A Potential-Field-Based Multilevel Algorithm for Drawing Large Graphs

Inaugural-Dissertation
zur
Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Universität zu Köln

vorgelegt von
Stefan Hachul
aus Kerpen

Köln 2005

Berichterstatter:	**Prof. Dr. Michael Jünger** **HD Dr. Bert Randerath**
Tag der mündlichen Prüfung:	**1. Februar 2005**

Acknowledgment

I would like to thank everyone who supported me in writing this thesis. In particular, many thanks are addressed to my supervisor Prof. Dr. Michael Jünger for giving me the possibility to research in the field of force-directed graph drawing and for the constructive and helpful discussions in the internal seminars.

I am very grateful to Dr. Sebastian Leipert for bringing me to the research field of automatic graph drawing and to HD Dr. Bert Randerath and Claudia Schlesiger for their proof-reading.

Many thanks go to Thomas Lange, Constantin Hellweg, Holger Flier, and Annette Menze for supporting me in all technical questions. Without their assistance and professional work, this dissertation would probably still be in progress.

I thank Michael Belling and Ursula Neugebauer for helping me in the acquisition of literature and all administrative processes.

A big thank goes also to Dr. Christoph Buchheim, Matthias Elf, Dr. Frauke Liers, Merijam Percan, and all previous mentioned colleagues for their "open doors" in the last years in Cologne.

Furthermore, I would like to thank Yehuda Koren, Daniel Tunkelang, and Roman Yusufov for making me available the implementations of their algorithms.

Finally, I am very grateful to my girlfriend Claudia Schlesiger and my parents Gerda Hachul and Paul Hachul for supporting me in all aspects of my life.

Abstract

The aim of automatic graph drawing is to compute a well-readable layout of a given graph $G = (V, E)$. One very popular class of algorithms for drawing general graphs are force-directed methods. These methods generate drawings of G in the plane so that each edge is represented by a straight line connecting its two adjacent nodes. The computation of the drawings is based on associating G with a physical model. Then, the algorithms iteratively try to find a placement of the nodes so that the total energy of the physical system is minimal. Several force-directed methods can visualize large graphs containing many thousands of vertices in reasonable time. However, only some of these methods guarantee a sub-quadratic running time in special cases or under certain assumptions, but not in general. The others are not sub-quadratic at all.

We develop a new force-directed algorithm that is based on a combination of an efficient multilevel strategy and a method for approximating the repulsive forces in the system by rapidly evaluating potential fields. The worst-case running time of the new method is $O(|V| \log |V| + |E|)$ with linear memory requirements. In practice, the algorithm generates nice drawings of graphs containing up to 100000 nodes in less than five minutes. Furthermore, it clearly visualizes even the structures of those graphs that turned out to be challenging for other tested methods.

Zusammenfassung

Das Ziel beim Automatischen Zeichnen von Graphen besteht darin, zu einem gegebenen Graphen $G = (V, E)$ eine Zeichnung zu berechnen, welche seine Struktur leicht verständlich visualisiert. Eine sehr verbreitete Klasse von Methoden zum automatischen Zeichnen von allgemeinen Graphen sind die so genannten kräftebasierten Verfahren. Diese Verfahren erzeugen Zeichnungen von G, bei denen Kanten durch Geraden repräsentiert werden. Dabei wird zunächst G mit einem physikalischen Modell identifiziert. Anschließend versuchen diese Algorithmen iterativ eine Platzierung der Knoten zu finden, so dass die Gesamtenergie des induzierten physikalischen Systems minimal ist. Einige kräftebasierte Verfahren können Graphen mit mehreren tausend Knoten in angemessener Zeit visualisieren. Allerdings garantieren nur manche dieser Verfahren in speziellen Fällen oder unter bestimmten Annahmen — jedoch nicht im Allgemeinen — eine subquadratische Laufzeit. Die anderen haben in keinem Fall ein subquadratisches Laufzeitverhalten.

Wir entwickeln einen neuen kräftebasierten Algorithmus, der auf der Kombination einer effizienten Multilevelstrategie mit einer Methode zur approximativen Berechnung abstoßender Kräfte durch eine schnelle Auswertung von Potentialfeldern basiert. Die obere asymptotische Laufzeitschranke ist $O(|V| \log |V| + |E|)$ bei linearem Speicherplatzverbrauch. Praktische Experimente zeigen, dass die neue Methode Graphen mit bis zu 100000 Knoten in weniger als fünf Minuten übersichtlich darstellen kann. Zudem wird auch die Struktur solcher Graphen, die anderen getesteten Verfahren Probleme bereiten, leicht verständlich visualisiert.

Contents

Introduction

> Wenn es nur eine einzige Wahrheit gäbe,
> könnte man nicht hundert Bilder
> über das selbe Thema malen. [1]

"Ziemlich verknotet" is the headline of an article of the weekly journal DIE ZEIT from February 2004 (see [112] page 33) describing the growing importance of network theory in science. Networks or graphs are used to model information that can be described as objects and connections between those objects. In a graph the objects are represented by nodes and the connection between two objects by edges that link the corresponding nodes.

For example, the biochemical reactions of proteins in baker's yeast, the ecosystem of plankton, sea perch, and anchovy, the email correspondence of the students of a German university, the American electricity network, the international air traffic, and the world-wide web can be modeled as graphs and have recently been studied by network analysts [112].

One fundamental tool for analyzing such graphs is the automatic generation of layouts that visualize the graphs and are easy to understand. For example, software packages that are used to analyze networks like *Pajek* [9] or *visone* [15] contain algorithms that generate structural information associated with a given graph and algorithms that try to display these graphs in an easy readable way. The latter is the field of research of *automatic graph drawing* on which we will focus in the following.

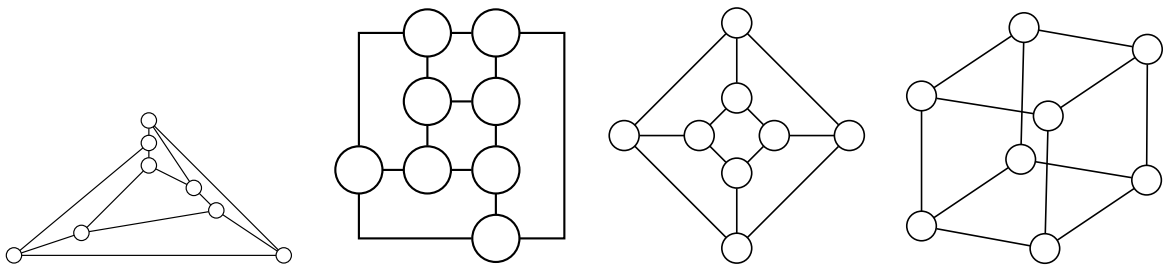
In a drawing of a given graph in the plane, the nodes are often represented by points, circles, or boxes, while the edges are represented by curves that connect the corresponding nodes. In order to help evaluating whether a given drawing of a graph is *nice*, some criteria have been defined that support this decision. Important criteria are the number of edge crossings, the number of edge bends, the used drawing area, the sum of the edge lengths, the uniformity of the edge lengths, the number of nodes that overlap each other, and the display of symmetries. The relevance of most of these criteria for the understandability of the drawings has been investigated and confirmed by several empirical studies [107, 108, 109].

An *ideal* drawing of a graph should have a minimum number of edge crossings, no edge bends, and should use few drawing area. The length of the edges should be as uniform as

[1] Pablo Picasso

possible, the nodes should not overlap, and the drawing should display the symmetries in a graph if some exist. However, since some of these criteria are in conflict with each other, such a two-dimensional drawing will never exist for all given graphs.

We will exemplify this in the following. Figure 1 shows different automatically generated drawings of a graph with 8 nodes and 12 edges that represents a cube.



(a) (b) (c) (d)

Figure 1: Different drawings of the same graph that represents a cube.

The drawing 1(a) is a *straight-line* drawing that does not contain any edge crossings, and the edges do not have any bends. However, the drawing contains some very short and some long edges. As a consequence, the nodes are drawn very small. Therefore, when scaling every drawing so that the nodes of each drawing are drawn in a predefined fixed size, the used drawing area of this drawing will be relatively big.

The drawing 1(b) does not contain any edge crossings, and the nodes are quite large, which indicates that the used drawing area will be relatively small, when scaling every drawing so that the nodes of each drawing are drawn in a predefined fixed size. The edge lengths are not uniform, and since the edges are drawn in an *orthogonal* style four edge bends appear.

Unlike the drawing 1(c), both previous mentioned drawings do not display the symmetry of the cube graph. Like the drawing 1(a), the drawing 1(c) is a straight-line drawing that contains no edge crossings and has non-uniform edge lengths.

The drawing 1(d) is a straight-line drawing that displays the symmetry of the cube graph and has uniform edge lengths. But unlike the other drawings, it contains two edge crossings.

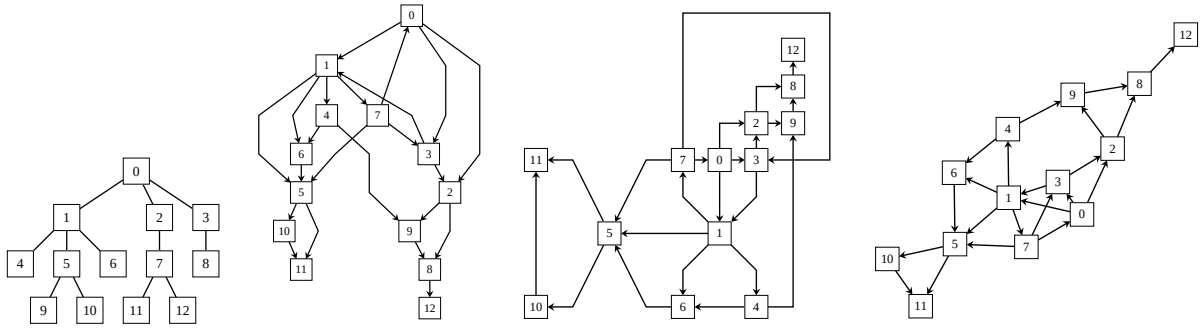
In general, the developers of graph-drawing algorithms and the users of these methods have to decide which criteria are most important to their applications. Therefore, a big variety of graph-drawing methods have been developed for different kinds of graph classes and aesthetic preferences. Some examples of different layout styles are presented in Figure 2.

Figure 2(a) shows a typical *tree layout* of a graph that describes a family tree. These layouts are generated for a special class of graphs called *trees*. They do not contain any edge crossings and have straight-line edges.

The *layered layout*, presented in Figure 2(b), is used to display hierarchical relationships that are associated with the edges of so called *directed* graphs. Furthermore, developers of algorithms that generate such layouts try to keep the number of edge crossings and edge bends as small as possible.

A *planarization-based layout* of a graph is presented in Figure 2(c). This layout style is used if it is very important that the number of edge crossings is minimized.

The graph shown in Figure 2(d) is drawn in the *force-directed layout* style. These drawings are characterized by straight-line edges, tend to have uniform edge lengths, and tend to display symmetries of the given graphs if some exist.



(a) (b) (c) (d)

Figure 2: (a) A tree layout of a family tree. (b - d) Different layouts of the same directed graph: (b) A hierarchical layout, (c) a planarization-based layout, and (d) a force-directed layout.

Several books are available that give a deeper introduction into the field of graph drawing [82, 30, 123, 79]. In [76] some of the state-of-the-art graph-drawing software tools are presented. The chapter *Technical Foundations* [77] in this book contains a short but comprehensive introduction into the field of graph drawing, too. Latest research results can be found in the annual conference proceedings of the *Graph Drawing* conferences that appear in the *Lecture Notes in Computer Science* series of *Springer-Verlag*, and [29] gives an overview of some of the most important publications in the field of graph drawing until 1994.

In the context of this dissertation, we will concentrate on algorithms that generate force-directed layouts and are called *force-directed methods*. Since most force-directed methods are very intuitive, easy to implement, applicable to nearly all kinds of graphs, and often generate pleasing drawings — regarding the modeled criteria of the force-directed layout style — they are quite popular.

These algorithms have in common that the computation of the drawings is based on associating a given graph with a physical model. For example, many methods associate the nodes with equally charged particles and the edges with springs. Then, these algorithms try to find a placement of the nodes so that the total energy of the physical system has a

(local) minimum value. Since the edges of the graph are drawn straight-line, the positions of the nodes in the drawing plane are sufficient to determine the final drawing.

Unfortunately, early versions of force-directed algorithms are not suitable for drawing large graphs containing several hundreds or thousands of vertices, since their running times grow — at least — quadratic in the number of nodes of the given graph. To give an impression what this could mean in practice, we present a simple arithmetic example: Suppose that such a quadratic algorithm needs 0.1 seconds for drawing a graph containing 100 nodes. Then, the same method would need 27 hours and 46 minutes for drawing a graph containing 100000 nodes. Another algorithm that needs 0.1 seconds for drawing a graph with 100 nodes, but with a linear running time, would need only 1 minute and 40 seconds for drawing a graph that contains 100000 nodes.

Since the demand on algorithms that can generate drawings of such large graphs is rapidly increasing, researchers have developed new force-directed methods that have improved the running times and the quality of the drawings. These algorithms generate pleasing drawings of several classes of large graphs in reasonable time. However, only some of these methods guarantee a sub-quadratic running time in special cases or under certain assumptions, but not in general. The others are not sub-quadratic at all.

This was the starting point of the presented dissertation and implied the following main goals of our work: First, we wanted to develop a force-directed algorithm that is designed for drawing general graphs and has a guaranteed sub-quadratic worst-case running time. Second, the algorithm should be fast in practice, too. Finally, the algorithm should generate pleasing drawings of a wide range of large graphs.

Whether we were able to realize these goals will be discussed in the following chapters. In particular, in Chapter 1 we will introduce some basic terminology that will be needed in this dissertation. Furthermore, we will give an overview about general force-directed graph drawing and force-directed methods that are used for drawing large graphs. Chapter 2 will sketch the basic algorithmic concept of a new force-directed graph-drawing method. In the following Chapters 3, 4, 5, and Section 6.1, we will discuss the different algorithmic steps in detail, before we will formulate our main theoretical result in Chapter 6.2. Finally, in Chapter 7 a detailed experimental study of the new force-directed method and an experimental comparison with state-of-the-art algorithms for drawing large graphs will evaluate the practical use of the new approach. An extended abstract of some of the main parts of this dissertation is [61].

Chapter 1

Preliminaries and Previous Work

Das Nicht-Haben ist der Anfang allen Denkens. [1]

In the first section of this chapter we will introduce some basic terminology in graph theory and graph drawing. The used notations are motivated by [77, 100, 31, 25, 30]. In Section 1.2 we will explain the basic concepts of force-directed graph drawing and sketch the most important classical force-directed algorithms. A detailed introduction to force-directed methods can also be found in [14, 30, 123]. In Section 1.3 we will review state-of-the-art force-directed methods that are designed for drawing large graphs and discuss the asymptotic running times of these methods. Some other methods for drawing large graphs that are related with force-directed methods will be presented in this section, too.

1.1 Graphs and Their Drawings

1.1.1 Undirected Graphs

An *(undirected) graph* G is a pair (V, E), where V is a finite set of *nodes* or *vertices*, and E is a finite set of *edges*, where each edge $e = (u, v) \in E$ consists of an unordered pair of vertices $u, v \in V$. V is called the *node set* or *vertex set* of G, and E is called the *edge set* of G. An edge $e = (v, v)$ is called a *loop*. If a graph contains a set of edges $e_1 = e_2 = \ldots = e_k = (u, v)$, for a $k \geq 2$, then, $e_1, \ldots, e_k$ are called *multiple* edges. A graph that contains no loops and no multiple edges is called *simple*. In the remainder of this sub-section, we will assume (unless otherwise stated) that the given graphs are simple.

Let $e \in E$ be an edge and $v \in V$ be one of its nodes, then e and v are called *incident*. If $e \in E$ has the incident nodes u and v, then u and v are called *adjacent*, and u is a *neighbour* of v. The set of all adjacent nodes of a node v is denoted by $adj(v)$. The *degree* of a node $v \in V$ is the number of its adjacent nodes and denoted by $deg(v)$. A node is called *isolated* if it has the degree zero.

[1] Robert Musil

A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$, and $E' \subseteq E \cap V' \times V'$. For a node set $V' \subseteq V$ we say that G' is a *vertex-induced subgraph* of $G = (V, E)$ if $E' = E \cap V' \times V'$ and also denote G' by $G[V']$.

A *walk* of *length* k in a graph is an alternating sequence of vertices and edges $v_1, e_1, v_2, e_2, \ldots, v_k, e_k, v_{k+1}$, *beginning* and *ending* with the nodes v_1 and v_{k+1}, respectively, and $e_i = (v_i, v_{i+1})$ for all $i = 1, \ldots, k$. This walk *connecting* v_1 and v_{k+1} can also be denoted by $W = (v_1, v_2, \ldots, v_{k+1})$. A walk is a *path* if all its nodes are distinct. A walk is called a *cycle* if all vertices are distinct, except for $v_1 = v_{k+1}$, and $k \geq 3$. A graph that does not contain a cycle is called a *forest*. The *(graph-theoretic) distance* of two nodes $u, v \in V$ is the length of the shortest path connecting u and v and denoted by $dist(u, v)$. If no such path exists between u and v, the distance between these two nodes is set to infinity. The largest distance between any two nodes of G is called the *diameter* of G and is denoted by $diam(G)$.

A graph G is *connected* if every pair of nodes is connected by a path. Otherwise the graph is called *disconnected*. A *component* is a maximal connected subgraph of G. A *cut-vertex* is a node, whose removal increments the number of components. A connected forest is called a *tree*. A graph $G = (V, E)$ is *k-connected* if at least k nodes must be removed from V to make the resulting induced subgraph disconnected. If $k = 2$ and $k = 3$, the graph is called *biconnected* and *triconnected*, respectively. The maximal biconnected components of a graph are called *blocks* and intersect in the cut-vertices.

A subgraph T' of a tree $T = (V, E)$ is called a *subtree* if T' is a tree. A tree $T = (V, E)$ is a *rooted tree* if one and only one node $r \in V$ is distinguished from the other nodes. This node r is called the *root* of the rooted tree T.

Let $T = (V, E)$ be a rooted tree with root r and $v \in V$. Any node u on the unique path connecting r and v is called an *ancestor* of v. If u is an ancestor of v, then v is a *descendant* of u. If furthermore $u \neq v$, then u is a *proper ancestor* of v, and v is a *proper descendant* of u. If u is an ancestor of v, and $e = (u, v)$ is an edge in E, then u is called the *parent* of v, and v is called the *child* of u. A node of a rooted tree that has no children is a *leaf*, otherwise it is an *interior node*. The *subtree rooted at* v is the subtree of T that is induced by v and the descendants of v, rooted at v. The length of the path from the root node r to a node $v \in V$ is the *depth of* v. The maximum depth of all nodes $v \in V$ is the *depth of the rooted tree* T. The *child degree* of a node v is the number of children of the node v. A *k-nary* tree is a rooted tree with maximum child degree k. *Binary trees* and *quaternary trees* are rooted trees with maximum child degree two and four, respectively. A k-nary tree $T = (V, E)$ is a *complete k-nary tree* if every node $v \in V$ except the leaves have child degree k and the depth of all leaves is identical. An *ordered tree* is a rooted tree in which the children of each node are ordered.

A graph $G = (V, E)$ is *complete* if every node $v \in V$ is adjacent to every node $w \in V \setminus \{v\}$. A complete graph with n nodes is denoted by K_n.

Suppose that $e = (u, v)$ is an edge of $G = (V, E)$, then G/e is the graph that is constructed by *contracting*, *shrinking*, or *collapsing* the edge e of G to a node v_e. More formally, we define $G/e := (V', E')$ with $V' := (V \setminus \{u, v\} \cup \{v_e\})$ and $E' := \{(w, x) \in E \mid \{w, x\} \cap \{u, v\} = \emptyset\} \cup \{(v_e, w) \mid (u, w) \in E \setminus \{e\} \quad \text{oder} \quad (w, v) \in E \setminus \{e\}\}$. Suppose that

$U \subseteq V$, then G/U is the graph that is obtained from G by contracting all edges of the vertex-induced subgraph $G[U]$. It has to be pointed out that G/e and G/U might contain multiple edges, even if G is simple (see Figure 1.1).

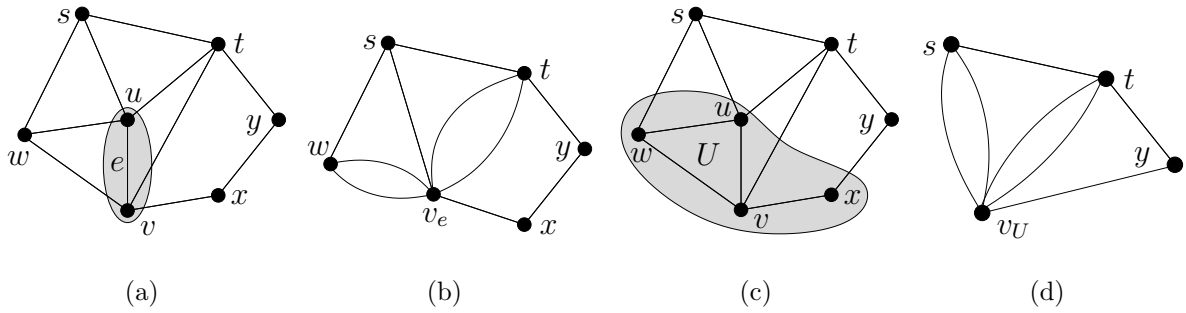


Figure 1.1: (a) A simple graph G with a selected edge e. (b) The graph G/e that contains multiple edges. (c) A selected node subset $U = \{u, v, w, x\}$ of G. (d) The graph G/U that contains multiple edges.

1.1.2 Weighted Graphs

An *(undirected) weighted graph* G is a triple (V, E, c), where (V, E) is a graph, and $c : E \to \mathbb{R}$ is a *weight function* or *cost function*. For an edge $e \in E$ the value $c(e)$ is called the *weight* or *cost* of e. The *length of a path* $P = (v_1, \ldots, v_k)$ in a weighted graph G is defined as $c(P) = \sum_{i=1}^{k-1} c((v_i, v_{i+1}))$. Hence, for unweighted graphs the length of a path has been defined as $c \equiv 1$. The *distance* between two nodes $u, v \in V$ in a weighted graph G is the length of the shortest path connecting u and v. If no such path exists between u and v, the distance between these two nodes is set to infinity. We define the *graph-theoretic distance* between two nodes in a weighted graph as the (graph-theoretic) distance between these nodes in the underlying unweighted graph. If $c : E \to \mathbb{R}^+$, we call G a *positive-weighted graph*.

1.1.3 Directed Graphs

A *directed graph* or *digraph* D is a pair (V, E), where V is a finite set of *nodes* or *vertices*, and E is a finite set of *(directed) edges* or *arcs*, where each edge $e = (u \to v) \in E$ consists of an ordered pair of vertices $u, v \in V$. The edge $e = (u \to v) \in E$ is said to be *directed from u to v*, and u is called the *source node* or *initial vertex* of e, while v is called the *target node* or *terminal vertex* of e. Ignoring for every edge the order of the vertices, we get an undirected graph that is called the *underlying undirected graph* of D. An edge $e = (v \to v)$ is called a *loop*.

If a digraph contains a set of edges $e_1 = e_2 = \ldots = e_k = (u \to v)$, for a $k \geq 2$, then $e_1, \ldots, e_k$ are called *parallel* edges. A set of edges is called multiple edges if these edges

are multiple edges in the underlying undirected graph. A digraph is *simple* if it contains no loops and no multiple edges.

A directed graph $D' = (V', E')$ is a *subgraph* of a directed graph $D = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E \cap V' \times V'$. For a node set $V' \subseteq V$ we say that D' is a *vertex-induced subgraph* of $D = (V, E)$ if $E' = E \cap V' \times V'$ and also denote D' by $D[V']$.

A *directed walk* W of *length* k in a directed graph is a walk $W = (v_1, v_2, \ldots, v_{k+1})$ in the underlying undirected graph of D with $e_i = (v_i \to v_{i+1}) \in E$, for all $i = 1, 2, \ldots, k$. A directed walk is a *directed path* if all its nodes are distinct. A directed walk is called a *directed cycle* if all vertices are distinct, except for $v_1 = v_{k+1}$, and $k \geq 3$. A directed graph that does not contain directed cycles or loops is called *acyclic digraph* or *dag*. The *(graph-theoretic) distance* between two nodes $u, v \in V$ in a digraph is the length of the shortest directed path connecting u and v. If no such path exists between u and v, the distance between these two nodes is set to infinity. A digraph is called *strongly connected* if each pair of vertices $u, v \in V$ is connected by a directed path from u to v. A digraph is *(k)-connected* if its underlying undirected graph is (k)-connected.

A *(positive-) weighted directed graph* $D = (V, E, c)$ is defined analogue to the (positive-) weighted undirected graph.

1.1.4 Drawings of Graphs

A graph $G = (V, E)$ is generally visualized by a two-dimensional or three-dimensional *drawing* $\Gamma(G)$. In the context of this dissertation we restrict to two-dimensional drawings.

In a two-dimensional drawing $\Gamma(G)$ of a graph $G = (V, E)$ the nodes are visualized as points, circles, ellipses, or polygons. We call the tightest axis-parallel rectangle that covers the graphics of a node v the *bounding box of the node* v. The *size (of the graphics)* of a node is given by the width and height of its bounding box. The edges are drawn as closed Jordan curves that connect their incident vertices. Popular styles for drawing the edges in automatic graph drawing are:

- *straight-line drawings:* Edges are drawn as straight lines.
- *orthogonal drawings:* Edges are drawn as polygonal chains of alternating horizontal and vertical segments.
- *quasi-orthogonal drawings:* Edges are drawn as polygonal chains of alternating horizontal and vertical segments, but the first and last *edge bend* is allowed to be non-orthogonal.
- *polyline drawings:* Edges are drawn as polygonal chains.

Additionally, the edges of directed graphs may be drawn as arrows pointing from the source nodes to the target nodes. The edge weights of weighted graphs or weighted digraphs may be displayed as labels that are placed near the drawings of the edges. But adding labels to the drawings of nodes and edges of unweighted graphs and digraphs is allowed, too. Figure 1.2 exemplifies this terminology.

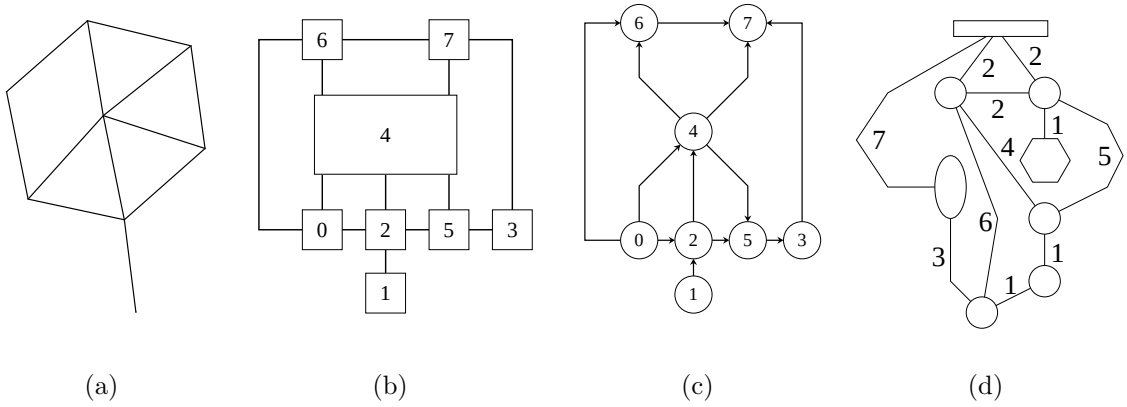


Figure 1.2: (a) A straight-line drawing with nodes drawn as points. (b) An orthogonal drawing with nodes drawn as labeled rectangles of different sizes. (c) A quasi-orthogonal drawing of a digraph with nodes drawn as labeled circles. (d) A polyline drawing of a weighted graph with nodes in different sizes and shapes containing labeled edges.

A graph is called *planar* if it can be drawn in the plane so that no two edges cross each other except at a common endpoint. An intersection of two edges in a drawing other than at their endpoints is called an *edge crossing*. The *crossing number* $\nu(G)$ of a graph G is the least integer k so that G can be drawn in the plane with at most k edge crossings. We call the ratio of the shortest and the longest edge in a drawing the *edge-length ratio* that can be used as a measure of the uniformity of the edge length. We call a drawing of a graph *symmetric* if there exists a rotation at an arbitrary center point or a reflection at an axis that fixes the drawing, i.e. that maps nodes to nodes and edges to edges. More formal definitions of symmetries in graphs are given in [36, 40, 19].

Given a drawing $\Gamma(G)$ of a graph G, the *bounding rectangle* of $\Gamma(G)$ is the tightest axis-parallel rectangle that covers $\Gamma(G)$. The *drawing area* of $\Gamma(G)$ is the product of the width and height of the bounding rectangle of $\Gamma(G)$. The *aspect ratio* of a drawing is given by the ratio of the width and height of the bounding rectangle of the drawing. Given a drawing $\Gamma(G)$ and a constant $r > 0$ called *desired aspect ratio*, we call the area of a smallest axes-parallel rectangle of breadth b and height h with $r = \frac{b}{h}$ that covers the drawing $\Gamma(G)$ the *aspect-ratio area of* $\Gamma(G)$ *according to the desired aspect ratio* r. Figure 1.3 illustrated these notations. The aspect-ratio area (ARA) according to the desired aspect ratio r of a drawing $\Gamma(G)$ can be obtained by calculating the width w, height h, and $r' = \frac{w}{h}$ of its drawing area and by using the following formula:

$$ARA = \begin{cases} w \cdot h \cdot \frac{r}{r'} & \text{if } r' < r \\ w \cdot h \cdot \frac{r'}{r} & \text{otherwise} \end{cases} \tag{1.1}$$

As already mentioned in the introduction of this dissertation, important *(aesthetic) criteria* that help evaluating the quality of a drawing are the number of edge crossings, the

number of edge bends, the used drawing area, the used aspect-ratio area, the sum of the edge lengths, the uniformity of the edge lengths, the number of nodes that overlap each other, and the display of symmetries if some exist.

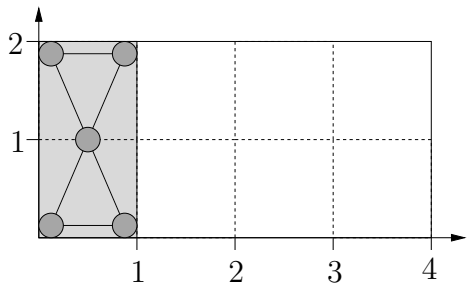


Figure 1.3: A drawing of a graph that contains 5 nodes. Its bounding rectangle (grey rectangle) needs drawing area $1 \cdot 2 = 2$ and has aspect ratio $\frac{1}{2}$. The aspect ratio area of this drawing according to the desired aspect ratio $r = 2$ is $4 \cdot 2 = 8$.

1.2 Force-Directed Graph Drawing

1.2.1 The Basic Concepts

Force-directed graph-drawing methods consist of two components: A *force model* that consists of physical objects (e.g., repelling particles, springs) which are related with the given graph $G = (V, E)$ and an algorithm that tries to compute a placement of the nodes in the plane so that the total energy of the related physical system has a (local) minimum value. The underlying assumption is that the energy-minimum states of suitable defined physical systems will correspond to pleasing drawings.

Depending on the desired aesthetic criteria, the force models are chosen in several different ways, and often the models are only weakly related with the physical reality. Therefore, one could also use the terminology *drawing based on physical analogies* (as used by Brandes [14]) instead of force-directed graph drawing.

Force-directed methods are frequently used in graph drawing for several reasons. First of all, they are designed to draw general graphs and, therefore, do not require the graphs to have special structural attributes. The relation with the physical reality makes them easy to understand, and — since many of the force-directed algorithms are comparatively simple — these methods are also easy to implement. Furthermore, the generated drawings are often satisfactory, regarding the modeled criteria. Finally, most of the methods can easily be extended to methods that can handle constraints (e.g., fixed nodes or fixed subgraph constraints) or requirements of special graph classes (e.g., directed graphs, graphs with a recursive clustering structure over the node set called *clustered graphs*).

A modeled criterion of all force-directed methods are straight-line drawings that reflect the desired edge length as well as possible. As a consequence — when the desired edge lengths of all edges are equal — adjacent nodes should be drawn close to each other. It is

important to note in this context that it is impossible to generate a straight-line drawing that preserves the desired edge length for each given graph G exactly (e.g., suppose G is a complete graph with 3 nodes and the desired edge lengths are $1, 1$, and 100).

Furthermore, the nodes should "spread well" in the drawing area (see [14] page 71). In particular, the drawings of nodes should not overlap. As a consequence of combining these two criteria, algorithms that are based on these models often display symmetries in graphs if some exist. In particular, Eades and Lin [40] prove that — given a symmetric graph G — there exist energy-minimal configurations of the nodes of G in the force models of several force-directed algorithms [130, 35, 80] so that the induced drawings are symmetric. Additionally, other criteria like the minimization of edge crossings [27] can be explicitly taken into account in the models.

Since several of the optimization problems that arise in (force-directed) graph drawing are $\mathcal{NP}$-hard, heuristics are used as drawing algorithms. For example, detecting symmetries of an arbitrary graph is $\mathcal{NP}$-hard [93, 19]. Fixed edge length straight-line graph drawing in any number of dimensions and crossing-free fixed edge length graph drawing of planar graphs are $\mathcal{NP}$-hard [74, 41]. It is also an $\mathcal{NP}$-complete decision problem, whether the crossing number $\nu(G)$ of a given graph G is less than or equal k, for a fixed given integer $k \geq 1$ [53]. The used heuristics either try to generate a distribution of the nodes that implies an (approximate) equilibrium configuration of the forces acting on each node or they try to minimize the energy directly without calculating the forces.

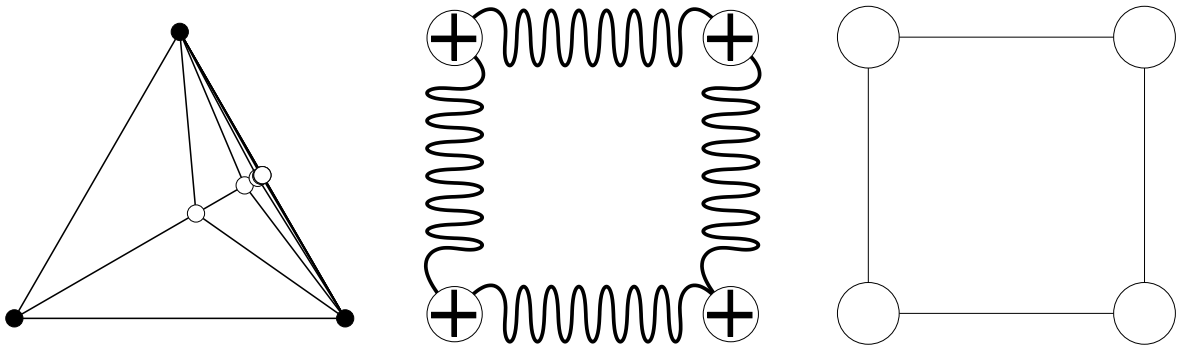
1.2.2 Classical Force-Directed Methods

The Barycenter Method

The paper *How to draw a graph* by Tutte [130] can be seen as the starting point of both graph drawing and force-directed graph drawing. In his method the node set is partitioned into a set of a constant number of at least three *fixed* nodes that are assigned fixed positions on a convex polygon and a set of *free* vertices. The edges are associated with springs and the nodes with steal rings. The problem of calculating an equilibrium configuration of the forces in the induced system is reduced to the problem of finding solutions of a system of linear equations. The number of equations and the number of unknowns are both equal to the number of free vertices. For planar graphs the matrix representing the equations is sparse and, thus, the equations can be solved in time that is sub-quadratic in the number of nodes [91, 14, 30]. Since the solution of this system corresponds to a drawing that places every free node at the barycenter of its adjacent nodes, it is called `Barycenter Method`. Another important property of the `Barycenter Method` is that if it is applied on a 3-connected planar graph and the nodes of a face of some embedding are fixed in a strictly convex planar drawing of this face, it generates a planar straight-line drawing.

In spite of these desirable properties the `Barycenter Method` is not frequently used today. One reason is that the arbitrary choice of the fixed nodes has a very big influence on the quality of the drawing, and the appropriate choice of these nodes is difficult for general graphs. Furthermore, the algorithm of Tutte uses drawing area that can be exponential

in the number of nodes [39], which is undesirable. Additionally, for planar graphs several linear time algorithms were developed that generate crossing-free straight-line drawings with linear drawing-area requirements [28, 121, 60]. Figure 1.4(a) shows a drawing of a planar graph containing 10 nodes and 24 edges that is generated with the `Barycenter Method`. Many free nodes overlap due to the exponential usage of drawing area. This drawing also contains very long and extremely short edges.



(a) (b) (c)

Figure 1.4: (a) A drawing of a planar graph that is generated by the `Barycenter Method`. The black circles are fixed nodes and the white circled free nodes. (b) An equilibrium configuration of the forces in the `Spring Embedder` model and (c) the corresponding drawing of the associated graph.

The Spring Embedder and Related Methods

Several publications in the research field of VLSI design describe force-directed methods that are developed for the problem of placing cells on a circuit board [111, 118, 43]. Nevertheless, the article *A heuristic for graph drawing* by Eades [35] is often referred as the seminal publication in the field of force-directed graph drawing. Since several methods are similar to this method (that is called `Spring Embedder`) we describe it in greater detail here. The used model associates the nodes with equally charged particles that repel each other and the edges with springs. The model abstracts from physical reality by defining logarithmic springs and allowing only nonadjacent nodes to repel each other. In particular, suppose that nodes u and v are placed at positions p_u and $p_v \in \mathbb{R}^2$ and are not adjacent. Then, the repulsive forces induced by u and acting on v are defined as

$$F^u_{rep}(v) = c_r \frac{p_v - p_u}{\|p_v - p_u\|^2} \text{ , with a suitable constant } c_r > 0.$$

If the nodes u and v are connected by an edge $e = (u, v)$, then the force that acts on v due to the spring e is defined as

$$F^e_{spring}(v) = c_s \log\left(\frac{\|p_v - p_u\|}{l}\right)(p_u - p_v)\ , \text{ with suitable constants } c_s > 0 \text{ and } l > 0.$$

This definition of the spring forces implies that a spring is in a zero-energy state if the distance between its adjacent nodes is l. Therefore, one can interpret l as the desired edge length of every edge.

Starting with an initial random distribution of the particles, the algorithm iteratively tries to find a distribution of the particles in the plane that induces an equilibrium configuration of the forces that act on each particle. This is realized by iteratively moving each particle in the direction of the resulting force that acts on it. Figure 1.4(b) shows a `Spring Embedder` model that is in an equilibrium configuration of the forces, and Figure 1.4(c) shows the corresponding drawing of the associated graph. The pseudo-code of this method is given in Algorithm 1, and its running time is $O((|V|^2 - |E|) + |E|) = O(|V|^2)$.

Algorithm 1: `Spring Embedder`

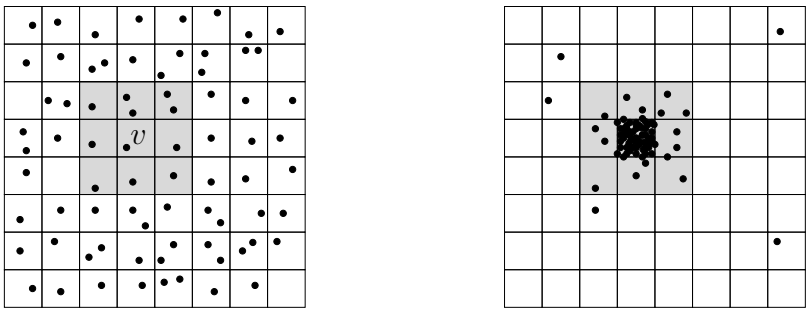
input : a graph $G = (V, E)$, an integer constant *max_iter*, and a small constant δ
output: a placement $p(V) = (p_v)_{v \in V}$

begin
  foreach $v \in V$ **do** $p_v \leftarrow$ random coordinate;
  for $i = 1$ **to** *max_iter* **do**
    foreach $v \in V$ **do**
      $$F(v) \leftarrow \sum_{u:(u,v)\notin E} F^u_{rep}(v) + \sum_{(u,v)\in E} F^{(u,v)}_{spring}(v);$$
    foreach $v \in V$ **do** $p_v \leftarrow p_v + \delta \cdot F(v)$;
end

Fruchterman and Reingold [47] define a force-directed method that is similar to the `Spring Embedder` of Eades [35]. In their force model the repulsive forces are defined for any pair of distinct nodes u and v and the absolute value of these forces is inverse to the Euclidean distance between u and v. The spring forces acting between two adjacent nodes grow quadratically with their Euclidean distance. Furthermore, the drawing area is restricted to a rectangle of predefined size. The drawing algorithm is constructed similar to the `Spring Embedder`, too, and its running time is $O(|V|^2 + |E|)$.

The most costly part of the `Spring Embedder` variant of Fruchterman and Reingold [47] is the calculation of the repulsive forces acting between all pairs of distinct nodes. This takes $\Theta(|V|^2)$ time. Therefore, Fruchterman and Reingold [47] introduce a faster variant of their method that has a better best-case running time and call it `Grid-Variant Algorithm`. The `Grid-Variant Algorithm` does only calculate the repulsive forces acting between nodes that are placed relatively near to each other. This is done by restricting the nodes

of the graph to be placed in a rectangular drawing area that is subdivided into a regular $\lfloor\sqrt{|V|}/g\rfloor \times \lfloor\sqrt{|V|}/g\rfloor$ grid. The parameter g that indicates the coarsening of the grid is set to the fixed value two. The repulsive forces that act on a node v that is contained in a grid box B are approximated by summing up only the repulsive forces that are induced by the nodes contained in B and the nodes in the grid boxes that are bordering on B. Such ideas for approximating the repulsive forces by laying out a regular grid on the rectangular simulation area are also known as *particle-in-cell codes* or *PIC codes* in the physical literature [105]. The best-case running time of the `Grid-Variant Algorithm` is $O(|V| + |E|)$ if in every iteration of the force calculation each grid box contains only a constant number of nodes (see Figure 1.5(a)). If, for example, in one iteration of the force calculation nearly all nodes are contained in one grid box (see 1.5(b)), the running time remains $O(|V|^2 + |E|)$.



(a) (b)

Figure 1.5: Distribution of the nodes that induce a linear (a) and a quadratic (b) running time in the actual iteration of the grid-variant algorithm. The edges are not drawn here. The nodes that are used for approximating the repulsive forces that act on the node v are the nodes contained in the grey shaded grid boxes.

It is interesting to note that Fruchterman and Reingold [47] mention the existence of a more accurate algorithm for approximating the repulsive forces acting between all pairs of distinct particles that is based on approximating potential fields by evaluation multipole expansions and has been developed by Greengard and Rokhlin [59]. This work will be of fundamental importance for our graph-drawing algorithm that will be developed later.

Another refinement of the `Spring Embedder` method is the algorithm `GEM` of Frick et al. [45]. One main difference to the `Spring Embedder` and to the methods of Fruchterman and Reingold [47] is that integer arithmetic is used to speed up the computations. Additionally, the integration of a gravitational force that drags the vertices to the barycenter makes it possible to generate drawings of disconnected graphs. Several other features like an approach that should prevent the drawing to oscillate and rotate are added to increase the drawing quality and to reduce the running time. The maximum number of iterations is set to $4|V|$. The algorithm stops either if the maximum number of iterations is reached

or if the forces acting on every node are approximately in an equilibrium configuration. Hence, the total running time of `GEM` is $O(|V|(|V|^2 + |E|))$.

Modeling Graph-Theoretic Distances in a Complete Spring Model

Kamada and Kawai [80] introduce a model that does not identify the nodes with charged particles. Instead, between each pair of distinct vertices u and v a spring with a zero-energy length that is proportional to the graph-theoretic distance between u and v is defined. The potential energy of this system of $n(n-1)$ springs for a given placement $p(V)$ of the nodes is defined as

$$E_{KK}(p(V)) = \sum_{v \in V} \sum_{u \in V, u \neq v} \left(\frac{\|p_v - p_u\| - l \cdot dist(u,v)}{dist(u,v)} \right)^2.$$

The idea is that the ideal distance between two nodes u, v is the product of the graph-theoretic distance $dist(u,v)$ between these nodes and the desired ideal edge length l of all edges. The algorithm finds a local energy-minimum configuration of the nodes by first setting all partial derivatives of E_{KK} to zero, which is a necessary condition for a local energy-minimum configuration. This results in a system of dependent nonlinear equations. Then, an iterative method is used that chooses a node v with the largest gradient and moves it in the direction of the gradient while all other positions of the nodes are temporarily fixed. The algorithm terminates if the size of the gradient falls below some threshold. Kamada and Kawai do not give an upper bound on the running time of their method. However, the worst-case running time that is needed for computing the shortest distances between all pairs of nodes is $\Omega(|V|^2)$. The memory requirements for storing these distances are quadratic, too.

Methods that Use General Energy Functions

Davidson and Harel [27] develop a force model with a general energy function that is the weighted sum of several modeled criteria. As the model of Eades [35] and Fruchterman and Reingold [47], they identify nodes with charged particles and edges with springs. They also define a potential that penalizes nodes that are placed close to the boundaries of the drawing area and a potential that penalizes short distances between nodes and edges. Both potentials are added to the energy function. Finally, they add a term that counts the number of edge crossings to the energy function. In order to obtain a local minimum of the defined energy function, they use an optimization technique called *simulated annealing*. Starting with an initial random distribution $p_{act}(V)$ of the nodes, first the induced energy E_{act} of the system is calculated. Then, their algorithm iteratively selects only a single node $v \in V$ and places it at a new random position. We denote the resulting new placement by $p_{new}(V)$. If the energy E_{new} induced by the new placement $p_{new}(V)$ is less than E_{act}, the new placement is taken as the actual placement p_{act}. If the energy induced by the new placement E_{new} is greater than E_{act}, the new placement is taken only with a certain probability as the actual placement $p_{act}(V)$. Then, the next node of the node set is assigned

a new random position. When all nodes of the node set have been touched, a *temperature* value t is reduced. The algorithm terminates when t is less than a predefined threshold. Assuming that convergence is reached after a constant number of temperature changes, the running time of this method is $O(|V|^2|E|)$ [27]. A refined version of a simulated annealing approach has been invented by Tunkelang [126].

1.2.3 The Freedom of Modeling

The methods that we have introduced in the previous section indicate that there is a big freedom in the definition of the force model and the design of the algorithm for generating a placement of the vertices that corresponds to a energy-minimal state in the associated force model. Therefore, several authors developed force-directed algorithms for special graph classes or tuned their methods to take constraints into account. In the following, we give some examples of such methods.

Harel and Sardas [67] and Bertault [11] introduce techniques for generating force-directed drawings of planar graphs that avoid edge crossings. Both methods first generate a planar straight-line drawing of the given graph that can be fast obtained in linear time, for example by using the methods described in [28, 121, 60]. Starting with this drawing as an initial placement, Harel and Sardas [67] use simulated annealing and Bertault [11] uses a modified `Spring Embedder` method to obtain a planar drawing with more uniform edge length. The running times of the methods of Harel and Sardas [67] and Bertault [11] are $O(|V|^3)$ and $O(|V|^2)$, respectively. Both methods can be applied to general graphs by computing a planar subgraph and using edge re-insertion techniques [67].

For directed graphs it is sometimes desirable that the directed edges point into roughly the same direction in order to recognize the dependencies easily. Therefore, the edges in the model of Sugiyama and Misue [124] are identified with magnetic springs and an external magnetic field is introduced to the model that forces the springs to point in the desired direction. Variants of force-directed methods that generate drawings of clustered graphs are presented in [135, 72].

Wang and Miyamoto [135], Harel and Koren [65], Tunkelang [128], and Chuang et al. [23] present methods that are designed to create drawings of graphs with nodes of non-uniform sizes. The most important aspects for the layout of such graphs are that the nodes should not overlap and that the desired edge lengths are preserved. We will concentrate on the force-directed layout of graphs with nodes and edges of different sizes in Section 3.1 in greater detail.

Many presented force-directed methods can be modified to take into account other simple constraints like fixed-node constraints or fixed-subgraph constraints, see for example [135, 81]. A general survey on constraints in graph drawing is presented in [125].

Finally, several of the ideas that we have described can be easily extended to three-dimensional models and methods. For example, Bruß and Frick [18] present a three dimensional version of the `GEM` algorithm of Frick et al. [45], while Cruz et al. [26] and Monien et al. [96] present a three-dimensional and a parallel three-dimensional version of the simulated annealing algorithm of Davidson and Harel [27], respectively.

1.3 Algorithms for Drawing Large Graphs

An experimental comparison of the basic algorithm of Fruchterman and Reingold [47], the `GEM` algorithm of Frick et al. [45], the method of Kamada and Kawai [80], the method of Davidson and Harel [27], and the simulated annealing approach of Tunkelang [126] was carried out by Brandenburg et al. [13]. The experiments show that all algorithms generate comparable good drawings of small graphs with $(|V| + |E|) < 180$ in less than two minutes and that `GEM` and the method of Kamada and Kawai [80] are the fastest upon these classical methods. Since the best-case and worst-case running time of these methods is at least quadratic, they are not suited for drawing graphs containing several thousands of nodes. For example, the `GEM` algorithm needs 71 seconds for drawing a graph containing 256 nodes that represents a regular square grid [45]. An estimated running time for drawing a graph containing 25600 nodes on the same machine and relying on the cubic running time of `GEM` would be more than two years. Since Brandenburg et al. [13] did not test the `Grid-Variant Algorithm` of Fruchterman and Reingold [47], one might hope that this method scales better in practice, since it has a quadratic worst-case but a linear best-case running time. Our experiments that we will present in Section 7.7 will demonstrate that, in practice, the running times of the `Grid-Variant Algorithm` are unsatisfactory for large graphs, too.

Consequently, researchers started to develop new concepts that enable force-directed methods to draw large graphs that contain several thousands of nodes in reasonable time. In the following, we will present state-of-the-art force-directed algorithms that are used to draw large graphs.

1.3.1 Methods Based on Approximating the Repulsive Forces

The most time consuming step in `Spring Embedder`-like algorithms (see Algorithm 1) is the calculation of the repulsive forces acting between all pairs of distinct particles/nodes. The naive exact calculation of these forces needs $\Theta(N^2)$ time.

The problem of efficiently approximating these forces has been widely studied in physics, since these approximations are needed in the interior loops of *N-body simulations* [1], where N denotes the number of particles (here, $N := |V|$). The PIC code used in the `Grid-Variant Algorithm` of Fruchterman and Reingold [47] is a simple — but inaccurate — way of approximating the repulsive forces acting between all pairs of distinct particles. More appropriate approximative methods have been invented in the physical literature that are faster and more accurate than PIC codes, in practice [105].

One very popular method is the algorithm of Barnes and Hut [8]. We will only sketch some ideas of this method here, while more details of the algorithm of Barnes and Hut [8] and other force-approximation methods will be described in Section 5.2: The idea is that the forces that act on a particle v due to particles placed in the near surrounding of v are calculated directly, while the force contribution of a group of particles $S := \{w_1, \ldots, w_k\} \subseteq V$ that are placed far away from v is only approximated. In particular, the force that acts on v due to the particles in S is replaced by a *group force* that is induced by one *group*

particle w_S of charge k that is assumed to be placed at the center of mass of the particles in S. The suitable choice of the group particles w_S that act on a particle v is done by using a spatial data structure that is called *quadtree* and will be explicitly introduced in Section 5.2.1. Unlike otherwise stated, the asymptotic running time of this method is not $O(N \log N)$ in general. Even worse, Aluru et al. [2] have proven that the running time of this method cannot be bounded by a function that depends on the number of particles only. However, it can be shown that the running time of the algorithm of Barnes and Hut is $O(N \log N)$ in some special cases (see Section 5.2.3).

The graph-drawing algorithms `JIGGLE` of Tunkelang [127, 128] and `FADE` of Quigley and Eades [110] use the method of Barnes and Hut [8] for approximating the repulsive forces in their `Spring Embedder`-like force models. Tunkelang [127, 128] assumes that in each iteration the depth of the quadtree is bounded by $O(\log N)$ — in order to obtain a $O(N \log N)$ scaling — but does not explain how this restriction is realized in his algorithm. Quigley and Eades [110] do not make any assumptions on the distribution of the particles. Hence, the worst-case running times of both methods are given by the worst-case running time of the method of Barnes and Hut [8].

Practical experiments in [110] show that `FADE` generates well-structured drawings of graphs containing up to 4700 nodes, but the authors do not announce the total running times that are needed to draw these graphs. Instead, they show that their implementation of the Barnes-Hut method [8] for approximating the repulsive forces is significantly faster than the naive quadratic algorithm on a not defined distribution of particles. The speed up factor is 92.6 for a graph containing 6000 particles. The largest graph drawn in practical experiments with `JIGGLE` [128] is a cycle containing 1024 nodes and the reported running time is roughly five minutes. The quality of the drawings is comparable with that generated in the tests of `FADE`.

A variant of the method `FADE` that constructs a *recursive Voronoi diagram* instead of the quadtree data structure and has an $O(|V|^2)$ worst-case running time is presented in [106].

1.3.2 Multilevel Methods

Besides the quadratic running time of calculating the repulsive forces naively, one major problem of classical force-directed methods is that for large graphs lots of iterations of the outer loop of the iterative force-directed algorithms are needed to generate a local energy-minimum configuration, when starting from an initial (mostly random) placement of the nodes. This statement is confirmed by observations of Harel and Koren (see [66] page 216), by Frick et al. [45] who defined the number of iterations as a function linear in $|V|$, and by our own experimental studies that we will present Section 7.3. Therefore, one might try to find an initial distribution of the nodes that is already close to a drawing of the graph, which induces an energy-minimal state in the associated force model. Starting from such an initial placement, the number of iterations that is needed by a force-directed algorithm to generate an energy-minimal configuration of the nodes will be comparatively small. One way to realize this goal is to integrate *multilevel* or *multi-scale* ideas into classical

force-directed algorithms that we denote by *single-level* methods to separate them from the multilevel algorithms. In principle, force-directed multilevel algorithms consist of two basic components:

- A *coarsening phase* that, starting with the initial graph $G =: G_0$, generates a sequence of graphs $G_0, G_1, \ldots, G_k$ of decreasing sizes by constructing G_{i+1} from G_i for $i = 0, \ldots, k-1$ so that the graphs are closely related. The graph G_{i+1} at *(multi)level* or *refinement level* $i+1$ can be seen as a *coarse* representation of the *fine* graph G_i at multilevel i.

- In the *refinement phase*, a force-directed single-level algorithm is used to generate a drawing of the coarsest graph G_k. Then — starting with the layout of the coarsest graph G_k — the layout of the coarse graph G_i is used to obtain an initial layout of the finer graph G_{i-1}, for $i = 1, \ldots, k$. A force-directed layout of G_{i-1} is generated by using the single-level algorithm that already has been used to draw G_k. In the lowest multilevel the graph $G = G_0$ is drawn.

The Fast Multi-scale Method

The first authors that introduced multilevel ideas into the field of graph drawing were Hadany and Harel [62] motivated by previous work in particle physics [16, 17]. An improved and simplified version of this method (called `Fast Multi-scale Method`) was developed by Harel and Koren [64]. It is suited for drawing connected unweighted graphs and its basic components are described in the following.

In order to create the sequence of coarse graphs, an $O(k|V|)$ algorithm that finds a 2-approximative solution of the $\mathcal{NP}$-hard *k-center problem* [54] is used. The optimization goal of this problem is to determine a subset $S \subseteq V$ of size k in a graph $G = (V, E)$ so that $max_{v \in V}\, min_{s \in S}\, dist(s, v)$ is minimized. The node set V_i of a graph G_i in the sequence $G_0, \ldots, G_k$ is determined by the approximative solution of the k_i-center problem on G with $k_i > k_{i+1}$ for all $i \in \{1, \ldots, k-1\}$.

Harel and Koren [64] use a variation of the algorithm of Kamada and Kawai [80] (see Section 1.2.2) as force-directed single-level algorithm. In order to speed up the computation of this method, they modify the energy function of Kamada and Kawai [80] that is associated with a graph G_i with $i \in \{0, \ldots, k-1\}$ to

$$E_{HK}(p(V_i)) = \sum_{v \in V_i} \sum_{u \in N^{c_i}(v)} \left(\frac{\|p_v - p_u\| - l \cdot dist(u, v)}{dist(u, v)} \right)^2.$$

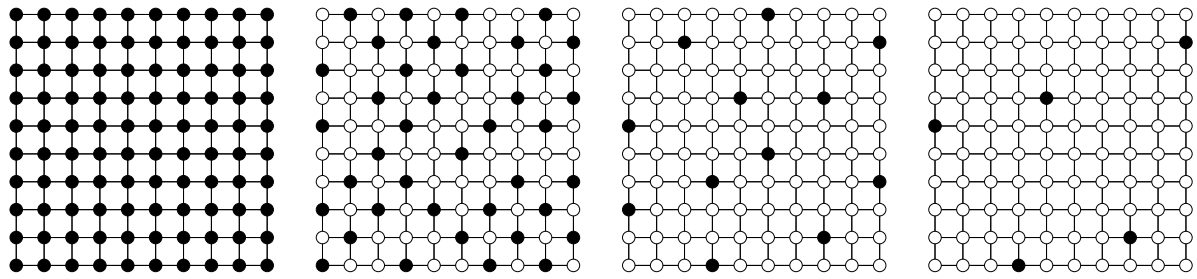
Here $p(V_i) = (p_v)_{v \in V_i}$ is a placement of the nodes of G_i, l is the desired edge length of an edge, and (for a suitable chosen integer c_i) $N^{c_i}(v)$ is the *c_i-neighborhood* of v, defined as $N^{c_i}(v) = \{u \in V_i \mid 0 \leq dist(u, v) \leq c_i\}$. The difference to the original energy of Kamada and Kawai [80] is that only some of the $n-1$ springs that are connected with a node v are considered. The single-level algorithm that is used to obtain a local energy-minimum

configuration of the nodes is the one proposed by Kamada and Kawai, but the induced system contains fewer nonlinear equations.

Harel and Koren [64] state that the asymptotic running time of the `Fast Multi-scale Method` is $\Theta(|V||E|)$. Since the distances between all pairs of nodes have to be memorized, the memory requirements are $\Theta(|V|^2)$. Practical experiments in [64] illustrate that pleasing drawings of the tested graphs, containing up to 6400 nodes, can be generated in less than one minute.

The Method GRIP

Motivated by [62, 64] Gajer et al. [49] and Gajer and Kobourov [50] developed the multilevel algorithm `GRIP`. In this method, the coarsening phase is based on the construction of a *maximum independent set filtration* or *MIS filtration* of the node set V. They define a MIS filtration as a family of sets $\{V =: V_0, V_1, \ldots, V_k\}$ with $\emptyset \subset V_k \subset V_{k-1} \ldots \subset V_0$ so that each V_i with $i \in \{1, \ldots, k\}$ is a maximal subset of V_{i-1} for which the graph-theoretic distance between any pair of its elements is at least $2^{i-1} + 1$. Note that this definition implies that V_1 is a maximal independent set of V (which is a maximal subset of non-adjacent nodes in V) and that the problem of computing a maximum independent set is $\mathcal{NP}$-hard [52]. An example of a MIS filtration is shown in Figure 1.6.



(a) V_0 (b) V_1 (c) V_2 (d) V_3

Figure 1.6: The MIS filtration of a graph that has the structure of a regular square 10×10 grid. The black nodes are the nodes that are contained in the MIS filtrations.

Gajer and Kobourov. [50] use a `Spring Embedder`-like method as single-level algorithm. They define the force that acts on a node $v \in V_i$ at multilevel i as

$$F^i_{GK}(v) = \sum_{u \in N_i(v)} \left(\frac{\|p_v - p_u\|}{l^2 \cdot dist(u, v)} - 1 \right) (p_u - p_v) \,.$$

This force vector is similar to that used in the Kamada Kawai method [80], but is restricted to a suitable chosen subset of V_i that is denoted by $N_i(v)$.

Other notable specifics of `GRIP` are that it computes the MIS filtration only and no edge sets of the coarse graphs $G_0, \ldots, G_k$ that are induced by the filtration. Furthermore,

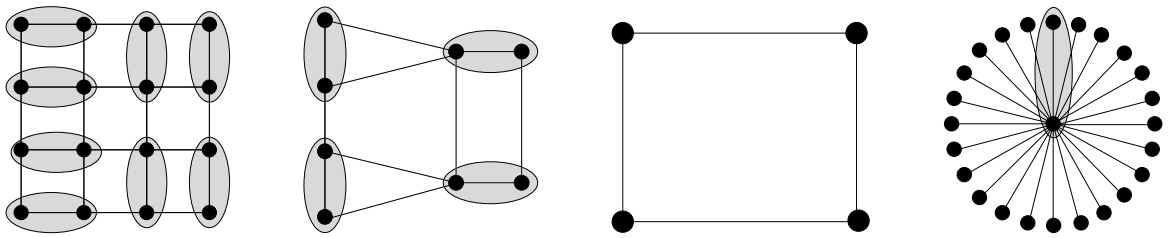
it is designed to place the nodes in an n-dimensional space $n \geq 2$, drawing the graph in this space, and then projecting it into two or three dimensions.

Gajer et al. [49] prove that the asymptotic running time of their algorithm — excluding the time that is needed to construct the MIS filtration — is $\Theta(|V|(\log diam(G)^2))$ for graphs with bounded maximum node degrees, where $diam(G)$ denotes the diameter of G. The total running time of `GRIP` for drawing arbitrary graphs — including the construction of the MIS filtration — has not been proven to be sub-quadratic. The authors state that their algorithm does generate good results for sparse graphs and graphs with a bounded maximum node degree. In practice (compare [49, 50]), the largest tested graph containing 16000 nodes could be drawn in less than a minute.

A Matching-Based Multilevel Method

Maybe the conceptually simplest, but fastest multilevel method in practice is the matching-based multilevel method of Walshaw [132, 133] that is designed for connected graphs. The method is motivated by multilevel ideas in the field of graph partitioning [68, 134].

A *matching* M in a graph $G = (V, E)$ is a subset $S \subseteq E$ so that each node $v \in V$ is adjacent to at most one edge $e \in M$. In order to create a coarse graph G_{i+1} from a coarse graph G_i, first a maximal matching M_i in G_i is created by using a simple linear time heuristic of Hendrickson and Leland [68]. G_{i+1} is obtained from G_i by first making G_{i+1} a copy of G_i, contracting all edges $e \in M_i$ in G_{i+1} to nodes, and finally deleting all multiple edges from G_{i+1}. Figure 1.7(a)-(c) demonstrate the coarsening process at an example.



(a) G_0 (b) G_1 (c) G_2 (d)

Figure 1.7: Coarsening by edge contraction: (a) A graph $G =: G_0$ that represents a regular square grid. The shaded regions mark the edges in M_0. (b) The graph G_1 that is obtained from G_0 and the edges in M_1. (c) The graph G_2 that is obtained from G_1. (d) A star graph that allows only one edge to be matched at each multilevel.

The single-level algorithm that is used to draw the graphs G_i is a variation of the `Grid-Variant Algorithm` of Fruchterman and Reingold [47]. Walshaw (see [133] page 265) states that the running time of the `Grid-Variant Algorithm` for drawing a sparse graph $G_i = (V_i, E_i)$ is "close to" $O(|V_i| + |E_i|)$. Since the construction of G_{i+1} from G_i needs $O(|V_i| + |E_i|)$ time, the best-case running time of the total algorithm — assuming that the number of nodes and edges decrease by a factor two at each level — is linear. The

practical experiments in [133] demonstrate that the algorithm generates well-structured drawings of the tested graphs containing up to 100000 nodes in less than seven minutes.

Unfortunately, the worst-case running time is not analyzed in [132, 133], although this is not difficult: In the worst case, for example, when G is a *star graph* (see Figure 1.7(d)), only a constant number of edges is contained in M_i for every graph G_i. Therefore, the number of multilevels is $\Theta(|V|)$, and $\Theta(|V|)$ coarse graphs $G_i = (V_i, E_i)$ exist with $|V_i| = \Theta(|V|)$. Since the worst-case running time of the `Grid-Variant Algorithm` is $O(|V_i^2| + |E_i|)$ (see Section 1.2.2) the worst-case running time of the multilevel algorithm of Walshaw is $O(|V|(|V|^2 + |E|))$. Consequently, it is likely that this algorithm is not well suited for drawing graphs that contain nodes with a very high degree nor for drawing graphs that induce a placement of $\Theta(|V|)$ nodes in a tiny subregion in a local energy-minimum state of the associated force model. This assumption is confirmed by remarks of Walshaw (see [132] page 182).

1.3.3 Fast Algebraic Methods

Besides force-directed methods, other algorithms that are based on techniques of linear algebra have been introduced into the field of graph drawing. These methods create straight-line drawings of general large graphs and (like force-directed methods) strive for the aesthetic criterion that the drawings should preserve the desired edge length as well as possible. However, these methods do not use models that are based on physical analogies.

High-Dimensional Embedding

Harel and Koren [66] introduce a very fast method for drawing unweighted graphs. It is based on a two phase approach that first generates an embedding of the graph in a very high-dimensional vector space and then projects this drawing into the plane.

The high-dimensional embedding of the graph is generated by first using the linear time algorithm for approximatively solving the k-center problem that has been used in [64]. A fixed value of $k = 50$ is chosen, and k is also the dimension of the high-dimensional vector space. Then, breadth-first search starting from each of the k center nodes is performed resulting in k $|V|$-dimensional vectors that store the graph-theoretic distances of each $v \in V$ to each of the k centers. These vectors are interpreted as a k-dimensional embedding of the graph.

In order to project the high-dimensional embedding of the graph into the plane, the k vectors are used to define a *covariance* matrix S. The x- and y-coordinates of the two-dimensional drawing are obtained by calculating the two eigenvectors of S that are associated with the two largest eigenvalues of S. This is done by using the iterative *power-iteration* method [138].

Assuming fast convergence of the *power-iteration* method, they can prove that this algorithm terminates in linear time. The algorithm can be modified for drawing weighted graphs by replacing breadth-first search with Dijkstra's algorithm (see e.g. [100]) resulting in an asymptotic $O(|V| \log |V| + |E|)$ running time. Practical experiments [66] prove that

this algorithm generates drawings of graphs containing up to 10^6 nodes in less than a minute, which is significantly faster than any force-directed method. The quality of the selected tested graphs is often comparable with that of force-directed methods, although the authors note that "frequently, the static 2-D results are inferior to those of the force-directed approach" and that their algorithm "is not suitable for drawing trees" (see [66] pages 218 and 216, respectively).

The last is not surprising, since it is easy to see that the positions of nodes in a k-dimensional embedding of a non-biconnected graph are not necessarily distinct (see also [15]). We will demonstrate this statement by giving an example:

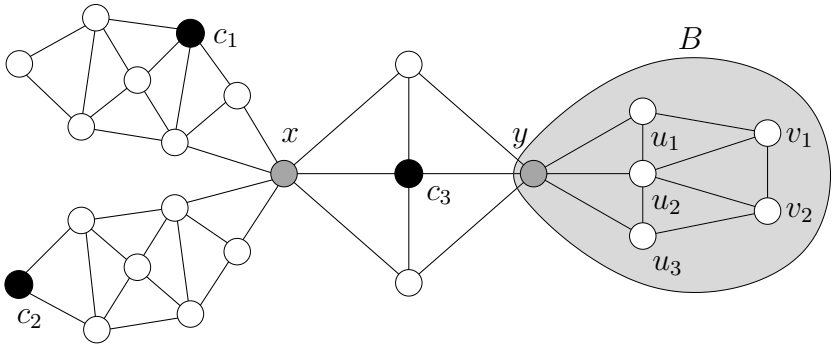


Figure 1.8: A graph that is not biconnected. The cut vertices are x and y. The black nodes c_1, c_2 and c_3 are the $k := 3$ selected center nodes. B is a block that does not contain any center node.

Suppose, we have given a non-biconnected graph as shown in Figure 1.8 and the black nodes c_1, c_2, and c_3 are the $k := 3$ center nodes. Furthermore, suppose that block B does not contain any center node. (Such a block does always exist if k is smaller than the number of blocks of the given graph.) The coordinates of each node $v \in V$ in the k-dimensional embedding are defined by $(dist(v, c_1), dist(v, c_2), \ldots, dist\ (v, c_k))$. Since in Figure 1.8 all paths that connect a center node c_i with a node contained in B visit the cut-vertex y, we get that all nodes $w_i, w_j \in B$ with $dist(y, w_i) = dist(y, w_j)$ are assigned the same k-dimensional coordinates. In particular, u_1, u_2, and u_3 are assigned the coordinate $(5, 6, 2)$ while v_1 and v_2 are assigned the coordinate $(6, 7, 3)$.

