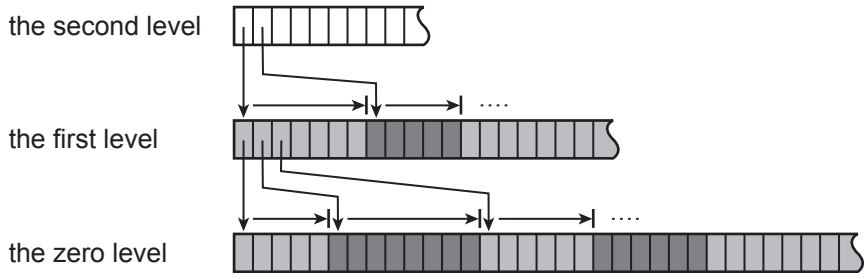


Figure 6.5: Scheme of how the hierarchy of clusters is stored on a hard disk. Cluster centres at higher levels have pointers to the whole clusters at lower levels.

Expansion of a cluster is not difficult. All points are simply inserted one by one into the triangulation using the incremental insertion algorithm; see Section 5.1.4 page 37 for details. We believed it could be possible to insert a pre-triangulated cluster all at once but it does not seem to be profitable. Perhaps we will make more research in this direction later.

Collapsing a cluster deserves a more detailed explanation. The algorithm is based on the technique of removing a vertex from the triangulation. The original technique removes a vertex along with all incident triangles. The resulting hole is then re-triangulated. Our algorithm extends the technique in that it enlarges the hole as much as possible by removing further vertices before the re-triangulation. The solution is simple, reliable and runs relatively fast. The algorithm proceeds as follows:

1. create a hole – remove the first vertex
2. enlarge the hole – continue with removing further vertices as long as they are on the hole boundary
3. re-triangulate the hole
4. if not all cluster vertices has been removed, go to 1

Upon creating the hole the algorithm remembers a chain of vertices along the hole boundary as well as a chain of neighbouring triangles. The algorithm then proceeds with removing further points. The chains are updated with each removal. Figure 6.6 shows a sample situation after removing three points. The hole is dark grey, removed edges and vertices are white. The chain of vertices is bold and the chain of triangles is light grey. The important condition is that a point can be removed only if it lies on the hole boundary. Otherwise

another hole will appear. Such holes could later touch each other or merge together. That would require complicated overhead and it is unlikely to bring any major improvement to the algorithm. For the same reason we disallow the degenerate case when a vertex appears more than once in the chain; see examples in Figure 6.7. If any vertex removal should result in such a situation, the removal is cancelled.

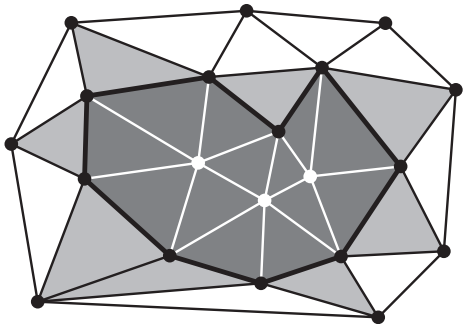


Figure 6.6: A hole after removing three vertices.

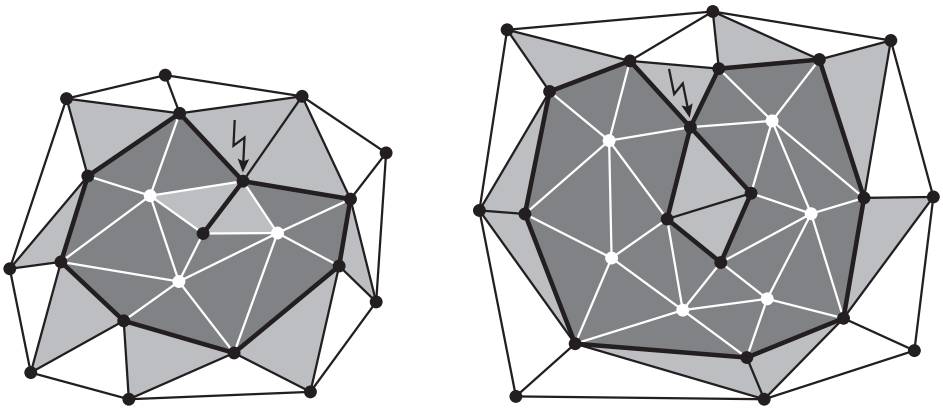


Figure 6.7: Examples of degenerate holes.

When no further vertex can be removed the hole is re-triangulated. We use a simple ear cutting algorithm. An ear is generally a valid triangle formed by three subsequent vertices v_i, v_{i+1}, v_{i+2} at the hole boundary. Cutting an ear means adding the triangle $v_i v_{i+1} v_{i+2}$ to the triangulation and removing the second vertex v_{i+1} from the boundary. Our implementation goes around the hole boundary (using the chain of vertices) and it tests ears whether they fulfil the Delaunay property. Only points in the chain are included in the test. Other points in the triangulation do not play any role. Ears that pass the test are cut. Proper triangle neighbourhood is established using the chain of triangles. Both chains are then updated to reflect the cut ear and the algorithm goes on until the hole is patched.

6.4 Anisotropic metrics

Anisotropic material has variable properties in different directions. A nice example is wood – it is very strong along the grain, but transversely, it breaks easily. Geophysics studies anisotropic materials in connection with variations in seismic wavespeed or for gas and oil exploration. Also medical ultrasound imaging uses the fact that soft tissues have different echo depending on the angle of the sound source. Anisotropy is often found in optical properties of minerals such as the birefringence of calcium crystal. Computer graphics may be interested in anisotropic surfaces, such as velvet, that change their appearance when rotated about their normal.

The above examples lead us to the belief that anisotropic metrics will be an effective feature in both clustering and triangulation. We decided to implement an elliptical metric because it is relatively simple and yet provides enough flexibility. An elliptical metric can be viewed as an elliptical elongation of the classical Euclidean space. Concerning clustering, it allows to shape clusters as ellipsoids rather than spheres. Ellipsoids can significantly better match phenomena like those mentioned in the previous paragraph. The effect on a Delaunay triangulation is that edges tend to go along the direction specified by the ellipse. This could be useful to prepare the triangulation for later deformations so as to avoid degenerate cases.

To get even more flexibility we added the possibility to define different elliptical metrics in limited regions of the data. Figure 6.8 shows an example. The first frame shows the data with three regions defining three different elliptical metrics as illustrated by the ellipses; standard Euclidean distance is used outside the regions. The next two frames show the clusters and the triangulation created using the specified metrics.

