Algorithmic Aspects of Triangle-Based Network Analysis

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der Fakultät für Informatik
der Universität Fridericiana zu Karlsruhe (TH)

genehmigte

Dissertation

von

Thomas Schank

aus Offenburg

Tag der mündlichen Prüfung:	14. Februar 2007
Erste Gutachterin:	Frau Prof. Dr. Dorothea Wagner
Zweiter Gutachter:	Herr Prof. Dr. Ulrik Brandes

Ausgabe August 20, 2007

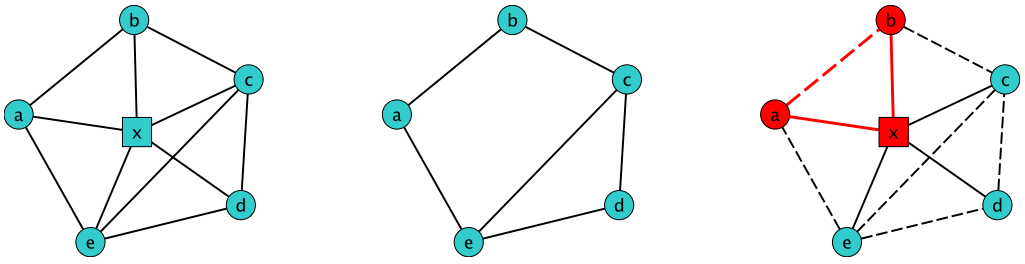
Contents

Chapter 1

Introduction

A network consists of nodes and edges between those nodes. Figure 1.1(a) depicts a network with 6 nodes and 11 edges. Many relations in our daily life can be modeled as networks. For example, a social network of people can be created by linking any two of them if they share a friendship relation. There are networks based on communication, e.g. one can construct a network of radio controlled sensors and put a link between two of them if they are close enough to receive and send information via radio waves. A network can be constructed merely from information. Let us consider the set of all existing web pages as nodes. Two of them are related if either of the two references the other. This example shows that networks can be quite huge.

Describing properties of a network is one aspect of network analysis. A network index for example maps the network to a number. The structure of the network or at least a particular aspect of it should be reflected by the associated number.



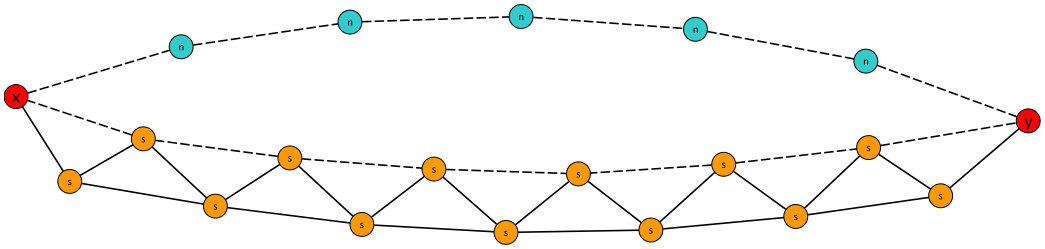
(a) network (b) neighborhood of x (c) triangle

Figure 1.1: Examples of networks.

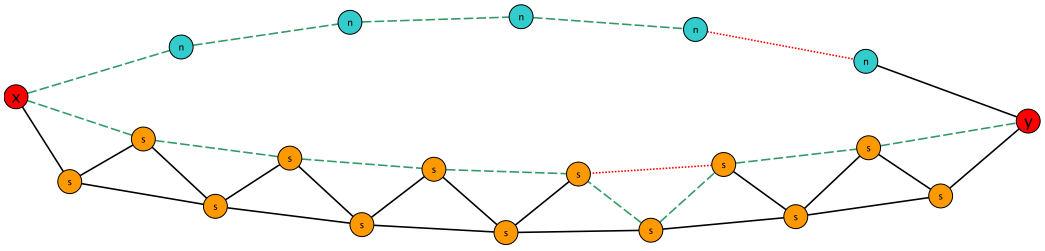
The number of nodes and the number of edges can serve as very simple network indices. From those an additional index can be determined. The ratio between the number of existing edges and the number of maximally possible edges, which can be determined from the number of nodes, is called the density of the network. For example the density for the network shown in Figure 1.1(a) is 11/15 .

All nodes that share a link with the node x are the neighbors of x. Figure 1.1(b) shows the induced graph of the neighbors of x. This leads to the definition of the neighborhood density, an index of the node x. Its value is 6/10 and it is the density of the graph depicted in Figure 1.1(b). The neighborhood density is more commonly known as the clustering coefficient of a node. This term was coined by Watts and Strogatz [1998]. The authors also defined the clustering coefficient of a graph, which is the average over all neighborhood densities of the nodes. The latter has become an extremely popular network index. The question arises how the neighborhood densities for all nodes of a network can be determined. The network induced by the neighborhood can be constructed and the number of edges and nodes in those networks can be counted. However, this approach is not the most efficient solution to the problem. We note that the number of neighbors can be determined easily. To calculate the neighborhood density it thus remains to compute the number of edges between those neighbors. In Figure 1.1(c) the edges between the neighbors of x are drawn with dashed lines. Additionally, a certain structure of the nodes a, b, and x is drawn in red color. This structure is called a triangle. Clearly, each edge between neighbors of x forms such a triangle with x, and each triangle containing the node x also contains exactly one edge between two neighbors of x. Therefore, the neighborhood density can be computed efficiently if the number of triangles of a node can be computed efficiently.

Let us consider another important issue in network analysis. The central problem in routing is to find "good" paths for traversal from one node to another node. There are two routes from x to y drawn with dashed lines in Figure 1.2(a). A package sent via the southern route uses seven edges and a package sent via the northern route only six edges. But what does happen if a link fails? Figure 1.2(b) depicts such situations. If an edge in the northern route fails the package has to be sent back to x and from there via the southern route to y. This can be quite a long route in the end. However, if an edge in the southern route fails the package can take a detour that is at most one edge longer. A closer look reveals that this is guaranteed by a triangular structure of the southern nodes. Batagelj and Zaveršnik [2003] discuss such kinds of triangular connectivity.



(a) northern and southern route.



(b) the routes under failure of edges.

Figure 1.2: Routes in a network.

Both of the two described problems in network analysis are related to the triangular structure of the network. We discuss these and similar applications in Chapter 5.

The key to handle such problems efficiently can be traced back to the algorithmic problem of finding and listing all triangles in the network. A triangle listing algorithm outputs all triangles of a graph. Consequently it cannot perform less operations than the number of triangles in the graph. A counting algorithm attributes each node of a graph with the associated number of triangles. There are two basic algorithms for triangle listing. Both are asymptotically equivalent in running time which can be coarsely bounded by the number of nodes times the number of edges. Itai and Rodeh [1978] introduced the first listing algorithm with improved running time bounded by the product of the number of edges and the square root of the latter. The achieved bound is sharp with respect to the number of edges. Chiba and Nishizeki [1985] give an improved bound that replaces the square root of the edges by the arboricity of the graph in the running time. With respect to triangle counting Alon et al. [1997] introduced the first algorithm that beats the bound given by Itai and Rodeh. All these publications discuss the counting or listing of triangles as a special case of small cliques and more commonly of short cycles. Chiba and Nishizeki extended their work also to

the listing of small subgraphs of other structure than cycles or cliques. This direction has been continued until very recently [Sundaram and Skiena, 1995; Kloks et al., 2000; Vassilevska et al., 2006]. However, triangle listing has not been considered with respect to practical applications. The work of Batagelj and Mrvar [2001] is the only one falling roughly into this category. Based on one of the two folklore algorithms all subgraphs up to tree nodes are counted. The feasibility of their approach is shown in one practical example. However, there still remains a gap in the running time of their approach and those algorithms that have an optimal running time with respect to the size of the graph. We discuss this topic in in Chapter 3, in which we consider primarily triangle listing algorithms with optimal running time with respect to the size of the graph. In Chapter 4 we investigate triangle listing and triangle counting algorithms for certain graph classes, i.e. for graphs that have certain properties.

The consideration of huge networks [Kumar et al., 2000; Abello et al., 2002; Eubank et al., 2004] requires algorithms that have at most linear or preferably sub-linear running time. In many cases it suffices to compute an approximate answer to the problem. This is likely to be accepted if the approximation can be computed much faster than the exact solution. As mentioned above the clustering coefficient is an average value, namely that of the neighborhood densities of the nodes. In such a setup sampling techniques are a common approach. Eubank et al. [2004] use such a method of sampling nodes to approximate the clustering coefficient of a graph. However, the neighborhood density of a node itself can be expressed as an average value. The question arises if this property can be exploited to approximate the clustering coefficient more directly and consequently more efficiently. We discuss this in Chapter 7.

The creation of random networks is important to understand existing networks and to test algorithms. The model of Erdős and Rényi [1959] is probably one of the best known. At that time other models were studied as well [Gilbert, 1959; Austin et al., 1959]. These models are not equivalent but very similar and for most applications the difference does not matter. A resulting graph of any of these models shares the property that its degree distribution does not show a high variation. This differs from findings in many real world networks [Faloutsos et al., 1999; Chen et al., 2001]. The linear preferential attachment graph generator considered in [Barabási and Albert, 1999; Albert et al., 1999] does generate random graphs with that property. Subsequently, it was adapted in various ways to fulfill other properties, too. Holme and Kim [2002] for example propose a method that is very close to preferential attachment, but it also considers the triangular structure. It is

supposed to achieve a high clustering coefficient of the generated network. The approach in [Li et al., 2004] changes an existing graph such that the resulting clustering coefficient is higher whilst preserving the degree distribution. We discuss an alternative method that is also based on preferential attachment in Chapter 6.

Chapter 2

Preliminaries

2.1 Notation and Mathematical Foundation

Symbols and Notation

We use lower case characters to denote numbers and functions to numbers. These might be Roman or Greek letters or other variations we think are suitable, e.g. n, ρ, π, ϖ etc. We might use $f(x)$, f_x or even simply f interchangeably, whenever the argument is clear from the context. We will use capital characters to denote sets and objects that are not simple numbers or functions to numbers, e.g. G, Π, Υ, Δ, etc.

We try to keep this rules throughout this work. However, if it does not seem convenient or some other notation is wildly used in literature we will follow the common usage. An example for this is the $\mathcal{O}$-notation in the next section. In general we allow ourselves to abuse notation to a more mnemonic appearance if the meaning is clear from context and it is very unlikely that misunderstandings will happen. In rare cases we may not even give a formal definition of those notations, again only if their meaning is clear from context.

Numbers and Asymptotic Growth

We denote positive integer numbers including zero with $\mathbb{N}$ and the real numbers with $\mathbb{R}$.

To capture the asymptotic behavior of polynomials, Landau's so called *O-notation* has become common practice. Let f and g be functions $f, g : \mathbb{N} \to \mathbb{N}$. Then f is in $\mathcal{O}(g)$, or equivalently g is in $\Omega(f)$, if there exist two numbers

$n_0 \in \mathbb{N}$ and $c \in \mathbb{R}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. Informally we say that f grows at most as fast as g. To describe equivalent asymptotic growth we define $\Theta(g)$ as $\mathcal{O}(g) \cap \Omega(g)$. Finally, if there exists a number $n_0 \in \mathbb{N}$ for any $c \in \mathbb{R}_{>0}$ such that $f(n) \leq c \cdot g(n)$, we say that f is in $o(g)$ for all $n > n_0$ or equivalently g is in $\omega(f)$.

Probability

A pair $(\Omega, \mathbb{P})$ with Ω a finite set and $\mathbb{P}$ a mapping from $\mathcal{P}(\Omega)$ all subsets of Ω to $\mathbb{R}$ is a finite *probability space* if $\mathbb{P}[\Omega] = 1$, $\mathbb{P}[A] \geq 0$ for $A \subset \Omega$, and $\mathbb{P}[A \cup B] = \mathbb{P}[A] + \mathbb{P}[B]$ for $A \cap B = \emptyset$. The mapping $\mathbb{P}$ is called the *probability measure* on Ω. The set Ω is called the *sample space* and a subset A of Ω is called an *event*.

An arbitrary mapping $X : \Omega \to \mathbb{R}$ is called a *random variable* and we can define shorthand

$$\mathbb{P}[X \leq x] = \mathbb{P}[\{\omega \in \Omega : X(\omega) \leq x\}]$$

for example. The *distribution* or *density* p of a random variable X is the probability measure $p : X(\Omega) \subset \mathbb{R} \to [0, 1]$ for which

$$p(x) \quad = \quad \mathbb{P}[X = x] \quad = \quad \mathbb{P}[\{\omega \in \Omega : X(\omega) = x\}].$$

The *expectation* of a random variable X with its distribution p is defined as

$$\mathbb{E}[X] = \sum_{x \in X(\Omega)} x \cdot p(x).$$

The *variance* of a random variable is

$$\operatorname{var}[X] = \mathbb{E}\left[X^2\right] - (\mathbb{E}[X])^2,$$

and the *standard deviation* is the square root of the variance

$$\sigma(X) = \sqrt{\operatorname{var}[X]}.$$

2.2 Graphs

A *graph* (*network*) consists of a set of *vertices* (*nodes*) V and a set of *edges* E. An edge connects two nodes. We denote the size of V with n, and the size of E with m. In an *undirected graph* $G = (V, E)$ the set of edges is conveniently

regarded as a set of two-element subsets of the nodes $E \subset \{\{u, w\} \subset V\}$. Two nodes u and v are *adjacent* if they are connected by an edge $\{u, v\} \in E$. The edge $\{u, v\}$ is *incident* to the nodes u and v.

For a *directed graph* or *digraph* $D = (V, A)$ we preferably use the the name *arcs* for the elements connecting vertices. The set of arcs A is conveniently regarded as a subset of the crossproduct of vertices $A \subset V \times V$. In the following, the properties are mainly defined for undirected graphs. However, in most cases they will have a natural extension to directed graphs. The natural mapping from a directed graph $D = (V, A)$ to an undirected graph $G = (V, E)$ is to connect two vertices in G whenever there exists an arc in either direction in A. Then G is called the *underlying* undirected graph of the digraph D.

A graph without looping edges, i.e. $\{u, u\} \notin E$ for all nodes u, is called *simple*. If not otherwise noted, we restrict all graphs to be simple. Note that

$$m \leq \binom{n}{2} = \frac{n(n-1)}{2} \tag{2.1}$$

holds for a simple undirected graph G; written in asymptotic notation $m \in \mathcal{O}(n^2)$. A graph is *complete* if equality holds in Equation 2.1. Such a graph with $m = \binom{n}{2}$ edges is also called an *n-clique*. The ratio

$$\rho(G) = \frac{m}{\binom{n}{2}} \tag{2.2}$$

is called the *density* of a graph and a series of graphs is called *dense* if m is in $\Omega(n^2)$.

Degree and Neighborhood

The *neighborhood* $\Gamma(v)$ of a vertex v is the set of all nodes adjacent to v

$$\Gamma(v) = \{u \in V : \{u, v\} \in E\}.$$

The neighborhood of a subset S of V is defined as

$$\Gamma(S) = \{v \in V \subset S : \exists s \in S : \exists \{s, v\} \in E\}.$$

The *degree* $d(v)$ of a node v is defined as the number of incident edges or the size of its neighborhood

$$d(v) = |\Gamma(v)|\,.$$

The *maximal degree* of G is defined by

$$d_{\max}(G) = \max\{d(v) : v \in V\}.$$

When summing up the degrees of all nodes, each edge is accounted twice, once for each of its endpoints. The corresponding equation

$$\sum_{v \in V} d(v) = 2m \tag{2.3}$$

is known as the *Handshake Lemma*.

In a digraph $D = (V, A)$ we distinguish between *in-degree*

$$d_{\text{in}}(v) = |\{u \in V : \exists (u, v) \in A\}|$$

and *out-degree*

$$d_{\text{out}}(v) = |\{w \in V : \exists (v, w) \in A\}|.$$

The degree of a vertex in a directed graph is the degree of the vertex in the underlying undirected graph.

Subgraphs

The subsets $V' \in V$ and $E' \in E$ define a *subgraph* $G' = (V', E')$ of $G = (V, E)$, if every incident vertex of an edge in E' is in V'. The *node induced* subgraph $G[V']$ of $V' \subset V$ is defined by $G[V'] = (V', E')$ where $E' = \{\{u, w\} \in E : u, w \in V'\}$. The *edge induced* subgraph $G[E']$ is defined by $G[E'] = (V', E')$ where V' contains all of the endpoints of the edges in E' and no more vertices. We allow us to use the short and mnemonic notation of $G \setminus v$ for $G[V \setminus \{v\}]$ and likewise $G \setminus e$ for $G[E \setminus \{e\}]$.

For a graph $G = (V, E)$ and $\mathcal{P}(G)$ the set of all subgraphs of G we define $\mathcal{V} : \mathcal{P}(G) \to V$ and $\mathcal{E} : \mathcal{P}(G) \mapsto E$ such that $\mathcal{V}(G') = V'$ and $\mathcal{E}(G') = E'$ for $G' = (V', E')$ in $\mathcal{P}(G)$.

Paths, Cycles, Connectedness and Distance

A sequence $v_1, e_1, v_2, e_2, \ldots, e_{n-1}, v_n$ is called a *path* P from v_1 to v_n in G if for all $i : e_i = \{v_i, v_{i+1}\} \in E$. The number of edges of a path is the *length* of the path. A path is simple if $e_i \neq e_j$ for $i \neq j$.