The Method ACE

Another fast but more complicated algorithm for drawing large (weighted) graphs is the method `ACE` of Koren et al. [86, 87]. Motivated by a quadratic placement algorithm of Hall [63], they define the quadratic optimization problem

$$(P) \qquad min\ x^T L x \quad \text{so that} \quad x^T x = 1 \quad \text{in the subspace} \quad x^T 1_n = 0\,.$$

Here $n = |V|$, w_{ij} is the weight of an edge $e = (v_i, v_j)$, and L is the *Laplacian* matrix of G that is defined as

$$L_{ij} = \begin{cases} \sum_{k=1}^{n} w_{ik} & \text{if} \quad i = j \\ -w_{ij} & \text{if} \quad i \neq j \end{cases} \quad i, j = 1, \ldots, n.$$

The minimum of (P) is obtained by the eigenvector that corresponds to the smallest positive eigenvalue of L and is called *Fiedler vector*. The problem of drawing the graph G in two dimensions is reduced to the problem of finding the two eigenvectors of L that are associated with the two smallest eigenvalues of L.

Instead of calculating the eigenvectors directly an *algebraic multigrid algorithm* is created. Similar to the multilevel ideas, the idea is to express the originally high-dimensional problem in lower and lower dimensions, solving the problem at the lowest dimension, and progressively solving a high-dimensional problem by using the solutions of the low-dimensional problems.

In practical experiments [86, 87], `ACE` generates drawings of selected graphs containing up to 7000000 nodes in less than two minutes. The quality of the presented drawings is comparable with that of force-directed methods, but the authors mention that "there is nothing in Hall's energy that prevents nodes from being very close. Hence, the drawings might show dense local arrangements" (see [87] page 669). The authors do not give an upper bound on the asymptotic running time of `ACE` in the number of nodes and edges, but Harel and Koren argue (see [66] page 218) that the running time of `ACE` depends on the graph's structure. This can be confirmed by concentrating only on one particular step of the used multigrid method:

In order to transform a high-dimensional problem into a low-dimensional problem an *interpolation* matrix is created. The authors develop two ways for constructing such matrices. One of this methods (that is also used as the standard method in the implementation of `ACE` which can be obtained from [88]) calculates these matrices by using an edge contracting method. An interpolation matrix is computed by fining a maximal matching in a graph and then contracting these edges to nodes in order to obtain a new graph. Starting with the given graph G, this is repeated until the smallest graphs contains only a constant number of nodes. Analogue to the force-directed graph-drawing method of Walshaw [132, 133], this can result in a quadratic running time (e.g. if the graph is a star graph).

Related Methods

Several methods that are related to the previous mentioned algebraic methods have been published:

In [85] Koren introduces a two-phase method for drawing graphs that is based on first generating a subspace that captures a desired good layout of a graph and then solving optimization problems in the subspace by using algebraic methods. The solutions of this optimization problems determine the drawings. As suitable subspaces he identifies the high-dimensional embedding that he developed in [66] and the first k eigenvectors of the Laplacian that he already used in the context of the algorithm `ACE` [86, 87].

Furthermore, the method `ACE` is used as an important part of an algorithm that is designed to generate layered drawings of directed graphs [22, 21].

A variation of the high-dimensional embedding approach of Harel and Koren [66] and a method that is based on ideas of the algorithm of Hall [63] have been implemented in the framework of the software package visone [15].

1.3.4 How to Display Drawings of Large Graphs

In this sub-section we assume that the straight-line drawing of the graph (that is completely described by assigning each node a position in $\mathbb{R}^2$) has already been calculated. One remaining problem is that it is impossible do display each graphical detail of arbitrary large graphs on a display medium with a bounded resolution. Therefore, several techniques have been developed to deal with this problem and are sketched in the following.

For example, in many software tools *zoom and pan* techniques are used that allow the user to zoom in a region of interest and to navigate through the graph.

Fisheye views are also common techniques in image visualization and graph drawing. These techniques imitate the fisheye-lens effect by enlarging an area of interest and showing the remainder of the image/graph less detailed [48, 117, 83].

Another way to display drawings of large graphs is to use *multilevel visualization techniques*. They have been introduced into graph drawing by Eades and Feng [38] to display clustered graphs. The methods have in common that, additionally to the given original drawing of the graph G, drawings of smaller graphs are generated and displayed. The drawings of the smaller graphs are obtained from the drawings of the larger graphs by using clustering methods [33, 34, 110]. It is important to note in this context that such multilevel drawings can be generated by force-directed multilevel methods as a by-product.

A technique that combines a fisheye view technique with that of multilevel visualization techniques has recently been developed by Ganser et al. [51].

Incremental exploration techniques display only a small subgraph of a given graph, which fits into a visible window and allow the user to move this window. In contrast to the zoom and pan techniques, these methods are used if a drawing of the whole graph does not exist. They incrementally calculate only the drawings of the subgraphs that fit into the visible window (see for example [73, 72, 37]). A survey on graph visualization and navigation techniques is [69].

Chapter 2

The Fast Multipole Multilevel Method

> Modelle sollten sich bemühen,
> dem Porträt ähnlich zu sehen. [1]

In Section 2.1 we will motivate the development of a new force-directed method that is designed for drawing large graphs, and the main goals of this approach will be formulated. The basic concept of this method will be sketched in Section 2.2.

2.1 Motivation and Goals

The previous discussion has shown that several force-directed algorithms exist that, in practice, generate well-structured drawings of selected classes of large graphs fast. But there remain some challenges:

First of all, none of the force-directed algorithms guarantees a sub-quadratic running time in general. Additionally, the running time of the algebraic method "ACE (like that of force-directed methods) depends on the graph's structure" (see [66] page 218). This is undesirable for algorithms that are designed for drawing general graphs.

Second, there exist classes of graphs for which the quality of the drawings might be improved. For example, C. Walshaw states that it "is likely that very dense graphs or even those which have a dense substructure are never going to be good candidates for any force-directed placement algorithm, and ours is no exception" (see [132] page 182). Gajer et al. (see [49] page 220) state that their algorithm "works very well for sparse graphs and graphs of low degree", but "it does not produce high quality drawings for all graphs". Harel and Koren (see [66] page 216) mention that their "algorithm is not suitable for drawing trees".

[1] Salvador Dali

Third, several of the algorithms that we introduced in Section 1.3 assume that the given graphs are connected or unweighted, and that all nodes have the same sizes and shapes. These restrictions should be avoided, since one important property of force-directed methods is that they should be able to draw general graphs.

An example of general graphs that arise in real-world applications are social networks. There, nodes correspond to persons, and family relationships can be modeled by edges of different weights that reflect the importance of the family relationships. For example, the edge that connects a father f and his child c has a different weight as the edge between f and his niece. The intuition is that strong familiar relationships should be displayed by nearby positions of the corresponding persons/nodes in the drawing. It might appear that the social relationship between a person p and a person q is not unique. For example, p might be the husband of q but also the cousin of q. Such multiple edges appear quite frequently in several social networks of the medieval German town Esslingen that has been extensively studied by C. Lipp (a complete list of all related publications of C. Lipp is available from [90]). Additionally, the nodes might store information like the name, sex, profession, and age of a person. Hence, the size and shape of the nodes is predefined.

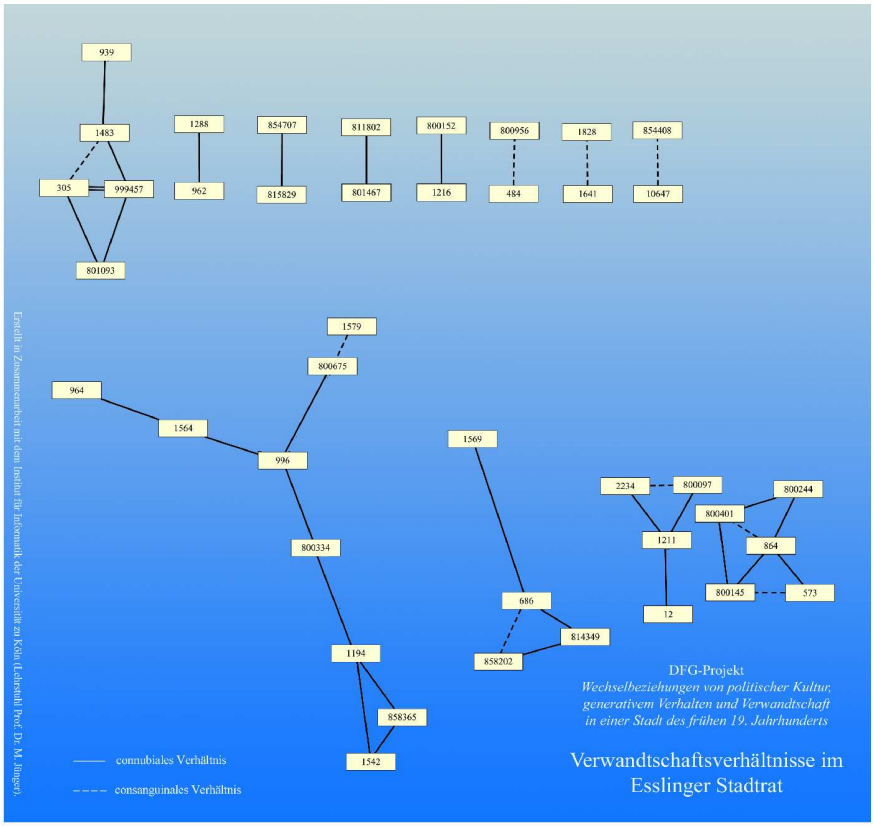


Figure 2.1: A disconnected weighted graph — displaying the family relationships of members of a political committee in the German town Esslingen in the early 19th century — that was drawn with an early version of a new force-directed method (FM^3) in cooperation with C. Lipp [90]. The edge weights are hidden.

Figure 2.1 and Figure 2.2 are examples of weighted disconnected graphs with nodes of predefined shapes and sizes. The graphs describe the family relationship in a political committee in the German town Esslingen (Figure 2.1) and clubs and societies in this town (Figure 2.2), respectively.

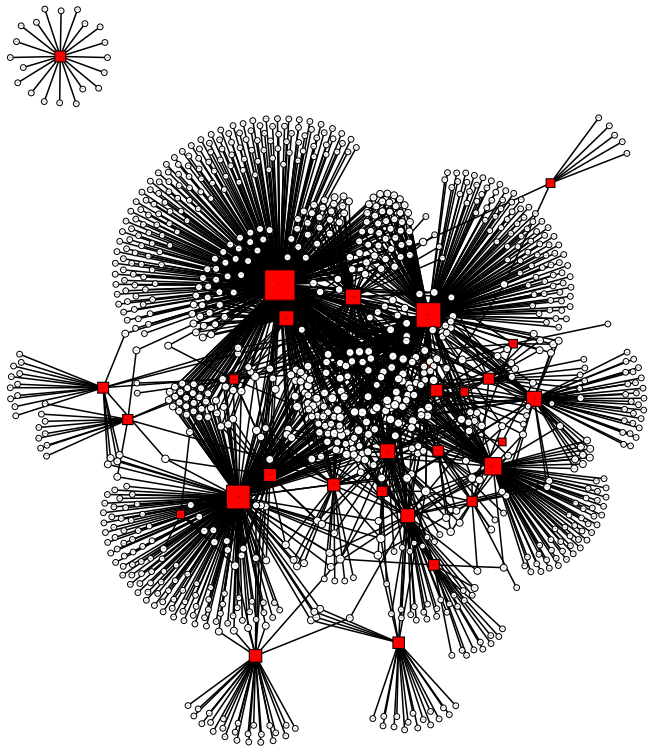


Figure 2.2: A disconnected weighted graph — displaying clubs and societies in the town Esslingen. The red nodes correspond to the clubs while the white nodes represent the people that are members of the clubs. The sizes of the nodes display the sizes of the clubs. This graph was drawn with a new force-directed method (FM^3) in cooperation with C. Lipp [90]. The edge weights are hidden.

Since another important goal of graph-drawing algorithms is the minimization of the used drawing area, we would expect that a good graph-drawing algorithm should take this criterion into account, too. Additionally, it is often desirable that the drawing should fit into a rectangular drawing area of a predefined fixed aspect ratio r. The latter is important in practice, since the aspect ratio of the drawing area will be restricted — by preferences of the user or by output devices like the shape of the computer screen or the page format of the plotter — in any case. Hence, it is reasonable to design graph-drawing algorithms that try to minimize the used aspect-ratio area according to a desired aspect ratio r.

Motivated by the previous observations, we strive for a force-directed method that should achieve the following goals:

- The algorithm should be designed for drawing general graphs and should not restrict on special graph classes only.
- Besides typical aesthetic requirements (like preserving the desired edge length, non-overlapping nodes, and the display of symmetry) the aspect-ratio area of the drawing should be minimized.
- The algorithm should guarantee a sub-quadratic worst-case running time.
- The algorithm should be fast in practice, too.
- The algorithm should generate well-structured drawings of a wide range of graphs.

2.2 The Basic Concept

We will now give an overview of the basic concept of a new force-directed method that we called ***F**ast **M**ultipole **M**ultilevel **M**ethod* or shorter FM^3. It has been developed to reach the previously stated goals.

2.2.1 The Input and Output Requirements

Since the algorithm is designed to generate drawings of general graphs, the input instances are graphs $G = (V, E)$ that are connected or disconnected, directed or undirected, weighted or unweighted. For weighted graphs, we interpret the edge weight of each edge as its desired edge length, while the desired edge length of all unweighted graphs are assumed to be identical for all edges and can be chosen by the user. This interpretation of the edge weight is only reasonable under the assumption that the edge weights are positive. If the graph contains edges with negative or zero edge weights, one can obtain positive edge weights by adding an appropriate positive offset on all edge weights.

Furthermore, the nodes of the graph may have predefined shapes and sizes, and the user is allowed to determine a predefined aspect ratio r of the desired rectangular drawing area.

FM^3 is designed to generate a drawing $\Gamma(G)$ of G in the plane with non-overlapping components. Except self-loops (that are drawn as loops) all edges are drawn as straight lines. Multiple edges are drawn parallel. Directed edges are drawn as arcs.

2.2.2 The Choice of the Force Model

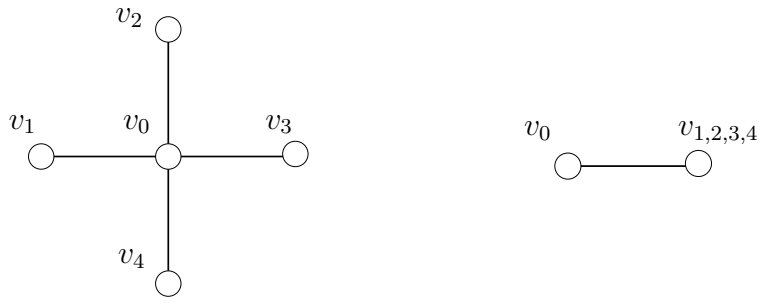
Next, we have to choose a suitable force model. On the one hand, the force model should allow the development of a sub-quadratic algorithm that is used to find a energy-minimal configuration of the nodes. One the other hand, it should model the most important aesthetic requirements on force-directed drawings (preserving the desired edge length, non-overlapping nodes, display of symmetry).

Therefore, we choose a simple model which identifies the nodes with equally charged particles that repel each other and that identifies edges with springs, similar to many

classical force-directed methods (see Section 1.2.2). If in $\mathbb{R}^2$ two charged particles u, v are placed a distance d from each other, the repulsive forces acting between u and v are proportional to $1/d$. Our choice of the spring forces is not strictly related to physical reality. We found that choosing the spring force of an edge $e = (u, v)$ to be proportional to $\log(d/l^{zero}(e)) \cdot d^2$ gives very good results in practice. Here d is the distance between u and v, and $l^{zero}(e)$ (with $l^{zero}(e) > 0$) denotes the *zero-energy length* of a spring $e \in E$. The appropriate choice of $l^{zero}(e)$ will be discussed in Section 3.1 and in Chapter 4.

The spring force of a spring $e = (u, v)$ is defined so that an energy-minimum state of the spring is reached — when ignoring all other springs and charged particles in the system — if the distance between u and v is equal to $l^{zero}(e)$. Furthermore, this model allows to assign each edge its individual zero-energy length, which will be of fundamental importance in the *preprocessing step* (see Section 3.1) and the *multilevel step* (see Chapter 4).

The repulsive forces of the particles imply that the nodes "spread well"(compare [14] page 71) in the plane. We try to illustrate the meaning of this fuzzy attribute at an example: Suppose that the given graph is a star graph that consists of five nodes, four edges, and all edges have the equal desired edge length l, and $l^{zero}(e) := l$. Then, an energy-minimum configuration of the nodes in our force model would distribute the four edges symmetrically around the center of the star like in Figure 2.3(a). Suppose, the same definition of the spring forces would be used, but no repulsive forces between the nodes would be defined. Then, even a placement of the nodes, in which all but the center of the star are placed at one point with distance l to the center, would induce an energy-minimum configuration in this model (see Figure 2.3(b)). And, of course, a good drawing should not contain nodes that overlap.



(a) (b)

Figure 2.3: (a) An energy-minimum drawing of a star graph in our force model. (b) An energy-minimum drawing of the same graph in a variation of our model that does not define repulsive forces acting between all pairs of nodes.

We do not distinguish between directed and undirected graphs in this model. But this could be done, for example, by introducing a magnetic force field and magnetic springs like in [124].

Like all other state-of-the-art methods for drawing large graphs, we did not model another important criterion, the reduction of the number of edge crossings, explicitly. In some cases (e.g. if G is a grid graph and the zero-energy length $l^{zero}(e)$ are identical for all $e \in E$) an energy-minimum placement of the nodes does necessarily imply a planar drawing in our model. In general, however, one cannot expect that an energy-minimum configuration of the nodes in a model that does only rely on identifying nodes with charged particles and edges with springs will imply a crossing-free straight-line drawing of an associated planar graph. This is confirmed by our experimental results in Section 7.8.2.

Nevertheless, as can be seen in in Section 7.6, even the choice of such a simple force model — in combination with a suitable energy-minimization algorithm — implies energy-minimal placements of the nodes that correspond to well-structured drawings of planar and non-planar graphs.

2.2.3 The Algorithm

Now we sketch the basic components of the algorithm FM³ that is used to find an energy-minimal configuration of the nodes. The detailed description of these components will be formulated in Chapters 3, 4, 5, and Section 6.1. Additionally, the inclusion relationships of these steps are illustrated in Figure 2.4.

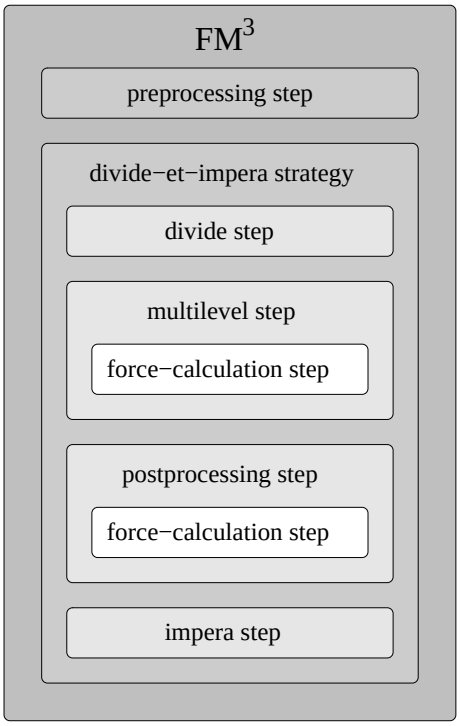


Figure 2.4: Inclusions of the basic algorithmic steps of FM³.

First, all requirements like the desired edge length and the node attributes have to be represented in the force model. This is done in the *preprocessing step* that we will describe

detailedly in Section 3.1. There, we will reduce the problem of drawing a weighted or unweighted graph G with nodes of different fixed sizes and shapes to the problem of drawing a positive-weighted undirected simple graph without node attributes using our force model.

In this model, an energy-minimum configuration of the nodes of a disconnected graph would require the components to be placed infinitely far away from each other. On the other hand, we require from a good drawing that it should use only a small amount of aspect-ratio area. This problem is solved using a *divide-et-impera strategy* (Chapter 3.2). In the *divide step* we find the components of the given graph. Then, drawings of the induced maximal connected subgraphs are generated in the *multilevel step* (Chapter 4). A refinement of these drawings is made in the *postprocessing step* (Section 6.1). Finally, in the *impera step* these drawings are put together in order to obtain a drawing in which the components do not overlap and that uses few aspect-ratio area.

In the multilevel step we can assume that the given graphs G are connected positive-weighted undirected simple graphs. Each of these graphs is drawn using a multilevel strategy that first creates a series of graphs $G =: G_0, G_1, \ldots, G_k$ with decreasing sizes. Then, the smallest graph G_k is drawn using a force-directed algorithm that is introduced in the *force-calculation step* (Chapter 5). This drawing is used to get an initial layout of the next larger graph G_{k-1} that is drawn afterwards. This process is repeated until the original graph G_0 is drawn.

In the postprocessing step the force-calculation step is reused to optimize the drawings of each maximal connected subgraph of the original graph.

Finally, in Chapter 5, we will introduce an iterative algorithm that tries to find an energy-minimal configuration of the nodes/particles. The most important part of this step is the accurate approximation of the repulsive forces acting between all pairs of charged particles. This is done by evaluating the field of potential energy in the system of charged particles using so called *multipole expansions* and a hierarchical data structure called *reduced bucket quadtree*.

Chapter 3

The Preprocessing Step and the Divide-Et-Impera Strategy

> Um das Große zu schätzen und zu lieben,
> musst du dich erst an dem Kleinen üben. [1]

In Section 3.1 we will explain how to transfer the problem of drawing a general graph $G = (V, E)$ — that is weighted or unweighted, directed or undirected, and that may contain nodes of different sizes and shapes — into the problem of drawing a positive-weighted undirected simple graph without node attributes. In Section 3.2, we will address the problem of drawing disconnected positive-weighted undirected simple graphs.

3.1 The Preprocessing Step

3.1.1 Reduction to Positive-Weighted Undirected Simple Graphs

In the last chapter we already pointed out that, for simplicity, we do not distinguish between directed or undirected graphs in our model. Therefore, in the following, we can assume that the given graph G is undirected by ignoring the orientation of the edges in the complementary case.

If the given graph $G = (V, E)$ is weighted and the edge weight $w(e)$ of each edge $e \in E$ is positive, we interpret the edge weight of each edge $e \in E$ as its individual desired edge length and set $\mathit{desired\_edge\_length}(e) := w(e)$ for all $e \in E$. If the graph is weighted, some edge weights are not positive, and w_{min} is the smallest non-positive edge weight of all edges, then we set $\mathit{desired\_edge\_length}(e) := w(e) + |w_{min}| + \epsilon$ for all $e \in E$, where $\epsilon > 0$ is a constant. Finally, if the given graph is unweighted, it is natural to assume that all edges should have the same non-negative desired edge length l that can be chosen by the user. Hence, we can set $\mathit{desired\_edge\_length}(e) := l$ for all edges $e \in E$.

[1] Georg Keil

If G is not simple, we do the following: Multiple edges $e_1, \ldots, e_k$ that connect two nodes u and v are replaced by exactly one edge $e_{new} = (u, v)$, and the desired edge length of e_{new} is set to the average of the desired edge length of $e_1, \ldots, e_k$. Self-loops $e = (v, v)$ can be deleted from G since their drawing is determined only by the position of v in the drawing $\Gamma(G)$ of G.

3.1.2 Drawing Graphs with Node Attributes

Now we explain how it is possible to integrate the desired edge length and the fact that nodes may have predefined fixed sizes and shapes into our force model.

In the easiest case the nodes are represented by points, and we can simply set the zero-energy length of each edge $e \in E$ to $l^{zero}(e) := \mathit{desired\_edge\_length}(e)$.

In the remaining case the nodes may have different shapes of predefined fixed sizes. The problem that arises here is that node overlaps might appear in the drawing of the graph G. Additionally, edges might cross the graphics of the nodes more likely than in the case that nodes are represented by points. Both eventualities are undesirable.

Wang and Miyamoto [135] have invented a force-directed model that tries to overcome this problem by modifying the force model of Fruchterman and Reingold [47]. In particular, they modify the definition of the strength of the spring force ($F_{spring}(u, v)$) of an edge (u, v) and the definition of strength of the repulsive force ($F_{rep}(u, v)$) acting between two nodes u and v as follows:

$$\|F_{spring}(u,v)\| = \begin{cases} 0 & \text{if } u \text{ and } v \text{ overlap} \\ \frac{d_{out}^2}{l+d_{in}} & \text{otherwise} \end{cases}; \quad \|F_{rep}(u,v)\| = \begin{cases} C\frac{l^2}{d} & \text{if } u \text{ and } v \text{ overlap} \\ \frac{l^2}{d} & \text{otherwise} \end{cases}$$

Here l is the desired edge length of all edges $e \in E$, $C > 1$ is a constant, d is the distance between the centers of u and v, d_{out} is the distance between the boundaries of the graphics of node u and v, and $d_{in} = d - d_{out}$. The idea of this revised model is that for an edge $e = (u, v)$ the forces $F_{spring}(u, v)$ and $F_{rep}(u, v)$ cancel each other if d_{out} and l are equal. Vertex overlaps shall be reduced by increasing the repulsive forces and canceling the spring forces for each pair of nodes that overlap.

One drawback of this model is that it requires to determine whether each pair of nodes is overlapping or not. Doing this naively requires $\Theta(|V|^2)$ time. Furthermore, this model does not allow to assign each edge its individual desired edge length.

Harel and Koren [65] and Tunkelang [128] developed other force-directed techniques for drawing graphs with node attributes that are related to that of Wang and Miyamoto [135]. They define suitable spring forces and repulsive forces in a `Spring Embedder`-like force model. Another model that is presented by Harel and Koren [65] is a modification of the complete spring model of Kamada and Kawai [80] and defines the zero-energy length of each edge $e = (u, v) \in E$ for a given actual placement of the nodes as the sum of the global desired edge length l and the length of the line segment that connects the nodes u and v and lies inside the graphics of u and v, respectively. A method that is designed to handle nodes

of different sizes in two and three dimensions that is based on placing charged particles on the boundaries of the nodes is presented in Chuang et al. [23]. All these methods and models are based on the assumption that all edges have the same desired edge length.

Since our method should be more flexible and allow each edge to have its individual desired edge length, we choose a different way to handle the problem of drawing graphs with node attributes: We do neither change the definition of the spring forces nor the definition of the repulsive forces in our model. Instead, we simply set the zero-energy length l^{zero} of each edge $e = (u, v) \in E$ to:

$$l^{zero}(e) = rad(u) + rad(v) + desired\_edge\_length(e)$$

Here rad(v) denotes the radius of the smallest circle that surrounds the graphics of node v. Figure 3.1 illustrates the definition of the zero-energy length at an example.

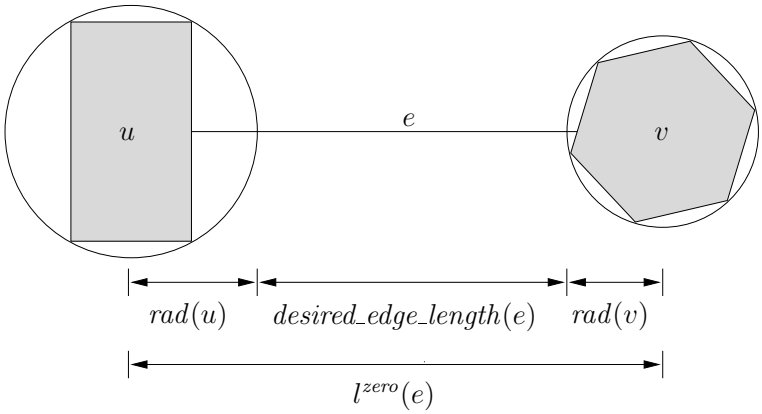
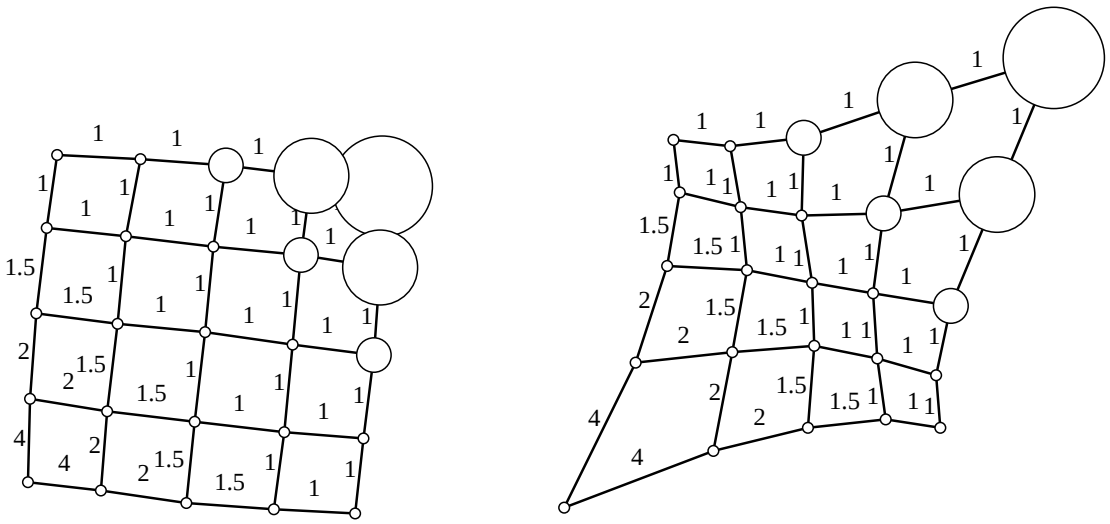


Figure 3.1: Definition of the zero-energy length of a spring $e = (u, v)$ with nodes of different shapes and sizes and a given desired edge length.

One advantage of this model — besides its simplicity and flexibility — is that the calculation of the zero-energy length of the springs has to be done only once, whereas in all previous introduced methods the graphical information of each node has to be used at each iteration of the used force-directed algorithms.

Furthermore, the fact that our definition of the repulsive forces does exclusively depend on the positions of the nodes will allow the development of an extremely fast and accurate method for approximating the repulsive forces that act between all pairs of nodes/particles.

On the other hand, it might be possible that the other more complicated techniques are better suited for representing nodes of different sizes and shapes (regarding the quality of the generated drawings) in practice. However, our method is designed for drawing graphs containing several thousands of nodes and, therefore, has to be optimized concerning the running time. Figure 3.2 shows example drawings of a weighted graph with nodes of different sizes. One drawing (see Figure 3.2(a)) has been generated with a force-directed algorithm that ignores edge weights and node attributes and contains overlapping nodes. The other drawing (see Figure 3.2(b)) has been generated with FM^3 and displays the edge weights in an appropriate way.



(a) (b)

Figure 3.2: Two drawings of a weighted undirected graph with nodes of different sizes. (a) A drawing that is generated with a force-directed algorithm that ignores edge weights and node attributes. (b) A drawing that is generated with FM^3.

3.1.3 Formal Description of the Preprocessing Step

Theorem 3.1 (Preprocessing Step). *In Function* `Preprocessing_Step` *an (un)weighted (un)directed graph $G = (V, E)$ that may contain nodes of different fixed sizes and shapes is transfered into a positive-weighted undirected simple graph without node attributes $G' = (V', E', l^{zero})$ with $V' = V$ and $E' \subseteq E$ in $O(|V| + |E|)$ time. The memory requirements of Function* `Preprocessing_Step` *are $O(|V| + |E|)$.*

Proof. The only non-trivial part of the preprocessing step is the transformation of a graph $G = (V, E)$ into a simple Graph G' in linear time. This can be done as follows:

Suppose, the nodes of G have the indices $1, 2, \ldots, |V|$, and that we mark for each edge $e = (i_1, i_2)$ the node with the larger index as the *large node* of e and the other node as the *small node*. If e is a self-loop the choice of the larger/smaller node is arbitrary.

Suppose that L is a list of the edges contained in E. First, we sort L in order to obtain a list in which multiple edges are in consecutive order. This can be done in $O(|V| + |E|)$ time by using a variation of bucket sort:

We create a set B_{large} of $|V|$ buckets that represent the values $\{1, \ldots, |V|\}$ and assign each edge $e \in E$ to the bucket that corresponds to the index of its large node. We create a second set B_{small} of $|V|$ buckets that represent the values $\{1, \ldots, |V|\}$. Then, we traverse the buckets of B_{large} in increasing order (according to the bucket indices) and insert each edge e into the bucket of B_{small} that corresponds to the index of the small node of e.

Finally, we traverse B_{small} in increasing order (according to the bucket indices) to obtain the sorted list of edges L.

In order to delete all self-loops and to replace each set of multiple edges in E by one new edge we only have to traverse L in increasing order. The running time and memory requirement of this procedure are linear in $|V| + |E|$. □

Function `Preprocessing_Step`(G)

input : an (un)weighted (un)directed graph $G = (V, E)$ that may contain nodes of different fixed sizes and shapes

output: a positive-weighted undirected simple graph $G' = (V', E', l^{zero})$ without node attributes

```
begin
    G' ← G;
    if G' is directed then make G' undirected;
    if G' is not positive weighted then make G' positive weighted;
    if G' is not simple then make G' simple;
    foreach e ∈ E' do
        calculate the zero-energy length l^zero(e);
end
```

3.2 The Divide-Et-Impera Strategy

In this section we will address the problem of drawing disconnected graphs and will review existing methods for drawing these graphs. After analyzing the complexity of the disconnected-graph layout problem that asks for a drawing of a graph with non-overlapping components, which uses a minimum amount of aspect-ratio area, we will present a simple heuristic to solve this problem.

3.2.1 Force-Directed Methods for Drawing Disconnected Graphs

In many force models an energy-minimum configuration of the nodes of a disconnected graph would require the components to be placed infinitely far away from each other. For example, this situation arises in all force models that only rely on identifying nodes with equally charged particles and on identifying edges with springs. Hence, force-directed algorithms that are based on theses models are not suited for drawing disconnected graphs. One way to solve this problem is to fix the drawing area in advance, like in the force models of Fruchterman and Reingold [47] or Davidson and Harel [27]: Fruchterman and Reingold [47] have interpreted the boundaries of the fixed drawing area as elastic or plastic walls that cannot be passed by the nodes. In contrast to this, Davidson and Harel [27]

have added a term to their energy functional that forces the nodes to be placed not too near to the borders of the drawing area.

These models are sufficient to keep the components close to each other. But they also have several disadvantages: First, these heuristics do not guarantee that the components will not overlap, which is undesirable. Second, the drawing of each component of a graph is influenced by the forces induced by the other components and the restricted drawing area. From the graph drawing viewpoint it is more desirable to draw a graph G always in the same way (modulo rotations and translations) — independent whether G is a component of a larger graph or whether G defines already the entire graph. Finally, it might be difficult to estimate a reasonable size of the drawing area without having any knowledge about the structure of the given graph.

3.2.2 The Disconnected-Graph Layout Problem: Complexity & Algorithms

The problems that are stated in the previous section can be avoided by first drawing each maximal connected subgraph $G_1, \ldots, G_c$ of a given graph G with a suitable algorithm and then putting these drawing together to obtain a drawing of the entire graph G. Here C is the set of components of G and $c := |C|$. In the following, we presuppose that the terms bounding rectangle, drawing area, aspect ratio, and aspect-ratio area that have been defined in Section 1.1.4 are known.

An informal description of the disconnected-graph layout problem is as follows: Given a graph G that contains $c \geq 1$ maximal connected subgraphs $G_1, \ldots, G_c$, a desired aspect ratio r, and for each G_i a drawing $\Gamma(G_i), i \in \{1, \ldots, c\}$. Find a drawing $\Gamma(G)$ with non-overlapping components that requires minimum aspect-ratio area according to r.

Identifying the drawing of each component with a polygon, we can now formally define the *general disconnected-graph layout problem* (GDGL):

(GDGL)	Given a set $S = \{S_1, \ldots, S_c\}$ of polygons, an aspect ratio $r > 0$, and integer constant $k_1, k_2, k_3 \geq 0$. Find a placement of $S_1, \ldots, S_c$ in the plane with non-overlapping S_i that uses a minimum amount of aspect-ratio area according to r. Translations, k_1 reflections on the x-axes, k_2 reflections on the y-axes, and k_3 rotations with predefined fixed angles are allowed for each S_i.

We additionally define the *constraint disconnected-graph layout problem* (CDGL) that does not allow reflections and rotations:

(CDGL)	Given a set $S = \{S_1, \ldots, S_c\}$ of polygons and an aspect ratio $r > 0$. Find a placement of $S_1, \ldots, S_c$ in the plane with non-overlapping S_i that uses a minimum amount of aspect-ratio area according to r. Only translations of the S_i are allowed.

Theorem 3.2 (The Disconnected-Graph Layout Problem). *The Problems* (GDGL) *and* (CDGL) *are $\mathcal{NP}$-hard.*

Proof. First, we prove that the following decision problem is $\mathcal{NP}$-complete:

(P)	Given a set $R = \{R_1, \ldots, R_c\}$ of axis parallel rectangles and positive constants r and A. Does there exist a placement of $R_1, \ldots, R_c$ in the plane with non-overlapping R_i that uses at most A aspect-ratio area according to the desired aspect ratio r? The rectangles must be placed orthogonally and may not be rotated.

The *strip-packing problem* (S) that is also known as *minimum height two-dimensional packing problem* is $\mathcal{NP}$-complete [7, 119] and defined as follows:

(S)	Given a set $R = \{R_1, \ldots, R_c\}$ of axis parallel rectangles with breadth $b_i \leq 1$ and height $h_i > 0$ for $i = \{1, \ldots, c\}$ and a constant $H > 0$. Does there exist a packing of $R_1, \ldots, R_c$ with non-overlapping R_i into a strip of breadth 1 and height of at most H? The rectangles must be packed orthogonally and may not be rotated.

Clearly, (P) is in $\mathcal{NP}$. Now we reduce (S) to (P):

Let $R = \{R_1, \ldots, R_c\}$, H be an instance of (S). Then, $R = \{R_1, \ldots, R_c\}$, $r := \frac{1}{H}$, $A := H$ is an instance of (P). Suppose that there exists an overlapping-free placement of the R_i into a strip of breadth 1 and height at most H. Then, there exists a rectangle of breadth 1 and height H that covers this placement, uses H area, and has aspect ratio $\frac{1}{H}$. Hence, there exists a packing of the R_i with aspect ratio $r = \frac{1}{H}$ that uses at most $A = H$ area. Now suppose that there exists an overlapping-free placement of the R_i that uses at most $A = H$ aspect-ratio area according to aspect ratio $r = \frac{1}{H}$. Therefore, all R_i are covered by a rectangle that uses H area and has aspect ratio exactly $\frac{1}{H}$. Hence, the breadth and height of this rectangle are exactly 1 and H, respectively. This implies that there exists an overlapping-free placement of the R_i into a strip of breadth 1 with height at most H.

Since (P) is $\mathcal{NP}$-complete, the decision problems that correspond to (CDGL) and (GDGL) are $\mathcal{NP}$-complete, too (this can be seen by setting $S := R = \{R_1, \ldots, R_c\}$ in (CDGL) and by additionally setting $k_1 = k_2 = k_3 = 0$ in (GDGL)). Since both (CDGL) and (GDGL) are in *NPO*, they are $\mathcal{NP}$-hard. □

Since we are interested in fast algorithms and not even thought about trying to prove that $\mathcal{P} = \mathcal{NP}$, it is reasonable to concentrate on heuristic solutions of the problems (GDGL) and (CDGL) in the following.

Motivated by classical methods for solving two-dimensional packing problems [7, 24, 119, 89], several heuristics have been invented to solve (CDGL) by Dogrusoz [32] and Freivalds et al. [44]. Theorem 3.2 justifies the development of heuristics for this problem.

Dogrusoz [32] identifies the drawing of each component of a graph with its bounding rectangle. These rectangles are denoted by $R_1, \ldots, R_{|C|}$ in the following. Dogrusoz [32] introduces three heuristics for placing these rectangles:

In the *strip-packing method*, first a fixed width of the drawing area is calculated. This is done by taking the desired aspect ratio and the sizes of the rectangles into account. Then, a simple strip-packing heuristic called *best-fit decreasing-height* is used to place the rectangles: First, the rectangles are sorted by non-increasing height. Then, the rectangles are placed left-justified into levels/rows. Each rectangle is placed into the row that implies the minimum left space to the right border of the strip. A new row is created if the actual rectangle does not fit in any existing row. The problem of this method is that the width of the strip has to be fixed in advance, and a bad choice of this width might result in a placement of the components that uses much aspect-ratio area. The running time of this method is $O(|C| \log |C|)$.

This problem is avoided in the *tiling method* in which the rectangles are placed into rows, too. The tiling method starts by creating an initial row and placing the first rectangle R_1 in this row. Then, a decision is made whether the next rectangle should be placed into one of the existing rows (that with the least utilization) or into a newly created row. The chosen placement is the placement that uses the fewest aspect-ratio area according to the desired aspect ratio r. Experimental results in [32] show that the aspect-ratio area could be reduced by adding a preprocessing step that sorts the rectangles by non-increasing height. Therefore, this method can also be seen as a variant of the *strip-packing method* with a dynamically changing width. The running time of the tiling method is $O(|C| \log |C|)$.

The *alternate-bisection method* recursively subdivides the set of rectangles as follows: The set of rectangles is split into two sets using the area of the rectangles as a metric and each partitioning is recursively laid out. This is repeated until a partitioning consists only of a constant number of rectangles whose placement is trivial. When placing two already placed partitions relatively, the orientation is alternated from left and right to top and down. For this step *ordered 1D packing* is used. For example, ordered 1D packing along the x-axis corresponds to the process of ordering the rectangles with respect to their x-coordinates without any overlaps and by preserving the relative position of the rectangles in order to minimize the width of the drawing area. The running time of this method is $O(|C| \log^2 |C|)$.

In contrast to the previous sketched methods, in the *polyomino packing approach* Freivalds et al. [44] identify the drawing of each component with a special kind of polygons. In particular, they assume that the graph is laid out in a plane with an underlying rectangular grid and define a *polyomino* of a component of a graph as the set of all grid boxes that are covered or partially covered by the drawing of this component. The algorithm sorts the polyominos by non-increasing sizes and, then places the polyominos iteratively on a rectangular grid. In order to prevent overlaps, each grid box that is covered by a polyomino is marked as *occupied*, and the algorithm searches for a placement of each polyomino that covers only non-occupied grid boxes. Freivalds et al. [44] prove that the running time of the polyomino packing approach is $O(|C|^2)$ if a reasonable grid resolution is assumed.

A practical comparison of the strip-packing, tiling, and alternate bisection method shows "that the tiling method clearly produces the most compact drawings" (see [44] page 385). In practice, only the polyomino packing method uses fewer aspect-ratio area, but needs significantly longer running times [44].

3.2.3 The Divide-Et-Impera Strategy for Solving (GDGL)

The previously sketched methods are heuristics that are developed for solving the constrained disconnected-graph layout problem. These methods are useful in some applications, for example, when it is already known that the components are rooted trees, and the user wants these trees to be drawn oriented so that each parent has a larger y coordinate than its children. However, our graph-drawing algorithm is designed for drawing general graphs. Hence, it is reasonable to allow reflections and rotations of each component, in order to obtain more compact drawings. An example that demonstrates the use of rotation operations to reduce the aspect-ratio area of a drawing is given in Figure 3.3.

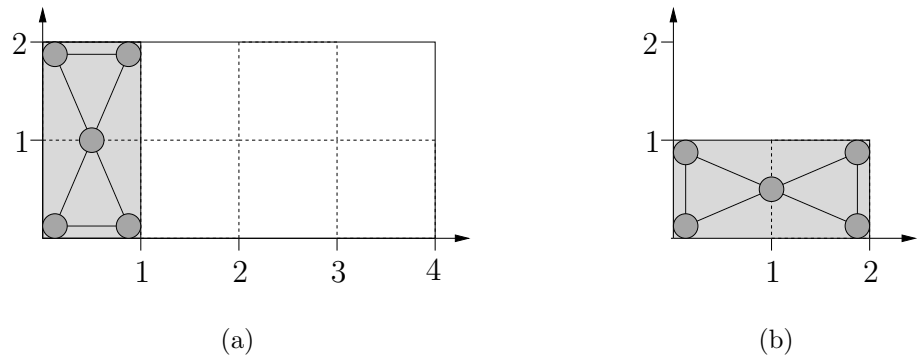


Figure 3.3: Two drawings of the same graph. Both drawings require the same drawing area 2. Choosing the desired aspect ratio $r = 2$, the drawing (a) needs $4 \cdot 2 = 8$ aspect-ratio area, while the drawing (b) that is obtained by translating and rotating drawing (a) with angle $\frac{\pi}{2}$ needs only $2 \cdot 1 = 2$ aspect-ratio area. This is optimal for this graph and the desired aspect ratio $r = 2$.

In the following, we will describe a heuristic for solving the general disconnected-graph layout problem (GDGL).

The Divide Step

Recall that in the postprocessing step we have generated a positive-weighted undirected simple graph $G' = (V', E', l^{zero})$ that is associated with the original graph G. Furthermore, a desired aspect ratio r is given by the user. In the *divide step* the set of components C of G' are computed. This can be done by using depth-first search. For later use, we denote the associated maximal connected subgraphs by $G'_1, \ldots, G'_{|C|}$.

The drawings $\Gamma(G'_1), \ldots, \Gamma(G'_{|C|})$ of $G'_1, \ldots, G'_{|C|}$ will be generated in the multilevel step and the postprocessing step that are introduced in the following chapters. Hence, we can assume that these drawings are given in the impera step that succeeds these steps.

The Impera Step

We now describe the algorithm that is designed to obtain a drawing $\Gamma(G)$ with non-overlapping components that uses few aspect-ratio area according to the desired aspect

ratio r. It is based on combining the tiling method of Dogrusoz [32] with suitable rotation operations and works in six phases.

Phase 1: Importation Phase

In the importation phase for each component $G'_i = (V'_i, E'_i, l^{zero})$ of $G' = (V', E', l^{zero})$ the drawing $\Gamma(G'_i)$ is used to obtain a drawing of the corresponding maximal connected subgraphs $G_i = (V_i, E_i)$ of the original graph $G = (V, E)$. This is done by simply assigning each node $v \in V_i$ the coordinate of the corresponding node $v' \in V'_i$. Note that in contrast to G' the nodes of G may have predefined non-zero sizes. In order to obtain fast running times in the following rotation phase, we identify the drawing of each node v by the smallest circle that surrounds the graphics of v. These circles have already been calculated in the preprocessing step (see Figure 3.1).

Phase 2: Rotation Phase

In the rotation phase, if G is connected, the drawing of G is rotated so that it uses few aspect-ratio area (see Figure 3.3). If G is disconnected, the drawings of each G_i are rotated in order to minimize the drawing area of each component. The idea is that one could expect to obtain more compact drawings if each components needs a minimum amount of drawing area. The following lemma is useful to speed up the calculation of the rotation phase.

Lemma 3.3. *Given a drawing $\Gamma(G)$ of a connected graph G and an arbitrary fixed angle α. In order to transfer $\Gamma(G)$ into a drawing $\Gamma_{\min}(G)$ — by allowing reflections, translations, and rotations of $\Gamma(G)$ — that uses a minimum amount of drawing area (or aspect-ratio area), rotations of $\Gamma(G)$ centered at the origin in the range $\left[\alpha, \alpha + \frac{\pi}{2}\right)$ (or $[\alpha, \alpha + \pi)$, respectively) are necessary and sufficient.*

Proof. It is obvious that translations, reflections on the x-axis, and reflections on the y-axes do neither change the area nor the aspect-ratio area of the drawing. Hence, it is sufficient to concentrate on rotations.

It is clear that rotating a point $P = (x, y)$ with the angle π and $\frac{\pi}{2}$ centered at the origin results in points $P' = (-x, -y)$ and $P'' = (-y, x)$, respectively. Hence, rotations with angle π do neither change the drawing area nor the aspect-ratio area of $\Gamma(G)$, while rotations with angle $\frac{\pi}{2}$ do not change the drawing area of $\Gamma(G)$ but possibly the aspect-ratio area. Thus, it suffices to rotate $\Gamma(G)$ in the range $\left[\alpha, \alpha + \frac{\pi}{2}\right)$ or $[\alpha, \alpha + \pi)$ to obtain a drawing that uses a minimum amount of drawing area or aspect-ratio area, respectively. It is not possible to obtain drawings that use a minimum amount of drawing area or aspect-ratio area by rotations with angles in smaller ranges in general, as can be seen in Figure 3.4: The angle α is set to 0 without loss of generality. The initial drawing $\Gamma(G)$ of the given graph G is shown in the right down quadrants of Figures 3.4(a) and 3.4(b). G is *nearly* drawn axis parallel to the y-axis. The minimum drawing area of $\Gamma(G)$ in Figure 3.4(a) is 5 and is obtained by rotating the drawing by angle $\frac{\pi}{2} - \epsilon$. The minimum aspect-ratio area according to the desired aspect ratio $\frac{1}{5}$ is also 5 and is obtained by rotating $\Gamma(G)$ by angle $\pi - \epsilon$ (Figure 3.4(b)). □

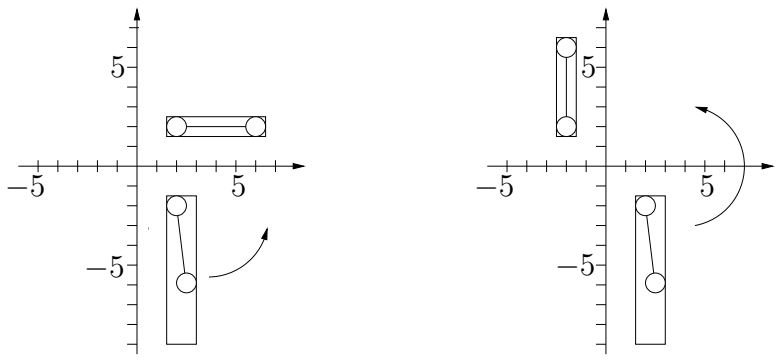


Figure 3.4: (a) An example in which a rotation with angle $\frac{\pi}{2} - \epsilon$ is necessary to obtain a drawing with minimum drawing area. (b) An example in which a rotation with angle $\pi - \epsilon$ is necessary to obtain a drawing with minimum aspect ratio area according to the desired aspect ratio $\frac{1}{5}$.

Using Lemma 3.3, we calculate a constant number of rotations of each component in the range $\left[0, \frac{\pi}{2}\right)$ or $[0, \pi)$ if G is disconnected or connected, respectively. Then, the placement that results in the least drawing area or aspect-ratio area, respectively, is stored. In the following, we associate the drawings of the components $\{G_1, \ldots, G_{|C|}\}$ of G with their bounding rectangles denoted by $\{R_1 \ldots, R_{|C|}\}$. If the given graph G is connected, we proceed with phase 6, otherwise we proceed with phases $3, 4, 5$, and 6.

Phase 3: Alignment Phase

First, in order to avoid that the drawings of two components are drawn too close to each other, an offset is added to the width and height of the bounding rectangles. Then, each rectangle R_i with breadth b_i and height h_i is aligned in the following way: If the desired aspect ratio r and $\frac{b_i}{h_i}$ are both at least one or both smaller than one, nothing is done. In the complementary case R_i is tipped over by angle $\frac{\pi}{2}$, which is realized by simply exchanging b_i and h_i. After this phase all rectangles R_i are aligned so that $\frac{b_i}{h_i}$ is at least one (if $r \geq 1$) or less than one (if $r < 1$).

Phase 4: Sorting Phase

Now the set of all rectangles is sorted by non-increasing height.

Phase 5: Packing Phase

This phase is a variation of the strip-packing method of Dogrusoz [32] in which we additionally (in some cases) allow each rectangle to be tipped over:

The rectangles are placed orthogonally and left justified into rows in the order that has been created in the sorting phase. We define the *height of a row* as the maximum height

among all rectangles that have been inserted into that row. The *width of a row* is the sum of the width of all rectangles that are placed in that row. Note that — since the rectangles are sorted by non-increasing height — the height of a row is the height of the first rectangle that has been inserted into that row.

The algorithm starts by inserting rectangle R_1 into the first row. Three alternatives are allowed for placing a rectangle $R_i, i \geq 2$: First, it could be placed left justified into a new row. Second, it could be placed left justified into an existing row with minimum width. Third, in contrast to the method of Dogrusoz [32], R_i could be tipped over by angle $\frac{\pi}{2}$ before placing it left justified into an existing row with minimum width. The last alternative is only allowed as long as the tipping operation does not increase the height of this row. For each R_i, we choose the placement alternative that results in the minimum aspect-ratio area of the actual packing among the three placement alternatives. Figure 3.5 demonstrates this decision process at an example.

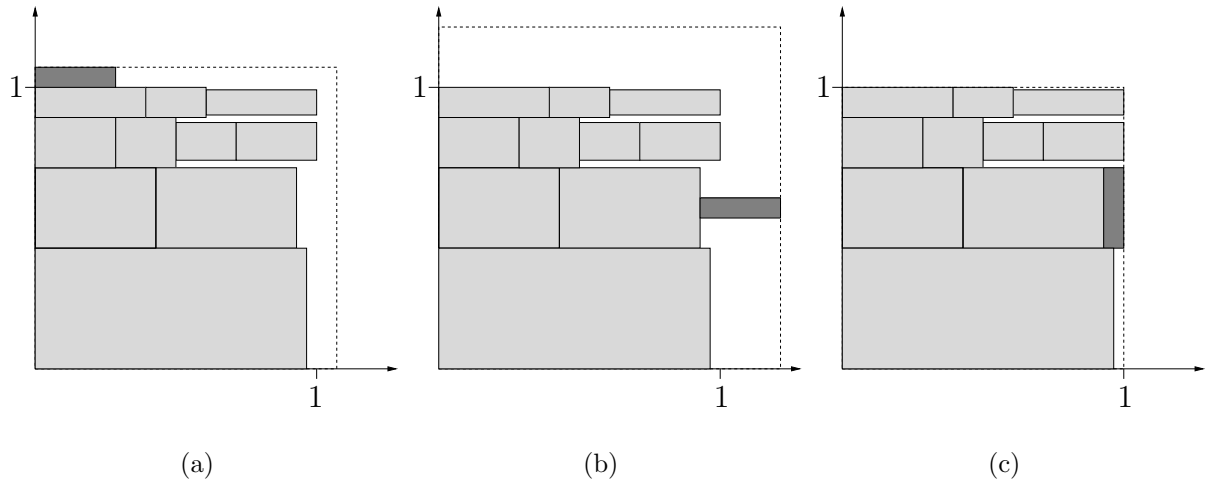


Figure 3.5: Possible placement positions when packing the dark rectangle R into an existing packing. The desired aspect ratio is one. The dashed lines indicate the aspect ratio area of the different packing alternatives. (a) Packing R into a new row. (b) Packing R into the row with minimum width. (c) Packing R into the row with minimum width but previously tipping R over. The minimum aspect-ratio area is obtained by packing alternative (c).

Phase 6: Placement Phase

After the packing phase, an overlapping-free placement of the rectangles R_i is determined. Since each R_i is dedicated to the drawing of a maximal connected subgraph G_i of G and the placement of each node v of G_i has been stored in the rotation phase, it is trivial to calculate the final position of each node v of G. Note that for these calculations we also have to store whether a rectangle R_i has been tipped over in the alignment phase or packing phase. Figure 3.6 shows a disconnected graph that has been drawn using the divide-et-impera strategy.

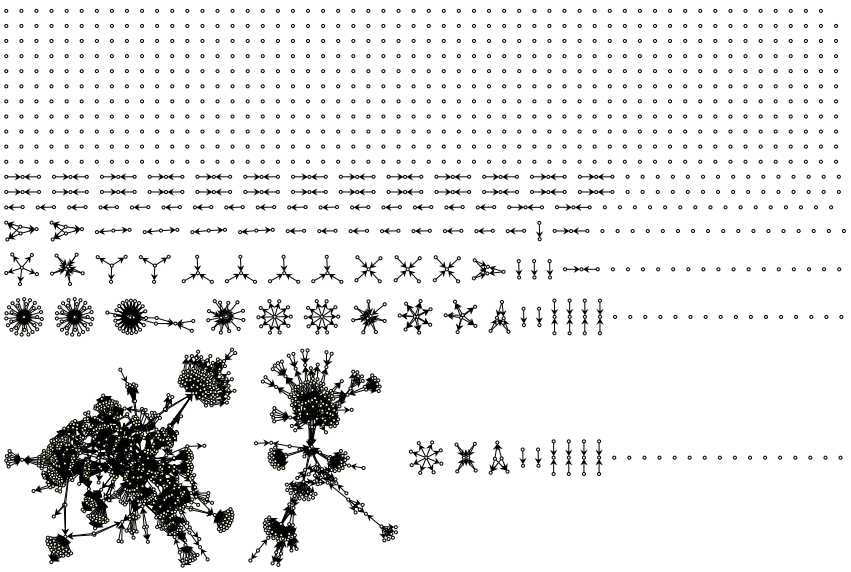


Figure 3.6: A disconnected graph with 829 components that is taken from [6] and drawn with FM^3. The desired aspect ratio is 1.5.

3.2.4 Formal Description of the Divide-Et-Impera Strategy

Function `Divide_Et_Impera_Strategy` shows the pseudocode of the divide-et-impera strategy. Experimental results to the divide-et-impera strategy can be found in Section 7.2

Theorem 3.4 (Divide-Et-Impera Strategy). *Suppose that $G = (V, E)$ is a general graph that may contain nodes of different fixed sizes and shapes, and $G' = (V', E', l^{zero})$ is the positive-weighted undirected simple graph that is obtained from G in the preprocessing step. Suppose that the set of components of G' is C, the maximal connected subgraphs of G' are $G'_1, \ldots, G'_{|C|}$, the aspect ratio of the desired drawing area is r, $\mathtt{A_{draw}}$ is the algorithm that is used to generate a drawing of all $G'_i = (V'_i, E'_i, l^{zero})$, and $\mathtt{A_{draw}}$ needs $t_{draw}(|V'_i|, |E'_i|)$ time and $m_{draw}(|V'_i|, |E|'_i)$ memory to generate a drawing of graph G'_i. Then, Function `Divide_Et_Impera_Strategy` generates a drawing $\Gamma(G)$ of G with non-overlapping components in $O(|V| + |E| + |C| \log |C| + \sum_{i=1}^{|C|} t_{draw}(|V'_i|, |E'_i|))$ time using $O(|V| + |E| + |C| + \sum_{i=1}^{|C|} m_{draw}(|V'_i|, |E'_i|))$ memory.*

Proof. In the divide step, the components of $G' = (V', E', l^{zero})$ can be found in $O(|V'| + |E'|) = O(|V| + |E|)$ time using depth-first search, and $O(|V| + |E|)$ memory is needed to store $\{G'_1, \ldots, G'_{|C|}\}$.

The importation phase of the impera step needs $O(|V| + |E|)$ time. In the rotation phase only a constant number k of rotations of the maximal connected subgraphs G_i of G are performed. Hence, this phase can be done in $O(\sum_{i=1}^{|C|}(k \cdot |V_i|)) = O(|V|)$ time. The memory requirements of the rotation phase are $O(\sum_{i=1}^{|C|} |V_i|) = O(|V|)$. The alignment

phase needs $O(|C|)$ time. The sorting phase can be realized in $O(|C| \log |C|)$ time using merge sort. The packing phase can be realizes in $O(|C| \log |C|)$ time by using a heap data structure [100] for finding and updating the row with the minimum width. Note that deciding which of the three packing alternatives for a rectangle R_i has to be used can be done in constant time by storing and updating the width and height of the bounding rectangle of the actual packing in combination with Formula (1.1) in Section 1.1.4. The memory requirements for storing the breadth and height of a row are bounded by $O(|C|)$. Finally, the placement phase needs $O(|V| + |E| + |C|)$ time.

In summary, the total running time of Function `Divide_Et_Impera_Strategy` is $O(|V| + |E| + |C| \log |C| + \sum_{i=1}^{|C|} t_{draw}(G'_i))$ and the memory requirements are $O(|V| + |E| + |C| + \sum_{i=1}^{|C|} m_{draw}(|V'_i|, |E'_i|))$. □

Function `Divide_Et_Impera_Strategy`(G, G', r)

input : a positive-weighted undirected simple graph $G' = (V', E', l^{zero})$ that is associated with the original graph $G = (V, E)$ and a desired aspect ratio r of the drawing area

output: a drawing $\Gamma(G)$ of G with non-overlapping components; except self-loops (that are drawn as loops), all edges are drawn as straight lines; multiple edges are drawn parallel; directed edges are drawn as arcs

```
begin
    let C denote the set of components of G';
    begin{Divide_Step}
        Find the maximal connected subgraphs G'_1, ..., G'_|C| of G';
    end
    foreach G'_i ∈ {G'_1, ..., G'_|C|} do
        Γ(G'_i) ← A_draw(G'_i);
    begin{Impera_Step}
        let {G_1, ..., G_|C|} denote the maximal connected subgraphs of G;
        for i = 1 to |C| do Γ(G_i) ← Import(Γ(G'_i));
        for i = 1 to |C| do Γ(G_i) ← Rotations(Γ(G_i));
        if |C| > 1 then
            let {R_1, ..., R_|C|} be the bounding rectangles of {Γ(G_1), ..., Γ(G_|C|)};
            for i = 1 to |C| do calculate R_i and R_i ← Align(R_i);
            {R_1, ..., R_|C|} ← Sort({R_1, ..., R_|C|});
            P ← Pack({R_1, ..., R_|C|});
        Γ(G) ← Place(P, {Γ(G_1), ..., Γ(G_|C|)})
    end
end
```

Chapter 4

The Multilevel Step

> Je höher du wirst aufwärtsgehen,
> dein Blick wird immer allgemeiner.
> Stets einen größeren Teil wirst du vom Ganzen sehn,
> doch alles Einzelne immer kleiner. [1]

In this chapter we will present a new multilevel strategy for drawing connected positive-weighted undirected simple graphs. After some remarks that will motivate the development of the new multilevel strategy in Section 4.1, we will describe its coarsening phase and its refinement phase in Section 4.2 and Section 4.3, respectively. Finally, we will give a formal description of the whole multilevel step in Section 4.4.

4.1 Motivation and Goals

4.1.1 Qualities of Multilevel Strategies

We have already seen in Section 1.3.2 that the existing multilevel methods [49, 50, 64, 132, 133] share the following basic ideas: Given a graph $G = (V, E)$ they create a series of graphs $G =: G_0, G_1, \ldots, G_k$ with decreasing sizes (the coarsening phase). Then, the smallest graph G_k at level k is drawn using (a variation of) a classical force-directed single-level algorithm that we denote by $\texttt{A}_{\texttt{single}}$. This drawing is used to obtain an initial layout of the next larger graph G_{k-1} that is drawn afterwards using $\texttt{A}_{\texttt{single}}$. This process (the refinement phase) is repeated until the original graph G_0 is drawn.

Multilevel methods have been used successfully to speed up force-directed algorithms in practice. In summary, the fast running times of these methods — in comparison with single-level algorithms — is caused by the fact that the initial placements of the multilevel graphs G_i (in particular that of the original graph $G_0 = G$) are frequently close to placements that induce local energy minima. Hence, only few iterations have to be calculated in $\texttt{A}_{\texttt{single}}$.

[1] Friedrich Rückert

This statement is confirmed by experimental results (see Section 7.3) with our multilevel strategy that will be developed in this chapter.

Another feature of multilevel strategies is that the multilevel graphs are coarse representations of the original graph and can be used to visualize the structure of the graph at several levels of abstraction (see also Section 1.3.4).

4.1.2 Problems of Current Multilevel Strategies

However, the multilevel strategies [49, 50, 64, 132, 133] have some disadvantages regarding the asymptotic worst-case running time, which is discussed in the following.

Suppose that $\texttt{A}_{\texttt{single}}$ needs time $t_{single}(|V|, |E|)$ to generate a drawing of a graph $G = (V, E)$. Then, it is clear that the total running time of any multilevel strategy — which uses $\texttt{A}_{\texttt{single}}$ for drawing the graphs at the multilevels — is bounded below by $\Omega(t_{single}(|V|, |E|))$. Hence, one cannot expect that a multilevel method has a better worst-case running time as the used single-level algorithm. But what is the worst-case running time of the existing multilevel methods [49, 50, 64, 132, 133] in relation to the running time of the used force-directed single-level algorithm?

The coarsening phase of the multilevel method of Walshaw [132, 133] works by successively finding maximal matchings and shrinking these edges to nodes. If, for example, $G = (V, E)$ is a star graph, only one edge is shrunk at each multilevel. Hence, the number of nodes and edges of the multilevel graphs $G_0 = (V_0, E_0), \ldots, G_k = (V_k, E_k)$ are $|V_0| := |V|, |V_1| := |V| - 1, \ldots, |V_k| := 1$, and $|E_0| := |E|, |E_1| := |E| - 1, \ldots, |E_k| := 0$. We know from the analysis in Section 1.3.2 that the worst-case running time of the used force-directed single-level algorithm is $O(|V|^2 + |E|)$. Thus, the worst-case running time of Walshaw's method [132, 133] is $\sum_{i=0}^{|V|-1} \texttt{A}_{\texttt{single}}(|V_i|, |E_i|) = O(|V|(|V|^2 + |E|)) = O(|V| \cdot \texttt{A}_{\texttt{single}}(V, E))$. Suppose now one could improve $\texttt{A}_{\texttt{single}}$ so that it needs only linear time. Then the multilevel strategy of Walshaw [132, 133] would need $\sum_{i=0}^{|V|-1} \texttt{A}_{\texttt{single}}(|V_i|, |E_i|) = O(|V|(|V| + |E|) = O(|V| \cdot \texttt{A}_{\texttt{single}}(|V| + |E|)$ time. Hence, in both cases the worst-case running time of the multilevel strategy is a factor $|V|$ slower than that of the used single-level algorithm.

The construction of the MIS-filtration that is used in the coarsening phase of the method `GRIP` by Gajer and Kobourov [50] and Gajer et al. [49] has not been analyzed in terms of the size of the input graph. Hence, the asymptotic running time of the multilevel strategy is unknown.

The multilevel strategy of Harel and Koren [64] is based on a $\Theta(|V|^2)$ preprocessing step which makes further analysis of its running time marginal.

4.1.3 A New Multilevel Strategy

As the previous discussion has shown, the best we could do is to develop a multilevel strategy that provably has the same asymptotic running time as the single-level algorithm $\texttt{A}_{\texttt{single}}$ that is used to draw all multilevel graphs G_i with $i = 0, \ldots, k$.

Since each multilevel graph G_i should be closely related to the next smaller and next larger graph G_{i-1} and G_{i+1}, respectively, all multilevel methods [49, 50, 64, 132, 133] exploit

the structure of the given graph (by calculating a MIS-filtration, approximating k-centers, and finding maximal matchings, respectively) to reach this goal. All these strategies are based on the implicit assumption that the used force-directed single-level algorithm tends to preserve the desired edge length. And, of course, this assumption is reasonable, because this is one of the most important goals of force-directed methods.

Our multilevel strategy is also based on this assumption, but is designed to require only the same asymptotic running time as the used force-directed single-level algorithm. The basic idea of our multilevel strategy is as follows: In the coarsening phase we will construct a series of connected positive-weighted undirected simple graphs $G =: G_0, \ldots, G_k$ with decreasing sizes by partitioning the node set of each graph G_i so that the induced subgraphs are disjoint, connected, and have small diameter. These subgraphs are shrunk to nodes in order to obtain the next smaller graph G_{i+1}. Thereby, reasonable values of the edge weights of the multilevel graphs G_{i+1} are calculated. Furthermore, useful information is generated that is needed to obtain good initial placements in the refinement phase.

4.2 The Coarsening Phase

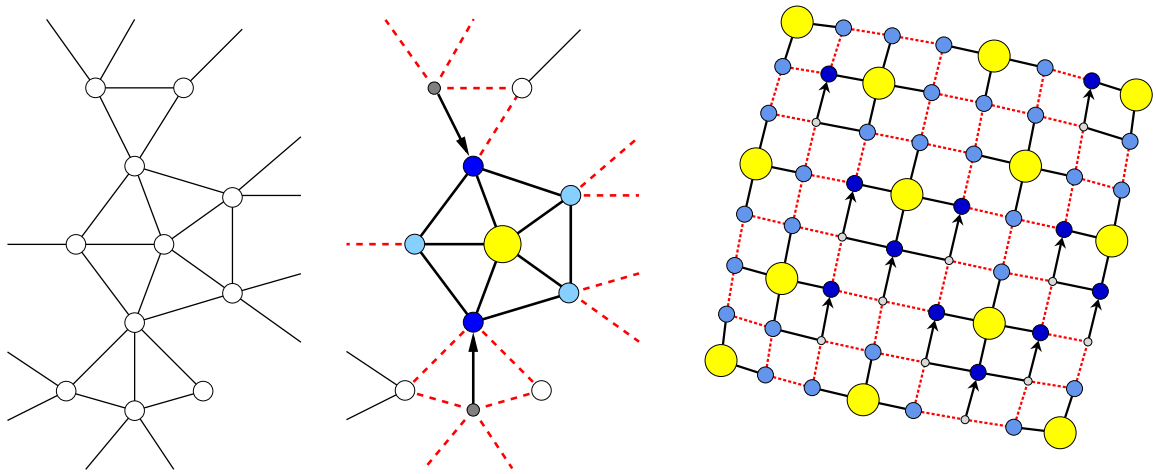
As a result of the divide-et-impera step we can suppose that the given graph $G = (V, E, l^{zero})$ is a connected positive-weighted undirected simple graph. In the remainder of this chapter (according to the notation in Section 1.1.2) we denote the distances in the underlying unweighted graph that corresponds to G by the *graph-theoretic distances* in G. We first explain how G can be partitioned into disjoint small subgraphs that we will denote by *solar systems*, before describing the use of this partitioning for constructing the multilevel graphs.