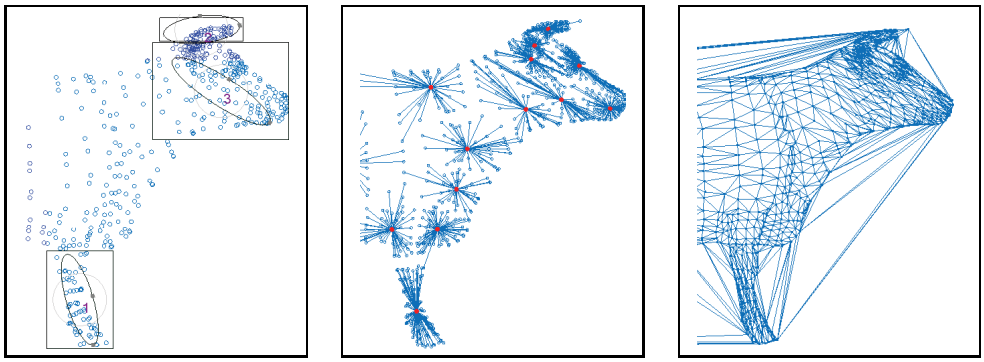


Figure 6.8: Illustration of regions with different elliptical metrics (left). The clustering and the triangulation computed using the defined metrics (middle and right).

Now on the mathematics how the elliptical distance is computed. The idea of computing a Delaunay triangulation using elliptical metrics was proposed by Vigo and Pla [140]. The equations they use are a bit obscure. Namely, point coordinates are scaled, but it does not matter when computing the triangulation. The following text explains the equations in detail. The algebra regarding ellipses and their matrix expressions can be found for example in [66].

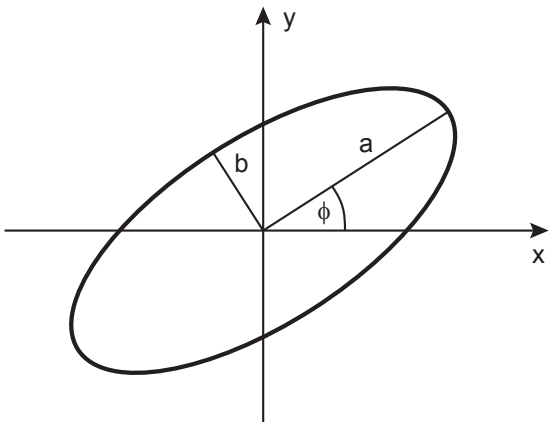


Figure 6.9: An ellipse defining the metric.

The following is an intuitive derivation of the equations used for computing with an elliptical metric. It slightly differs from [140] so as to keep the scale correct which is especially important for clustering. For the sake of simplicity we will work in 2D space. The situation for higher dimensions is analogous.

The metric is defined by an ellipse so that any radius of the defining ellipse has a unit length, i.e., using the elliptical metric the defining ellipse appears to be a unit circle. The ellipse is defined by its major and minor axes. The position in space is irrelevant.

The simplest way to use the elliptical metric is to transform points into the Euclidean space and then to work with them as normal. So we are looking for a linear mapping $\mathbf{M}$ that maps the defining ellipse onto a unit circle. This is done by a rotation $\mathbf{R}$ (to align the ellipse axes with coordinate axes) followed by a nonuniform scaling $\mathbf{S}$ (to scale the ellipse to a unit circle). Note that no translation is needed since the defining ellipse is centred at the origin. Finally a reverse rotation $\mathbf{R}^{\mathrm{T}}$ should be applied. It has no effect on point distances, however, we do the rotation because the transformation matrix will then be symmetric which has advantages in algebraic calculations. Please refer to [66] for more details.

Figure 6.9 shows a defining ellipse as a reference for the following equations. Assuming a column vector notation, point p is transformed to the Euclidean space as

$$p' = \mathbf{M}p \tag{6.1}$$

We construct the transformation matrix $\mathbf{M}$ as

$$\mathbf{M} = \mathbf{R}^{\mathrm{T}}\mathbf{S}\mathbf{R} = \begin{pmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} \frac{1}{a} & 0 \\ 0 & \frac{1}{b} \end{pmatrix} \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} \tag{6.2}$$

$$\mathbf{M} = \begin{pmatrix} \frac{1}{a}\cos^2\phi + \frac{1}{b}\sin^2\phi & \left(\frac{1}{b} - \frac{1}{a}\right)\sin\phi\cos\phi \\ \left(\frac{1}{b} - \frac{1}{a}\right)\sin\phi\cos\phi & \frac{1}{a}\sin^2\phi + \frac{1}{b}\cos^2\phi \end{pmatrix} \tag{6.3}$$

The scaling is where we differ from [140]. They start from the matrix represen-

tation of ellipse and so they compute the scaling as

$$\mathbf{S}_{Vigo} = \begin{pmatrix} b & 0 \\ 0 & a \end{pmatrix} \tag{6.4}$$

which also results in a circle, but scaled by a factor of ab. As mentioned earlier, this does not matter when triangulations are computed.

We can now derive how to compute the elliptical distance without transforming the points. Let $|pq|_E$ denote the elliptical distance between points p, q. Let p', q' be the points p, q transformed into the Euclidean space. We can write

$$|pq|_E = |p'q'| = \sqrt{(p'-q')^{\mathrm{T}}(p'-q')} = \sqrt{(\mathbf{M}p - \mathbf{M}q)^{\mathrm{T}}(\mathbf{M}p - \mathbf{M}q)} \tag{6.5}$$

Using linear algebra rules we rearrange the equation as

$$|pq|_E = \sqrt{[\mathbf{M}(p-q)]^{\mathrm{T}}\mathbf{M}(p-q)} = \sqrt{(p-q)^{\mathrm{T}}\mathbf{M}^{\mathrm{T}}\mathbf{M}(p-q)} \tag{6.6}$$

Now we utilise the fact that the matrix $\mathbf{M}$ is symmetric. The elliptical distance is computed as

$$|pq|_E = \sqrt{(p-q)^{\mathrm{T}}\mathbf{M}^2(p-q)} \tag{6.7}$$

If we look back on how the matrix $\mathbf{M}$ was constructed we find out that $\mathbf{M}^2$ can be computed as $\mathbf{R}^{\mathrm{T}}\mathbf{S}_{sqr}\mathbf{R}$ where $\mathbf{S}_{sqr}$ is the matrix $\mathbf{S}$ with its components squared.

$$\mathbf{M}^2 = \mathbf{R}^{\mathrm{T}}\mathbf{S}_{sqr}\mathbf{R} = \begin{pmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} \frac{1}{a^2} & 0 \\ 0 & \frac{1}{b^2} \end{pmatrix} \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} \tag{6.8}$$

Constructing a triangulation requires the input points to be transformed into the Euclidean space. This is done using Equation 6.1. There is no need to transform the points when computing a clustering. The elliptical distance can be computed directly using Equation 6.7.

6.5 Euclidean matching

The idea was proposed by Jiří Bittner from the CTU in Prague. He thought the clustering could be used as a space partitioning method for the visibility culling and occlusion queries for ray tracing acceleration. We would like to examine whether it could bring better results than traditional approaches such as a kD-tree.

The problem is that the partition should be binary, i.e., each cluster should have exactly two members. We made attempts to adapt our clustering algorithm to that demand but it would require substantial changes. Therefore we decided to implement a special method. After a brief research we identified the problem as the minimal Euclidean matching. The goal is to group the points into pairs so that the sum of distances between all pairs is minimal. See Figure 6.10 for an example. There is the Edmonds algorithm [40] that finds an optimal solution in $\mathcal{O}(N^4)$ time. Gabow [49] proposed a more efficient implementation

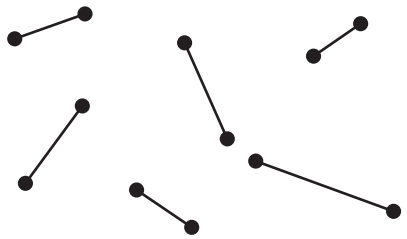


Figure 6.10: Example of an Euclidean matching.

running in $\mathcal{O}(N^3)$ time. However, the algorithms turned out to be too intricate to implement just as a trial. We chose to implement a simple heuristics instead. The algorithm is designed for small data. The data stream processing and building the hierarchy is handled by our clustering system.