We call a path a *cycle* C if $v_1 = v_n$. An edge $\{v_i, v_j\}$ is a *chord* of a cycle if it is not an edge *in* the cycle: $\{v_i, v_j\} \neq e_k$ for all edges e_k of the cycle. A chordless cycle is a *hole*.

A graph is *connected* if there exists a path between any two pair of nodes in G. If not otherwise noted, we require a graph to be connected from now on. This implies $m \in \Omega(n)$ which allows us to write shorthand $\mathcal{O}(m)$ instead of $\mathcal{O}(n+m)$.

The *distance* $\text{dist}(u, w)$ of two nodes u and w is the length of a shortest path between the nodes.

Trees

A graph is called a *tree* T if it is cycle free and connected. A tree has exactly $m = n-1$ edges. A node of degree one is called a *leaf*. A tree T is a *spanning tree* of a graph G if T is a subgraph of G and $\mathcal{V}(T) = \mathcal{V}(G)$.

Rooted Trees. Let us declare some node $r \in \mathcal{V}(T)$ the *root* of T. For some edge $\{u, w\}$ let us assume that u is closer to the root than w, i.e. $\text{dist}(u, r) = \text{dist}(w, r) - 1$. Then u is the *predecessor* of w (with respect to the root r). Likewise, w is a *successor* of w. Note, that every node except the root r has exactly one predecessor.

Tree Orders. The *subtree* for a node v with respect to a root r is the induced subgraph of all nodes w, for which v lies on the path from w to r. A numbering of the nodes $(v_1, \ldots, v_2)$ is in *preorder* if any node v appears directly before all other nodes of v's subtree. Likewise a numbering is in *postorder* if for all nodes v all nodes of its subtree appear directly before v.

Neighborhood Density, Clustering Coefficient and Transitivity

We briefly introduce the clustering coefficient and the transitivity here. Section 5.4 on page 85 has a more in-depth exploration along with extended bibliographical remarks.

Triangles and Wedges. A *triangle* Δ_{uvw} (or less specific simply Δ) of G is a complete subgraph of three distinct nodes u, v, and w: $\mathcal{V}(\Delta_{uvw}) = \{u, v, w\}$ and $\mathcal{E}(\Delta_{uvw}) = \{\{u, v\}, \{v, w\}, \{w, u\}\}$. Motivated by this, the number of triangles of node v is defined as the number of edges between neighbors

$$\delta(v) = |\mathcal{E}(G[\Gamma(v)])| \,. \tag{2.4}$$

Since each triangle contains three nodes (hence if we sum over all nodes each triangle is counted three times) the following definition is intuitive

$$\delta(G) = \frac{1}{3}\sum_{v \in V} \delta(v). \tag{2.5}$$

For an n-clique $\delta = \binom{n}{3}$ holds, and we get we get $\delta(G) \in \mathcal{O}(n^3)$ for a graph in general. In terms of m this yields

$$\delta(G) \in \mathcal{O}\left(m^{3/2}\right). \tag{2.6}$$

Note that a simple cycle of length three would have been a structurally equivalent definition of a triangle. As a consequence, triangles occasionally appear in literature as small complete subgraphs or short cycles.

A *wedge* Υ_{uvw} of G is a subgraph of three distinctive nodes u, v, and w: $\mathcal{V}(\Upsilon_{uvw}) = \{u, v, w\}$ and two edges $\mathcal{E}(\Upsilon_{uvw}) = \{\{u, v\}, \{v, w\}\}$. Unlike a triangle, a wedge has a structurally distinguished center node v. The number of wedges of a node v with $d(v) \geq 2$ is then defined as

$$\tau(v) = \binom{d(v)}{2}, \tag{2.7}$$

and summing the wedges of all nodes defines the number of wedges of the graph

$$\tau(G) = \sum_{v \in V} \tau(v). \tag{2.8}$$

A wedge can also be seen as a path of length two. This alternative definition is rather common in literature. If we consider an n-clique K_n we get $3\,\delta(K_n) = \tau(K_n)$, and in general the following inequality

$$3\,\delta(G) \leq \tau(G) \tag{2.9}$$

holds for a graph G.

Neighborhood Density. The *neighborhood density* of a vertex v is the density of the subgraph induced by the neighbors of v. It can be equivalently expressed by the quotient of triangles and wedges of v

$$\varrho(v) \;=\; \rho(G[\Gamma(v)]) \;=\; \frac{\delta(v)}{\tau(v)}. \tag{2.10}$$

Note that we require $d(v) \geq 2$ as in Equation 2.7. The neighborhood density is also known as the *clustering coefficient* of the node [Watts and Strogatz, 1998].

Clustering Coefficient. The *clustering coefficient* [Watts and Strogatz, 1998] $c(G)$ of a graph G with $d(v) \geq 2$ for all vertices is the average neighborhood density of all nodes in G

$$c(G) = \frac{1}{|V|} \sum_{v \in V} \varrho(v). \tag{2.11}$$

Transitivity. The *transitivity* [Newman et al., 2002] of a graph G is defined as

$$t(G) = \frac{3\,\delta(G)}{\tau(G)}. \tag{2.12}$$

As well as the clustering coefficient, it involves wedges and triangles and its value ranges between zero and one by Equation 2.9.

2.3 Algorithms

An *algorithm* is a procedure that computes the desired output from any allowed input instance in a finite number of *steps*. The running time of an algorithm assigns the number of computational steps to any input. Formally the running time T_A of algorithm A is a mapping from the set of all allowed input instances Π to the natural numbers, $T_A : \Pi \to \mathbb{N}$.

In the context of this work the input is a graph $G = (V, E)$ in most cases. To compare the running times we combine equivalent input elements into a collection, e.g. the set $\mathbb{G}_{n,m}$ of all graphs with n nodes and m edges. The *worst case running time* of an algorithm A is then the maximum running time over all elements in such a collection. For the case of $\mathbb{G}_{n,m}$ graphs the definition reads

$$T'_A(\mathbb{G}_{n,m}) = \max\{T_A(G) : G \in \mathbb{G}_{n,m}\}.$$

In many cases the only sensible general approach to compare running times is the worst case running time. Therefore, we will use the term running time, respectively the symbol T, in the meaning of worst case running time from now on.

We say that an algorithm has *linear running time*, if its running time is in $\mathcal{O}(i + o)$ where i is the size of the input and o is the size of the output. We follow common usage in giving a statement like "algorithm A can be implemented with a running time in $\mathcal{O}(n)$" in the shorthand form of "algorithm A has $\mathcal{O}(n)$ running time".

2.4 Core Structure

The concept of *cores* was introduced in [Seidman, 1983]. The *k-core* $\mathfrak{C}_k$ of a graph G is the largest node induced subgraph with minimum degree of k for each node

$$\mathfrak{C}_k(G) = G[\{v : d_{\mathfrak{C}_k}(v) \geq k\}]. \tag{2.13}$$

We use $d_{G'}(v)$ for the degree of v in the subgraph G'. The *core number* $\kappa(v)$ of a node v is the maximum k of all cores it belongs to

$$\kappa(v) = \max\{k : v \in \mathcal{V}(\mathfrak{C}_k(G))\}, \tag{2.14}$$

and the core number $\kappa(G)$ of a graph is the maximum core number of all of its nodes

$$\kappa(G) = \max_{v \in V}\{\kappa(v)\}. \tag{2.15}$$

The nodes with maximum core number induce the *main core* or simply *the core* of a graph.

The *degeneracy* of a graph is the largest over all minimum degrees of all subgraphs, see for example [Bollobás, 2004]. It can be easily shown that the degeneracy is equal to the core number κ of a graph. Also related is the *arboricity*, the minimum number of disjoint forrest to cover all edges of a graph, which is asymptotically growing linear in the core number κ.

Besides giving an interesting decomposition of the graph in its own interest, the k-core is used in many other graph related problems. Computing the core is one of the applied heuristics to find cliques in a graph, as a k-clique is necessarily contained in the $k-1$-core. An example is given in [Batagelj and Zaveršnik, 2002]. The core concept can also be used to thin out graphs to their most interesting part, this is applied for example in [Gaertler and Patrignani, 2004].

The recursive definition in Equation 2.13 is quite compact. However, it is not directly suitable for computing the k-core. An alternative and constructive characterization of the k-core is to remove iteratively all nodes with degree less then k until only nodes with degree k or larger remain.

A straight-forward implementation of this idea leads to a non-linear time algorithm for computing the k-core of a graph. However, with some modifications it is possible to achieve a linear time algorithm. We give a rough description in the following. Algorithm 2.1 computes the k-core of a graph G. First each node v is added to the set indexed by the degree of v (line 1). Then all nodes in the sets L_i with $i < k$ are removed from G (line 2). In

course of removing a node v, the sets L_i of all its neighbors are updated (line 3). To achieve running time in $\mathcal{O}(m)$ the data structures have to be carefully chosen and some details must be considered. Since this is not our focus the interested reader might want to consult [Batagelj and Zaveršnik, 2002].

Algorithm 2.1: k-CORE

Input: Graph $G = (V, E)$, $k \in \mathbb{N}$
Output: k-core of Graph G
Data: $L_0, \ldots, L_{d_{\max}}$ initially empty sets of nodes

```
1 forall v ∈ V do
      append v → L_{d(v)};
2 while ∃i < k : L_i ≠ ∅ do
      v ← pop L_{min{i:L_i≠∅}};
3     forall u ∈ Γ(v) do
          remove u ← L_{d(u)};
          append u → L_{d(u)−1};
      remove v from G;
```

Corollary 1 (folklore, [Batagelj and Zaveršnik, 2002]) *The k-core of a graph can be computed in linear time.*

We note that Algorithm 2.1 can be easily modified such that all the cores from 0 to $\kappa(G)$ are computed in one single run whilst maintaining running time in $\mathcal{O}(m)$; e.g. by enclosing line 2 in a loop from 0 to $\kappa(G)$.

Chapter Notes

In this chapter we introduced the most important concepts and required definitions in a very compact manner. For a more in-depth exploration many textbooks are available. Diestel [2000] and Bollobás [1998] give an introduction to graph theory. Algorithms are treated by Cormen, Leiserson, Rivest, and Stein [2001]. Ross [2003] gives an introduction to probability which is discussed in context of algorithms by Motwani and Raghavan [1995].

Chapter 3

Algorithms for Listing and Counting all Triangles in a Graph

Chapter Introduction

This chapter is devoted to algorithms for listing all triangles of a graph efficiently. The problem of triangle finding, counting and listing has been studied before mainly from a theoretical point of view. The two basic algorithms are attributed to the folklore. They both achieve running time in $\mathcal{O}(d_{\max} \cdot m)$. Itai and Rodeh [1978] introduced the first algorithm with running time in $\mathcal{O}(m^{3/2})$. With respect of the number of edges the bound achieved by Itai's and Rodeh's algorithm cannot be further improved for triangle listing algorithms. However, Chiba and Nishizeki [1985] can bound the running time of their algorithm in $\mathcal{O}(a \cdot m)$, in which a is the *arboricity* of the graph. They also show that $a \leq \sqrt{m}$.

The $\Omega(m^{3/2})$ running time bound for listing algorithms does not hold for counting algorithms. In this context Alon, Yuster, and Zwick [1997] give a counting algorithm with running time only dependent on m that indeed beats this bound. This is achieved with fast matrix multiplication. Triangle counting or listing can be seen as a special case of counting or listing either short cycles or small cliques. All the mentioned publications extend in one or both of this fields.

All of the above publications do not consider any practical evaluation of the presented algorithms. Batagelj and Mrvar [2001] give an extension of one

of the folklore algorithms to count triads (all directed subgraphs with up to three nodes). This is also the only work we are aware of that contains practical results. They show, that an implementation of their algorithms achieves very fast execution times. However, the algorithm they used has a running time in $\omega(m^{3/2})$.

Contribution and Related Work. Our goal is to close the gap between algorithms that are known to perform very good in practice, but do not achieve the theoretic best running times, and those algorithms that do so. We focus on the optimal bound given by the number of edges in $\mathcal{O}(m^{3/2})$. Based on the known folklore algorithms we develop new triangle listing algorithms that achieve optimal running time with respect to the input size m. Our experimental work shows that one of the new algorithms which we call "*forward*" performs best with respect to execution time. This algorithm essentially uses the same basic operation as the algorithm of Batagelj and Mrvar [2001]. But it also uses some further techniques that guarantee a running time in $\mathcal{O}(m^{3/2})$. Our experiments show that it performs much better for graphs with unbounded maximal degree.

The essence of our contribution was published in [Schank and Wagner, 2005b] and there exists also an extended technical report [Schank and Wagner, 2005c].

All the introduced listing algorithms can be implemented with linear space consumption. However, they can differ with respect to a constant factor. In a follow up work to [Schank and Wagner, 2005b], Latapy [2006] focuses mainly on memory consumption. Most notably he gives an improvement of our algorithm *forward* that he calls *compact-forward*.

Organization. This chapter is organized as follows. We introduce some basic concepts and notations directly following this section. In Section 3.1 we will first introduce some well known algorithms for listing and counting triangles followed by new variations of those. We will also prove that most of these new variations have good asymptotic running time bounds. The most promising candidates are evaluated in Section 3.2. We do this by experiments of the execution time on generated and real world graphs. We close this chapter with a conclusion.

Preliminaries

Recall that a triangle Δ_{uvw} of a graph $G = (V, E)$ is a three node subgraph with $\mathcal{V}(\Delta) = \{u, v, w\} \subset V$ and $\mathcal{E}(\Delta) = \{\{u, v\}, \{v, w\}, \{w, u\}\} \subset E$. We use the symbol $\delta(G)$ to denote the number of triangles in graph G. Note that an n-clique has exactly $\binom{n}{3}$ triangles and asymptotically $\delta_{\text{clique}} \in \Theta(n^3)$. In dependency to m we have accordingly $\delta_{\text{clique}} \in \Theta\big(m^{3/2}\big)$ and by concentrating as many edges as possible into a clique we get the following lemma.

Lemma 1 *There exists a series of graphs G_m such that*

$$\delta(G_m) \in \Theta\big(m^{3/2}\big)\,.$$

We call an algorithm a *triangle counting algorithm* if it outputs the number of triangles $\delta(v)$ for each node v and a *triangle listing algorithm* if it outputs the three participating nodes of each triangle. As a listing algorithm requires at least one operation per triangle we get the following by Lemma 1.

Corollary 2 *A triangle listing algorithm has an asymptotic lower bound running time in $\Omega(n^3)$ in terms of n and in $\Omega\big(m^{3/2}\big)$ in terms of m.*

Note that one could further distinguish between counting the triangles for the whole graph and counting the triangles for all of the nodes. However, there is no complexity difference in running time known up to date and actually all known algorithms for counting the triangles of the whole graph do this by counting them for all nodes first. For this reason we do not distinguish between these two cases.

3.1 Algorithms

We will list some algorithms for listing and counting triangles in the following.

3.1.1 Basic Algorithms

Algorithm "*try-all*"

A very simple algorithm with a running time in $\Theta(n^3)$ is to check for edges between nodes of every three element subset of V.

Algorithm "*matrix-multiplication*"

If A is the adjacency matrix of graph G, then the diagonal elements of A^3 contain two times the number of triangles of the corresponding node. This immediately leads to a counting algorithm with running time $\Theta(n^3)$. However, it can also be implemented with fast matrix multiplication to run in $\Theta(n^\gamma)$ time, where γ is the *matrix multiplication exponent*. It is known that $\gamma \leq 2.376$ [Coppersmith and Winograd, 1990].

Corollary 3 (folklore) *All triangles of an undirected graph can be counted in $\mathcal{O}(n^\gamma)$ time.*

Algorithm "*tree-lister*"

Itai and Rodeh [1978] presented the fist algorithm that finds a triangle if it exists in time in $\mathcal{O}\left(m^{3/2}\right)$. Their original algorithm stops after finding the first triangle. However, it can be easily extended to list all triangles of a graph without additional cost in asymptotic running time. We call the triangle listing variant "*tree-lister*". The pseudo code is shown in Algorithm 3.1.

Algorithm 3.1: TRIANGLE LISTER "*tree-lister*"

```
   Input: graph G = (V, E)
   Output: all triangles of G
   while E ≠ ∅ do
       T ← Spanning Tree of G;
       for {v, w} ∈ E \ E(T) do
1          if {pred(v), w} ∈ E then
               output Triangle {pred(v), v, w};
2          else if {v, pred(w)} ∈ E then
               output Triangle {pred(w), v, w};
3      E ← E \ E(T)
```

Theorem 1 *All triangles of an undirected graph can be listed in $\mathcal{O}\left(m^{3/2}\right)$ time.*

We will reproduce a proof for the above theorem with the following two Lemmas.

Lemma 2 *Algorithm* tree-lister *outputs all triangles of a graph and no more. Each triangle is listed once and only once.*

Proof: Clearly Algorithm 3.1 outputs only triangles. It remains to be shown that each triangle is listed exactly once.

As a tree T is cycle free, each triangle has at least one edge in $E \setminus \mathcal{E}(T)$. Assume first, that all the edges of an arbitrary triangle are contained in $E \setminus \mathcal{E}(T)$, then the triangle is preserved after line 3. Since E is empty in the end, for any triangle there exists an iteration step where an edge is in $E \setminus \mathcal{E}(T)$ and at least one edge is in $\mathcal{E}(T)$. Clearly after this iteration the triangle is destroyed. It remains to be shown that the triangle is listed exactly once during this iteration.