4.2.1 Constructing Galaxies

Definition 4.1 (Galaxies, Solar Systems, and Their Elements). *Suppose that $G = (V, E, l^{\text{zero}})$ is a connected positive-weighted undirected simple graph, and $U \subseteq V$. The vertex-induced subgraph $S := G[U]$ is called* solar system *if the following conditions hold: Exactly one node in U is marked as* sun node *(or s-*node*). Each of its neighbors is marked as* planet node *(p-*node*) or as* planet-with-moon node *(pm-*node*) and is also contained in U. The other nodes in U are marked as* moon nodes *(or m-*nodes*), each m-node is required to have graph-theoretic distance two to the s-node in U, and each m-node is assigned to exactly one pm-node in U. Furthermore, for each pm-node exists at least one m-node that is assigned to it.*

A galaxy *is a partitioning of the node set V of a connected positive-weighted undirected simple graph $G = (V, E, l^{zero})$ into disjoint subsets $U_1, \ldots, U_l$ so that the vertex-induced subgraphs $S_i := G[U_i]$ for $i \in \{1, \ldots, l\}$ are solar systems. All edges $e = (v_i, v_j)$ with $v_i, v_j \in U_i$ are called* intra solar-system edges*, while all edges $e = (v_i, v_j)$ with $v_i \in U_i$, and $v_j \in U_j$ and $i \neq j$ are called* inter solar-system edges*.*

In the remainder of this chapter we will visualize these terms for a better understanding as follows: The s-nodes are drawn as big yellow disks, p-nodes as medium sized blue disks, pm-nodes as medium sized dark-blue disks, and m-nodes as small grey disks. The intra solar-system edges are drawn as black solid edges, while the inter solar-system edges are drawn as red dotted edges. An intra solar system edge that connects an m-node with its associated pm-node is drawn as a directed edge indicating that this m-node is associated with this pm-node. Figure 4.1 demonstrates these terms at an example. Figure 4.1(b) shows a solar system of the undirected simple graph of Figure 4.1(a). Figure 4.1(c) shows a galaxy partitioning of a connected undirected simple graph.



(a) (b) (c)

Figure 4.1: (a) A part of a graph G and (b) a solar system S of G, induced by one s-node, three p-nodes, two pm-nodes, and two m-nodes. (a) A galaxy consisting of 13 solar systems.

Lemma 4.2 (Galaxy Partitioning (1)). *A galaxy partitioning of a connected positive-weighted undirected simple graph $G = (V, E, l^{zero})$ can be constructed in $O(|V| + |E|)$ time using $O(|V|)$ memory, and the graph-theoretic distance between any two nodes of a solar system is at most four.*

Proof. The statement on the graph-theoretic distance between two nodes in a solar system follows directly from the definition of solar systems. The other part of the statement will be shown by describing a construction algorithm:

First, the algorithm creates the sun nodes. Therefore, a candidate set V' that is a copy of V is created, and a first sun node s_1 is selected from V' randomly with uniform probability. An alternative technique for selecting the sun nodes from V' is described in Section 4.2.2. Then, s_1 and all nodes that have graph-theoretic distance at most two to s_1 in G are deleted from V'. Afterwards, the algorithm iteratively selects the next sun nodes in the same way, until V' is empty, and $Suns = s_1, \ldots, s_l \subset V$ is the list of all sun nodes. Second, for each $s_i \in Suns$ all its neighbors in V are labeled as planet nodes. Finally, there

might exist a subset $M \subset V$ that consists of nodes, which are neither labeled as planet nodes nor as sun nodes. Each node $m \in M$ is neighbor of at least one node $v \in V$ that has been marked as planet node. (It is clear that m is no neighbor of a sun node since otherwise it would have been marked as planet node. If m would have only unmarked neighbors, the construction procedure of the sun nodes and planet nodes would imply that every sun node would have graph-theoretic distance at least 3 to m. Hence, m or one of its neighbors would have been marked as sun node, which is a contradiction.) Thus, the nodes in M can be marked as moon nodes. Each moon node is assigned to the planet node that is its nearest neighbor in G, and this planet node is relabeled as *pm*-node. It is clear that this algorithm needs $O(|V| + |E|)$ time using $O(|V|)$ memory for storing the candidate set V'. □

4.2.2 Collapsing of Solar Systems

We now describe how the series $G =: G_0, \ldots, G_k$ of connected positive-weighted undirected simple multilevel graphs is constructed by using galaxy partitionings. Given a galaxy partitioning of $G_i = (V_i, E_i, l_i^{zero})$, we construct a smaller graph $G_{i+1} = (V_{i+1}, E_{i+1}, l_{i+1}^{zero})$ by collapsing (shrinking) the node set of each solar system into one single node and deleting multiple edges. It remains to be described how the edge weights of G_{i+1} are created. This is done by measuring the length of special paths in G_i that connect the sun nodes of two distinct solar systems. It is important to note that (as a product of the preprocessing step) the given edge weights of $G = G_0$ are the zero-energy length of the springs in our force model which are defined to reflect the desired edge length. Hence, the new edge weights should reflect the desired edge length, too.

Definition 4.3 (Inter Solar-System Path, Ancestor, Descendant). *Suppose that $e = (v, w)$ is an inter solar-system edge in a connected positive-weighted undirected simple graph $G = (V, E, l^{zero})$ and that v belongs to the solar system of sun node s, and w belongs to the solar system of sun node t of G. The* inter solar-system path *P_e that corresponds to edge e is defined as follows: If both v and w are planet nodes or pm-nodes, then $P_e := (s, v, w, t)$. If v is a moon node with dedicated pm-node u and w is a planet or pm-node, then $P_e := (s, u, v, w, t)$. If v is a planet or pm-node and w is a moon node with dedicated pm-node x, then $P_e := (s, v, w, x, t)$. Finally, if v is a moon node with dedicated pm-node u and w is a moon node with dedicated pm-node x, then $P_e := (s, u, v, w, x, t)$.*

Suppose that a node s_{i+1} of G_{i+1} is obtained by collapsing a solar system S_i of G_i. Then, s_{i+1} is called the descendant *of each node that is contained in S_i, while each node that is contained in S_i is called an* ancestor *of s_{i+1}.*

We initialize the edge weight of an edge $e_{i+1} = (s_{i+1}, t_{i+1})$ of G_{i+1} as follows: Suppose that S_i and T_i are solar systems of G_i and that s_{i+1} and t_{i+1} are obtained by shrinking S_i and T_i, respectively, to nodes. Three cases can arise:

If S_i and T_i are not connected by an inter solar-system edge in G_i directly, s_{i+1} and t_{i+1} also will not be connected by an edge in G_{i+1}, and, hence, no edge weight has to be calculated. If S_i and T_i are connected by exactly one inter solar-system edge e_i in G_i,

then s_{i+1} and t_{i+1} will be connected by an edge $e_{i+1} = (s_{i+1}, t_{i+1})$ in G_{i+1}, and we set $l_{i+1}^{zero}(e_{i+1}) := l_i^{zero}(P_{e_i})$. If S_i and T_i are connected by more than one inter solar-system edge in G_i (namely by $\{e_i^1, \ldots, e_i^k\}$), then s_{i+1} and t_{i+1} will be connected by an edge $e_{i+1} = (s_{i+1}, t_{i+1})$ in G_{i+1}, and we set $l_{i+1}^{zero}(e_{i+1}) := (l_i^{zero}(P_{e_i^1}) + \ldots + l_i^{zero}(P_{e_i^k}))/k$.

Figure 4.2 demonstrates the initialization of the edge weights at an example: Only one inter solar-system edge $e_i = (u_i, v_i)$ connects the two solar systems of G_i in Figure 4.2(a). The corresponding inter solar-system path is $P_{e_i} = (s_i, u_i, v_i, t_i)$ and has length 4. Suppose that s_{i+1} and t_{i+1} are nodes of G_{i+1} that are obtained by shrinking the two solar systems that are shown in Figure 4.2(a) to nodes. Then, s_{i+1} and t_{i+1} are the descendants of the nodes of these two solar systems and the edge weight of the edge $e_{i+1} = (s_{i+1}, t_{i+1})$ is set to $l_i^{zero}(P_{e_i}) = 4$ (Figure 4.2(b)). In Figure 4.2(c) three inter solar-system edges ($e_i^1 = (u_i, v_i)$, $e_i^2(x_i, v_i)$, and $e_i^3 = (x_i, y_i)$) connect the two shown solar systems of G_i. The corresponding paths are $P_{e_i^1} = (s_i, u_i, v_i, t_i)$, $P_{e_i^2} = (s_i, w_i, x_i, v_i, t_i)$, and $P_{e_i^3} = (s_i, w_i, x_i, y_i, t_i)$, and their length are 4, 6, and 5, respectively. Hence, the weight of the edge $e_{i+1} = (s_{i+1}, t_{i+1})$ of G_{i+1} is set to $\frac{4+6+5}{3} = 5$ (Figure 4.2(d)).

The Stopping Criterion

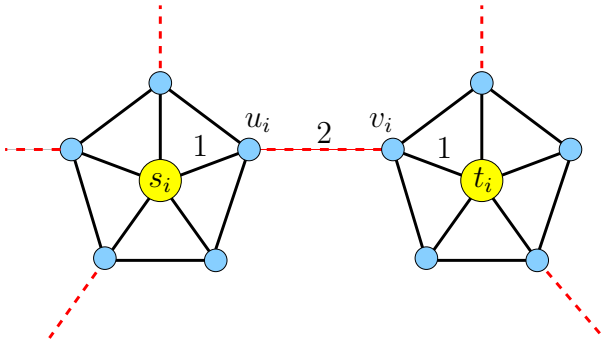
The partitioning and collapsing process can be iterated until the actual graph G_i contains less than a constant number c of nodes. However, motivated by arguments that will be explained in Section 4.3, the partitioning and collapsing process is stopped, whenever either the actual graph G_i contains less than a constant number c of nodes or if the following statement holds:

Given a positive integer constant d and a rational constant s with $1 < s \leq 2$. There exist more than d multilevel graphs $G_j \in \{G_0, \ldots, G_{i-1}\}$ so that $|E_{j+1}| > |E_j|/s$.

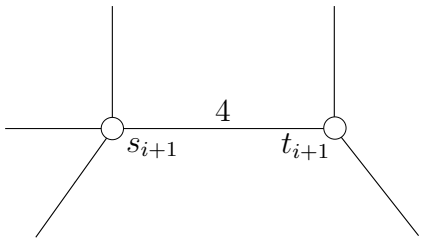
Improving the Partitioning Process

We concentrate on the partitioning process again. In the proof of Lemma 4.2, we have described an algorithm that creates a galaxy partitioning by choosing the set of sun nodes from the candidate set V' randomly with uniform probability. This process will work in any case, however, practical experiments have shown that the quality of the generated drawings could be improved by introducing a different strategy for selecting the sun nodes from the candidate set V'. The goal of the new strategy is to select the sun nodes from V' so that the coarsening process does not need too few multilevels. We need the following definitions:

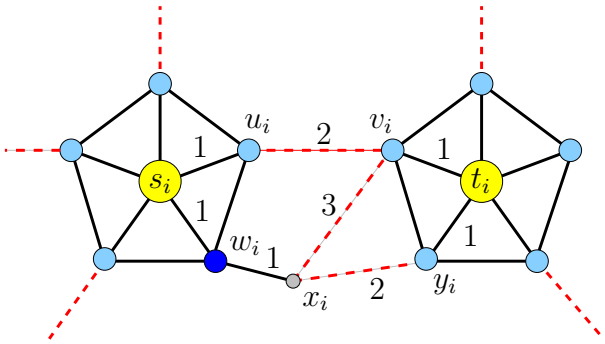
Definition 4.4 (Mass, Star Mass). *Suppose, $G_0, \ldots, G_k$ is a series of multilevel graphs that is created by successively collapsing solar systems, like described above. The* mass *of a node v_0 of the multilevel graph G_0 is set to one. For $i \in \{0, \ldots, k-1\}$ The* mass *of a node v_{i+1} of the multilevel graph G_{i+1} is defined as the sum of the masses of its ancestors in G_i. We define the* star mass *of a node v_i in the multilevel graph G_i as the sum of the masses of v_i and its neighbors.*



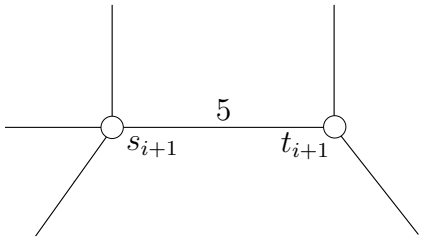
(a) A part of G_i



(b) The corresponding part of G_{i+1}



(c) A part of G_i



(d) The corresponding part of G_{i+1}

Figure 4.2: Two examples that visualize the calculation of the edge weight of an edge of G_{i+1} using the length of inter solar-system paths of G_i.

For example, assuming that $i = 0$ in Figure 4.2, the masses of the nodes s_{i+1} and t_{i+1} in Figure 4.2(d) are 7 and 6, respectively. Under the same assumption the star masses of the nodes s_i and t_i in Figure 4.2(c) are 6 and 6, respectively. This example also shows that the star mass can serve as a coarse approximation of the mass of an entire solar system. Note that for each multilevel graph G_i the sum of the masses of all its nodes is equal to the number of nodes of $G = G_0$.

The new strategy for selecting the sun nodes (that we denote by `Select_By_Star_Mass`) is a greedy method that iteratively selects a sun node s from V' so that the solar system of s will probably have a comparatively low mass. Hence, one can hope that the coarsening process will not be too fast. `Select_By_Star_Mass` works as follows:

Let $G_i = (V_i, E_i, l_i^{zero})$, and the candidate set V' is a copy of V_i. First, the star mass of each node of G_i is calculated. Then, the algorithm randomly selects a constant number of nodes of V' and selects this node s of the sample as sun node that has the lowest star mass. After deleting s and all nodes that have graph-theoretic distance at most two to

s from V', the next random sample of constant size is chosen from V' and so forth. The process ends, when V' is empty.

Corollary 4.5 (Galaxy Partitioning (2)). *A galaxy partitioning of a connected positive-weighted undirected simple graph $G = (V, E, l^{zero})$ with a given mass of each node $v \in V$ can be constructed in $O(|V| + |E|)$ time with $O(|V|)$ memory requirements by using the strategy* `Select_By_Star_Mass` *for creating the sun nodes.*

Proof. Creating the star masses can be done $O(|E|)$ time. Since each random sample contains only a constant number of nodes, finding the node with the minimum star mass in this random sample can be done in constant time. Hence, all sun nodes can be found in linear time. The memory requirements for storing the masses and star masses are linear in $|V|$. The remainder of the proof follows from Lemma 4.2 and the fact that the other parts of the construction algorithm are identical to those of the algorithm described in the proof of Lemma 4.2. □

Generating useful Information for the Refinement Phase

In the refinement phase we have to generate an initial placement of the nodes of G_i using the placement of the nodes of G_{i+1}. In order to do this satisfactory, we will additionally store information about the relative positions of the p-, pm-, and m-nodes of G_i on the inter solar-system paths of G_i.

In particular, suppose a p-, pm-, or m-node v is contained in an inter solar-system path $P = (s, \ldots, v, \ldots, t)$ connecting the sun nodes s and t. If P has length p, and the distance from s to v in P is l, then we store the quadruple $(v, s, t, \frac{l}{p})$. This quadruple indicates that the length of the sub-path $(s, \ldots, v)$ of the inter solar-system path $P = (s, \ldots, v, \ldots, t)$ is exactly $\frac{l}{p}$ of the length of the path P. If v belongs to more than one inter solar-system path, such a quadruple is stored for each path that contains v. For later use we denote the list of a node v that contains these quadruples by the *relative-path-position list* of node v and the last entry of a quadruple as the *position factor* of this quadruple.

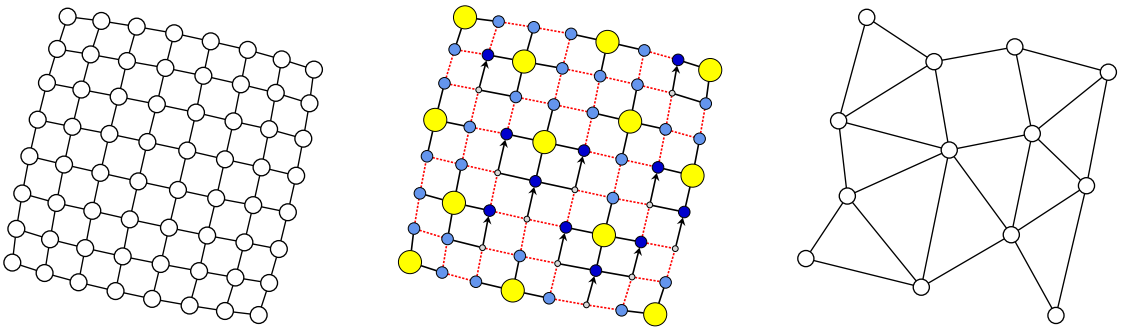
For example, the relative-path-position lists of the nodes u_i and v_i in Figure 4.2(a) are $\{(u_i, s_i, t_i, \frac{1}{4})\}$ and $\{(v_i, s_i, t_i, \frac{3}{4})\}$, respectively. The relative-path-position lists of the nodes w_i and x_i in Figure 4.2(c) are $\{(w_i, s_i, t_i, \frac{1}{6}), (w_i, s_i, t_i, \frac{1}{5})\}$ and $\{(x_i, s_i, t_i, \frac{2}{6}), (x_i, s_i, t_i, \frac{2}{5})\}$, respectively. Figure 4.3 exemplifies the coarsening phase.

Lemma 4.6 (The Coarsening Phase). *Suppose, $G_i = (V_i, E_i, l_i^{zero})$ is a connected positive-weighted undirected simple graph. Then, a connected positive-weighted undirected simple graph $G_{i+1} = (V_{i+1}, E_{i+1}, l_{i+1}^{zero})$ and for each node $v_i \in V_i$ a relative-path-position list can be constructed by using the previously described partitioning and collapsing methods in $O(|V_i|+|E_i|)$ time using $O(|V_i|+|E_i|)$ memory. Furthermore, if $|V_i| \geq 2$, then $|V_{i+1}| \leq \frac{|V_i|}{2}$.*

Proof. It follows from Corollary 4.5 that a galaxy partitioning of G_i can be constructed in $O(|V_i| + |E_i|)$ time using $O(|V_i|)$ memory. The collapsing of the solar systems and the deletion of the multiple edges from G_{i+1} in the collapsing phase can be done in linear

time, too. The latter can be realized by using the same techniques as described in the proof of Theorem 3.1. Since the graph-theoretic distance between any two nodes in a solar system is at most four (see Lemma 4.2), each inter solar-system path contains at most five edges. Since there exist at most $|E_i|$ inter solar-system edges, the number of inter solar-system paths is at most $|E_i|$, too. Hence, all nodes of the inter solar-system paths can be identified in $O(|E_i|)$ time, and the sum of the length of all relative-path-position lists is bounded above by $O(|E_i|)$. Each quadruple in a relative-path-position lists of a node can be generated in constant time. Thus, the construction of G_{i+1} and the construction of the relative-path-position lists can be done in $O(|V_i| + |E_i|)$ time. The memory requirements for storing these lists are $O(|E_i|)$.

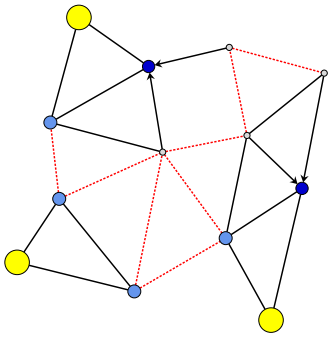
The second statement can be seen as follows: Every node of G_i belongs to a solar system in the galaxy partitioning of G_i. Furthermore, since G_i is connected — under the assumption that $|V_i| \geq 2$ — every solar system of G_i contains at least two nodes. Thus, by the description of the collapsing process $|V_{i+1}| \leq \frac{|V_i|}{2}$. □



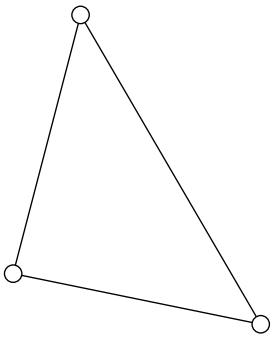
(a) $G =: G_0$

(b) A galaxy partitioning of G_0

(c) G_1



(d) A galaxy partitioning of G_1



(e) G_2

Figure 4.3: The coarsening phase on a grid graph with 64 nodes. The edge weights are hidden.

4.3 The Refinement Phase

Suppose, the series of multilevel graphs that was created in the coarsening phase is $G = G_0, \ldots, G_k$. Furthermore, suppose we have given a force-directed single-level algorithm $\mathtt{A_{single}}$ that will be used to generate a drawing of each multilevel graph $G_i = (V_i, E_i, l_i^{zero})$. Then, the refinement phase works as follows:

First, the smallest graph G_k is drawn with $\mathtt{A_{single}}$ starting with a random initial placement. Afterwards, the next larger graph G_{k-1} is drawn. This is done in two steps. First, a good initial placement of the nodes of G_{k-1} is created by using the placement of the nodes in the drawing of G_k and by using the information that is stored in the relative-path-position lists of the nodes of G_{k-1}. Second, $\mathtt{A_{single}}$ is used to generate a drawing of G_{k-1} starting with the generated initial placement. Then, the next larger graphs are successively drawn, until a drawing of the original graph $G = G_0$ is obtained.

Since the force-directed single-level algorithm $\mathtt{A_{single}}$ will be detailedly explained in Chapter 5, we only have to explain how the initial placements of the nodes of a multilevel graph G_i with $i \in \{0, \ldots, k-1\}$ are generated.

Suppose, we have given a drawing of $G_{i+1} = (V_{i+1}, E_{i+1}, l_{i+1}^{zero})$ and the relative-path-position list of each node of $G_i = (V_i, E_i, l_i^{zero})$. The initial placement of the nodes of G_i is obtained in two steps:

First, each sun node s_i of G_i is placed at the position of its ancestor s_{i+1} in the drawing of G_{i+1}.

Second, the other nodes of G_i (that are p-, pm-, or m-nodes) are placed. This is done by using the positions of the already placed sun nodes of G_i in combination with the relative-path-position lists. To be more precise, suppose, the next node that has to be placed is $v_i \in V_i$, its dedicated sun node is $s_i \in V_i$, and $pos(s_i)$ denotes the position of s_i. Three cases can arise: In the first case the relative-path-position list of v_i is empty. In this case v_i is placed randomly on a circle with radius $l_i^{zero}(s_i, v_i)$ with center $pos(s_i)$. In the second case the relative-path-position list of v_i contains exactly one quadruple $(v_i, s_i, t_i, \lambda_i)$. Then, v_i is placed on the line that connects the sun nodes s_i and t_i in the actual drawing of G_i. The exact position of v_i on this line is given by the following formula:

$$pos(v_i) := pos(s_i) + \lambda_i(pos(t_i) - pos(s_i)) \tag{4.1}$$

Finally, if the relative-path-position list of v_i contains more than one quadruple (namely $\{q_1, \ldots, q_l\}$), first for each quadruple q_j a position p_j is calculated according to Formula (4.1). Then, v_i is placed at the barycenter of these positions (namely $pos(v_i) := (p_1 + \ldots + p_l)/k$). Additionally, in all three cases a small random noise is added to the calculated positions of the p-, pm-, and m-nodes.

Figure 4.4 exemplifies the calculation of an initial placement of the nodes of a graph G_i. The sun nodes s_1, s_2, and s_3 of G_i in Figure 4.4(c) are placed at the positions of their ancestors (see Figure 4.4(a)). The relative-path-position list of node u in G_i is empty, and u has distance one (see Figure 4.4(b)) to its dedicated sun node s_1. Hence, u is placed at a random position with radius one centered at s_1 (see Figure 4.4(c)). Node v is

part of exactly one inter solar-system path of G_i (that connects s_1 and s_2). Its relative-path-position list contains one quadruple $(v, s_1, s_2, \frac{1}{3})$. Hence, v is placed on the line that connects s_1 and s_2 as shown in Figure 4.4(c). Node w is part of two inter solar-system paths of G_i (that connect the solar systems of s_1 with that of s_2 and s_3, respectively). Its relative-path-position list contains the two quadruples $(w, s_1, s_2, \frac{1}{4})$ and $(w, s_1, s_3, \frac{1}{3})$. The resulting barycenter position of w is shown in Figure 4.4(c). The other planet nodes of G_i are placed analogue. A complete example of the refinement phase is shown in Figure 4.5.

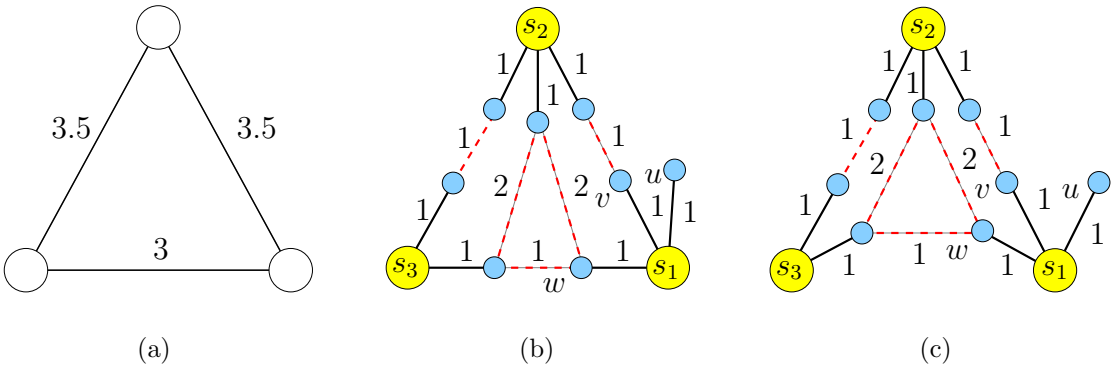


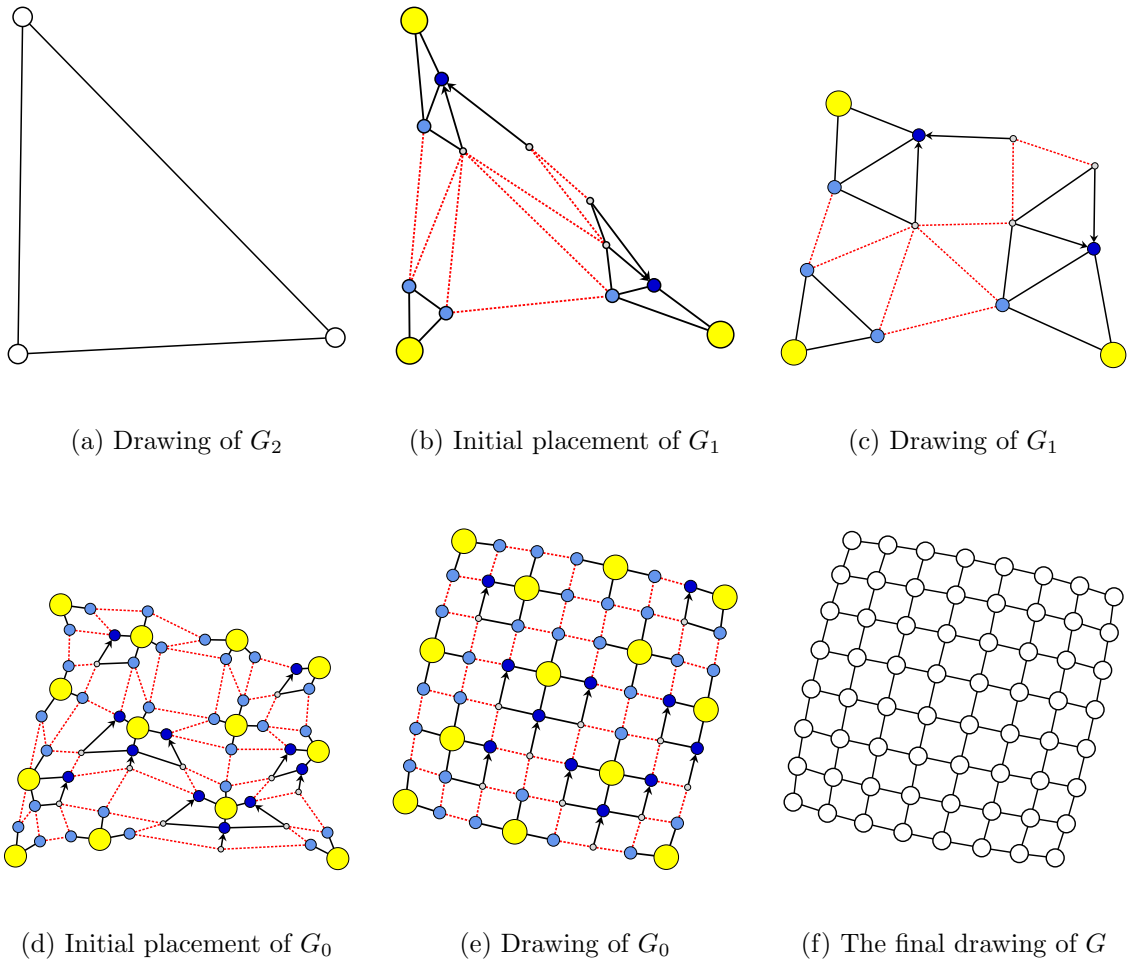
Figure 4.4: (a) A drawing of G_{i+1}. (b) The graph G_i. (c) The initial placement of the nodes of G_i.

Lemma 4.7 (Refinement Phase). *Given a series of connected positive-weighted undirected simple graphs $G_0, \ldots, G_k$ as described in the coarsening phase, an integer $i \in \{0, \ldots, k-1\}$, a drawing of G_{i+1}, for each node of $G_i = (V_i, E_i, l_i^{zero})$ a relative-path-position list, and a force-directed single-level algorithm $\mathtt{A}_{\mathtt{single}}$. Suppose that $\mathtt{A}_{\mathtt{single}}$ needs time $t_{single}(|V_i|, |E_i|)$ and memory $m_{single}(|V_i|, |E_i|)$ to generate a drawing of G_i starting with an arbitrary initial placement of the nodes of G_i. Then, the refinement phase needs time $O(|V_i| + |E_i|) + t_{single}(|V_i|, |E_i|)$ to generate a drawing of G_i and uses $m_{single}(|V_i|, |E_i|)$ memory.*

Proof. If $i = k$, a random initial placement of the nodes of G_i can be found in $O(|V_i|)$ time. If $i \neq k$, only constant time is needed to obtain an initial placement of each sun node of G_i. The time that is needed to obtain initial placements of the other nodes of G_i is proportional to the sum of the length of all relative-path-position lists of the nodes of G_i, which is bounded above by $O(|E_i|)$ (see proof of Lemma 4.6). Except the memory requirements of $\mathtt{A}_{\mathtt{single}}$, no additional memory is needed. □

4.4 Formal Description of the Multilevel Step

Now we have everything on hand to give a formal description of the multilevel step (see Function `Multilevel_Step`). An experimental study of the multilevel step will be presented in Section 7.3.



(a) Drawing of G_2 (b) Initial placement of G_1 (c) Drawing of G_1

(d) Initial placement of G_0 (e) Drawing of G_0 (f) The final drawing of G

Figure 4.5: The refinement phase on a grid graph with 64 nodes. The edge weights are hidden.

Theorem 4.8 (Multilevel Step). *Suppose that G is a connected positive-weighted undirected simple graph and* `A_single` *is a force-directed single-level algorithm that needs time $t_{single}(|V|, |E|)$ and memory $m_{single}(|V|, |E|)$ for drawing a graph $G = (V, E)$ starting with an arbitrary initial placement $p(V) := (p_v)_{v \in V}$ of the nodes of G. Furthermore, suppose that* `A_single` *is used for drawing the multilevel graphs in the refinement phase of the multilevel step. Then, Function* `Multilevel_Step` *generates a straight-line drawing $\Gamma(G)$ in $\Theta(t_{single}(|V|, |E|))$ time using $O(|V| + |E|) + m_{single}(|V|, |E|)$ memory.*

Proof. Suppose that the multilevel graphs that are generated in the multilevel step are $G =: G_0 = (V_0, E_0, l_0^{zero}), \ldots, G_k = (V_k, E_k, l_k^{zero})$. The total running time t_{mult} of the multilevel step is given by the following formula:

$$t_{mult}(|V|, |E|) = \sum_{i=0}^{k-1} t_{coarse}(|V_i|, |E_i|) + t_{single}(|V_k|, |E_k|) + \sum_{i=0}^{k-1} t_{refine}(|V_i|, |E_i|) \quad (4.2)$$

Here $t_{coarse}(|V_i|, |E_i|)$ denotes the time that is needed to create the multilevel graph G_{i+1} from G_i in the coarsening phase, and $t_{refine}(|V_i|, |E_i|))$ denotes the time that is needed to obtain a drawing of G_i in the refinement phase.

We know from Lemma 4.6 that $t_{coarse}(|V_i|, |E_i|) = O(|V_i|, |E_i|)$ and from Lemma 4.7 that $t_{refine}(|V_i|, |E_i|) = O(|V_i| + |E_i|) + t_{single}(|V_i|, |E_i|)$. Hence, Formula 4.2 simplifies to

$$t_{mult}(|V|, |E|) \leq \sum_{i=0}^{k} (O(|V_i|, |E_i|) + t_{single}(|V_i|, |E_i|)). \tag{4.3}$$

Furthermore, we know from Lemma 4.6 that if $|V_i| \geq 2$, then $|V_{i+1}| \leq \frac{|V_i|}{2}$ for all $i = 0, \ldots, k-1$. Let us for a moment assume that additionally $|E_{i+1}| \leq \frac{|E_i|}{2}$ for all $i = 0, \ldots, k-1$. Then, Formula 4.2 becomes

$$\begin{aligned} t_{mult}(|V|, |E|) &\leq \sum_{i=0}^{k} \left(O\left(\frac{|V|}{2^i}, \frac{|E|}{2^i}\right) + t_{single}\left(\frac{|V|}{2^i}, \frac{|E|}{2^i}\right)\right) \\ &\leq \left(\sum_{i=0}^{k} \frac{1}{2^i}\right) \cdot (O(|V|, |E|) + t_{single}(|V|, |E|)) \\ &\leq 2(O(|V|, |E|) + t_{single}(|V|, |E|)) \\ &= O(t_{single}(|V|, |E|)). \end{aligned} \tag{4.4}$$

The last equality in Formula 4.4 holds since $t_{single}(|V|, |E|) = \Omega(|V| + |E|)$, and the second inequality holds for sufficiently large values of $|V|$ and $|E|$. Thus, since $t_{mult}(|V|, |E|) = \Omega(t_{single}(|V|, |E|)$, we get that $t_{mult}(|V|, |E|) = \Theta(t_{single}(|V|, |E|))$.

Certainly, the assumption $|E_{i+1}| \leq \frac{|E_i|}{2}$ for all $i = 0, \ldots, k-1$ cannot be guaranteed in general. Let us weaken this assumption so that $|E_{i+1}| \leq \frac{|E_i|}{s}$ for all $i = 0, \ldots, k-1$ and some rational constant s with $1 < s \leq 2$. In this case the same argumentation that has been used to transfer Formula 4.3 to Formula 4.4 can be used to obtain

$$t_{mult}(|V|, |E|) \leq \left(\frac{s}{s-1}\right) (O(|V|, |E|) + t_{single}(|V|, |E|)) = O(t_{single}(|V|, |E|)). \tag{4.5}$$

Thus, even under the weakened assumption we get that $t_{mult}(|V|, |E|) = \Theta(t_{single}(|V|, |E|))$.

If this assumption is weakened further so that $|E_{i+1}| \leq \frac{|E_i|}{s}$ for all but a constant number d of $i \in \{0, \ldots, k-1\}$, the equality $t_{mult}(|V|, |E|) = \Theta(t_{single}(|V|, |E|))$ holds as well.

Hence, in order to guarantee that $t_{mult}(|V|, |E|) = \Theta(t_{single}(|V|, |E|))$, we modeled the stopping criterion of the while-loop in the coarsening phase (see Section 4.2.2 and Function `Multilevel_Step`) so that the coarsening phase stops if the weakest assumption is violated.

Since this assumption (respectively the stopping criterion) implies that $\sum_{i=0}^{k}(|V_i|, |E_i|) = O(|V| + |E|)$, the memory requirements of the multilevel step are linear in $|V| + |E| + m_{single}(|V|, |E|)$. □

Remark 4.9. *Finally, we want to examine the running time of a variation of the multilevel step, in which our stopping criterion is replaced by a naive criterion that stops the coarsening process if and only if the actual multilevel graph G_i contains less than a constant number $c \geq 2$ of nodes: Since it is known from Lemma 4.6 that $|V_{i+1}| \leq \frac{|V_i|}{2}$ for all $i = 0, \ldots, k-1$ with $|V_i| \geq 2$, the number k of multilevel graphs is bounded above by $\log|V_0| = \log|V|$. Hence, if — in the worst case — additionally $|E_i| = \Theta(|E|)$ for all $i = 0, \ldots, k$, the total running time of the variation of the multilevel step would be bounded above by $t_{mult}(|V|, |E|) = O(\log|V| \cdot t_{single}(|V|, |E|))$.*

Function `Multilevel_Step`(G,$\mathtt{A}_{\mathtt{single}}$)

---

input : a connected positive-weighted undirected simple graph $G = (V, E, l^{zero})$ and a force-directed single-level algorithm $\mathtt{A}_{\mathtt{single}}$

output: a straight-line drawing $\Gamma(G)$ of G

begin
- **begin**{`Coarsening Phase`}
  - $i \leftarrow 0$;
  - $G_i \leftarrow G$;
  - **while** not `Stopping_Criterion`(G_i) **do**
    - create a galaxy partitioning of G_i using `Select_By_Star_Mass`;
    - create G_{i+1} by collapsing the solar systems of G_i;
    - create the relative-path-position list for each node of G_i;
    - $i \leftarrow i + 1$;
- **end**
- **begin**{`Refinement Phase`}
  - let $\{G_0 = (V_0, E_0, l_0^{zero}), \ldots, G_k = (V_k, E_k, l_k^{zero})\}$ be the set of multilevel graphs;
  - generate a random initial placement $p(V_k) := (p_v)_{v \in V_k}$ of the nodes of G_k;
  - $\Gamma(G_k) \leftarrow \mathtt{A}_{\mathtt{single}}(G_k, p(V_k))$;
  - $i \leftarrow k - 1$;
  - **while** $i \geq 0$ **do**
    - generate an initial placement $p(V_i) := (p_v)_{v \in V_i}$ of the nodes of G_i by using $\Gamma(G_{i+1})$ and the relative-path-position lists of the nodes of G_i;
    - $\Gamma(G_i) \leftarrow \mathtt{A}_{\mathtt{single}}(G_i, p(V_i))$;
    - $i \leftarrow i - 1$;
- **end**

end

Chapter 5

The Force-Calculation Step

> Niemand weiß, wie weit seine Kräfte gehen,
> bis er sie versucht hat. [1]

In this chapter we will describe the force-directed single-level algorithm that is used in the refinement phase of the multilevel step. After the description of the basic framework of this algorithm in Section 5.1, we will concentrate on its most important part — the approximation of the repulsive forces acting between all pairs of nodes. We will review existing methods for solving this problem in Section 5.2, before we will describe a new approximative method in Section 5.3 and summarize our results in Section 5.4.

5.1 The Framework of the Force-Calculation Step

5.1.1 Motivation and Goals

As we have seen in Section 1.2.2, the classical force-directed single-level algorithms are not suited for drawing large graphs because their asymptotic running time is at least quadratic in general. Hence, these algorithms are also not suited to serve as parts in multilevel frameworks. Consequently the other multilevel methods (that have been sketched in Section 1.3.2) use variations of these classical methods in the refinement phases:

Gajer et al. [49, 50] and Harel and Koren [64] use variations of the complete spring model of Kamada and Kawai [80], which ignore lots of the $\Theta(|V|^2)$ springs that are defined in the original model [80], to speed up the computation. In particular, no spring forces are calculated between nodes that have a large graph-theoretic distance.

The force model of Walshaw [132, 133] is based on pairwise repelling particles and identifying edges with springs, like in the methods of Eades [35] and Fruchterman and

[1] Johann Wolfgang von Goethe

Reingold [47]. In order to speed up the computation, the `Grid-Variant Algorithm` of Fruchterman and Reingold [47] is used that ignores the repulsive forces acting between particles that are placed geometrically far away from each other.

In summary, all these variations of the classical single-level algorithms are only crude approximation schemes, since the forces that act on a node v due to all nodes that are far away (either in a graph-theoretic sense or in a geometrical sense) are ignored. As a consequence of this, the energy minimization in the refinement phases of the corresponding multilevel frameworks is only local. This obvious circumstance has already been noted by Harel and Koren (see [66] page 216), who have stated that the methods [49, 62, 64, 132] "improve running times by rapidly constructing a simplified initial globally nice layout and then refining it locally".

In contrast to this, Tunkelang [127, 128] and Quigley and Eades [110] have used the repulsive-force-approximation method of Barnes and Hut [8] to approximate the repulsive forces acting in their model of charged particles and springs accurately. In particular, the force contribution of particles that are placed far away is calculated as a part of a group force (see Section 1.3.1). However, they did not embed their algorithm in a multilevel framework.

Our force model is also based on a system of pairwise repelling particles and on identifying edges with springs (see Section 2.2.2), and our single-level algorithm is motivated by the work of Tunkelang [127, 128] and Quigley and Eades [110]. The idea of our force-directed single-level algorithm is to overcome the quadratic running time of a naive approach for calculating the repulsive forces acting between all pairs of charged particles by inventing a method that approximates the repulsive forces in sub-quadratic time — but without ignoring the forces of particles that are placed far away. This goal has not been reached by any force-directed algorithm (compare Sections 1.2 and 1.3) we are aware of. As a consequence of this — in contrast to the other multilevel methods [49, 50, 64, 132, 133] — the energy minimization in the refinement phase in our multilevel framework can be described as a *global* refinement phase. We will examine in Section 7.7 if — in practice — the combination of our multilevel step with an accurate approximative single-level algorithm is advantageous in comparison with other multilevel methods that are based on local refinement phases only.

5.1.2 The Algorithm Embedder

Suppose, we have given a positive-weighted undirected simple graph $G = (V, E, l^{zero})$ and an initial placement $p(V) := (p_v)_{v \in V}$ of the nodes of G (as a result of the multilevel step that we have described in Chapter 4). Furthermore, suppose that G is the multilevel graph at level $l \in \{0, \ldots, k\}$ in a series of $k + 1$ multilevel graphs.

According to the definition of the absolute values of the forces in our force model (see Section 2.2.2), we define the repulsive forces that act on a node v at position p_v due to a node u at position p_u and the spring forces that act on v due to a spring $e = (u, v)$ as follows:

$$
\begin{aligned}
F^{u}_{rep}(v) &= \begin{cases} \frac{p_v - p_u}{\|p_v - p_u\|^2} & \text{if} \quad p_u \neq p_v \\ 0 & \text{otherwise} \end{cases} \\
F^{e=(u,v)}_{spring}(v) &= \begin{cases} -\log\left(\frac{\|p_v - p_u\|}{l^{zero}(e)}\right) \cdot \|p_v - p_u\| \cdot (p_v - p_u) & \text{if} \quad p_u \neq p_v \\ 0 & \text{otherwise} \end{cases}
\end{aligned} \tag{5.1}
$$

Definition 5.1 (Resulting Forces, Equilibrium State of the Forces, and Others). *Suppose, $p(V) = (p_v)_{v \in V}$ is a placement of the nodes of a positive-weighted undirected simple graph $G = (V, E, l^{zero})$. For each node $v \in V$ we define*

$$F_{rep}(v) = \sum_{u \in V \setminus v} F^{u}_{rep}(v) \quad \textit{and} \quad F_{spring}(v) = \sum_{u \in adj(v)} F^{(u,v)}_{spring}(v)$$

as the repulsive force *and* spring force*, respectively* that act on v in the system induced by $p(V)$ and G. *If λ_{rep} and λ_{spring} are positive constants, called* repulsion factor *and* spring stiffness factor*, then $F_{res}(v) = \lambda_{rep} \cdot F_{rep}(v) + \lambda_{spring} \cdot F_{spring}(v)$ denotes the* resulting (or total) force that acts on v in the system induced by $p(V)$ and G. *A node v is in an* equilibrium state of the forces *if $F_{res}(v) = 0$.*

Algorithm 5: `Embedder`$(G, p(V), l, k)$

input : a positive-weighted undirected simple graph $G = (V, E, l^{zero})$, an initial placement $p(V) := (p_v)_{v \in V}$ of the nodes of G, a repulsive-forces-approximation algorithm $\mathtt{A}_{\mathtt{approx}}$, and integer constants l, k that indicate that G is the multilevel graph at level $l \in \{0, \ldots, k\}$ in a series of $k + 1$ multilevel graphs

output: a straight-line drawing $\Gamma(G)$

```
begin
    initialize λ_spring, λ_rep, δ and t;
    calculate Max_Iter(l, k);
    i ← 1;
    repeat
        foreach v ∈ V do  F_spring(v) ← Σ_{u∈adj(v)} F^{(u,v)}_spring(v);
        F_rep ← A_approx(p(V));
        foreach v ∈ V do  F_res(v) ← λ_rep · F_rep(v) + λ_spring · F_spring(v);
        foreach v ∈ V do  F_displ(v) ← Displ_Factor(v, i, δ) · F_res(v)/‖F_res(v)‖;
        foreach v ∈ V do p_v ← p_v + F_displ(v);
        avg_force ← Σ_{v∈V} F_displ(v);
        i ← i + 1;
    until ((i > Max_Iter(l, k)) or (avg_force < t));
end
```

We call the algorithm that is used to obtain a drawing $\Gamma(G)$ `Embedder`, and its pseudocode is shown in Algorithm 5. Like other single-level algorithms, it iteratively tries to find a placement of the nodes, in which all nodes are *nearly* in an equilibrium state of the resulting forces (which means that for each $v \in V$ the resulting force $F_{res}(v)$ is very small). Starting with the given placement $p(V)$, the algorithm first calculates the spring force $F_{spring}(v)$ for each node $v \in V$. Then, like in [127, 128, 110] a repulsive-forces-approximation algorithm $\mathtt{A}_{\mathtt{approx}}$ (that does not ignore forces that are placed far away) is used to obtain an approximation of the repulsive forces $F_{rep}(v)$ that act on each node v. These forces are used to calculate the resulting forces $F_{rep}(v)$ that act on each node $v \in V$. After this, the position of each node $v \in V$ is updated by moving v in direction of $F_{rep}(v)$. The calculation of the optimal length (the *displacement factor*) of this vector (the *displacement vector* $F_{displ}(v)$) is discussed next.

An Adaptive Displacement Factor

The easiest way to define the displacement factor is to set $\texttt{Displ\_Factor}(v) := \delta \cdot \|F_{res}(v)\|$, where δ is a suitable positive constant, like in [35]. In order to obtain control on the absolute length of the resulting displacement vector, Fruchterman and Reingold [47] define the displacement factor as a global *cooling function* that depends on the actual iteration of their algorithm and set $\texttt{Displ\_Factor}(v, i) := min\{\|F_{res}(v)\|, cool(i)\}$. Here, i is the actual iteration of the algorithm, and *cool* is a monotonously decreasing function with positive values. Frick et al. [45] have introduced the idea of a local cooling function that is defined for each node v. It is used for calculating the displacement factor of each node v and takes previous calculated forces into account. This is done in order to avoid oscillations and rotations and to accelerate the convergence behavior of their algorithm.

The definition of the displacement factor that is used in our framework is motivated by that of Frick et al. [45], but it is less technical. The length of this displacement factor is determined depending on the direction and length of the displacement vector of v in the last iteration. The calculation of the displacement factor is described in the following.

Suppose that $F^i_{res}(v)$ is the resulting force acting on node v at iteration i and that $F^{i-1}_{displ}(v)$ is the displacement vector of node v in iteration $i-1$. If the angle between these vectors is $\alpha(i,v) := \sphericalangle(F^i_{res}(v), F^{i-1}_{displ}(v))$ and if δ is a suitable positive constant, then the displacement factor of a node v in iteration i is given by $\texttt{Displ\_Factor}(v, i, \delta) = \delta \cdot \texttt{Adaptive\_Cool}(v, i)$, with

$$\texttt{Adaptive\_Cool}(v,i) = \begin{cases} c_{i,v} \cdot \|F^{i-1}_{displ}(v)\| & \text{if} \quad \|F^i_{res}(v)\| > c_{i,v} \cdot \|F^{i-1}_{displ}(v)\| > 0 \\ \|F^i_{res}(v)\| & \text{otherwise} \end{cases}, \text{ and}$$

$$c_{i,v} = \begin{cases} 2 & \text{if} \quad -\frac{1}{6}\pi \le \alpha(i,v) \le \frac{1}{6}\pi \\ \frac{3}{2} & \text{if} \quad \frac{1}{6}\pi < \alpha(i,v) \le \frac{2}{6}\pi \quad \text{or} \quad -\frac{2}{6}\pi \le \alpha(i,v) < -\frac{1}{6}\pi \\ 1 & \text{if} \quad \frac{2}{6}\pi < \alpha(i,v) \le \frac{3}{6}\pi \quad \text{or} \quad -\frac{3}{6}\pi \le \alpha(i,v) < -\frac{2}{6}\pi \\ \frac{2}{3} & \text{if} \quad \frac{3}{6}\pi < \alpha(i,v) \le \frac{4}{6}\pi \quad \text{or} \quad -\frac{4}{6}\pi \le \alpha(i,v) < -\frac{3}{6}\pi \\ \frac{1}{3} & \text{otherwise} \end{cases}.$$

Hence, consecutive movements of a node in the same direction can result in longer displacement vectors than consecutive movements of a node in different directions. For example, suppose that at the last iteration $i-1$ node v has been moved into direction $F_{displ}^{i-1}(v)$ and that the length of this displacement was $\|F_{displ}^{i-1}(v)\|$. Then, if in iteration i node v has to be moved into the opposite direction of $F_{displ}^{i-1}(v)$, the displacement factor implies that the length of the displacement vector is bounded above by $\frac{1}{3} \cdot \|F_{displ}^{i-1}(v)\|$.

The Stopping Criterion

After all nodes in the actual iteration have been moved, the average length of all displacement vectors $F_{\text{displ}}(v)$ is calculated and the algorithm either stops, when the average length of these vectors falls below some fixed threshold value t or when the maximum number of iterations is reached. Instead of setting the maximum number of iterations to the same constant for any multilevel graph G, we define it as a function of the actual multilevel. In particular, let $c_{large} > c_{small} > 0$ be constants and let `Max_Iter`(l,k) denote the maximum number of iterations in Algorithm `Embedder` if G is a multilevel graph at level l of $k+1$ multilevel graphs. Then, we set `Max_Iter`$(0,k) = c_{large}$, `Max_Iter`$(k,k) = c_{small}$ and define for $l \in \{0,\ldots,k\}$ `Max_Iter`(l,k) as a function with values that are linear decreasing from c_{large} to c_{small}. This choice of `Max_Iter`(l,k) is motivated by the fact that (in order obtain faster running times) the maximum number of iterations for the smaller multilevel graphs can be chosen larger than for the larger multilevel graphs.

Lemma 5.2 (`Embedder`). *Suppose that $G = (V, E, l^{zero})$ is a positive-weighted undirected simple graph, $p(V) := (p_v)_{v \in V}$ is an initial placement of the nodes of G, and that G is the multilevel graph at level $l \in \{0,\ldots,k\}$ in a series of $k+1$ multilevel graphs. If $\mathtt{A}_{\mathtt{approx}}$ is a repulsive-forces-approximation algorithm that uses time $t_{approx}(|V|)$ and memory $m_{approx}(|V|)$ to approximate the repulsive forces that act in a system of $|V|$ charged particles, then Algorithm `Embedder` needs $O(|V| + |E| + t_{approx}(|V|))$ time to generate a straight-line drawing $\Gamma(G)$ of G and uses $O(m_{approx}(|V|))$ memory.*

Proof. The initialization of the constants $\lambda_{spring}, \lambda_{rep}, \delta, t$ and the calculation of `Max_Iter`(l, k) take constant time. In each iteration of the repeat-loop, the calculation of the spring forces can be done in $O\left(\sum_{v \in V} deg(v)\right) = O(|E|)$ time, and all other operations (except possibly $\mathtt{A}_{\mathtt{approx}}$) need $O(|V)|$ time. Since by construction `Max_Iter`(l,k) is a constant, the Algorithm `Embedder` needs $O(|V| + |E| + t_{approx}(|V|))$ time and uses $O(m_{approx}(|V|))$ additional memory. □

5.1.3 The Algorithm Grid_Embedder

For later use, we introduce a variant of Algorithm `Embedder` that is called `Grid_Embedder` and forces the nodes to be placed on an integer grid with a resolution that is polynomial in the number of nodes.

In particular, we want to restrict the particle positions to an integer grid in the range $[-d_1 N^{d_2}, d_1 N^{d_2}] \times [-d_1 N^{d_2}, d_1 N^{d_2}]$ for some positive integer constants d_1 and d_2. This

can be done by introducing Function `Restrict_to_Grid` that updates the position $p(v)$ of each node $v \in V$ as follows: If $p(v)$ is already an integer in the given range, nothing is done. If $p(v)$ is a position in the given range, but not whole-numbered, then we assign $p(v) = (p_x(v), p_y(v)) := (\lfloor p_x(v) \rfloor, \lfloor p_y(v) \rfloor)$. If $p(v)$ is not contained in $[-d_1 N^{d_2}, d_1 N^{d_2}] \times [-d_1 N^{d_2}, d_1 N^{d_2}]$, then $p(V) := p'(v)$, where $p'(v)$ is an integer position in $[-d_1 N^{d_2}, d_1 N^{d_2}] \times [-d_1 N^{d_2}, d_1 N^{d_2}]$ so that the Euclidean distance between $p'(v)$ and $p(v)$ is comparatively short. Such a position $p'(v)$ can be found for example by calculating the intersection point $i(v)$ of the line that connects the origin and $p(v)$ and one of the four line-segments that bound the integer grid. If $i(v)$ is already integer $p'(v) := i(v)$, otherwise $i(v)$ is rounded so that the resulting point $p'(v)$ is integer and contained in the integer grid. The pseudocode of Algorithm `Grid_Embedder` is given in Algorithm 6.

Algorithm 6: `Grid_Embedder`$(G, p(V), l, k)$

input : a positive-weighted undirected simple graph $G = (V, E, l^{zero})$, an initial placement $p(V) := (p_v)_{v \in V}$ of the nodes of G, a repulsive-forces-approximation algorithm $\mathtt{A}_{\mathtt{approx}}$, integer constants l, k that indicate that G is the multilevel graph at level $l \in \{0, \ldots, k\}$ in a series of $k+1$ multilevel graphs, and positive integer constants d_1 and d_2

output: a straight-line drawing $\Gamma(G)$ so that in $\Gamma(G)$ and before each call of $\mathtt{A}_{\mathtt{approx}}$ all nodes are placed on an integer grid in the range $[-d_1 N^{d_2}, d_1 N^{d_2}] \times [-d_1 N^{d_2}, d_1 N^{d_2}]$

begin
- initialize $\lambda_{spring}, \lambda_{rep}, \delta$ and t;
- calculate `Max_Iter`(l, k);
- $i \leftarrow 1$;
- **repeat**
  - $p(V) \leftarrow$ `Restrict_To_Grid`$(p(V), d_1, d_2)$;
  - **foreach** $v \in V$ **do** $F_{spring}(v) \leftarrow \sum_{u \in adj(v)} F^{(u,v)}_{spring}(v)$;
  - $F_{rep} \leftarrow \mathtt{A}_{\mathtt{approx}}(p(V))$;
  - **foreach** $v \in V$ **do** $F_{res}(v) \leftarrow \lambda_{rep} \cdot F_{rep}(v) + \lambda_{spring} \cdot F_{spring}(v)$;
  - **foreach** $v \in V$ **do** $F_{displ}(v) \leftarrow$ `Displ_Factor`$(v, i, \delta) \cdot F_{res}(v) / \|F_{res}(v)\|$;
  - **foreach** $v \in V$ **do** $p_v \leftarrow p_v + F_{displ}(v)$;
  - $avg\_force \leftarrow \sum_{v \in V} F_{displ}(v)$;
  - $i \leftarrow i + 1$;
- **until** $((i >$ `Max_Iter`$(l, k))$ or $(avg\_force < t))$;
- $p(V) \leftarrow$ `Restrict_To_Grid`$(p(V), d_1, d_2)$;

end

Corollary 5.3 (`Grid_Embedder`). *Suppose that $G = (V, E, l^{zero})$ is a positive-weighted undirected simple graph, $p(V) := (p_v)_{v \in V}$ is an initial placement of the nodes of G that G is the multilevel graph at level $l \in \{0, \ldots, k\}$ in a series of $k+1$ multilevel graphs,*

and d_1 and d_2 are positive integer constants. If $\texttt{A}_{\texttt{approx}}$ *is a repulsive-forces-approximation algorithm that uses time $t_{approx}(|V|)$ and memory $m_{approx}(|V|)$ to approximate the repulsive forces that act in a system of $|V|$ charged particles, then Algorithm* `Grid_Embedder` *needs $O(|V| + |E| + t_{approx}(|V|))$ time to generate a straight-line drawing $\Gamma(G)$ of G and uses $O(m_{approx}(|V|))$ memory. Before each call of* $\texttt{A}_{\texttt{approx}}$ *and in the final drawing $\Gamma(G)$, all nodes are placed on an integer grid in the range $[-d_1 N^{d_2}, d_1 N^{d_2}] \times [-d_1 N^{d_2}, d_1 N^{d_2}]$*

Proof. The claim follows from the fact that one call of Function `Restrict_To_Grid` needs $O(|V|)$ time in combination with Lemma 5.2. □

5.2 N-body Simulations in Physics

In the next Sections 5.2 and 5.3 we will concentrate on the problem of approximating the repulsive forces that act between all pairs of $|V|$ particles/nodes of a graph G for a fixed given placement $p(V)$. For simplicity, we will assume in Sections 5.2 to 5.3.5 that the positions of all nodes/particles are distinct. In Section 5.3.6 we will generalize our results to the unrestricted case in which any two nodes are allowed to be placed at identical positions.

In the following, we set $N := |V|$, identify V with a set $C = \{c_1, \ldots, c_N\}$ of charged particles with charges $q(C) := \{q_1 := 1, \ldots, q_N := 1\}$, and let $p(C) = (p_i)_{i \in \{1,\ldots,N\}} := p(V)$ denote the distinct positions of these charges in the plane. We call an algorithm that approximates the repulsive forces that are induced by a fixed placement $p(C)$ in a system of N charged particles C a *(repulsive-)force-approximation method*.

We have already noted in Section 1.3.1 that this problem has been widely studied in physics and is part of the interior loop of iterative *N-body simulation* algorithms [1]. These algorithm try to simulate the movements of N masses or charged particles in a specified time interval by iteratively calculating the forces acting in the system and updating the positions of the objects. An introduction into the field of force-approximation methods is the book *Many-body tree methods in physics* by Pfalzer and Gibbon [105] that presents some of the most important force-approximation methods. However, the remarks on the asymptotic running times of several algorithms that are sketched in this book should not be taken for granted (compare [3, 2]). The first kind of force-approximation methods (the *particle-in-cell codes* or *PIC codes*) are based on laying out a regular grid on the simulation area and on grouping particles that are placed in the same cell together. These groupings of the charges are used to calculate the forces (see [105]). Since PIC codes turned out to be useful only for distributions in which each cell contains approximately the same number of particles, so called *hierarchical* methods or *tree codes* have been developed. The most important hierarchical methods are introduced in the following. These methods have in common that they are based on a two phase approach. In the first phase, a hierarchical space-decomposition data structure is built up that is used to calculate an approximation of the forces acting in the system in the second phase.

Since the use of efficient data structures is of fundamental importance in order to obtain fast running times, we will introduce the hierarchical space-decomposition data

structures that are used by the presented N-body methods in Section 5.2.1. There, we will additionally present some construction methods and discuss their running times. We will formulate a lower bound on construction times for some hierarchical data structures in Section 5.2.2. The tree codes that use these data structures will be sketched in the subsequent Sections 5.2.3 to 5.2.6.

5.2.1 Spatial Data Structures

In general, *hierarchical space-decomposition data structures* or shorter *spatial data structures* are a class of data structures whose common property is the recursive decomposition of space. They can be differentiated on the following bases (see [116]):

- The type of data that they are used to represent (e.g., points, curves, surfaces, or volumes)
- The principle that guides the decomposition process
- The resolution (i.e., the number of times the decomposition process is applied)

We can restrict on spatial data structures that are designed to represent point data, since our data are the distinct positions $p(C) = (p_i)_{i\in\{1,\ldots,N\}}$ of the N charged particles that we will for simplicity often denote by *(data) points*. We will focus on regular space decomposition techniques that are represented by *point-region quadtrees* or shorter *PR quadtrees* according to the notation of H. Samet [116]. These methods are based on recursively subdividing a square region into four sub-squares of equal size. Furthermore, the data points are stored at the leaves of these trees. In order to help distinguishing the several different kinds of PR quadtrees (that are mostly denoted by *quadtree* in the literature) we will classify them in the following.

Other spatial data structures (like point quadtrees, pseudo point quadtrees, k-d trees, or adaptive k-d trees) and their application in computational geometry are detailedly described in the books *The Design and Analysis of Spatial Data Structures* [116] and *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS* [115] by Samet.

The Quadtree

The simplest and most popular kind of a PR quadtree that we denote by *quadtree* is defined after introducing some terminology.

Definition 5.4 (Cell, Box). *If D is a square, then a* (sub)-cell *or* (sub)-box *is a quadratic sub-region of a D that can be generated by a recursive decomposition of D into four squares of equal size.*

Suppose, $\{p_1, \ldots, p_N\}$ is a set of N data points that are distributed within a square D. The principle guiding the decomposition is as follows: If D contains more than one data

point, D is subdivided into four squares of equal size D_1, D_2, D_3, and D_4. Every box D_i with $i \in \{1, \ldots, 4\}$ that contains more than one data point is recursively subdivided into four smaller boxes. The decomposition stops, when the actual box contains at most one data point. The decomposition is represented by an orderer rooted tree T of maximum child degree four. Each node v of T is associated with a sub-box of D that is denoted by $box(v)$. The box of the root is D. Furthermore, each node v can be associated with the subset of the points in $\{p_1, \ldots, p_N\}$ that are covered by $box(v)$. The order of the children of an internal node v is defined by the relative positions of the four boxes that subdivide $box(v)$ (e.g., the first child corresponds to the left top sub-box, the second child corresponds to the right top sub-box, the third child corresponds to the left bottom sub-box, and the fourth child corresponds to the right bottom sub-box). If a point is located on a horizontal or vertical line that separates two sub-boxes, it belongs to the uppermost or rightmost sub-box, respectively. The data points are stored in the leaves of the quadtree, and often leaves that contain no data points (that we call *empty leaves*) are not stored in T for memory reasons. Unless otherwise stated, we will not represent empty leaves in the presented hierarchical data structures. Figure 5.1 shows an example of a quadtree.

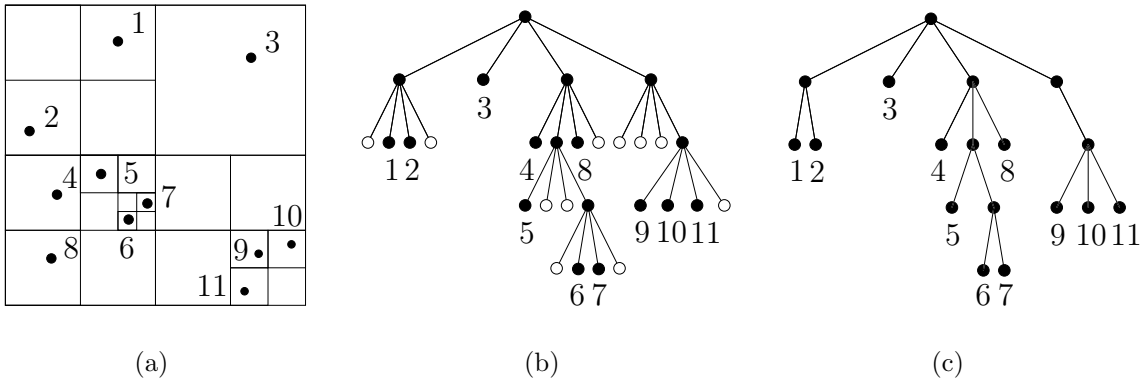


Figure 5.1: (a) The recursive decomposition of a square region that contains point data. (b) The corresponding quadtree in which white leaves represent empty sub-boxes. (c) The quadtree that corresponds to (a) but that does not contain empty leaves.

Maybe the first authors who used this data structure were Anderson [4] and Rosenfeld et al. [113]. They applied it for storing endpoints of line segments and in a geometric information system, respectively. In the context of force-approximation methods, this data structure was used by Barnes and Hut [8] first.

The definition of the quadtree data structure naturally implies a tree construction algorithm that we denote by $\mathtt{A}_1$ here. It works as follows: First, a root node is created, which represents the square D that contains all data points. Then, if this square contains more than one data point, the four sub-boxes and the four children are created and the contained data points are shifted from the actual node to the appropriate children. This process is recursively repeated until no further subdivisions can be carried out.

An alternative quadtree construction algorithm (presented in [8, 116, 128] and denoted

by $\mathtt{A}_2$ here) starts by inserting each data point iteratively into the quadtree starting at the root node: First, T consists of only one root node that represents the region D and contains the first data point. Then, the next data points are successively added to T starting at the root. If the actual visited node v is an internal node, the algorithm visits the child node that corresponds to the sub-box in which the data point is contained. If the actual visited node v is an empty leaf, the new data point is added to v. If v is a leaf so that $box(v)$ already contains a data point, recursively four new children of v are created, and the two data points are assigned to those children in which corresponding sub-box of $box(v)$ they are contained. This recursive subdivision process is repeated as long as both points are contained in the box of the same leaf.

Aluru et al. [3, 2] proved that the running time of these tree construction methods is bounded below by $\Omega(N \log N)$ and that no upper bound on the construction times can be given that depends only on the number of data points.

We will demonstrate at an example that there exist distributions of the data points that imply a $\Theta(N^2)$ construction time for both algorithm, even though the corresponding quadtree contains only $O(N)$ nodes: Suppose, we have given a distribution of the point data like in Figure 5.2 that we call *converging* distribution. Suppose that the down left corner of D is $(0, 0)$ and the upper right corner of D is a point p, then such a distribution is formally given by assigning the i-th data point the position $p_i = \frac{3}{4}\frac{p}{2^{i-1}}$ for all $i \in \{1, \ldots, N\}$. Figure 5.2 shows a converging distribution, the recursive space decomposition, and the corresponding quadtree.

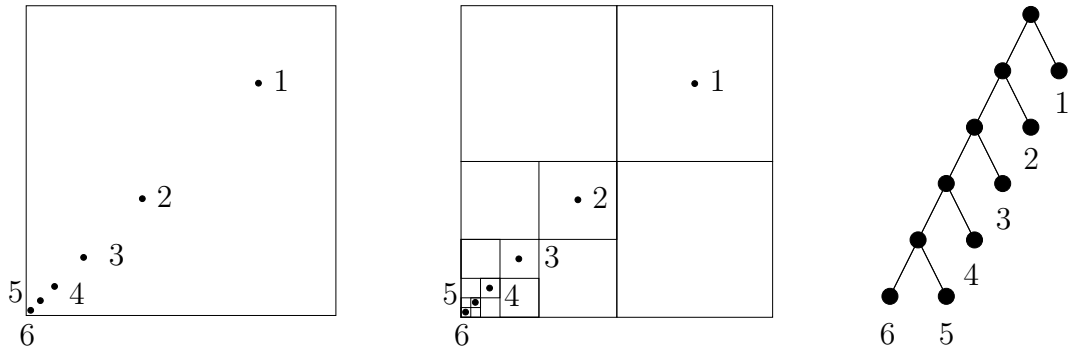


Figure 5.2: (a) A converging point distribution of $N = 6$ points. (b) A recursive space decomposition according to the point distribution in (a). (c) The associated quadtree.

Lemma 5.5 (A Pathologic Distribution for Some Quadtree Construction Methods). *Suppose that $\{p_1, \ldots, p_N\}$ is a converging point distribution of N data points. Then, the described quadtree construction methods $\mathtt{A}_1$ and $\mathtt{A}_2$ need $\Theta(N^2)$ time to build up the associated quadtree that contains $O(N)$ nodes.*

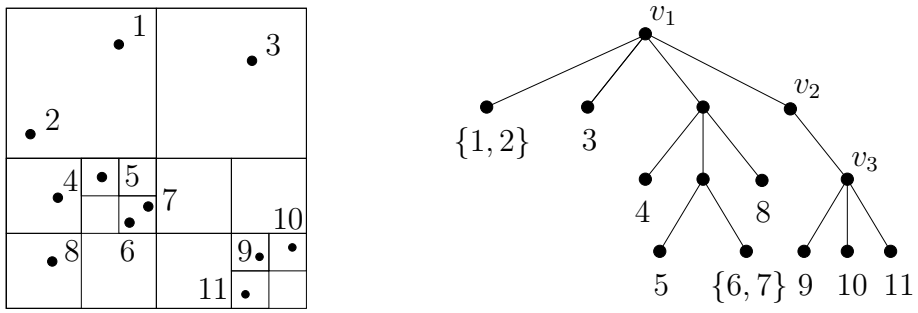
Proof. In the method $\mathtt{A}_1$ all N particles are assigned to the root node, which takes $\Theta(N)$ time. Then, $N - 1$ particles are assigned to the child node v that corresponds to the left

bottom sub-box of D and one point is assigned to the child node that corresponds to the right top sub-box of D. This also takes $\Theta(N)$ time. Now the four children of v are created, $N-2$ points are assigned to the child node that corresponds to the left bottom sub-box of $box(v)$, while only one child is assigned to the child node that corresponds to the right top sub-box of $box(v)$. This needs $\Theta(N-1)$ time. Repeating this subdivision and assigning process leads to the total running time of $\sum_{i=1}^{N} \Theta(i) = \Theta(N^2)$.