The work described hereafter is a joined work with student Jan Hyka who participated on the algorithm design and did most of the programming. Let us briefly describe the progress of the development. We think of the problem as a graph problem which is formally stated as follows. Given a set of $2N$ vertices corresponding to nodes of a complete graph with edge weights equal to Euclidean distances, find the minimum weight perfect matching. A perfect matching is a matching where every vertex is incident to exactly one edge of the matching.

6.5.1 Initial attempts

The most straightforward solution would be a greedy algorithm. It would successively select the closest pair of points and match them together. However, this simple approach can go terribly wrong [135]. Next we tried another greedy algorithm that starts with a complete graph and successively discards the longest edge. If any edge is the last one incident to a vertex, the edge is fixed, i.e., it is declared as a part of the matching and is never removed. The algorithm terminates when all edges have been either removed or fixed. However, the algorithm may still run in a situation, from where a perfect matching cannot be achieved [135].

As the next attempt we addressed the problem as a matching in a bipartite graph. A *bipartite graph* is a graph whose vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to a vertex in V, i.e., U and V are independent sets. See Figure 6.11 for an example. Matching in a bipartite graph is relatively easy to solve by the Kuhn-Munkres algorithm [92, 115], also known as the Hungarian method, running in $\mathcal{O}(N^3)$ time.

The problem is that we generally do not have a bipartite graph. Our idea was to construct a bipartite graph $G = (X \cup Y, E)$ by putting all the input vertices into the first set $X = \{a, b, c, \ldots\}$ and then duplicate them into the second set $Y = \{a', b', c', \ldots\}$. The problem is that the result in the bipartite graph is not always symmetric. If $[a, b']$ is a pair, then not necessarily [b,a'] is a pair too. We did not find a solution for that, so we tried yet another approach.

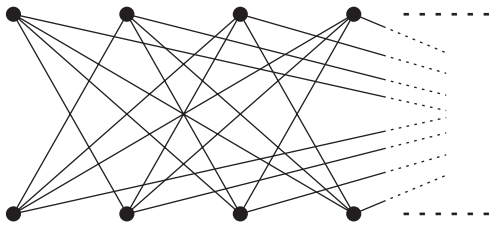


Figure 6.11: The bipartite graph where the matching is constructed.

6.5.2 The working algorithm

Despite the previous bad success, we stayed with the Kuhn-Munkres algorithm and tried a different approach to construct the two sets X and Y. We developed a Monte Carlo method [112]. It is a stochastic technique for solving problems for which an analytical solution is unknown or is too complex. In general, a Monte Carlo method performs statistical simulations using random numbers. To be more specific, in our case it repeatedly generates random possible solutions and evaluates their cost. At the end, the best solution is selected.

Each iteration of the Monte Carlo method starts by randomly distributing all the vertices into two equally sized sets. Each vertex in one set is connected to all vertices in the other set. The matching is then constructed by the Kuhn-Munkres algorithm without any problem. This process is run repeatedly; we recommend doing N iterations [135]. At the end, the solution with the lowest sum of pairwise distances is accepted. Figure 6.12 shows several instances of the matching with the best one framed in bold. The number at lower right shows the sum of distances.

The algorithm can be formally stated as follows. We are looking for the minimal Euclidean matching $M_{\min}$.

1. Initialise the minimal cost $c_{\min}$ to positive infinity.
2. Randomly distribute the vertices into two equally sized sets X and Y. Make the set of edges $E = \{\{x, y\} | x \in X, y \in Y\}$, i.e., make an edge from every vertex $x \in X$ to every vertex $y \in Y$.
3. In the bipartite graph $G = (X \cup Y, E)$ construct matching M by the Kuhn-Munkres algorithm. Let the cost of the matching be c.
4. If $c < c_{\min}$, set $c_{\min} := c$ and $M_{\min} := M$.
5. While a stopping condition has not been met, go to 2.
6. Output $M_{\min}$.

According to our empirical experiments, it is necessary to perform N or perhaps $10N$ iterations of the Monte Carlo method. More detailed experiments can be found in our recent paper [135].

Figure 6.13 shows an example of the matching including the hierarchy. The first frame shows the matching alone, the second one shows the progress of

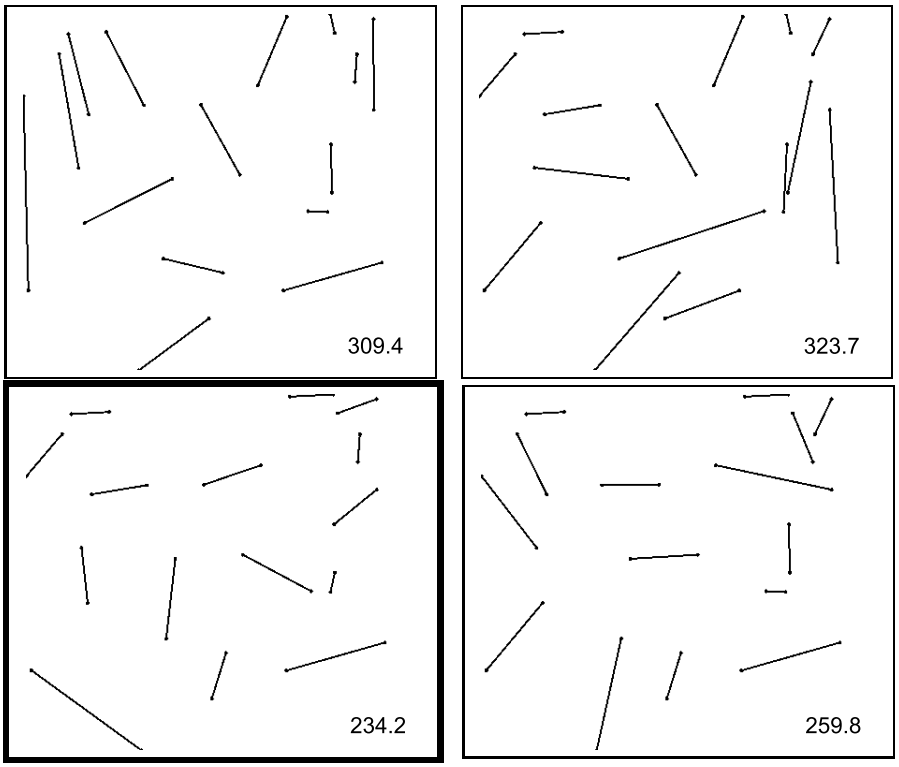


Figure 6.12: Example of several matching instances tried by the Monte Carlo method.

building the hierarchy, finally the last frame shows a complete tree. Especially the middle frame demonstrates that the space partitioning works well.