Now, let Δ with $\mathcal{V}(\Delta) = \{u, v, w\}$ be the triangle destroyed in the current iteration. Without loss of generality, let $\{v, w\}$ be in $E \setminus \mathcal{E}(T)$. Then, $\text{pred}(v) = u$ (the triangle is listed in line 1) or $\text{pred}(w) = u$ (the triangle is listed in line 2, if not already listed in line 1) must be true, otherwise T would not be a valid spanning tree or none of the edges would be in T, a contradiction to the assumption. □

Lemma 3 *Algorithm* tree-lister *can be implemented to run in* $\mathcal{O}(m^{3/2})$ *time.*

Proof: First note that with appropriate data structures the steps inside the outer loop can be performed in $\mathcal{O}(m)$ time. We have to show that the outer loop is executed at most $c \cdot \sqrt{m}$ times, for some constant c.

In each loop at least the root node of each computed tree is split from the graph and so the number of connected components γ increases in each iteration at least by one. We consider the costs while there are less (respectively more) than $n - \sqrt{m}$ components.

We first consider the costs until there are at most $n - \sqrt{m}$ components. Note, that in each iteration $n - \gamma$ edges are removed. Hence, at least $n - \gamma \geq n - (n - \sqrt{m}) = \sqrt{m}$ edges are removed in each step. After $\sqrt{m}$ such steps m edges would be removed and consequently there cannot be more than $\sqrt{m}$ steps.

Now, we consider the case for $\gamma \geq n - \sqrt{m}$. The largest component has node size less than $n - \gamma \leq \sqrt{m}$. As already mentioned, in each iteration at least one node is split from each component, and hence, after at most $\sqrt{m}$ steps, we are left with only disconnected nodes. □

We have seen the first existing m-optimal algorithm for triangle listing. However, it is not a good candidate for an implementation. Let us mention the

iterative construction of spanning trees and the tests for existence of edges with respect to the execution time. Moreover the dynamic modification of the graph imposes comparatively high costs either in respect of execution time or memory usage. This will become clear when we will have a closer look on the experiments carried out on the algorithms which will be introduced in the following sections.

3.1.2 Algorithm *node-iterator* and Related Algorithms

Algorithm "*node-iterator*"

Algorithm *node-iterator* iterates over all nodes and tests for each pair of neighbors if they are connected by an edge. Line 1 ensures that each triangle is listed only once. In practice the required order "$<$" can be easily gained e.g. from array position or pointer addresses.

Algorithm 3.2: TRIANGLE LISTER "*node-iterator*"

Input: graph $G = (V, E)$, arbitrary order $<$ of the nodes
Output: all triangles of G
for $v \in V$ **do**
  for *all pairs of Neighbors* $\{u, w\}$ *of* v **do**
    if $\{u, w\} \in G$ **then**
1   **if** $u < v < w$ **then**
        output triangle $\{u, v, w\}$;

The asymptotic running time is given by the expression

$$\sum_{v \in V} \binom{d(v)}{2} \tag{3.1}$$

and therefore bounded by $\mathcal{O}(d_{\max}^2 \cdot n)$.

Corollary 4 (folklore) *The algorithm* node-iterator *can be implemented to run in* $\mathcal{O}(d_{\max}^2 \cdot n)$ *time.*

Algorithm "*ayz*" and "*listing-ayz*"

The counting algorithm due to Alon, Yuster, and Zwick [1997] combines the techniques of the algorithms *node-iterator* and *matrix-multiplication*.

Theorem 2 ([Alon et al., 1997]) *All triangles of an undirected graph can be counted in time in* $\mathcal{O}\left(m^{2\gamma/(\gamma+1)}\right) \subset \mathcal{O}(m^{1.41})$.

Informally the algorithm splits the node set into low degree vertices $V_{\text{low}} = \{v \in V : d(v) \leq \beta\}$ and high degree vertices $V_{\text{high}} = V \setminus V_{\text{low}}$ where $\beta = m^{\gamma-1/\gamma+1}$ and γ is the matrix multiplication exponent. The standard method *node-iterator* is performed on the low degree nodes and (fast) matrix multiplication on the induced subgraph of V_{high}, see Algorithm 3.3 for details. The running time is in $\mathcal{O}\left(m^{2\gamma/(\gamma+1)}\right)$.

Algorithm 3.3: TRIANGLE COUNTER "*ayz*"

Input: Graph $G = (V, E)$; matrix multiplication parameter γ
Output: number of triangles $\delta(v)$ for each node

```
1  β ⟵ m^((γ−1)/(γ+1));
2  for v ∈ V do
       δ(v) ← 0;
       if d(v) ≤ β then
           V_low ← V_low ∪ {v};
       else
           V_high ← V_high ∪ {v};
3  for v ∈ V_low do
       for all pairs of Neibours {u, w} of v do
4          if edge between u and w exists then
5              if u, w ∈ V_low then
                   for v ∈ {v, u, w} do
                       δ(v) ← δ(v) + 1/3;
6              else if u, w ∈ V_high then
                   for v ∈ {v, u, w} do
                       δ(v) ← δ(v) + 1;
7              else
                   for v ∈ {v, u, w} do
                       δ(v) ← δ(v) + 1/2;
8  A ← adjacency matrix of node induced subgraph of V_high;
9  M ← A^3;
10 for v ∈ V_high do
11     δ(v) ← δ(v) + M(i, i)/2 where i is index of v;
```

We will give an extended version of the proof from [Alon et al., 1997] for Theorem 2.

Proof: The correctness of the algorithm can be easily seen by checking the case distinction for different types of triangles consisting of exactly three (line 5), two (line 7), one (line 6) or zero (line 11) low degree nodes.

To prove the time complexity we first note that the lines 1, 2, 8, and 10 can be clearly implemented to run in linear time. We proof that loop beginning at line 3 is in time in $\mathcal{O}\left(m^{2\gamma/(\gamma+1)}\right)$, that is $\mathcal{O}\left(\sum_{v \in V_{\text{low}}} \binom{d(v)}{2}\right) \subset \mathcal{O}\left(m^{2\gamma/(\gamma+1)}\right)$. Note that with a hashed edge set the test of line 4 can be implemented to run in constant time. The following tests in line 5, 6 and 7 can be clearly performed in constant time. The claim follows from $\sum_{v \in V_{\text{low}}} \binom{d(v)}{2} \in \mathcal{O}(m\beta)$. Now, we proof that line 9 is in time in $\mathcal{O}\left(m^{2\gamma/(\gamma+1)}\right)$. We have to show that $\mathcal{O}\left(n_{\text{high}}^{\gamma}\right) \subset \mathcal{O}\left(m^{2\gamma/(\gamma+1)}\right)$, which follows from $n_{\text{high}} \leq 2m/\beta$. □

We can derive a listing algorithm *listing-ayz* with a running time in $\mathcal{O}\left(m^{3/2}\right)$ by using *node-iterator* also for the induced subgraph of V_{high}, in this case $\beta = \sqrt{m}$.

Corollary 5 *Algorithm* listing-ayz *has a running time in* $\mathcal{O}\left(m^{3/2}\right)$.

Note that β is only bounded asymptotically to achieve the running times in the lemmas above. Latapy [2006] shows that it behaves well for a range of values. However, an optimal value still depends on the graph structure. We use $\beta = \sqrt{m}$ when we compare the algorithms experimentally in Section 3.2.

Algorithm *node-iterator-core*

The algorithm *node-iterator-core* listed in Algorithm 3.4 takes iteratively a node v with currently lowest degree, proceeds with it as in algorithm *node-iterator*, and then removes node v.

Note that an order of nodes that would give a node with currently minimal degree, as required by the algorithm, can be computed in the same fashion as the core numbers, see Section 2.4. Consequently, line 1 does not impose more than an additional linear time cost during the execution of the algorithm.

We recall further from Section 2.4 that the k-core of a graph is the largest node induced subgraph with minimum degree at least k. The core number $\kappa(v)$ of a node v is the maximum k of all cores it belongs to. The core number

Algorithm 3.4: TRIANGLE LISTER "*node-iterator-core*"

```
Input: Graph G = (V, E)
Output: all triangles of G
while V ≠ ∅ do
1   v ← node with currently minimal degree;
    for all pairs of Neighbors {u, w} of v do
        if {u, w} ∈ G then
            output triangle {u, v, w} ;
    remove v from G;
```

of a graph κ_G is the maximal core number of all of its nodes. Hence, the current degree of the node v from line 1 is less or equal than $\kappa(v)$, and the running time is bounded by

$$\sum_{v \in V} \binom{\kappa(v)}{2}. \tag{3.2}$$

Analogously to algorithm *node-iterator*, we can bound the running time by $\mathcal{O}(\kappa_G^2 \cdot n)$. Note that after removing all nodes v with $\kappa(v) \leq \sqrt{m}$ the remaining graph is a subgraph of the node induced subgraph V_{high} of *listing-ayz*. Hence, this algorithm is an improvement to *listing-ayz* and the running time is in $\mathcal{O}\big(m^{3/2}\big)$, too.

3.1.3 Algorithm *edge-iterator* and Derived Algorithms

Algorithm "*edge-iterator*"

The algorithm *edge-iterator* iterates over all edges and compares the neighborhood of the two incident nodes. For an edge $\{u, w\}$ the nodes $\{u, v, w\}$ induce a triangle if and only if node v is present in both neighborhoods $\Gamma(u)$ and $\Gamma(w)$. It is possible to compare two neighborhoods $\Gamma(u)$ and $\Gamma(w)$ in $\mathcal{O}(d(u) + d(w))$ time if those neighborhoods are stored as sorted adjacency arrays. The sorting can be achieved in linear time* or in $\mathcal{O}\big(\sum_{v \in V} d(v) \log d(v)\big) \subset \mathcal{O}(m \log n)$ time with standard sorting methods which has been used in our implementation. We will see in the experimental section how this preprocessing influences the overall execution time.

*e.g. by proceeding in a similar manner as in Algorithm 3.6

We display algorithm *edge-iterator* in Algorithm 3.5. Note that the listing is not presented in the shortest and most compact way. We list it as given to keep the differences minimal to an algorithm that will be introduced later (Algorithm 3.7 on page 35). The sorted array representation guarantees that the function "nextNeighborIndex" can be performed in constant time. Line 2 ensures that each triangle is listed only once.

Algorithm 3.5: TRIANGLE LISTER "*edge-iterator*"

Input: graph G, array of vertices $(v_1, \ldots, v_n)$ in arbitrary order, adjacencies arrays sorted in the same order as the array of vertices

Output: all triangles

Functions:

$$\text{firstNeighborIndex}(v_i) = \begin{cases} \min\{\nu : v_\nu \in \Gamma(v_i)\} & \text{if exists} \\ n & \text{else} \end{cases}$$

$$\text{nextNeighborIndex}(v_i, j) = \begin{cases} \min\{\nu : v_\nu \in \Gamma(v_i), \nu > j\} & \text{if exists} \\ n & \text{else} \end{cases}$$

```
for v_i = (v_1, ..., v_n) do
    forall v_l ∈ Γ(v_i) do
        j ← firstNeighborIndex(v_i);
        k ← firstNeighborIndex(v_l);
1       while j < n and k < n do
            if j < k then
                j ← nextneighborindex(v_i, j);
            else if k < j then
                k ← nextNeighborIndex(v_l, k)
            else
2               if i < k < l then
                    output triangle{v_k, v_i, v_l};
                j ← nextNeighborIndex(v_i, j);
                k ← nextNeighborIndex(v_l, k);
```

For now we will disregard the preprocessing in the running time, which can then be expressed with

$$\sum_{\{u,w\} \in E} d(u) + d(w). \tag{3.3}$$

As with the previous algorithms we can give some less accurate bounds for the running time which are: $\mathcal{O}(d_{\max} \cdot m) \subset \mathcal{O}(n \cdot m)$.

Corollary 6 (folklore, [Batagelj and Mrvar, 2001]) *Algorithm 3.5 lists exactly the triangles of a graph. It can be implemented to run in* $\mathcal{O}(d_{\max} \cdot m)$ *time.*

Comparing $\mathcal{O}(d_{\max} \cdot m)$ with $\mathcal{O}(d_{\max}^2 \cdot n)$ of *node-iterator* suggests that *edge-iterator* is an improvement to *node-iterator*. However, this is not true. Consider the following amortized analysis: We split the costs $d(u) + d(w)$ for an edge $\{u, w\}$ in $d(u)$ and $d(w)$ units and assign $d(u)$ to node u and $d(w)$ to node w. In the outer loop each node v is passed $d(v)$ times. Hence, the running time captured by Equation 3.3 can be equivalently expressed with

$$\sum_{v \in V} d(v)^2. \tag{3.4}$$

Corollary 7 *Disregarding preprocessing, algorithm* edge-iterator *has the same asymptotic time complexity as algorithm* node-iterator.

The algorithm *edge-iterator* is equivalent to an algorithm introduced by Batagelj and Mrvar [2001]. It has been implemented within the software package Pajek [Batagelj and Mrvar, 1998].

Algorithm "*edge-iterator-hashed*"

This algorithm is based on *edge-iterator*. If we use a hashed container for the neighborhoods, we can ask for every node of the smaller container whether it is present in the larger container in $\mathcal{O}(1)$ time. This leads to a running time asymptotically growing with

$$\sum_{\{u,w\} \in E} \min\{d(u), d(w)\}. \tag{3.5}$$

It has been shown in [Chiba and Nishizeki, 1985] that the expression of Equation 3.5 is in $\mathcal{O}\big(m^{3/2}\big)$.

Algorithm "*forward*"

This is a refinement of algorithm *edge-iterator*. Instead of comparing the full neighborhood of two adjacent nodes, a subset A of those is compared. See Algorithm 3.6 for the pseudo code and Figure 3.1 for an example.

As suggested by Figure 3.1 the algorithm is conveniently regarded on the directed graph induced by the order of the nodes. The size of the data

Algorithm 3.6: TRIANGLE LISTER "*forward*"

```
Input: graph G, array of vertices (v_1, ..., v_n) in order of in increasing
       degrees
Output: all triangles
Data: initially empty array of nodes for each node v: A(v);
for v_i = (v_1, ..., v_n) do
    for v_l ∈ Γ(v_i) do
        if i < l then
            foreach v ∈ A(v_i) ∩ A(v_l) do
                output triangle {v, v_i, v_l} ;
            append v_i → A(v_l)
```

structure $A(v)$ is then bounded by the in-degree of node v. Node also that all arrays A are sorted with respect of the input order of the nodes during the whole execution. Hence, the computation of the set of the common nodes of $A(v_i)$ and $A(v_l)$ in line 1 can be performed in $|A(v_i)| + |A(v_l)|$ steps. The running time for an arbitrary order is then bounded by the expression

$$\sum_{\{u,w\}\in E} d_{\text{in}}(u) + d_{\text{in}}(w). \tag{3.6}$$

As a "heuristic" to minimize Equation 3.6, the nodes are considered in non increasing order of their degrees, which implies a preprocessing in $\mathcal{O}(n \log n)$. This order suffices to achieve the desired $\mathcal{O}\left(m^{3/2}\right)$ bound.

Lemma 4 *Algorithm* forward *lists all triangles of a graph and no more. Each triangle is listed once and only once.*

Proof: Let Δ_{uvw} be a triangle and without loss of generality $u < v < w$ with respect to the used order of the nodes. Then, there exists a time step t_1 when $v_i = u$ and $v_l = v$ during the execution of Algorithm 3.6 when u is appended to $A(v)$, because of $u < v$. Likewise, there exist time steps t_2 (respectively t_3) when $v_i = u$ and $v_l = w$ (resp. $v_i = v$ and $v_l = w$). The time steps t_1 and t_2 happen before time step t_3, again because $u < v < w$. Consequently the node u is in both arrays $A(v)$ and $A(w)$ at time step t_3 and the triangle is listed. Thus, we have shown, that all triangles are listed.

Clearly nothing else but triangles are listed. Since the ordering $u < v < w$ induces exactly one time step when the listing happens, each triangle is listed

edge	$A(b)$	$A(c)$	$A(d)$	$A(e)$	triangles
1 (a,b)	a				
2 (a,c)	a	a			
3 (a,d)	a	a	a		
4 (a,e)	a	a	a	a	
5 (b,c)	a,b	a	a	a	$\{a,b,c\}$
6 (b,d)	a,b	a	a,b	a	
7 (d,e)	a,b	a	a,b	a,d	$\{a,d,e\}$

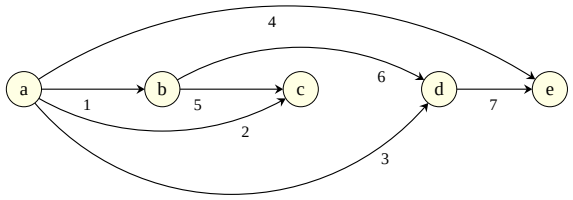


Figure 3.1: An example for algorithm *forward*.

once and only once. □

Lemma 5 *Algorithm* forward *can be implemented to have a running time in* $\mathcal{O}(m^{3/2})$.