We define the *total leaf-path length* of a tree as the sum of the length of all paths that connect the root of the quadtree with each leaf. By definition of the iterative quadtree construction method $\mathtt{A_2}$, it is easy to see that the running time is proportional to the total leaf-path length (see also [128]), which is $\left(\sum_{i=1}^{N} i\right) - 1 = \Theta(N^2)$ for converging distributions. □

The Bucket Quadtree

Suppose, we have given an arbitrary set $\{p_1, \ldots, p_N\}$ of N points that are distributed within a square region D and define a data structure that is exactly defined as the quadtree data structure, with one exception: Instead of stopping the decomposition process, whenever the actual sub-box contains at most one point, one could stop the decomposition, when the actual sub-box contains at most l points, where l is a positive integer that is called the *bucket capacity* or *leaf capacity*. We call the resulting data structure *(bucket) quadtree with leaf capacity l*. Note that the previously introduced quadtree is a bucket quadtree with leaf capacity one. This data structure has the property that all leaves contain at most l points. We call a path $P = (v_1, \ldots, v_k)$ in a bucket quadtree *degenerate path* if v_1 and v_k have at least two nonempty children and $v_2, \ldots, v_{k-1}$ each have exactly one nonempty child. The nodes $v_2, \ldots, v_{k-1}$ are called *degenerate nodes*. For example, Figure 5.3 shows a bucket quadtree with leaf capacity 2 that contains the degenerate path (v_1, v_2, v_3) in which v_2 is the unique degenerate node.



(a) (b)

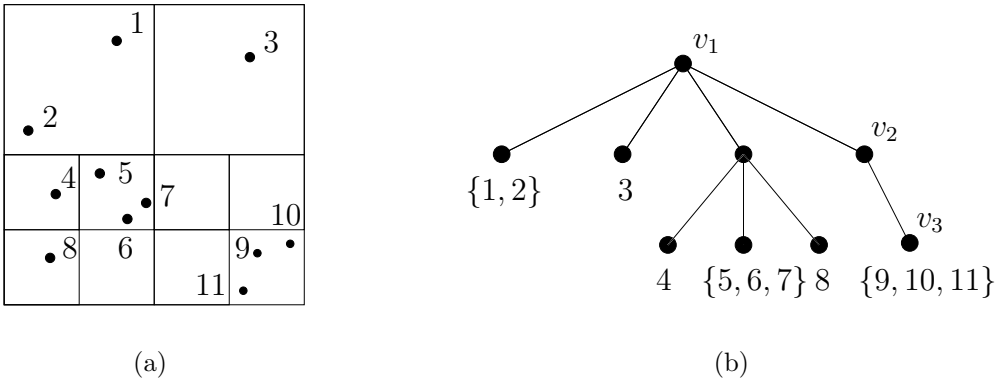
Figure 5.3: (a) The recursive decomposition of a square region that contains point data. (b) The corresponding bucket quadtree with leaf capacity $l = 2$.

This data structure has been invented by Matsuyama et al. [94] and is used by the adaptive force-approximation method of Greengard [58].

It is obvious that bucket quadtrees with leaf capacity $l > 1$ can be constructed by generalizing the quadtree construction methods $\mathtt{A}_1$ and $\mathtt{A}_2$. If we assume that the leaf capacity l of bucket quadtrees is a constant, it is also easy to see that these tree construction methods would need $\Theta(N^2)$ time to construct the bucket quadtrees in the case that the point data is distributed convergingly. The formal proof is analogue to the proof of Lemma 5.5 and left to the reader.

The Truncated Quadtree and Truncated Bucket Quadtree

Suppose, we have given a positive integer d and we construct a quadtree decomposition of the square D (with width $width(D)$) according to the given point data in D. If the recursive decomposition of the sub-boxes is stopped either if the actual sub-box contains at most $l = 1$ point or if the sub-box has width at most $width(D)/(2^d)$, the data structure is called *truncated quadtree with maximal depth d*. If $l > 1$, the corresponding data structure is called *truncated bucket quadtree with leaf capacity l and maximal depth d*. This data structure has the properties that the depth of the tree is bounded above by d and that all leaves of depth smaller than d contain at most l points, while the leaves of depth d might contain more than l points. We call a path $P = (v_1, \ldots, v_k)$ in a truncated bucket quadtree *degenerate path* if v_1 has at least two nonempty children, $v_2, \ldots, v_{k-1}$ each have exactly one nonempty child, and v_k has at least two nonempty children or is a leaf. The nodes $v_2, \ldots, v_{k-1}$ are called *degenerate nodes*. For example, Figure 5.4 shows a truncated bucket quadtree with leaf capacity 2 and maximal depth 2 that contains the degenerate path (v_1, v_2, v_3) in which v_2 is the unique degenerate node. This data structure will be useful in Section 5.3.3.

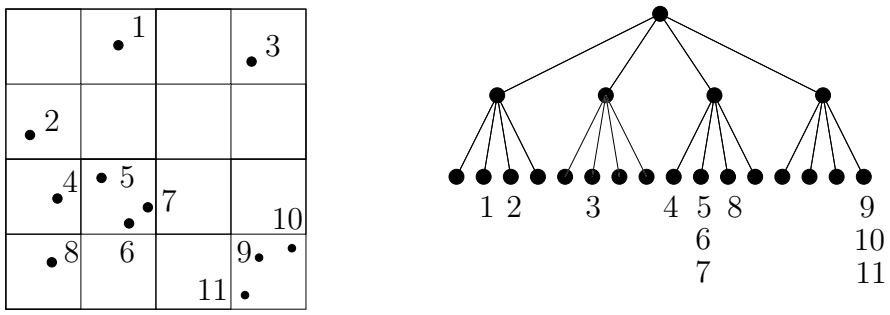


(a) (b)

Figure 5.4: (a) The recursive decomposition of a square region that contains point data. (b) The corresponding truncated bucket quadtree with leaf capacity $l = 2$ and maximal depth $d = 2$.

The Truncated Pseudo Quadtree

Suppose, we have given a positive integer d, the square D (with width $width(D)$), and the point data in D, but — in contrast to the definition of the truncated quadtree — every sub-box is subdivided until it has width at most $width(D)/(2^d)$. Then, the resulting data structure is called *truncated pseudo quadtree of depth d*. Note that by this definition, the recursive subdivision is also performed in the case that the actual sub-box contains no points. Hence, the truncated pseudo quadtree of depth d is a complete tree with child degree 4 and depth d, in which each leaf possibly stores an arbitrary number of data points. We denoted this data structure by pseudo quadtree, since — unlike the previous types of PR quadtrees — the decomposition is not guided by the distribution of the point data in D. Figure 5.5 shows an example of a truncated pseudo quadtree of depth 2.



(a) (b)

Figure 5.5: (a) The recursive decomposition of a square region that contains point data. (b) The corresponding truncated pseudo quadtree of depth $d = 2$.

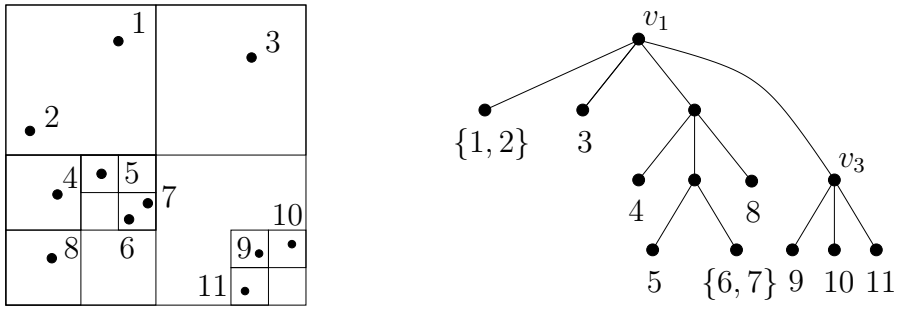
Ohya et al. [99] used this data structure that they called *quaternary tree* for constructing Voronoi Diagrams, while it has been used in the context of force-approximation methods as a part of the `Fast_Multipole_Method` of Greengard and Rokhlin [59] and by the tree code of Xue [139].

A truncated pseudo quadtree of depth d can be constructed by first building a complete tree of depth d and child degree 4 and then assigning each data point to the leaf that corresponds to the sub-box at level d in which it is placed (compare [59, 58, 99]). Since the structure of the tree is predefined and since it contains $O(4^d)$ nodes, the construction of the tree and the assignment of the N data points to the leaves can be done in $O(N + 4^d)$ time.

The Reduced Quadtree, Reduced Bucket Quadtree, and Reduced Truncated Bucket Quadtree

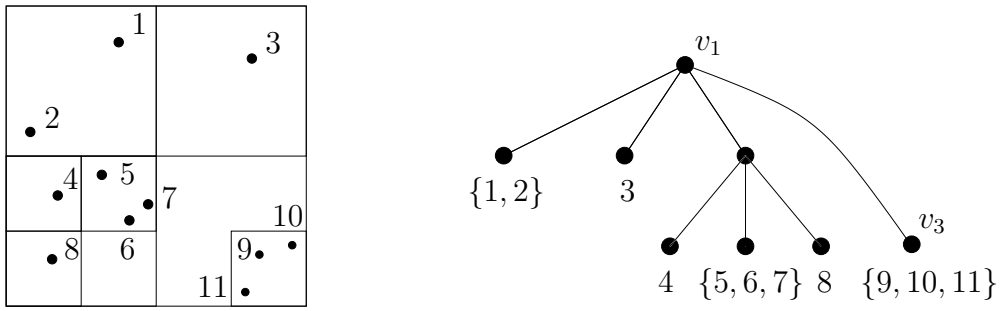
A *reduced quadtree* can be obtained from a quadtree T by shrinking all degenerate paths $P = (v_1, \ldots, v_p)$ in T to edges (v_1, v_p). This data structure has been invented by Aluru et

al. [3, 2], who called it *modified tree*, in the context of their force-approximation method. Similar, a *reduced bucket quadtree with leaf capacity l* and a *reduced truncated bucket quadtree with leaf capacity l and maximal depth d* can be obtained from a bucket quadtree with leaf capacity l and a truncated bucket quadtree with leaf capacity l and maximal depth d, respectively by shrinking all degenerate paths $P = (v_1, \ldots, v_p)$ to edges (v_1, v_p). Figures 5.6 and 5.7 show a reduced bucket quadtree with leaf capacity 2 and a reduced truncated bucket quadtree with leaf capacity 2, respectively. These data structures will be used in Section 5.3.



(a) (b)

Figure 5.6: (a) The recursive decomposition of a square region that contains point data. (b) The corresponding reduced (bucket) quadtree with leaf capacity $l = 2$.



(a) (b)

Figure 5.7: (a) The recursive decomposition of a square region that contains point data. (b) The corresponding reduced truncated (bucket) quadtree with leaf capacity $l = 2$ and maximal depth $d = 2$.

In contrast to the quadtree and bucket quadtree data structures in which the distribution of the point data possibly implies that the corresponding trees contains a huge number of degenerate nodes, the following estimation on the number of nodes in a reduced quadtree holds.

Lemma 5.6. *Suppose, $\{p_1, \ldots, p_N\}$ is an arbitrary set of N points in a square region D, then a reduced quadtree $T = (V, E)$ for this point set contains $\frac{4}{3}N - \frac{1}{3} \leq |V| \leq 2N - 1$ nodes.*

Proof. If $N = 1$ we are done. Otherwise, since T is a tree, we get that

$$2(|V| - 1) = 2|E| = \sum_{v \in V} deg(v). \tag{5.2}$$

Since T is a reduced quadtree and $N > 1$, the root r has degree $deg(r) \in \{2, 3, 4\}$, each other interior node $v \in V$ has degree $deg(v) \in \{3, 4, 5\}$, while all leaves have degree $deg(v) = 1$. Let $I(T)$ denote the set of all interior nodes of T (excluding the root) and $L(T)$ denote the set of the leaves of T, then (5.2) can be formulated as

$$2(|V| - 1) = deg(r) + \sum_{v \in I(T)} deg(v) + \sum_{v \in L(T)} deg(v), \tag{5.3}$$

and the following estimation holds:

$$2 \cdot 1 + 3 \cdot (|V| - N - 1) + 1 \cdot N \leq 2(|V| - 1) \leq 4 \cdot 1 + 5 \cdot (|V| - N - 1) + 1 \cdot N \tag{5.4}$$

Trivial simplifications of (5.4) lead to

$$|V| \leq 2N - 1 \quad \text{and} \quad |V| \geq \frac{4N - 1}{3}, \tag{5.5}$$

which completes the proof. □

Corollary 5.7 (Node-Number in Reduced Bucket Quadtrees). *Suppose, $\{p_1, \ldots, p_N\}$ is an arbitrary set of N points in a square region D, then for any given positive integer constant l, a reduced bucket quadtree with leaf capacity l contains $O(N)$ nodes.*

Note that this property will be of fundamental importance in the design of our force-approximation method in Section 5.3.

Aluru et al. [2] present a method for constructing the reduced quadtree in $O(N \log N)$ time, which is cost optimal — as we will prove in Theorem 5.8. This method is based on a merging strategy that first builds up roughly $N/2$ reduced quadtrees that contain at most two data points. Then, pairwise two of these reduced quadtrees are merged in order to obtain roughly $N/4$ reduced quadtrees that contain at most 4 data points. After the next merging operations the number of reduced quadtrees decreases to roughly $N/8$ reduced quadtrees that contain at most 8 data points, and so forth. In order to merge two subtrees a *box-shrinking* technique has been invented that is non-trivial and will be explained in greater detail in Section 5.3.2. Since the total running time of the merging operations at each level is $O(N)$ and the number of recursion levels is bounded by $O(\log N)$, the total running time is $O(N \log N)$ (compare [2]).

A reduced bucket quadtree with leaf capacity $l > 1$ can be constructed in $O(N \log N)$ time, by first constructing a reduced quadtree T in $O(N \log N)$ time using the method of

Aluru et al. [2]. Then, one could shrink all maximal subtrees of T that contain at most l data points in their leaves to one leaf that contains all these points. This can be one in $O(N)$ time since T has $O(N)$ nodes (see Corollary 5.7). Alternative ways for constructing reduced bucket quadtrees will be developed in Sections 5.3.2 and 5.3.3. In Section 5.3.3, we will also introduce a method for constructing reduced truncated bucket quadtrees with maximal depth d in $O(N + 4^d)$ time.

5.2.2 A Lower Bound on PR Quadtree Construction Methods

Aluru et al. [2] state that the construction of a reduced quadtree for an arbitrary given distribution $\{p_1, \ldots, p_N\}$ of N points is $\Omega(N \log N)$. Since they do not present a formal proof of this claim, we will prove a more general result in the following.

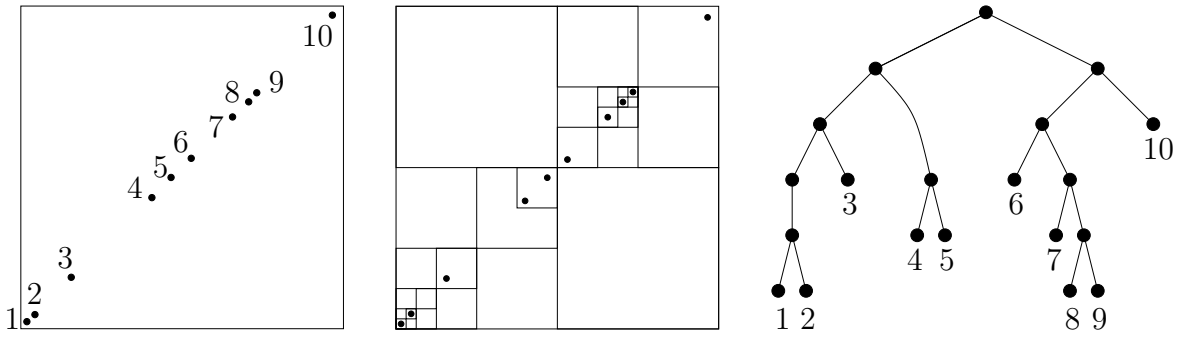
Theorem 5.8 (A Lower Bound on (Reduced) Bucket Quadtree Construction). *Suppose, $\{p_1, \ldots, p_N\}$ is an arbitrary set of N points that are contained in an arbitrary fixed square region D, then for any given positive integer constant l, neither a bucket quadtree with leaf capacity l nor a reduced bucket quadtree with leaf capacity l can be constructed in $o(N \log N)$ time.*

Proof. We prove this theorem by contradiction. Let us for simplicity assume that $T = (V, E)$ is a reduced bucket quadtree with leaf capacity $l = 1$, and that T can be constructed in $o(N \log N)$ time. Suppose that $S = \{s_1, \ldots, s_N\}; s_i \in \mathbb{N}$ is a set of keys. Then, we create an algorithm `A` that sorts the keys in S as follows:

We define a mapping $f\colon S \longrightarrow S^2$ so that $i \mapsto (i, i)$ and assign each key $s_i \in S$ a position in the plane according to f (see Figure 5.8(a)). Then, we build up a reduced bucket quadtree with leaf capacity $l = 1$ for the resulting point set (see Figures 5.8(b) and (c)), initialize an empty list L, and visit the nodes of T by a pre-order traversal. If (during this traversal) a leaf is visited, the x-coordinate of the contained point is appended at the beginning of L as a new element. Since the children of each internal node in the tree are ordered (like defined in Section 5.2.1), L does contain the keys of S in nondecreasing order after the traversal.

Using Corollary 5.7, all parts of this algorithm, except possibly the construction of the tree, need $O(N)$ time. Thus, Algorithm `A` sorts the keys in S in $o(N \log N)$ time. This is a contradiction, since it is known that any general sorting algorithm needs $\Omega(N \log N)$ time for sorting N arbitrary keys in the worst case [100].

Now let us suppose that T is a reduced bucket quadtree with leaf capacity $l > 1$. Then, we can construct a contradiction by an analogue argumentation. The only thing that changes in the argumentation is that the points that are contained in a leaf of T have to be pre-sorted by non-increasing x-coordinates before appending the x-coordinates consecutively in this order to L. Since the number of points that are contained in a leaf is bounded by the constant l, for each leaf this sorting can be done in constant time. After the tree traversal, the resulting list L is a sorted list of the keys, and the total running time of the sorting algorithm is $o(N \log N)$— a contradiction.



(a) (b) (c)

Figure 5.8: (a) Assigning $N = 10$ keys positions in the plane according to f. The positions are labeled according to the relative values of their x-coordinates (e.g., the point with the smallest x-coordinate has label 1). (b) A recursive decomposition of the quadratic drawing plane D. (c) The reduced quadtree that corresponds to this decomposition. A pre-order traversal of the tree will visit the leaves from right to left (i.e., first the leaf that contains point 10, finally the leaf that contains point 1).

If T is a bucket quadtree with leaf capacity l, we can assume that the number of nodes in T is $o(N \log N)$ (otherwise we already obtain a contradiction, since each node of T has to be created somehow). Under this assumption the contradiction proof is analogue to the previous one. □

5.2.3 The Method of Barnes and Hut

Maybe caused by its simplicity, the force-approximation method of Barnes and Hut [8] is one of the most popular methods that is used as part of N-body simulation algorithms. It works as follows: For the given set $C = \{c_1, \ldots, c_N\}$ of equally charged particles that are placed at distinct positions $p(C) = \{p_1, \ldots, p_N\}$, first a quadtree decomposition of a square region that contains all particles is calculated according to the construction algorithm $\mathtt{A}_1$ in Section 5.2.1. Each internal node of the quadtree stores additional information, a so called *group particle*. In particular, suppose that an internal node v of the quadtree T corresponds to a sub-box that contains k particles that are placed at positions $p_1, \ldots, p_k$. Then, a group particle c with group charge $q_c := k$ and position $p_c := (\sum_{i=1}^{k} p_i)/k$ is assigned to v.

In the second phase of the algorithm the forces are approximated by calculating *particle-(group particle)* interactions for each particle $c_i \in C$ that is stored in a leaf of T. This is described next. First, a *precision parameter* $\alpha > 0$ is fixed that is used to steer the accuracy of the approximation. Then, the quadtree T is traversed top down starting at the root on the following basis: Suppose, we want to calculate the force $F(c_i)$ that acts on a particle $c_i \in C$ that is placed at position p_i and stored in a leaf u of T. Furthermore, suppose that the actual visited node of T is v and $box(v)$ contains a group particle c_v with

charge q_c at position p_c. Suppose that the width of the sub-box that is represented by v is d_v and that $d_{i,c}$ denotes the Euclidean distance between p_i and p_c. Then, if $u = v$ nothing is done. If the sub-box of v covers the sub-box of u the algorithm proceeds to the children of v. If the sub-box of v does not cover the sub-box of u and v is a leaf, then the force that acts on p_i due to the particle that is stored in v is added to $F(c_i)$. If the sub-box of v does not cover the sub-box of u, v is an interior node, and $(d_v/d_{i,c}) < \alpha$, the force that acts on c_i due to the group particle c is calculated and added to $F(c_i)$. If the sub-box of v does not cover the sub-box of u, v is an interior node, and $(d_v/d_{i,c}) \geq \alpha$ the algorithm proceeds to the children of v.

Aluru et al. [3, 2] have proven that the tree construction time is bounded below by $\Omega(N \log N)$ and that no upper bound on the construction time can be given that depends only on the number of particles. Nevertheless, one could think about the possibility of using other methods for constructing the quadtree in order to obtain a sub-quadratic running time of the force approximation. In the following theorem we will show that the worst-case running time of the method of Barnes and Hut [8] will never become sub-quadratic, even if one could build up the quadtree efficiently.

Theorem 5.9 (The Method of Barnes and Hut Will Never Become Sub-Quadratic). *There exists a distribution of N particles in the plane so that the second phase of the force-approximation method of Barnes and Hut needs $\Theta(N^2)$ time for approximating the forces for any given precision parameter $\alpha > 0$.*

Proof. Suppose that the particles $C = \{c_1, \ldots, c_N\}$ are distributed convergingly, like defined in Section 5.2.1 and that the corresponding quadtree T has already been built up in the first phase of the algorithm (see Figure 5.2). Suppose, the particles in C are ordered so that c_1 is the particle with the smallest x-coordinate and c_N is the particle with the largest x-coordinate.

It is easy to see (by the definition of the second phase of the method of Barnes and Hut) that in order to approximate the forces that act on a particle c_i each node of the tree will be visited at most once. Hence, at most $2N - 1$ nodes of T will be visited for each $c_i \in C$. Of course, the exact number of nodes that have to be visited depends on the precision parameter α. But, due to the inclusion relationships of the sub-boxes for the converging distribution (and by the definition of the second phase of the method of Barnes and Hut) $2i - 1$ is a lower bound on the number of nodes that have to be visited in order to obtain the approximation of the forces that act on a particle c_i.

Hence, we obtain that the second phase of the method of Barnes and Hut needs $O(N \cdot (2N - 1)) = O(N^2)$ and $\Omega(\sum_{i=1}^{N}(2i - 1)) = \Omega(N^2)$ time for approximating the forces for the given converging distribution of particles. □

For completeness, we would like to note that the asymptotic running time of the method of Barnes and Hut [8] is $O(N \log N)$ in the special case that the placement of the particles implies a depth of the quadtree that is $O(\log N)$ (see [114, 56]). In particular, the expected running time of the method of Barnes and Hut [8] is $O(N \log N)$ if the particles are distributed randomly with uniform probability.

5.2.4 The Fast Multipole Method

Other popular force-approximation methods are so called *multipole methods* that are based on building up a hierarchical data structure and approximating the forces by evaluating the field of potential energy at the positions of the charged particles using *multipole expansions*.

The pioneer work is the `Fast_Multipole_Method` of Greengard [58] and Greengard and Rokhlin [59]. In the first phase of the algorithm a truncated pseudo quadtree of depth $\lfloor \log_4 N \rfloor$ is constructed in $O(N)$ time according to the investigated construction algorithm in Section 5.2.1.

In the second phase of the algorithm coefficients of so called *p-term multipole expansions* are calculated for the set of particles that are contained in each leaf. We will introduce the formal descriptions of multipole expansions as well as many of the analytical tools that have been invented by Greengard [58] and Greengard and Rokhlin [59] for working with these series in Section 5.3.4. Now it is sufficient to know that these are truncated Laurent series (consisting of only the first p terms) with complex values. Each series describes the field of potential energy that is induced by the contained particles.

After the coefficients of the p-term multipole expansions of the particles in the leaves have been calculated, the tree is traversed bottom up, and thereby coefficients of p-term multipole expansions of the interior nodes are obtained. Afterwards, the tree is traversed top down, and suitable coefficients of p-term multipole expansions are used to calculate coefficients of special truncated power series that are called *p-term local expansions*. Finally, these expansions are evaluated to obtain the repulsive forces.

The best-case running time of the `Fast_Multipole_Method` is $O(N)$, while it has an $O(N^2)$ worst-case running time (see [58]). In particular, the expected running time of the `Fast_Multipole_Method` is linear if the particles are distributed randomly with uniform probability.

A more complicated `Adaptive_Algorithm` that is based on first building up a bucket quadtree with constant leaf capacity $l > 1$ was developed by Greengard [58]. Then, this tree is traversed bottom up and top down in order to obtain an approximation of the field of the potential energy in the system. This is done by using similar techniques as in the `Fast_Multipole_Method`.

Aluru et al. [3, 2] proved that the `Adaptive_Algorithm` of Greengard is $\Omega(N \log N)$ and that no upper bound on the running time can be given that depends only on the number particles due to the tree construction procedure.

Despite these drawbacks of the `Fast_Multipole_Method` and the `Adaptive_Algorithm`, the "excellent exposition" (see [20] page 86) of Greengard [58] motivated lots of research in the field of N-body simulation methods and in particular important parts of this dissertation.

5.2.5 An $O(N \log N)$ Multipole Method

One of the most remarkable algorithms that are based on the work of Greengard [58] and Greengard and Rokhlin [59] is the force-approximation method of Aluru et al. [2] that is

sketched next.

First, a reduced quadtree is built up in $O(N \log N)$ time according to the algorithm that we have described in Section 5.2.1. Then, similar to [58, 59], the tree is traversed bottom up in order to calculate coefficients of p-term multipole expansions. Afterwards, the tree is traversed top down using suitable coefficients of p-term multipole expansions in order to calculate coefficients of p-term locale expansions. Finally, these expansions are evaluated at the leaves to obtain an approximation of the forces.

Since Aluru et al. [2] prove that the second phase of their algorithm needs $O(N)$ time, the total running time of their algorithm is $O(N \log N)$.

5.2.6 Related Work

A large number of other publications are related with the previously sketched methods and, hence, we can only mention some of them.

Notably, Petersen et al. [103] invented a new version of the `Fast_Multipole_Method` [58, 59] that is a factor 1.2 faster than the original version in practice, while Hrycak and Rokhlin [71] improved the experimental running time of the `Adaptive_Algorithm` of Greengard [58] by a factor 2 to 4.

Warren and Salmon [136] integrated multipole ideas into their Barnes-Hut-like tree code. This is done by calculating *particle-multipole* interactions instead of *particle-(group particle)* interactions like in the original method of Barnes and Hut [8]. Grama et al. [56] extended these ideas, by keeping the multipole degree variable in order to reduce the error of the computation. Board et al. [12] have invented another hybrid version that is a combination of the `Fast_Multipole_Method` [58, 59] and the method of Barnes and Hut [8] and runs on parallel machines. Since the data sets of practical applications can be enormous, many other parallel versions of tree codes have been invented (see e.g., [137, 92, 55, 122, 141]).

Xue [139] improved one of the oldest tree codes of Appel [5] by building up a truncated pseudo quadtree in linear time. Under the assumption that the particles are distributed randomly with uniform probability, the running time of this method is linear.

Theoretical error estimations for the `Fast_Multipole_Method` were presented by Peterson et al. [104, 102], and a multipole method that uses a relaxed version of a k-d tree data structure (called *fair split tree*) was presented by Callahan and Kosaraju[20].

We have concentrated on approximation methods in two-dimensional systems that contain charged particles here. It is clear that variations of the presented techniques are also used in the context of astrophysical simulations and in three dimensions. In three dimensions, the PR quadtree concept is transfered into an analogue PR oct-tree concept. The transfer of the tree construction methods is straight forward, like the use of Barnes-Hut-like tree codes that rely on PR oct-trees decompositions. In contrast to this, the variations of multipole based tree codes are technically more complicated in three dimensions due to the used mathematical tools (see Greengard [58]). As a consequence of this, three-dimensional versions of the `Fast_Multipole_Method` turned out to be slow in practice (see [137, 120]). In particular, for uniformly distributed particles and in two dimensions,

the `Fast_Multipole_Method` is faster than the naive exact force-computation method for instances containing more than few hundreds particles. In contrast to this, the crossover point of the three dimensional version of the `Fast_Multipole_Method` is at around 70000 particles in practice [137, 120].

5.3 The New Multipole Method

5.3.1 Motivation and Goals

We have seen in Section 5.2 that several force-approximation methods have been developed in order to reduce the quadratic running time of a naive exact calculation of the forces acting in a system of N charged particles. Some of these methods guarantee an $O(N \log N)$ or an $O(N)$ running time under certain assumptions on the given distribution, but not in general. In particular, these methods perform well if the particles are distributed randomly with uniform probability. In contrast to this, the multipole-based force-approximation method of Aluru et al. [2] guarantees an $O(N \log N)$ scaling in general. Hence, it would be sufficient to use the method of Aluru et al. [2] in order to keep the running time of our force-calculation step sub-quadratic.

However, we will invent a new variant of a multipole method in order to obtain an $O(N \log N)$ method that is faster than the method of Aluru et al. [2] in practice. We will call this multipole method ***N**ew **M**ultipole **M**ethod* or shorter `NM`^{2}. Like other tree codes it is based on a two phase approach: First, a reduced bucket quadtree with fixed constant leaf capacity $l > 1$ is built up. Then, this data structure is used for approximating the repulsive forces in a multipole framework in the second phase.

Why Do We Construct a Reduced Bucket Quadtree?

We have decided to use the reduced bucket quadtree data structure because of the following reasons: First, we know from the previous discussion that this data structure can be built up in $O(N \log N)$ time, which is cost optimal.

Second, it provides some properties that will be important for the design of the multipole framework in the second phase of the approximation method that runs in $O(N)$ time. Most notably are the properties that it has $O(N)$ nodes that all leaves contain at most l particles and that the decomposition is based on a regular subdivision of the space.

Third, since this data structure is related to the bucket quadtree data structure and to the reduced quadtree data structure, many analytical tools that are provided by Greengard [58], Greengard and Rokhlin [59], and Aluru et al. [2] can be reused as parts of our multipole framework in the second phase. If, instead, we would use other spatial data structures that are based on different subdivision and data storage techniques — like point quadtrees [42], pseudo point quadtrees [101], k-d trees [10], adaptive k-d trees [46], or BD trees [97, 98] — we would lose several of the useful properties of the reduced bucket quadtree and, therefore, we would have to invent a completely new multipole framework

from scratch. Additionally, to our knowledge, none of the construction methods of these other data structures has been proven to be $o(N \log N)$ in general.

In Section 5.2.1 we have already sketched a method for constructing reduced bucket quadtrees that is based on the method of Aluru et al. [2]. However, such a simple approach has two disadvantages. First, the tree construction method of Aluru et al. [2] which is non-trivial has to be used. Second, this approach would perform unnecessary work by first creating a tree with leaf capacity 1 (which in general contains more nodes than a reduced bucket quadtree with leaf capacity $l > 1$) and then deleting the unnecessary nodes from this tree. Consequently, we have developed alternative ways for constructing reduced bucket quadtrees and will present them in Sections 5.3.2 and 5.3.3.

Why Do We Develop a Multipole Method?

Aluru et al. [2] extended the multipole techniques of Greengard [58] and Greengard and Rokhlin [59] so that the running time of their algorithm is proportional to the number of nodes of the used reduced quadtree data structure. However, it can be observed that lots of time-consuming mathematical operations with the coefficients of the multipole expansions are needed in the second phase of the method of Aluru et al. [2] due to the large size of the tree.

Therefore, we wanted to generalize the second phase of the approach of Aluru et al. [2] so that the running time is proportional to the number of nodes in the reduced bucket quadtree with any fixed constant leaf capacity $l \geq 1$. On the one hand, this will guarantee that the second phase of our algorithm is $O(N)$. On the other hand, this approach will keep the hidden constant of this bound comparatively small in practice (see Section 7.4). The reason for this time saving is simple: Since a bucket quadtree with leaf capacity $l > 1$ contains at most as many nodes as a reduced quadtree, fewer calculations with multipole expansions have to be done. Furthermore — if the constant leaf capacity l is chosen appropriate — in our experiments (see Section 7.4) it has turned out to be faster to calculate the forces acting between all particles that are covered by the box of a single leaf exactly than approximating these forces by using multipole expansions. The second phase of $\texttt{NM}^2$ will be presented in Section 5.3.4, and a formal description of $\texttt{NM}^2$ will be given in Section 5.3.5.

5.3.2 Construction of the Reduced Bucket Quadtree (Way A)

In the following, let us suppose that the particles $C = \{c_1, \ldots, c_N\}$ are distributed at distinct positions $p(C) = \{p_1, \ldots, p_N\}$ and that a fixed constant leaf capacity $l \geq 1$ is given. The tree construction method that we will present next is based on combining a box-shrinking technique of Aluru et al. [2] with techniques that are motivated by ideas of Callahan and Kosaraju [20] in their construction method of the fair split tree. First, we will sketch the box-shrinking technique of Aluru et al. [2], before we will give an informal and a formal description of the algorithm.

A Box-Shrinking Technique

We concentrate on the following problem: Given a square D with width $width(\mathrm{D})$, left bottom corner at position $(x_{min}(D), y_{min}(D))$, and right top corner at position $(x_{max}(D), y_{max}(D))$. Suppose, R is a rectangle of width $width(R)$, height $height(R)$, left bottom corner at position $(x_{min}(R), y_{min}(R))$, right top corner at position $(x_{max}(R), y_{max}(R))$, and D covers R. We want to determine the smallest sub-box B of D so that B covers R. Figure 5.9(a) shows an example.

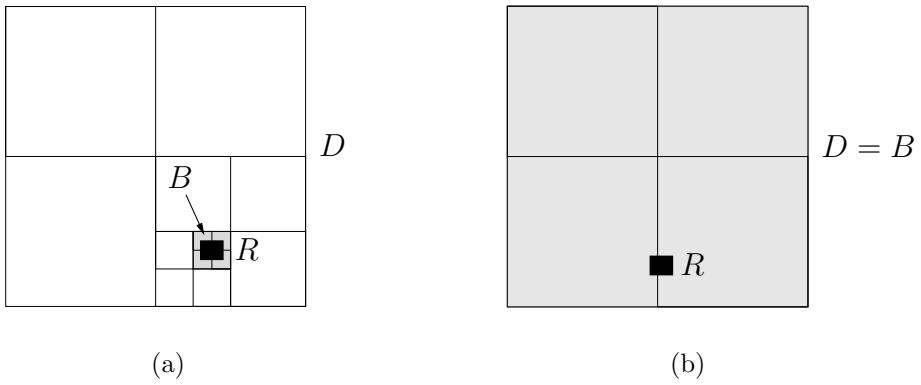


Figure 5.9: (a) A black rectangle R that is contained in a square D. The grey box B is the smallest sub-box of D that can be constructed by a recursive decomposition of D into four boxes of equal size so that B covers D. (b) Shifting R to a different position in D results in $B = D$.

A simple way to calculate B is given by recursively decomposing D into four sub-boxes, until R does not fit into one of the generated sub-boxes for the first time. For example, in Figure 5.9(a) four such subdivisions are needed to determine the size and position of B, while in Figure 5.9(b) only one decomposition of D is needed. In general, such a naive approach would need up to $\lceil \log_2(width(D)/max\{width(R), height(R)\}) \rceil$ recursive subdivisions of D.

In contrast to this, Aluru et al. [2] have developed a way to find B with a fixed constant number of operations for any values of D and R. We will not discuss the details of this strategy here, since it is very technical. Instead, we will present the needed formulas in order to enable an easy implementation. The readers that are interested in more details are referred to the paper of Aluru et al. [2].

It is clear that the width of the box B and the coordinate of its down left corner can be obtained by

$$\begin{aligned} width(B) &:= \tfrac{width(D)}{2^{max\_sub}}\,, \\ x_{min}(B) &:= x_{min}(D) + \lfloor \tfrac{x_{min}(R) - x_{min}(D)}{width(B)} \rfloor \cdot width(B)\,, \text{ and} \\ y_{min}(B) &:= y_{min}(D) + \lfloor \tfrac{y_{min}(R) - y_{min}(D)}{width(B)} \rfloor \cdot width(B)\,, \end{aligned} \tag{5.6}$$

where *max_sub* denotes the maximum number of recursive subdivisions of D that can be performed so that after each subdivision one of the four constructed sub-boxes covers R. For example, in Figure 5.9(a) $max\_sub = 3$ and in Figure 5.9(b) $max\_sub = 0$. Let

$$j_x := \left\lceil \log_2 \frac{width(D)}{width(R)} \right\rceil, \quad a_x^1 := \left\lceil \frac{(x_{min}(R) - x_{min}(D)) \cdot 2^{j_x}}{width(D)} \right\rceil, \quad a_x^2 := \left\lfloor \frac{(x_{max}(R) - x_{min}(D)) \cdot 2^{j_x}}{width(D)} \right\rfloor,$$

$$j_y := \left\lceil \log_2 \frac{width(D)}{height(R)} \right\rceil, \quad a_y^1 := \left\lceil \frac{(y_{min}(R) - y_{min}(D)) \cdot 2^{j_y}}{width(D)} \right\rceil, \quad a_y^2 := \left\lfloor \frac{(y_{max}(R) - y_{min}(D)) \cdot 2^{j_y}}{width(D)} \right\rfloor,$$

$$a_x := \begin{cases} a_x^1 & \text{if } a_x^1 = a_x^2 \\ a_x^1 & \text{if } a_x^1 \neq a_x^2 \text{ and } a_x^1 \text{ is even} \\ a_x^2 & \text{if } a_x^1 \neq a_x^2 \text{ and } a_x^2 \text{ is even} \end{cases}, \qquad a_y := \begin{cases} a_y^1 & \text{if } a_y^1 = a_y^2 \\ a_y^1 & \text{if } a_y^1 \neq a_y^2 \text{ and } a_y^1 \text{ is even} \\ a_y^2 & \text{if } a_y^1 \neq a_y^2 \text{ and } a_y^2 \text{ is even} \end{cases},$$

$$k_x := j_x - \log_2((a_x \oplus (a_x - 1)) + 1), \text{and} \qquad k_y := j_y - \log_2((a_y \oplus (a_y - 1)) + 1),$$

where $\oplus$ is the bitwise exclusive or operation. Then, B can be calculated by setting $max\_sub := min\{k_x, k_y\}$ in Formula (5.6). Since only a constant number of operations are needed to determine $x_{min}(B)$, $y_{min}(B)$, and $width(B)$, the total running time of the box-shrinking technique is $O(1)$.

The Basic Strategy of the Tree Construction Algorithm

The basic strategy of this algorithm is to first compute a partial reduced bucket quadtree T^1 that is a subtree of the desired reduced bucket quadtree T. In T^1, each leaf corresponds to a box that contains at most $max\{l, \lfloor N/2 \rfloor\}$ particles. Then, recursively for each leaf v of T^1, which contains more than l particles, a subtree of the reduced bucket quadtree T that is rooted at v is built up. Each leaf of these subtrees corresponds to a box that contains at most $max\{l, \lfloor N/4 \rfloor\}$ particles. The recursion ends, when all leaves correspond to regions that contain at most l particles. In the following, we will present a detailed description of the tree construction algorithm and illustrate it at an example in which we set the leaf capacity l to 2.

Part 1: Initialize the square D

First, we have to define the square D that covers the particles in C. This can be done by calculating the maximum and minimum values of the x- and y-coordinates of all particles (namely $\{x_{min}, x_{max}, y_{min}, y_{max}\}$) and using this information to obtain a small square D with width $width(D) := \max\{x_{max} - x_{min}, y_{max} - y_{min}\} + \epsilon$ (where $\epsilon > 0$ is a constant) that covers the particles in C. Figure 5.10(a) shows the square D that has been created for a distribution of $N = 11$ particles.

Part 2: Create Sorted Coordinate Lists S_x and S_y

Now two lists S_x and S_y of length N are created. Each element of the list S_x contains a different particle of C and two other entries that are initialized as empty entries and

will be used later. Analogue, each element of the list S_y contains a different particle of C and two other entries that are initialized as empty entries. The list elements of S_x and S_y that contain the same particle are linked by a cross reference to enable efficient access operations. Then, the lists S_x and S_y are sorted according to increasing x-coordinates and y-coordinates of the positions of the contained particles, respectively. Table 5.1(top) shows the sorted S_x and S_y lists of the particle distribution of Figure 5.10(a).

S_x	2 $\emptyset,\emptyset$	8 $\emptyset,\emptyset$	4 $\emptyset,\emptyset$	5 $\emptyset,\emptyset$	1 $\emptyset,\emptyset$	6 $\emptyset,\emptyset$	7 $\emptyset,\emptyset$	11 $\emptyset,\emptyset$	3 $\emptyset,\emptyset$	9 $\emptyset,\emptyset$	10 $\emptyset,\emptyset$
S_y	11 $\emptyset,\emptyset$	8 $\emptyset,\emptyset$	9 $\emptyset,\emptyset$	10 $\emptyset,\emptyset$	6 $\emptyset,\emptyset$	7 $\emptyset,\emptyset$	4 $\emptyset,\emptyset$	5 $\emptyset,\emptyset$	2 $\emptyset,\emptyset$	3 $\emptyset,\emptyset$	1 $\emptyset,\emptyset$

$C_x(v_1)$	2	8	4	5	1	6	7	11	3	9	10
$C_y(v_1)$	11	8	9	10	6	7	4	5	2	3	1

Table 5.1: (top) The sorted lists S_x and S_y of the particle distribution of Figure 5.10(a) and (bottom) the corresponding lists $C_x(v_1)$ and $C_y(v_1)$ of root node v_1.

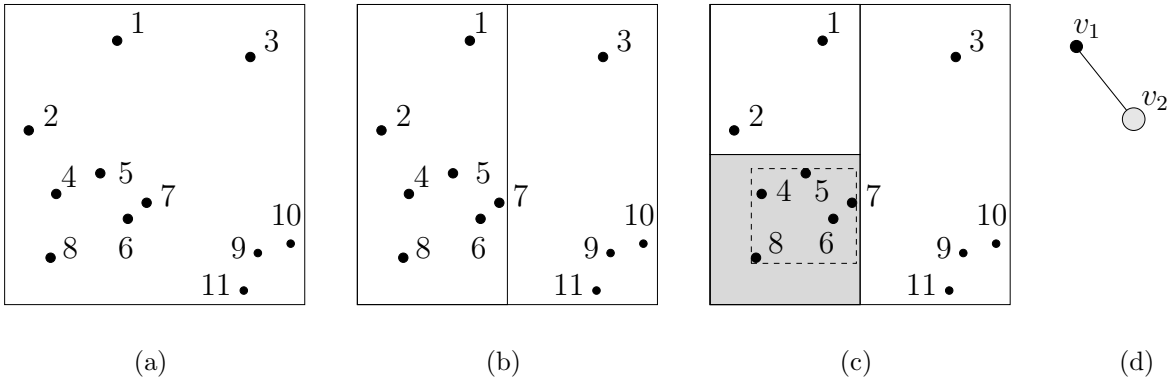


Figure 5.10: (a) A distribution of $N = 11$ particles in a square D that is associated with the root node v_1 of T. (b) Subdividing $D = box(v_1)$ into two halves. The left half of $box(v_1)$ contains more particles than the right half. (c) Subdividing the left half of D into two boxes. The left bottom box contains more particles than the left top box. The dashed rectangle is the smallest rectangle that covers all particles contained in the grey sub-box of D. (d) Creating a new child v_2 of v_1 that corresponds to the left bottom sub-box of $box(v_1)$.

Part 3: Create the Root Node and Copies of S_x and S_y

The root node of the reduced bucket quadtree T (which corresponds to the square D) is created next. Since this node is also the actual visited node, we denote it by v_{act}, and the corresponding square region is $box(v_{act}) := D$.

Furthermore, a copy of S_x and a copy of S_y are created. In contrast to S_x and S_y, they contain the particle entries only. These lists are assigned to the root node and are denoted by $C_x(v_{act})$ and $C_y(v_{act})$. To enable efficient access operations, for each $i \in \{1, \ldots, N\}$ the i-th entry in S_x is linked with the i-th entry of $C_x(v_{act})$ and vice versa. Analogue, the i-th entry in S_y is linked with the i-th entry of $C_y(v_{act})$ and vice versa. Additionally, like for S_x and S_y, cross references between elements of $C_x(v_{act})$ and $C_y(v_{act})$ that contain the same particle are created. If $N < l$ we assign all particles in C to v_1 and are done, since v_1 is already the reduced bucket quadtree. Otherwise, we proceed with the next parts of the algorithm.

For example, Table 5.1(bottom) shows the lists $C_x(v_1)$ $C_x(v_2)$ of the root node v_1 and the distribution of Figure 5.10(a). Since $N = 11 > 2 = l$, the algorithm proceeds to the next part.

Part 4: Alternating $C_x(v_{act})$ Traversal

Suppose, v_{act} is the actual visited node and $box(v_{act})$ is the sub-box of D that is represented by v_{act}. Then, $C_x(v_{act})$ is traversed alternating from the beginning and the end of $C_x(v_{act})$ heading toward the middle. This is done in order to decide which particles of $C_x(v_{act})$ belong to the left half of $box(v_{act})$ and which particles belong to the right half of $box(v_{act})$.

Let $M = \{c_1, \ldots, c_k\} \subseteq C$ be the set of particles in S_x that are contained in the half of $box(v_{act})$ that contains the fewest particles. Then, for each particle $c_i \in M$ a link to v_{act} and an index that identifies that c_i belongs to the left (index l) or right (index r) part of $box(v_{act})$ is added to the elements containing c_i in S_x and S_y.

Afterwards, all elements of $C_x(v_{act})$ and $C_y(v_{act})$ that contain a particle $c_i \in M$ are deleted from $C_x(v_{act})$ and $C_y(v_{act})$, respectively.

For example, let v_1 be the root node that corresponds to the box $box(v_1) = D$ shown in Figure 5.10(a). Then, the 7 particles $\{2, 8, 4, 5, 1, 6, 7\}$ are contained in the left half of $box(v_1)$ and the 4 particles $\{11, 3, 9, 10\}$ are contained in the right half of $box(v_1)$ (see Figure 5.10(b)). Hence, each element of S_x and S_y that contains a particle of the set $\{11, 3, 9, 10\}$ is assigned a link to node v_1 and the index r. Afterwards, the elements that contain the particles 11, 3, 9 and 10 are deleted from $C_x(v_{act})$ and $C_y(v_{act})$ (see Table 5.2).

S_x	2	8	4	5	1	6	7	11	3	9	10
	$\emptyset, \emptyset$	$\emptyset, \emptyset$	$\emptyset, \emptyset$	$\emptyset, \emptyset$	$\emptyset, \emptyset$	$\emptyset, \emptyset$	$\emptyset, \emptyset$	v_1, r	v_1, r	v_1, r	v_1, r
S_y	11	8	9	10	6	7	4	5	2	3	1
	v_1, r	$\emptyset, \emptyset$	v_1, r	v_1, r	$\emptyset, \emptyset$	$\emptyset, \emptyset$	$\emptyset, \emptyset$	$\emptyset, \emptyset$	$\emptyset, \emptyset$	v_1, r	$\emptyset, \emptyset$

$C_x(v_1)$	2	8	4	5	1	6	7
$C_y(v_1)$	8	6	7	4	5	2	1

Table 5.2: The lists $S_x, S_y, C_x(v_1)$ and $C_y(v_1)$ after the alternating $C_x(v_1)$ traversal.

Part 5: Alternating $C_y(v_{act})$ Traversal

Let *large_half*(v_{act}) be the half of *box*(v_{act}) that contains the most particles. Then, analogue to the $C_x(v_{act})$ traversal, $C_y(v_{act})$ is traversed alternating from the beginning and the end of $C_y(v_{act})$ heading toward the middle in order to decide which particles of $C_y(v_{act})$ belong to the bottom half of *large_half*(v_{act}) and to the top half of *large_half*(v_{act}). Note that these regions are boxes of equal size.

Let $M = \{c_1, \ldots, c_k\} \subseteq C$ denote the set of particles in S_y that are contained in the box of *large_half*(v_{act}) that covers the fewest particles. Then, for each particle $c_i \in M$ a link to v_{act} and an index that identifies that c_i belongs to the top half or bottom half of *large_half*(v_{act}) is added to the entries of c_i in S_x and S_y. In particular, if *large_half*(v_{act}) is the left half of *box*(v_{act}), the indices are lt for left top sub-box and lb for left bottom sub-box. If *large_half*(v_{act}) is the right half of *box*(v_{act}) the indices are rt for right top sub-box and rb for right bottom sub-box. Afterwards, all entries in $C_x(v_{act})$ and $C_y(v_{act})$ that contain a particle $c_i \in M$ are deleted from $C_x(v_{act})$ and $C_y(v_{act})$, respectively.

In our example, the 5 particles $\{8, 6, 7, 4, 5\}$ are contained in the left bottom sub-box of *box*(v_1) and the 2 particles $\{2, 1\}$ are contained in the left top sub-box of *box*(v_1) (see Figure 5.10(c)). Hence, each element of S_x and S_y that contains a particle of the set $\{2, 1\}$ is assigned a link to node v_1 and the index lt. Afterwards, the elements that contain the particles 2 and 1 are deleted from $C_x(v_{act})$ and $C_y(v_{act})$ (see Table 5.3).

S_x	2 v_1, lt	8 $\emptyset, \emptyset$	4 $\emptyset, \emptyset$	5 $\emptyset, \emptyset$	1 v_1, lt	6 $\emptyset, \emptyset$	7 $\emptyset, \emptyset$	11 v_1, r	3 v_1, r	9 v_1, r	10 v_1, r
S_y	11 v_1, r	8 $\emptyset, \emptyset$	9 v_1, r	10 v_1, r	6 $\emptyset, \emptyset$	7 $\emptyset, \emptyset$	4 $\emptyset, \emptyset$	5 $\emptyset, \emptyset$	2 v_1, lt	3 v_1, r	1 v_1, lt

$C_x(v_1)$	8	4	5	6	7
$C_y(v_1)$	8	6	7	4	5

Table 5.3: The lists S_x, S_y, $C_x(v_1)$ and $C_y(v_1)$ after the alternating $C_y(v_1)$ traversal.

Part 6: Create a Child Node

As a result of parts 5 and 6, we know that the particles that are stored in $C_x(v_{act})$ and $C_y(v_{act})$ are exactly the particles that are contained in one sub-box of *sqaure*(v_{act}). Now a child node v_{child} of v_{act} is created that corresponds to this sub-box of *box*(v_{act}) (denoted by *box*(v_{child})). Furthermore, $C_x(v_{act})$ and $C_y(v_{act})$ are assigned to v_{child} and labeled $C_x(v_{child})$ and $C_y(v_{child})$, respectively. If the box *box*(v_{child}) contains at most l particles we proceed with part 8 of this algorithm, otherwise we proceed with part 7.

In our example, a child node v_2 of v_1 that corresponds to the left bottom sub-box of *box*(v_1) is created (see Figures 5.10(c) and (d)). Since *box*(v_2) contains more than $l = 2$ particles, we proceed with part 7 of the algorithm.

Part 7: Use the Box-Shrinking Technique

Let R be the smallest axis parallel rectangle that covers all particles that are contained in $box(v_{child})$. Since more than l particles are contained in $box(v_{child})$, we have to decompose $box(v_{child})$ further in order to construct the reduced bucket quadtree. Additionally, we have to avoid that the constructed tree contains degenerate nodes. Hence, the box-shrinking technique of Aluru et al. [2] can be used to find the smallest sub-box B of $box(v_{child})$ so that B covers R. Finally, we set $box(v_{child}) := B$.

In our example, $box(v_2)$ is already the smallest sub-box of $box(v_2)$, which covers the smallest rectangle that contains the particles $\{8, 6, 7, 4, 5\}$ (see Figure 5.10(c)).

Part 8: Create a Path

The previously described parts 4 to 7 can be reused to create a path P in our reduced bucket quadtree T by setting $v_{act} := v_{child}$ and by repeating parts 4 to 7, until the box of the actual node ($box(v_{act})$) contains less than l particles and, hence, is a leaf of T. Finally, the remaining particles of $C_x(v_{act})$ are assigned to the leaf v_{act}.

In our example, after performing parts 4 to 5 of our algorithm on node v_2, we obtain the S_x, S_y, $C_x(v_2)$, and $C_y(v_2)$ lists that are shown in Table 5.4. In parts 6 and 7 a new child v_3 of v_2 is added to the tree (see Figure 5.11(a)) that corresponds to the shaded box shown in Figure 5.11(b).

S_x	2 v_1, lt	8 v_2, l	4 v_2, l	5 $\emptyset, \emptyset$	1 v_1, lt	6 $\emptyset, \emptyset$	7 $\emptyset, \emptyset$	11 v_1, r	3 v_1, r	9 v_1, r	10 v_1, r
S_y	11 v_1, r	8 v_2, l	9 v_1, r	10 v_1, r	6 $\emptyset, \emptyset$	7 $\emptyset, \emptyset$	4 v_2, l	5 $\emptyset, \emptyset$	2 v_1, lt	3 v_1, r	1 v_1, lt

$C_x(v_2)$	5	6	7
$C_y(v_2)$	6	7	5

Table 5.4: The lists S_x, S_y, $C_x(v_2)$ and $C_y(v_2)$ after the alternating $C_x(v_2)$ and $C_y(v_2)$ traversal.

Since the sub-box that corresponds to v_3 contains more than $2 = l$ particles, parts 4 to 5 of the algorithm are performed on node v_3. Hence, we obtain the S_x, S_y, $C_x(v_3)$, and $C_y(v_3)$ lists that are shown in Table 5.5. In part 6 a new child v_4 is added to the tree (see Figure 5.11(c)) that corresponds to the shaded sub-box shown in Figure 5.11(d). Since $box(v_4)$ contains only $2 = l$ particles the construction of the path stops and the remaining particles in $C_x(v_4)$ (the particles 6 and 7) are assigned to v_4. The constructed path is $P = (v_1, v_2, v_3, v_4)$.

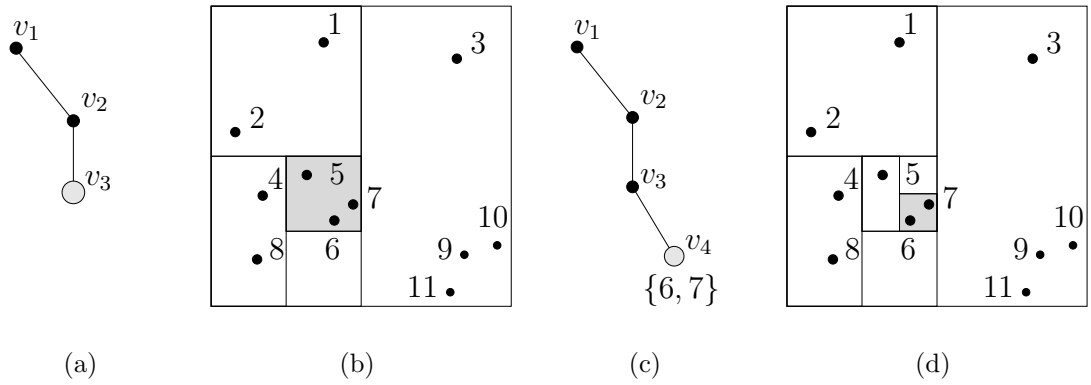


Figure 5.11: Constructing a path of the reduced bucket quadtree T. (a) A new node v_3 is added to the tree that corresponds to the grey box in (b). (c) A new node v_4 is added to the tree that corresponds to the grey box in (d). Since the box of v_4 contains only $l = 2$ particles the construction of the path stops, and the particles 6 and 7 are assigned to v_4.

S_x	2 v_1, lt	8 v_2, l	4 v_2, l	5 v_3, l	1 v_1, lt	6 $\emptyset, \emptyset$	7 $\emptyset, \emptyset$	11 v_1, r	3 v_1, r	9 v_1, r	10 v_1, r
S_y	11 v_1, r	8 v_2, l	9 v_1, r	10 v_1, r	6 $\emptyset, \emptyset$	7 $\emptyset, \emptyset$	4 v_2, l	5 v_3, l	2 v_1, lt	3 v_1, r	1 v_1, lt

$C_x(v_2)$	6	7
$C_y(v_2)$	6	7

Table 5.5: The lists S_x, S_y, $C_x(v_2)$ and $C_y(v_2)$ after the alternating $C_x(v_2)$ and $C_y(v_2)$ traversal.

Part 9: Create All Children of the Nodes to Which Links in S_x Exist

Now for each node in the actual constructed tree T — to which links in S_x are stored — children are created. (Note that the set of these nodes are exactly the nodes contained in P here. In further recursive calls of part 9, the set of these nodes will be the node sets of several paths in general (see description of part 10)). These children are created as follows:

First, the list S_x is traversed from left to right starting with the entry that contains the particle with the smallest x-coordinate. If, during this process, one element $(c_j, v_i, \textit{region\_index})$ of S_x contains a link to a node v_i — and it is the first time that an element of S_x with a link to v_i has been visited — a new list $C_x(v_i, \textit{region\_index})$ is created that contains the particle c_j. If, during this process, one element $(c_j, v_i, \textit{region\_index})$ of S_x contains a link to a node v_i, and a corresponding list $C_x(v_i, \textit{region\_index})$ already exists, c_j is appended at the end of $C_x(v_i, \textit{region\_index})$. Afterwards, the list S_y is traversed from left to right starting with the entry that contains the particle with the lowest y-coordinate and lists $C_y(v_i, \textit{region\_index})$ are created in an analogue way.

Note that by construction each list $C_x(v_i, \textit{region\_index})$ and $C_y(v_i, \textit{region\_index})$ stores

all particles that are contained in the sub-region of $box(v_i)$ that corresponds to the region indicated by *region_index* in increasing order according to the x- and y-coordinates of the particles, respectively. Table 5.6 shows the $C_x(v_i, region\_index)$ and $C_y(v_i, region\_index)$ lists that are obtained from the S_x and S_y lists of Table 5.5.

$C_x(v_1, lt)$	2	1
$C_y(v_1, lt)$	2	1

$C_x(v_2, l)$	8	4
$C_y(v_2, l)$	8	4

$C_x(v_3, l)$	5
$C_y(v_3, l)$	5

$C_x(v_1, r)$	11	3	9	10
$C_y(v_1, r)$	11	9	10	3

Table 5.6: The sorted lists of $C_x(v_i, region\_index)$ and $C_y(v_i, region\_index)$ that are obtained from the S_x and S_y lists of Table 5.5.

In order to enable fast access to the elements of these lists, cross references between the elements of S_x, S_y, and the $C_x(v_i, region\_index)$ and $C_y(v_i, region\_index)$ lists are built up. In particular, each element of S_x that contains a particle c_j is linked with the element of the corresponding list $C_x(v_i, region\_index)$ that contains the particle c_j and vice versa. Analogue, a cross reference between each element of S_y that contains a particle c_j and the element of the corresponding list $C_y(v_i, region\_index)$ that contains the particle c_j is constructed. Finally, for each element of a list $C_x(v_i, region\_index)$ that contains a particle c_j, a link to the element of the list $C_y(v_i, region\_index)$ that contains particle c_j is constructed and vice versa. For later use, the elements of S_x and S_y are reinitialized by replacing each element $(c_j, v_i, region\_index)$ with the element $(c_j, \emptyset, \emptyset)$.

Suppose that we have given a node v_i of the actual tree T for which the lists $C_x(v_i,$ *region_index*$)$ and $C_y(v_i,$ *region_index*$)$ exist, and *region_index* is either l or r. Then, we know that the particles that are stored in these lists are covered by the left or right half of $box(v_i)$, respectively. But in order to create children of v_i, we have to know which particles of these lists are stored in the top sub-box and which particles are stored in the bottom sub-box of this half of $box(v_i)$. Hence, we split each list $C_x(v_i, region\_index)$ with $region\_index \in \{l, r\}$ into two sub-lists $C_x(v_i, lt)$ and $C_x(v_i, lb)$ (if $region\_index = l$) and $C_x(v_i, rt)$ and $C_x(v_i, rb)$ (if $region\_index = r$), respectively. The $C_x(v_i, lt)$ and $C_x(v_i, lb)$ lists contain the particles of $C_x(v_i, l)$ that are stored in the left top and left bottom sub-box of $box(v_i)$, respectively. Analogue, the $C_x(v_i, rt)$ and $C_x(v_i, rb)$ list contain the particles of $C_x(v_i, r)$ that are stored in the right top and right bottom sub-box of $box(v_i)$, respectively. This splitting of a list $C_x(v_i, region\_index)$ with $region\_index \in \{l, r\}$ into two sub-lists can be done by traversing $C_x(v_i, region\_index)$ from left to right. Each list $C_y(v_i, region\_index)$ with $region\_index \in \{l, r\}$ is split into two sub-lists in an analogue way. Note that no cross references between the elements of the $S_x, S_y, C_x(v_i, region\_index)$ and $C_x(v_i, region\_index)$ lists are destroyed by these splitting operations. Table 5.7 shows the $C_x(v_i, region\_index)$ and $C_y(v_i, region\_index)$ lists of Table 5.6 after the splitting operations.

Now we have everything on hand to create the children of the nodes in P. This is done by creating for each list $C_x(v_i, region\_index)$ a child node w_i of v_i that corresponds to the

$C_x(v_1, lt)$	2	1
$C_y(v_1, lt)$	2	1

$C_x(v_2, lb)$	8
$C_y(v_2, lb)$	8

$C_x(v_2, lt)$	4
$C_x(v_2, lt)$	4

$C_x(v_3, lt)$	5
$C_x(v_3, lt)$	5

$C_x(v_1, rb)$	11	9	10
$C_y(v_1, rb)$	11	9	10

$C_x(v_1, rt)$	3
$C_x(v_1, rt)$	3

Table 5.7: The splat sorted lists of $C_x(v_i, \textit{region\_index})$ and $C_y(v_i, \textit{region\_index})$ that are obtained from the $C_x(v_i, \textit{region\_index})$ and $C_y(v_i, \textit{region\_index})$ lists of Table 5.6.

sub-box of $box(v_i)$ that is defined by index *region_index*. Furthermore, we assign w_i the lists $C_x(w_i) := C_x(v_i, \textit{region\_index})$ and $C_y(w_i) := C_y(v_i, \textit{region\_index})$ for later use.

If the list $C_x(w_i)$ has length at most l, then less than l particles are contained in this sub-box. Hence, w_i is a leaf in the desired reduced bucket quadtree, and all particles that are contained in $C_x(w_i)$ are assigned to w_i. If the list $C_x(w_i)$ has length larger than l (indicating that more than l particles are contained in this sub-box) the box-shrinking technique of part 7 is used to obtain the smallest sub-box B of $box(w_i)$, which covers the smallest rectangle R that contains the particles of $C_x(w_i)$. Then, we set $box(w_i) := B$.

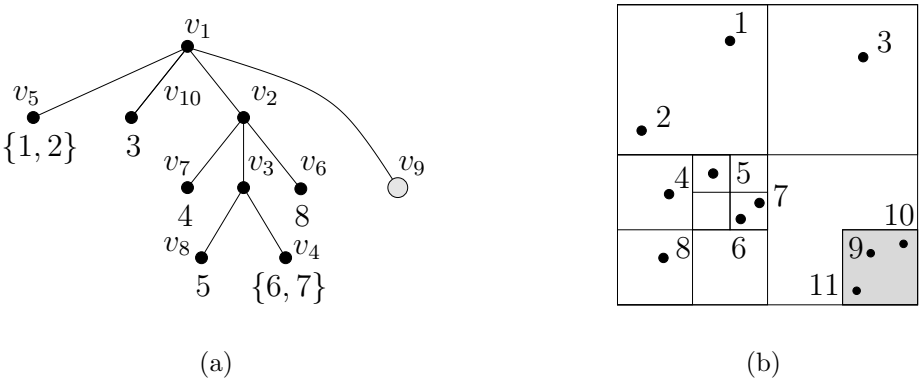


Figure 5.12: (a) The tree T after part 9 of the construction algorithm. (b) The corresponding sub-boxes of the square D. The node v_9 in (a) corresponds to the grey shaded region in (b).

Figure 5.12 demonstrates this process at an example: Suppose, the lists $C_x(v_1, lt)$, $C_x(v_2,\ lb)$, $C_x(v_2, lt)$, $C_x(v_3, lt)$, $C_x(v_1, rb)$, and $C_x(v_1, rt)$ of Table 5.7 are visited in consecutive order. Then, a child v_5 of v_1, two children v_6 and v_7 of v_2, a child v_8 of v_3, and two additional children v_9 and v_{10} of v_1 are created. Since the lists $C_x(v_5) := C_x(v_1, lt)$, $C_x(v_6) := C_x(v_2, lb)$, $C_x(v_7) := C_x(v_2, lt)$, $C_x(v_8) := C_x(v_3, lt)$, and $C_x(v_{10}) := C_x(v_1, rt)$ have length at most $l = 2$, the particles that are contained in these lists are assigned to the corresponding new leaves of T. In contrast to this, the list $C_x(v_9) := C_x(v_1, rb)$ in Table 5.7 has length $3 > l$. Hence, the smallest sub-box of $box(v_9)$, which covers the tight rectangle R that contains the particles of $C_x(w_i)$ (namely $\{9, 10, 11\}$) is created using the box-shrinking technique of part 7.

Part 10: Recursion

If (as a result of part 9) all lists $C_x(w_i)$ have length at most l, the resulting tree is already the desired reduced bucket quadtree with leaf capacity l. Otherwise, let $\{w_1, \ldots, w_k\}$ be the set of nodes for which the corresponding lists $C_x(w_i)$ have length larger than l. Then, parts 4 to 8 of the algorithm are applied on each node w_i and its corresponding lists $C_x(w_i)$ and $C_y(w_i)$. As a results of this process, paths $\{P_1, \ldots, P_k\}$ that are rooted at $\{w_1, \ldots, w_k\}$ are generated. In order to create all children of the nodes to which links in the S_x lists are stored, part 9 is performed. Note that the set of these nodes are exactly the nodes contained in $\{P_1, \ldots, P_k\}$. Let $\{u_1, \ldots, u_m\}$ be the set of the newly created children for which the length of $C_x(u_i)$ is larger than l. Then, parts 4 to 8 of the algorithm are applied on each node u_i, followed by one call of part 9 and so forth. The recursion ends if for all newly created children the corresponding C_x and C_y lists have length at most l.

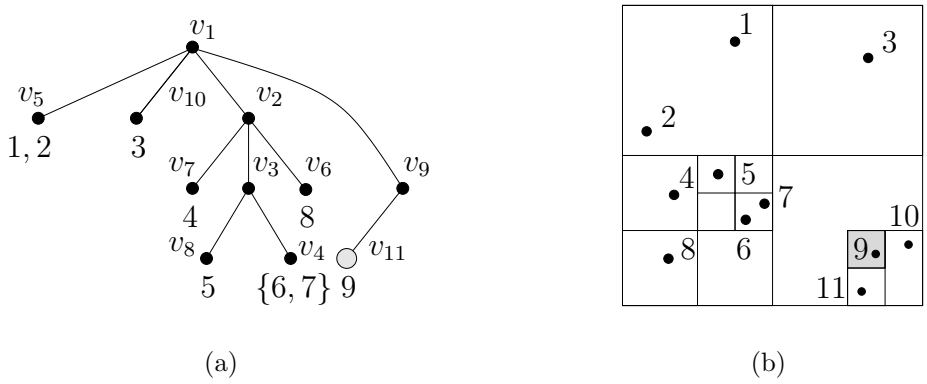


Figure 5.13: Recursion: (a) Creating a new child node v_{11} of v_9. (b) The grey shaded region that corresponds to v_{11}.