Due to the high complexity of the method, it is not practically possible to run it directly on large data. We use the hierarchical clustering technique to process the data in pieces. First, clusters are identified in the data. The matching is then constructed within each cluster separately. Then a matching among particular clusters is constructed to merge the results together. The output is a binary tree that defines the space partitioning for the ray tracing. Our first implementation is capable of processing 5 million vertices in several hours. The proposed solution is currently being evaluated by Jiří Bittner.

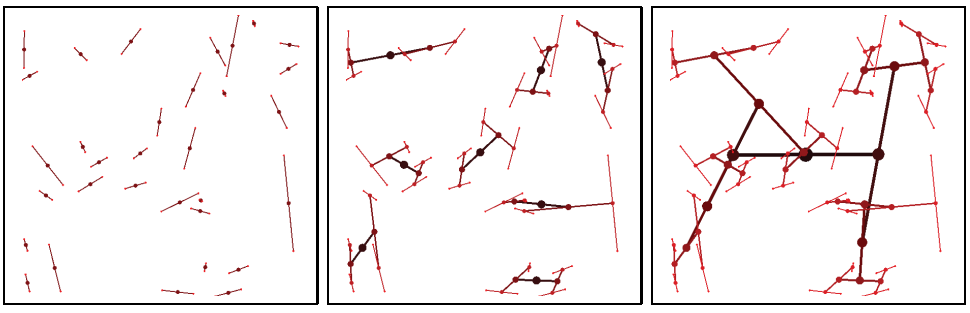


Figure 6.13: Example of matching and building the hierarchy.

Chapter 7

Conclusion

A conclusion comes at the end to summarise presented knowledge and review the work done so far. An outlook to the future follows where we introduce the interesting challenges we would perhaps take up next.

7.1 Summary of the work done

This work presented a thorough state of the art concerning large data in computer graphics. It started with a lightweight overview of application areas, methods of acquisition and general approaches to manipulation. It then concentrated on data streams as a technique for processing really huge amounts of data. The next chapter described fundamental clustering concepts and further focused on methods suitable for large data and particularly data streams. A chapter on Delaunay triangulation was included for completeness since it is a substantial part of the proposed solution. The chapter recalls the notoriously known methods of construction along with point location strategies and discusses their suitability for large data. It ends with a description of the streaming Delaunay triangulation as one of the few applications of data streams in computer graphics.

The last but one chapter presents the contributions we made to large data manipulation. It was namely the idea to use data stream clustering to hierarchically reduce the amount of data. We adapted the algorithm to better fit our needs. We then developed the dynamic hierarchical triangulation. It utilises the hierarchy of clusters to provide a triangulation with varying level of detail. So far the solution is only 2D; we are going to extend it into 3D in the future.

As a step aside we extended the clustering and the triangulation so that they can compute with anisotropic (elliptical) metrics. It considerably extends possibilities especially for the clustering since it allows to create elliptically elongated clusters. These can fit a wider variety of applications where spherical clusters would be inadequate.

7.2 Perspective to the future

In the future we would like to make further research on the dynamic hierarchical triangulation. One particular concern is the expansion and collapse of clusters. We would like to further investigate the possibility of inserting more points of a cluster at the same time; ideally the whole pre-triangulated cluster. Unfortunately, this does not seem to be possible because some clusters may overlap. The clustering algorithm cannot ensure that there will be no overlaps due to the data stream input – data are processed in independent pieces. Upon removing points, we already can remove more of them at the same time, nevertheless, there seems to be place for further improvements.

It would be nice during the visualisation, if the cluster expansions and collapses were done automatically based on the distance from the viewer or the presence on the object contour. We suppose this to be rather easy.

For sure we will work on extending the triangulation to 3D. This will be, on the other hand, a more difficult task.

To our pleasure we have a good feedback on our clustering of large data. Our colleague Michal Zemek already uses it for acceleration of tunnel searching in protein molecules [148]. We currently work on a modification for space partitioning for Jiří Bittner from the CTU in Prague to try it for ray tracing acceleration. A preliminary result may be seen in Figure 6.13. Another interesting extension comes from this application – to cluster not just points but also more complex objects such as a triangle soup. This requires to develop a different distance measure for the clustering algorithm.

A bachelor thesis on clustering in digital images is currently in progress. It focuses on an application of the clustering for segmentation in digital images and for image compression. One of the interesting challenges is how to incorporate the colour information into the distance measure. Yet another formula was developed for this purpose. A preliminary result of our work can be seen in Figure 7.1. The original image is on the left, the clustered image is on the right. The areas of constant colour are the particular clusters. The colour is the average colour of all the pixels in the cluster. Here we would like to thank Anders Hast from the University of Gävle, Sweden, for interesting ideas and inspiration.

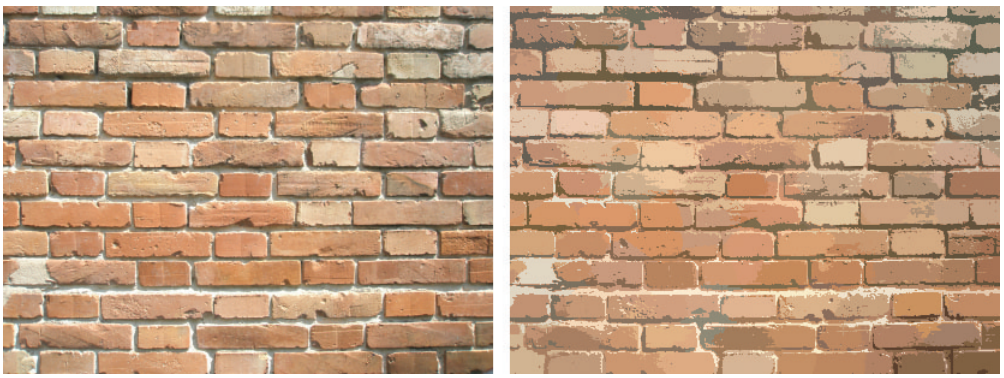


Figure 7.1: The original image and the clusters found in it.

We register further interest in the clustering. Our colleague Libor Váša believes it could contribute to his compression of dynamic triangle meshes. The task is to cluster vertices of the triangular mesh so that points in a cluster are all connected by edges of the mesh. In other words, every cluster should represent a connected subset of the mesh. We have another idea from our colleague Martin Janda who might use the clustering to reduce the huge amount of data in digital hologram synthesis. The requirement is not to damage sharp edges.

We realise that perhaps the clustering will not be perfectly suitable for all of the proposed applications. There are other approaches in solving some of the tasks and the clustering is presented as an alternative. We will focus on the applications where the clustering brings perspective results and where the appropriate colleague is keen to continue with further development.

Bibliography

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3(1):293–327, 1988.

[2] M. Ajtai, T. S. Jayram, R. Kumar, and D. Sivakumar. Approximate counting of inversions in a data stream. In *STOC '02: Proceedings of the 34th annual ACM symposium on Theory of computing*, pages 370–379, New York, NY, USA, 2002. ACM.

[3] K. S. Al-Sultan. A Tabu search approach to the clustering problem. *Pattern Recognition*, 28(9):1443–1451, 1995.