Proof: We actually show that Equation 3.6 is in $\mathcal{O}(m^{3/2})$ for the chosen order based on non increasing degrees. We can bound the expression of Equation 3.6 by $\mathcal{O}(\max\{d_{\text{in}}\} \cdot m)$, similar as in the case of algorithm *edge-iterator*. Now, we show that $d_{\text{in}}(v) \in \mathcal{O}(\sqrt{m})$ for all nodes $v \in V$.

Assume that there exists a node v with $d(v) > \sqrt{m}$, otherwise $d_{\text{in}}(v) \leq \sqrt{m}$ for all nodes is trivially true. Now, consider the vertices $v_1, \ldots, v_n$ where without loss of generality $d(v_1) \geq \ldots \geq d(v_n)$ holds, i.e. in the nodes are in order of non increasing degrees. Let k be the index where $d(v_k) > \sqrt{m}$ but $d(v_{k+1}) \leq \sqrt{m}$. Clearly, $d_{\text{in}}(v_i) \leq \sqrt{m}$ holds for $i > k$.

It remains to be shown that $d_{\text{in}}(v_i) \in \mathcal{O}(\sqrt{m})$ for $i \leq k$. First note that $k\sqrt{m} \leq \sum_{i \leq k} d(v_i)$. Moreover the Handshake Lemma gives $\sum_{i \leq k} d(v_i) \leq \sum_{i \leq n} d(v_i) = 2m$. It follows that $k \leq 2\sqrt{m}$ and we can conclude that $d_{\text{in}}(v_i) \leq 2\sqrt{m}$ for $i \leq k$, too. □

Let us remark that Chiba and Nishizeki [1985] did also use the order of nodes sorted by non increasing degree to list a all triangles in a graph. However,

their approach does work differently. It specifically requires dynamic graph modifications. The authors mention the usage of doubly linked lists, which we already mentioned to be too memory consuming.

Algorithm "*forward-hashed*"

We can combine the methods of *forward* with *edge-iterator-hashed*. The upcoming experiments will show if this manifests in an improved execution time.

$$\sum_{\{u,w\}\in E} \min\{d_{\text{in}}(u), d_{\text{in}}(w)\}. \tag{3.7}$$

Algorithm "*compact-forward*"

Very recently Latapy [2006] proposed an improvement to *forward*. The basic idea is to use iterators to compare the subsets of adjacencies, as it is done in *edge-iterator*. To this end, the adjacencies must be sorted and the comparison must be stopped once a certain index is reached. We give Latapy's algorithm, adopted to our notation, in Algorithm 3.7.

Note that there are only three small differences of Algorithm 3.7 compared to Algorithm 3.5. First algorithm *compact-forward* uses an ordering with non increasing degree as algorithm *forward*. Second the while-loop of line 1 now stops at index i instead of n. Third the test in line 2 of Algorithm 3.5 can be omitted.

Further note that stopping at index l in line 1 corresponds to comparing only the neighborhood with nodes which are before node v_i in the given order. This is exactly what algorithm *forward* does.

Corollary 8 ([Latapy, 2006]) *Algorithm 3.7 (compact-forward) lists all triangles of a graph. It can be implemented to run in time in* $\mathcal{O}\left(m^{3/2}\right)$.

Algorithm *forward* and *compact-forward* are very similar and have the same asymptotic worst case bounds in running time and space consumption. However, algorithm *compact-forward* does not require the additional arrays A. According to [Latapy, 2006] it is therefore more time and space efficient in practice.

Algorithm 3.7: TRIANGLE LISTER "*compact-forward*"

Input: graph G, array of vertices $(v_1, \ldots, v_n)$ in order of non increasing degrees, adjacencies arrays sorted in the same order as the array of vertices

Output: all triangles

Functions:

$$\text{firstNeighborIndex}(v_i) = \begin{cases} \min\{\nu : v_\nu \in \Gamma(v_i)\} & \text{if exists} \\ n & \text{else} \end{cases}$$

$$\text{nextNeighborIndex}(v_i, j) = \begin{cases} \min\{\nu : v_\nu \in \Gamma(v_i), \nu > j\} & \text{if exists} \\ n & \text{else} \end{cases}$$

```
for v_i = (v_1, ..., v_n) do
    forall v_ℓ ∈ Γ(v_i), ℓ < i do
        j ← firstNeighborIndex(v_i);
        k ← firstNeighborIndex(v_ℓ);
1       while j < ℓ and k < ℓ do
            if j < k then
                j ← nextNeighborIndex(v_i, j);
            else if k < j then
                k ← nextNeighborIndex(v_ℓ, k);
            else
                output triangle{v_k, v_i, v_ℓ};
                j ← nextNeighborIndex(v_i, j);
                k ← nextNeighborIndex(v_ℓ, k);
```

3.1.4 Overview of the Algorithms

Figure 3.2 shows an overview of the presented algorithms with their guaranteed running time. The interesting listing algorithms will be experimentally compared in the next section.

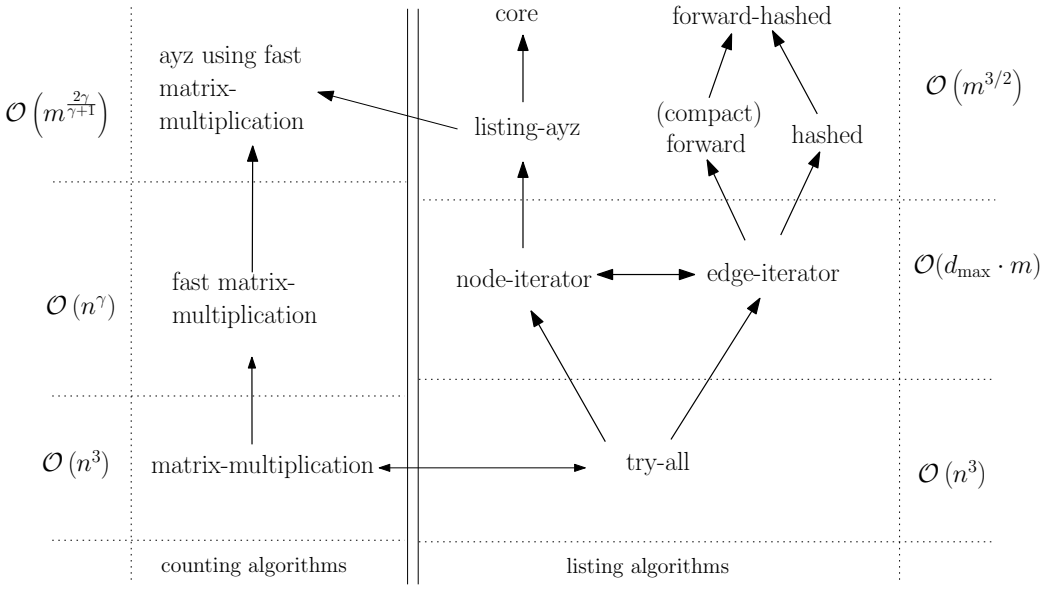


Figure 3.2: Overview of the algorithms.

3.2 Experimental Results

Notes to the Implementation and Experiments

The algorithms were implemented in `C++` using the *Standard Template Library* and compiled with the *gnu* `g++` compiler version 3.4 with Options "-g -O2". The experiments were carried out on a 64-bit machine with two AMD Opteron processors clocked at 2.20GHz. The implemented algorithms only used one of the processors and were terminated if they used more than 6GB of memory. The execution times of the implemented algorithms were measured in seconds using the `getrusage` function.

The graphs are represented using node pointers of `std::vector` type for the nodes and their adjacency data structure. The algorithms *node-iterator*,

listing-ayz and *node-iterator-core* use hashing for testing edge existence in constant time. The hash function combines two random numbers of size `size_t` with `XOR`, one for each node. The `__gnu_cxx::hash_map` was used as a hash container. Creating the random bits is not contained in the execution time of the experiments, whereas filling the container is included. We also compare the performance between a hashed container and a balanced binary tree in Section 19.

Let us note that a straight forward implementation of algorithm *node-iterator-core* requires doubly linked adjacency lists, due to the dynamic modification of the graph. Such data structures turned out to be too space consuming to be competitive. However, in the case of this algorithm we were able to use an equivalent static implementation instead.

The algorithm *edge-iterator* requires to find common nodes in two adjacency data structures. For that purpose the used `std::vector` containers are sorted in a preprocessing step using the `std::sort` function. The sorting is included in the execution time of the algorithms. The generation of the node order for *forward* is also included in the execution time.

The implemented algorithms are evaluated on several networks originating from real world as well as on generated graphs. The algorithms are tested in two ways. On the one hand we list the execution time of the algorithms. On the other hand we give the *number of triangle operations*, which in essence captures the asymptotic running time of the algorithm without preprocessing. The kind of operation considered as a triangle operation differs for the various algorithms but in all cases it represents the number of triangle tests, e.g. for the algorithm *node-iterator* the number of triangle operations is equal to $\sum_{v \in V} \binom{d(v)}{2}$ and for the *edge-iterator* equal to $\sum_{v \in V} d(v)^2$.

Experiments on Real World Networks

We will evaluate the introduced algorithms of Section 3.1 on three networks: a road network (Section 3.2), a movie actor collaboration network (Section 3.2) and one generated from hyper links of an Internet sub domain (Section 3.2).

Road Network Germany

This network is based on the roads in Germany. Hence, it is very sparse, almost planar, has very low average and maximum degree, and in consequence,

a very low deviation from the average degree, see Figure 3.3(a). The standard deviation from the average degree is an interesting measure. The square of it bounds how much improvement is possible for the more refined algorithms compared to the two standard algorithms *node-iterator* and *edge-iterator*.

The performance of the algorithms is listed in Figure 3.3(b) and Figure 3.3(c). As expected algorithm *edge-iterator* has the highest asymptotic effort in the number of triangle operations. However, in terms of execution time it outperforms all the other algorithms. The two algorithms *edge-iterator-hashed* and *forward-hashed*, which use hashed data structures for every node, perform badly.

Movie Actor Network 2004

This graph is constructed from *The Internet Movie Database* of the year 2004. Each node represents an actor. Two actors share a link if and only if they ever played together in a movie. The properties and results are shown in Figure 3.4.

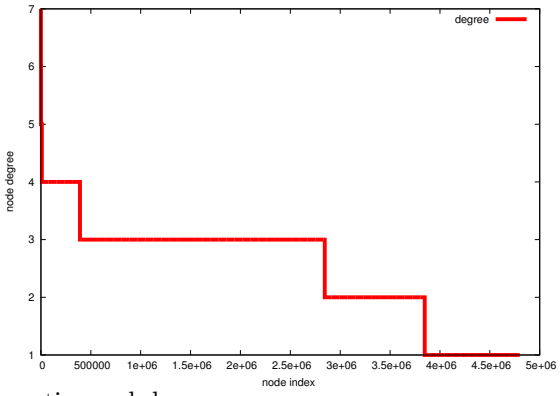
This network has a more interesting structure compared to the network of Section 3.2. The degree distribution is somewhat skewed with a std. deviation of 183 from the average degree, see Figure 3.4(a). The algorithms *node-iterator-core* and *forward-hashed* are very efficient with respect to the number of triangle operations, Figure 3.4(c). As in Section 3.2 the algorithm *listing-ayz* is no improvement to algorithm *node-iterator*, since $\sqrt{m} = 5252$ is higher than the maximum degree. Again *edge-iterator* performs relatively well in execution time, see Figure 3.4(b), considering highest number of triangle operations, see Figure 3.4(c). The implementation of algorithm *forward* performs best in execution time.

Notre Dame WWW

In this Network each node represents a web page within the `nd.edu` domain. An edge between two web pages exists if one of them was linking the other. Properties and results are shown in Figure 3.5.

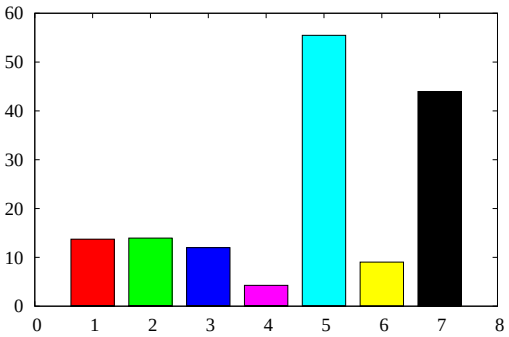
Compared to the road network of Section 3.2 and even to the collaboration network of Section 3.2 the difference between maximal degree to the average degree is more pronounced and there is a very sharp bend in the degrees of the nodes visible, see Figure 3.5(a). From this property one expects, that the more refined algorithms can reduce the number of operations comparatively

nodes	4799875
edges	5947001
$d_{\max}$	7
$d_{\min}$	1
d_{avr}	2.5
d stddeviation	0.90
core number	3
wedges	10752498
triangles	172699
transitivity	0.048
clustering coefficient	0.050



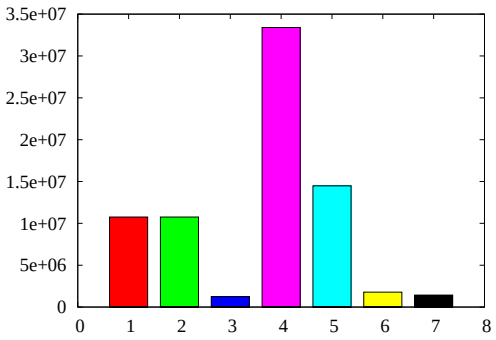
(a) properties and degrees

	algorithm	seconds
1	*node-iterator*	13.73
2	*listing-ayz*	13.95
3	*node-iterator-core*	11.99
4	*edge-iterator*	4.27
5	*edge-iterator-hashed*	55.47
6	*forward*	9.03
7	*forward-hashed*	43.97



(b) execution times

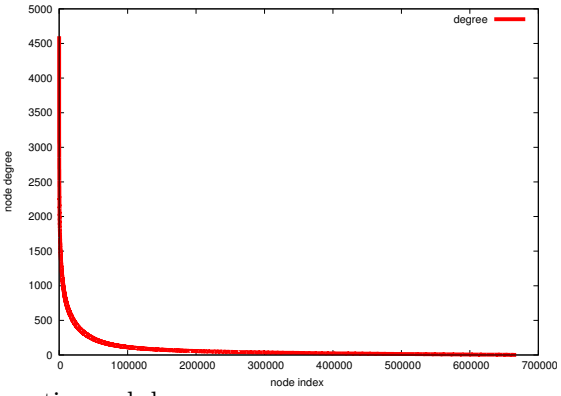
	algorithm	operations
1	*node-iterator*	10752498
2	*listing-ayz*	10752498
3	*node-iterator-core*	1244194
4	*edge-iterator*	33398998
5	*edge-iterator-hashed*	14476855
6	*forward*	1786531
7	*forward-hashed*	1426197



(c) triangle operations

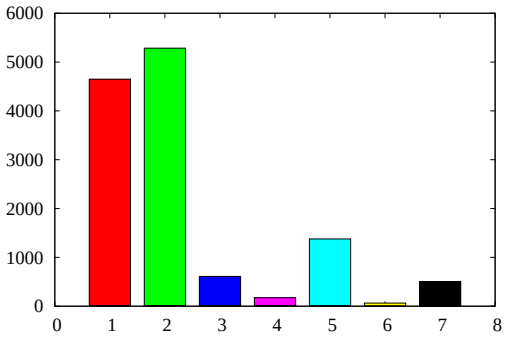
Figure 3.3: *Road Network Germany*

nodes	667609
edges	27581275
$d_{\max}$	4605
$d_{\min}$	1
d_{avr}	82.6
$d_{\text{stddeviation}}$	183.18
core number	1005
wedges	13452269555
triangles	1176613576
transitivity	0.262
clustering coefficient	0.796



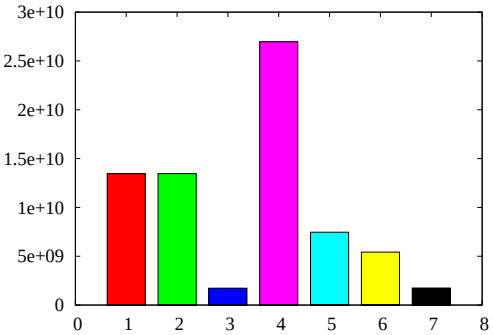
(a) properties and degrees

	algorithm	seconds
1	*node-iterator*	4649.05
2	*listing-ayz*	5285.50
3	*node-iterator-core*	610.61
4	*edge-iterator*	174.04
5	*edge-iterator-hashed*	1377.73
6	*forward*	64.48
7	*forward-hashed*	504.26



(b) execution times

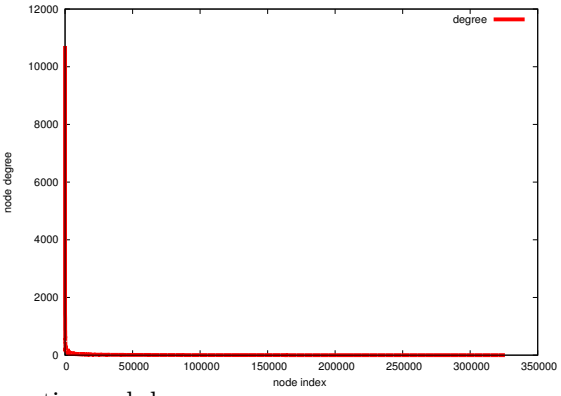
	algorithm	operations
1	*node-iterator*	13452269555
2	*listing-ayz*	13452269555
3	*node-iterator-core*	1725685526
4	*edge-iterator*	26959701660
5	*edge-iterator-hashed*	7460874664
6	*forward*	5423623560
7	*forward-hashed*	1745104092



(c) triangle operations

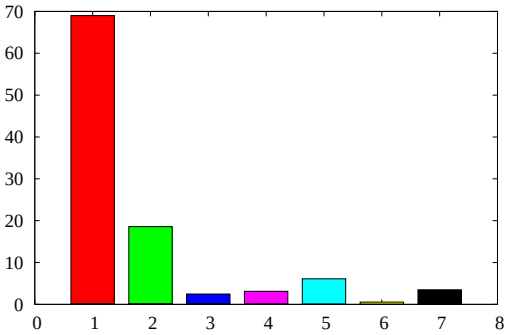
Figure 3.4: *Movie Actor Network 2004*

nodes	325729
edges	1090108
$d_{\max}$	10721
$d_{\min}$	1
d_{avr}	6.7
$d_{\text{stddeviation}}$	42.82
core number	155
wedges	304881174
triangles	8910005
transitivity	0.088
clustering coefficient	0.466



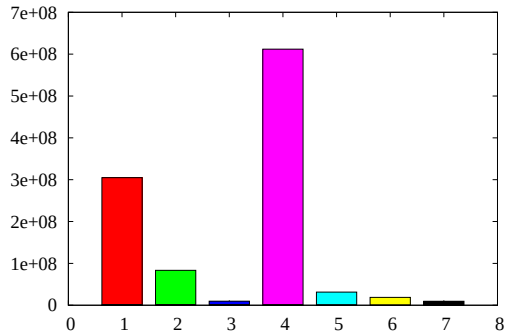
(a) properties and degrees

	algorithm	seconds
1	*node-iterator*	69.00
2	*listing-ayz*	18.58
3	*node-iterator-core*	2.47
4	*edge-iterator*	3.12
5	*edge-iterator-hashed*	6.11
6	*forward*	0.53
7	*forward-hashed*	3.49



(b) execution times

	algorithm	operations
1	*node-iterator*	304881174
2	*listing-ayz*	83374735
3	*node-iterator-core*	9664634
4	*edge-iterator*	611942564
5	*edge-iterator-hashed*	31373326
6	*forward*	18742108
7	*forward-hashed*	9473218



(c) triangle operations

Figure 3.5: *Notre Dame WWW*

well, this is confirmed in Figure 3.5(c). However, algorithm *forward* again performs best in execution time, see Figure 3.5(b).