For example, the list $C_x(v_9) = C_x(v_1, rb)$ in Table 5.7 that corresponds to the particles in the grey shaded region of Figure 5.12(b) and to node v_9 of Figure 5.12(a) has length $3 > l$. This is the only C_x list of Table 5.7 that has length larger than l. Hence, parts 4 to 8 of the algorithm are applied to v_9 with lists $C_x(v_9)$ and $C_y(v_9)$. As a result of the alternating $C_x(v_9)$ traversal and the alternating $C_y(v_9)$ traversal in parts 4 and 5, a new child v_{11} is created in part 6 (see Figure 5.13(a)) that corresponds to the grey shaded box in Figure 5.13(b). Since $box(v_{11})$ contains only one particle, part 7 is skipped, and the particle 9 is assigned to v_{11}. Hence, the newly created path is $P = (v_9, v_{11})$. Since no other lists of Table 5.7 have length larger than l, the algorithm proceeds with part 9 and creates all other children of nodes to which links in S_x are stored. These children are v_{12} and v_{13}. The corresponding lists $C_x(v_{12})$ and $C_x(v_{13})$ both have length $1 < l$. Therefore, the particles 10 and 11 that are stored in $C_x(v_{12})$ and $C_x(v_{13})$ are assigned to v_{12} and v_{13}, respectively. Since no new children v_i with length $C_x(v_i) > l$ have been created, the algorithm terminates, and the resulting tree is the reduced bucket quadtree with leaf capacity $l = 2$ (see Figure 5.14).

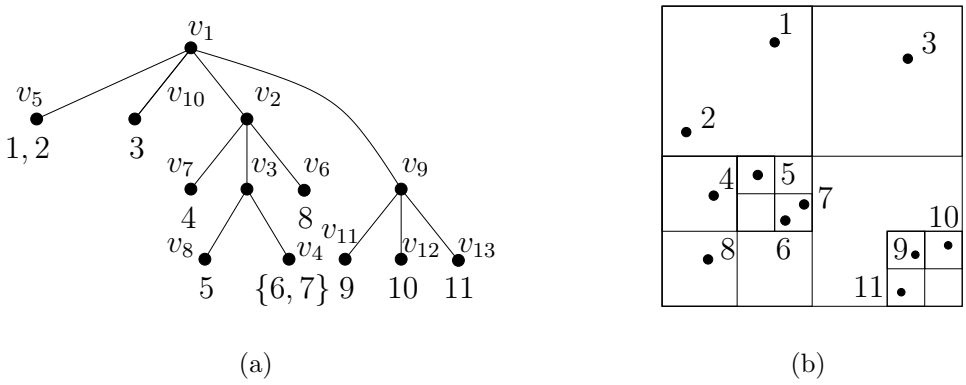


Figure 5.14: Recursion: (a) Creating the other children v_{11} and v_{12} of v_9. (b) The subdivision of D that corresponds to the tree T in (a). T is the desired reduced bucket quadtree with leaf capacity 2.

Formal Description of the Tree Construction Algorithm (Way A)

In Function $\texttt{TC}_\texttt{a}$we present the pseudocode of the tree construction algorithm. Furthermore, we will analyze its running time. An experimental study of this tree construction method, including comparisons with other tree construction methods, will be given in Section 7.4.1.

Lemma 5.10 (Reduction of the Number of Particles in Leaves). *Suppose, the previously described tree construction algorithm is used to construct a reduced bucket quadtree with leaf capacity l. Let v_{act} be a node that corresponds to a box box(v_{act}), the lists $C_x(v_{act})$ and $C_y(v_{act})$ are associated with v_{act}, and box(v_{act}) contains $|C_x(v_{act})|$ particles. Furthermore, suppose, the parts 4 to 9 of the tree construction algorithm have to be performed on v_{act} and that the leaves of the newly created subtree that is rooted at v_{act} are denoted by $\{v_1, \ldots, v_k\}$. Then, for each leaf $v_i \in \{v_1, \ldots, v_k\}$ the corresponding box box(v_i) contains at most* $\max\{l, \lfloor |C_x(v_{\text{act}})|/2 \rfloor\}$ *particles.*

Proof. It is clear that after performing parts 4 to 8 of the algorithm, a path P has been created that contains one leaf v_i, and the corresponding region $box(v_i)$ contains at most l particles. In part 9 the other children of the nodes of P (that are leaves of the newly created subtree that is rooted at v_{act}) are created. It is clear that for $v_i \in \{v_1, \ldots, v_k\}$ the $C_x(v_i)$ and $C_y(v_i)$ lists have length at most $max(l, \lfloor |C_x(v_{act})|/2 \rfloor)$ (we assign non-empty links only to those elements of S_x and S_y, which contain particles that are covered by the half of two alternative sub-regions that contains the fewest particles). The length of each such list $C_x(v_i)$ corresponds to the number of particles that are covered by the sub-box of the leaf v_i, which completes the proof. □

Theorem 5.11 (Tree Construction Way A). *Suppose, $C = \{c_1, \ldots, c_N\}$ is a set of particles that are placed at distinct positions $p(C) = \{p_1, \ldots, p_N\}$ and $l \geq 1$ is an integer constant. Then, Function $\texttt{TC}_\texttt{a}$ constructs a reduced bucked quadtree with leaf capacity l in $O(N \log N)$ time using $O(N)$ memory.*

Function $\texttt{TC}_\texttt{a}(C, l)$

input : a set $C = \{c_1, \ldots, c_N\}$ of particles that are placed at distinct positions $p(C) = \{p_1, \ldots, p_N\}$ and a constant integer leaf capacity $l \geq 1$

output: a reduced bucket quadtree T with leaf capacity l

begin

- $i \leftarrow 1$;
- calculate a square region D that contains all particles of C;
- create sorted coordinate lists S_x and S_y;
- create a root node v_{act} and the lists $C_x(v_{act})$ and $C_y(v_{act})$;
- $T \leftarrow v_{act}$;
- **if** $N \leq l$ **then** assign all particles in $C_x(v_{act})$ to v_{act} and `EXIT`;
- **while** $box(v_{act})$ contains more than l particles **do**
  - `Alternating_`$C_x(v_{act})$`_Traversal`;
  - `Alternating_`$C_y(v_{act})$`_Traversal`;
  - create a new child node v_{child} of v_{act};
  - **if** more than l particles are contained in $box(v_{child})$ **then**
    - $box(v_{child}) \leftarrow \texttt{Shrink}(box(v_{child}))$;
  - $v_{act} \leftarrow v_{child}$;
- assign all particles in $C_x(v_{act})$ to v_{act};
- Create new children $\{v_1, \ldots, v_s\} =: M^i$ of all nodes to which links are stored in S_x;
- let $L^i \leftarrow \{v_1, \ldots, v_k\}$ be the set of the nodes in M^i so that for each $i \in \{1, \ldots, k\}$ $box(v_i)$ contains more than l particles;
- **while** $L^i \neq \emptyset$ **do**
  - **while** $L^i \neq \emptyset$ **do**
    - pick some $v_{act} \in L^i$, and set $L^i \leftarrow L^i \setminus \{v_{act}\}$;
    - **while** $box(v_{act})$ contains more than l particles **do**
      - `Alternating_`$C_x(v_{act})$`_Traversal`;
      - `Alternating_`$C_y(v_{act})$`_Traversal`;
      - create a new child node v_{child} of v_{act};
      - **if** more than l particles are contained in $box(v_{child})$ **then**
        - $box(v_{child}) \leftarrow \texttt{Shrink}(box(v_{child}))$;
      - $v_{act} \leftarrow v_{child}$;
    - assign all particles in $C_x(v_{act})$ to v_{act};
  - Create new children $\{v_1, \ldots, v_s\} =: M^{i+1}$ of all nodes to which links are stored in S_x;
  - let $L^{i+1} \leftarrow \{v_1, \ldots, v_k\}$ be the set of the nodes in M^{i+1} so that for each $i \in \{1, \ldots, k\}$ $box(v_i)$ contains more than l particles;
  - $i \leftarrow i + 1$;

end

Proof. The calculation of the square D (part 1) needs $O(N)$ time. The construction of the sorted lists S_x, S_y of length N with cross references (part 2) needs $O(N \log N)$ time and $O(N)$ additional memory is needed. The construction of the C_x and C_y lists with cross references of the root node (part 3) needs $O(N)$ time and $O(N)$ additional memory. If $N < l$ we are already done, and the reduced bucket quadtree T is given by the root node.

We concentrate on the running time of the parts 4 to 8 (i.e., the top while-loop in Function $\texttt{TC}_\texttt{a}$) now: Let v_{act} be the actual visited node. Then, the alternating $C_x(v_{act})$ traversal (part 4) needs time that is proportional to the number of particles in the half of $box(v_{act})$ that contains the fewest particles (due to the cross references). Since in S_x and S_y all elements are marked that contain these particles, the running time is proportional to the number of elements in S_x for which non-empty links to nodes of T have been created, too. We denote the half of $box(v_{act})$ that contains the most particles by $large\_half(v_{act})$. Analogue, the alternating $C_y(v_{act})$ traversal (part 5) needs time that is proportional to the number of particles in the half of $large\_half(v_{act})$ that contains the fewest particles. Since in S_x and S_y all elements are marked that contain these particles, the total running time of parts 4 and 5 is proportional to the number of elements in S_x for which non-empty links to nodes of T exist.

The creation of a child v_{child} of v_{act} and the relabeling of $C_x(v_{act})$ and $C_y(v_{act})$ to $C_x(v_{child})$ and $C_y(v_{child})$, respectively, in part 6 need $O(1)$ time. If the length of $C_x(v_{child})$ is at most l, the contained particles are assigned to v_{child}, which needs $O(l) = O(1)$ time. Otherwise, possibly the box $box(v_{child})$ has to be shrunk (part 7). Since the lists $C_x(v_{child})$ and $C_x(v_{child})$ are ordered, the smallest rectangle R, which covers the particles that are contained in $box(v_{child})$, can be found in $O(1)$ time. Using the box-shrinking technique of Aluru et al. [2], the shrinking of $box(v_{child})$ needs $O(1)$ time in total. We can summarize that the total running time of parts 4 to 7 is proportional to the number of elements in S_x for which nonempty links to nodes of T have been created. Therefore, the running time of the iterative path construction (part 8) is bounded above by the initial length of $C_x(v_{act})$. Since v_{act} is the root of T, the initial length of $C_x(v_{act})$ has been N.

In part 9 the remaining children of the nodes in T to which links in S_x exist are created in $O(N)$ time, since S_x and S_y have length N, since the sum of the length of all $C_x(v_i, region\_index)$ and $C_y(v_i, region\_index)$ lists after the splitting process is $O(N)$, and due to the cross references.

Let $L^1\{v_1, \ldots, v_k\}$ be the set of all leaves that have been created in part 9 and that contain more than l particles. If $L^1 = \emptyset$, the reduced bucked quadtree has been constructed in $O(N \log N)$ time using $O(N)$ memory. Otherwise, we analyze the running time of one recursion of part 10 (i.e., one iteration of the outer while-loop in Function $\texttt{TC}_\texttt{a}$): Let v_i be a node of L^1 with associated lists $C_x(v_i)$ and $C_y(v_i)$. We have already discussed that parts 4 to 8 need time that is proportional to the initial length of the list $C_x(v_i)$. Since $\sum_{i=1}^{k} initial\_length(C_x(v_i)) \leq N$, the execution of parts 4 to 8 for all nodes $v_i \in L^1$ needs $O(N)$ time in total. Afterwards, the children of all nodes to which links are stored in S_x have to be created. This needs $O(N)$ time by traversing S_x and S_y once and then, performing the needed splitting operations. Hence, the running time of one recursion of part 10 is $O(N)$, too.

We can summarize that the total running time of the algorithm is $O(N \log N + N \cdot |\mathit{recursion\_levels}|)$, where *recursion_levels* is the number of recursions (i.e., the number of iterations of the outer while-loop in Function $\texttt{TC}_\texttt{a}$). We estimate the number of recursion levels next: It is known from Lemma 5.10 that after the first execution of parts 4 to 9 of the algorithm on the root node v_r, the number of particles that is covered by a box that is associated with a leaf is at most $max\{l, \lfloor N/2 \rfloor\}$. Hence, It follows from Lemma 5.10 that after applying parts 4 to 9 on all nodes in L^1, the number of particles covered by the box of each leaf is at most $max\{l, \lfloor N/4 \rfloor\}$. After the next recursion level the number of particles covered by the box of each leaf is at most $max\{l, \lfloor N/8 \rfloor\}$ and so forth. Therefore, the number of recursion levels is bounded above by $O(\log N)$, and the total running time of Function $\texttt{TC}_\texttt{a}$ is $O(N \log N)$.

Since at each stage of the algorithm, the sum of the length of all lists is $O(N)$ and since the actual constructed tree contains $O(N)$ nodes, the memory requirements are $O(N)$, too. □

5.3.3 Construction of the Reduced Bucket Quadtree (Way B)

In the last section we have presented a method for constructing a reduced bucket quadtree with fixed constant leaf capacity $l \geq 1$ in $O(N \log N)$ time. However, like the tree construction method of Aluru et al. [2] it has some drawbacks: The method is quite complicated, the best-case running time is not better than $O(N \log N)$, and the construction time still could be improved in practice (see Section 7.4.1). Hence, we developed an alternative simpler algorithm for constructing reduced bucket quadtrees that has a linear best-case running time, and is faster than Function $\texttt{TC}_\texttt{a}$ in practice (see Section 7.4.1). It is described in the following.

Like in the previous section, we suppose that the N particles $C = \{c_1, \ldots, c_N\}$ are distributed at distinct positions $p(C) = \{p_1, \ldots, p_N\}$. Furthermore, we suppose that a fixed constant leaf capacity $l \geq 1$ is given and that we want to construct a reduced bucket quadtree with leaf capacity l. We will illustrate the algorithm at an example in which we set the leaf capacity l to 2.

Part 1: Initialize the square D

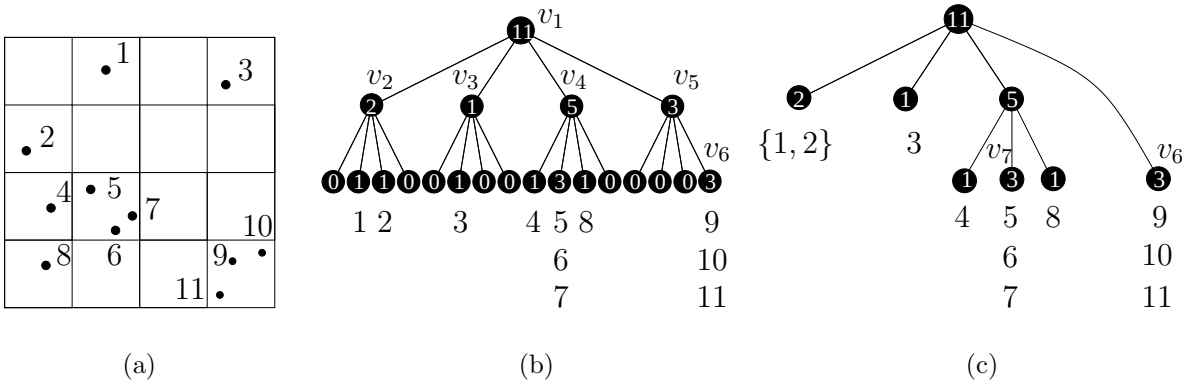
First, we have to define a small square D that covers the particles in C. This can be done exactly like in part 1 of the previously described tree construction Function $\texttt{TC}_\texttt{a}$.

Part 2: Building up a Truncated Pseudo Quadtree T^1

Then, we build up a truncated pseudo quadtree T^1 of depth $d = max\{1, \lfloor \log_4 N/l \rfloor\}$ by using the same technique as described in Section 5.2.1.

Part 3: Transfer T^1 to a Reduced Truncated Bucket Quadtree

In the next part of the algorithm we thin out T^1 in order to transfer it into a reduced truncated bucket quadtree with leaf capacity l and maximal depth d. This is done by traversing the tree bottom up and thereby calculating for each tree node the number of particles that are contained in the box that it represents. Figure 5.15(b) shows the tree T^1 after the bottom-up traversal. It has been constructed for the distribution of $N = 11$ particles shown in Figures 5.15(a).



(a) (b) (c)

Figure 5.15: (a) A distribution of $N = 11$ particles. (b) A truncated pseudo quadtree T^1 of depth 2 that corresponds to this distribution. The labels of the nodes display the number of particles that are contained in the corresponding sub-boxes. (c) The reduced truncated bucket quadtree with leaf capacity 2 and maximal depth 2 that is obtained by thinning out T^1.

Then, T^1 is traversed top down starting at the root of T^1. Suppose, the actual visited node is v, its parent is u and its children are w_1, w_2, w_3, and w_4. If v is a leaf of T^1, nothing is done. Otherwise, each subtree that is rooted at a child w_i of v and contains no particles is deleted from T^1. If after this deletion process v has only one nonempty child w_i (i.e., v is a degenerate node of T^1) and if additionally v is the root of a subtree that contains more than l particles, v is deleted from T^1, and w_i is made a child of u. If v is the root of a subtree that contains at most l particles, all subtrees that are rooted at the children of v are deleted, and all particles that were stored in the leaves of the deleted subtrees are assigned to v.

In our example, after visiting node v_1 of Figure 5.15(b), the algorithm proceeds to node v_2 and deletes the two subtrees that are rooted at the children of v_2 and which contain no particles. Since v_2 is the root of a subtree that contains $2 = l$ particles, the two remaining subtrees that are rooted at the children of v_2 are deleted, and the particles 1 and 2 are assigned to v_2. Then, the algorithm proceeds to node v_3 and deletes the three subtrees that are rooted at the children of v_3 and which contain no particles. Since the subtree that is rooted at the remaining child of v_3 contains only $1 < l$ particle, this subtree of v_3 is deleted, and the particle 3 is assigned to v_3. Then, the algorithm visits node v_4 and deletes the subtree that it rooted at the child of v_4 which contains no particles. Since the

remaining subtree that is rooted at v_4 contains $5 > l$ particles, the three children of v_4 are visited next. Afterward, node v_5 is visited, and the three subtrees that are rooted at the children of v_5 and which contain no particles are deleted. Since v_5 is a degenerate node with the unique nonempty child v_6, and the remaining subtree that is rooted at v_5 contains $3 > l$ particles, v_5 is deleted, and v_1 is made the new parent of v_6. After visiting node v_6, the reduced truncated bucket quadtree with leaf capacity $l = 2$ and maximal depth 2 is constructed (see Figure 5.15(c)).

Part 4: Recursions

If none of the leaves of T^1 contains more than l particles, the algorithm ends, and T^1 is already a reduced bucket quadtree with leaf capacity l. Otherwise, the previous described parts 2 and 3 are repeated recursively for each leaf of T^1 that contains more than l particles.

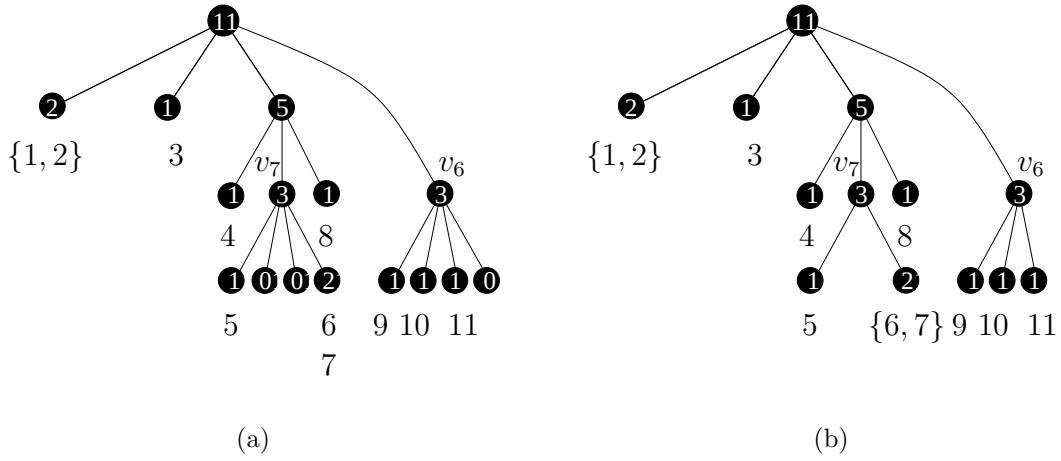


Figure 5.16: (a) Recursive construction of truncated pseudo quadtrees of depth 1 rooted at nodes v_6 and v_7. (b) The resulting reduced bucket quadtree with leaf capacity 2 that is obtained after thinning out the subtrees which are rooted at v_6 and v_7.

For example, the leaves v_6 and v_7 in Figure 5.15(c) both contain $3 > l$ particles. Therefore, truncated pseudo quadtrees $T^2(v_6)$ rooted at v_6 and $T^2(v_7)$ rooted at v_7 are built up. Both subtrees have depth $d := max\{1, \lfloor \log_4 3/l \rfloor\} = 1$. Then, in a bottom-up traversal of $T^2(v_6)$ and $T^2(v_7)$ the number of contained particles in the corresponding subboxes is calculated (see Figure 5.16(a)). After thinning out $T^2(v_6)$ and $T^2(v_7)$, all leaves in the resulting tree contain at most l particles. Hence, the algorithm terminates, and the resulting tree is a reduced bucket quadtree with leaf capacity l (see Figure 5.16(b)).

Formal Description of the Tree Construction Algorithm (Way B)

The tree construction algorithm is formalized in Function $\texttt{TC}_\texttt{b}$. Besides an analysis of its asymptotic running time that we will present here, an experimental study of this tree

construction method, including comparisons with other tree construction methods, will be given in Section 7.4.1.

Function $\texttt{TC}_\texttt{b}(C, l)$

input : a set $C = \{c_1, \ldots, c_N\}$ of particles that are placed at distinct positions $p(C) = \{p_1, \ldots, p_N\}$ and a constant integer leaf capacity $l \geq 1$

output: a reduced bucket quadtree T with leaf capacity l

begin

$i \leftarrow 1$;
$d \leftarrow max\{1, \lfloor \log_4 N/l \rfloor\}$;
calculate a square region D that contains all particles of C;
$T^i \leftarrow$ `Create_Truncated_Pseudo_Quadtree`(C, d);
$T^i \leftarrow$ `Transfer_To_Reduced_Truncated_Bucket_Quadtree`(T^i);
let $L^i \leftarrow \{v_1, \ldots, v_k\}$ denote the set of the leaves of T^i that contain more than l particles;
while $L^i \neq \emptyset$ **do**
  if L^{i+1} has not been created **then** $L^{i+1} \leftarrow \emptyset$;
  pick some $v \in L^i$, and set $L^i \leftarrow L^i \setminus \{v\}$;
  let $C(v)$ be the set of particles that are contained in v;
  $d_v \leftarrow max\{1, \lfloor \log_4 |C(v)|/l \rfloor\}$;
  $T^{i+1}(v) \leftarrow$ `Create_Truncated_Pseudo_Quadtree`$(C(v), d)$;
  $T^{i+1}(v) \leftarrow$ `Transfer_To_Reduced_Truncated_Bucket_Quadtree`$(T^{i+1}(v))$;
  let $L^{i+1}(v) \leftarrow \{v_1, \ldots, v_k\}$ denote the set of the leaves of $T^{i+1}(v)$ that contain more than l particles.;
  $L^{i+1} \leftarrow L^{i+1} \cup L^{i+1}(v)$;
  if $L^i = \emptyset$ **then** $i \leftarrow i + 1$;
$T \leftarrow T^1$;

end

Lemma 5.12 (Construction of a Truncated Reduced Bucket Quadtree). *Suppose, $C = \{c_1, \ldots, c_N\}$ is a set of particles that are placed at distinct positions $p(C) = \{p_1, \ldots, p_N\}$ in a square region D, and $l, d \geq 1$ are integers. Then, a truncated reduced bucket quadtree with leaf capacity l and maximal depth d can be constructed in $O(N + 4^d)$ time using $O(N + 4^d)$ memory.*

Proof. We use parts 2 and 3 of the previously described tree construction algorithm: The construction of a truncated pseudo quadtree T^1 of depth d needs $O(N + 4^d)$ time (see Section 5.2.1). Since T^1 contains $O(4^d)$ nodes and N particles, the memory requirements are $O(N + 4^d)$, too. T^1 can be thinned out in $O(4^d)$ time, since every node of T^1 is visited only once in the bottom-up traversal and only once in the top-down traversal. Hence, a truncated reduced bucket quadtree with leaf capacity l and maximal depth d can be constructed in $O(N + 4^d)$ time using $O(N + 4^d)$ memory. □

Theorem 5.13 (Tree Construction Way B (General Case)). *Suppose that* $C = \{c_1, \ldots, c_N\}$ *is a set of particles that are placed at distinct positions* $p(C) = \{p_1, \ldots, p_N\}$ *and that* $l \geq 1$ *is an integer constant. Suppose that the maximum and minimum of the Euclidean distances between any two particles in* C *are denoted by* d_{max} *and* d_{min}*, respectively. Then, Function* `TC_b` *constructs a reduced bucked quadtree with leaf capacity* l *in* $O(N \cdot \log(d_{max}/d_{min}))$ *time in the worst case and uses* $O(N)$ *memory. The best-case running time of Function* `TC_b` *is* $O(N)$*.*

Proof. The calculation of D needs $O(N)$ time. The construction of the truncated pseudo quadtree T^1 with leaf capacity l and maximal depth $d = max\{1, \lfloor \log_4 N/l \rfloor\}$ needs $O(N + 4^d) = O(N)$ time and uses $O(N+4^d) = O(N)$ memory (see Lemma 5.12). If all leaves of T^1 contain at most l particles, T^1 is already the reduced bucket quadtree and the construction time has been $O(N)$.

Otherwise, the algorithm builds up truncated reduced bucket quadtrees $T^1(v_1), \ldots, T^1(v_k)$ of all leaves $v_1, \ldots, v_k$ of T^1 that contain more than l particles. For each $j \in \{1, \ldots, k\}$ let $C(v_j)$ denote the set of all particles that are contained in the subtree of node v_j. Then, by using Lemma 5.12, the total running time for constructing all $T^1(v_j)$ is bounded above by $\sum_{j=1}^{k} O(|C(v_j)| + 4^{\log(|C(v_j)|)}) = O(N)$. For each other recursion levels $i \geq 2$, the running time for building up all truncated reduced bucket quadtrees $T^i(v_1), \ldots, T^i(v_k)$ is bounded above by $O(N)$, too. Hence, the running time of Function `TC_b` is bounded above by $O(N \cdot |\mathit{recursion\_levels}|)$, where *recursion_levels* corresponds to the number of iterations of the while-loop in Function `TC_b`.

Clearly, the number of recursion levels depends on the distribution $p(C)$ of the particles in C. Now we will prove that the number of recursion levels is bounded above by $\log(d_{max}/d_{min})$: Since the width of the four sub-boxes of a box B is half the width of B, it follows that after $O(\log_2(d_{max}/d_{min}))$ recursive subdivisions of the square D no sub-box contains more than one particle.

Suppose, node v is a leaf of subtree $T^i(u)$ that contains more than l particles and corresponds to a sub-square of D of width $width(v)$. Then, by construction of our algorithm, first a truncated pseudo quadtree $T^{i+1}(v)$ rooted at v is built up that has depth at least one. Thus, all leaves of $T^{i+1}(v)$ correspond to sub-squares of D with width at most $width(v)/2$. Possibly some of these leaves (denoted by $\{w_1, \ldots, w_k\}$) contain more than l particles and the corresponding boxes will be subdivided further at the next recursion level. At this recursion level, the leaves of the constructed subtrees $T^{i+2}(w_1), \ldots, T^{i+2}(w_k)$ have width at most $width(v)/4$ and so forth. Hence, the maximum number of recursion levels is bounded above by $O(\log_2(d_{max}/d_{min}))$.

Since at each stage of the algorithm, the number of nodes in the actual tree is $O(N)$, the total memory requirements are $O(N)$. □

Note that the simplicity and linear best-case running time of Function `TC_b` has a price: Its worst-case running time depends on the distribution of the particles, which is undesirable. However, we will see that this function in combination with the following corollary will be very useful to describe an efficient force-calculation step in Section 5.4.

Corollary 5.14 (Tree Construction Way B (Special Case)). *Suppose that* $C = \{c_1, \ldots, c_N\}$ *is a set of particles that are placed at distinct positions* $p(C) = \{p_1, \ldots, p_N\}$ *on a regular square integer grid with a resolution which is polynomial in* N*, and* $l \geq 1$ *is an integer constant. Then, Function* `TC_b` *constructs a reduced bucked quadtree with leaf capacity* l *in* $O(N \log N)$ *worst-case running time using* $O(N)$ *memory. The best-case running time of Function* `TC_b` *is* $O(N)$*.*

Proof. We can assume that the particles are placed on an integer grid with x- and y-coordinates in the range $[-\mathcal{P}(N), \mathcal{P}(N)]$, where $\mathcal{P}(N)$ is a whole-numbered polynomial in N of constant maximum degree s with positive coefficients. Then, the maximum Euclidean distance between any two particles in C is bounded above by $2\sqrt{2} \cdot \mathcal{P}(N)$. The minimum Euclidean distance between any two particles in C is bounded above by one. Hence, the tree construction time is bounded above by $O(N \cdot \log(2\sqrt{2} \cdot \mathcal{P}(N)) = O(N \cdot s \cdot \log N) = O(N \log N)$. □

5.3.4 The Multipole Framework

In this section we will concentrate on the second phase of our force-approximation method `NM²`. We will introduce the analytical tools from complex analysis and some terminology, before we will give an informal and a formal description of the second phase of our force-approximation method.

Like in the previous sections, we suppose that $C = \{c_1, \ldots, c_N\}$ is a set of N charged particles with charges $q(C) = \{q_1 = 1, \ldots, q_N = 1\}$ that are located at distinct positions $p(C) = \{p_1, \ldots, p_N\} \in \mathbb{R}^2$. Furthermore, we suppose that a reduced bucket quadtree $T = (V, E)$ with fixed constant leaf capacity l has been built up for the particles in C.

Tools From Complex Analysis

In the following, we will present some definitions and theorems that have been invented by Greengard [58] and Greengard and Rokhlin [59] and that are of fundamental importance for multipole methods.

Suppose, in a two-dimensional physical model a charged particle c_i of charge q_i is located at point p_i and $(x, y) \neq p_i$ is an arbitrary point in $\mathbb{R}^2$. Then, the potential energy $E^{c_i}(x, y)$ at a point (x, y) due to c_i and the repulsive force $F^{c_i}_{rep}(x, y)$ that acts on a particle of unit charge that is placed at point (x, y) due to c_i are given by

$$E^{c_i}(x, y) = -q_i \cdot \log(\|(x, y) - p_i\|) \quad \text{and} \quad F^{c_i}_{rep}(x, y) = \frac{q_i \cdot ((x, y) - p_i)}{\|(x, y) - p_i\|^2} ,$$

respectively (see [58, 59]). In order to obtain useful tools for their `Fast_Multipole_Method`, Greengard [58] and Greengard and Rokhlin [59] identify each point $(x, y) \in \mathbb{R}^2$ with a point $z = x + iy \in \mathbb{C}$. Following standard practice, they refer the complex function $\mathcal{E}^{c_i}(z) = q_i \log(z - p_i)$ as the potential energy function due to a charged particle c_i of charge q_i. Furthermore, the following lemma holds (see [58, 59]).

Lemma 5.15 (Obtaining the Forces from a Complex Energy Function). *If* $E(x,y) = Re(\mathcal{E}(z))$ *describes the potential energy field at a point* $(x,y) \in \mathbb{R}^2$ *and* $z := x + iy \in \mathbb{C}$, *then the corresponding force field is given by* $F_{rep}(x,y) = (Re(\mathcal{E}'(z)), -Im(\mathcal{E}'(z)))$.

Only special cases of the following Lemmas 5.16, 5.20, 5.21, 5.23, and Theorem 5.17 have been shown in [58, 59]. In particular, the corresponding lemmas in [58, 59] assume that either z_0 or z_1 is the origin. Since we need the theorems in [58, 59] in the general context and since not all proofs of the generalized versions are trivial, we reproved the lemmas for the general case, in which z_0 and z_1 are arbitrary points in $\mathbb{C}$. As a benefit of this, we can apply these lemmas and theorems in our algorithm directly. Otherwise, we would have to perform time-consuming translation operations on the set of particles in order to apply the restricted lemmas and theorems given in [58, 59] in our algorithm.

Lemma 5.16 (Potential Field of a Charged Particle). *Suppose, a particle* c_i *of charge* q_i *is located at point* $p_i \in \mathbb{C}$, *and* z_0 *is an arbitrary point in* $\mathbb{C}$. *Then, for any* $z \in \mathbb{C}$ *so that* $|z - z_0| > |p_i - z_0|$ *the potential energy at point* z *due to particle* c_i *is given by*

$$\mathcal{E}^{c_i}(z) = q_i \log(z - p_i) = q_i \left(\log(z - z_0) - \sum_{k=1}^{\infty} \frac{(p_i - z_0)^k}{k \cdot (z - z_0)^k} \right).$$

Proof.

$$\log(z - p_i) - \log(z - z_0) = \log\left(\frac{z - p_i}{z - z_0}\right) = \log\left(1 - \frac{p_i - z_0}{z - z_0}\right) = -\sum_{k=1}^{\infty} \left(\frac{p_i - z_0}{z - z_0}\right)^k / k$$

The last equation is obtained by using the following expansion of the complex logarithm $\log(1 - w) = -\sum_{k=1}^{\infty} \frac{w^k}{k}$ that holds for any $w \in \mathbb{C}$ with $|w| < 1$. □

Theorem 5.17 (Multipole Expansion Theorem). *Suppose that* m *charged particles* $\{c_1, \ldots, c_m\}$ *with charges* $\{q_1, \ldots, q_m\}$ *are located at points* $\{p_1, \ldots, p_m\}$ *inside a circle of radius* r *with center* z_0. *Then, for any* $z \in \mathbb{C}$ *with* $|z - z_0| > r$ *the potential energy* $\mathcal{E}(z)$ *at point* z *induced by the* m *charged particles is given by*

$$\mathcal{E}(z) = a_0 \log(z - z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(z - z_0)^k}, \text{ where} \tag{5.7}$$

$$a_0 = \sum_{i=1}^{m} q_i \quad \text{and} \quad a_k = \sum_{i=1}^{m} \frac{-q_i (p_i - z_0)^k}{k}. \tag{5.8}$$

Furthermore, for any $p \geq 1$

$$\left| \mathcal{E}(z) - a_0 \log(z - z_0) - \sum_{k=1}^{p} \frac{a_k}{(z - z_0)^k} \right| \leq \left(\frac{A}{c - 1}\right) \left(\frac{1}{c}\right)^p, \text{ where} \tag{5.9}$$

$$c = \left| \frac{z - z_0}{r} \right| \quad \text{and} \quad A = \sum_{i=1}^{m} |q_i|.$$

Proof.

$$\begin{aligned}
\mathcal{E}(z) &= \sum_{i=1}^{m} \mathcal{E}^{c_i}(z) = \sum_{i=1}^{m} q_i \log(z - p_i) = \sum_{i=1}^{m} q_i \left(\log(z - z_0) - \sum_{k=1}^{\infty} \frac{(p_i - z_0)^k}{k \cdot (z - z_0)^k} \right) \\
&= \sum_{i=1}^{m} (q_i \log(z - z_0)) - \sum_{i=1}^{m} \left(q_i \sum_{k=1}^{\infty} \frac{(p_i - z_0)^k}{k \cdot (z - z_0)^k} \right) \\
&= \left(\sum_{i=1}^{m} q_i \right) \log(z - z_0) + \sum_{k=1}^{\infty} \left(\sum_{i=1}^{m} \frac{-q_i (p_i - z_0)^k}{k \cdot (z - z_0)^k} \right)
\end{aligned}$$

Estimation (5.9) is given by

$$\begin{aligned}
&\left| \mathcal{E}(z) - a_0 \log(z - z_0) - \sum_{k=1}^{p} \frac{a_k}{(z - z_0)^k} \right| = \left| \sum_{k=p+1}^{\infty} \sum_{i=1}^{m} \frac{-q_i (p_i - z_0)^k}{k \cdot (z - z_0)^k} \right| \leq \sum_{k=p+1}^{\infty} \frac{A \cdot r^k}{k \cdot |z - z_0|^k} \\
&\leq A \sum_{k=p+1}^{\infty} \left| \frac{r}{z - z_0} \right|^k = \frac{A}{1 - \left| \frac{r}{z - z_0} \right|} \cdot \left| \frac{r}{z - z_0} \right|^{p+1} = \frac{A}{1 - \frac{1}{c}} \cdot \left(\frac{1}{c} \right)^{p+1} = \left(\frac{A}{c - 1} \right) \left(\frac{1}{c} \right)^{p}.
\end{aligned}$$

□

Definition 5.18 (Multipole Expansion, p-term Multipole Expansion). *The infinite Laurent series in Formula* (5.7) *is called* multipole expansion of the potential energy field due to the charged particles in $\{c_1, \ldots, c_m\}$. *The variables a_i in Formula* (5.8) *are the* coefficients of the multipole expansion. *The finite series $M^p(z) = a_0 \log(z - z_0) + \sum_{k=1}^{p} \frac{a_k}{(z - z_0)^k}$ that is obtained by calculating only the coefficients $a_0, \ldots, a_p$ in Formula* (5.7) *is called* p-term multipole expansion of the potential energy due to the charged particles in $\{c_1, \ldots, c_m\}$, *and p is the* precision parameter.

Remark 5.19 (Error Estimation and Relationship Between Multipole Expansions and Center-of-Mass Approximations).

(a) The Estimation (5.9) implies that the accuracy of the approximation of the potential energy $\mathcal{E}(z)$ due to a p-term multipole expansion $M^p(z)$ increases with increasing precision parameter p and increasing distance between the center z_0 and the point z.

(b) If in Theorem 5.17 one chooses $p = 1$ and z_0 as the center of mass c of the charged particles $\{c_1, \ldots, c_m\}$, Formula (5.7) simplifies to $\mathcal{E}(z) = a_0 \log(z - z_0)$. Hence, the corresponding force that acts on a particle of unit charge that is placed at a position $z = (x, y) \in \mathbb{R}^2$ with $|z - c| > r$ is given by $F_{rep}(z) = (Re(\mathcal{E}'(z)), -Im(\mathcal{E}'(z))) = \frac{a_0 (z - c)}{\|z - c\|^2}$. This expression is exactly the center-of-mass approximation which is used in the force-approximation method of Barnes and Hut [8]. The corresponding multipole expansion with center c is called monopole expansion.

The following Lemma 5.20 shows how the center of a multipole expansion can be shifted. Lemma 5.21 describes how a multipole expansion can be converted into a power series (that will be called *local expansion*) in a circular region of analyticity, and Lemma 5.23 shows how the center of a finite local expansion can be shifted.

Lemma 5.20 (Translation of a Multipole Expansion). *Suppose that* $\mathcal{E}(z) = a_0 \log(z - z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(z-z_0)^k}$ *is a multipole expansion of the potential energy due to a set of charged particles that are located inside a circle of radius* r *and center* z_0*. Then, for any* z *outside a circle of radius* $r + |z_0 - z_1|$ *and center* z_1*, the potential energy induced by these particles is given by*

$$\mathcal{E}(z) = a_0 \log(z - z_1) + \sum_{l=1}^{\infty} \frac{b_l}{(z - z_1)^l} \text{ , where}$$

$$b_l = \frac{-a_0 (z_0 - z_1)^l}{l} + \sum_{k=1}^{l} a_k (z_0 - z_1)^{l-k} \binom{l-1}{k-1} .$$

Proof. Since obviously $|z_0 - z_1| < |z - z_1|$, we get

$$\begin{aligned} a_0 \log(z - z_0) &= a_0 \log\left((z - z_1) \left(1 - \frac{z_0 - z_1}{z - z_1} \right) \right) \\ &= a_0 \log(z - z_1) + a_0 \log\left(1 - \frac{z_0 - z_1}{z - z_1} \right) \\ &= a_0 \log(z - z_1) - a_0 \sum_{l=1}^{\infty} \frac{(z_0 - z_1)^l}{l \cdot (z - z_1)^l} . \end{aligned} \tag{5.10}$$

It can be shown by induction over k that for $k \geq 2$

$$\frac{1}{(z - z_0)^k} = \frac{(-1)^{k-1}}{(z_0 - z_1)(k-1)!} \cdot \left(\frac{z - z_1}{z - z_0} \right)^{(k-1)} \text{ and} \tag{5.11}$$

$$\left(\sum_{l=0}^{\infty} \left(\frac{z_0 - z_1}{z - z_1} \right)^l \right)^{(k-1)} = \sum_{l=k}^{\infty} \frac{(-1)^{k-1} (z_0 - z_1)^{l-k+1} \cdot (l-k+1) \cdot \ldots \cdot (l-1)}{(z - z_1)^l} . \tag{5.12}$$

Since

$$\frac{z - z_1}{z - z_0} = \frac{1}{1 - \frac{z_0 - z_1}{z - z_1}} = \sum_{l=0}^{\infty} \left(\frac{z_0 - z_1}{z - z_1} \right)^l ,$$

we obtain with Formulas (5.11) and (5.12)

$$\begin{aligned}\sum_{k=1}^{\infty}\frac{a_k}{(z-z_0)^k} &= \sum_{k=1}^{\infty} a_k \cdot \frac{(-1)^{k-1}}{(z_0-z_1)(k-1)!}\cdot\left(\frac{z-z_1}{z-z_0}\right)^{(k-1)} \\ &= \sum_{k=1}^{\infty} a_k \cdot \left(\sum_{l=k}^{\infty}(z_0-z_1)^{l-k}\binom{l-1}{k-1}(z-z_1)^{-l}\right) \\ &= \sum_{l=1}^{\infty}\sum_{k=1}^{l} a_k (z_0-z_1)^{l-k}\binom{l-1}{k-1}(z-z_1)^{-l}\,.\end{aligned} \tag{5.13}$$

The proof is complete by combining Formulas 5.10 and 5.13. □

Lemma 5.21 (Conversion of a Multipole Expansion into a Local Expansion). *Suppose that a set of charged particles is located inside a circle C_0 of radius r and center z_0, the corresponding multipole expansion is given by $\mathcal{E}(z) = a_0\log(z-z_0) + \sum_{k=1}^{\infty}\frac{a_k}{(z-z_0)^k}$, and that z_1 is a point with $|z_1-z_0| > 2r$. Then, inside a circle of radius r and center z_1 the potential energy due to these particles is given by the power series*

$$\mathcal{E}(z) = \sum_{l=0}^{\infty} b_l\cdot(z-z_1)^l\,,\ \textit{where} \tag{5.14}$$

$$b_0 = a_0\log(z_1-z_0) + \sum_{k=1}^{\infty}\frac{a_k}{(z_1-z_0)^k} \quad \textit{and} \tag{5.15}$$

$$b_l = \frac{(-1)^{l+1}a_0}{(z_1-z_0)^l\cdot l} + \left(\frac{1}{z_0-z_1}\right)^l\sum_{k=1}^{\infty}\binom{l+k-1}{k-1}\frac{a_k}{(z_1-z_0)^k}\ ,\ \textit{for } l\geq 1\,. \tag{5.16}$$

Proof. The power series is obtained by $\mathcal{E}(z) = \sum_{i=0}^{\infty}\frac{\mathcal{E}^{(i)}(z_1)}{i!}(z-z_1)^i$ using the definition of the multipole expansion. □

Definition 5.22 (Local Expansion, p-term Local Expansion). *The infinite power series in Formula* (5.14) *is called* local expansion of the potential energy field due to the set of charged particles contained in C_0. *The variables b_i in Formulas* (5.15) *and* (5.16) *are the* coefficients of the local expansion. *For $i = \{1,\ldots,p\}$ let b_i^p denote an approximation of the coefficient b_i in Formulas* (5.15) *and* (5.16)*, respectively so that only the first p coefficients a_k of the infinite series in Formulas* (5.15) *and* (5.16) *are calculated. Then, the finite power series $L^p(z) = \sum_{l=0}^{p} b_l^p\cdot(z-z_1)^l$ that is obtained by calculating only the coefficients $b_0^p,\ldots,b_p^p$ in Formula* (5.14) *is called* p-term local expansion of the potential energy due to the set of charged particles contained in C_0.

Lemma 5.23 (Translation of a Local Expansion). *For any complex z_0, z_1, z and $\{a_0,\ldots,a_p\}$*

$$\sum_{k=0}^{p} a_k(z-z_0)^k = \sum_{l=0}^{p}\left(\sum_{k=l}^{p} a_k\binom{k}{l}(z_1-z_0)^{k-l}\right)(z-z_1)^l\,.$$

Proof.

$$\begin{aligned}\sum_{k=0}^{p} a_k(z-z_0)^k &= \sum_{k=0}^{p} a_k((z-z_1)+(z_1-z_0))^k = \sum_{k=0}^{p} a_k \sum_{l=0}^{k} \binom{k}{l}(z-z_1)^l(z_1-z_0)^{k-l} \\ &= \sum_{l=0}^{p}\left(\sum_{k=l}^{p} a_k \binom{k}{l}(z_1-z_0)^{k-l}\right)(z-z_1)^l\end{aligned}$$

□

Since we are more interested in approximating the forces in the system rather than the potential energy, the following corollary of Lemma 5.15 can be used to obtain the forces that act in a potential energy field which is described by a multipole expansion or local expansion.

Corollary 5.24 (Obtaining the Forces from a Multipole or Local Expansions).

(a) Suppose, $\mathcal{E}(z) = a_0 \log(z-z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(z-z_0)^k}$ describes the potential energy field in a system of charged particles, and z is contained in a region of analyticity. Then, the forces that act on a particle of unit charge at position z are given by $(Re(\mathcal{E}'(z)), -Im(\mathcal{E}'(z)))$ with $\mathcal{E}'(z) = \frac{a_0}{z-z_0} - \sum_{k=1}^{\infty} \frac{k \cdot a_k}{(z-z_0)^{k+1}}$.

(b) Suppose, $\mathcal{E}(z) = \sum_{l=0}^{\infty} b_l \cdot (z-z_1)^l$ describes the potential energy field in a system of charged particles, and z is contained in a region of analyticity. Then, the forces that act on a particle of unit charge at position z are given by $(Re(\mathcal{E}'(z)), -Im(\mathcal{E}'(z)))$ with $\mathcal{E}'(z) = \sum_{l=1}^{\infty} l \cdot b_l (z-z_1)^{l-1}$.

Remark 5.25 (Working with p-Term Multipole and p-Term Local Expansions). *Theorem 5.17, Lemmas 5.20, 5.21, and 5.23, and Corollary 5.24 can also be used for working with p-term multipole expansions and p-term local expansions, respectively, which we will substantiate in the following.*

(a) Using Theorem 5.17, the coefficients of a p-term multipole expansion due to m charged particles can be obtained in $O(pm)$ time.

(b) Using Lemma 5.20, the coefficients of a shifted p-term multipole expansion can be obtained in $O(p^2)$ time.

(c) Using Lemma 5.21, the coefficients of a p-term local expansion can be obtained from the coefficients of a p-term multipole expansion in $O(p^2)$ time.

(d) Using Lemma 5.23, the coefficients of a shifted p-term local expansion can be obtained in $O(p^2)$ time.

(e) Using Corollary 5.24, the derivative of a p-term multipole expansion and the derivative of a p-term local expansion can be obtained in $O(p)$ time.

(f) Using Corollary 5.24, an approximation of the force that acts on a particle of unit charge at a position z — which is induced by the potential energy field that is described by a p-term multipole expansion or a p-term local expansion — can be obtained in $O(p)$ time assuming that z is contained in a region of analyticity.

We will demonstrate how p-term multipole expansions can be used to speed up force calculations in systems of charged particles by giving an example: Suppose that m particles of unit charge are located within a circle C_0 of radius r with center z_0 and that another m particles of unit charge are located within a circle C_1 of radius r with center z_1, and let $|z_0 - z_1| > 2r$ (see Figure 5.17).

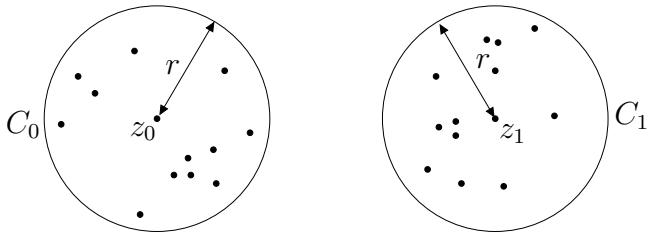


Figure 5.17: An example distribution showing the use of p-term multipole expansions.

Computing the repulsive forces acting on each particle in C_0 due to all particles in C_1 naively would need $\Theta(m^2)$ time. Now suppose that we first compute the coefficients of a p-term multipole expansion of the potential energy due to the particles in C_1. This needs $O(pm)$ time (see Remark 5.25(a)). Then, we calculate the derivative of the p-term multipole expansion in $O(p)$ time (see Remark 5.25(e)) before evaluating it for each particle in C_0. This needs $m \cdot O(p)$ time (see Remark 5.25(f)). Hence, the total running time for approximating the forces is $O(pm)$, which is significantly faster than the naive approach if $m >> p$.

Some Terminology

We will introduce some definitions that will make the explanation of the multipole framework easier and that will be used in the remainder of this section.

In the previous sections we have associated each node v with a box $box(v)$. (Recall that a *cell* or *box* is a quadratic sub-region of a square D that can be generated by a recursive decomposition of D into four squares of equal size.) In the following, we will associate two boxes with a node v that are defined next. Figures 5.18(a) and 5.18(b) explain this terminology at an example.

Definition 5.26 (Small Cell, Large Cell of a Node). *Suppose, we have given a reduced bucket quadtree T with fixed constant leaf capacity l for a given set $C = \{c_1, \ldots, c_N\}$ of distinct particles that are distributed in a square D. The* small cell *or* small box *of a node v (shorter $Sm(v)$) is the smallest sub-box of D that covers all particles that are associated with v. If v is a leaf that contains only one particle, then $Sm(v)$ is a point. If v is the*

root of T, the large cell *or* large box *of v (shorter $Lg(v)$) is equal to $Sm(v)$. Otherwise, let parent(v) be the parent of v in T. Then, $Lg(v)$ is the largest sub-box of $Sm(parent(v))$ with size smaller than $Sm(parent(v))$ that covers $Sm(v)$.*

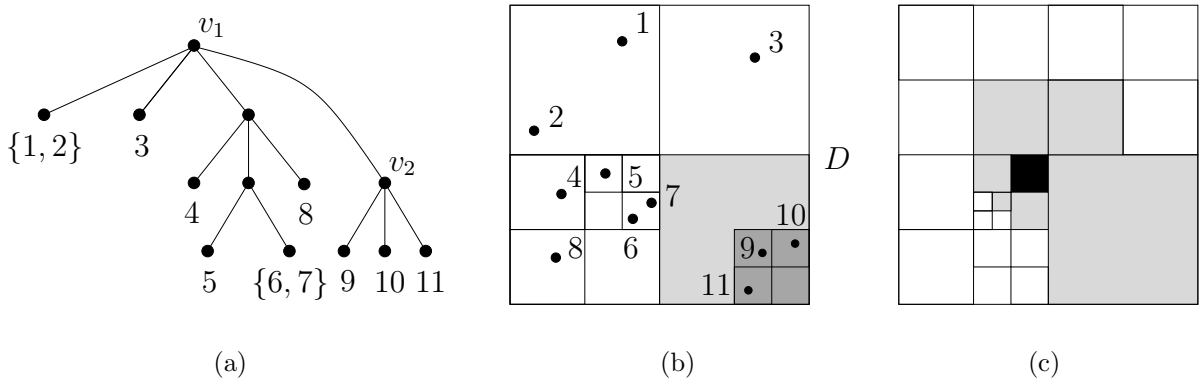


Figure 5.18: (a) A reduced bucket quadtree that corresponds to the particle distribution in the square D shown in (b). The small cell $Sm(v_2)$ corresponds to the small grey box. The small cell of $parent(v_2) = v_1$ corresponds to the square D. Hence, the large cell $Lg(v_2)$ corresponds to the big grey box that covers the small grey box in (b). (c) Unlike the white boxes, the grey boxes are neighbors of the black box.

Remark 5.27 (Simple Properties of Small and Large Cells). *It follows from the definition of $Sm(v)$, $Lg(v)$, and the reduced bucket quadtree T that the small cell $Sm(v)$ of an interior node v is exactly box(v). For a leaf v of T, $Sm(v)$ is covered by box(v). Furthermore, $Lg(v) \setminus Sm(v)$ covers no particles in C, and the size of $Lg(v)$ is $\frac{1}{4}$ of the size of $Sm(parent(v))$ if v is not the root of T.*

We have seen that the Multipole Theorem 5.17 and the lemmas for shifting, converting, and evaluating these expansions can only be applied in well-defined regions of analyticity. In particular, the conversion of a multipole expansion to a local expansion (see Lemma 5.21) can only be done if the corresponding two circles of equal size with centers z_0 and z_1 do not overlap. Furthermore, we have seen in Remark 5.19(a) that the accuracy of a p-term multipole expansion becomes the better, the greater the distance between the circular region that contains the particles and the data point. Since in all quadtree data structures the regions are squares, the terminology of *well-separateness* is used to indicate that the operations of the Multipole Theorem 5.17 and the lemmas for working with these expansions can be applied.

Definition 5.28 (Neighbor, Well-Separated, Ill-Separated). *Two boxes B_1 and B_2 are called* neighbors *if the boundaries of B_1 and B_2 touch, but B_1 and B_2 do not overlap. Two boxes B_1 and B_2 of same size are* well-separated *if they are no neighbors. Otherwise, B_1 and B_2 are* ill-separated. *Suppose, nodes u and v are nodes of a reduced bucket quadtree*

T and $Sm(u) \geq Sm(v)$. Then, u and v are well-separated *if and only if $Sm(u)$ and the cell that covers $Sm(v)$ and that has the same size as $Sm(u)$ are well-separated. Otherwise u and v are* ill-separated. *Suppose that $Sm(u) < Sm(v)$. Then, u and v are* well-separated *if and only if $Sm(v)$ and the cell that covers $Sm(u)$ and that has the same size as $Sm(v)$ are well-separated. Otherwise u and v are* ill-separated.

Figure 5.18(c) shows a box B and its neighbors, while Figure 5.19(a) and (b) show the small cells of two nodes that are well-separated and ill-separated, respectively.

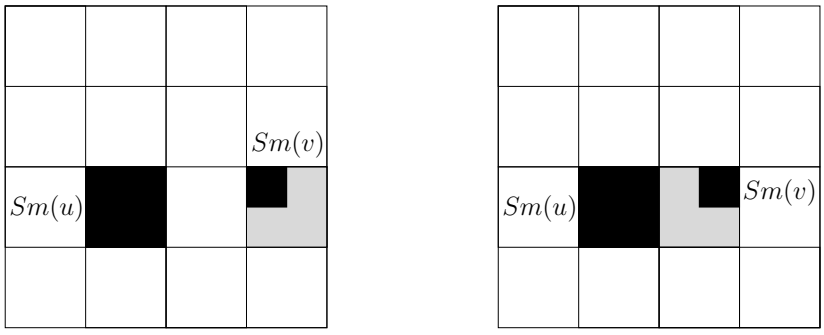


Figure 5.19: The black boxes $Sm(u)$ and $Sm(v)$ are small cells of two nodes u and v of a reduced bucked quadtree. (a) Since the grey box is no neighbor of $Sm(u)$, u and v are well-separated. (b) Since the grey box is a neighbor of $Sm(u)$, u and v are ill-separated.

Like in [58, 59, 2] the terminology of well-separateness is used to define an interaction set that is used do generate p-term local expansions from suitable p-term multipole expansions. The following definition of an interaction set of a node v of T has been invented in [2] and is a generalization of the definition of the interaction set defined in [58, 59].

Definition 5.29 (Interaction Set, Minimal Ill-Separated Set). *Suppose, nodes u and v are nodes of a reduced (bucket) quadtree T. The* interaction set *$I(v)$ of a node v is the set of all nodes u that are ill-separated from the parent of v, well-separated from v, and the parent of u is ill-separated from v. Thus, $I(v) = \{u \mid$ `Well_Separated`(v, u), `Ill_Separated` $(parent(v), u)$, `Ill_Separated`$(parent(u), v)\}$. The* minimal ill-separated set *$R(v)$ of a node v is the set of all nodes that are ill-separated from v and have the small cell smaller or equal and the large cell larger or equal than the small cell of v. Hence, $R(v) = \{u \mid$ `Ill_Separated`$(v, u), Sm(u) \leq Sm(v)) \leq Lg(u)\}$.*

In the multipole framework of Aluru et al. [2] it is sufficient to associate the lists $I(v)$ and $R(v)$ with each node v of the reduced quadtree in order to approximate the repulsive forces in the system. Since we use the reduced bucket quadtree data structure, the leaves possibly contain more than one particle (in contrast to the reduced quadtree data structure used in [2]). We will see that this generalization can be modeled in our framework by defining

some additional sets which are assigned to each node of the tree and that are introduced next.

Definition 5.30 (The Sets $D_1(v)$, $D_2(v)$, $D_3(v)$ and $K(v)$). *For each node v of the reduced bucket quadtree $T = (V, E)$, $D_1(v)$ is the set of all leaves $w \in V$ so that $Sm(v) < Sm(w)$ and $Sm(v)$ and $Sm(w)$ are neighbors. $D_2(v)$ is the set of all leaves w of T so that $Sm(v) < Sm(w)$, $Sm(v)$ and $Sm(w)$ are no neighbors, v and w are ill-separated, and w is not contained in the set $D_2(u)$ of an ancestor u of v. For each leaf $v \in V$ we additionally define the sets $D_3(v)$ and $K(v)$, where $D_3(v)$ is the set of all leaves $w \in V$ so that $Sm(v) \geq Sm(w)$ and $Sm(v)$ and $Sm(w)$ are neighbors. For a leaf $v \in V$ the set $K(v)$ contains all nodes w with parent parent(w) so that $Sm(v)$ and $Sm(w)$ are no neighbors. Additionally, it is required that either $w \in R(v)$ or an ancestor of w is contained in $R(v)$ and $Sm(parent(w))$ and $Sm(v)$ are neighbors.*

Formal Description of the Multipole Framework

We can assume that we have given a set $C = \{c_1, \ldots, c_N\}$ of charged particles of unit charge that are placed at distinct positions $p(C) = \{p_1, \ldots, p_N\}$, a reduced bucket quadtree $T = (V, E)$ with fixed constant leaf capacity l that is associated with C, and a fixed constant precision parameter p.

In this context we have to mention that besides the space for storing the sets $R(v)$, $I(v)$, $D_1(v)$, $D_2(v)$ (and possibly $D_3(v)$ and $K(v)$), additional space for storing the $p+1$ coefficients $\{a_0, \ldots, a_p\}$ of p term multipole expansions $M^p(v)$ and space for storing the $p+1$ coefficients $\{b_0, \ldots, b_p\}$ of p term local expansions $L^p(v)$ is needed for each node v of the reduced bucket quadtree T.

Like the force-approximation methods of Greengard [58], Greengard and Rokhlin [59], and Aluru et al. [2] the multipole framework is based on a bottom-up traversal and a top-down traversal of the given quadtree data structure with a fixed constant precision parameter p and can be seen as a generalization of the multipole framework of Aluru et al. [2]. The pseudocode of the multipole framework is shown in Function `MF` and Function `Calculate_Local_Expansions_and_Node_Sets`, and its parts are explained in the following.

Part 1: Trivial Case and Initializations

If the reduced bucket quadtree $T = (V, E)$ consists of only one node, the forces that act on each particle p_i due to all other particles in C are calculated directly, and the algorithm terminates. Otherwise, for each node v of T the sets $R(v), I(v), D_1(v), D_2(v)$ (and for the leaves additionally $D_3(v)$, and $K(v)$) are initialized.

Part 2: Bottom-Up Traversal of the Tree

Now for each leaf v of T the coefficients $\{a_0, \ldots, a_p\}$ of the p-term multipole expansion $M^p(v)$ that reflects an approximation of the potential energy field due to the particles that

Function `MF`(C, T, p)

input : a set $C = \{c_1, \ldots, c_N\}$ of charged particles of unit charge that are placed at distinct positions $p(C) = \{p_1, \ldots, p_N\}$, a reduced bucket quadtree $T = (V, E)$ with fixed constant leaf capacity l that is associated with C, and a fixed constant precision parameter p

output: a function $F_{rep}\colon C \longrightarrow \mathbb{R}^2$ so that $F_{rep}(c_i)$ is an approximation of the repulsive forces that act on c_i due to all other particles in C

begin
- **begin**{`Part 1: Trivial Case and Initializations`}
  - **if** T contains only one node **then**
    - **foreach** $c_i \in C$ **do**
      - $F_{rep}(c_i) \leftarrow$ `Naive_Direct_Force_Calculation`;
    - `Exit`;
  - **foreach** $v \in V$ **do** $R(v) \leftarrow I(v) \leftarrow D_1(v) \leftarrow D_2(v) \leftarrow \emptyset$;
  - **foreach** leaf $v \in V$ **do** $D_3(v) \leftarrow K(v) \leftarrow \emptyset$;
- **end**
- **begin** {`Part 2: Bottom-Up Traversal`}
  - **foreach** leaf $v \in V$ **do**
    - calculate the coefficients of $M^p(v)$ due to all particles contained in $Sm(v)$;
  - **foreach** interior node $v \in V$ for which coefficients of $M^p(w)$ of all children w of v have been calculate **do**
    - calculate the coefficients of $M^p(v)$ due to all particles contained in $Sm(v)$ by adding coefficients of shifted $M^p(w)$ of all children w of v;
- **end**
- **begin** {`Part 3: Top-Down Traversal`}
  - **foreach** child v of the root of T **do**
    - `Calculate_Local_Expansions_and_Node_Sets`(T, C, p, v);
- **end**
- **begin** {`Part 4: Obtain the Forces`}
  - **foreach** leaf $v \in V$ **do**
    - **foreach** particle $c_i \in Sm(v)$ **do**
      - let $C(v)$ be the set of charged particles that are contained in $Sm(v)$;
      - calculate $F_{local}(c_i)$ using the coefficients of $L^p(v)$;
      - calculate $F_{direct}(c_i)$ using $D_1(v) \cup D_3(v) \cup C(v)$;
      - calculate $F_{multipole}(c_i)$ using $K(v)$;
      - $F_{rep}(c_i) \leftarrow F_{local}(c_i) + F_{direct}(c_i) + F_{multipole}(c_i)$;
- **end**

end

Function `Calculate_Local_Expansions_and_Node_Sets`(T, C, p, v)

input : C, T, p are defined like in Function `MF`, a node v of T

output: the coefficients $\{b_0, \ldots, b_p\}$ of the p-term local expansions of v, the sets $R(v)$, $I(v)$, $D_1(v)$, $D_2(v)$, and additionally $D_3(v)$ and $K(v)$ if v is a leaf

begin

  begin {`Part 3.1: Find` $R(v), I(v), D_1(v)$, `and` $D_2(v)$}

    if *parent*(v) is the root of T **then** $E(v) \leftarrow$ *parent*(v);

    else $E(v) \leftarrow R($*parent*$(v)) \cup D_1($*parent*$(v))$;

    while $E \neq \emptyset$ **do**

      pick some $u \in E(v)$, and set $E(v) \leftarrow E(v) \setminus \{u\}$;

      if `Well_Separated`(u, v) **then** $I(v) \leftarrow I(v) \cup \{u\}$;

      else if $Sm(v) \geq Sm(u)$ **then** $R(v) \leftarrow R(v) \cup \{u\}$;

      else if v is no leaf of T **then** $E(v) \leftarrow E(v) \cup$ *children*(u);

      else if `Neighbors`$(Sm(u), Sm(v))$ **then** $D_1(v) \leftarrow D_1(v) \cup \{u\}$;

      else $D_2(v) \leftarrow D_2(v) \cup \{u\}$;

  end

  begin {`Part 3.2: Calculate Coefficients of` $L^p(v)$}

    foreach $u \in I(v)$ **do**

      convert $M^p(u)$ to $L^p(u)$, and add coefficients of $L^p(u)$ to coeff. of $L^p(v)$;

    foreach $u \in D_2(v)$ **do**

      foreach $c_i \in Sm(u)$ **do**

        calculate $M^p(c_i)$, convert $M^p(c_i)$ to $L^p(c_i)$, and add coefficients of $L^p(c_i)$ to coefficients of $L^p(v)$;

    if coefficients of $L^p($*parent*$(v))$ have been calculated **then**

      add coefficients of shifted $L^p($*parent*$(v))$ to $L^p(v)$;

  end

  begin {`Part 3.3: Find` $D_3(v)$ `and` $K(v)$ `for Leaves`}

    if v is a leaf of T **then**

      set $E(v) \leftarrow R(v)$;

      while $E(v) \neq \emptyset$ **do**

        pick some $u \in E(v)$, and set $E(v) \leftarrow E(v) \setminus \{u\}$;

        if not `Neighbors`$(Sm(u), Sm(v))$ **then** $K(v) \leftarrow K(v) \cup \{u\}$;

        if `Neighbors`$(Sm(u), Sm(v))$ and u is leaf **then** $D_3(v) \leftarrow D_3(v) \cup \{u\}$;

        else $E(v) \leftarrow E(v) \cup$ *children*(u);

  end

  begin {`Part 3.4: Recursion`}

    if v is no leaf of T **then foreach** child w of v **do**

      `Calculate_Local_Expansions_and_Node_Sets`(T, C, p, w);

  end

end

are contained in $Sm(v)$ are calculated. This is done by using Theorem 5.17 and choosing the center of $Sm(v)$ as the variable z_0 in Theorem 5.17.

The p-term multipole expansions of the interior nodes are calculated by traversing the tree bottom up: Suppose, v is an interior node of T with small cell $Sm(v)$, the center of $Sm(v)$ is z_1, and the p-term multipole expansions $M^p(w)$ of each of v's children w have been calculated. Then, the coefficients of the p-term multipole expansion $M^p(v)$ that reflects an approximation of the potential energy field due to the particles that are contained in $Sm(v)$ are obtained by first shifting the center of each $M^p(w)$ to z_1 using Lemma 5.20 and then adding the coefficients of this shifted p-term multipole expansion to the corresponding coefficients of $M^p(v)$.

As a result of part 2, for each node $v \in V$ the coefficients of p-term multipole expansions $M^p(v)$ are obtained that reflect an approximation of the potential energy field induced by all charged particles that are contained in $Sm(v)$.

Part 3: Top-Down Traversal of the Tree

In the top-down traversal of the tree T, the root of T is skipped, and the recursive Function `Calculate_Local_Expansions_and_Node_Sets` is called for each of its children.

Part 3.1: Find $R(v)$, $I(v)$, $D_1(v)$, and $D_2(v)$

In this part of Function `Calculate_Local_Expansions_and_Node_Sets` the sets $R(v)$, $I(v)$, $D_1(v)$, and $D_2(v)$ are constructed starting with a set $E(v)$ that is assigned $R(\mathit{parent}(v)) \cup D_1(\mathit{parent}(v))$. Note that if $\mathit{parent}(v)$ is the root of T, it is clear that $R(\mathit{parent}(v)) = v$ and $D_1(\mathit{parent}(v)) = \emptyset$. Then, iteratively a node u is taken from $R(\mathit{parent}(v)) \cup D_1(\mathit{parent}(v))$, and it is checked if u already belongs to one of the previous mentioned sets or if one has to explore the subtrees that are rooted at u recursively in order to assign its children to these sets.

In particular, if u and v are well-separated, then u belongs to $I(v)$. (This can be seen as follows: If u is a node of $R(\mathit{parent}(v)) \cup D_1(\mathit{parent}(v))$, then since u and v are well-separated $u \in R(\mathit{parent}(v))$. Hence, $\mathit{parent}(v)$ and u are ill-separated by definition of $R(\mathit{parent}(v))$. $Lg(u) \geq Sm(\mathit{parent}(v))$ by definition of $R(\mathit{parent}(v))$ and, hence, $\mathit{parent}(u)$ and v are ill-separated, too. If in the complementary case, u is a proper ancestor of a node in $R(\mathit{parent}(v)) \cup D_1(\mathit{parent}(v))$, u is a proper ancestor of a node $R(\mathit{parent}(v))$ since the nodes in $D_1(\mathit{parent}(v))$ are leaves. Hence, $\mathit{parent}(v)$ and u are ill-separated. Furthermore, $\mathit{parent}(u)$ and v are ill-separated, since otherwise $\mathit{parent}(u) \in I(v)$.)

If u and v are ill-separated and additionally $Sm(v) \geq Sm(u)$ u, then u belongs to $R(v)$ (To see this, we have to prove that $Lg(u) \geq Sm(v)$: Suppose, $u \in R(\mathit{parent}(v)) \cup D_1(\mathit{parent}(v))$, then it follows from definition of $D_1(\mathit{parent}(v))$ that $u \in R(\mathit{parent}(v))$. Therefore, $Lg(u) \geq Sm(\mathit{parent}(v))$ and hence $Lg(u) \geq Sm(v)$. In the complementary case, u is a proper ancestor of a node in $R(\mathit{parent}(v))$ by definition of $D_1(\mathit{parent}(v))$. In

this case $Lg(u) \geq Sm(v)$ holds since $Sm(\mathit{parent}(u)) > Sm(v)$ by the dynamic construction of $E(v)$.)

If u and v are ill-separated, $Sm(v) < Sm(u)$, u is a leaf, and $Sm(u)$ and $Sm(v)$ are neighbors, then u is contained in $D_1(v)$.

If u and v are ill-separated, $Sm(v) < Sm(u)$, u is a leaf, and $Sm(u)$ and $Sm(v)$ are no neighbors, then u is contained in $D_2(v)$, since $R(\mathit{parent}(v)) \cup D_1(\mathit{parent}(v))$ and $D_2(\mathit{parent}(v))$ are disjoint and u is an element of $R(\mathit{parent}(v)) \cup D_1(\mathit{parent}(v))$ or an ancestor of an element in $R(\mathit{parent}(v)) \cup D_1(\mathit{parent}(v))$.