[4] N. Amenta, S. Choi, and G. Rote. Incremental constructions con BRIO. In *ACM Symposium on Computational Geometry*, pages 211–219, 2003.

[5] G. Ball and D. Hall. ISODATA: A Novel method of data analysis and pattern classification. Technical report, Stanford Research Institute, Menlo Park, 1965.

[6] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. *Randomization and Approximation Techniques in Computer Science*, 2483:952–961, 2002.

[7] A. Baraldi and P. Blonda. A survey of fuzzy clustering algorithms for pattern recognition. ii. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 29(6):786–801, 1999.

[8] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[9] J. Bezdek. *Pattern recognition with fuzzy objective function algorithms*. Kluwer Academic Publishers Norwell, MA, USA, 1981.

[10] J. Bezdek and R. Ehrlich. FCM: The fuzzy c-means clustering algorithm. *Computers & Geosciences*, 10(2):191–203, 1984.

[11] M. Botsch, A. Wiratanaya, and L. Kobbelt. Efficient high quality rendering of point sampled geometry. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, pages 53–64, Aire-la-Ville, Switzerland, 2002. Eurographics Association.

[12] K. Q. Brown. Voronoi diagrams from convex hulls. *Information Processing Letters*, 9(5):223–228, 1979.

[13] D. R. Chand and S. S. Kapur. An algorithm for convex polytopes. *Journal of the ACM*, 17(1):78–86, 1970.

[14] J. Chang and W. Lee. Finding recent frequent itemsets adaptively over online data streams. In *Proceedings of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 487–492. ACM New York, NY, USA, 2003.

[15] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Automata, Languages and Programming*, 2380:784–794, 2002.

[16] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *IEEE Symposium on Foundations of Computer Science*, pages 378–388, 1999.

[17] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: How much is enough? In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 436–447, New York, NY, USA, 1998. ACM.

[18] H. H. Chen and T. S. Huang. A survey of construction and manipulation of octrees. *Computer Vision, Graphics, and Image Processing*, 43(3):409–431, 1988.

[19] F. A. Chudak. Improved approximation algorithms for uncapacitated facility location. *Lecture Notes in Computer Science*, 1412:180–194, 1998.

[20] P. Cignoni, C. Montani, and R. Scopigno. DeWall: A fast divide and conquer Delaunay triangulation algorithm in E^d. *Computer Aided Design*, 30:333–342, 1998.

[21] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[22] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, and W. Wright. Simplification envelopes. *Computer Graphics*, 30(Annual Conference Series):119–128, 1996.

[23] G. Cormode, M. Datar, P. Indyk, and S. Muthukrishnan. Comparing data streams using Hamming norms (How to zero in). *IEEE Transactions on Knowledge and Data Engineering*, 15(3):529–540, 2003.

[24] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding hierarchical heavy hitters in streaming data. *ACM Transactions on Knowledge Discovery from Data*, 1(4):1–48, 2008.

[25] G. Cormode and S. Muthukrishnan. An Improved data stream summary: The Count-min sketch and its applications. *LATIN 2004: Theoretical Informatics*, pages 29–38, 2004.

[26] G. Cormode and S. Muthukrishnan. What's hot and what's not: Tracking most frequent items dynamically. *ACM Transactions on Database Systems (TODS)*, 30(1):249–278, 2005.

[27] G. Cormode, S. Muthukrishnan, and S. Sahinalp. Permutation editing and matching via embeddings. *Lecture Notes in Computer Science*, pages 481–492, 2001.

[28] C. Cortes and D. Pregibon. Signature-based methods for data streams. *Data Mining and Knowledge Discovery*, 5(3):167–182, 2001.

[29] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *SIAM Journal on Computing*, pages 635–644, 2002.

[30] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry*, chapter 14: Quadtrees, pages 291–306. Springer-Verlag, 2nd edition, 2000.

[31] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry*, chapter 5.2: Kd-Trees, pages 99–105. Springer-Verlag, 2nd edition, 2000.

[32] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry*, chapter 12: Binary Space Partitions, pages 251–265. Springer-Verlag, 2nd edition, 2000.

[33] B. N. Delaunay. Sur la sphère vide. *Izvestia Akademia Nauk SSSR*, 7:793–800, 1934.

[34] O. Devillers. The Delaunay hierarchy. *International Journal of Foundations of Computer Science*, 13:163–180, 2002. special issue on triangulations.

[35] O. Devillers, S. Pion, and M. Teillaud. Walking in a triangulation. In *SCG '01: Proceedings of the seventeenth annual symposium on computational geometry*, pages 106–114, New York, NY, USA, 2001. ACM.

[36] E. Diday and J. Simon. 3. Clustering analysis. *Digital Pattern Recognition*, pages 47–94, 1976.

[37] R. Dubes and A. Jain. Clustering methodologies in exploratory data analysis. *Advances in Computers*, 19:113–228, 1980.

[38] B. Duran and P. Odell. Cluster analysis: A Survey. *Lecture Notes in Economics and Mathematical Systems*, 1974.

[39] R. A. Dwyer. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. *Algorithmica*, 2(1):137–151, 1987.

[40] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *J. of Res. the Nat. Bureau of Standards*, 69B:125–130, 1965.

[41] M. Ester, H. Kriegel, J. Sander, and X. Xu. A Density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining*, pages 226–231, 1996.

[42] T. Fang and L. Piegl. Delaunay triangulation using a uniform grid. *IEEE Computer Graphics and Applications*, 13(3):36–47, 1993.

[43] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery: An overview. *Advances in Knowledge Discovery and Data Mining*, pages 1–34, 1996.

[44] J. Feigenbaum, S. Kannan, and J. Zhang. Computing diameter in the streaming and sliding-window models. *Algorithmica*, 41(1):25–41, 2004.

[45] R. Finkel and J. Bentley. Quad trees: A Data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[46] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[47] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 124–133, New York, NY, USA, 1980. ACM.

[48] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Towards an adaptive approach for mining data streams in resource constrained environments. *Data Warehousing and Knowledge Discovery*, pages 189–198, 2004.

[49] H. N. Gabow. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *Journal of the ACM*, 23(2):221–234, 1976.

[50] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. *Computer Graphics*, 31(Annual Conference Series):209–216, 1997.

[51] C. Giannella, J. Han, J. Pei, X. Yan, and P. Yu. Mining frequent patterns in data streams at multiple time granularities. *Next Generation Data Mining*, 212:191–212, 2003.

[52] P. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proceedings of the International Conference on Very Large Data Bases*, pages 541–550, 2001.

[53] A. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of the 34th annual ACM symposium on Theory of computing*, pages 389–398. ACM New York, NY, USA, 2002.

[54] D. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[55] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 58–66, New York, NY, USA, 2001. ACM.

[56] J. P. Grossman and W. J. Dally. Point sample rendering. In G. Drettakis and N. L. Max, editors, *Proceedings of Eurographics Workshop on Rendering*, pages 181–192. Springer, 1998.

[57] S. Guha and S. Khuller. Greedy strikes back: Improved facility location algorithms. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, pages 649–657, 1998.

[58] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proceedings of the 33rd annual ACM symposium on Theory of computing*, pages 471–475. ACM New York, NY, USA, 2001.

[59] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams: Theory and practice. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):515–528, 2003.