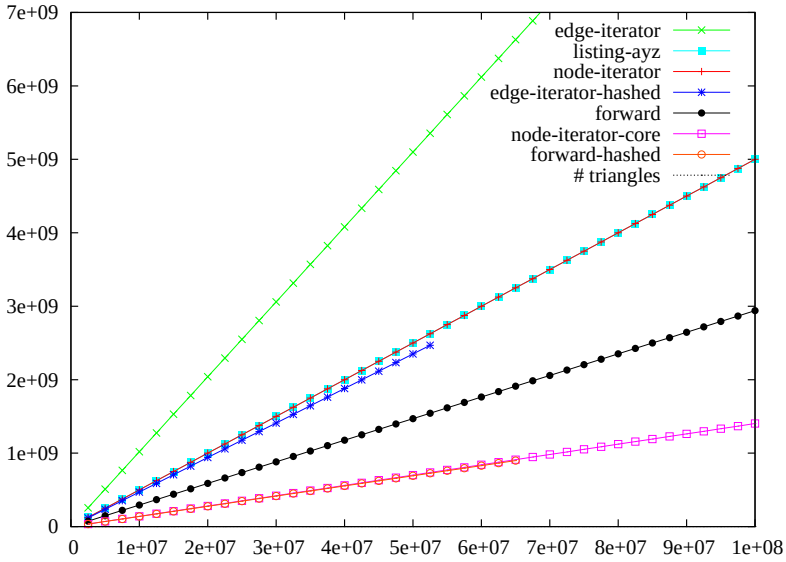
Experiments on Generated Graphs

The networks used in Section 3.2 give an indication on the performance difference of the various algorithms. To get a clearer picture, we investigate how the algorithms perform on a series of random graphs with increasing size. In Section 3.2 we evaluate on standard random graphs. This is then amended in Section 19 with a modified graph model that has a more interesting degree distribution.

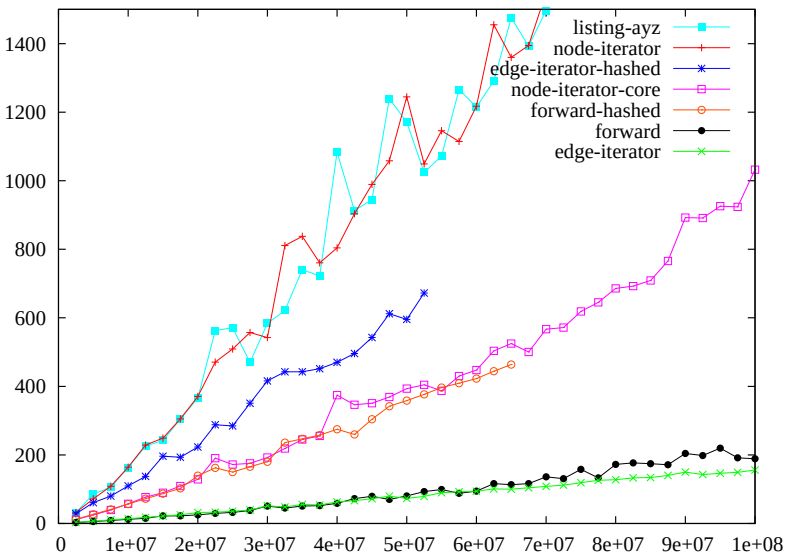
Generated $G_{n,m}$ Graphs Figure 3.6 shows the results on generated graphs where m edges are inserted randomly between n nodes. We do not allow loops or multiple links in our $G_{n,m}$ graphs. Note that the term $G_{n,m}$ graph is usually associated with the model of Erdős and Rényi [Erdős and Rényi, 1959], in which the set of non isomorphic graphs with n nodes and m edges is considered. Our generator is also related to the $G_{n,p}$ model of Gilbert [Gilbert, 1959]. We differ from the latter in generating exactly m edges, which are inserted uniformly at random between the $\binom{n}{2}$ possible endpoints, whereas Gilberts model generates graphs with an expected number of $\binom{n}{2}p$ edges. Nevertheless we stick to the symbol $G_{n,m}$ since n and m are our input parameters.

We consider graphs in which the relation of the parameters n and m is chosen such that the average degree equals 50. Figure 3.6(a) shows the number of triangle operations versus the number of edges. Since $G_{n,m}$ graphs are very unlikely to have high degree nodes, algorithm *listing-ayz* and *node-iterator* perform equally well with respect to the number of triangle operations. The algorithms *node-iterator-core* and *forward-hashed* most efficiently limit the number of triangle operations. They are very close to the optimum, i.e. the number of triangles.

Figure 3.6(b) shows the execution time in seconds versus the number of edges. Again the algorithm *edge-iterator* performs best together with *forward*. Both algorithms are very simple and do not use complicated data structures. In contrast, the use of hashing slows down the execution time of the corresponding algorithms. The plots of the algorithms using hashing show some notable pikes which are caused by automatic rehashing of the container.



(a) triangle operations vs number of edges



(b) execution times in seconds vs number of edges

Figure 3.6: Generated $G_{n,m}$ graphs.

Generated Graphs with High Degree Nodes The $G_{n,m}$ graphs like in Section 3.2 tend to have no high degree nodes and to have a very low deviation from the average degree in general. However, many real world networks show a different behavior, see Section 3.2 and Section 3.2 for an example. The famous power law distributions in the degree found in many networks [Faloutsos et al., 1999] actually hint that skewed degree distributions are very common.

We use a very simple modification of the $G_{n,m}$ model to achieve high degree nodes. As in Section 3.2 we first generate a prime graph $G_{n,m}$ which we extend to a graph $G_{n,m,h}$. For each node $1 \leq i \leq h$ we add links to randomly selected nodes until the degree of the i-th node is $\frac{n}{2}\frac{h-i}{h}$. The parameter h is set to $h = \sqrt{n}$ for the results shown in Figure 3.7.

Now, the number of triangle operations is obviously not linear in the graph size, see Figure 3.7(a). Again the algorithms *node-iterator-core* and *forward-hashed* perform best in limiting the asymptotic effort. However, they are only in second and third place in the execution time. Algorithm *forward* performs clearly best.

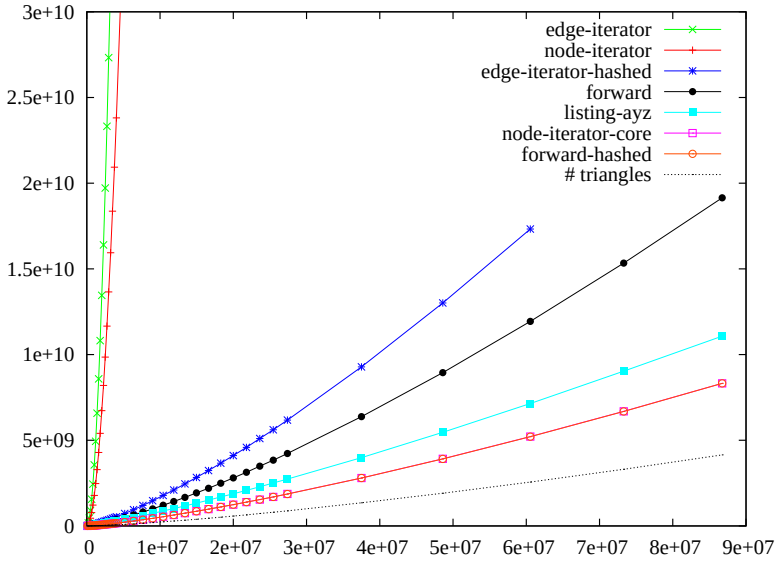
Remarks on the Experiments

Before discussing the results we complement the experiments by comparing Hashing versus Balanced Tree data structures for storing edges, see Section 19. We also consider the statistical variation of the execution times in Section 19.

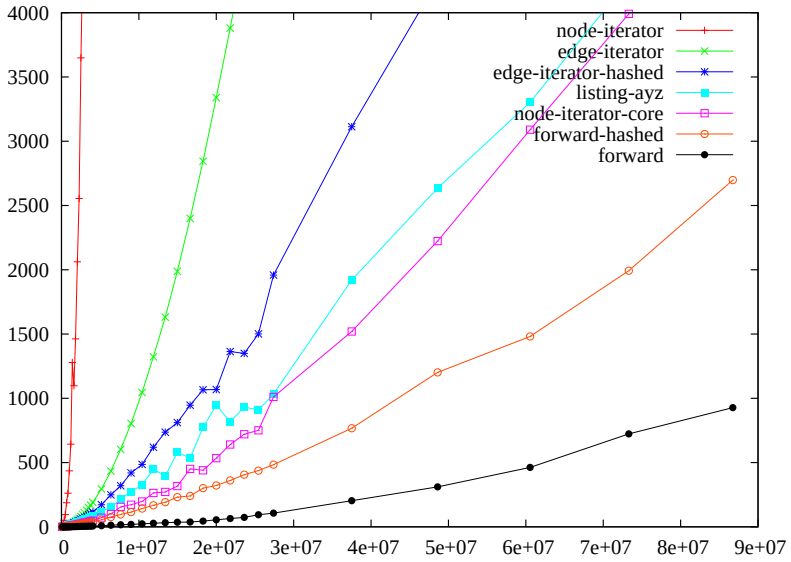
Hashing versus Balanced Trees

We investigate the performance between hashed containers and balanced tree containers for storing the edges. The template from `__gnu_cxx::hash_map` was used for the hashed container. We store two pointers from the incident nodes of an edge. The hashing function combines two random bit fields of size `size_t` (one for each node) with the `XOR` operation. The tree container is based on the template `std::set`, which is an implementation of a red-black tree. An edge is coded with the lexicographic order of the two addresses of the incident node pointers.

The results are shown in the Figure 3.8. We can observe that the hashed data structure outperforms the balanced tree structure. This justifies the chosen hashing function and the use of a hashed container in general for the purpose of testing edge existence.



(a) triangle operations vs number of edges



(b) execution times in seconds vs number of edges

Figure 3.7: Generated graphs with high degree nodes.

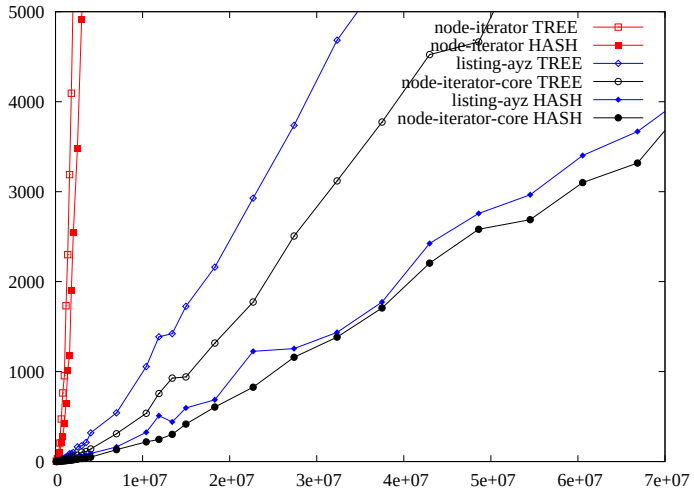


Figure 3.8: Hashing versus balanced trees: Execution time in seconds (y-axis) depending on the number of edges (x-axis).

Statistical Experiments of the Execution Times

The algorithms show some unsteady behavior in the experiments for the execution times. The obvious pikes for the algorithms using a hashed container for the edges are caused by rehashing of the container. But there are other influences like the variations in the structure of the generated graph or other processes running on the same machine. To investigate these influences we generated for each algorithm 100 graphs. The sizes of the graphs are chosen such that the average execution time is not far from 60 seconds.

The results are shown in the Table 3.1. The variation of the execution times behaves well. As expected it is lower when the algorithm runs on the same graph (Table 3.0(a)) compared to test it on 100 different graphs (Table 3.0(b)).

Chapter Conclusion

The two known standard algorithms *node-iterator* and *edge-iterator* are asymptotically equivalent (Lemma 7). However, the algorithm *edge-iterator* can be implemented with much lower overhead (Figure 3.6(b)). It works very well for graphs with degrees that do not differ much from the average degree (Figure 3.3, Figure 3.6).

(a) 100 graphs: one execution for each graph (time/sec)

	min	mean	max	std. deviation
node-iterator	58.83	66.94	83.18	6.15
listing-ayz	59.46	64.79	80.86	4.78
node-iterator-core	58.85	66.36	80.57	5.36
edge-iterator	59.72	60.62	61.18	4.06
edge-iterator-hashed	58.68	60.59	68.13	1.93
forward	56.11	60.08	67.49	3.18
forward-hashed	58.76	62.23	67.75	2.71

(b) 100 executions on one single graph (time/sec)

	min	mean	max	std. deviation
node-iterator	59.48	61.92	76.96	2.80
listing-ayz	62.72	69.15	81.39	5.33
node-iterator-core	58.98	66.12	79.99	5.07
edge-iterator	59.78	60.60	61.22	0.25
edge-iterator-hashed	59.95	66.08	72.88	2.71
forward	57.74	61.15	64.31	1.66
forward-hashed	59.18	64.05	69.23	2.81

Table 3.1: Statistical evaluation of the execution times.

If the degree distribution is skewed, refined algorithms are required. The algorithms *node-iterator-core* and *forward-hashed* are most efficient in reducing the number of triangle operations (Figure 3.4(c), 3.5(c), 3.6(a), 3.7(a)). However, they require advanced data structures which result in high overhead (Figure 3.3(b), 3.4(b), 3.6(b), 3.7(b)). The algorithm *forward* experimentally shows to be a good compromise. Its execution time is close to the one of algorithm *edge-iterator* for networks with low deviation from the average degree (Figure 3.4(b), 3.6(b)). However, it clearly performs best for graphs with notably skewed degree distributions (Figure 3.5(b), 3.7(b)). In general, algorithm *forward* achieves the best execution times. Altogether, we have shown that listing and counting triangles can be performed in reasonable time even for huge graphs.

We have shifted the future focus rather on memory consumption by the given fast algorithm. As a first step in this direction algorithm *forward* has been improved, mainly with respect of lower memory consumption, to algorithm *compact-forward* in [Latapy, 2006]. However, both algorithms require already only linear space and consequently the next step might be in limiting

the usage of central memory. Since triangles can not be listed by streaming algorithms with reasonable models [Bar-Yosseff et al., 2002] further approaches could be based on memory hierarchy methods. A survey on this matter is [Meyer et al., 2003]. Indeed, in the meanwhile an apdoption of the algorithm *forward* has been implemented into the secondary memory library `STXXL` [Dementiev et al., 2005].

Chapter 4

Algorithms for Counting and Listing Triangles in Special Graph Classes

Chapter Introduction

In this chapter we will give efficient algorithms for listing and counting triangles for certain graph classes. We strive to achieve running time in $\mathcal{O}(m + \delta)$, where δ is the number of triangles in the graph for listing triangles. In the case of counting triangles for each node we aim to achieve running time in $\mathcal{O}(m)$. One could regard such running times as optimal with respect to the parameters m respectively δ.