In the remaining case $Sm(v) < Sm(u)$ and u is an interior node of T that is ill-separated from v. Hence, one has to check whether its children belong to one of the sets $R(v)$, $I(v)$, $D_1(v)$, or $D_2(v)$.

Note that by construction of part 3.1 the particles that are covered by the small cells of all nodes in $R(\mathit{parent}(v)) \cup D_1(\mathit{parent}(v))$ are exactly the particles that are covered by the small cells of all nodes in $R(v) \cup I(v) \cup D_1(v) \cup D_2(v)$. Furthermore, each such particle is covered by the small cell of exactly one node in $R(v) \cup I(v) \cup D_1(v) \cup D_2(v)$.

Part 3.2: Calculate Coefficients of $L^p(v)$

Like in other multipole methods, for each u in the interaction set $I(v)$ the coefficients of p-term multipole expansions $M^p(u)$ are converted to coefficients of p-term local expansions $L^p(u)$ and added to the corresponding coefficients of $L^p(v)$ using Lemma 5.21 and choosing $z_0 := \mathit{center}(Sm(u))$ and $z_1 := \mathit{center}(Sm(v))$.

Some difficulties arise for the nodes in $D_2(v)$: Since each $u \in D_2(v)$ is a leaf that is ill-separated from v with $Sm(v) < Sm(u)$, one cannot apply Lemma 5.21 on u. However, we found another way to calculate the local expansions due to the particles that are contained in u: Let $\{c_1, \ldots, c_k\}$ be the set of charged particles that are contained in $Sm(u)$ at positions $\{p_1, \ldots, p_k\}$. First, we calculate the coefficients of the p-term multipole expansion $M^p(c_i)$ for each single particle c_i using Theorem 5.17 and choosing $z_0 := p_i$. Since we can interpret each point p_i as a dimensionless box B_i, the largest cell that covers B_i and that has the same size as $Sm(v)$ is no neighbor of $Sm(v)$ due to the definition of $D_2(v)$ (see Figure 5.20(a)). Hence, $Sm(v)$ and B_i are well-separated, and Lemma 5.21 can be used to convert $M^p(c_i)$ to a p-term local expansion $L^p(c_i)$ choosing $z_0 := p_i$ and $z_1 := \mathit{center}(Sm(v))$. Finally, for each $c_i \in \{c_1, \ldots, c_k\}$ the coefficients of $L^p(c_i)$ are added to $L^p(v)$.

Following standard practice, the coefficients of the shifted p-term local expansion $L^p(\mathit{parent}(v))$ of the parent of v are added to the corresponding coefficients of $L^p(v)$. Thus, an approximation of the potential energy field of the region that is reflected by $L^p(\mathit{parent}(v))$ is inherited to v using Lemma 5.23 and choosing the center of $Sm(v)$ as z_1.

As a result of part 3.2, the coefficients of $L^p(v)$ reflect an approximation of the potential energy field due to all particles contained in the small cells of the nodes in $I(v)$, $D_2(v)$, and in the $I(u)$ and $D_2(u)$ sets of all ancestors of v. Furthermore, the small cells of the nodes in $R(v) \cup D_1(v)$ are exactly the regions that have not been considered yet in the calculation of the potential energy in $Sm(v)$ due to all particles in the region $D \setminus Sm(v)$.

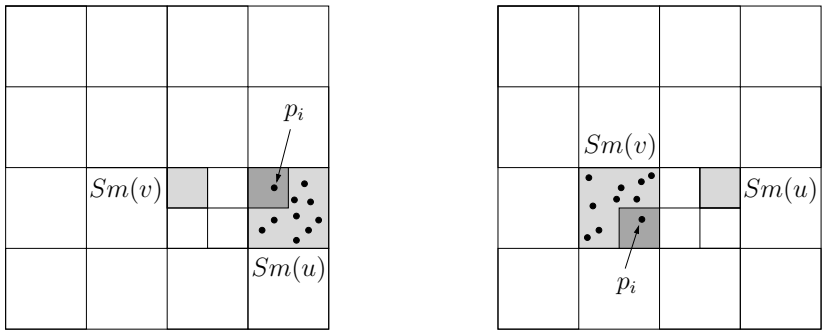


Figure 5.20: (a) The grey boxes $Sm(v)$ and $Sm(u)$ are ill-separated, no neighbors, and $Sm(v) < Sm(u)$. $Sm(v)$ and the sub-box of $Sm(u)$ that contains particle c_i at position p_i and has the same size as $Sm(v)$ (the dark-grey box) are well-separated. (b) The grey boxes $Sm(v)$ and $Sm(u)$ are ill-separated, no neighbors, and $Sm(v) > Sm(u)$. $Sm(u)$ and the sub-box of $Sm(v)$ that contains particle c_i at position p_i and has the same size as $Sm(u)$ (the dark-grey box) are well-separated.

Part 3.3: Find $D_3(v)$ and $K(v)$ for Leaves

Suppose that v is a leaf of T and u is a node in $R(v)$. By definition of $R(v)$ we know that u and v are ill-separated and that $Sm(v) \geq Sm(u)$. Three cases can arise: If u and v are no neighbors, u is an element of $K(v)$. If u and v are neighbors and u is a leaf, u is an element of $D_3(v)$. Otherwise, u is a neighbor of v but an interior node of T. Thus, we can recursively assign the children of u to either $K(v)$ or $D_3(v)$, since all children of u are ill-separated from v, their small cell is smaller than $Sm(v)$, and $Sm(u)$ is a neighbor of $Sm(v)$.

Note that the particles that are covered by the small cells of the nodes in $R(v)$ are exactly the particles that are covered by the small cells of the nodes in $K(v) \cup D_3(v)$. Furthermore, each such particle is covered by the small cell of exactly one node in $K(v) \cup D_3(v)$.

Part 3.4: Recursion

Suppose that v is an interior node of T. The small cells of the nodes in $R(v) \cup D_1(v)$ are exactly the regions that have not been considered yet in the calculation of the potential energy in $Sm(v)$ due to all particles that are contained in the region $D \setminus Sm(v)$. Hence, we can simply inherit the sets $R(v)$ and $D_1(v)$ to each child w of v and call Function `Calculate_Local_Expansions_and_Node_Sets` for w.

As a result of parts 1 to 3 we have given the coefficients of the p-term local expansions $L^p(v)$ for each leaf v of T, and $L^p(v)$ reflects an approximation of the potential energy field induced by the particles that are not covered by the small cells of all nodes contained in $D_1(v) \cup D_3(v) \cup K(v) \cup Sm(v)$.

Part 4: Obtain the Forces

Suppose that v is a leaf of T so that $Sm(v)$ contains the particles $\{c_1, \ldots, c_k\}$, which are placed at positions $\{p_1, \ldots, p_k\}$. We can obtain an approximation of the repulsive forces that act on each $c_i \in \{c_1, \ldots, c_k\}$ due to all particles in the system $C = \{c_1, \ldots, c_N\}$ as follows:

Let L' be the derivative of $L^p(v)$. We can use Corollary 5.24(b) in order to obtain the forces that are induced by L' and act on c_i by setting $F_{local}(c_i) = (Re(L'(p_i)), -Im(L'(p_i)))$.

The forces that act on c_i due to all particles that are contained in $Sm(v) \cup \{Sm(u) \mid u \in D_1(v) \cup D_3(v)\}$ are calculated directly by a naive exact force calculation, since v and all nodes $u \in D_1(v) \cup D_3(v)$ are leaves. The resulting forces are denoted by $F_{direct}(c_i)$.

We only have to concentrate on the particles that are covered by the small cells of the nodes in $K(v)$. Let u be a node in $K(v)$. Since u is ill-separated from v and $Sm(v) > Sm(u)$, we cannot apply Lemma 5.21 in order to convert $M^p(u)$ to $L^p(u)$ and to add the coefficients of $L^p(u)$ to the corresponding coefficients of $L^p(v)$. However, we can use a similar trick as the trick that we have invented in part 3.2: Let c_i be a particle that is placed at position $p_i \in Sm(v)$. Since we can interpret p_i as a dimensionless box B_i, the largest cell that contains c_i and that has the same size as $Sm(u)$ are no neighbors due to the definition of $K(v)$ (see Figure 5.20(b)). Hence, $Sm(u)$ and B_i are well-separated, and Corollary 5.24(a) can be used in order to obtain the forces that act on c_i due to all particles contained in $Sm(u)$. In particular, let M' be the derivative of $M^p(u)$. Then, the force on c_i that is induced by $M^p(u)$ is given by $F_{multipole}(c_i) = (Re(M'(p_i)), -Im(M'(p_i)))$.

Therefore, the approximation of the repulsive force that acts on a particle c_i due to all particles in $C = \{c_1, \ldots, c_N\}$ is given by the sum of $F_{direct}(c_i)$, $F_{local}(c_i)$, and $F_{multipole}(c_i)$.

The Running Time of the Multipole Framework

In order to prove that the running time of Function `MF` is linear in the number of particles, we need the following lemma.

Lemma 5.31 (The Sizes of the Sets $R(v)$, $I(v)$, $D_1(v)$, $D_2(v)$, $D_3(v)$, and $K(v)$). *Let $T = (V, E)$ be a reduced bucket quadtree with constant leaf capacity l that is associated with a set $C = \{c_1, \ldots, c_N\}$ of N charged particles that are placed at distinct positions $\{p_1, \ldots, p_N\}$, and leaves(T) is the set of the leaves of T. Then,*

$$\sum_{v \in V} |R(v)| = O(N), \tag{5.17}$$

$$\sum_{v \in V} |I(v)| = O(N), \tag{5.18}$$

$$\sum_{v \in V} |D_1(v)| = O(N), \tag{5.19}$$

$$\sum_{v \in V} |D_2(v)| = O(N), \tag{5.20}$$

$$\sum_{v \in \text{leaves}(T)} |D_3(v)| = O(N), \text{ and} \tag{5.21}$$

$$\sum_{v \in \text{leaves}(T)} |K(v)| = O(N). \tag{5.22}$$

Proof. The proofs of Equations (5.17) and (5.18) are similar to the corresponding proofs in [2]. We prove Equation (5.17) first. Let v be an arbitrary node of T. Each node $u \in R(v)$ is ill-separated from v and $Sm(v) \geq Sm(u)$. Hence, $Sm(u)$ is either covered by $Sm(v)$ or covered by a neighbor box B of $Sm(v)$ that has the same size as $Sm(v)$. In the first case, we know from the definition of $R(v)$ that $Sm(v) \leq Lg(u)$. By definition of $Lg(u)$ it follows that $u = v$. In the second case, by using that $Sm(v) \leq Lg(u)$ and Remark 5.27, we know that $Lg(u) \setminus Sm(u)$ contains no particles. Hence, $B \setminus Sm(u)$ contains no particles, too. Since $Sm(v)$ has exactly 8 neighbor boxes B of equal size, we get that $\sum_{v \in V} |R(v)| \leq (1+8) \cdot |V| = O(|V|) = O(N)$.

We now prove Equation (5.18). Suppose, w is a node in $I(v)$, then either w is in $R(\mathit{parent}(v))$ or an ancestor u of w is in $R(\mathit{parent}(v))$ or w is in $D_1(\mathit{parent}(v))$ or an ancestor u of w is in $D_1(\mathit{parent}(v))$. The last two options can be excluded by the fact that all elements of $D_1(\mathit{parent}(v))$ are neighboring leaves of $Sm(\mathit{parent}(v))$ with a small cell that is larger than $Sm(\mathit{parent}(v))$ and, hence, are ill-separated from v and have no ancestors. Therefore,

$$\begin{aligned} \textstyle\sum_{v \in V} |\{w \mid w \in I(v)\}| \;\; &= \;\; \textstyle\sum_{v \in V} |\{w \mid w \in I(v), w \in R(\mathit{parent}(v))\}| + \\ &\quad\; \textstyle\sum_{v \in V} |\{w \mid w \in I(v), w \notin R(\mathit{parent}(v))\}|. \end{aligned}$$

By Equation (5.17) we know that $\sum_{v \in V} |\{w \mid w \in I(v), w \in R(\mathit{parent}(v)\}| = O(N)$. We can rewrite the second term of this equation as follows:

$$\sum_{v \in V} |\{w \mid w \in I(v), w \notin R(\mathit{parent}(v))\}| = \sum_{w \in V} |\{v \mid w \in I(v), w \notin R(\mathit{parent}(v))\}|$$

Hence, in order to show that Equation (5.18) holds, it is sufficient to show that $S(w) := |\{v \mid w \in I(v), w \notin R(\mathit{parent}(v))\}| = O(1)$. Let w be arbitrarily fixed. Since $w \notin R(\mathit{parent}(v))$ there exists a node $u \in R(\mathit{parent}(v))$ that is an ancestor of w. Thus, $\mathit{parent}(v)$ and $\mathit{parent}(w)$ are ill-separated, and $Sm(\mathit{parent}(v)) \geq Sm(\mathit{parent}(w))$. It can be shown that there exists a box B that is a neighbor of box $Sm(\mathit{parent}(w))$, B has the same size as $Sm(\mathit{parent}(w))$, and B is a sub-box of $Sm(\mathit{parent}(v))$ that contains $Sm(v)$. (To see this: $Sm(v)$ cannot be larger than $Sm(\mathit{parent}(w))$, since — by definition of $I(v)$ — v and $\mathit{parent}(w)$ are ill-separated but v and w are well-separated. Suppose that $Sm(v)$ is not covered by a box B that is a neighbor of $Sm(\mathit{parent}(w))$ and that has the same size as $Sm(\mathit{parent}(w))$, then v and $\mathit{parent}(w)$ are well-separated, which contradicts the fact that $w \in I(v)$ by definition of $I(v)$. Finally, $Sm(\mathit{parent}(v))$ covers B, since $Sm(\mathit{parent}(v)) \geq Sm(\mathit{parent}(w))$.) For each such box B there exist at most 4 sub-boxes like $Sm(v)$ since $Sm(\mathit{parent}(v))$ covers B. Since for each w, the number of neighbor boxes of equal size is

bounded above by 8, $|S(w)|$ is bounded above by $4 \cdot 8 = 32$, which completes the proof of Equation (5.18).

The proof of Equation (5.19) is easy. Let v be an arbitrary node of T. Since there exist at most 8 boxes that are neighbors of $Sm(v)$ and that have the same size as $Sm(v)$, there exist less than 8 leaves w that are neighbors of v with $Sm(w) > Sm(v)$. Hence, $\sum_{v \in V} |D_1(v)| < 8|V| = O(N)$.

Next, we prove Equation (5.20). Let us suppose that the leaf capacity l is 1 and that v is a node of T. Then, the small cells of all leaves have size zero. By definition of $D_1(v)$ and $D_2(v)$ it follows that $D_1(v) = D_2(v) = \emptyset$. Thus, by Equations (5.17) and (5.18) for $l = 1$ the equality $\sum_{v \in V} |R(v)| + |I(v)| + |D_1(v)| + |D_2(v)| = O(N)$ holds. Furthermore, the elements of $R(\mathit{parent}(v)) \cup D_1(\mathit{parent}(v))$ or the descendants of the elements of $R(\mathit{parent}(v)) \cup D_1(\mathit{parent}(v))$ are partitioned into the disjoint subsets $R(v)$ and $I(v)$ (since $D_1(v) = D_2(v) = \emptyset$). Now let us suppose that $l > 1$. Then, the nodes contained in $D_2(v)$ are either elements of the set $R(\mathit{parent}(v)) \cup D_1(\mathit{parent}(v))$ or are descendants of the elements of $R(\mathit{parent}(v)) \cup D_1(\mathit{parent}(v))$. In the first case, the number of elements of $D_2(v)$ that are contained in $R(\mathit{parent}(v)) \cup D_1(\mathit{parent}(v))$ is bounded above by $O(N)$ using Equations (5.17) and (5.19). In the second case, we use that by Equations (5.17), (5.18), and (5.19) $\sum_{v \in V} |R(v)| + |I(v)| + |D_1(v)| = O(N)$. Hence, it is sufficient to show that $\sum_{v \in V} |R(v)| + |I(v)| + |D_1(v)| + |D_2(v)| = O(N)$. This can be shown as follows: Suppose that leaf u is an ancestor of a node in $R(\mathit{parent}(v)) \cup D_1(\mathit{parent}(v))$, contains k particles with $2 \leq k \leq l$, and is neither assigned to $I(v)$ nor to $R(v)$. Then, u is assigned to either $D_1(v)$ or $D_2(v)$ since it has no further descendants in T. Hence, the cardinality of the actual set $R(v) \cup I(v) \cup D_1(v) \cup D_2(v)$ is increased by one. In contrast to this, in the case that $l = 1$ the subtree rooted at u would have been explored further. Since in this case T is a reduced quadtree, this exploration would result in adding at least two nodes to $R(v) \cup I(v)$. Therefore, in the case that $l > 1$ the expression $\sum_{v \in V} |R(v)| + |I(v)| + |D_1(v)| + |D_2(v)|$ is bounded above by $O(N)$, too.

In order to prove Equation (5.21), we can use that

$$\sum_{v \in \mathit{leaves}(T)} |\{w \mid Sm(v) \geq Sm(w), \mathit{leaf}(w)\}| = \sum_{w \in \mathit{leaves}(T)} |\{v \mid Sm(v) \geq Sm(w), \mathit{leaf}(w)\}|.$$

It is sufficient to show that for each leaf w the set $S'(w) := |\{v \mid Sm(v) \geq Sm(w), \mathit{leaf}(w)\}|$ is bounded by a constant. Since for each such w at most 8 boxes exist that are neighbors of $Sm(w)$ and have the same size as $Sm(w)$, we get that $|S'(w)| \leq 8$.

Finally, we prove Equation (5.22). By definition of $K(v)$, we know that for each leaf v of the tree $K(v) = K_1(v) \cup K_2(v)$ so that $K_1(v)$ is the set of all nodes w so that $Sm(v)$ and $Sm(w)$ are no neighbors and $w \in R(v)$. $K_2(v)$ is the set of all nodes w so that $Sm(v)$ and $Sm(w)$ are no neighbors, $Sm(\mathit{parent}(w))$ and $Sm(v)$ are neighbors, and an ancestor of w is contained in $R(v)$. Hence, we get that

$$\sum_{v \in \mathit{leaves}(T)} |\{w \mid w \in K(v)\}| = \sum_{v \in \mathit{leaves}(T)} |\{w \mid w \in K_1(v)\}| + \sum_{v \in \mathit{leaves}(T)} |\{w \mid w \in K_2(v)\}|.$$

The first term is bounded by $O(N)$ using Equation (5.17). To estimate the second term, we use that

$$\sum_{v \in leaves(T)} |\{w \mid w \in K_2(v)\}| = \sum_{w \in leaves(T)} |\{v \mid w \in K_2(v)\}|.$$

Let $S''(w) := |\{v \mid w \in K_2(v)\}|$, then it is sufficient to show that $|S''(w)|$ is bounded by a constant for an arbitrary node w of T. Suppose, there exists a leaf v of T so that $w \in K_2(v)$. Then, $Sm(parent(w))$ and $Sm(v)$ are neighbors. Since $parent(w)$ or an ancestor of $parent(w)$ is contained in $R(v)$, it is clear that $Sm(v) \geq Sm(parent(w))$. Since at most 8 such leaves v of size larger or equal than $Sm(parent(w))$ are neighbors of $Sm(parent(w))$, we get that $|S''(w)| \leq 8$, which completes the proof. □

Theorem 5.32 (Multipole Framework). *Suppose, $C = \{c_1, \ldots, c_N\}$ is a set of charged particles of unit charge that are placed at distinct positions $p(C) = \{p_1, \ldots, p_N\}$, $T = (V, E)$ is a reduced bucket quadtree with fixed constant leaf capacity l that is associated with C, and p is a fixed constant precision parameter. Then, Function* `MF` *approximates the repulsive force that acts on each particle c_i due to all other particles in C in $O(N)$ time using $O(N)$ memory.*

Proof. Using Lemma 5.31, we know that the sum of the length of all sets $R(v)$, $I(v)$, $D_1(v)$, $D_2(v)$, $D_3(v)$, and $K(v)$ is bounded above by $O(N)$. Additionally $O(p \cdot |V|)$ memory is needed to store the coefficients of the p-term multipole expansions and p-term local expansions. Since p is a constant and $|V| = O(N)$, the total memory requirements are $O(N)$.

Now we analyze the running time of Function `MF`. If T contains only one node, $|C| = N \leq l$. Since l is a constant, the exact naive force calculation in part 1 needs constant time. Otherwise, the initialization of the sets in part 1 needs $O(|V|) = O(N)$ time.

In part 2 of Function `MF` the coefficients of the p-term multipole expansions $M^p(v)$ are calculated for all leaves v, using Theorem 5.17. Let $m(v)$ denote the number of particles that are contained in $Sm(v)$ for each leaf v of T. Then, using Remark 5.25(a), this needs $\sum_{v \in leaves(v)} O(p \cdot m(v)) = O(p \cdot N) = O(N)$ time. The coefficients of the p-term multipole expansions of the interior nodes are obtained using Lemma 5.20. Using Remark 5.25(b) and the fact that $|V| = O(N)$ the total running time of this step is $O(p^2 \cdot |V|) = O(N)$.

In part 3.1 the sets $R(v)$, $I(v)$, $D_1(v)$, and $D_2(v)$ are found for a fixed node v. This is done by exploring a collection of $|R(parent(v)) \cup D_1(parent(v))|$ rooted subtrees, where each node in $R(parent(v)) \cup D_1(parent(v))$ is a root. The nodes in $R(parent(v)) \cup D_1(parent(v))$ are either assigned to one of the sets $R(v)$, $I(v)$, $D_1(v)$, and $D_2(v)$ directly, or their children are examined later. Since T is a reduced bucket quadtree, each interior node has at least two children. Hence, the total number of nodes that are visited in the exploration of all subtrees of the nodes in $R(parent(v)) \cup D_1(parent(v))$ is proportional to $|R(v)| + |I(v)| + |D_1(v)| + |D_2(v)|$. It follows from Lemma 5.31 that applying parts 3.1 to all nodes v of T needs $O(N)$ time.

In part 3.2 for each node $u \in I(v)$ converting the coefficients of the p-term multipole expansion $M^p(u)$ to the p-term local expansion $L^p(u)$ and adding these coefficients to the corresponding coefficients of p-term local expansion $L^p(v)$ of v can be done in $O(p^2)$ time using Lemma 5.21 (see Remark 5.25(c)). For each $u \in D_2(v)$ there exists at most l particles c_i that are contained in $Sm(u)$. For each such particle the work needed to calculate $M^p(c_i)$, to convert it to $L^p(c_i)$, and to add its coefficients to the corresponding coefficients of $L^p(v)$ is $O(p^2)$ using Theorem 5.17 and Lemma 5.21 (see Remarks 5.25(a) and (c)). Adding the coefficients of the shifted p-term local expansions $L^p(\mathit{parent}(v))$ of the parent of a fixed node v to $L^p(v)$ can be done in $O(p^2)$ time using Lemma 5.23 (see Remark 5.25(d)). Since we know from Lemma 5.31 that $\sum_{v \in V} I(v) \cup D_2(v) = O(N)$, $O(l \cdot p^2 \cdot N) = O(N)$ time is needed to apply part 3.2 on all nodes $v \in V$.

In part 3.3 the sets $D_3(v)$ and $K(v)$ are constructed by exploring a set of $|R(v)|$ rooted subtrees with roots in $R(v)$. Like in part 3.1 the total number of nodes that are visited in the exploration of all subtrees that are rooted at the nodes in $R(v)$ is proportional to $|D_3(v)| + |K(v)|$. Using Lemma 5.31, applying part 3.3 on all leaves v of T needs $O(N)$ time in total.

In part 4 the forces $F_{\mathit{direct}}(c_i)$, $F_{\mathit{local}}(c_i)$, and $F_{\mathit{multipole}}(c_i)$ are calculated for each particle c_i that is contained in a leaf v of T. The calculation of $F_{\mathit{local}}(c_i)$ needs $O(p)$ time using Corollary 5.24(b) (see Remark 5.25(e) and (f)). Calculating $F_{\mathit{direct}}(c_i)$ can be done in $O(l \cdot (|D_1(v)| + |D_3(v)| + 1))$ time. Calculating $F_{\mathit{multipole}}(c_i)$ can be done in $O(p \cdot |K(v)|)$ time using Corollary 5.24(a) (see Remark 5.25(e) and (f)). Adding $F_{\mathit{direct}}(c_i)$, $F_{\mathit{local}}(c_i)$, and $F_{\mathit{multipole}}(c_i)$ to obtain $F_{\mathit{rep}}(c_i)$ needs $O(1)$ time. Hence, it follows from Lemma 5.31 that the running time of part 4 is bounded by $\sum_{v \in \mathit{leaves}(T)} \sum_{c_i \in Sm(v)} O(p + l \cdot (|D_1(v)| + |D_3(v)| + 1) + p \cdot |K(v)| + 1) = O(N)$. □

5.3.5 Formal Description of The New Multipole Method

We now give a formal description of the new multipole method $\mathtt{NM}^2$. Since we have developed two ways for building up the reduced bucket quadtree, two variants of $\mathtt{NM}^2$ can be defined that are denoted by $\mathtt{NM}^2_{\mathtt{a}}$ and $\mathtt{NM}^2_{\mathtt{b}}$. An experimental study of $\mathtt{NM}^2_{\mathtt{a}}$ and $\mathtt{NM}^2_{\mathtt{b}}$ and an experimental comparison with other force-approximation methods will be given in Section 7.4.2. Note that the variable *crossover_point* in $\mathtt{NM}^2_{\mathtt{a}}$ and $\mathtt{NM}^2_{\mathtt{b}}$, respectively, is a constant. It should be chosen so that for almost all $N \leq$ *crossover_point* the naive direct force calculation is faster than our hierarchical multipole method in practice. In the complementary case, for almost all $N >$ *crossover_point* the hierarchical multipole method should be faster in practice. Our exact choice of this value will be given in Section 7.1.

Corollary 5.33 (The New Multipole Method (Version A)). *Suppose, $C = \{c_1, \ldots, c_N\}$ is a set of charged particles of unit charge that are placed at distinct positions $p(C) = \{p_1, \ldots, p_N\}$, and suppose that l, p are positive integer constants. Then, Function $\mathtt{NM}^2_{\mathtt{a}}$ approximates the repulsive force that acts on each particle c_i due to all other particles in C in $O(N \log N)$ time using $O(N)$ memory.*

Proof. The corollary follows directly from Theorem 5.11, Theorem 5.32, and the fact that *crossover_point* is a constant. □

Corollary 5.34 (The New Multipole Method (Version B)). *Suppose that* $C = \{c_1, \ldots, c_N\}$ *is a set of particles that are placed at distinct positions* $p(C) = \{p_1, \ldots, p_N\}$*, and suppose that* l, p *are positive integer constants. Suppose that the maximum and minimum of the Euclidean distances between any two particles in* C *are denoted by* d_{max} *and* d_{min}*.*

(a) Then, Function $\mathtt{NM}_{\mathtt{b}}^2$ *approximates the repulsive force that acts on each particle* c_i *due to all other particles in* C *in* $O(N \cdot \log(d_{max}/d_{min}))$ *time using* $O(N)$ *memory.*

(b) If it is additionally assumed that the particles in C *are placed at distinct positions* $p(C) = \{p_1, \ldots, p_N\}$ *on a regular square integer grid with a resolution which is polynomial in* N*, then Function* $\mathtt{NM}_{\mathtt{b}}^2$ *approximates the repulsive force that acts on each particle* c_i *due to all other particles in* C *in* $O(N \log N)$ *time using* $O(N)$ *memory.*

In both cases the best-case running time of Function $\mathtt{NM}_{\mathtt{b}}^2$ *is* $O(N)$*.*

Proof. The corollary follows directly from Theorem 5.13, Corollary 5.14, Theorem 5.32, and the fact that *crossover_point* is a constant. □

Function $\mathtt{NM}_a^2(C, l, p)$

input : a set $C = \{c_1, \ldots, c_N\}$ of charged particles of unit charge that are placed at distinct positions $p(C) = \{p_1, \ldots, p_N\}$ and positive integer constants l and p

output: a function $F_{rep} \colon C \longrightarrow \mathbb{R}^2$ so that $F_{rep}(c_i)$ is an approximation of the repulsive force that acts on c_i due to all other particles in C

```
begin
    if N ≤ crossover_point then
        foreach c_i ∈ C do
            F_rep(c_i) ← Naive_Direct_Force_Calculation;
    else
        T ← TC_a(C, l) ;
        F_rep ← MF(C, T, p);
end
```

5.3.6 Exception Handling

In the previous discussion of the force-approximation methods in Sections 5.2 and 5.3 we have for simplicity assumed that the positions p_i and p_j of any two charged particles p_i and p_j in $C = \{c_1, \ldots, c_N\}$ are distinct. Although extremely seldom in practice, it might

Function $\texttt{NM}_b^2(C, l, p)$

input : a set $C = \{c_1, \ldots, c_N\}$ of charged particles of unit charge that are placed at distinct positions $p(C) = \{p_1, \ldots, p_N\}$ and positive integer constants l and p

output: a function $F_{rep}\colon C \longrightarrow \mathbb{R}^2$ so that $F_{rep}(c_i)$ is an approximation of the repulsive force that acts on c_i due to all other particles in C

begin

  if $N \leq$ *crossover_point* **then**

    foreach $c_i \in C$ **do**

      $F_{rep}(c_i) \leftarrow$ `Naive_Direct_Force_Calculation`;

  else

    $T \leftarrow \texttt{TC}_\texttt{b}(C, l)$;

    $F_{rep} \leftarrow \texttt{MF}(C, T, p)$;

end

theoretically happen that several particles are placed at the same position. Thus, our new multipole method should also work in this case. We shortly comment which extensions have been made to handle the relaxed case.

We concentrate on $\texttt{NM}_\texttt{a}^2$ first: In the case that $N \leq$ *crossover_point* the force $F_{rep}^{c_j}(c_i)$ that acts on particle c_i at position p_i due to a particle c_j at position $p_j = p_i$ is zero (see Formula 5.1 in Section 5.1.2). However, in order to separate c_i and c_j, we set $F_{rep}^{c_j}(c_i) := F_{random}(c_i)$, where $F_{random}(c_i)$ is a random force vector. If $N >$ *crossover_point*, the reduced bucket quadtree $T = (V, E)$ with leaf capacity l has to be built up. If in the tree construction function $\texttt{TC}_\texttt{a}$ a node v is visited, we first calculate the smallest rectangle that contains all particles that are associated with v. This can be done in $O(1)$ time using the sorted $C_x(v)$ and $C_y(v)$ lists. If this rectangle is a point P, but the length of $C_x(v)$ is larger than one, it follows that all particles in $box(v)$ are placed at the same position. Hence, we replace this set of particles (that have unit charge) by one *group particle* of charge $|C_x(v)|$ that is placed at position P and proceed with the tree construction like in the usual case. Finally, we obtain a new set $C_{group} \subset C$ of charged particles and a reduced bucket quadtree T_{group} with leaf capacity l for the particles in C_{group}. Hence, the multilevel framework can be applied on T_{group}. In order to obtain $F_{rep}(c_i)$ for all particles $c_i \in C \setminus C_{group}$, we do the following: First $F_{rep}(c_i)$ is assigned the repulsive force $F_{rep}(c_{group})$ that acts on its associated group particle c_{group} due to the particles in C_{group}. Then, in order to separate the particles with same coordinates, we add a random force vector $F_{random}(c_i)$ to $F_{rep}(c_i)$. Since — in total — all these extra operations take at most $O(N)$ time, and the additional memory for storing the group particles is bounded above by $O(N)$, Corollary 5.33 holds also in the relaxed case in which same particle positions are allowed.

We now concentrate on $\texttt{NM}_\texttt{b}^2$. The method $\texttt{NM}_\texttt{b}^2$ differs from method $\texttt{NM}_\texttt{a}^2$ only in the tree-construction method. Hence, the same techniques as in the relaxed version of $\texttt{NM}_\texttt{a}^2$ can be used. It remains to be described how the reduced bucket quadtree T_{group} for C_{group} is

constructed. This is done as follows: Suppose, in Function $\texttt{TC}_\texttt{b}$ the list L^i (containing all leaves of the actual tree that contain more than l particles) has been constructed at the last recursion level i. All other newly created leaves contain at most l particles. Let S^i be the set of these newly created leaves. We first test for each $v \in L^i \cup S^i$ if the smallest rectangle that covers the particles which are associated with v is a point P. If this is the case, v is made a leaf of T_{group} containing one group particle c_{group} that is associated with the particles that are placed at point P. Furthermore — in contrast to the original tree construction method — no subtree $T^{i+1}(v)$ rooted at v is constructed. At each recursion level i the total running time for creating the smallest rectangles that cover the particles associated with each leaf $v \in L^i \cup S^i$ is bounded by $O(N)$. Since the running time of $\texttt{TC}_\texttt{b}$ is proportional to $N \cdot |\mathit{recursion\_levels}|$) (see proof of Theorem 5.13), Theorem 5.13 and Corollary 5.14 hold even in the relaxed case in which identical particle positions are allowed. Hence, Corollary 5.34 holds in the relaxed case, too.

5.4 Running Time of the Force-Calculation Step

Theorem 5.35 (Force-Calculation Step). *Suppose that $G = (V, E, l^{\text{zero}})$ is a positive-weighted undirected simple graph, $p(V) := (p_v)_{v \in V}$ is an initial placement of the nodes of G, G is the multilevel graph at level $l \in \{0, \ldots, k\}$ in a series of $k+1$ multilevel graphs, and d_1 and d_2 are integer constants.*

(a) If in the framework of Algorithm `Embedder` *the force-approximation method* $\texttt{NM}^2_\texttt{a}$ *is used, Algorithm* `Embedder` *generates a straight-line drawing $\Gamma(G)$ of G in $O(|V| \log |V| + |E|)$ time using $O(|V|)$ memory.*

(b) If in the framework of Algorithm `Grid_Embedder` *the force-approximation method* $\texttt{NM}^2_\texttt{b}$ *is used, Algorithm* `Grid_Embedder` *generates a straight-line drawing $\Gamma(G)$ of G with integer node positions in the range $[-d_1 N^{d_2}, d_1 N^{d_2}] \times [-d_1 N^{d_2}, d_1 N^{d_2}]$ in $O(|V| \log |V| + |E|)$ time using $O(|V|)$ memory. The best-case running time of this method is $O(|V| + |E|)$.*

Proof. This claim follows directly from Lemma 5.2 and Corollaries 5.3, 5.33, and 5.34(b). □

Chapter 6

The Postprocessing Step and Formal Description of FM^3

> Willst du, dass wir mit hinein
> in das Haus dich bauen,
> lass es dir gefallen, Stein,
> dass wir dich behauen! [1]

In Section 6.1 we will describe a simple postprocessing step that is used to improve the quality of the drawings, before we will summarize our main theoretical results concerning FM^3 in Section 6.2.

6.1 The Postprocessing Step

6.1.1 Motivation and Goals

The previously introduced steps of Algorithm FM^3 would be sufficient to build an algorithm that efficiently generates a drawing of any given graph $G = (V, E)$. However, in practice, the drawing quality of the generated drawings can be improved by introducing a postprocessing step.

In particular, suppose that C is the set of components of G, and G' is the positive-weighted undirected simple graph that is obtained from G in the preprocessing step. Furthermore, let $G'_1, \ldots, G'_{|C|}$ be the maximal connected subgraphs of G' and $\Gamma(G'_1), \ldots, \Gamma(G'_{|C|})$ be the drawings of $G'_1, \ldots, G'_{|C|}$ that are generated in the multilevel step.

In the postprocessing step the drawings $\Gamma(G'_1), \ldots, \Gamma(G'_{|C|})$ are modified in order to obtain more pleasing drawings concerning the desired edge length. Remember that it is one of the most important goals of all force-directed algorithms that the desired edge length should be preserved as well as possible in the drawing.

[1] Friedrich Rückert

Although preserving the desired edge length of each edge exactly is impossible in general (see Section 1.2.1), it is easy to generate a drawing in which the average of the desired edge length is equal to the average of the edge length in the generated drawing ($\rightarrow$ *goal 1*). Furthermore, one could try to keep the edge length of an edge e of $\Gamma(G'_i)$ close to the desired edge length of e ($\rightarrow$ *goal 2*). These are exactly the two goals of the postprocessing step.

6.1.2 The Algorithm

Suppose, we have given a connected positive-weighted undirected simple graph $G = (V, E, l^{zero})$ (in which the weight l^{zero} of an edge $e \in E$ is the zero-energy lengths of the corresponding spring in our force model and l^{zero} reflects the desired edge lengths of e) and a drawing $\Gamma(G)$. If $|E| = 0$ nothing is done, otherwise the postprocessing step works in three phases:

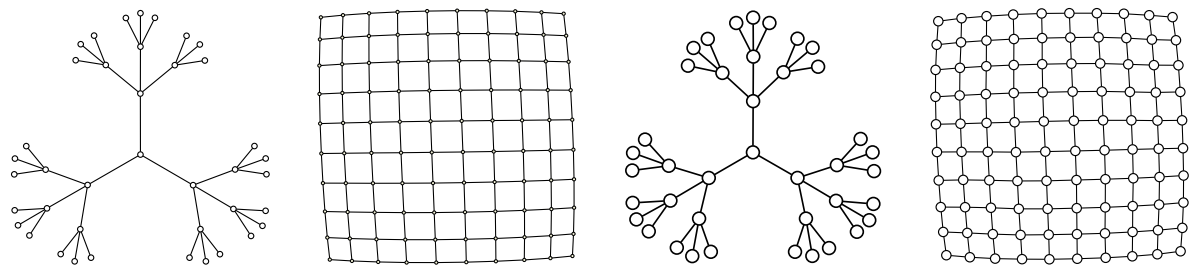
First, the drawing $\Gamma(G)$ is scaled so that goal 1 is reached. This is done as follows: If *av_des* is the average of the desired edge length and *av_real* is the average of the edge length in $\Gamma(G)$, then a drawing $\Gamma'(G)$ in which goal 1 holds can be generated by setting $p'(v) := (av\_des/av\_real) \cdot p(v)$. Here $p(v)$ and $p'(v)$ denote the position of a node $v \in V$ in $\Gamma(G)$ and $\Gamma'(G)$, respectively.

Second, in order to reach goal 2, the force-directed single-level algorithm that was used in the force-calculation step (i.e., Algorithm `Embedder` or Algorithm `Grid_Embedder`) is used to generate a drawing of G starting with the initial placement $p'(V)$ that corresponds to $\Gamma'(G)$. In contrast to the original version of `Embedder` and `Grid_Embedder`, we reduce the repulsion factor λ_{rep} extremely, while the spring stiffness factor λ_{spring} is increased. The maximum number of iterations `Max_Iter` in the repeat-loop of the force-directed single-level algorithms is a small constant. For later use, the modified algorithm is denoted by $\mathtt{A}_{\mathtt{post}}$, and the exact values of these parameters will be defined in Section 7.1.

The reason for the change of λ_{spring} and λ_{rep} is as follows: On the one hand (like demonstrated in Section 2.2.2), the combination of the spring forces and the repulsive forces is important to minimize the probability that nodes are placed at exactly the same position. Furthermore, in practice, a high value of λ_{rep} turned out to be indispensable to obtain a fast convergence behavior in the force-calculation step. On the other hand, however, the repulsive forces lengthen the springs. But — since the drawings that are generated in the multilevel step have to be improved regarding the edge length here — giving the spring forces significantly more importance than the repulsive forces has turned out to be a reasonable and simple way to optimize the drawing according to goal 2.

Third, in order to guarantee that goal 1 is reached even after the computations in the second step, the drawing is scaled again.

Figure 6.1 shows example drawings that demonstrates the effect of the postprocessing step. Function `Postprocessing_Step` shows the pseudocode of the postprocessing step, and quantitative experimental results to the postprocessing step can be found in Section 7.5.



(a) (b) (c) (d)

Figure 6.1: (a) A tree and (b) a grid graph that are drawn by $\texttt{FM}^3$ without using the postprocessing step. The same graphs that are drawn by $\texttt{FM}^3$ under the usage of the postprocessing step are shown in (c) and (d).

6.1.3 Formal Description of the Postprocessing Step

Function $\texttt{Postprocessing\_Step}(G, \Gamma(G), \texttt{A}_{\texttt{post}})$

input : a connected positive-weighted undirected simple graph $G = (V, E, l^{zero})$, a drawing $\Gamma(G)$, and a variation of the force-directed single-level algorithm that has been used in the force-calculation step denoted by $\texttt{A}_{\texttt{post}}$

output: a straight-line drawing $\Gamma'(G)$ of G in which the average edge length equals to $\sum_{i=1}^{|E|} l^{zero}(e)$

begin
  if $|E| = 0$ **then** $\Gamma'(G) \leftarrow \Gamma(G)$;
  else
    let $p(V)$ denote the placement of the nodes in $\Gamma(G)$;
    $p(V) \leftarrow \texttt{Scale}(p(V))$;
    $\Gamma'(G) \leftarrow \texttt{A}_{\texttt{post}}(G, p(V))$;
    let $p'(V)$ denote the placement of the nodes in $\Gamma'(G)$;
    $\Gamma'(G) \leftarrow p'(V) \leftarrow \texttt{Scale}(p'(V))$;
end

Theorem 6.1 (Postprocessing Step). *Suppose that $G = (V, E, l^{zero})$ is a connected positive-weighted undirected simple graph, $\Gamma(G)$ is a drawing of G, and $\texttt{A}_{\texttt{post}}$ is a variation of the force-directed single-level algorithm that has been used in the force-calculation step (either Algorithm* Embedder *with* $\texttt{NM}_{\texttt{a}}^2$ *or Algorithm* Grid_Embedder *with* $\texttt{NM}_{\texttt{b}}^2$*), but uses the set of parameters that are defined in Section 6.1.2. Then, the postprocessing step generates a straight-line drawing $\Gamma'(G)$ so that in $\Gamma'(G)$ the average edge length equals to $\sum_{i=1}^{|E|} l^{zero}(e)$ in $O(|V| \log |V| + |E|)$ time using $O(|V| + |E|)$ memory. If $\texttt{A}_{\texttt{post}}$ is a variation of Algorithm* Grid_Embedder *with force-approximation method* $\texttt{NM}_{\texttt{b}}^2$*, the best-case running time is $O(|V| + |E|)$.*

Proof. The two scaling phases need linear time. The remainder of the proof follows from Theorem 5.35. □

6.2 Formal Description of FM^3

As a result of the previous chapters we can now formulate the pseudocode of FM^3 and analyze its asymptotic running time. Experimental studies of FM^3 and a practical comparison of FM^3 with other state-of-the-art methods for drawing large graphs can be found in Sections 7.6 and 7.7, respectively.

Algorithm 14: The Fast Multipole Multilevel Method (FM^3)

input : an (un)weighted (un)directed graph $G = (V, E)$ that may contain nodes of different fixed sizes and shapes and a desired aspect ratio r of the drawing area

output: a drawing $\Gamma(G)$ of G with non-overlapping components; except self-loops (that are drawn as loops) all edges are drawn as straight lines; multiple edges are drawn parallel; directed edges are drawn as arcs

```
begin
    G' ← Preprocessing_Step(G);
    let C denote the set of components of G';
    begin{Divide_Step}
        Find the maximal connected subgraphs G'_1, ..., G'_|C| of G';
    end
    foreach G'_i ∈ {G'_1, ..., G'_|C|} do
        Γ(G'_i) ← Multilevel_Step(G'_i, A_single);
    foreach G'_i ∈ {G'_1, ..., G'_|C|} do
        Γ'(G'_i) ← Postprocessing_Step(G'_i, Γ(G'_i), A_post);
    begin{Impera_Step}
        Use the drawings Γ(G'_i), ..., Γ(G'_|C|) of G'_1, ..., G'_|C| to obtain a drawing Γ(G)
        of G that fits into a (small) rectangle of aspect ratio r;
    end
end
```

Theorem 6.2 (Fast Multipole Multilevel Method). *Suppose, $G = (V, E)$ is an (un)weighted (un)directed graph that may contain nodes of different fixed sizes and shapes. Then, the* Fast Multipole Multilevel Method (FM^3) *generates a drawing $\Gamma(G)$ of G with non-overlapping components, in which except self-loops (that are drawn as loops), all edges are drawn as straight lines. The worst-case running time of* FM^3 *is $O(|V| \log |V| + |E|)$, and the memory requirements are $O(|V| + |E|)$.*

Proof. Using Theorems 3.1 and 3.4 The total running time of FM^3 is given by

$$t_{\texttt{FM}^3} = O\left(|V| + |E| + |C|\log|C| + \sum_{i=1}^{|C|} t_{mult}(|V_i'|,|E_i'|) + t_{post}(|V_i'|,|E_i'|)\right), \tag{6.1}$$

where $t_{mult}(|V_i'|,|E_i'|)$ and $t_{post}(|V_i'|,|E_i'|)$ denote the time that is needed to generate drawings $\Gamma(G_i')$ and $\Gamma'(G_i')$ of $G_i' = (V_i', E_i')$ in the multilevel step and postprocessing step, respectively. Using Theorem 4.8 Equation (6.1) simplifies to

$$t_{\texttt{FM}^3} = O\left(|V| + |E| + |C|\log|C| + \sum_{i=1}^{|C|} t_{single}(|V_i'|,|E_i'|) + t_{post}(|V_i'|,|E_i'|)\right), \tag{6.2}$$

where $t_{single}(|V_i'|,|E_i'|)$ is the time that is needed by the force-directed single-level algorithm $\texttt{A}_{\texttt{single}}$ to draw $G_i' = (V_i', E_i')$. Two alternatives can arise. Per default, $\texttt{A}_{\texttt{single}}$ is Algorithm `Grid_Embedder` with the force-approximation method $\texttt{NM}_{\texttt{b}}^2$. In the second case, $\texttt{A}_{\texttt{single}}$ can be chosen as the Algorithm `Embedder` with the force-approximation method $\texttt{NM}_{\texttt{b}}^2$. In both cases (using Theorem 5.35, Theorem 6.1 and the fact that $\sum_{i=1}^{|C|}|V_i'| = |V|$ and $\sum_{i=1}^{|C|}|E_i'| \leq |E|$) Equations (6.2) simplifies to

$$\begin{aligned} t_{\texttt{FM}^3} &= O\left(|V| + |E| + |C|\log|C| + \sum_{i=1}^{|C|} |V_i'|\log|V_i'| + |E_i'|)\right) \\ &= O(|V|\log|V| + |E|). \end{aligned}$$

Using Theorems 3.1 and 3.4 the memory requirements of FM^3 are given by

$$m_{\texttt{FM}^3} = O\left(|V| + |E| + |C| + \sum_{i=1}^{|C|} m_{mult}(|V_i'|,|E_i'|) + m_{post}(|V_i'|,|E_i'|)\right), \tag{6.3}$$

where $m_{mult}(|V_i'|,|E_i'|)$ and $m_{post}(|V_i'|,|E_i'|)$ denote the memory requirements for generating drawings $\Gamma(G_i')$ and $\Gamma'(G_i')$ of $G_i' = (V_i', E_i')$ in the multilevel step and postprocessing step, respectively. Using Theorem 4.8, Theorem 5.35, and the fact that $\sum_{i=1}^{|C|}|V_i'| = |V|$ and $\sum_{i=1}^{|C|}|E_i'| \leq |E|$, Equations (6.3) simplifies to

$$\begin{aligned} m_{\texttt{FM}^3} &= O\left(|V| + |E| + |C| + \sum_{i=1}^{|C|} |V_i'| + |E_i'| + m_{single}(|V_i'|,|E_i'|) + m_{post}(|V_i'|,|E_i'|)\right) \\ &= O\left(|V| + |E| + |C| + \sum_{i=1}^{|C|} |V_i'| + |E_i'|)\right) \\ &= O(|V| + |E|), \end{aligned}$$

where $m_{single}(|V_i'|,|E_i'|)$ is the memory that is needed by the force-directed single-level algorithm $\texttt{A}_{\texttt{single}}$. □

Chapter 7

Experimental Results

> Es ist nicht genug zu wissen,
> man muss es auch anwenden.
> Es ist nicht genug zu wollen,
> man muss es auch tun. [1]

In this chapter we will examine the new graph-drawing method FM^3 in practice. After giving some general remarks on the test-environment, implementation, and parameter settings in Section 7.1, we will present experimental results concerning the most important parts of the new graph-drawing algorithm in Sections 7.2 to 7.5. In Section 7.6 we will study the running times of FM^3 and the drawings that are generated by FM^3 on a wide range of graphs. An experimental comparison of FM^3 with some of the fastest algorithms for drawing general graphs will be given in Section 7.7. Further experiments will be presented in Section 7.8.

7.1 General Remarks

The method FM^3 was implemented in C++ within the framework of *AGD* [75] that itself is based on the libraries *LEDA* [95] and *ABACUS* [78]. The functionalities of ABACUS were not used in the context of this dissertation.

Unless otherwise noted, all programs in this chapter were implemented by us in C++ using LEDA [95]. All random numbers were generated with the random number generator that is provided in *The Stanford GraphBase* [84], while all random graphs were generated with the random graph generator that is provided by LEDA [95].

Besides several other classes of artificially generated graphs we tested some graphs from real-world applications. In particular, we selected all graphs from the *AT&T graph library* [6] and from C. Walshaw's graph collection [131] that contain between 500 and

[1] Johann Wolfgang von Goethe

200000 nodes. We added a disconnected graph that describes a protein structure with 797 nodes that we obtained from Carsten Gutwenger of the graph drawing group of the foundation *caesar* [57], and a graph that describes a social network of 2113 people that we obtained from Carola Lipp of the Universität Göttingen [90]. This class of real-world instances is referred here as *real-world* graph collection. Details concerning the structure of the contained graphs will be introduced later.

All experiments were performed on a 2.8 GHz Intel Pentium 4 PC with one gigabyte of memory running Linux. Some implementations of other graph-drawing methods that are used in Section 7.7 are executable on Microsoft Windows platforms only. These algorithms were tested on the same machine, but using Windows instead of Linux.

Unless otherwise stated, all tests of $\texttt{FM}^3$ (and all tests of modules of $\texttt{FM}^3$) were performed with the same type of standard parameters. In particular, the desired edge length of each edge of unweighted graphs was set to 100, the desired aspect ratio of the drawing area was set to 1, and each component was allowed to be rotated with 10 specified angles in the rotation phase of the impera step. In the multilevel step, the size of the random sample for selecting the sun nodes with the `Select_By_Star_Mass` strategy was set to 20, and the constants c, d, and s of the `Stopping_Criterion` (see Section 4.2.2) were set to $50, 5$, and 1.25, respectively. In the force-calculation step, we chose Algorithm `Grid_Embedder` with the force-approximation method $\texttt{NM}^2_\texttt{b}$. The parameters $\texttt{Max\_Iter}(0, k)$, $\texttt{Max\_Iter}(k, k)$, and the threshold t in Algorithm `Grid_Embedder` were set to 300, 30, and 0.01, respectively. The spring stiffness factor λ_{spring}, the repulsion factor λ_{rep}, and the constant δ were set to 1, 1, and 0.05, respectively (see Section 5.1). The leaf capacity of a leaf in the quadtree and the precision parameter p for calculating the p-term multipole expansions were fixed to 25 and 4, respectively. The constant *crossover_point* in Functions $\texttt{NM}^2_\texttt{a}$ and $\texttt{NM}^2_\texttt{b}$ was set to 175. In the postprocessing step, the repulsion factor λ_{rep} was set to 0.01, the spring stiffness factor λ_{spring} was set to 2, and the maximum number of iterations `Max_Iter` in the repeat-loop of the force-directed single-level algorithms was set to 20.

7.2 Experiments with the Divide-Et-Impera Strategy

We want to compare the divide-et-impera strategy of $\texttt{FM}^3$ (used for creating layouts of disconnected graphs) with other methods. Since in the divide step the components can be easily found in linear time by depth-first search, we can concentrate on the impera step. We will compare our impera step with the $O(|C| \log |C|)$ tiling methods of U. Dogrusoz [32] that we have already introduced in Section 3.2.2. Here C denotes the set of components of the graph. We implemented U. Dogrusoz's tiling method (denoted by $\texttt{Tile}_1$ here), the variant of $\texttt{Tile}_1$ that sorts each component by non-increasing height in a preprocessing step ($\texttt{Tile}_2$), and our impera step that is an extension of Dogrusoz's tiling method (denoted by $\texttt{Tile}_3$ here). It is important to note that the experimental results in [44] demonstrate that $\texttt{Tile}_2$ generates more compact drawings than computationally more time-consuming algorithms. Therefore, the tiling method $\texttt{Tile}_2$ can serve as a good benchmark.

We tested these three methods on three classes of graphs. The first class of graphs

(denoted by *rand*$_1$ graphs) are 11 graphs with an increasing number of components. The smallest rand$_1$ graph consists of one component, while the largest rand$_1$ graph consists of 2^{10} components. Each component is a connected graph containing a random number of nodes in the range 1 to 200.

The second class of graphs are 11 random graphs with a fixed number of 100 components, but with an increasing number of nodes that are contained in each component. They are called *rand*$_2$ graphs. Each component of the smallest rand$_2$ graph contains either 1 or 2 nodes, while each component of the largest rand$_2$ graph contains a random number of nodes in the range 1 to 2^{11}.

The third class of graphs are all graphs out of our real-world graph collection that contain more than 500 nodes and at least 10 components. These 12 graphs contain between 797 and 44775 nodes and 10 to 829 components. They are referred here as *disconnected real-world* graphs.

We determined two desired aspect ratios (1 and 2) and tested the three tiling methods on all classes of graphs and both desired aspect ratios. In all cases we used our multilevel step and postprocessing step for generating a layout of each component. We measured the running times of the divide steps, the running times of the tiling methods in the impera step, and the used aspect-ratio areas.

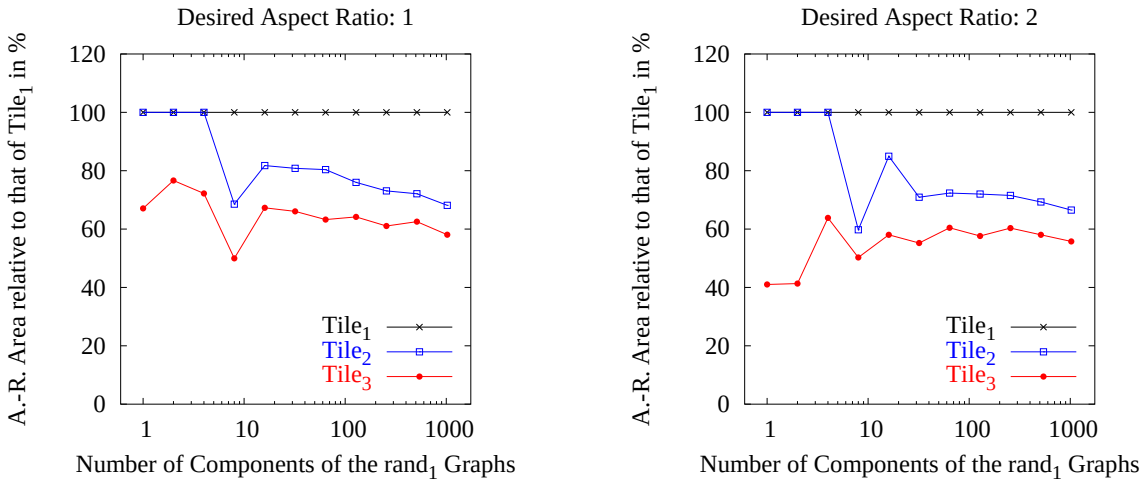


Figure 7.1: Used aspect-ratio area of `Tile`$_1$, `Tile`$_2$, and `Tile`$_3$ relative to the used aspect-ratio area of `Tile`$_1$ for the drawings of the rand$_1$ graphs. The desired aspect ratios are 1 (left) and 2 (right).

Figure 7.1 demonstrates that for all rand$_1$ graphs and both desired aspect ratios the drawings of `Tile`$_2$ are at least as compact as those of `Tile`$_1$. This is in agreement with the experimental results presented in [32]. The drawings produced by `Tile`$_3$ are more compact than the previous ones and need only 50 to 77% of the aspect-ratio area of `Tile`$_1$, when setting the desired aspect ratio to 1. If the desired aspect ratio is 2, only 41% to 64% of the aspect-ratio area of `Tile`$_1$ are needed.

Figure 7.2 presents the results of the rand$_2$ graphs. The drawings of those graphs that

are generated with `Tile3` use less aspect-ratio area than those that are generated using `Tile2` and `Tile1` for both desired aspect ratios. For the desired aspect ratios 1 and 2 `Tile3` uses 51 to 79% and 50 to 72% of the aspect-ratio area of `Tile1`, respectively.

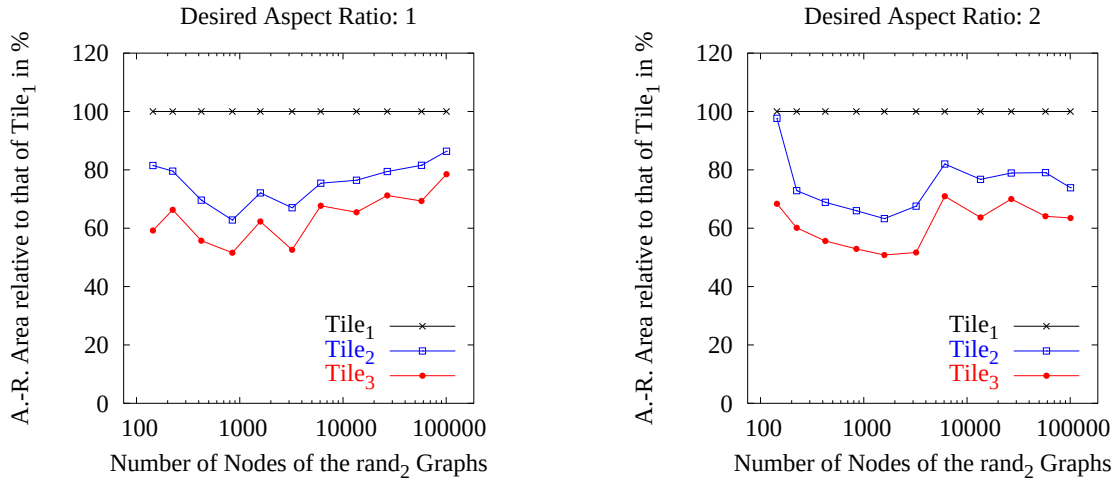


Figure 7.2: Used aspect-ratio area of `Tile1`, `Tile2`, and `Tile3` relative to the used aspect-ratio area of `Tile1` for the drawings of the rand$_2$ graphs. The desired aspect ratios are 1 (left) and 2 (right).

The drawings of the disconnected real-world graphs (see Figure 7.3) that are generated by `Tile2` are at least as compact as those of `Tile1` for both desired aspect ratios. Like before, the drawings that are positioned by `Tile3` are at least as compact as the other ones, with only one exception.

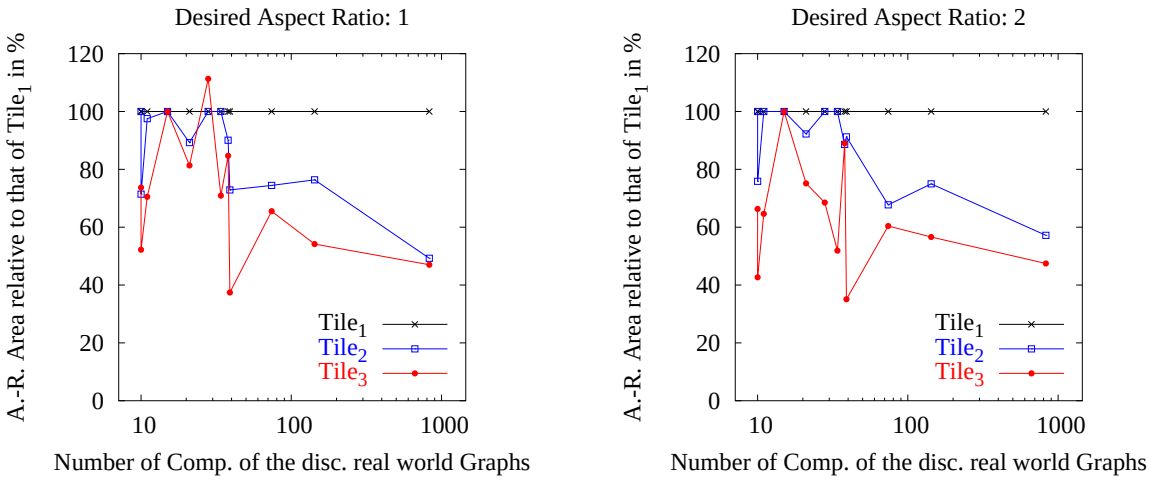


Figure 7.3: Used aspect-ratio area of `Tile1`, `Tile2`, and `Tile3` relative to the used aspect- ratio area of `Tile1` for the drawings of the disconnected real-world graphs. The desired aspect ratios are 1 (left) and 2 (right).

The experiments show that $\texttt{Tile}_3$ reduces the used aspect-ratio area significantly in comparison with $\texttt{Tile}_1$ and $\texttt{Tile}_2$ for nearly all tested graphs. The largest running times of $\texttt{Tile}_1$, $\texttt{Tile}_2$, and $\texttt{Tile}_3$ are 0.11 seconds, 0.11 seconds, and 0.23 seconds, respectively. The total running times of the divide step and impera step are bounded above by 0.41 seconds for all tested methods and graphs.

7.3 Experiments with the Multilevel Step

In this section we will examine if the multilevel step of $\texttt{FM}^3$ is suitable for finding an energy-minimal configuration of the nodes of a given graph in our force model fast. This is done by comparing $\texttt{FM}^3$ with a single-level algorithm (SINGLE) that is given by replacing the multilevel step in the framework of $\texttt{FM}^3$ with the single-level algorithm that is used in $\texttt{FM}^3$ (namely Grid_Embedder with force-approximation method $\texttt{NM}^2_\texttt{a}$).

We modified the stopping criterion of the force-calculation step for both methods. In particular, we stopped SINGLE if and only if in the actual iteration the average strength of the forces that acted on each node was smaller than a threshold t. For each graph G, we set t to 10^{-4} times the length of the smallest square box that covers the drawing of G in the actual iteration. Similarly, we modified $\texttt{FM}^3$ by changing the stop criterion of the force-calculation step on the lowest multilevel. This variation of $\texttt{FM}^3$ is denoted by MULT. In particular, MULT terminated if and only if the average strength of the forces in the actual iteration of the force-calculation step of the lowest multilevel was smaller than the threshold t. The stopping criterion in the force-calculation step in the other multilevels kept unchanged and, hence, was bounded above by a constant.

We selected two kinds of test graphs: Random connected graphs and graphs that describe regular two-dimensional square grids that we call *grid* graphs. The sizes of the test graphs range between 100 to 100000 nodes.

Both algorithms SINGLE and MULT were started using a random initial placement. We measured the number of iteration of SINGLE, the number of iterations of MULT on the lowest multilevel, and the total running times of SINGLE and MULT for all tested graphs.

The left graphic of Figure 7.4 demonstrates that MULT needs significantly fewer iterations in the lowest multilevel than SINGLE for both kinds of graphs. Furthermore, the number of iterations of SINGLE grows (although not monotonously) with the sizes of the graphs. For the grid graphs and random graphs SINGLE needs up to 2612 and 1303 iterations, respectively, until it terminates. MULT needs at most 170 and 143 iterations in the last multilevel for the grid graphs and random graphs, respectively. Therefore (see right graphic of Figure 7.4), for both kinds of tested graphs the number of iterations of MULT in the lowest multilevel is less than 18.3% of the iterations of SINGLE. For the graphs that contain more than 10000 nodes less than 7.7% of the iterations of SINGLE are needed by MULT in the lowest multilevel.

As a consequence of this, the total running times of MULT are faster than those of SINGLE for all tested graphs. The running time of MULT relative to the running time of SINGLE is ranging between 47.7% for the smallest grid graph and 2.5% for the largest grid

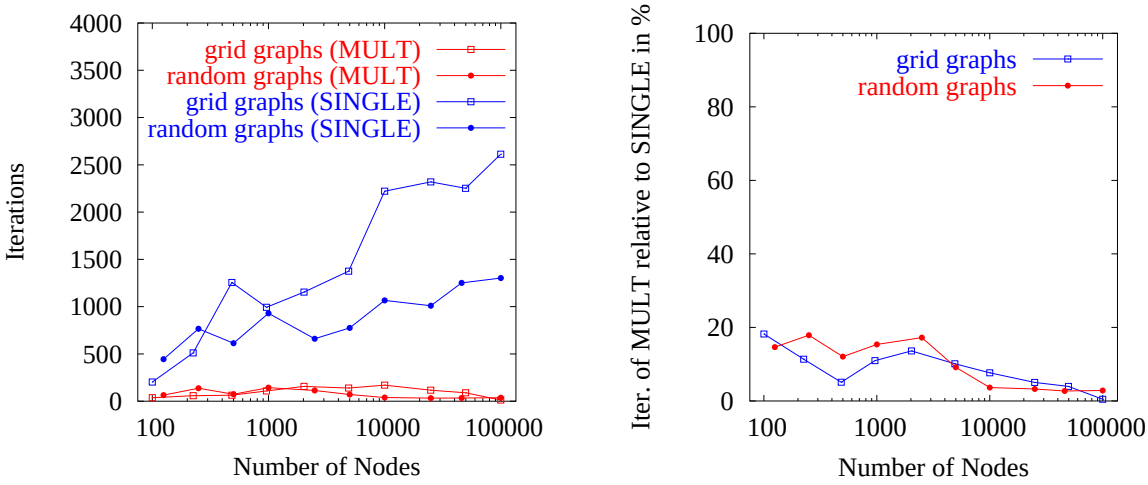


Figure 7.4: (left) The number of iterations of `SINGLE` and the number of iterations of `MULT` in the lowest multilevel for grid and random graphs. (right) The number of iterations at the lowest multilevel of `MULT` relative to the iterations needed by `SINGLE` in percent for grid and random graphs.

graph. The range for the random graphs is 76.4% to 17.7%.

In summary, these experiments demonstrate that the multilevel step accelerates the convergence behavior for both regular structured and random graphs.

7.4 Experiments with the Force-Calculation Step

In this section we will compare the running time and accuracy of several force-approximation methods that we have introduced in Sections 5.2 and 5.3. In particular, we implemented the PIC-code used in the `Grid-Variant Algorithm` of Fruchterman and Reingold [47] (`FrRe`), the method of Barnes and Hut [8] (`BaHu`), the `Fast_Multipole_Method` of Greengard and Rokhlin [59] (`GrRo`), the method of Aluru et al. [2] (`Al`), and the two variants of the new multipole method ($\texttt{NM}_\texttt{a}^2$ and $\texttt{NM}_\texttt{b}^2$). Finally, we implemented the naive exact force-calculation algorithm (`Ex`) that is used as a benchmark.

Since one essential part of all hierarchical force-approximation methods is the construction of the used tree data structure, we will focus on the running times of the tree construction phases of these methods for different distributions of N particles first.

7.4.1 Experiments with the Tree Construction Methods

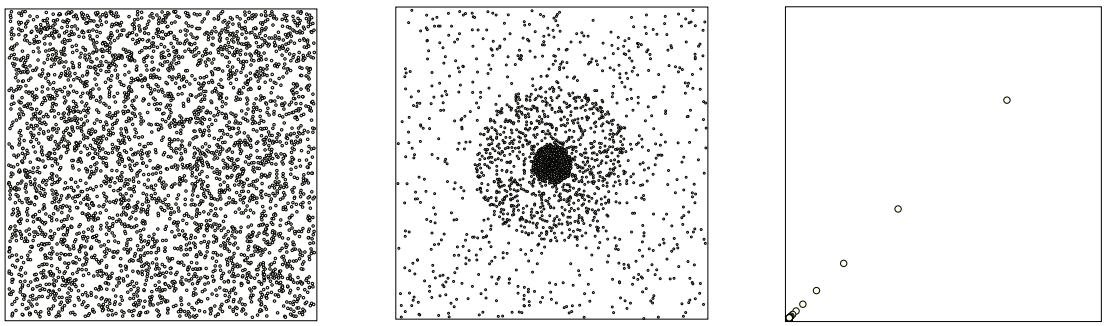
We compared the quadtree construction method used in `BaHu`, the method used in `GrRo` for constructing a truncated pseudo quadtree of depth $max\{\lfloor \log_4 N \rfloor, 1\}$, the method used in `Al` for constructing a reduced quadtree, and the two different methods $\texttt{TC}_\texttt{a}$ and $\texttt{TC}_\texttt{b}$ for constructing a reduced bucket quadtree with leaf capacity $l = 25$.

We tested the tree construction phases of these methods on three different classes of distributions of N particles. For each distribution we let N range from 8000 up to 128000.

We first distributed the N particles randomly with uniform probability within the square $[0,1] \times [0,1]$ and — following standard practice — call these distributions *uniform* distributions.

The second class of distributions are *non-uniform* distributions. They were created by distributing 20% of the particles randomly with uniform probability within the box $[0,1] \times [0,1]$. Another 20% of the particles were randomly distributed within a disc of radius $\frac{1}{4}$ with center $(\frac{1}{2}, \frac{1}{2})$. Another 20% of the particles were distributed within a disc of radius $\frac{1}{16}$ with center $(\frac{1}{2}, \frac{1}{2})$. The rest of the particles was distributed analogue within discs of radii $\frac{1}{64}$ and $\frac{1}{256}$ with center $(\frac{1}{2}, \frac{1}{2})$.