[60] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *IEEE Symposium on Foundations of Computer Science*, pages 359–366, 2000.

[61] S. Guha, R. Rastogi, and K. Shim. CURE: An efficient clustering algorithm for large databases. In *ACM SIGMOD International Conference on Management of Data*, pages 73–84, 1998.

[62] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[63] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(1):381–413, 1992.

[64] A. Gupta and F. X. Zane. Counting inversions in lists. In *SODA '03: Proceedings of the 14th annual ACM-SIAM symposium on Discrete algorithms*, pages 253–254, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[65] J. A. Hartigan. *Clustering Algorithms*. John Wiley & Sons, 1975.

[66] J. Holenda. *O maticích*. Vydavatelský servis, Plzeň, 2007.

[67] H. Hoppe. Progressive meshes. *Computer Graphics*, 30(Annual Conference Series):99–108, 1996.

[68] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 35–42, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[69] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. *Computer Graphics*, 27(Annual Conference Series):19–26, 1993.

[70] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. Kirchner, and J. Klosowski. Chromium: A Stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics*, 21(3):693–702, 2002.

[71] A. Imiya and T. Sakai. Combinatorial properties of scale space singular points. *Combinatorial Image Analysis*, pages 333–346, 2006.

[72] P. Indyk. Algorithms for dynamic geometric problems over data streams. In *STOC '04: Proceedings of the 36th annual ACM symposium on Theory of computing*, pages 373–380, New York, NY, USA, 2004. ACM.

[73] P. Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *Journal of the ACM*, 53(3):307–323, 2006.

[74] M. Isenburg and P. Lindstrom. Streaming meshes. In *Visualization '05*, pages 231–238, 2005.

[75] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Shewchuk. Streaming compression of tetrahedral volume meshes. In *Proceedings of Graphics Interface 2006*, pages 115–121. Canadian Information Processing Society Toronto, Ontario, Canada, 2006.

[76] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink. Large mesh simplification using processing sequences. *IEEE Visualization, 2003*, pages 465–472, 2003.

[77] M. Isenburg, P. Lindstrom, and J. Snoeyink. Streaming compression of triangle meshes. In *ACM SIGGRAPH 2005 Sketches*, page 136, New York, NY, USA, 2005. ACM.

[78] M. Isenburg, Y. Liu, J. R. Shewchuk, and J. Snoeyink. Streaming computation of Delaunay triangulations. *ACM Transactions on Graphics*, 25(3):1049–1056, 2006.

[79] A. Jain and P. Flynn. Image segmentation using clustering. *Advances in Image Understanding: A Festschrift for Azriel Rosenfeld*, pages 65–83, 1996.

[80] A. Jain and J. Mao. Neural networks and pattern recognition. *Computational Intelligence: Imitating Life*, pages 194–212, 1994.

[81] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall advanced reference series. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[82] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A Review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[83] K. Jain and V. V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. In *IEEE Symposium on Foundations of Computer Science*, pages 2–13, 1999.

[84] C. Jin, W. Qian, C. Sha, J. Yu, and A. Zhou. Dynamically maintaining frequent items over a data stream. In *Proceedings of the 12th international conference on Information and knowledge management*, pages 287–294. ACM New York, NY, USA, 2003.

[85] V. Karamcheti, D. Geiger, Z. Kedem, and S. Muthukrishnan. Detecting malicious network traffic using inverse distributions of packet contents. In *MineNet '05: Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*, pages 165–170, New York, NY, USA, 2005. ACM.

[86] L. Kaufman and P. Rousseeuw. *Finding Groups in Data: an introduction to cluster analysis*. John Wiley & Sons, 1990.

[87] B. King. Step-wise clustering procedures. *Journal of the American Statistical Association*, 62(317):86–101, 1967.

[88] J. Kohout. Selected problems of parallel computer graphics. Technical report, University of West Bohemia, Univerzitní 22, Pilsen, 2004.

[89] J. Kohout. *Delaunay Triangulation in Parallel and Distributed Environment*. PhD thesis, University of West Bohemia, Pilsen, Czech Republic, 2005.

[90] Y.-M. Koo and B.-S. Shin. An Efficient point rendering using octree and texture lookup. In *Computational Science and Its Applications - ICCSA 2005*, volume 3482 of *Lecture Notes in Computer Science*, pages 1187–1196. Springer, 2005.

[91] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Analysis of a local search heuristic for facility location problems. In *SODA: ACM-SIAM Symposium on Discrete algorithms*, pages 1–10, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[92] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistic Quarterly*, 2:83–97, 1955.

[93] Lakes Environmental – Digital Terrain Data. `http://www.weblakes.com/lakesdem.html`.

[94] A. Lakhina, M. Crovella, and C. Diot. Mining anomalies using traffic feature distributions. *SIGCOMM Computer Communication Review*, 35(4):217–228, 2005.

[95] G. Lance and W. Williams. A General theory of classificatory sorting strategies: 1. Hierarchical systems. *The Computer Journal*, 9(4):373–380, 1967.

[96] C. Lawson. Software for c^1 surface interpolation. *Mathematical Software*, 3:161–194, 1977.

[97] J. Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 259–266, Washington, DC, USA, 2002. IEEE Computer Society.

[98] M. Levoy and T. Whitted. The use of points as rendering primitives. Technical Report TR 85-022, Department of Computer Science, University of North Carolina at Chapel Hill, 1985.

[99] J.-H. Lin and J. S. Vitter. Approximation algorithms for geometric median problems. *Information Processing Letters*, 44:245–249, 1992.

[100] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 109–118, New York, NY, USA, 1996. ACM.

[101] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. *Computer Graphics*, 31:199–208, 1997.

[102] D. Luebke, B. Watson, J. D. Cohen, M. Reddy, and A. Varshney. *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA, 2002.

[103] M. O. M. de Berg, M. van Kreveld and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 1997.

[104] J. B. Macqueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[105] P. Mahalanobis. On the generalized distance in statistics. *Proceedings of the National Institute of Science of India*, 12:49–55, 1936.

[106] M. Mahdian, E. Markakis, A. Saberi, and V. Vazirani. A greedy facility location algorithm analyzed using dual fitting. *Lecture Notes in Computer Science*, 2129:127–137, 2001.

[107] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 346–357. VLDB Endowment, 2002.

[108] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, July 2008. Also available online at `http://informationretrieval.org/`.

[109] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*, chapter 17 Hierarchical clustering. Cambridge University Press, July 2008. Also available online at `http://informationretrieval.org/`.

[110] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*, chapter 16 Flat clustering. Cambridge University Press, July 2008. Also available online at `http://informationretrieval.org/`.

[111] D. H. McLain. Two dimensional interpolation from random data. *The Computer Journal*, 19(2):178–181, 1976.

[112] N. Metropolis and S. Ulam. The monte carlo method. *Journal of the American Statistical Association*, 44(247):335–341, 1949.

[113] A. Meyerson. Online facility location. In *FOCS '01: IEEE Symposium on Foundations of Computer Science*, pages 426–431, Washington, DC, USA, 2001. IEEE Computer Society.

[114] R. Michalski, R. Stepp, and E. Diday. A recent advance in data analysis: Clustering objects into classes characterized by conjunctive concepts. *Progress in Pattern Recognition*, 1:33–55, 1981.