However, it is possible to beat such optimal running times, for example if we know that we have a complete graph. While this case certainly is pathological and uninteresting, we will give an example in which we achieve $\mathcal{O}(n)$ running time for counting by using a certain encoding of the graph. Note that we purposely do not handle properties relying on solution methods that involve high constants in any implementation of the algorithms. We have seen that there are very efficient general algorithms in Chapter 3. This renders algorithms with high constants disadvantageous to use.

Contribution and Related Work. As far as we know efficient triangle listing algorithms have not been considered for special graph classes. There exist work for triangle counting that considers other parameters such as degeneracy or arboricity, see [Chiba and Nishizeki, 1985; Alon et al., 1997] for

example. For some parts triangle counting has been considered previously, albeit by other methods. In these cases we will give a reference to the relevant work.

Organization. We consider graphs with bounded core number in Section 4.1. Planar graphs and graphs constructed by preferential attachment belong to this class. Section 4.2 treats comparability graphs and Section 4.3 chordal graphs. In Section 4.4 distance hereditary graphs are discussed.

Before we start let us agree on the following. We will use a sentence like "Algorithm xyz lists all triangles." in the meaning that algorithm xyz lists all triangles, each triangle is listed exactly once, and nothing else but triangles are listed by the algorithm. This convention makes many upcoming statements compact and easy to read.

4.1 Graphs with Bounded Core Numbers

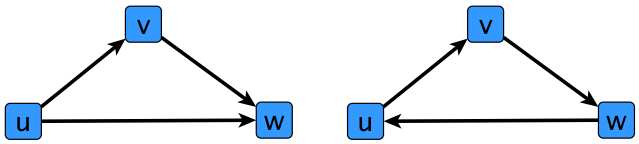
We introduced the core concept in Section 2.4 on page 18. And in Section 3.1.2 we have seen that there exists a listing algorithm with running time $\mathcal{O}(\kappa^2 \cdot n)$ where κ is the core number of the graph. See *node-iterator-core* listed in Algorithm 3.4 on page 29. If κ is bounded by a constant we get a running time in $\mathcal{O}(m)$ and we list a few graph classes which benefit from a bounded core number in the following.

Planar Graphs

A *planar graph* is characterized by the property that it can be embedded in the plane such that no two edges intersect except in the incident node. A linear time algorithm for finding a triangle in a planar graph was given by Itai and Rodeh [1978], see also page 24 in Section 3.1.1. It can be extended to counting and listing all triangles without additional costs with respect to asymptotic running time.

Theorem 3 *All triangles of a planar graph can be listed in linear time.*

Our approach is slightly simpler than the algorithm in [Itai and Rodeh, 1978]. We use the fact that each planar graph has at least one node with degree five or less. Clearly, a planar graph remains planar when parts of it are removed. Hence, the core number of any planar graph is also 5 or less.



(a) transitive triangle (b) cyclic triangle

Figure 4.1: Directed triangle orientations.

Preferential Attachment Graphs

The so called *preferential attachment* graphs are generated graphs where in every iteration a node v and μ new edges, each connecting v to some existing node, are added. See Section 6.1 on page 95 for more details. We point out that the last added node v has degree μ and consequently the core number is bounded by the parameter μ.

Theorem 4 *All triangles of generated preferential attachment graphs can be listed in linear time for fixed parameters μ.*

4.2 Comparability Graphs

An undirected graph that has an acyclic transitive orientation is called a *comparability* graph. A digraph $D = (V, A)$ is transitive oriented if (u, v), $(v, w) \in A$ implies the existence of $(u, w) \in A$, see Figure 4.1(a). We will show Theorem 5 in this section.

Theorem 5 *All triangles of a comparability graph can be listed in $\mathcal{O}(m + \delta)$ time and counted in $\mathcal{O}(m)$ time.*

Golumbic [1977] gave an algorithm to compute a transitive orientation of a comparability graph in time in $\mathcal{O}(d_{\max} \cdot m)$. The first linear time algorithm was given much later by McConnell and Spinrad [1997].

A Triangle Listing Algorithm for Comparability Graphs

The required existence of the arc (u, w) in Figure 4.1(a) leads to a simple algorithm for listing triangles. For every node v one simply outputs all pairs consisting of one adjacent incoming with one adjacent outgoing node. Here,

incoming respectively outgoing refers to the edge by which the node is connected to v. This is idea is realized in Algorithm 4.1.

Algorithm 4.1: TRIANGLE LISTER FOR COMPARABILITY GRAPHS

Input: comparability graph $D = (V, A)$ in transitive orientation
Output: all triangles

```
1 forall v ∈ V do
2     forall u ∈ {u ∈ V : ∃(u, v) ∈ A} do
3         forall w ∈ {w ∈ V : ∃(v, w) ∈ A} do
              output triangle {u, v, w};
```

Lemma 6 *Algorithm 4.1 lists all triangles of comparability graph given in transitive orientation. It can be implemented to run in $\mathcal{O}(n + \delta)$ time.*

Proof: We start with the correctness. Every triangle has to be in transitive orientation. Clearly each transitive triangle has a center node v, as in Figure 4.1(a). Hence, each triangle is listed exactly once by the algorithm. Clearly, each output $\{u, v, w\}$ is guaranteed to be a triangle.

Let us consider the time complexity. If we consider the loop starting at line 1 to be empty we iterate in n steps over all nodes. We start the loop of line 2 conditional only if the outgoing links from v are not empty. Thus, we guarantee that every iteration produces an output and both loops are in $\mathcal{O}(\delta)$. Note that we still get running time in $\mathcal{O}(m + \delta)$ if we do not perform this test. □

A Triangle Counting Algorithm for Transitively Oriented Graphs

Algorithm 4.1 lists the counting algorithm for comparability graphs in transitive orientation.

Lemma 7 *Algorithm 4.1 counts all triangles of a transitive orientation for each node. It can be implemented to run in $\mathcal{O}(m)$ time.*

Proof: Note that the number of outgoing and incoming edges (line 5 and line 3) of a node sum up to the degree. Then the running time is due to the Handshake Lemma.

Algorithm 4.2: TRIANGLE COUNTER FOR COMPARABILITY GRAPHS

Input: comparability graph $D = (V, A)$ in transitive orientation
Output: number of triangles $\delta(v)$ for each node v

1 **forall** $x \in V$ **do**
2 $\quad \delta(x) \leftarrow d_{\text{in}}(x) \cdot d_{\text{out}}(x)$;
3 $\quad$ **forall** $v \in \{v \in V : \exists (x, v) \in A\}$ **do**
4 $\quad\quad \delta(x) \leftarrow \delta(x) + d_{\text{out}}(v)$;
5 $\quad$ **forall** $v \in \{v \in V : \exists (v, x) \in A\}$ **do**
6 $\quad\quad \delta(x) \leftarrow \delta(x) + d_{\text{in}}(v)$;

We distinguish between three types of triangles for each node x. There are exactly $d_{in}(x) \cdot d_{out}(x)$ triangles for which x is the center node ($x = v$ in Figure 4.1(a)). They are counted in line 2. The node x is the "first" node ($x = u$ in Figure 4.1(a)) of the triangles with the pair $(v, w) \in V \times V$ for which $(x, v), (v, w) \in A$. There are exactly $\sum_{v:(x,v)\in A} d_{\text{out}}(v)$ of those and they are counted in the loop starting at line 3. The node x is the "last" node ($x = w$ in Figure 4.1(a)) of the triangles with the pair $(u, v) \in V \times V$ for which $(u, v), (v, x) \in A$. There are exactly $\sum_{v:(v,x)\in A} d_{\text{in}}(v)$ of those and they are counted in the loop starting at line 5.

□

4.3 Chordal Graphs

A graph is *chordal* if every simple cycle of length at least four has a chord, i.e. an edge between two nodes that do not induce an edge in the cycle. Equivalently, a chordal graph does not have an induced k-cycle for k larger than three. A *perfect elimination order* (PEO) is an order of the nodes $v_1, \ldots, v_n$ such that all neighbors of v_i in $G[\{v_j : j \leq i\}]$ induce a clique. A graph is chordal if and only if it has a perfect elimination order [Fulkerson and Gross, 1965]. Testing chordality and moreover computing a perfect elimination order can be done in linear time [Rose et al., 1976].

This is done with so called *lexicographic breadth first search* (lexBFS) a variant of BFS which ensures that unvisited neighbors of a currently visited node are discovered before non-neighbors. A survey of lexBFS and related elimination orders can be found in [Corneil, 2004].

Now, we have collected all ingredients to efficiently perform triangle computation in chordal graphs and in the end show the following Theorem:

Theorem 6 *All triangles of a chordal graph can be listed in $\mathcal{O}(m+\delta)$ time and counted in $\mathcal{O}(m)$ time.*

To do so, we first compute a PEO and then iterate over the nodes in the inverse order. In each iteration we determine all triangles of the current node with those nodes of lower PEO number. How this is precisely done depends on whether we want to count, see Algorithm 4.4, or list all triangles, see Algorithm 4.3.

Algorithm 4.3: TRIANGLE LISTER FOR CHORDAL GRAPHS

Input: chordal graph G; PEO $(v_1, \ldots, v_n)$;
Output: all triangles

```
1 for v_k = (v_n, ..., v_1) do
2     forall v_i ∈ Γ(v_k) do
3         forall v_j ∈ Γ(v_k), i < j do
4             output triangle {v_i, v_j, v_k};
5     remove v_k from G;
```

Lemma 8 *Algorithm 4.3 lists all triangles of a chordal graph. It can be implemented, such that it runs in $\mathcal{O}(m+\delta)$ time.*

Proof: We start with the correctness of the algorithm. In line 4 only triangles are listed, since all neighbors of v_k with lower PEO number than k induce a clique together with v_k. Note that $i < k$ (i.e. all neighbors of v_k have lower PEO number than v_k) is always true because of line 5.

A triangle is listed at most once since any combination of i, j, k can appear at most once. This is due to the removal of node v_k after processing it in line 5 and the requirement $i < j$ in line 3.

It remains to be shown that all triangles are listed. Let $G[\{u, v, w\}]$ be a triangle and without loss of generality assume $u < v < w$ with respect of the given PEO. Since k iterates over all vertices there is a value of k such that $v_k = w$. Note that at this time u and v are still present in the graph because we iterate in invers PEO in line 1. Then there must exist some value of i such that $v_i = u$ and some value of j such that $v_j = v$. Therefore the triangle $\{u, v, w\}$ is listed.

We turn to the time complexity. A crucial point is to execute line 2 and line 3 efficiently, i.e. incrementing i and j in constant time. This can be achieved if the neighborhoods are also sorted according to the PEO. Sorting all neighborhoods according to a given order of nodes can be done in linear time, e.g. by proceeding in a similar manner as in Algorithm 3.6 on page 32.

By the proof of correctness above line 4 is executed exactly δ times. However, the Loop in line 3 might be empty. Therefore, it remains to be shown that all loops including those of line 3 are in linear time. This is nothing else but an iteration over all edges. □

Algorithm 4.4: TRIANGLE COUNTER FOR CHORDAL GRAPHS

Input: chordal graph G; PEO $(v_1, \ldots, v_n)$;
Output: all triangles
for $i = (1, \ldots, n)$ **do**
  $\delta(v_i) \leftarrow 0$;
1 **for** $v_k = (v_n, \ldots, v_1)$ **do**
2 $\delta(v_k) \leftarrow \delta(v_k) + \binom{d(v_k)}{2}$;
3 **forall** $v_i \in \Gamma(v_k)$ **do**
4 $\delta(v_i) \leftarrow \delta(v_i) + d(v_k) - 1$;
5 remove v_k from G;

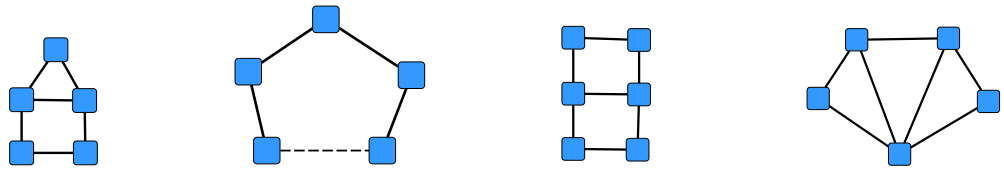
Algorithm 4.4 is merely an adoption of Algorithm 4.3 for counting.

Lemma 9 *Algorithm 4.4 counts all triangles of a chordal graph. It can be implemented, such that it runs in $\mathcal{O}(m)$ time.*

Proof: We start with the time complexity. Analogous as in the previous proof all loops together can be implemented to run in the order of the number of edges. Nothing else remains to be shown here.

We show in the following that in every iteration of the loop beginning at line 1 all currently existing triangles containing the node v_k (and no more) are counted.

Only neighbors of v_k with lower PEO are present in the current loop k. There are $|\Gamma(v_k)| = d(v_k)$ such neighbors with which v_k induces a clique and hence there are $\binom{d(v_k)}{2}$ triangles in this clique containing node v_k. Consequently in line 2 all triangles of node v_k with nodes of lower PEO number than k are counted for node v_k.



(a) house (b) hole (c) domino (d) gem

Figure 4.2: Forbidden induced subgraphs.

It remains to be shown that the triangles are counted for the nodes with PEO number less than k. In the loop starting at line 3 the triangle number for these nodes is updated. There are again $d(v_k)$ of them. Each forms triangles with v_k and the rest of the $d(v_k) - 1$ nodes in the induced clique. Consequently the number $d(v_k) - 1$ has to be added, which is done in line 4.

Arriving at line 5 all triangles containig node v_k have been counted and v_k can be savely removed. □

4.4 Distance Hereditary Graphs

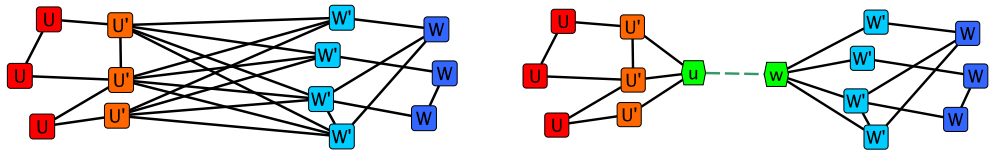
The distance hereditary graphs will be the last graph class we consider in this chapter.

Introduction and Preliminaries

A connected graph G is *distance hereditary* if for any path P of G the induced subgraph $G[\mathcal{V}(P)]$ is distance preserving (*isometric*). That means that any two nodes of P have the same distance in G as in $G[\mathcal{V}(P)]$. We will use the abbreviation DH for distance hereditary in the following.

There exists an equivalent definition by forbidden subgraphs [Hammer and Maffray, 1990]: a graph is DH if and only if it does not have an induced subgraph of the type: house, hole (induced cycle of length at least five), domino or gem. See Figure 4.2. The abbreviation HHDG-free is frequently used in the literature.

Distance hereditary graphs have an equivalent description, like chordal graphs, based on an elimination order. A graph is distance hereditary if and only if it has a *2-simplicial elimination order* [Nicolai, 1996]. See for example the



(a) split (b) split decomposition

Figure 4.3: A split and its decomposition.

survey [Corneil, 2004] for the definition and computation of such an order. We used the perfect elimination order to efficiently list and count triangles for the case of chordal graphs in Section 4.3. However, here we will use a further equivalent formulation for DH graphs based on splits.

We have listed the important and more common ways to describe DH graphs, but there are more. The book "*Graph classes, a survey*" [Brandstädt et al., 1999] is a good starting point for the interested reader.

Splits

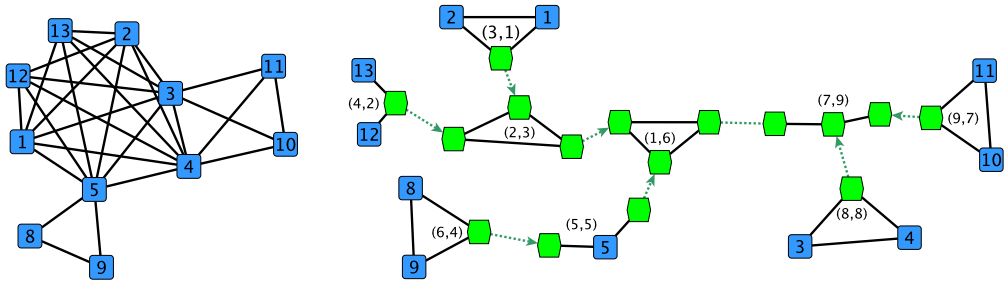
A *split* [Cunningham, 1982] in a graph $G = (V, E)$ is a partition of the set V into two sets U and W, such that $|U| > 1$ as well as $|W| > 1$, and such that for all $u', w' : u' \in U' = U \cap \Gamma(W), w' \in W' = W \cap \Gamma(U) \Rightarrow \{u', w'\} \in E$. See Figure 4.3(a) for an example.

Given a split $S = (U, W)$ the graph can be split decomposed by

1. adding the split nodes u and w,
2. adding the split edge $\{u, w\}$,
3. adding the edges $\{\{u', u\} : u' \in U'\}$ and $\{\{w, w'\} : w' \in W'\}$ respectively, and
4. removing all edges $\{\{u', w'\} : u' \in U', w' \in W'\}$.