Finally, we wanted to construct a class of distributions that is similar to the converging distribution described in Section 1.3. Unfortunately converging distributions cannot be generated for arbitrary large N in reality, since the machine accuracy is bounded. Therefore, we constructed *quasi-converging* distributions by distributing the N particles non-uniform on the line connecting $(0,0)$ and $P := (10^{25}, 10^{25})$. In particular, the first node was placed at position $\frac{3}{4}P$. The i-th node was placed at position $p_i := \frac{3}{4}\frac{P}{2^{i-1}}$, for each i with x-coordinate of $p_i \geq Q := 10^{-25}$. The other nodes were placed uniformly on the line connecting $(0,0)$ and Q. Figure 7.5 illustrates the three classes of distributions.



(a) (b) (c)

Figure 7.5: (a) A uniform and (b) a non-uniform distribution of 4000 particles. (c) A quasi-converging distribution of 125 particles.

The experimental results are displayed in Table 7.1. For each size of N, each class of distributions, and each algorithm, the reported running times are the average times of 100 tests.

The construction of the truncated pseudo quadtree in the method `GrRo` is trivial and needs only negligible time for each size of N. Among the other methods, $\texttt{TC}_\texttt{b}$ is the fastest method for uniform and non-uniform distributions and all sizes of N. For these distributions, the tree construction phase of `BaHu` is roughly a factor 3 slower than $\texttt{TC}_\texttt{b}$. For uniform and non-uniform distributions, $\texttt{TC}_\texttt{a}$ and the tree construction phase of `Al` are more than a factor 7 slower than $\texttt{TC}_\texttt{b}$. As expected, for quasi-converging distributions, the tree construction times of $\texttt{TC}_\texttt{b}$ and `BaHu` reach or exceed those of `Al` and $\texttt{TC}_\texttt{a}$, since the

latter are $O(N \log N)$ methods. $\texttt{TC}_\texttt{a}$ is faster for quasi-converging distributions than for uniform and non-uniform distributions and same sizes of N. This is not surprising, since it can be shown that for converging distributions $\texttt{TC}_\texttt{a}$ needs $O(N \log N)$ time for sorting the particle lists S_x and S_y but only $O(N)$ time for the other parts of the tree construction phase described in Section 5.3.2. In contrast to uniform and non-uniform distributions, $\texttt{TC}_\texttt{b}$ is faster than `Al` for quasi-converging distributions and all sizes of N.

Class of Distributions	Number of Particles	CPU Time in Seconds for the Tree Construction by:				
		BaHu	GrRo	Al	$\texttt{TC}_\texttt{a}$	$\texttt{TC}_\texttt{b}$
uniform	8000	0.04	< 0.01	0.18	0.18	0.01
	16000	0.08	0.01	0.38	0.45	0.03
	32000	0.18	0.02	0.81	1.11	0.06
	64000	0.40	0.03	1.73	2.42	0.12
	128000	0.88	0.06	3.67	5.75	0.30
non-uniform	8000	0.05	< 0.01	0.16	0.15	0.02
	16000	0.11	0.01	0.33	0.37	0.03
	32000	0.22	0.01	0.71	0.92	0.08
	64000	0.48	0.02	1.53	2.02	0.15
	128000	1.02	0.05	3.25	4.87	0.34
quasi-converging	8000	0.53	< 0.01	0.30	0.14	0.22
	16000	1.03	< 0.01	0.55	0.35	0.44
	32000	1.96	0.01	1.13	0.66	0.71
	64000	3.96	0.02	2.19	1.54	1.49
	128000	7.97	0.03	4.57	3.45	2.75

Table 7.1: Comparison of the running times for building up the tree data structures for three different classes of distributions.

It can be summarized that since the hidden constants of the $O(N \log N)$ tree construction phases of `Al` and $\texttt{TC}_\texttt{a}$ are comparatively high, $\texttt{TC}_\texttt{b}$ is competitive with these methods for quasi-converging distributions, while it is much faster than these methods for the other tested distributions.

7.4.2 Experiments with Force-Approximation Methods

In the following, we will study the force-approximation methods `FrRe`, `BaHu`, `GrRo`, `Al`, $\texttt{NM}^2_\texttt{a}$, $\texttt{NM}^2_\texttt{b}$, and the naive exact force-calculation algorithm `Ex`. Therefore, we created uniform, non-uniform, and quasi-converging distributions of N particles, with N ranging from 1000 to 128000, like in Section 7.4.1. Then, we applied each force-approximation method on each class of distribution and each size of N and compared the running times and calculated forces with the running times and calculated forces of the naive exact algorithm `Ex`. Since it is reasonable to compare approximative algorithms at the same level of accuracy, we need the following definitions:

Definition 7.1 (Error and Accuracy of an Approximation). *Suppose, N charged particles are distributed in the plane,* `A` *is a force-approximation method, $F_{\texttt{A}}(i)$ denotes the approximation of the force that acts on particle i due to all other particles, and $F_{\texttt{Ex}}(i)$ denotes the exact force that acts on particle i due to all other particles. The* error *of the approximation of the forces acting in the system of the N particles that is generated by* `A` *is defined as:*

$$Error(\texttt{A}) = \sqrt{\frac{\sum_{i=1}^{N} \|F_{\texttt{Ex}}(i) - F_{\texttt{A}}(i)\|^2}{\sum_{i=1}^{N} \|F_{\texttt{Ex}}(i)\|^2}}$$

The approximation that is generated by `A` *is of* low, medium, *and* high *accuracy if $Error(\texttt{A}) < 10^{-2}$, $Error(\texttt{A}) < 10^{-3}$, and $Error(\texttt{A}) < 10^{-4}$), respectively.*

For each force-approximation method and class of distributions, we determined the parameters that guarantee low, medium, and high accuracy for all sizes of N. Note that the meaning of these parameters has been explained in Sections 1.2.2, 5.2, and 5.3. For `FrRe` we let the grid-coarsening parameter g take the values $2, 3, \ldots, 10, 12, \ldots, 20, 30, \ldots, 100, 120$. For `BaHu` we variated the tolerance parameter α in the range $0.01, 0.02, 0.03, 0.05, 0.07, 0.1, 0.15, 0.2, 0.3, \ldots, 1.0$. The precision parameter p of the multipole methods `GrRo`, `Al`, $\texttt{NM}_\texttt{a}^2$, and $\texttt{NM}_\texttt{b}^2$ was variated in the range $1, 2, \ldots, 8$. Then, we selected the set of parameters that resulted in the fastest running times for the fixed accuracies. These parameters are listed in Table 7.2.

Class of Distributions	Accuracy	Parameters of Algorithm:					
		`FrRe`	`BaHu`	`GrRo`	`Al`	$\texttt{NM}_\texttt{a}^2$	$\texttt{NM}_\texttt{b}^2$
uniform	low	$g = 120$	$\alpha = 0.7$	$p = 3$	$p = 3$	$p = 3$	$p = 3$
	medium	$g = 120$	$\alpha = 0.2$	$p = 4$	$p = 4$	$p = 4$	$p = 4$
	high	$g = 120$	$\alpha = 0.1$	$p = 6$	$p = 6$	$p = 6$	$p = 6$
non-uniform	low	$g = 90$	$\alpha = 0.7$	$p = 3$	$p = 3$	$p = 3$	$p = 3$
	medium	$g = 120$	$\alpha = 0.2$	$p = 4$	$p = 5$	$p = 5$	$p = 5$
	high	$g = 120$	$\alpha = 0.1$	$p = 6$	$p = 7$	$p = 7$	$p = 7$
quasi-converging	low	$g = 2$	$\alpha = 0.3$	$p = 1$	$p = 4$	$p = 4$	$p = 4$
	medium	$g = 2$	$\alpha = 0.1$	$p = 1$	$p = 6$	$p = 6$	$p = 6$
	high	$g = 2$	$\alpha = 0.03$	$p = 1$	$p = 8$	$p = 8$	$p = 8$

Table 7.2: Parameters of force-approximation methods that guarantee a desired accuracy for three classes of distributions.

We shortly comment on the values of these parameters. It is clear that for an increase of the desired accuracy g and p have to be increased, while α has to be reduced. The grid-coarsening parameter g that guarantees medium and high accuracy for all but the quasi-converging distribution is 120. This implies that the underlying grids are 1×1 or 2×2 grids. Therefore, the force calculation of `FrRe` is exact in these cases. For the quasi-converging distributions $g = 2$ is sufficient, which is not surprising, since these

distributions imply that nearly all particles are placed on the line connecting $[0, 0]$ and $[10^{-25}, 10^{-25}]$. Hence, nearly all particles are contained in the lowest leftmost grid box even for the smallest value of g, and the interactions between the nodes in this grid box are calculated exactly. Choosing $p = 1$ in the method `GrRo` for quasi-converging distributions is sufficient to guarantee all desired accuracies, since there exists a leaf in each complete truncated quadtree that contains nearly all particles. Since the interactions between these nodes are calculated exactly, the choice of p is marginal. The parameters of $\texttt{NM}^2_\texttt{a}$ and $\texttt{NM}^2_\texttt{b}$ are identical, since both methods differ only in the tree construction procedure.

Now we will concentrate on the running times of the force-approximation algorithms. For brevity, we only present the results for low and high desired accuracies. The CPU times for medium accuracies are in between. For each size of N, each class of distribution, and each algorithm, the reported running times are the average times of 100 tests.

For uniform distributions (see Table 7.3) and low and high accuracy all approximative algorithms except `FrRe` are much faster than the exact method. The slow running times of `FrRe` are caused by the fact that the calculation is exact by the choice of g. In comparison with `BaHu` the multipole methods scale much better if high accuracy is desired. The expected running time of `GrRo` and $\texttt{NM}^2_\texttt{b}$ is linear for these distributions. Therefore, it is not surprising that they are very fast. $\texttt{NM}^2_\texttt{b}$ is the fastest method for both accuracies. `Al` is roughly a factor 3 to 5 slower than $\texttt{NM}^2_\texttt{b}$. $\texttt{NM}^2_\texttt{a}$ is roughly a factor 2 slower than $\texttt{NM}^2_\texttt{b}$.

Particles	Accuracy	CPU Time in Seconds for Uniform Distributions						
		Approximative Methods						Exact
		`FrRe`	`BaHu`	`GrRo`	`Al`	$\texttt{NM}^2_\texttt{a}$	$\texttt{NM}^2_\texttt{b}$	Method
1000	Low	0.13	0.03	0.02	0.07	0.02	0.02	0.09
2000		0.57	0.08	0.08	0.16	0.06	0.05	0.36
4000		2.49	0.18	0.09	0.34	0.15	0.08	2.03
8000		10.70	0.42	0.34	0.70	0.39	0.21	8.59
16000		46.58	0.95	0.40	1.46	0.83	0.40	35.52
32000		190.44	2.07	1.42	2.93	1.90	0.93	142.58
64000		1023.44	4.57	1.73	6.01	3.96	1.79	572.90
128000		4128.44	10.16	5.86	12.28	9.00	4.07	2288.73
1000	High	0.13	0.35	0.04	0.13	0.03	0.02	0.09
2000		0.57	1.12	0.18	0.29	0.09	0.06	0.36
4000		2.49	3.23	0.20	0.60	0.18	0.11	2.03
8000		10.70	8.49	0.78	1.24	0.49	0.29	8.59
16000		46.58	21.30	0.85	2.57	0.96	0.50	35.52
32000		190.44	51.22	3.25	5.22	2.36	1.29	142.58
64000		1023.44	120.42	3.59	10.60	4.64	2.26	572.90
128000		4128.44	282.20	13.33	21.50	10.91	5.54	2288.73

Table 7.3: Comparison of the running times for the calculation of the repulsive forces and uniform distributions.

Now we concentrate on the non-uniform distributions (see Table 7.4). For FrRe and low desired accuracy the choice of $g = 90$ implies an underlying grid that does not result in an exact force calculation (a 3×3 grid) only for $N = 128000$. As expected, the running times of GrRo grow significantly for these distributions in comparison with the running times of GrRo for uniform distributions. The method BaHu is up to a factor 2 slower in comparison with the CPU times that are needed by BaHu for uniform distributions. Only the running times of Al, NM_a^2, and NM_b^2 keep nearly unchanged. Again, for both desired accuracies NM_b^2 is the fastest method.

Particles	Accuracy	CPU Time in Seconds for Non-Uniform Distributions						
		Approximative Methods						Exact
		FrRe	BaHu	GrRo	Al	NM_a^2	NM_b^2	Method
1000	Low	0.09	0.05	0.05	0.07	0.03	0.02	0.09
2000		0.45	0.12	0.18	0.16	0.07	0.05	0.39
4000		2.76	0.26	0.53	0.33	0.17	0.12	2.06
8000		11.06	0.58	1.95	0.68	0.36	0.22	8.57
16000		47.13	1.29	10.33	1.41	0.81	0.46	35.54
32000		164.81	2.72	25.07	2.81	1.65	0.91	142.37
64000		1005.44	5.97	109.34	5.78	3.67	1.92	572.12
128000		3300.34	12.71	230.60	11.87	7.92	3.80	2283.39
1000	High	0.09	0.44	0.08	0.15	0.04	0.03	0.09
2000		0.45	1.60	0.28	0.33	0.09	0.07	0.39
4000		2.76	5.31	0.64	0.70	0.20	0.14	2.06
8000		11.19	15.34	2.46	1.44	0.48	0.32	8.57
16000		47.24	38.97	11.01	2.98	0.97	0.61	35.54
32000		177.53	91.76	28.44	6.07	2.26	1.40	142.37
64000		1014.77	212.94	120.67	12.30	4.53	2.58	572.12
128000		4054.58	468.64	250.17	25.04	10.45	5.88	2283.39

Table 7.4: Comparison of the running times for the calculation of the repulsive forces and non-uniform distributions.

For quasi-converging distributions (see Table 7.5) the running times of FrRe and GrRo are nearly identical with that of Ex, since nearly all particles are contained in the lowest leftmost grid boxes and in only one leaf of the complete pseudo quadtrees, respectively (Recall that the forces that act between all particles that are contained in one grid box and one leaf of the truncated pseudo quadtree, respectively, are calculated exactly). NM_a^2 and NM_b^2 are the fastest methods for both accuracies and need nearly the same amounts of running time. The method Al is a factor 2 to 3 slower than NM_a^2 and NM_b^2. For BaHu and low accuracy, the running times in comparison with uniform and non-uniform distributions increase significantly, too. On the other hand, the running times of BaHu for high desired accuracies are faster than that of BaHu for uniform and non-uniform distributions and high desired accuracy. This indicates that the force approximation phase of FrRe that

succeeds the quadtree construction phase is faster for quasi-converging distributions and high desired accuracy than for the other tested distributions and high desired accuracy.

Particles	Accuracy	CPU Time in Seconds for Quasi-Converging Distributions						
		Approximative Methods						Exact Method
		FrRe	BaHu	GrRo	Al	$\texttt{NM}_\texttt{a}^2$	$\texttt{NM}_\texttt{b}^2$	
1000	Low	0.09	0.18	0.10	0.06	0.02	0.03	0.09
2000		0.41	0.40	0.43	0.13	0.04	0.07	0.39
4000		1.98	0.87	2.06	0.26	0.08	0.15	2.03
8000		8.72	1.79	8.83	0.53	0.18	0.27	8.53
16000		36.03	3.67	35.92	1.04	0.39	0.55	35.83
32000		143.70	7.34	144.59	2.10	0.77	0.97	142.14
64000		574.08	14.99	577.18	4.11	1.67	1.95	573.30
128000		2303.50	30.92	2315.93	8.44	3.61	3.67	2280.16
1000	High	0.09	0.30	0.10	0.09	0.02	0.04	0.09
2000		0.41	0.74	0.43	0.19	0.04	0.08	0.39
4000		1.98	1.67	2.06	0.39	0.10	0.16	2.03
8000		8.72	3.65	8.83	0.80	0.21	0.29	8.53
16000		36.03	7.89	35.92	1.60	0.45	0.60	35.83
32000		143.70	16.11	144.59	3.18	0.94	1.06	142.14
64000		574.08	35.62	577.18	6.32	2.03	2.13	573.30
128000		2303.50	79.57	2315.93	12.83	4.34	4.00	2280.16

Table 7.5: Comparison of the running times for the calculation of the repulsive forces and quasi-converging distributions.

We can summarize that the multipole methods Al, $\texttt{NM}_\texttt{a}^2$, and $\texttt{NM}_\texttt{b}^2$ are best suited for approximating repulsive forces in the plane if high accuracy is desired. For low desired accuracy one might also use BaHu, although it is up to a factor 9 slower than $\texttt{NM}_\texttt{b}^2$. Since the constant factor of the $O(N \log N)$ methods Al and $\texttt{NM}_\texttt{a}^2$ is quite large, $\texttt{NM}_\texttt{b}^2$ outperforms the other approximative algorithms for all desired accuracies and uniform and non-uniform distributions, while it is comparable with $\texttt{NM}_\texttt{a}^2$ for quasi-converging distributions. Both methods $\texttt{NM}_\texttt{a}^2$ and $\texttt{NM}_\texttt{b}^2$ are faster than Al for all kinds of tested distributions, all desired accuracies, and all tested sizes of N.

7.5 Experiments with the Postprocessing Step

In this section we want to examine if the postprocessing step can improve the quality of drawings regarding the criterion that the desired edge lengths of the user should be preserved.

The set of test graphs that we selected were the random graphs and grid graphs that we already used in the tests of the multilevel step. The choice of these test graphs was motivated by the observation that for the grid graphs (with uniform desired edge lengths)

straight-line drawings that preserve the desired edge lengths exist, while the existence of such drawings for the random graphs is not guaranteed. The desired edge length of each edge was set to the fixed value 100.

We created an algorithm A by deleting the postprocessing step from FM^3. Then, we let FM^3 and A draw the test graphs. Since a good drawing of a graph with uniform desired edge length should have a small edge-length ratio (see Section 1.1.4 for a definition of the edge-length ratio), and the average edge length in the drawing should be equal to the average of the desired edge lengths, we measured for each drawing the average edge length and the edge-length ratio.

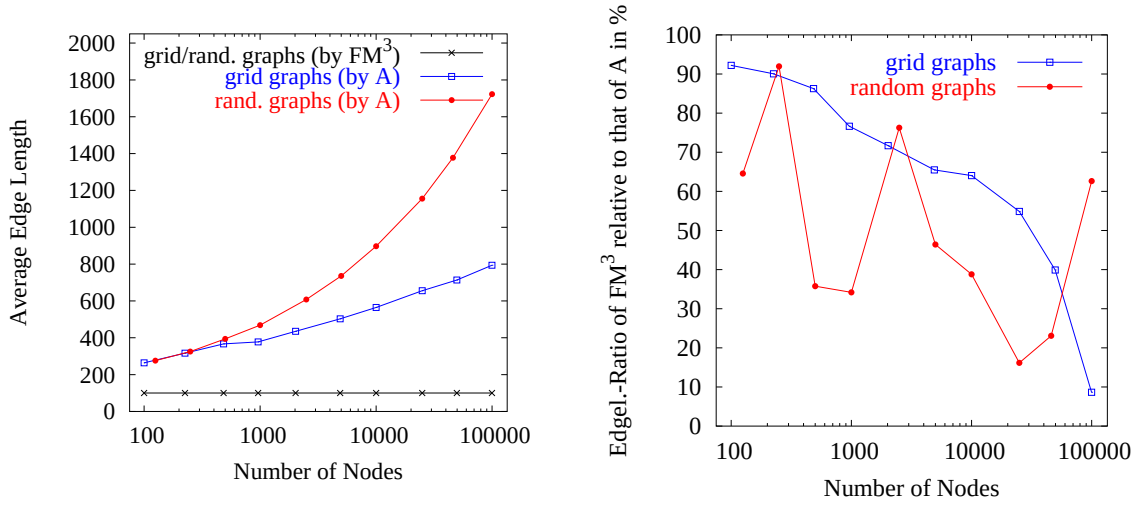


Figure 7.6: (left) Average edge length of the drawings of grid and random graphs generated by FM^3 and A. The desired edge length of each edge is 100. (right) Percentage of the edge-length ratio of FM^3 relative to the edge-length ratio of A for grid and random graphs.

The results are displayed in Figure 7.6. It can be seen in the left graphic of Figure 7.6 that the average edge length of the drawings generated with A grows monotonously with the size of the graphs and is up to 8 and 17.3 times larger than the average of the desired edge lengths for the grid graphs and random graphs, respectively. The average edge length of all drawings that are generated with FM^3 is equal to the average of the desired edge lengths. The right graphic of Figure 7.6 demonstrates that the edge-length ratio can be significantly reduced by FM^3 in comparison with A for both kinds of graphs. For the grid graphs it monotonously decreases from 92.2% for the smallest graph to 8.6% for the largest grid graph. Due to the structure of the random graphs, the reduction of the edge-length ratio is not as deterministic as for the grid graphs. But the edge-length ratio of FM^3 relative to that of A still ranges between 92.0% and 16.1%.

7.6 Experiments with FM^3

In this section we will examine the ability of FM^3 for drawing graphs. The natural criteria to evaluate a graph-drawing algorithm in practice are the needed running times and the

quality of the drawings. Unlike evaluating the first criterion, evaluating the quality of a drawing that is generated by a force-directed algorithm is a difficult task.

One could evaluate the quality of such a drawing by measuring the total energy in the underlying force-model and by comparing this value with the energy that is induced by the energy-minimum configuration of the nodes. However, this viewpoint has few practical use, since the energy-minimum configuration is unknown for many of the tested graphs.

One could also quantify the quality of the drawing, by measuring important aesthetic criteria (see Section 1.1.4) like the number of edge crossings, the sum of the edge lengths, the edge-length ratio, the used aspect-ratio area or how well the drawing displays the symmetries of the graphs (if such symmetries exist). Again, the optimal values of these measures are not known for every graph and the computation of the optimal values might be impossible for large graphs, since many of the corresponding optimization problems are $\mathcal{NP}$-hard (see Section 1.2.1). But even if we would know the optimal values of all these measures for each graph, the crucial requirement of a graph drawing is that an individual user is satisfied with the drawing of the graph. Therefore, it is a common and reasonable way to print the drawings and to comment how well they display the structure of each graph by keeping the previous mentioned aesthetic criteria in mind. The final evaluation of the quality of each drawing is left to the reader.

7.6.1 The Test Graphs and General Results

We have to determine the set of test graphs first. These are the real-world graphs that already were mentioned in the beginning of this chapter and several classes of artificial graphs. The artificial graphs were created to examine the scaling of `FM`^{3} on graphs with predefined structures but different sizes. In particular, we generated for each class of artificial graphs between 7 and 11 instances that range from approximate 100 nodes to approximate 100000 nodes. We generated the following classes of artificial graphs:

The *random grid* graphs were obtained by first creating regular square grid graphs and then randomly deleting 3% of the nodes. The *sierpinski* graphs were created by associating the *Sierpinski Triangles* [2] with graphs.

Furthermore, we tested the graph classes rand_1 and rand_2 that already were introduced in Section 7.2. They contain disconnected graphs, each consisting of many biconnected components.

The next two classes of artificial graphs were designed to test how well `FM`^{3} (and other force-directed algorithms) can handle highly non-uniform distributions of the nodes. Therefore we created these graphs in a way so that one can expect that an energy-minimal configuration of the nodes in a `Spring Embedder`-like force-model induces a tiny subregion of the drawing area which contains $\Theta(|V|)$ nodes. In particular, we constructed trees that contain a root node r with $|V|/4$ neighbors. The other nodes were subdivided into six

[2] A Sierpinski Triangle is a fractal that is generated by subdividing an equilateral triangle into four congruent smaller triangles. This subdivision process is recursively applied on the three outer triangles until a specified recursive depth is reached.

subtrees of equal size rooted at r. We call these graphs *snowflake* graphs. Additionally we created *spider* graphs by constructing a circle C containing 25% of the nodes. Each node of C is also adjacent to 12 other nodes of the circle. The other nodes were distributed on 8 paths of equal length that were rooted at one node of C. One important difference to the snowflake graphs is that the spider graphs have bounded maximum degree.

The last kind of artificial graphs are graphs with a relatively high edge density (in particular $|E|/|V| \geq 14$). We call them *flower* graphs. They were constructed by joining 6 circles of equal length at a single node before replacing each of the nodes by a complete Kuratowski subgraph with 30 nodes (K_{30}).

We partitioned the artificial and real-world graphs into two sets. The first set consists of graphs that are connected, consist of few biconnected components, have a constant maximum node degree, a low edge density, and one can expect that an energy-minimal configuration of the nodes in an associated `Spring Embedder`-like force-model of such a graph does not contain $\Theta(|V|)$ nodes in an extremely tiny subregion of the drawing area. Since the graphs contained in this set do not cause problems for many force-directed graph-drawing algorithms, we call the set of these graphs *kind*. The second set is exactly the complement of the first one, and we call the set of these graphs *challenging*. For example, the random grid graphs and sierpinski graphs are kind artificial graphs, while all other artificial graphs are challenging artificial graphs. The real-world graphs were partitioned into these two sets as well.

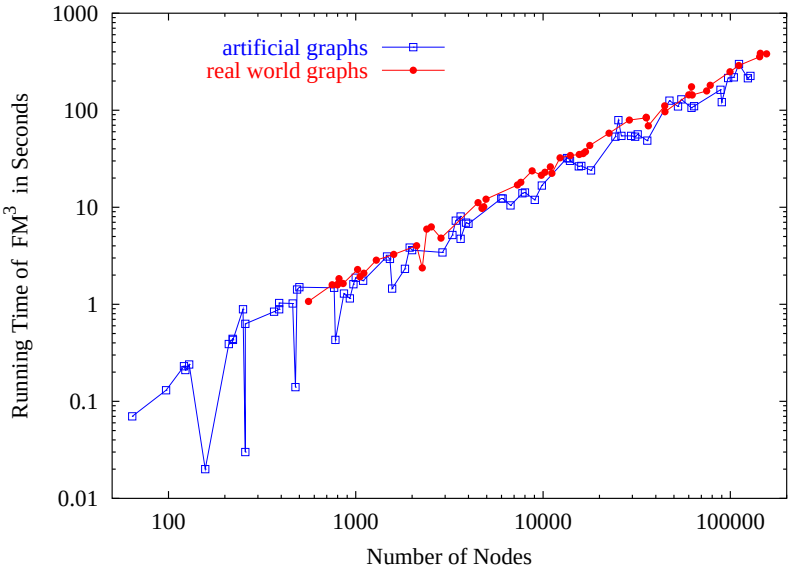


Figure 7.7: The running times of `FM`^{3} for drawing all artificial graphs and all real-world graphs.

Before going into details, we want to give an overview about the total running times of `FM`^{3} for all artificial graphs and all real-world graphs independent of their structure. The results are shown in Figure 7.7. Except for some graphs that contain less than 1000 nodes, the running times of `FM`^{3} for drawing graphs containing a similar number of nodes

are similar. The only graphs that induce a significantly faster running time of FM^3 are the smaller rnd_2 graphs that contain a fixed number of 100 components. Since for these graphs the number of nodes of each component is very small (much smaller than the variable *crossover_point* = 175 defined in Section 5.3.5) the force calculation in `Grid_Embedder` is done by a direct calculation instead of a multipole approximation (see Section 5.3.5). Therefore much CPU time is saved as long as each component contains significantly less than *crossover_point* nodes. For the larger rnd_2 graphs the approximative calculation of the repulsive forces is used in NM^2. Therefore, the running times are similar to those of the other tested graphs.

Furthermore, it can be observed that all tested graphs containing less than 1000 nodes can be drawn in less than 1.9 seconds. The graphs containing less than 10000 and 100000 nodes can be drawn in less than 23.8 and 262.3 seconds, respectively. The largest graph containing 156317 nodes and 1059331 edges can be drawn in 381.4 seconds.

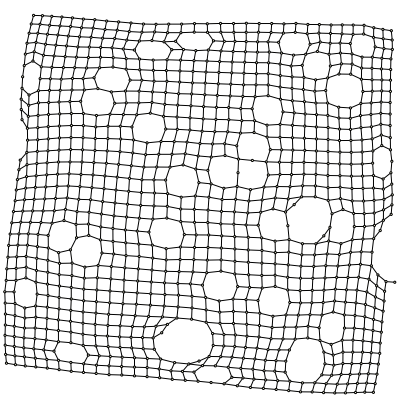
7.6.2 Drawing the Kind Graphs

Table 7.6 displays details of the structure of selected kind artificial and kind real-world graphs and the running times that are needed by FM^3 to display these graphs. All graphs are connected, have a low edge density, a bounded maximum degree, and the number of biconnected components (in comparison with the number of nodes) is small.

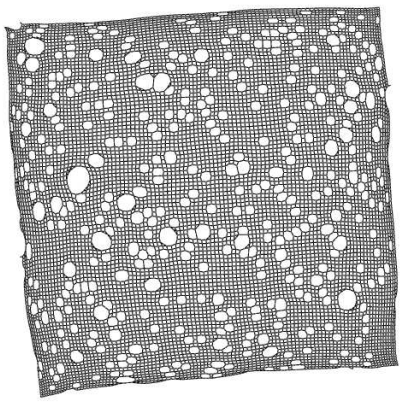
Graph Information								CPU Times of FM^3 in Seconds
Type	Name	$\|V\|$	$\|E\|$	$\|C\|$	$\|B\|$	$\frac{\|E\|}{\|V\|}$	max. degree	
Kind Artificial	rnd_grid_032	985	1834	1	2	1.86	4	1.9
	rnd_grid_100	9497	17849	1	6	1.88	4	19.1
	rnd_grid_320	97359	184532	1	2	1.90	4	215.4
	sierpinski_06	1095	2187	1	1	2.00	4	1.8
	sierpinski_08	9843	19683	1	1	2.00	4	16.8
	sierpinski_10	88575	177147	1	1	2.00	4	162.0
Kind Real World	crack	10240	30380	1	1	2.97	9	23.0
	fe_pwt	36463	144794	1	55	3.97	15	69.0
	finan_512	74752	261120	1	1	3.49	54	158.2
	fe_ocean	143437	409593	1	39	2.86	6	355.9

Table 7.6: Selected kind artificial and kind real-world graphs and the running times that are needed by FM^3 to draw them. C denotes the set of components, while B denotes the set of biconnected components of the graphs.

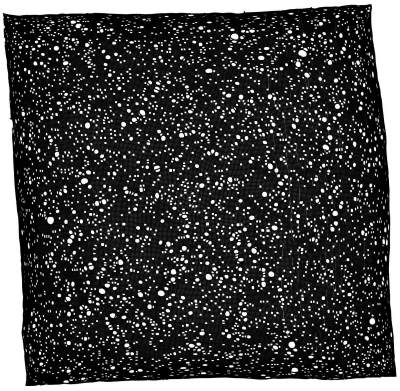
The drawings of the random grid graphs and sierpinski graphs are displayed in Figure 7.8. The drawings of selected kind real-world graphs are shown in Figure 7.9. All drawings clearly visualize the regular (respectively symmetric) structures of the graphs.



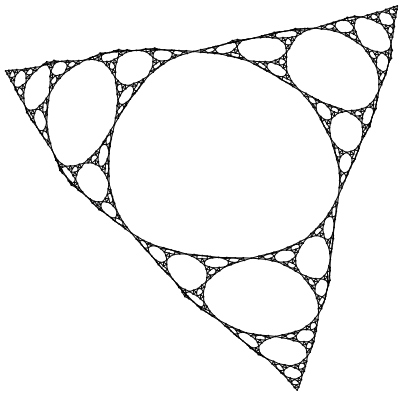
(a) rnd_grid_032



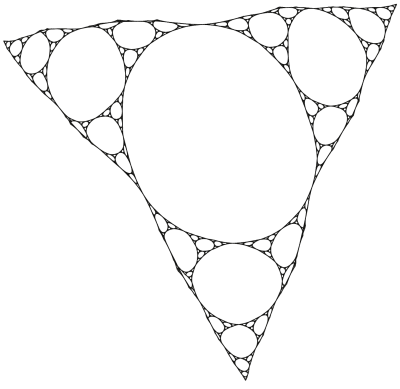
(b) rnd_grid_100



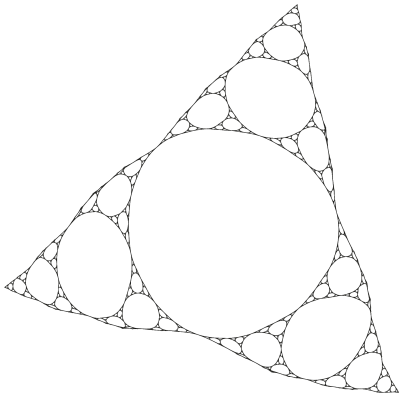
(c) rnd_grid_320



(d) sierpinski_06

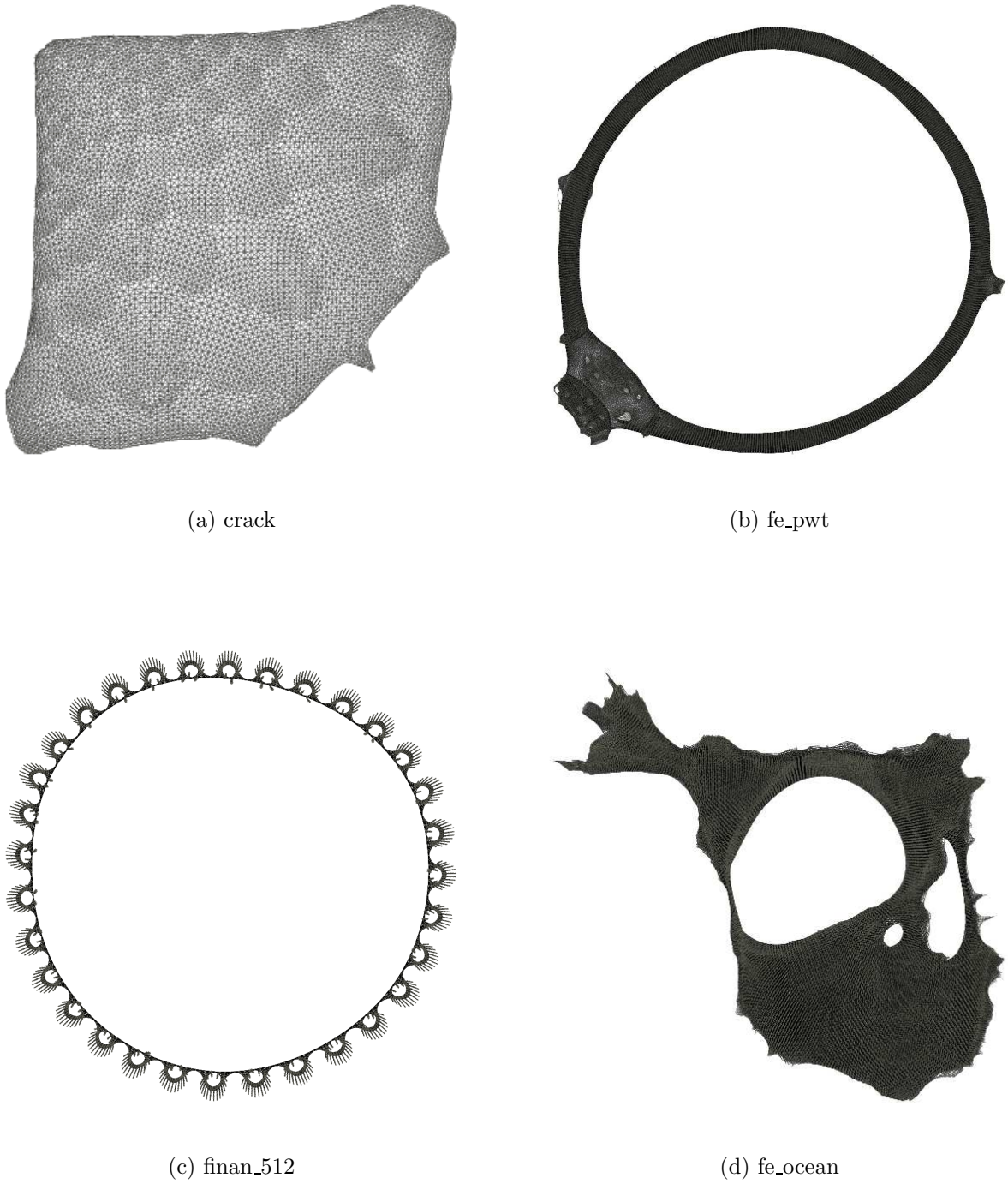


(e) sierpinski_08



(f) sierpinski_10

Figure 7.8: Drawings of random grid graphs and sierpinski graphs that are generated by FM^3.



(a) crack

(b) fe_pwt

(c) finan_512

(d) fe_ocean

Figure 7.9: Drawings of crack, fe_pwt, finan_512, and fe_ocean that are generated by FM^3.

7.6.3 Drawing the Challenging Graphs

Detailed information about selected challenging graphs and the running times of FM^3 are given in Table 7.7. The selected challenging disconnected real-world graphs are fe_body and dg_3691. The graphs ug_380 and dg_1087 each contain a node with a very high degree. Like dg_3691 and add_32 (that describes a 32 bit adder), dg_1087 consists of many biconnected components. The graphs bcsstk_31_con, bcsstk_32, and bcsstk_33 have a high edge density. Like for the kind graphs, for several challenging graph classes (e.g, the snowflake graphs and the flower graphs) a linear scaling of FM^3 can be observed.

Graph Information								CPU Times of FM^3 in Seconds
Type	Name	$\|V\|$	$\|E\|$	$\|C\|$	$\|B\|$	$\frac{\|E\|}{\|V\|}$	max. degree	
Challenging Artificial	rnd_1_A	1324	1527	10	692	1.15	31	2.9
	rnd_1_B	8897	10269	100	4766	1.15	34	17.8
	rnd_1_C	101492	117312	1000	54366	1.16	41	213.4
	rnd_2_A	953	1039	100	563	1.09	6	0.7
	rnd_2_B	9843	11337	100	5362	1.15	38	19.7
	rnd_2_C	98173	113990	100	51881	1.16	327	262.3
	snowflake_A	971	970	1	970	1.00	256	1.6
	snowflake_B	9701	9700	1	9700	1.00	2506	17.4
	snowflake_C	97001	97000	1	97000	1.00	25006	166.5
	spider_A	1000	2200	1	801	2.20	18	1.9
	spider_B	10000	22000	1	8001	2.20	18	17.7
	spider_C	100000	220000	1	80001	2.20	18	177.2
	flower_A	930	13521	1	1	14.54	30	1.2
	flower_B	9030	131241	1	1	14.53	30	11.9
	flower_C	90030	1308441	1	1	14.53	30	121.4
Challenging Real World	ug_380	1104	3231	1	27	2.93	856	2.1
	dg_3691	2266	3277	829	1221	1.45	119	2.4
	add_32	4960	9462	1	951	1.91	31	12.1
	dg_1087	7602	7601	1	7601	1.00	6566	18.1
	bcsstk_33	8738	291583	1	1	33.37	140	23.8
	bcsstk_31_con	35586	572913	1	48	16.10	188	83.6
	bcsstk_32	44609	985046	1	3	22.08	215	110.9
	fe_body	44775	163734	39	90	3.66	28	96.5

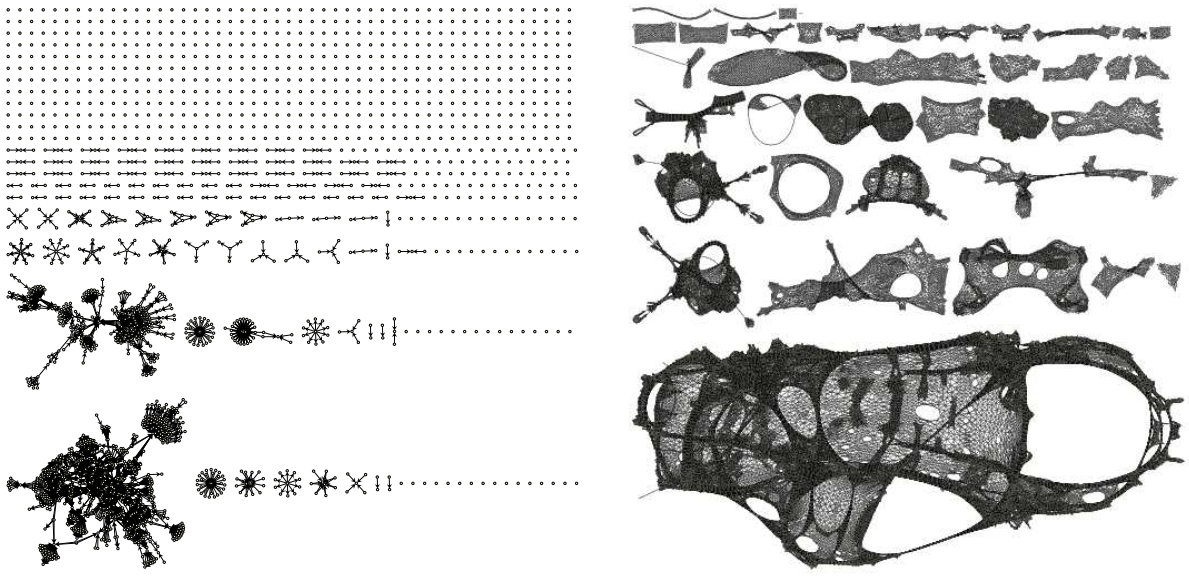
Table 7.7: Selected challenging artificial and challenging real-world graphs and the running times that are needed by FM^3 to draw them. C denotes the set of components, while B denotes the set of biconnected components of the graphs.

The drawings of the random disconnected graphs (see Figure 7.11) are quite compact, but the drawing of the smallest $rand_1$ graph (see Figure 7.11(a)) could be improved re-

garding the used drawing area. Although the components are random graphs, some nice sub-structures of these graphs are visualized by FM^3.

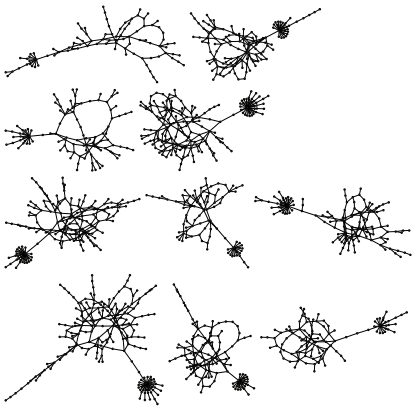
The drawings of the snowflake graphs, spider graphs, and flower graphs (see Figures 7.12 and 7.13) illustrate the symmetric (sub)-structures of these graphs. Furthermore, local details of these graphs are drawn nicely. As expected, the center sub-regions of the snowflake graphs and spider graphs that contain $\Theta(|V|)$ nodes are very small by construction of the graphs. For the larger snowflake graphs and spider graphs, these regions can be visualized only by zooming into the center of the drawings. In combination with the running times shown in Table 7.7 the drawings of the snowflake and spider graphs confirm the theoretical result that the asymptotic running time of FM^3 is not influenced by the distribution of the nodes during the computation.

We concentrate on the selected challenging real-world graphs next. Like the drawings of the random disconnected graphs, the drawings of dg_3691 and fe_body (displayed in Figure 7.10) are very compact, and one can recognize that fe_body is a representation of parts of a car (with the car body turned upside down). Figures 7.14(a) and 7.14(b) show the nodes with high degree of ug_380 and dg_1087, respectively. Figure 7.14(b) illustrates that dg_1087 is a tree with small diameter. Due to its many biconnected components the drawing of add_32 in Figure 7.14(c) has a tree-like structure. The drawings of the dense graphs bcsstk_31_con, bcsstk_32, and bcsstk_33 in Figure 7.14(d-f) display the regular (sub)-structures of the graphs. One can imagine that bcsstk_31_con is a car body in a top view.

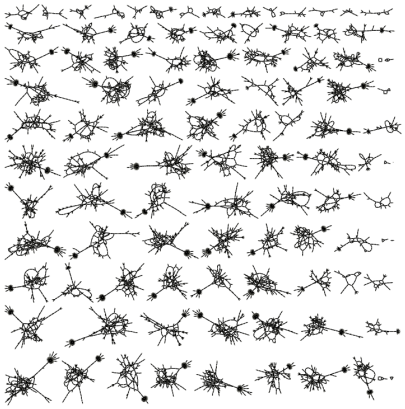


(a) dg_3691 (b) fe_body

Figure 7.10: Drawings of dg_3691 and fe_body that are generated by FM^3.

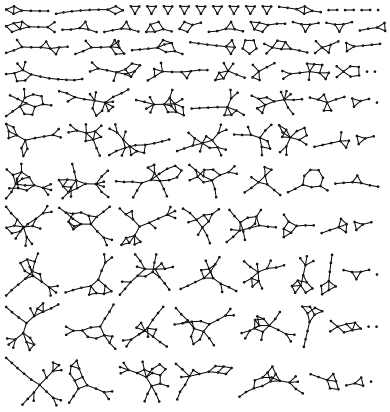


(a) rnd_1_A

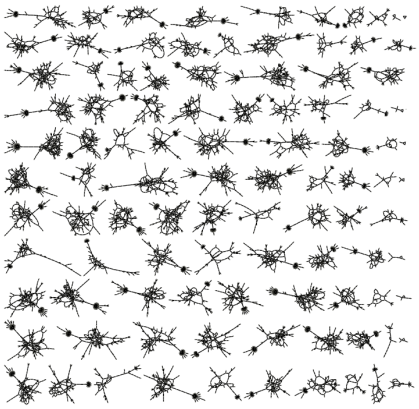


(b) rnd_1_B

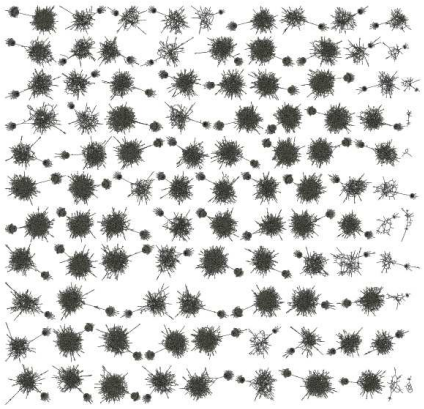
(c) rnd_1_C



(d) rnd_2_A

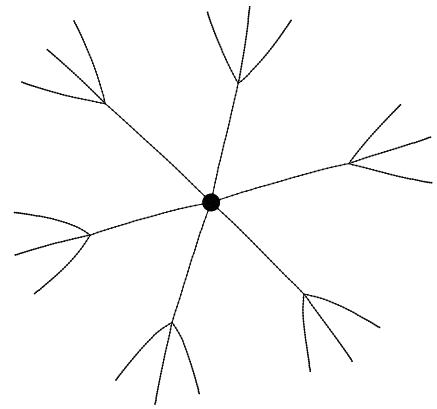


(e) rnd_2_B

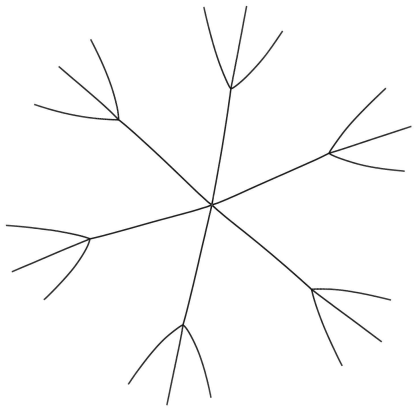


(f) rnd_2_C

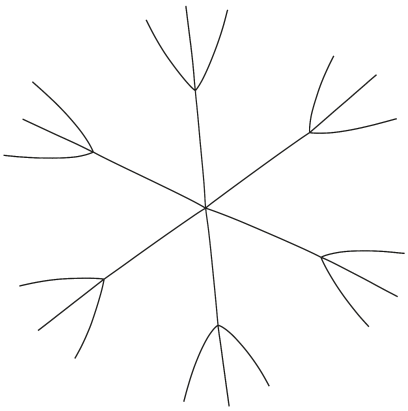
Figure 7.11: Drawings of the rand_1 graphs and rand_2 graphs that are generated by FM^3.



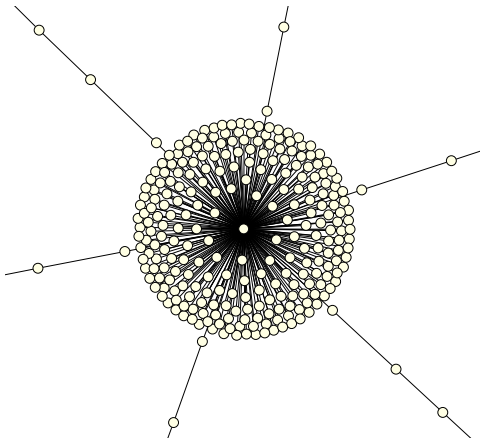
(a) snowflake_A



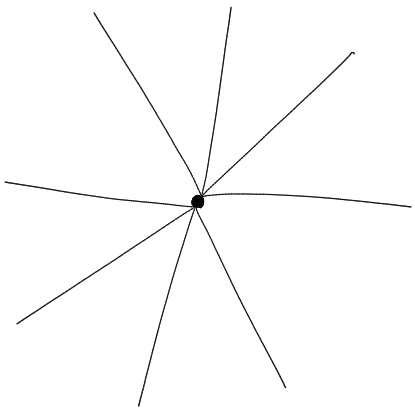
(b) snowflake_B



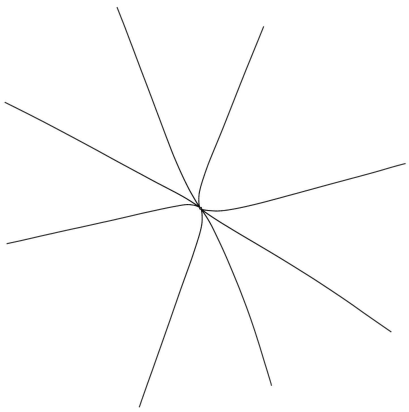
(c) snowflake_C



(d) Detail of snowflake_A

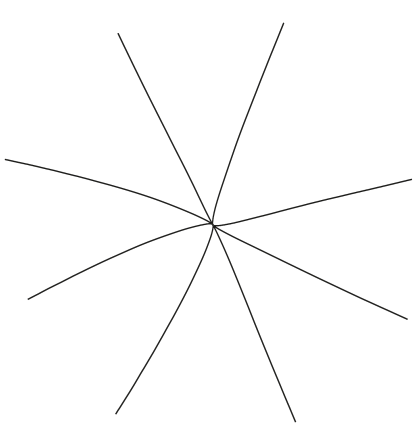


(e) spider_A

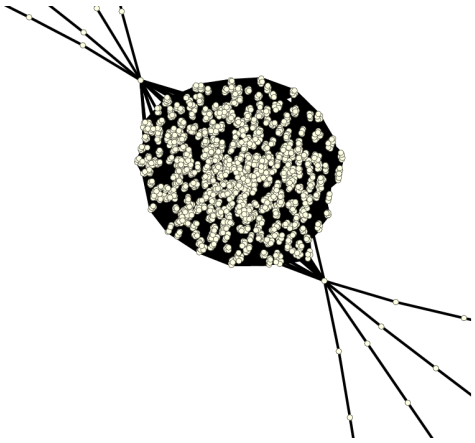


(f) spider_B

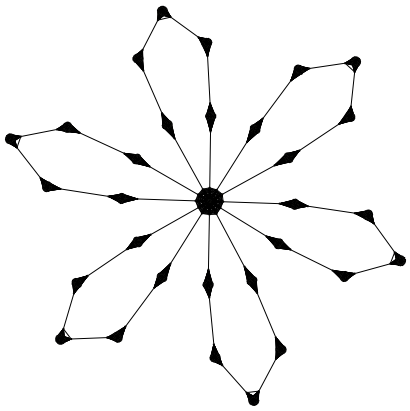
Figure 7.12: Drawings of the snowflake graphs, a detail of snowflake_A, the spider_A graph, and the spider_B graph generated by FM^3.



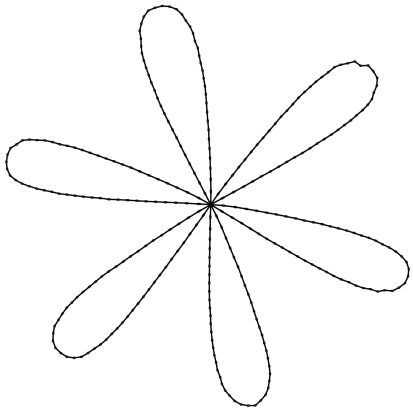
(a) spider_C



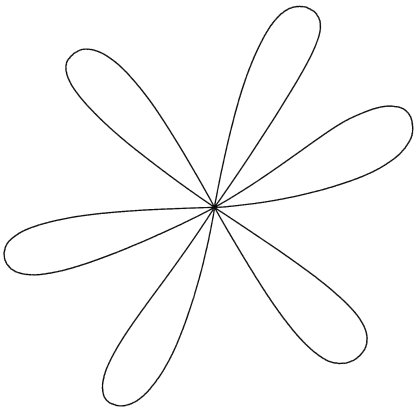
(b) Detail of spider_B



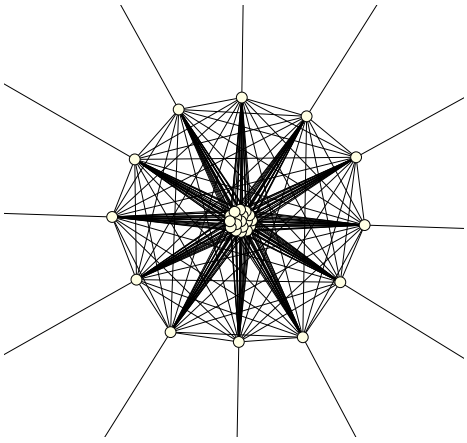
(c) flower_A



(d) flower_B

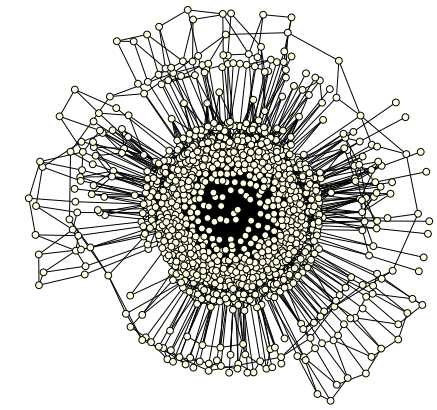


(e) flower_C

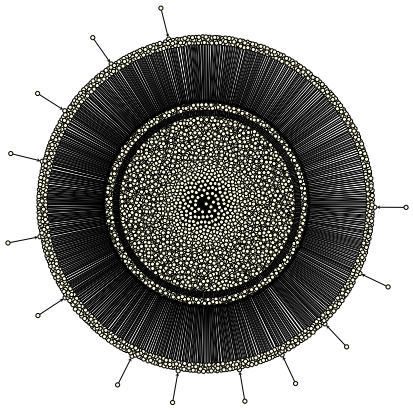


(f) Detail of flower_A

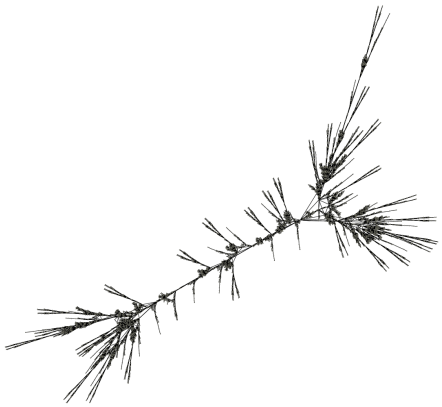
Figure 7.13: Drawings of the spider_C graph, a detail of spider_B, the flower graphs, and a detail of flower_A generated by FM^3.



(a) ug_380



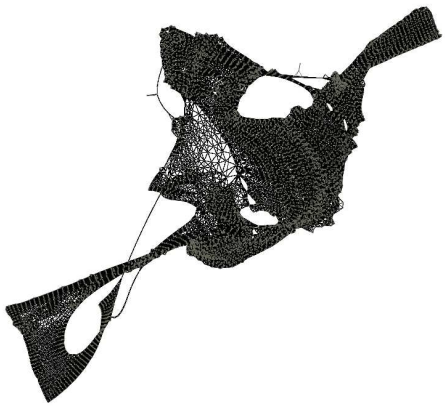
(b) dg_1087



(c) add_32

(d) bcsstk_33

(e) bcsstk_31_con



(f) bcsstk_32

Figure 7.14: Drawings of ug_380, dg_1087, add_32, bcsstk_33, bcsstk_31_con, and bcsstk_32 that are generated by FM^3.

7.7 Experimental Comparison of FM^3 with Other Algorithms

In this section we will test some of the fastest force-directed and algebraic graph-drawing methods on the selected kind and challenging graphs listed in Table 7.6 and Table 7.7 and compare the results with those of FM^3.

The first tested algorithm is an implementation of the Grid-Variant Algorithm of Fruchterman and Reingold [47] (see Section 1.2.2) that was implemented in the framework of AGD [75] by Stefan Näher and David Alberts. We chose this method, since it is one of the faster classical force-directed methods and denote it by GVA for brevity.

Furthermore, we tested two force-directed multilevel methods that were introduced in Section 1.3.2: An implementation of GRIP [50, 49] by Roman Yusufov that is available from [140] and the Fast Multi-scale Method (FMS) [64] that was implemented by Yehuda Koren and is available from [88].

Finally, we compared the force-directed methods with two fast algebraic graph-drawing methods that have been sketched in Section 1.3.3: The method ACE [86, 87] and the high-dimensional embedding method [66] that we denote by HDE here. Like FMS, both methods were implemented by Yehuda Koren and can be obtained from [88].

We ran all algorithms with the given sets of standard settings, measured the running times, and printed the generated drawings. Unfortunately, it was not possible to draw each graph with each algorithm: The implementations of FMS, ACE, and HDE are designed for connected graphs, only. The memory requirements of FMS are quadratic in the size of the graphs. Therefore, the implementation of FMS restricts to graphs that contain at most 10000 nodes. It was not possible to draw any graph containing more than 30000 nodes and some of the disconnected graphs with GRIP because of an error in the implementation. Furthermore, since in some cases the running times were extremely high, we decided to stop each computation if the CPU time exceeded 10 hours.

7.7.1 Drawing the Kind Graphs

Table 7.8 compares the running times of the methods GVA, GRIP, FMS, ACE, HDE, and FM^3 for the selected kind graphs that have been described in Table 7.6.

GVA is 6 to 8 times slower than FM^3 for the smaller graphs containing less than 1000 nodes. This factor grows with the sizes of the graphs. The largest graph fe_ocean is drawn by GVA in 5 hours and 20 minutes. This is a factor 54 slower than FM^3. GRIP is a factor 3 to 6 faster than FM^3 on the smaller and medium graphs (containing roughly 10000 nodes), but the implementation does not work on the really large instances. FMS is a factor 2 faster than FM^3 on the smallest random grid and sierpinski graphs but a factor 2 slower than FM^3 on the medium sized graphs. The real-world graphs cannot be drawn by FMS, since they contain more than 10000 nodes. The algorithm ACE is much faster than the force-directed algorithms on nearly all graphs. Only fe_pwt cannot be drawn in the time limit of 10 hours but we are not aware of a theoretical reason for this long running time.

Graph Information		CPU Times in Seconds					
Type	Name	Force-Directed Methods				Algebraic Methods	
		FM³	GVA	GRIP	FMS	ACE	HDE
Kind Artificial	rnd_grid_032	1.9	12.5	0.3	1.0	< 0.1	< 0.1
	rnd_grid_100	19.1	203.4	4.4	32.0	0.5	0.1
	rnd_grid_320	215.4	6316.1	(E)	(M)	4.1	1.3
	sierpinski_06	1.8	13.1	0.3	1.0	< 0.1	< 0.1
	sierpinski_08	16.8	171.7	4.8	33.0	1.0	0.1
	sierpinski_10	162.0	3606.4	(E)	(M)	23.4	1.0
Kind Real World	crack	23.0	317.5	6.8	(M)	0.4	0.2
	fe_pwt	69.0	1869.1	(E)	(M)	(T)	0.5
	finan_512	158.2	6319.8	(E)	(M)	7.5	1.0
	fe_ocean	355.9	19247.0	(E)	(M)	4.0	3.4

Table 7.8: Comparison of the CPU times of some of the fastest force-directed and algebraic graph-drawing algorithms on kind graphs. Explanations: (C) No drawing was generated, because the algorithm is designed for connected graphs. (E) No drawing was generated because of an error in the executable. (M) No drawing was generated because the memory is restricted to graphs with ≤ 10.000 nodes. (T) No drawing was generated within 10 hours of CPU time.

We suppose this might be caused by an error in the implementation. HDE is by far the fastest algorithm on the kind graphs. It needs less than 3.4 seconds for drawing even the largest tested graph. This is significantly faster than any tested force-directed algorithm.

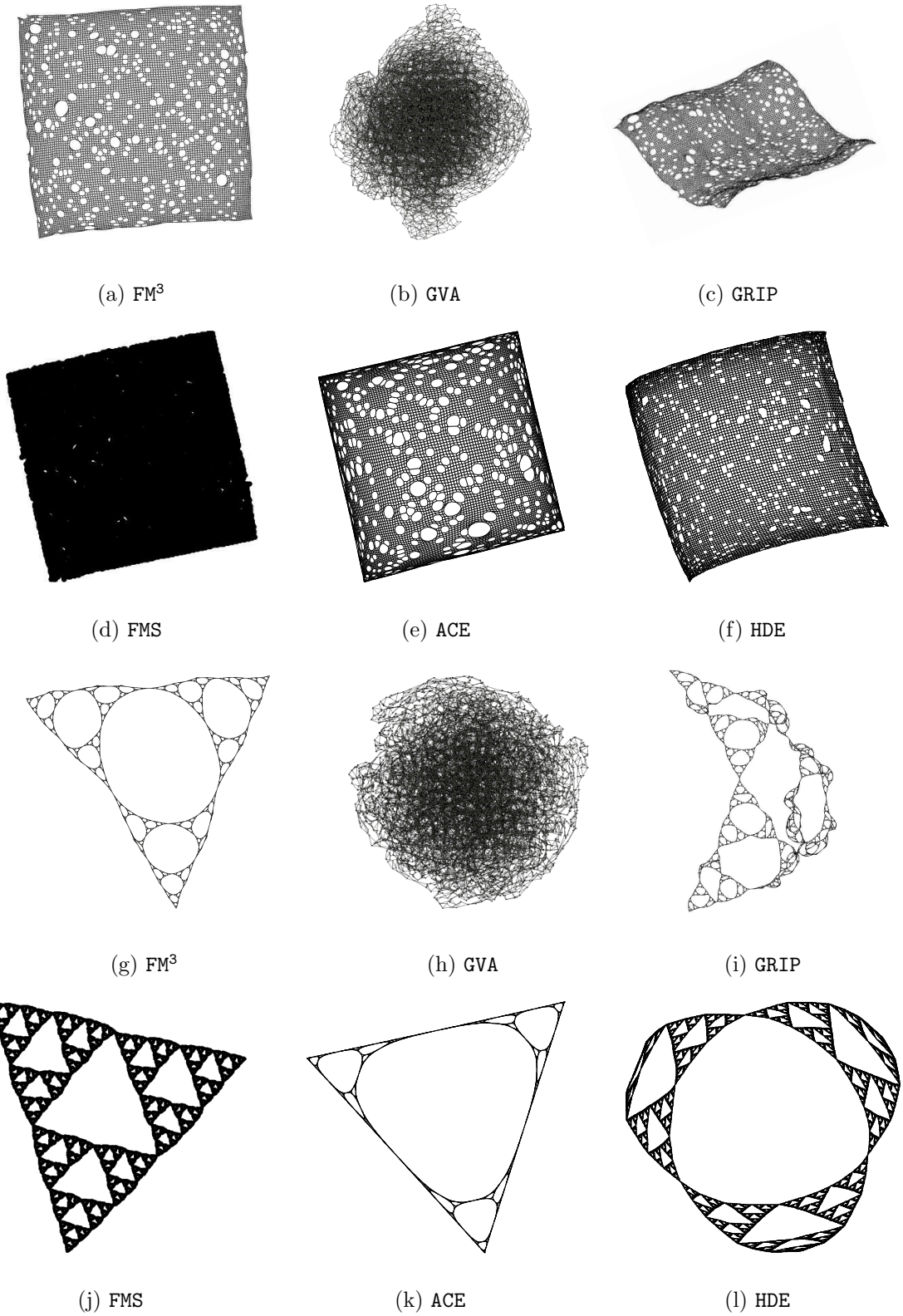
However, at least as important as the running times is the quality of the created drawings. Consequently, we will present the drawings generated by the methods GVA, GRIP, FMS, ACE, HDE, and FM³ for some of the tested graphs next.

Nearly all drawings of the grid_rnd_100 graph (see Figure 7.15(a)-(f)) and the sierpinski_08 graph (see Figure 7.15(g)-(l)) clearly display the regular (respectively symmetric) structures of the graphs: The drawing of the sierpinski_08 graph generated by GRIP (see Figure 7.15(i)) could be improved since many edge crossing appear. GVA does not untangle the drawings that were induced by the random initial placement for both graphs.

The drawings of the other tested random grid and sierpinski graphs listed in Table 7.8 that are generated by GRIP, FMS, ACE, HDE, and FM³ also display the structures of the graphs nicely, while the drawings of GVA are comparable with the ones presented in Figure 7.15(b) and (h). In the following, we only comment on those drawings of a graph which visualize parts of its structure in an acceptable way for brevity.

The drawings of the crack graph that are generated by ACE and HDE (see Figure 7.16(d)-(e)) are comparable with that of FM³ (see Figure 7.16(a)). Again the drawing of GRIP (see Figure 7.16(c)) could be improved since many edge crossings appear.

The structure of the other kind real-world graphs are clearly visualized by the algorithms ACE, HDE, and FM³ (see Figures 7.16(f)-(l) and Figure 7.17) as well.



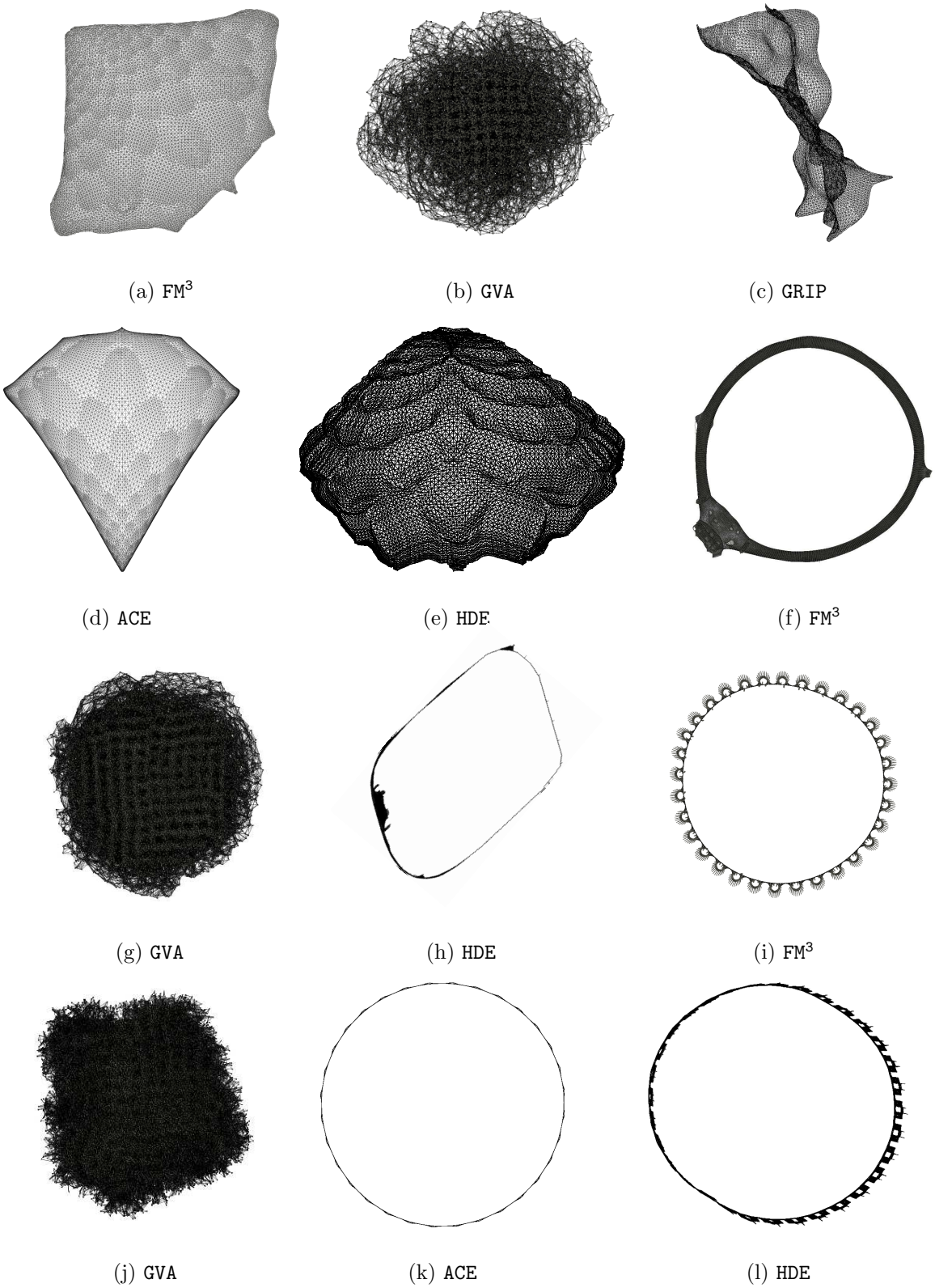
(a) FM^3 (b) GVA (c) GRIP

(d) FMS (e) ACE (f) HDE

(g) FM^3 (h) GVA (i) GRIP

(j) FMS (k) ACE (l) HDE

Figure 7.15: (a)-(f) Drawings of the rnd_grid_100 graph and (g)-(l) the sierpinski_08 graph generated by different algorithms.