[115] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[116] J. Munkres. *Topology*. Prentice Hall, 2nd edition, December 1999.

[117] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers Inc, 2005. Also issued as *Foundations and trends in theoretical computer science*, 1(2), 2005. Manuscript available at `http://www.cs.rutgers.edu/~muthu/stream-1-1.ps`.

[118] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *20th International Conference on Very Large Data Bases*, pages 144–155, Los Altos, CA 94022, USA, 1994. Morgan Kaufmann Publishers.

[119] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani. Streaming-data algorithms for high-quality clustering. In *IEEE International Conference on Data Engineering*, pages 685–694, 2002.

[120] R. Pajarola. Efficient level-of-details for point based rendering. In *Computer Graphics and Imaging*, pages 141–146. IASTED/ACTA Press, 2003.

[121] R. Pajarola. Stream-processing points. In *IEEE Visualization*, pages 239–246. IEEE Computer Society, 2005.

[122] N. H. Park and W. S. Lee. Statistical grid-based clustering over data streams. *ACM SIGMOD Record*, 33(1):32–37, 2004.

[123] F. Preparata and M. Shamos. *Computational geometry: An Introduction*. Springer, 1985.

[124] E. Rasmussen. Clustering algorithms. *Information retrieval: Data structures and algorithms*, pages 419–442, 1992.

[125] K. Rose, E. Gurewitz, and G. C. Fox. Deterministic annealing approach to constrained clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15:785–794, 1993.

[126] J. R. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering complex scenes. In *Geometric Modelling in Computer Graphics*, pages 455–465. Springer Verlag, 1993.

[127] S. Rusinkiewicz and M. Levoy. QSplat: A Multiresolution point rendering system for large meshes. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 343–352, New York, NY, USA, 2000. ACM Press / Addison-Wesley Publishing Co.

[128] E. Ruspini. A new approach to clustering. *Information and control*, 15(1):22–32, 1969.

[129] M. Sainz and R. Pajarola. Point-based rendering techniques. *Computers & Graphics*, 28(6):869–879, 2004.

[130] D. Schmalstieg and G. Schaufler. Smooth levels of detail. In *In Proc. of IEEE 1997 Virtual Reality Annual International Symposium*, pages 12–19. IEEE Computer Society Press, 1997.

[131] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. *Computer Graphics*, 26(2):65–70, 1992.

[132] D. B. Shmoys. Approximation algorithms for facility location problems. In *APPROX '00: Approximation Algorithms for Combinatorial Optimization*, volume 1913 of *Lecture Notes in Computer Science*, pages 27–33, London, UK, 2000. Springer-Verlag.

[133] D. B. Shmoys, É. Tardos, and K. Aardal. Approximation algorithms for facility location problems (extended abstract). In *ACM Symposium on Theory of Computing*, pages 265–274, 1997.

[134] J. Skála and I. Kolingerová. Clustering geometric data streams. In *SIGRAD 2007*, pages 17–23, 2007.

[135] J. Skála, I. Kolingerová, and J. Hyka. A Monte Carlo solution to the minimal Euclidean matching. In *ALGORITMY 2009*, pages 402–411, 2009.

[136] P. H. A. Sneath and R. R. Sokal. *Numerical taxonomy: The principles and practice of numerical classification.* W.H. Freeman, San Francisco, 1973.

[137] Stanford CG laboratory data archives. `http://graphics.stanford.edu/data/`.

[138] G. Turk. Re-tiling polygonal surfaces. *Computer Graphics*, 26(2):55–64, 1992.

[139] USGS (U.S. Geological Survey) EROS. `http://edc.usgs.gov/products/elevation/`.

[140] M. Vigo Anglada and N. Pla Garcia. Computing directional constrained Delaunay triangulations. *Computers & Graphics*, 24(2):181–190, 2000.

[141] The Visible human project. `http://www.nlm.nih.gov/research/visible/`.

[142] G. Voronoi. Nouvelles applications des parametres continus a la theorie des formes quadratiques. *J. reine angew. Math*, 134:198–287, 1908.

[143] The Walkthru project. `http://www.cs.unc.edu/~walk/`.

[144] D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *The Computer Journal*, 24(2):167–172, 1981.

[145] WebGIS – Free Terrain Data. `http://www.webgis.com/terr_world.html`.

[146] K. Xu, Z.-L. Zhang, and S. Bhattacharyya. Profiling internet backbone traffic: Behavior models and applications. *SIGCOMM Computer Communication Review*, 35(4):169–180, 2005.

[147] M. Zadravec and B. Žalik. An Almost distribution-independent incremental Delaunay triangulation algorithm. *The Visual Computer*, 21(6):384–396, 2005.

[148] M. Zemek, J. Skála, I. Kolingerová, P. Medek, and J. Sochor. Fast method for computation of channels in dynamic proteins. In *Vision, Modeling, and Visualization 2008*, pages 333–342, 2008.

[149] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An efficient data clustering method for very large databases. In *ACM SIGMOD International Conference on Management of Data*, pages 103–114, 1996.

Appendix A

Pseudo-codes

This part presents pseudo-codes for the Local Search clustering algorithm.

A.1 The *gain* function

The *gain* function is the core of the Local Search clustering algorithm. Here is a detailed pseudo-code of the function.

function gain(point p)

variables

`p`	the point for which to compute gain
`gain`	determines whether to open a facility at p
`facilityCost`	global read-only variable, the facility cost
`distanceSpare`	the distance we spare by reassigning a point to p
`listReassign`	the list of points that should be reassigned to p
`closeSpare`	the cost we can spare by closing a facility and reassigning its points to p
`listClose`	the list of facilities that should be closed and their points reassigned to p
`f.accumulator`	the accumulator of costs for reassigning points from facility f to facility p

code

```
if there is already a facility at p
    gain = 0
else
    gain = - facilityCost

// compute distance spares
for each point q in the input set (not excluding p)
{
    // find the current facility
    fq = facility for q
```

```
    // compute the difference between the distances
    // to the current facility and to the facility candidate
    distanceSpare = distance(q, fq) - distance(q, p)

    if distanceSpare > 0
        // we will spare by reassigning q to p
        add q to listReassign
        gain += distanceSpare
    else
        // add the cost to the facility accumulator
        fq.acculumator += distanceSpare
        // note that distanceSpare is negative
        // (now it is a cost, not a spare)
}

// compute close spares
for each facility f
{
    // facilityCost is what we can spare by closing f,
    // accumulator holds the cost for reassigning points from f to p
    closeSpare = facilityCost + f.accumulator
    // note that f.accumulator is negative
    // (it is a cost for the reassignments)

    if closeSpare > 0
        // we will spare by closing f
        add f to listClose
        gain += closeSpare
}
```

A.2 The Local Search clustering algorithm

The following pseudo-code describes the complete Local Search algorithm from a higher level point of view.

```
generate initial solution

// local search improvements
repeat N log N times
{
    pick a point p at random

    compute gain(p)

    if gain(p) > 0
        perform reassignments
}
```

Appendix B

Clustering of the world

This example shows the clustering of 22 million points of the digital elevation map of the whole world. The data can be obtained from one of the following sites [145, 93, 139]. The processing was done in three levels (the input stream plus two more levels) and took about half an hour on a Pentium 4 3.2 GHz processor.