See Figure 4.3 for an example. We will call the edges which are not created in Step 2 the *regular* edges.

Without the split edge $\{u, w\}$ the remaining graph falls into two connected components which are called the split components. This procedure can be applied recursively until no further suitable split is found. If the result consists of components of the size three the graph is called totally split decomposable,



(a) original graph (b) totally decomposed graph

Figure 4.4: A graph and its total decomposition.

see Figure 4.4. As the split edges always form bridges between the components the resulting graph is a tree of components which is called the *split decomposition* tree.

A graph is totally decomposable if and only if it is distance hereditary [Hammer and Maffray, 1990]. The total decomposition can then be computed in linear time [Hammer and Maffray, 1990]. A later result also gives a linear time decomposition algorithm for the general case [Dahlhaus, 1994].

Let us have a closer look to adjacent nodes in Figure 4.4(a) and their connection in the decomposition in Figure 4.4(b). We can see that all of them are connected by an alternating path of regular and split edges. This observation holds generally and we emphasize it in Lemma 10 since it will become very useful in later sections of this chapter.

Lemma 10 *Two nodes are adjacent in G if and only if they are connected by a path of alternating regular and split edges in the decomposition tree T.*

Both directions can be easily seen by induction over the splits. Note, that such a path always starts and ends with a regular edge.

There are only 9 variants of components in a total decomposition possible, see Figure 4.5 on page 71. As the total decomposition is a tree of components we divide the components into *leaf components* (Figure 4.5(a) to Figure 4.5(c)), and *internal components*(Figure 4.5(d) to Figure 4.5(h)). Component COMP-0 (Figure 4.5(i)) is listed for completion. From now on, we will exclude graphs of size less than 4 from the discussion.

The Size of the Total Decomposition Tree

The algorithms for counting or listing triangles, which we will give later in this section, take the decomposition tree of a DH graph as an input. We already mentioned that it can be efficiently computed in linear time. However, it is even more interesting if DH graphs are stored implicitly as a total decomposition.

Lemma 11 *Let T be an arbitrary total decomposition tree of a DH graph G. The size of T is linear in the number of the nodes of G.*

Proof: Let T denote the total decomposition tree of a DH graph G. Let l_1 be the number of leaf components, l_2 the number of internal components with exactly 2 split nodes, and l_3 the number of internal components with 3 split nodes. Then $|T| \in \Theta(l_1 + l_2 + l_3)$ clearly holds since T is a tree. We show that $l_1 + l_2 + l_3 \in \mathcal{O}(|V|)$ in the following. Since any of the leaf components includes two nodes of G we have

$$l_1 \in \mathcal{O}(|V|)\,. \tag{4.1}$$

Likewise $l_2 \in \mathcal{O}(|V|)$ is true, because each such component includes one node of G. It remains to be shown that $l_3 \in \mathcal{O}(|V|)$.

Let T' be the tree in which all of the internal components with exactly 2 split nodes have been replaced by a split edge. All the leaf components and internal components with 3 split nodes remain. Since T' is a connected tree it follows by induction that $l_1 = 2 + l_3$. Now, by Equation 4.1 $l_3 \in \mathcal{O}(|V|)$ holds, too. □

The Listing Algorithm

We have collected all ingredients to formulate an algorithm and prove the following theorem by the end of this chapter.

Theorem 7 *All triangles of a distance hereditary graph can be listed in linear time.*

The algorithm we will develop in this section takes a total decomposition tree T as an input. The more or less straight forward solution would be to reconstruct the original graph G by iteratively rebuilding the splits. Let

us again consider Figure 4.3 on page 57. If we store the "inter split" edges between W'-nodes (as well those between U'-nodes) we can easily output all the triangles that occur across this split: each node in U' forms a triangle with each inter W' edge (and also the other way round). No other triangles are involved in this split.

We use the basic idea just mentioned. However, we proceed in a slightly more complicated fashion. The reason is that we can then reuse some concepts in the next section, where we will discuss efficient triangle counting.

Instead of simply merging some random split we proceed from outside to inside, towards a randomly chosen split edge which we call the *root*. We thereby use an order that ensures that the components lying to the outside of the currently processed split have already been processed. Such an order is the well known postorder tree traversal for example. Note that during the traversal we do not rebuild G from its decomposition representation. We only carry along and modify interesting sets of nodes and edges that are potential candidates for triangles.

Let us go into details, Algorithm 4.5 on page 61 contains the core steps. Handling of the internal components is separated into Function 4.6 on page 62. In the first part of Algorithm 4.5 starting at line 1 the leaf components are initialized. For each leaf component we create two sets: one containing edges and the other containing nodes. These sets are associated with the split node of the component. The part starting at line 3 handles the processing of the internal components. Here, the associated data of already processed neighboring components are evaluated (e.g. triangles lying across the affected split are listed), updated and passed on. The chosen postorder in line 3 guarantees that components lying outside of a current component have already been processed. In Figure 4.4 on page 58 the direction of the split edges implies which components are outside with respect to the undirected root edge. The tuple of integers listed with each component is one valid pre- and postorder with respect to the root.

The actual handling of each internal component is listed in Function 4.6 on page 62. Note that distinguishing between components does not suffice. Let us consider for example the component COMP-IIc:

- If the component is traversed as in Figure 4.6(c), by Lemma 10 the node x does not share an edge with any of the nodes of the set associated with u'. This case is handled in line 1 and following.
- On the other hand, if the component is traversed as in Figure 4.6(d), again by Lemma 10 the node x forms triangles with all the edges in

Algorithm 4.5: TRIANGLE LISTER FOR DH GRAPHS

Input: total decomposition of a DH graph G
Output: all triangles of G

```
1 for comp ∈ LeafComponents do
      switch comp do
          case comp-Ia
              V(w) ← {x, y}, E(w) ← {{x, y}}
          case comp-Ib
              V(w) ← {x, y}, E(w) ← ∅
          case comp-Ic
              V(w) ← {y}, E(w) ← ∅
2 RootEdge: {u, w} ← arbitrary split edge between two internal
  components;
3 forall comp = (int. components in postorder with resp. RootEdge) do
      OutIn(comp)                          see Function 4.6 on page 62
4 for x ∈ V(u) do
      for {y, z} ∈ E(w) do
          output triangle {x, y, z};
5 for x ∈ V(w) do
      for {y, z} ∈ E(u) do
          output triangle {x, y, z};
```

Function 4.6: TRIANGLE LISTER FOR DH GRAPHS: OUTIN

case COMP-II*a* *Figure 4.6(a)*
- **for** $\{y, z\} \in E(u')$ **do**
  - output triangle $\{x, y, z\}$;
- $V(w) \leftarrow \{x\}$, $E(w) \leftarrow \emptyset$;

case COMP-II*b* *Figure 4.6(b)*
- **for** $\{y, z\} \in E(u')$ **do**
  - output triangle $\{x, y, z\}$;
- $E(w) \leftarrow E(u')$;
- **for** $y \in V(u')$ **do**
  - $E(w) \leftarrow E(w) \cup \{\{x, y\}\}$;
- $V(w) \leftarrow V(u') \cup \{x\}$;

case COMP-II*c case 1* *Figure 4.6(c)*
- $E(w) \leftarrow E(u')$, $V(w) \leftarrow V(u') \cup \{x\}$;

case COMP-II*c case 2* *Figure 4.6(d)*
- **for** $\{y, z\} \in E(u')$ **do**
  - output triangle $\{x, y, z\}$;
- $E(w) \leftarrow E(u')$, $V(w) \leftarrow V(u')$;

case COMP-III*a* *Figure 4.6(e)*
- $E(w) \leftarrow E(u') \cup E(v')$;
- **for** $x \in V(u')$ **do**
  - **for** $\{y, z\} \in E(v')$ **do**
    - output triangle $\{x, y, z\}$;
- **for** $x \in V(v')$ **do**
  - **for** $\{y, z\} \in E(u')$ **do**
    - output triangle $\{x, y, z\}$;
- **for** $x \in V(v')$ **do**
  - **for** $y \in V(u')$ **do**
    - $E(w) \leftarrow E(w) \cup \{\{x, y\}\}$;
- $V(w) \leftarrow V(u') \cup V(v')$;

case COMP-III*b case 1* *Figure 4.6(f)*
- **for** $x \in V(u')$ **do**
  - **for** $\{y, z\} \in E(v')$ **do**
    - output triangle $\{x, y, z\}$;
- **for** $x \in V(v')$ **do**
  - **for** $\{y, z\} \in E(u')$ **do**
    - output triangle $\{x, y, z\}$;
- $E(w) \leftarrow E(v'), V(w) \leftarrow V(v')$;

case COMP-III*b case 2* *Figure 4.6(g)*
- $V(w) \leftarrow V(u') \cup V(v')$, $E(w) \leftarrow E(u') \cup E(v')$

the set associated with node u' but not with any nodes in the data sets associated with w. This case is handled in line 2 and the following.

The component COMP-IIIb has to be treated in a similar way. The remaining components are symmetric with respect to the traversal.

We come to the third and final part of Algorithm 4.5. After all internal components have been treated the root edge $\{u, v\}$ has associated data to both nodes. By Lemma 10 the nodes in the node set attached to u form triangles with all the edges of the attached set to w and vice versa. The triangles are computed in the same fashion as for component COMP-IIIa (see line 5 of Function 4.6).

Lemma 12 *Algorithm 4.5 lists exactly the triangles of a DH graph.*

Proof: We prove by induction over the processing of the components in the same order as Algorithm 4.5 proceeds.

To do so we need some definitions first. Let us call the split node of a component which is closest to the root edge the *inward* split node. Each component has such a node, one and only one. We will denote this node for the currently regarded component with w throughout this proof. Further we use w' for the node connected with a split edge $\{w, w'\}$ to w. The remaining split nodes of a component are denoted with u and v, if they exist. This notation is consistent with Figure 4.6 on page 72.

For a decomposition tree T and some split edge $\{u, w\}$ we assume that $B_{\{u,w\}}$ would be a decomposition of G with only one split, namely the one induced by $\{u, w\}$. Further let B_u be the connected component of $B_{\{u,w\}} \setminus \{u, w\}$ that includes u. Let us regard Figure 4.3 on page 57 for an example, where B_u corresponds to the induced graph of the nodes $U \cup U' \cup \{u\}$. Note also that our algorithm never rebuilds any B_v actually. It is a merely theoretical concept used only within this proof.

We will show that the following induction hypotheses hold after processing the current component with the inward split node w:

1. All the triangles induced by $\mathcal{V}(B_w)$ in G have been listed.
2. The set $E(w)$ contains exactly all the edges of $\mathcal{E}(G)$ that are part of a triangle with the node w in B_w: $E(w) = \{\{x, y\} \in \mathcal{E}(G) : \{x, w\}, \{y, w\}, \{x, y\} \in \mathcal{E}(B_w)\}$.

3. The set $V(w)$ contains exactly all the nodes of $\mathcal{V}(G)$ that are adjacent to w in B_w: $V(w) = \{x \in \mathcal{V}(G) : \{x, w\} \in \mathcal{E}(B_w)\}$.

Note that only invariant 1 is relevant to finally show the Lemma, the remaining ones are only helper statements. The inductive base cases consist on verifying the properties listed above for the components handled before line 2 is reached. These are all the leaf components and the correctness for those is easy to see.

In the following we show that the statements hold when processing component "COMP-IIc case 2" as in Figure 4.6(d). This case is handled by line 2 and following in Function 4.6. By Lemma 10 the node x is connected to all nodes in $V(u')$ and hence forms triangles with all the edges in $E(u')$. All these triangles (and no others) are listed in the Loop beginning at line 3. According to Lemma 10 the node x is not connected to any of the nodes of $\mathcal{V}(B_w) \cap \mathcal{V}(G)$ in G. Hence, it must not be included in $V(w)$, and consequently does not have an endpoint in $E(w)$. The node u is incident to all nodes listed in $V(u')$, hence again by Lemma 10 we can simply assign the set $V(u')$ and $E(u')$ to the node w, which is done in line 4. This shows that invariant 2 and 3 hold from which the correctness of invariant 1 follows.

Showing the invariants for the remaining 6 cases of Function 4.6 is quite similar to what we have just seen. We omit the details here. We continue with the final part, processing the root edge $\{w, u\}$. By the inductive hypothesis all triangles induced by $\mathcal{V}(B_w)$ and $\mathcal{V}(B_u)$ in G have been listed. By the invariants each edge in the set $E(u)$ forms triangles with each node in $V(w)$ and the other way round. Also by the invariants and Lemma 10 no further triangle exists that contains both nodes from $\mathcal{V}(B_w)$ and $\mathcal{V}(B_u)$. All these triangles and no further are listed in line 4 respectively line 5 of Algorithm 4.5. □

Lemma 13 *Algorithm 4.5 can be implemented to run in $\mathcal{O}(m + \delta)$ time and $\mathcal{O}(m)$ space.*

Proof: We consider linked lists as data structures to store the edges and nodes sets. They can be merged in constant time when the formation of unions is required. The linear space requirement now follows from the fact that each edge respectively node is contained in at most one of the sets.

For the time complexity we note that finding and initializing all the leaf components can be done in linear time. Computing a postorder of the

internal components can also be done in linear time. We turn to the internal components handled in Function 4.6. We first disregard all the for-loops, in this case only set merging operations remain. Due to the linked lists such an operation can be done in constant time and all merging operations together are in the size of the number of components. Now, we disregard all the loops containing triangle output. The remaining loops create edges and add them to the sets or they add nodes to the sets. As already mentioned with the space complexity each edge and each node is listed only once. Hence, all these operations are in $\mathcal{O}(m)$.

Finally we include the for-loops with triangle output in our consideration. With exception of component COMP-IIIb every iteration of the loops also outputs a triangle. For component COMP-IIIb some empty iterations of the outer loops iterating over the nodes could occur if the corresponding edge sets are empty. However, this can be avoided if the edge sets are previously tested whether they are empty. In that case only one operation is required in the other case every operation also outputs a triangle. Therefore, the number of all operations is in $\mathcal{O}(m + \delta)$ time. □

The Counting Algorithm

We will show Theorem 8 by the end of this section. Note, that the $\mathcal{O}(n)$ bound requires the graph to be encoded in total decomposition. Otherwise a decomposition preprocessing step in $\mathcal{O}(m)$ time and space is required.

Theorem 8 *All triangles of a distance hereditary graph can be counted in* $\mathcal{O}(n)$ *time and space.*

The main code is listed in Algorithm 4.7. As promised we reuse some concepts that we developed with the listing algorithm of Section 26. Due to this we can keep this introduction brief and focus on the main differences between the two algorithms. Instead of storing sets of nodes and edges we only store the size of those. We call the associated variables $n(v)$ for the size of the node set, respectively $m(v)$ for the size of the edge set. In the listing case we proceeded from outside to inside and listed the triangles along this traversal. Here we traverse from outside to inside, too. See line 2 of Algorithm 4.7. However, we then also traverse from inside to outside (line 2) where we take along the data to the nodes. The actual output is then computed in a final step in line 4.

Now, we proceed with actually showing Theorem 8 in several parts. These parts are self contained but generally rather compact compared to those of the previous chapter.

Algorithm 4.7: TRIANGLE COUNTER FOR DH GRAPHS

Input: total decomposition of a DH Graph
Output: number of triangles $\delta(v)$ for each node v

1 *RootEdge* $\leftarrow$ arbitrary split edge between two internal components;
2 **forall** *comp* = (*components in postorder with resp. RootEdge*) **do**
  `OutIn(`*comp*`)` see Function 4.8 on page 69
3 **forall** *comp* = (*components in preorder with resp. RootEdge*) **do**
  `InOut(`*comp*`)` see Function 4.9 on page 70
4 **forall** *comp* $\in$ *components* **do**
  `Calc-`δ (*comp*) see Function 4.10 on page 71

Lemma 14 *Algorithm 4.7 can be implemented to run in* $\mathcal{O}(n)$ *time and space.*

Proof: We consider the time complexity first. Note that the postorder and the preorder can be computed in time linear in the size of a tree. Processing each component in the Functions 4.8, 4.9 and 4.10 can be done in constant time. By Lemma 11 there are only $\mathcal{O}(n)$ components. Each component is processed at most once in each of the three loops starting at line 2, 3, and 4. Hence, the total running time is in $\mathcal{O}(n)$.

Now, let us consider the space complexity. We associate at most 2 integers with each node. There are no more than 3 nodes in any component and by Lemma 11 the number of components is in $\mathcal{O}(n)$. □

Lemma 15 *Let* T *be the total decomposition tree of* G*. After the execution of line 3 for each split node* u *the following holds:*

- $n(u)$ *is equal to the number of regular nodes that are connected to* u *in* T *by an alternating path,*
- $m(u)$ *is equal to the number of pairs that consist of regular nodes, which are themselves connected by an alternating path, and also connected by an alternating path to* u*.*

Proof: Let T be the total split decomposition tree of a DH graph G with a designated root edge $\{u, w\}$. Let us call the split node of a component which is closest to the root edge the *inward* split node. Each component has exactly one such node. In Figure 4.4(b) on page 58 the inward split nodes are those split nodes which are not pointed by an arrowhead. The remaining split nodes are called the *outward* split nodes. Note that the in- and outward split nodes define a partition on the set of the split nodes. Further, for some split edge $\{w, w'\}$ let T_w be the connected subtree of $T \setminus \{w, w'\}$ that includes w.