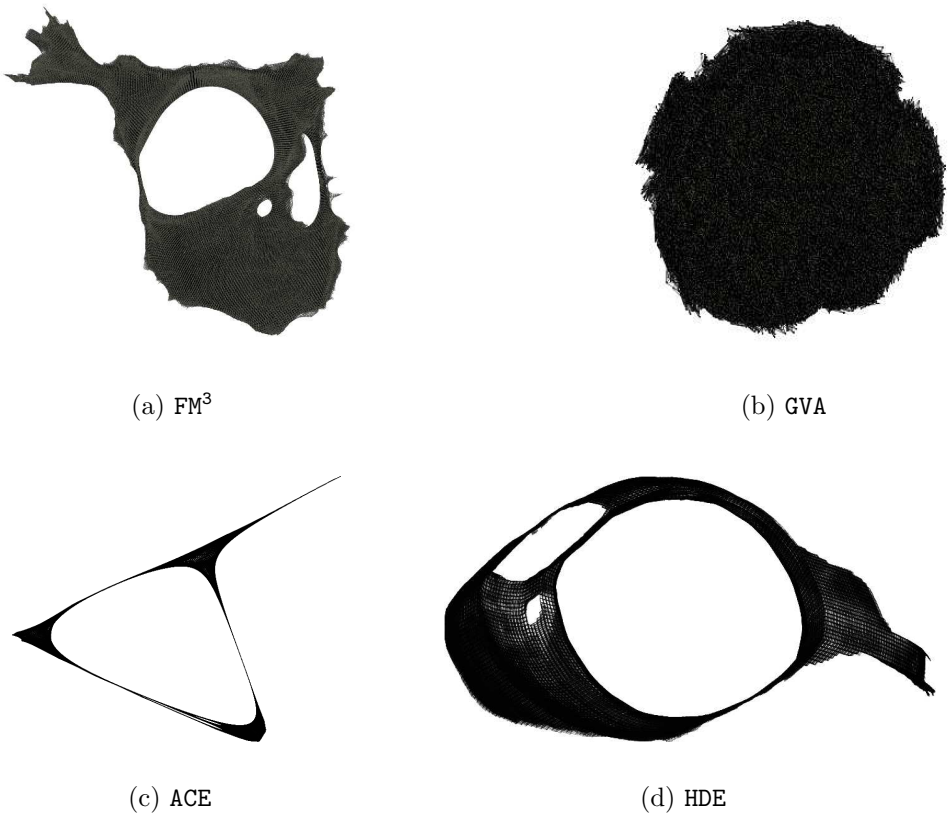
(a) FM^3 (b) GVA (c) GRIP

(d) ACE (e) HDE (f) FM^3

(g) GVA (h) HDE (i) FM^3

(j) GVA (k) ACE (l) HDE

Figure 7.16: (a)-(e) Drawings of the crack graph, (f)-(h) fe_pwt, and (i)-(l) finan_512 generated by different algorithms.



(a) FM^3 (b) GVA

(c) ACE (d) HDE

Figure 7.17: Drawings of fe_ocean generated by different algorithms.

7.7.2 Drawing the Challenging Graphs

The running times that are needed for drawing the selected challenging graphs are displayed in Table 7.9, and details about the structures of the graphs can be found in Table 7.7.

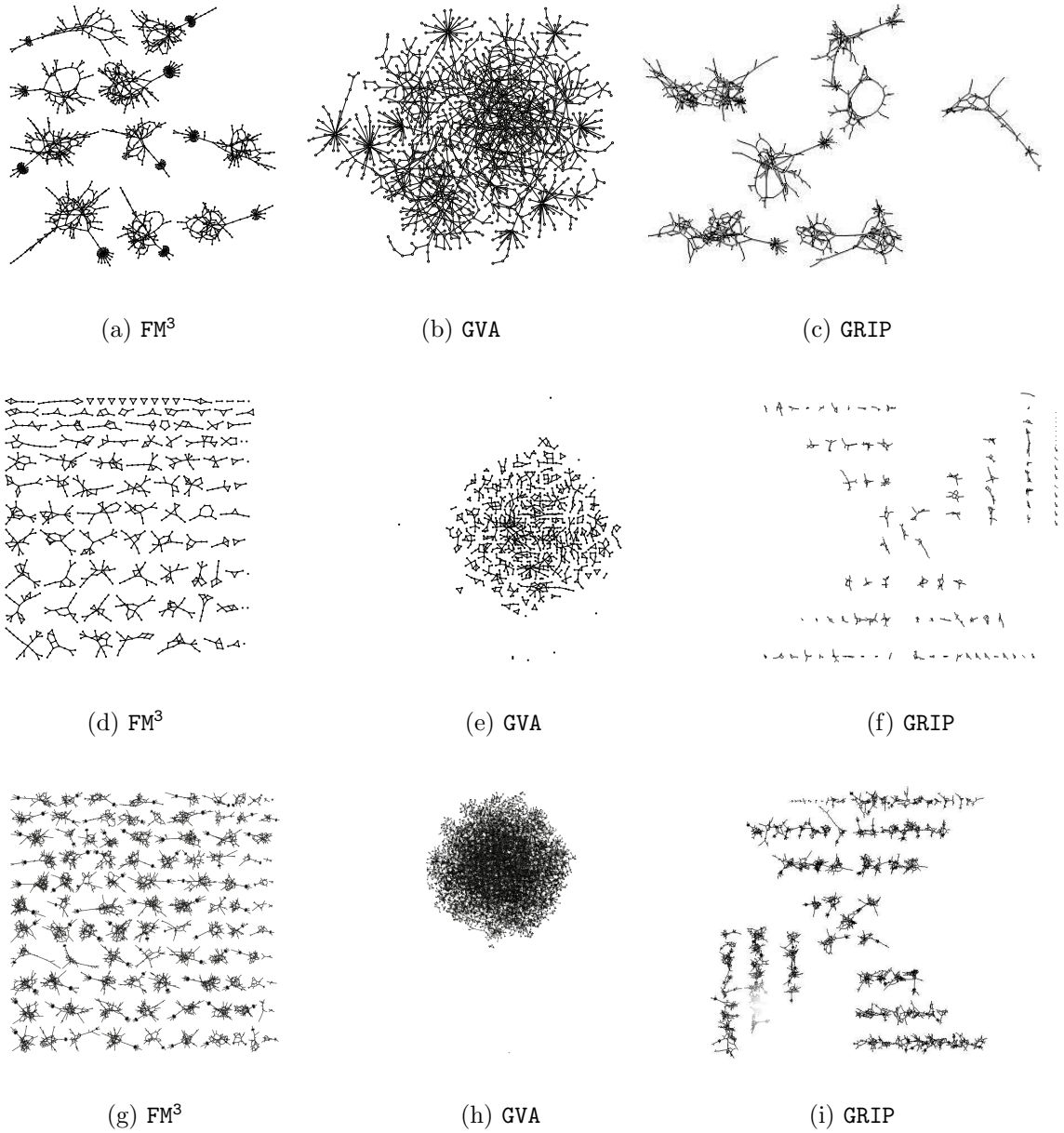
For nearly all graphs GVA is the slowest method and up to a factor 98 slower than FM^3. GVA needs more than 4 hours and 5 minutes for drawing the snowflake_C graph. Except for the graphs bcsstk_33 and flower_B, GRIP is faster (a factor 2 to 7) than FM^3. But GRIP cannot draw the largest graphs and many disconnected graphs. In comparison with the running times that are needed for drawing the kind graphs, the CPU time of FMS increases drastically for the flower graphs, dg_1087, and bcsstk_33. These graphs either contain nodes with a very high degree or have a high edge density. For many challenging graphs ACE is faster than the force-directed methods and needs less than 4 seconds to draw them. These running times grow extremely, when ACE is used to draw the snowflake graphs, the spider graphs, or dg_1087, which all contain a node with a very high degree. The running times of the algebraic method HDE are outstanding. Every tested connected challenging graph can be drawn in less than 2 seconds.

Graph Information		CPU Times in Seconds					
Type	Name	Force-Directed Methods				Algebraic Methods	
		FM^3	GVA	GRIP	FMS	ACE	HDE
Challenging Artificial	rnd_1_A	2.9	12.4	0.7	(C)	(C)	(C)
	rnd_1_B	17.8	119.3	(E)	(C)	(C)	(C)
	rnd_1_C	213.4	3255.9	(E)	(C)	(C)	(C)
	rnd_2_A	0.7	7.3	0.2	(C)	(C)	(C)
	rnd_2_B	19.7	138.5	6.9	(C)	(C)	(C)
	rnd_2_C	262.3	5930.1	(E)	(C)	(C)	(C)
	snowflake_A	1.6	8.0	0.4	73.0	0.4	< 0.1
	snowflake_B	17.4	143.2	6.1	3320.0	(T)	< 0.1
	snowflake_C	166.5	14685.7	(E)	(M)	(T)	0.8
	spider_A	1.9	17.6	0.4	1.0	1.1	< 0.1
	spider_B	17.7	189.0	7.2	47.0	8.9	0.1
	spider_C	177.2	4568.3	(E)	(M)	280.7	1.3
	flower_A	1.2	61.7	0.7	1.0	< 0.1	< 0.1
	flower_B	11.9	595.1	19.3	46.0	1.4	0.2
	flower_C	121.4	11841.5	(E)	(M)	(T)	1.4
Challenging Real World	ug_380	2.1	23.1	0.4	1.0	< 0.1	< 0.1
	dg_3691	2.4	21.5	(E)	(C)	(C)	(C)
	add_32	12.1	80.6	1.6	17.0	0.5	< 0.1
	dg_1087	18.1	624.8	3.6	5402.0	108.4	< 0.1
	bcsstk_33	23.8	1494.6	29.1	6636.0	0.4	0.3
	bcsstk_31_con	83.6	4338.4	(E)	(M)	1.9	0.7
	bcsstk_32	110.9	6387.1	(E)	(M)	3.6	0.9
	fe_body	96.5	2095.6	(E)	(C)	(C)	(C)

Table 7.9: Comparison of the CPU times of some of the fastest force-directed and algebraic graph-drawing algorithms on challenging graphs. Explanations: (C) No drawing was generated because the algorithm is designed for connected graphs. (E) No drawing was generated because of an error in the executable. (M) No drawing was generated because the memory is restricted to graphs with ≤ 10.000 nodes. (T) No drawing was generated within 10 hours of CPU time.

We first concentrate on the drawings of the random disconnected graphs (see Figure 7.18). The drawings that are generated by GRIP separate the components, but contain large empty regions that increase the used aspect-ratio area. The drawings of the disconnected graphs that are generated by GVA (see Figure 7.18(b), (e), and (h)) contain many overlapping components.

Except FM^3 none of the tested algorithms displays the global structure of the snowflake graphs. Even the drawings of the smallest snowflake graph (see Figure 7.19(b)-(f)) leave room for improvements. However, GVA and GRIP visualize parts of its structure in an appropriate way.



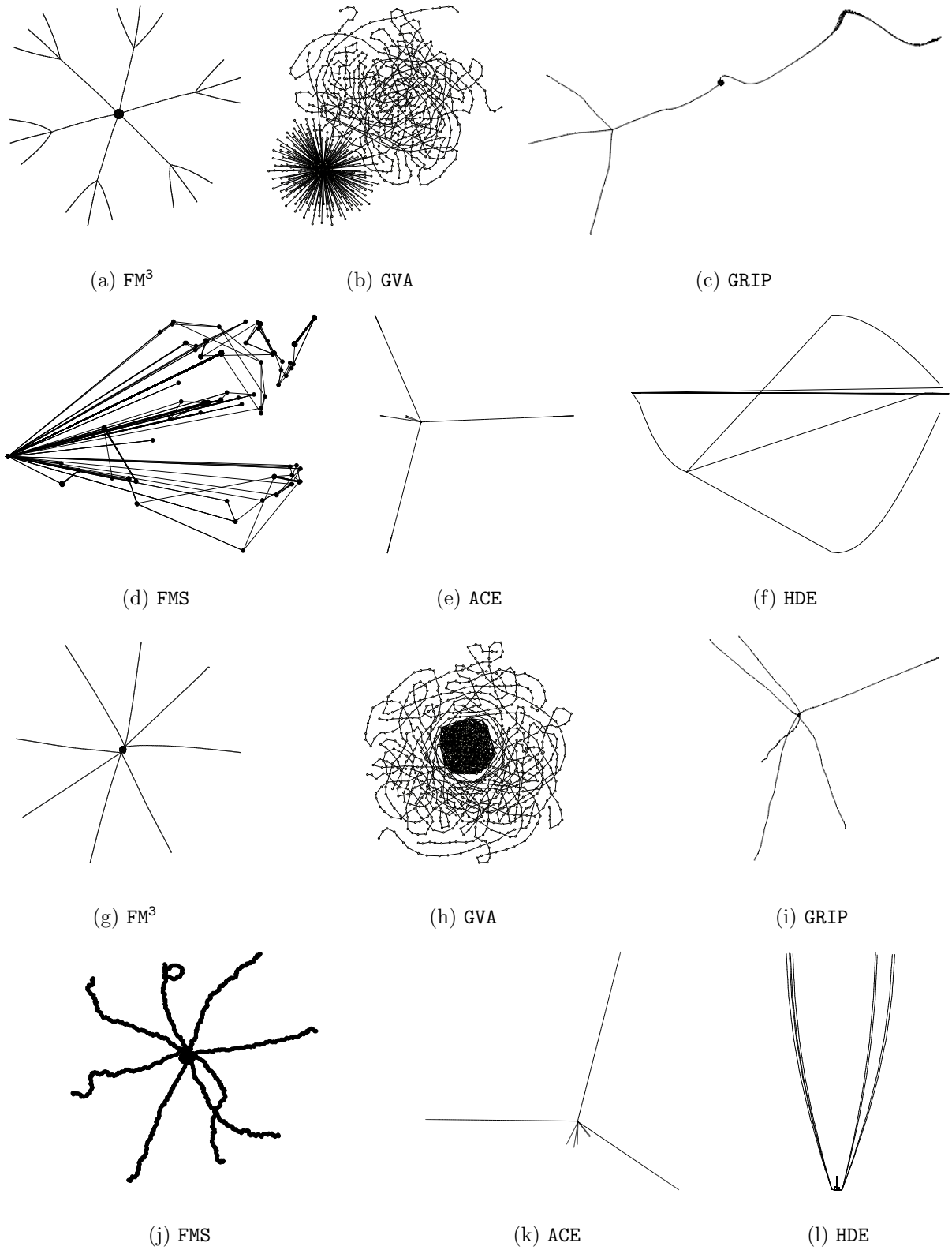
(a) FM^3 (b) GVA (c) GRIP

(d) FM^3 (e) GVA (f) GRIP

(g) FM^3 (h) GVA (i) GRIP

Figure 7.18: (a)-(c) Drawings of rnd_1_A, (d)-(f) rnd_2_A, and (g)-(i) rnd_2_B generated by different algorithms.

The drawings of the spider_A graph that are generated by GRIP, FMS, and HDE (see Figure 7.19(i), (j), and (l)) are not as symmetric as that generated by FM^3 (see Figure 7.19(g)). But they still display the global structure of the graph. The drawing generated by GVA (Figure 7.19(h)) shows the dense subregion, but GVA does not untangle the 8 paths. The paths in the drawing of ACE (Figure 7.19(k)) are not displayed in the same length although each of the paths contains the same number of nodes. We do not present the drawings of the larger spider graphs, since they are comparable with those of the spider_A graph.



(a) FM^3 (b) GVA (c) GRIP

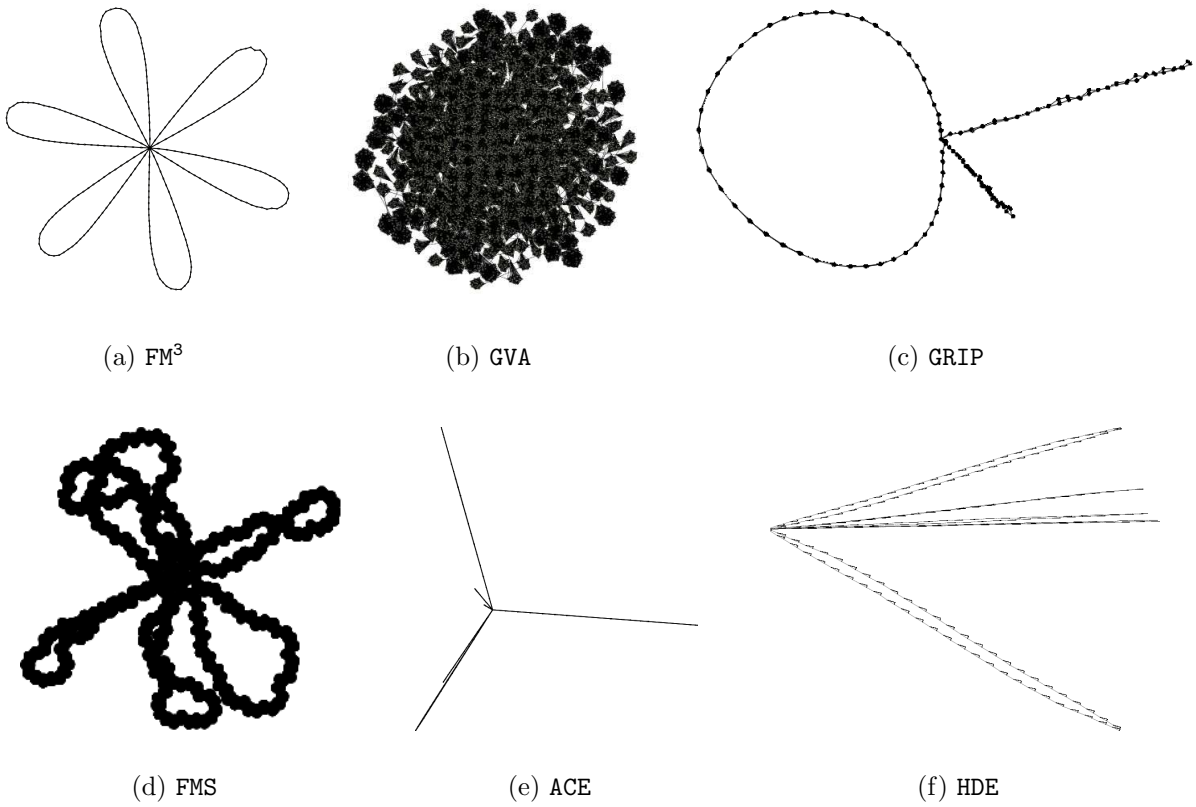
(d) FMS (e) ACE (f) HDE

(g) FM^3 (h) GVA (i) GRIP

(j) FMS (k) ACE (l) HDE

Figure 7.19: (a)-(f) Drawings of snowflake_A and (g)-(l) spider_A generated by different algorithms.

The drawings of the flower_B graph (see Figure 7.20) that are generated by FMS and HDE display the global structure of the graph. But none of the other methods displays the symmetric structure of the flower graphs as clear as FM^3. The drawing that is generated by ACE is similar to the drawings that ACE generated for the spider and snowflake graphs. The drawings of the other flower graphs are of comparable quality as the corresponding drawings of the flower_B graph.



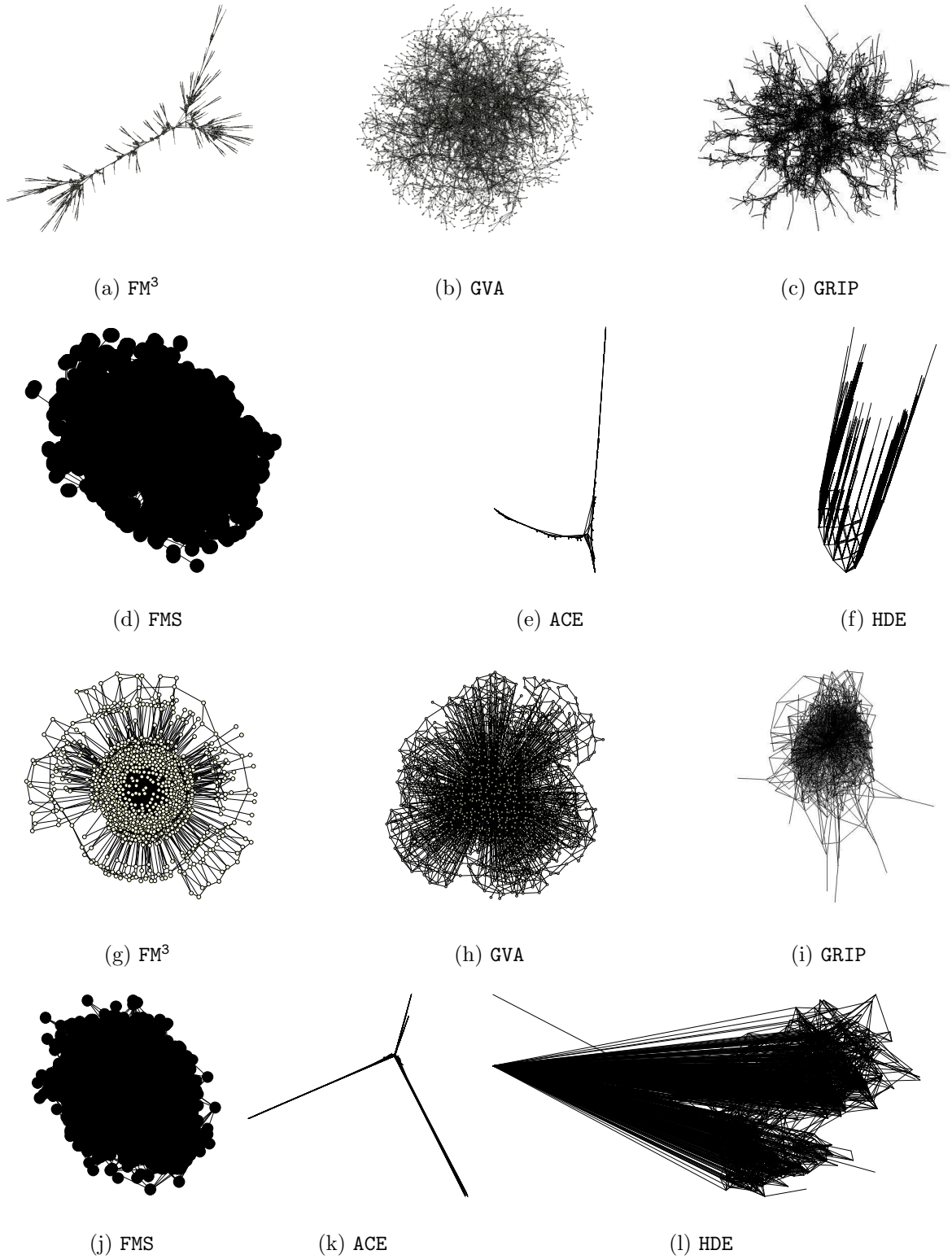
(a) FM^3 (b) GVA (c) GRIP

(d) FMS (e) ACE (f) HDE

Figure 7.20: Drawings of flower_B generated by different algorithms.

We concentrate on the challenging real-world graphs now. Since add_32 contains many biconnected components, we expect that the drawings have a tree-like shape. This structure is visualized by GVA, GRIP, ACE, and FM^3 (see Figure 7.21(a-c) and (e)). The drawings of GVA and GRIP contain many edge crossings, while the drawing of ACE displays the global structure, but hides local details.

The graphs ug_380 and dg_1087 both contain one node with a very high degree and dg_1087 additionally many biconnected components, since it is a tree. Only the drawings that are generated by GVA, GRIP, and FM^3 (see Figure 7.21(g)-(l) and Figure 7.22(a)-(f)) clearly display the central regions of these graphs. However, it can be observed that the edge lengths of the drawing of dg_1087 that is generated by FM^3 (Figure 7.22(a)) are more uniform than in the drawings of dg_1087 that are generated by GVA and GRIP (Figure 7.22(b) and (c)).



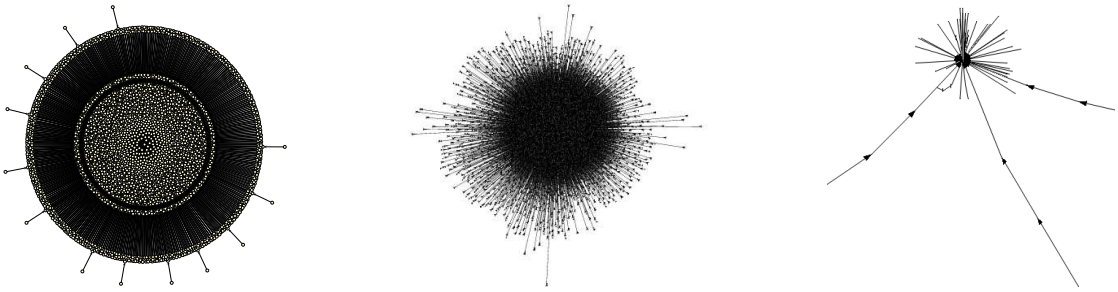
(a) FM^3 (b) GVA (c) GRIP

(d) FMS (e) ACE (f) HDE

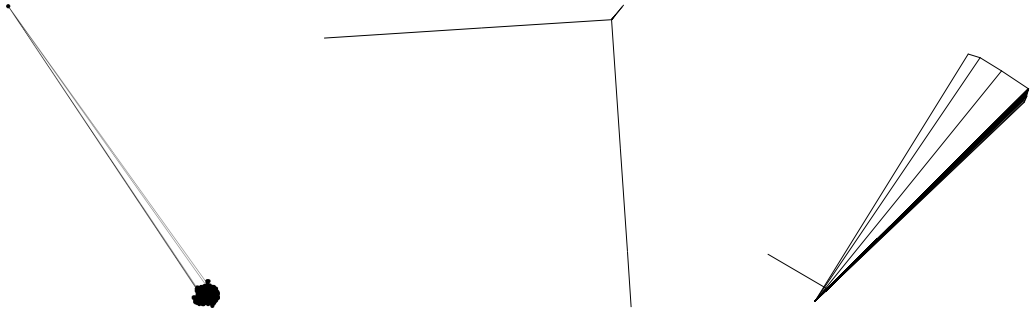
(g) FM^3 (h) GVA (i) GRIP

(j) FMS (k) ACE (l) HDE

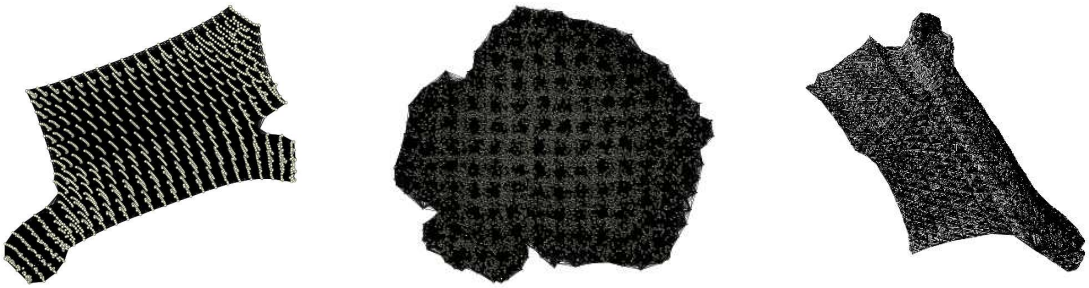
Figure 7.21: (a)-(f) Drawings of add_32 and (g)-(l) ug_380 generated by different algorithms.



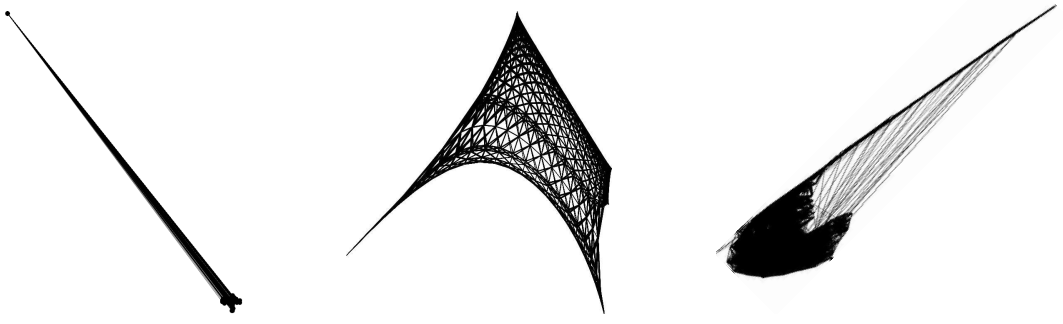
(a) FM^3 (b) GVA (c) GRIP



(d) FMS (e) ACE (f) HDE



(g) FM^3 (h) GVA (i) GRIP

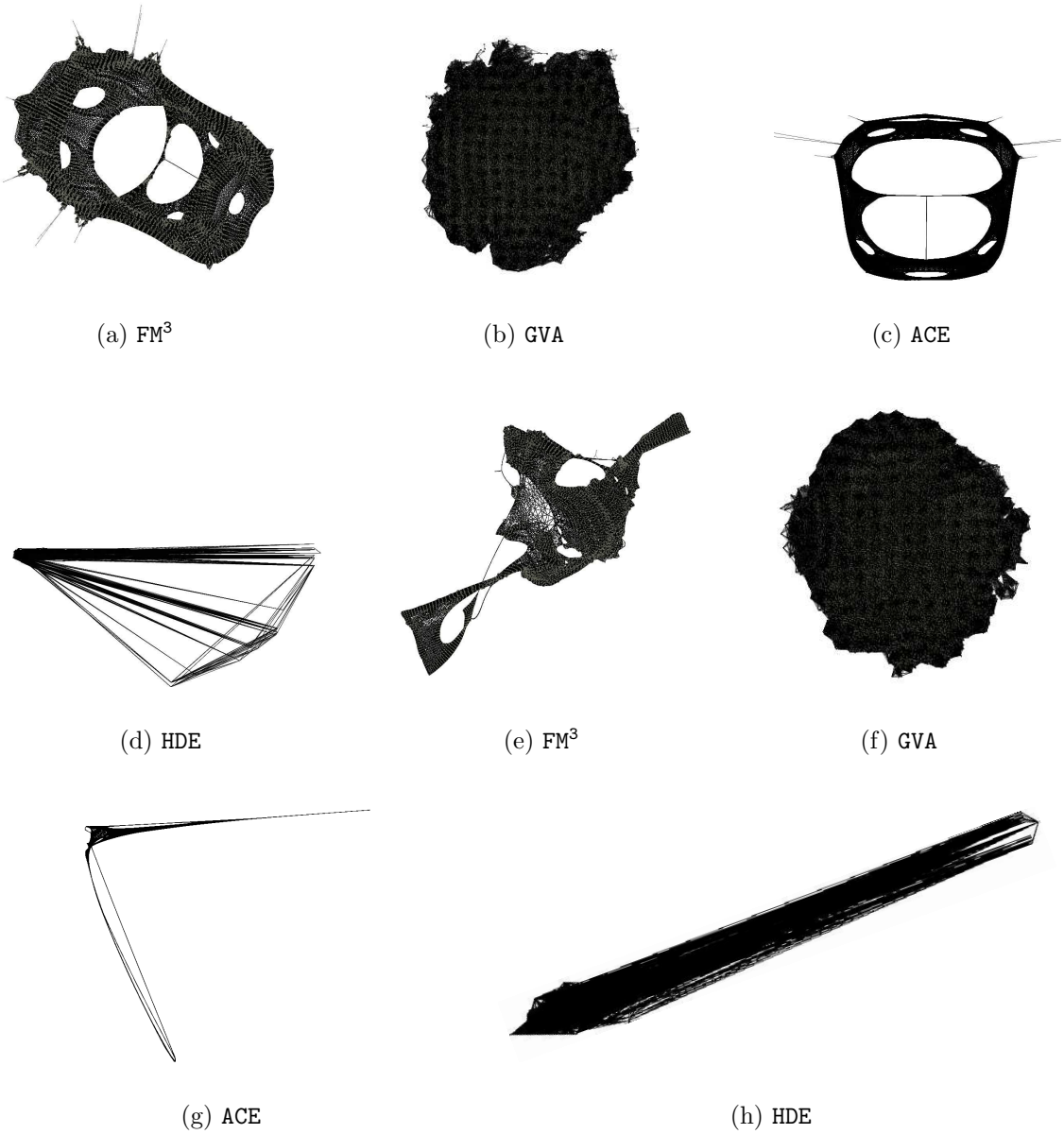


(j) FMS (k) ACE (l) HDE

Figure 7.22: (a)-(f) Drawings of dg_1087 and (g)-(l) bcsstk_33 generated by different algorithms.

Finally, we discuss the drawings of the graphs bcsstk_31_con, bcsstk_32, and bcsstk_33 that have a very high edge density.

The drawings of bcsstk_33 (see Figure 7.22(g)-(l)) that are generated by GRIP, ACE, and FM^3 are comparable and visualize the regular structure of the graph. The car body that is modeled by the graph bcsstk_31_con (see Figure 7.23(a)-(d)) is visualized by ACE and FM^3 only. All drawings of bcsstk_32 are completely different (see Figure 7.23(e)-(h)) and an evaluation of the drawings is left to the reader.



(a) FM^3 (b) GVA (c) ACE

(d) HDE (e) FM^3 (f) GVA

(g) ACE (h) HDE

Figure 7.23: (a)-(d) Drawings of bcsstk_31_con and (e)-(h)bcsstk_32 generated by different algorithms.

We can summarize that only GVA and FM³ are able to generate drawings of all tested graphs. HDE generates drawings of all connected graphs. HDE is by far the fastest method. The running time of FM³ is comparable with that of the fastest force-directed methods for kind graphs and often faster for the challenging graphs. The running time of GRIP does not increase significantly for these graphs, but it is not possible to examine its scaling on really large graphs because of an error in the implementation. The running times of FMS and ACE grow drastically for some challenging graph, while the scaling of FM³ is good for all tested graphs. All tested algorithms, except GVA, generate pleasing drawings for the kind graphs. But unlike FM³ all other tested methods cannot clearly visualize the structure of many challenging graphs.

7.7.3 Further Important Algorithms

We compared FM³ with some of the fastest force-directed and algebraic graph-drawing methods, but some important algorithms that have been sketched in Chapter 1.3 are missing. In particular, we did not test the single-level algorithms FADE [110] and JIGGLE [127, 128] that are based on the force-approximation method of Barnes and Hut [8]. We also did not test the matching-based multilevel method of Walshaw [132, 133].

Tunkelang's implementation of JIGGLE (available from [129]) does not allow the user to import graphs. Furthermore, the actual drawings are updated on the screen every couple of iterations. This makes the implementation quite slow. For example, the algorithm does not generate a crossing-free drawing of a 30×30 square grid within 10 minutes on our machine. Since the basic principles of JIGGLE and FADE are similar, we do not expect a significant better performance of FADE.

Unfortunately, an implementation of the method of Walshaw [132, 133] has not been made available to us. The reported running time [133] for drawing a square mesh that contains 103081 nodes is 431 seconds on a machine that is comparable with ours. However, challenging graphs containing nodes with a very high degree (like the snowflake graphs or dg_1087) or graphs that induce a distribution of the nodes so that $\Theta(|V|)$ nodes are contained in a tiny subregion (like the spider graphs and snowflake graphs) have not been tested [132, 133]. Since these graphs are pathological for this algorithm, anything else than a drastic increase of the running times would be surprising.

7.8 Further Experiments

7.8.1 But There Is a But!

In the previous experiments the measured CPU times exclude the time that is needed for the input and output of the data. Unfortunately, for many tested graphs the time that is needed for the input and output of the data dominates the running time of FM³. For example, Table 7.10 displays the time that is needed for importing random grid graphs that are stored in *GML* format [70] (with and without graphical information) into the LEDA/graph format with the standard functionalities provided by LEDA [95].

Graph Information			CPU Times in Seconds	
Name	$\|V\|$	$\|E\|$	Only Structural Information	Structural and Graphical Information
grid_rnd_030	865	1606	0.01	0.30
grid_rnd_040	1521	2821	0.06	0.96
grid_rnd_060	3420	6387	0.25	4.72
grid_rnd_080	6087	11441	0.75	14.33
grid_rnd_120	13698	25840	3.69	72.85
grid_rnd_160	24343	45985	11.88	233.44
grid_rnd_240	54766	103693	60.07	1194.55
grid_rnd_320	97359	184532	192.45	3833.04

Table 7.10: CPU times for importing graphs in GML formal into the LEDA/graph format

Hence, in order to make FM^3 interactive, the next step should be an improvement of the used input and output functionalities.

7.8.2 Are Energy-Minimum Drawings of Planar Graphs Crossing Free?

In the last section we will examine experimentally whether drawings of planar graphs that induce an energy-minimum configuration of the nodes in an underlying physical force model are necessarily edge-crossing free. Therefore, we investigated the force model of Fruchterman and Reingold [47] and the force model of Davidson and Harel [27] (see Section 1.2.2). We restricted on the repulsive forces induced by the nodes/particles and the attractive forces defined by the edges/springs. The desired edge length l of each edge was set to 100. The forces and the energies are defined as in the literature. In particular, suppose that nodes v_i and v_j are placed at positions p_i and p_j and that v_i and v_j are connected by an edge $e = (v_i, v_j)$. Then, the repulsive force acting on v_i due to v_j and the attractive force acting on v_i due to the spring e in the model of Fruchterman and Reingold [47] are defined as

$$F_{rep}^{v_j}(v_i) = -\frac{l^2}{\|p_j - p_i\|}\frac{p_j - p_i}{\|p_j - p_i\|} \quad \text{and} \quad F_{spring}^{e}(v_i) = \frac{\|p_j - p_i\|^2}{l}\frac{p_j - p_i}{\|p_j - p_i\|}.$$

Therefore, the total energy of the system of Fruchterman and Reingold [47] is

$$E_{FR} = \sum_{v_i \in V} \sum_{v_j \in V, v_j \neq v_i} -l^2 \log \|p_j - p_i\| + \sum_{e=(v_i,v_j)\in E} \frac{1}{3l}\|p_j - p_i\|^3.$$

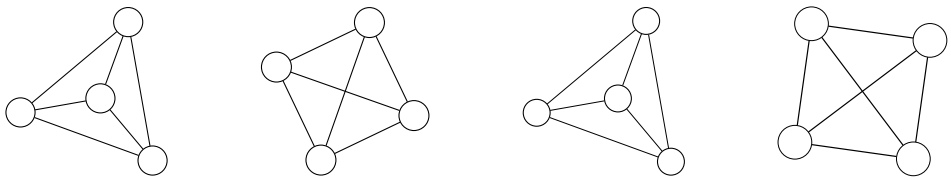
Davidson and Harel [27] define the energy of the system of repelling particles and attracting springs as

$$E_{DH} = \sum_{v_i \in V} \sum_{v_j \in V, v_j \neq v_i} \frac{\lambda_1}{\|p_j - p_i\|^2} + \sum_{e=(v_i,v_j)\in E} \lambda_2 \|p_j - p_i\|^2.$$

Choosing $\lambda_1 = l^4$ and $\lambda_2 = 1$, the forces on v_i induced by v_j and e in this system are

$$F^{v_j}_{rep}(v_i) = -\frac{2l^4}{\|p_j - p_i\|^3} \frac{p_j - p_i}{\|p_j - p_i\|} \quad \text{and} \quad F^{e}_{spring}(v_i) = 2\|p_j - p_i\| \frac{p_j - p_i}{\|p_j - p_i\|}.$$

Given these forces and energies we chose a complete graph with four nodes (K_4) as test instance. Then, we implemented for each force model an iterative graph-drawing algorithm that terminates if all nodes are in an equilibrium configuration of the induced forces. By choosing suitable initial placements, we forced the algorithms to generate both kinds of possible energy-minimal drawings: Symmetric planar drawings of the K_4 and symmetric drawings of the K_4 that each contain one edge crossing (see Figure 7.24).



(a) (b) (c) (d)

Figure 7.24: Planar and non-planar energy-minimal drawings of K_4 according to the force model of Fruchterman and Reingold ((a) and (b)) and to the force model of Davidson and Harel ((c) and (d)).

Then, we calculated the induced energies of these drawings. The results are shown in Table 7.11 and demonstrate that for both force models the energy induced by the non-planar drawing of the K_4 was lower than the energy induced by the planar drawing.

Force Model	Energy Induced by the	
	Planar Drawing of the K_4	Non-Planar Drawing of the K_4
Fruchterman & Reingold	-520347.5	-527447.0
Davidson & Harel	207846.1	189736.7

Table 7.11: Energies induced by the drawings of K_4 that are shown in Figure 7.24

We can conclude that — in general — simple physical models that rely only on identifying nodes with charged particles and edges with springs cannot guarantee that energy-minimum configurations of the nodes imply crossing-free drawings.

Conclusion

> Der gerade Weg ist der kürzeste,
> aber es dauert meist am längsten,
> bis man auf ihm zum Ziele gelangt. [1]

We have developed a new force-directed graph-drawing algorithm, called FM^3 that is based on a simple model which identifies nodes with charged particles and edges with springs. In order to find an energy-minimal configuration of the nodes in this physically motivated force model, tools from complex analysis have been combined with useful data structures and combinatorial optimization techniques. In this sense, this work joins ideas of distinct research fields.

The method FM^3 is suitable for drawing weighted and unweighted, connected and disconnected graphs with and without node attributes and allows the user to specify an aspect ratio of the desired drawing area.

The most important parts of the algorithm are the efficient multilevel step and the multipole methods $\mathtt{NM}_\mathtt{a}^2$ and $\mathtt{NM}_\mathtt{b}^2$ that are used to obtain accurate approximations of the repulsive forces acting in the system of charged particles fast. These methods are based on new strategies for building up a reduced bucked quadtree data structure and on a linear time method for approximating the potential energy in the system using this data structure and analytical tools taken from complex analysis.

For a given graph $G = (V, E)$ the worst-case running time of FM^3 is $O(|V| \log |V| + |E|)$ and only a linear amount of memory is required during the computation. This is an improvement in comparison with current force-directed methods that either do not scale sub-quadratic at all or guarantee a sub-quadratic running time only in special cases or under certain assumptions.

The algorithm creates nice drawings of a wide range of graphs, and is also very fast in practice. In particular, it draws graphs containing up to 100000 nodes in less than 5 minutes.

The experimental comparisons with other state-of-the-art graph-drawing algorithms for visualizing large graphs show that FM^3 belongs to the fastest force-directed algorithms. Furthermore, the running times of the other force-directed methods either grow drastically for

[1] Georg Christoph Lichtenberg

some tested instances or the methods can not draw these graphs. The algebraic methods ACE of Koren et al. [86, 87] and the high-dimensional-embedding approach of Harel and Koren [66] are significantly faster than all tested force-directed methods for several (respectively all) tested instances. But the quality of the generated drawings of the examined existing state-of-the-art graph-drawing methods is not convincing for all graphs. Since the practical experiments demonstrate that FM^3 even visualizes the structures of those graphs clearly that cause problems for the other tested methods, the development of FM^3 extends the class of graphs whose structures can be visualized well by force-directed or algebraic methods.

These results encourage the development of an extended version of FM^3 in order to handle constraints like fixed nodes, fixed subgraphs, or to adapt it on the requirements of special graphs like hierarchical graphs or clustered graphs.

Some parts of FM^3 might be interesting outside graph drawing, too. For example, the force-approximation methods $\mathtt{NM}^2_\mathtt{a}$ and $\mathtt{NM}^2_\mathtt{b}$ could be used in N-body simulations in natural science. One also could examine whether it is possible to use a modification of FM^3 for solving placement problems in VLSI-design. For example, one question that arises in this field is as follows: Given a large number of cells and nets. Place the cells in the plane or into a bounded square region so that no two cells overlap and the estimated total wire length is minimized.

We are aware that the new graph drawing method FM^3 does not belong to the simplest force-directed algorithms. But we have also tried to explain that it is not always the best alternative to choose the easiest way.

Bibliography

[1] S. J. Aarseth. *Gravitational N-body Simulations: Tools and Algorithms*. Cambridge monographs on mathematical physics. Cambridge University Press, 2003.

[2] S. Aluru, J. Gustafson, G. M. Prabhu, and F. E. Sevilgen. Distribution-Independent Hierarchical Algorithms for the N-body Problem. *Journal of Supercomputing*, 12:303–323, 1998.

[3] S. Aluru, G. M. Prabhu, and J. Gustafson. Truly Distribution-Independent Algorithms for the N-body Problem. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, pages 420—428, 1994.

[4] D. P. Anderson. Techniques for reducing pen plotting time. *ACM Transactions on Graphics*, 2(3):197–212, 1983.

[5] A. W. Appel. An efficient program for many-body simulations. *SIAM Journal on Scientific and Statistical Computing*, 6:85–103, 1985.

[6] The AT&T graph collection: `www.graphdrawing.org`.

[7] B. S. Baker, E. G. Coffman, and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4):846–855, 1980.

[8] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324(4):446–449, December 1986.

[9] V. Batagelj and A. Mrvar. *Graph Drawing Software*, volume XII of *Mathematics and Visualization*, chapter Pajek - Analysis and Visualization of Large Networks, pages 77–103. Springer-Verlag, 2004.

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

[11] F. Bertault. A Force-Directed Algorithm that Preserves Edge Crossing Properties. In J. Kratochvíl, editor, *Graph Drawing 1999*, volume 1731 of *Lecture Notes in Computer Science*, pages 351–358. Springer-Verlag, 1999.

[12] J. A. Board, Z. S. Hakura, W. S. Elliot, D. C. Gray, W. J. Blanke, and J. F. Leathrum. Scalable implementations of multipole-accelerated algorithms for molecular dynamics. Technical Report 94-002, Duke University, 1994.

[13] F. J. Brandenburg, M. Himsolt, and C. Rohrer. An Experimental Comparison of Force-Directed and Randomized Graph Drawing Methods. In F. J. Brandenburg, editor, *Graph Drawing 1995*, volume 1027 of *Lecture Notes in Computer Science*, pages 76–87. Springer-Verlag, 1996.

[14] U. Brandes. *Drawing Graphs*, volume 2025 of *Lecture Notes in Computer Science*, chapter Drawing on Physical Analogies, pages 71–86. Springer-Verlag, 2001.

[15] U. Brandes and D. Wagner. *Graph Drawing Software*, volume XII of *Mathematics and Visualization*, chapter visone - Analysis and Visualization of Social Networks, pages 321–340. Springer-Verlag, 2004.

[16] A. Brandt. Multilevel computations of integral transforms and particle interactions with oscillatory kernels. *Comput. Phys. Comm.*, 65:24–38, 1991.

[17] A. Brandt. Multigrid methods in latice field computations. *Nuclear Physics B (Proceedings Supplements)*, 26:137–180, 1992.

[18] I. Bruß and A. Frick. Fast Interactive 3-D Visualization. In F. J. Brandenburg, editor, *Graph Drawing 1995*, volume 1027 of *Lecture Notes in Computer Science*, pages 99–110. Springer-Verlag, 1996.

[19] C. Buchheim and M. Jünger. Detecting Symmetries by Branch & Cut. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing 2001*, volume 2265 of *Lecture Notes in Computer Science*, pages 178–188. Springer-Verlag, 2001.

[20] P. B. Callahan and S. R. Kosaraju. A Decomposition of Multidimensional Point Sets with Applications to k-Nearest-Neighbors and n-Body Potential Fields. *Journal of the Association for Computing Machinery*, 42(1):67–90, 1995.

[21] L. Carmel and D. Harel. Combining Hierarchy and Energy for Drawing Directed Graphs. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):46–57, 2004.

[22] L. Carmel, D. Harel, and Y. Koren. Drawing Directed Graphs Using One-Dimensional Optimization. In M. T. Goodrich and S. G. Kobourov, editors, *Graph Drawing 2002*, volume 2528 of *Lecture Notes in Computer Science*, pages 193–206. Springer-Verlag, 2002.

[23] J.-H. Chuang, C.-C. Lin, and H.-C. Yen. Drawing Graphs with Nonuniform Nodes Using Potential Fields. In G. Liotta, editor, *Graph Drawing 2003*, volume 2912 of *Lecture Notes in Computer Science*, pages 460–465. Springer-Verlag, 2004.

[24] E. G. Coffman, M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms . *SIAM Journal on Computing*, 9(4):808–826, 1980.

[25] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT-Press, 1990.

[26] I. F. Cruz and J. P. Twarog. 3D Graph Drawing with Simulated Annealing. In F. J. Brandenburg, editor, *Graph Drawing 1995*, volume 1027 of *Lecture Notes in Computer Science*, pages 162–165. Springer-Verlag, 1996.

[27] R. Davidson and D. Harel. Drawing Graphs Nicely Using Simulated Annealing. *ACM Transactions on Graphics*, 15(4):301–331, 1996.

[28] H. De Fraysseix, J. Pach, and R. Pollak. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

[29] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for Drawing Graphs: an Annotated Bibliography. *Computational Geometry Theory & Applications*, 4(5):235–282, 1994.

[30] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing - Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[31] R. Diestel. *Graphentheorie*. Springer-Verlag, 1996.

[32] Ugur Dogrusoz. Two-dimensional packing algorithms for layout of disconnected graphs. *Information Sciences Informatics and Computer Science*, 143(1-4):147–158, 2002.

[33] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Balanced Aspect Ratio Trees and Their Use for Drawing Very Large Graphs. In S. H. Whitesides, editor, *Graph Drawing 1998*, volume 1547 of *Lecture Notes in Computer Science*, pages 111–124. Springer-Verlag, 1998.

[34] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Planarity-Preserving Clustering and Embedding for Large Planar Graphs. In J. Kratochvíl, editor, *Graph Drawing 1999*, volume 1731 of *Lecture Notes in Computer Science*, pages 186–196. Springer-Verlag, 1999.

[35] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[36] P. Eades. Symmetry finding algorithms. *Computational Morphology*, pages 41–51, 1988.

[37] P. Eades, R. F. Cohen, and M. L. Huang. Online Animated Graph Drawing for Web Navigation. In G. Di Battista, editor, *Graph Drawing 1997*, volume 1353 of *Lecture Notes in Computer Science*, pages 330–335. Springer-Verlag, 1997.

[38] P. Eades and Q.-W. Feng. Multilevel Visualization of Clustered Graphs. In S. North, editor, *Graph Drawing 1996*, volume 1190 of *Lecture Notes in Computer Science*, pages 101–112. Springer-Verlag, 1997.

[39] P. Eades and P. Garvan. Drawing Stressed Planar Graphs in Three Dimensions. In F. J. Brandenburg, editor, *Graph Drawing 1995*, volume 1027 of *Lecture Notes in Computer Science*, pages 212–223. Springer-Verlag, 1996.

[40] P. Eades and X. Lin. Spring algorithms and symmetry. *Theoretical Computer Science*, 240:379 – 405, 2000.

[41] P. Eades and N. C. Wormald. Fixed edge-length graph drawing is NP-hard. *Discrete Applied Mathematics*, 28:111–134, 1990.

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

[43] C. J. Fisk, D. L. Caskey, and L. E. West. ACCEL: Automated circuit card etching layout. *Proceedings of the IEEE*, 55(11):1971–1982, 1967.

[44] K. Freivalds, U. Dogrusoz, and P. Kikusts. Disconnected Graph Layout and the Polyomino Packing Approach. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing 2001*, volume 2265 of *Lecture Notes in Computer Science*, pages 378–391. Springer-Verlag, 2001.

[45] A. Frick, A. Ludwig, and H. Mehldau. A Fast Adaptive Layout Algorithm for Undirected Graphs. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing 1994*, volume 894 of *Lecture Notes in Computer Science*, pages 388–403. Springer-Verlag, 1995.

[46] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.

[47] T. M. J. Fruchterman and E. M. Reingold. Graph Drawing by Force-directed Placement. *Software–Practice and Experience*, 21(11):1129–1164, 1991.

[48] G. W. Furnas. Generalized fisheye views. In *Proceedings of ACM Conference on Human Factors in Computing Systems (CHI '86)*, pages 16–23, 1986.

[49] P. Gajer, M. T. Goodrich, and S. G. Kobourov. A Multi-dimensional Approach to Force-Directed Layouts of Large Graphs. In J. Marks, editor, *Graph Drawing 2000*, volume 1984 of *Lecture Notes in Computer Science*, pages 211–221. Springer-Verlag, 2001.

[50] P. Gajer and S. G. Kobourov. GRIP: Graph Drawing with Intelligent Placement. In J. Marks, editor, *Graph Drawing 2000*, volume 1984 of *Lecture Notes in Computer Science*, pages 222–228. Springer-Verlag, 2001.

[51] E. Ganser, Y. Koren, and S. North. Topological Fisheye Views for Visualizing Large Graphs. In *Proceedings of IEEE Information Visualization 2004 (InfoVis '04)*, pages 175–182, 2004.

[52] M. R. Garey and D. S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[53] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):312–316, 1983.

[54] T. Gonzalez. Clustering to Minimize the Maximum Inter-Cluster Distance. *Theoretical Computer Science*, 38:293–306, 1985.

[55] A. Y. Grama, V. Kumar, and A. Sameh. N-body simulations using message passing parallel computers. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 355–360, 1995.

[56] A. Y. Grama, V. Sarin, and A. Sameh. Improving Error Bounds For Multipole-Based Treecodes. *SIAM Journal on Scientific Computing*, 21(5):1790–1803, 1998.

[57] Graph drawing group of foundation caesar: `www.caesar.de/graphdrawing.0.html`.

[58] L. F. Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems*. ACM distinguished dissertations. The MIT Press, Cambridge, Massachusetts, 1988.

[59] L. F. Greengard and V. Rokhlin. A Fast Algorithm for Particle Simulations. *Journal of Computational Physics*, 73:325–348, 1987.

[60] C. Gutwenger and P. Mutzel. Grid embedding of biconnected planar graphs, Extended Abstract. Technical report, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1997.

[61] S. Hachul and M. Jünger. Drawing Large Graphs with a Potential-Field-Based Multilevel Algorithm (Extended Abstract). In J. Pach, editor, *Graph Drawing 2004*, volume 3383 of *Lecture Notes in Computer Science*, pages 285–295. Springer-Verlag, 2005.

[62] R. Hadany and D. Harel. A multi-scale algorithm for drawing graphs nicely. In *Proceedings of the 25th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '99)*, volume 1665 of *Lecture Notes in Computer Science*, pages 262–277. Springer-Verlag, 1999.

[63] K. M. Hall. An r-dimensional quadratic placement algorithm. *Management Science*, 17(3):219–229, 1970.

[64] D. Harel and Y. Koren. A Fast Multi-scale Method for Drawing Large Graphs. In J. Marks, editor, *Graph Drawing 2000*, volume 1984 of *Lecture Notes in Computer Science*, pages 183–196. Springer-Verlag, 2001.

[65] D. Harel and Y. Koren. Drawing Graphs with Non-Uniform Vertices. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI' 02)*, pages 157–166. ACM Press, 2002.

[66] D. Harel and Y. Koren. Graph Drawing by High-Dimensional Embedding. In M. T. Goodrich and S. G. Kobourov, editors, *Graph Drawing 2002*, volume 2528 of *Lecture Notes in Computer Science*, pages 207–219. Springer-Verlag, 2002.

[67] D. Harel and M. Sardas. Randomized Graph Drawing with Heavy-Duty Preprocessing. *Journal of Visual Languages and Computing*, 6(3):233–253, 1995.

[68] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In S. Karin, editor, *Proceedings 9th ACM International Conference on Supercomputing '95*. ACM Press, 1995. (Formerly, Technical Report SAND93-1301, 1993).

[69] I. Herman, G. Melancon, and M. S. Marshall. Graph Visualization and Navigation in Information Visualization: a Survey. *IEEE Transactions on Visualization and Computer Graphics*, 6:24–43, 2000.

[70] M. Himsolt. GML: A portable Graph File Format. Technical report, Universität Passau, 1996. `infosun.fmi.uni-passau.de/Graphlet/GML/gml-tr.html`.

[71] T. Hrycak and V. Rokhlin. An improved fast multipole algorithm for potential fields. *SIAM Journal on Scientific Computing*, 19(6):1804–1826, 1998.

[72] M. L. Huang and P. Eades. A Fully Animated Interactive System for Clustering and Navigation of Huge Graphs. In S. H. Whitesides, editor, *Graph Drawing 1998*, volume 1547 of *Lecture Notes in Computer Science*, pages 374–383. Springer-Verlag, 1998.

[73] M. L. Huang, P. Eades, and J. Wang. Online Animated Graph Drawing using a Modified Spring Algorithm. *Journal of Visual Languages and Computing*, 9(6), 1998.

[74] D. S. Johnson. The NP-completeness column: an ongoing guide. *Journal of Algorithms*, 3:89–99, 1982.

[75] M. Jünger, G. W. Klau, P. Mutzel, and R. Weiskircher. *Graph Drawing Software*, volume XII of *Mathematics and Visualization*, chapter AGD - A Library of Algorithms for Graph Drawing, pages 149–172. Springer-Verlag, 2004.

[76] M. Jünger and P. Mutzel, editors. *Graph Drawing Software*, volume XII of *Mathematics and Visualization*. Springer-Verlag, 2004.

[77] M. Jünger and P. Mutzel. *Graph Drawing Software*, volume XII of *Mathematics and Visualization*, chapter Technical Foundations, pages 9–53. Springer-Verlag, 2004.

[78] M. Jünger and S. Thienel. The ABACUS system for branch-and-cut and price algorithms in integer programming and combinatorial optimization. *Software Practice and Experience*, 30:1325–1352, 2000.

[79] T. Kamada. *Visualizing Abstract Objects and Relations*, volume 5 of *Series in Computer Science*. World Scientific, 1989.

[80] T. Kamada and S. Kawai. An Algorithm for Drawing General Undirected Graphs. *Information Processing Letters*, 31:7–15, 1989.

[81] T. Kamps and J. Kleinz. Constraint-Based Spring-Model Algorithm for Graph Layout. In F. J. Brandenburg, editor, *Graph Drawing 1995*, volume 1027 of *Lecture Notes in Computer Science*, pages 349–360. Springer-Verlag, 1996.

[82] M. Kaufmann and D. Wagner. *Drawing Graphs: Methods and Models*. Number 2025 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[83] K. Kaugars, J. Reinfelds, and A. Brazma. A Simple Algorithm for Drawing Large Graphs on Small Screens. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing 1994*, volume 894 of *Lecture Notes in Computer Science*, pages 278–281. Springer-Verlag, 1995.

[84] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison–Wesley, 1993.

[85] Y. Koren. Graph Drawing by Subspace Optimization. In O. Deussen, C. Hansen, D. A. Keim, and D. Saupe, editors, *Proceedings of 6th Joint Eurographics (VisSym '04)*, IEEE TCVG Symposium on Visualization, pages 65–74, 2004.

[86] Y. Koren, L. Carmel, and D. Harel. ACE: A Fast Multiscale Eigenvector Computation for Drawing Huge Graphs. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis' 02)*, pages 137–144. IEEE Computer Society, 2002.

[87] Y. Koren, L. Carmel, and D. Harel. Drawing Huge Graphs by Algebraic Multigrid Optimization. *Multiscale Modeling and Simulation*, 1(4):645–673, 2003.

[88] Y. Koren's algorithms: `research.att.com/~yehuda/index_programs.html`.

[89] T. Lengauer. *Combinatorial algorithms for integrated circuit layout*. John Wiley & Sons, 1990.

[90] Prof. Dr. Carola Lipp: `www.kaee.uni-goettingen.de/personal/lipp/lipp.htm`.

[91] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16(2):346–358, 1979.

[92] P. Liu and S. N. Bhatt. Experiences with Parallel N-body Simulations. *IEEE Transactions on Parallel Distributed Systems*, 11(12):1306–1323, 2000.

[93] J. Manning. Computational complexity of geometric symmetry detection in graphs. In *Great Lakes Computer Science Conference*, volume 507 of *Lecture Nodes in Computer Science*, pages 1–7. Springer-Verlag, 1990.

[94] T. Matsuyama, L. V. Hao, and N. Nagao. A file organization for geographic information systems based on spatial complexity. *Computer Vision, Graphics, and Image Processing*, 26(3):303–318, 1984.

[95] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[96] B. Monien, F. Ramme, and H. Salmen. A parallel Simulated Annealing Algorithm for Generating 3D Layouts of Undirected Graphs. In F. J. Brandenburg, editor, *Graph Drawing 1995*, volume 1027 of *Lecture Notes in Computer Science*, pages 396–408. Springer-Verlag, 1996.

[97] Y. Ohsawa and M. Sakauchi. Multidimensional Data Management Structure with Efficient Dynamic Characteristics. *Systems, Computers, Controls*, 14(5):1193–1200, 1983.

[98] Y. Ohsawa and M. Sakauchi. The BD-tree - A new n-dimensional data structure with highly efficient dynamic characteristics. *Information Proceedings*, 83:539–544, 1983.

[99] T. Ohya, M. Iri, and K. Murota. Improvements of the incremental method for the voronoi diagram with computational comparison of various algorithms. *Journal of Operations Research*, 27(4):306–336, 1984.

[100] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen*, volume 70 of *Reihe Informatik*. BI-Wissenschaftsverlag, 1993.

[101] M. H. Overmars and J. van Leeuwen. Dynamic multi-dimensional data structures based on quad- and k-d trees. *Acta Informatica*, 17(3):267–285, 1982.

[102] H. G. Petersen, E. R. Smith, and D. Soelvason. Error estimates for the fast multipole method. II. The three-dimensional case. In *Proc. R. Soc. Lond A*, volume 448, pages 401–418, 1995.

[103] H. G. Petersen, D. Soelvason, and J. W. Perram. The very fast multipole method. *The Journal of Chemical Physics*, 101(10):8870–8876, 1994.

[104] H. G. Petersen, D. Soelvason, J. W. Perram, and E. R. Smith. Error estimates for the fast multipole method. I. The two-dimensional case. In *Proc. R. Soc. Lond A*, volume 448, pages 389–400, 1995.

[105] S. Pfalzer and P. Gibbon. *Many-body tree methods in physics*. Cambridge University Press, 1996.

[106] K. J. Pulo. Recursive Space Decomposition in Force-Directed Graph Drawing Algorithms. In *Australian Symposium on Information Visualization*, volume 9, pages 95–102, 2001.

[107] H. C. Purchase. Which Aesthetic Has the Greatest Effect on Human Understanding? In G. Di Battista, editor, *Graph Drawing 1997*, volume 1353 of *Lecture Notes in Computer Science*, pages 248–261. Springer-Verlag, 1997.

[108] H. C. Purchase, J.-A. Allder, and D. Carrington. User Preference of Graph Layout Aesthetics: A UML Study. In J. Marks, editor, *Graph Drawing 2000*, volume 1984 of *Lecture Notes in Computer Science*, pages 5–18. Springer-Verlag, 2001.

[109] H. C. Purchase, R. F. Cohen, and M. James. Validating Graph Drawing Aesthetics. In F. J. Brandenburg, editor, *Graph Drawing 1995*, volume 1027 of *Lecture Notes in Computer Science*, pages 435–446. Springer-Verlag, 1996.

[110] A. Quigley and P. Eades. FADE: Graph Drawing, Clustering, and Visual Abstraction. In J. Marks, editor, *Graph Drawing 2000*, volume 1984 of *Lecture Notes in Computer Science*, pages 197–210. Springer-Verlag, 2001.

[111] N. R. Quinn and M. A. Breuer. A Forced Directed Component Placement Procedure for Printed Circuit Boards. *IEEE Transactions on Circuits and Systems*, CAS-26(6):377–388, 1979.

[112] Max Rauner. Ziemlich verknotet. *DIE ZEIT*, 10:33–34, 2004.

[113] A. Rosenfeld, H. Samet, C. Shaffer, and R. E. Webber. Applications of hierarchical data structures to geographical information systems. Technical Report Computer Science TR-1327, University of Maryland, College Park, MD, 1983.

[114] J. K. Salmon. *Parallel hierarchical N-body methods*. PhD thesis, California Institute of Technology, 1991. UMI Order No. GAX91-37285.

[115] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, 1990.

[116] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[117] M. Sarkar and M. H. Brown. Graphical fisheye views. *Communications of the ACM*, 37(12):73–84, 1994.

[118] F. T. Scanlon. Automated placement of multi-terminal components. In *Proceedings of the 8th workshop on Design automation*, pages 143–154. ACM Press, 1971.

[119] I. Schiermeyer. Reverse-Fit: A 2-Optimal Algorithm for Packing Rectangles. In *Proceedings of the Second Annual European Symposium on Algorithms*, pages 290–299. Springer-Verlag, 1994.

[120] K. E. Schmidt and M. A. Lee. Implementing the fast multipole method in three dimensions. *Journal of Computational Physics*, 63(5):1223–1235, 1991.

[121] W. Schnyder. Embedding planar graphs on the grid. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, pages 138–148, 1990.

[122] J. P. Singh, J. L. Hennessy, and A. Gupta. Implications of hierarchical n-body methods for multiprocessor architecture. *ACM Transactions on Computer Systems*, 13(2):141–202, 1995.

[123] K. Sugiyama. *Graph Drawing and Applications for Software and Knowledge Engineers*, volume 11 of *Series on Software Engineering and Knowledge Engineering*. World Scientific, 2002.

[124] K. Sugiyama and K. Misue. A Simple and Unified Method for Drawing Graphs: Magnetic-Spring Algorithm. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing 1994*, volume 894 of *Lecture Notes in Computer Science*, pages 364–375. Springer-Verlag, 1995.

[125] R. Tamassia. Constraints in Graph Drawing Algorithms. *Constraints*, 3(1):89–122, 1998.

[126] D. Tunkelang. A practical approach to drawing undirected graphs. Technical Report CMU-CS-94-161, School of Computer Science, Carnegie Mellon University, 1994.

[127] D. Tunkelang. JIGGLE: Java Interactive Graph Layout Environment. In S. H. Whitesides, editor, *Graph Drawing 1998*, volume 1547 of *Lecture Notes in Computer Science*, pages 413–422. Springer-Verlag, 1998.

[128] D. Tunkelang. *A Numerical Optimization Approach to General Graph Drawing*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1999. CMU-CS-98-189.

[129] D. Tunkelangs's implementation of JIGGLE: `www-2.cs.cmu.edu/~quixote`.

[130] W. T. Tutte. How to draw a graph. *Proceedings of the London Mathematical Society*, 3(13):743–768, 1963.

[131] C. Walshaw's graph collection: `staffweb.cms.gre.ac.uk/~c.walshaw/partition`.

[132] C. Walshaw. A Multilevel Algorithm for Force-Directed Graph Drawing. In J. Marks, editor, *Graph Drawing 2000*, volume 1984 of *Lecture Notes in Computer Science*, pages 171–182. Springer-Verlag, 2001.

[133] C. Walshaw. A Multilevel Algorithm for Force-Directed Graph Drawing. *Journal of Graph Algorithms and Applications*, 7(3):253–285, 2003.

[134] C. Walshaw and M. Cross. Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, 2000.

[135] X. Wang and I. Miyamoto. Generating Customized Layouts. In F. J. Brandenburg, editor, *Graph Drawing 1995*, volume 1027 of *Lecture Notes in Computer Science*, pages 504–515. Springer-Verlag, 1996.

[136] M. S. Warren and J. K. Salmon. Astrophysical N-body Simulations Using Hierarchical Tree Data Structures. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 570–576, 1992.

[137] M. S. Warren and J. K. Salmon. A Parallel Hashed Oct-Tree N-Body Algorithm. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 12–21, 1993.

[138] D. S. Watkins. *Fundamentals of Matrix Computations*. John Wiley, 1991.

[139] G. Xue. An $O(n)$ time hierarchical tree algorithm for computing force fields in n-body simulations. *Theoretical Computer Science*, 197:157–169, 1998.

[140] R. Yusufov's implementation of GRIP: `www.cs.arizona.edu/~kobourov/GRIP`.

[141] F. Zhao and S. L. Johnsson. The parallel multipole method on the connection machine. *SIAM Journal on Scientific and Statistical Computing*, 12(6):1420–1437, 1991.

Erklärung

Ich versichere, dass ich die von mir vorgelegte Dissertation selbstständig angefertigt, die benutzten Quellen und Hilfsmittel vollständig angegeben und die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken im Wortlaut oder dem Sinn nach entnommen sind, in jedem Einzelfall als Entlehnung kenntlich gemacht habe; dass diese Dissertation noch keiner anderen Fakultät oder Universität zur Prüfung vorgelegen hat; dass sie – abgesehen von unten angegebenen Teilpublikationen – noch nicht veröffentlicht worden ist sowie, dass ich eine solche Veröffentlichung vor Abschluss des Promotionsverfahrens nicht vornehmen werde. Die Bestimmungen der Promotionsordnung sind mir bekannt. Die von mir vorgelegte Dissertation ist von Prof. Dr. Michael Jünger betreut worden.

Köln, 7.3.2005

Teilpublikationen

S. Hachul and M. Jünger. Drawing Large Graphs with a Potential-Field-Based Multilevel Algorithm (Extended Abstract). In J. Pach, editor, *Graph Drawing 2004*, volume 3383 of *Lecture Notes in Computer Science*, pages 285–295, Springer-Verlag, 2005