Figure B.1 shows the first level (the input stream is level zero) of the clustering containing about 80 000 points. You may notice negligible breaks between some clusters. This is because the data are provided in 33 blocks. Little discontinuities may occur where the blocks meet.

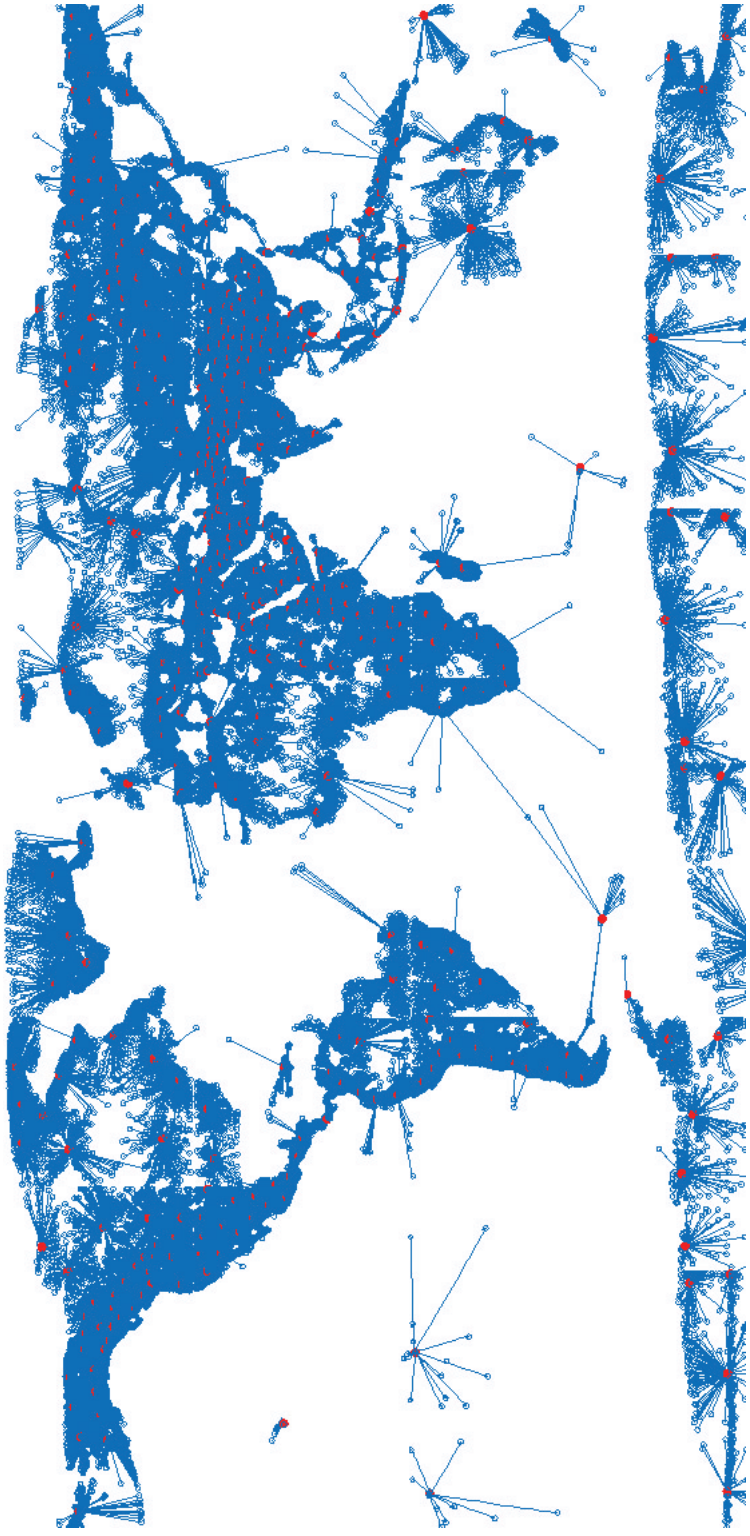


Figure B.1: Clustering of the whole world.

Appendix C

Professional activities

Reviewed publications

J. Skála, I. Kolingerová, and J. Hyka. A Monte Carlo solution to the minimal Euclidean matching. In *ALGORITMY 2009*, pages 402–411, 2009.

M. Zemek, J. Skála, I. Kolingerová, P. Medek, and J. Sochor. Fast Method for Computation of Channels in Dynamic Proteins. In *Vision, Modeling, and Visualization 2008*, pages 333–342, 2008.

J. Skála and I. Kolingerová. Clustering geometric data streams. In *SIGRAD 2007*, pages 17–23, 2007.

Other technical publications

J. Skála. Tvorba zásuvných modulů pro Adobe Photoshop. Students' bulletin, University of West Bohemia, Univerzitní 22, Pilsen, 2008.

J. Skála. Úprava nahrávky z televizní karty. Available online at `http://informatika.zcu.cz/pro_studenty/zajimavosti/uprava_nahravky.html`, 2007.

J. Skála. Masking images for DTP needs: Implemented as Adobe Photoshop plug-in. Master's thesis, University of West Bohemia, Univerzitní 22, Pilsen, 2006. Supervisor Petr Lobaz.

Stays abroad

University of Maribor, Slovenia, November 15–21, 2007

MADALGO Summer School 2007 on DATA STREAM ALGORITHMS, Århus, Denmark, August 19–24, 2007

Significant scientific talks

Hierarchical Triangulation of Clustered Data. At the University of West Bohemia, Pilsen, Czech Republic, June 4, 2008.

Clusterování data streamů a hierarchická triangulace. At the VŠB – Technical University of Ostrava, November 26, 2008.

Hierarchical Clustering of Large Geometric Data. At the University of Maribor, Slovenia, November 11, 2007.

Poster Clustering geometric data streams. At the MADALGO Summer School 2007 on DATA STREAM ALGORITHMS, Århus, Denmark, August 20, 2007.

Clustering Geometric Data Streams. At the University of West Bohemia, Pilsen, Czech Republic, May 29, 2007.

Participation on scientific projects

Triangulated Models for Haptic and Virtual Reality. Project leader Ivana Kolingerová. Funded by The Czech Science Foundation (GACR), project code 201/09/0097.

VIRTUAL – Virtual Research-Educational Center of Computer Graphics and Visualization. Project leader Václav Skala. Funded by The Ministry of Education, Youth and Sports (MSMT), project code 2C 06002.

Bilateral Cooperation in Computational Geometry Research for Visualization. Project leader Ivana Kolingerová. Funded by The Ministry of Education, Youth and Sports (MSMT), project code KONTAKT 5/2005-06.

CPG – Center of Computer Graphics – National Network of Fundamental Research Centers. Project leader Václav Skala. Funded by The Ministry of Education, Youth and Sports (MSMT), project code LC 06008.

Other scientific or academic activities

Data Stream Hierarchical Clustering Library. Authorised software made freely available to the public.

Adviser for the bachelor thesis Využití shlukování pro digitalizované obrazy by Pavel Hulej. Supervisor Ivana Kolingerová.