We first prove the statement for all inward split nodes. We do so by induction over processing the components in the same (post)order as used in line 2 of Algorithm 4.7. The induction basis is covered by the processing of the leaf components* handled in the Lines 1, 2, and 3 in Function 4.8. It is straight forward to see that the statement holds after processing the leaf components by Function 4.8.

We proceed to the inductive step where we consider exemplarily the processing of component "COMP-IIc case 1" starting at line 4 in Function 4.8. By induction hypothesis and by using postorder processing the associated $n(u')$ and $m(u')$ data for node u' has been computed such that the statements hold. The node w is connected by an alternating path ending with a split edge to u'. All the alternating paths from u' to regular nodes in $T_{u'}$ start with non split edges. Hence, w is connected by an alternating path to all those nodes, too. Further w is connected by an alternating path to x. The number of regular nodes connected to w by an alternating path in T_w is consequently given by $n(u') + 1$. In T_w the node x is not connected by an alternating path to any regular node that is connected to u' by such a path. Hence, $m(w)$ is equal to $m(u')$ in T_w.

The correctness for the remaining components can be shown analogously. This is rather straight forward and we do not go into further details. Now the statements have to be shown for all of the outward nodes. This can be done by induction over the components in preorder. It is again very similar to the inward case and we omit the details here. □

For a split node u let $\{u, w\}$ be its incident split edge. Assume that $B_{\{u,w\}}$ would be the split decomposition of G with only one split, namely the one induced by split edge $\{u, w\}$. Further let B_u be the connected component of

*following strictly Algorithm 4.7 the set of bases cases would be empty and the leaf nodes are handled in the inductive step, either way yields the same result

$B_{\{u,w\}} \setminus \{u, w\}$ which includes the node u. Figure 4.3 on page 57 shows such an setup with G being the graph on the left (Figure 4.3(a)) and $B_{\{u,w\}}$ being the graph on the right (Figure 4.3(b)).

If we consider Lemma 10 together with Lemma 15 we can immediately state the following. After performing the out-in and in-out part of Algorithm 4.7 the number of the non split nodes adjacent to u in B_u is equal to $n(u)$ and the number of edges between those nodes in B_u is equal to $m(u)$. Corollary 9 gives the statement in a more formal way.

Corollary 9 *After the execution of line 3 in Algorithm 4.7 for each split node u and B_u defined as above*

$$n(u) = |\{v \in \mathcal{V}(G) : \exists\{u, v\} \in \mathcal{E}(B_u)\}| \tag{4.2}$$

and

$$m(u) = |\{\{x, y\} \in \mathcal{E}(G) : \exists\{u, x\} \text{ and } \{u, y\} \in \mathcal{E}(B_u)\}| \tag{4.3}$$

hold.

We are going to use Corollary 9 to show the last Lemma in this section.

Lemma 16 *Line 4 of Algorithm 4.7, respectively Function 4.10, computes the number of triangles $\delta(v)$ for each node v of a distance hereditary graph.*

Proof: We show exemplarily that component COMP-Ia is correctly handled in line 1 of Function 4.10. Let w' denote the adjacent split node of w. By Lemma 10 node x forms triangles with all edges between neighbors of w' in $B_{w'}$. There are exactly $m(w')$ such edges by Corollary 9. Further x forms triangles with y and all nodes that are neighbors of w' in $B_{w'}$. There are exactly $n(w')$ such neighbors. Consequently, these triangles sum up to $n(w') + m(w')$. By Lemma 10 we have considered all triangles that include x. Thus, $\delta(x)$ is computed correctly in line 2. For node y the same arguments hold and there remains nothing to be shown for this component.

Again showing the correctness for the remaining 6 components is analogous to this case and we omit further details. □

Function 4.8: TRIANGLE COUNTER FOR DH GRAPHS: OUTIN

switch *comp* **do**

1 **case** COMP-Ia — *Figure 4.5(a)*
  $n(w) \leftarrow 2$, $m(w) \leftarrow 1$;

2 **case** COMP-Ib — *Figure 4.5(b)*
  $n(w) \leftarrow 2$, $m(w) \leftarrow 0$;

3 **case** COMP-Ic — *Figure 4.5(c)*
  $n(w) \leftarrow 1$, $m(w) \leftarrow 0$;

case COMP-IIa — *Figure 4.6(a)*
  $n(w) = 1$;
  $m(w) = 0$;

case COMP-IIb — *Figure 4.6(b)*
  $n(w) = n(u') + 1$;
  $m(w) = m(u') + n(u')$;

4 **case** COMP-IIc *case 1* — *Figure 4.6(c)*
  $n(w) = n(u') + 1$;
  $m(w) = m(u')$;

case COMP-IIc *case 2* — *Figure 4.6(d)*
  $n(w) = n(u')$;
  $m(w) = m(u')$;

case COMP-IIIa — *Figure 4.6(e)*
  $n(w) = n(u') + n(v')$;
  $m(w) = m(u') + m(v') + n(u') \cdot n(v')$;

case COMP-IIIb *case 1* — *Figure 4.6(f)*
  $n(w) = n(v')$;
  $m(w) = m(v')$;

case COMP-IIIb *case 2* — *Figure 4.6(g)*
  $n(w) = n(u') + n(v')$;
  $m(w) = m(u') + m(v')$;

Function 4.9: TRIANGLE COUNTER FOR DH GRAPHS: INOUT

switch *comp* **do**

case COMP-II*a* — *Figure 4.6(a)*
- $n(u) = 1;$
- $m(u) = 0;$

case COMP-II*b* — *Figure 4.6(b)*
- $n(u) = n(w') + 1;$
- $m(u) = m(w') + n(w');$

case COMP-II*c case 1* — *Figure 4.6(c)*
- $n(u) = n(w');$
- $m(u) = m(w');$

case COMP-II*c case 2* — *Figure 4.6(d)*
- $n(u) = n(w') + 1;$
- $m(u) = m(w');$

case COMP-III*a* — *Figure 4.6(e)*
- $n(u) = n(v') + n(w');$
- $m(u) = m(v') + m(w') + n(v') \cdot n(w');$
- $n(v) = n(u') + n(w');$
- $m(v) = m(u') + m(w') + n(u') \cdot n(w');$

case COMP-III*b case 1* — *Figure 4.6(f)*
- $n(u) = n(v');$
- $m(u) = m(v');$
- $n(v) = n(u') + n(w');$
- $m(v) = m(u') + m(w');$

case COMP-III*b case 2* — *Figure 4.6(g)*
- $n(u) = n(w');$
- $m(u) = m(w');$
- $n(v) = n(w');$
- $m(v) = m(w');$

Function 4.10: TRIANGLE COUNTER FOR DH GRAPHS: CALC-δ

switch *comp* **do**

case COMP-Ia *Figure 4.5(a)*

$\delta(x) \leftarrow m(w') + n(w')$;

$\delta(y) \leftarrow m(w') + n(w')$;

case COMP-Ib *Figure 4.5(b)*

$\delta(x) \leftarrow m(w')$;

$\delta(y) \leftarrow m(w')$;

case COMP-Ic *Figure 4.5(c)*

$\delta(y) \leftarrow m(w')$;

$\delta(x) \leftarrow 0$;

case COMP-IIa *Figure 4.5(d)*

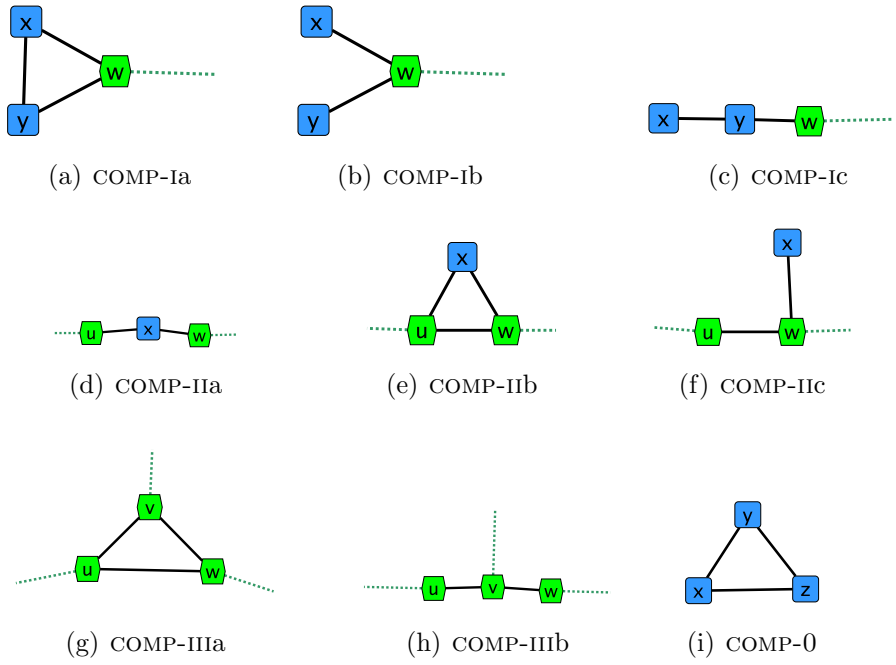
$\delta(x) \leftarrow m(u') + m(w')$;

case COMP-IIb *Figure 4.5(e)*

$\delta(y) \leftarrow m(u') + m(w') + n(u')n(w')$;

case COMP-IIc *Figure 4.5(f)*

$\delta(x) \leftarrow m(w')$;

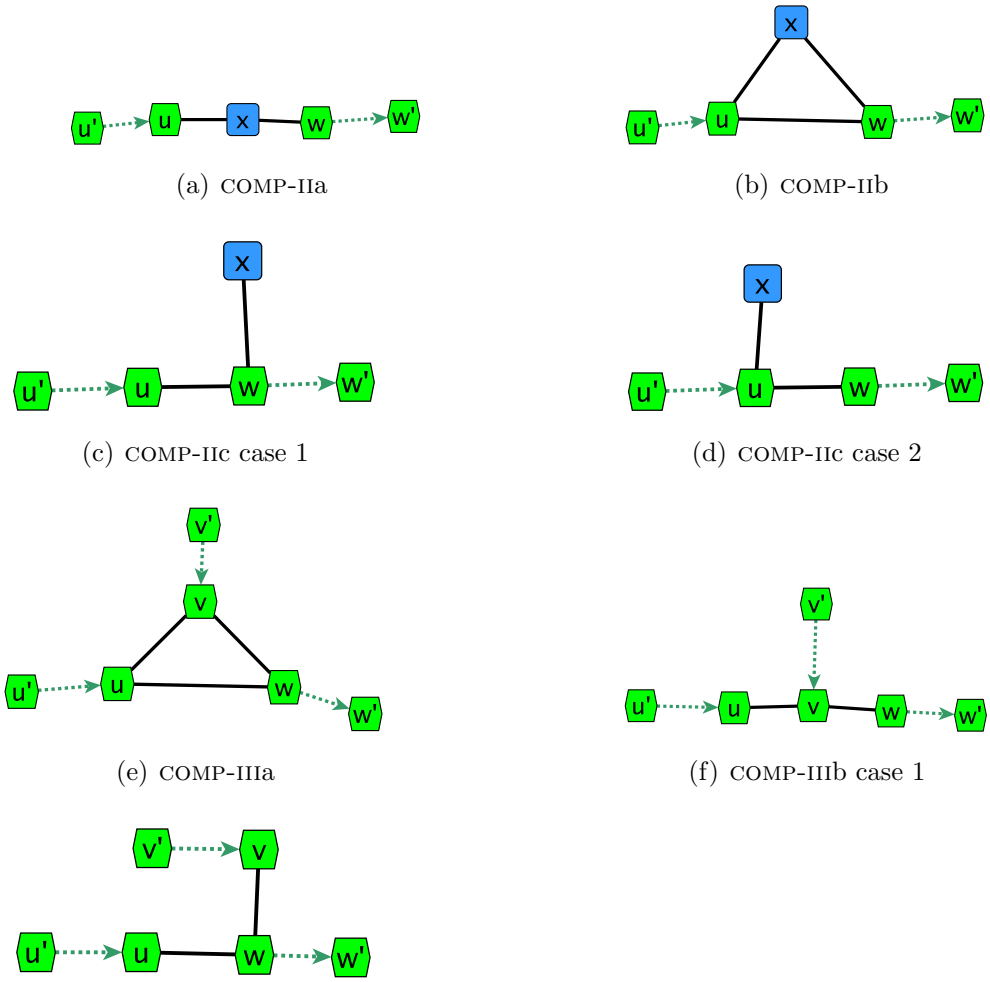


(a) COMP-Ia (b) COMP-Ib (c) COMP-Ic

(d) COMP-IIa (e) COMP-IIb (f) COMP-IIc

(g) COMP-IIIa (h) COMP-IIIb (i) COMP-0

Figure 4.5: Components of a total decomposition.



(a) COMP-IIa

(b) COMP-IIb

(c) COMP-IIc case 1

(d) COMP-IIc case 2

(e) COMP-IIIa

(f) COMP-IIIb case 1

(g) COMP-IIIb case 2

Figure 4.6: Directions for the processing the internal components.

Chapter 5

Applications of Triangle Listing in Network Analysis

Chapter Introduction

In this chapter we will discuss applications in network analysis for which the computation of the related indices is based either on triangle counting or triangle listing.

The clustering coefficient introduced by Watts and Strogatz [1998] is probably the most established network index that is computationally based on triangle counting. Given three actors u, v and w with mutual relations between v and u as well as between v and w, the clustering coefficient of the node v represents the likelihood that u and w are also related. In other words it is the average value of edges between pairs of neighbors. When we consider the graph induced by the neighbors of v it is the density of this graph. Therefore we prefer to use the term neighborhood density. The average value over all nodes is the clustering coefficient coefficient of the graph. It is one of the most popular network indices. At the time of this writing a search with `scholar.google.com` for the term yields about 112000 results.

Harary and Paper [1957] introduced the index of transitivity in the context of linguistics. It has become popular again for some time. However, mainly because it has been used intendedly or unintendedly in the place of the clustering coefficient, in [Newman et al., 2002] for example.

Various versions of the edge weighted neighborhood density have become popular recently [Grindrod, 2002; Lopez-Fernandez et al., 2004; Barrat et al.,

2004; Onnela et al., 2005; Zhang and Horvath, 2005; Kalna and Higham, 2006]. The computation of those differs for the various cases.

Connectivity measures based on short cycles or small cliques have been proposed by Batagelj and Zaveršnik [2003] and considered by Palla et al. [2005]. As a triangle is simultaneously the smallest non trivial clique and the shortest non trivial cycle, efficient listing of triangles plays an important role in the computation of this measures.

Contribution and Related Work. By the nature of this chapter we mostly reproduce results from other publications and present them in a standardized manner.

In the case of the clustering coefficient and transitivity we mainly focus on the properties and relation of theses parameters. The inequality of both indices was discussed by Bollobás and Riordan [2002] and independently by Ebel, Mielsch, and Bornholdt [2002]. We introduce a variant of the clustering coefficient that takes weights of the nodes into account. We show that the relation of both indices can be given by certain functions for the node weights. The resulting equations of this approach were presented in [Bollobás and Riordan, 2002].

The other indices are considered with respect of efficient computation. Our basic approach is to give algorithms that compute the desired output by *generic triangle listing*. This simply means that the algorithm can make use of a procedure that lists all triangles of a graph, each triangle once and only once. In the case of short cycle connectivity we can improve the algorithms from [Batagelj and Zaveršnik, 2003]. The edge weighted clustering coefficient has been used in many variations, we present a general algorithm that is applicable for the different weight functions that have been considered.

Organization. We consider the connectivity induced by short cycles in Section 5.1. A similar measure induced by small cliques follows in Section 5.2. Section 5.3 contains the neighborhood density and most notably an extension to weighted edges. Finally Section 5.4 discusses the clustering coefficient and the related index of transitivity.

5.1 Short Cycle Connectivity

Batagelj and Zaveršnik investigated the connectivity induced by short cycles in [Batagelj and Zaveršnik, 2003]. The idea is that nodes (respectively edges)

have to be connected by a sequence of cycles which are overlapping in some terms. The authors introduce the concepts for cycles of length 3 and give also algorithms for the computation in this case. Some properties are proven for general k-cycles but without algorithmic context.

Triangular Node Connectivity

Two nodes u and w are *triangularly node connected* if there exists a sequence of triangles $(\Delta_1, \ldots, \Delta_s)$ with $u \in \mathcal{V}(\Delta_1)$, $\mathcal{V}(\Delta_i) \cap \mathcal{V}(\Delta_{i+1}) \neq \emptyset$ for $1 \leq i < s$, and $w \in \mathcal{V}(\Delta_s)$. The triangular node connectivity defines an equivalence relation $\mathbf{V}^3$ on the set of nodes.

Batagelj and Zaveršnik give an algorithm to compute the equivalence classes of $\mathbf{V}^3$. It is based on algorithm *edge-iterator* (see page 29 in Section 3.1.3) and it inherits the running time in $\mathcal{O}(d_{\max} \cdot m)$ of that algorithm.

Lemma 17 ([Batagelj and Zaveršnik, 2003]) *The equivalence classes of the relation* $\mathbf{V}^3$ *can be computed in* $\mathcal{O}(d_{\max} \cdot m)$ *time.*

By the above definition two nodes u and w are related by $\mathbf{V}^3$ if and only if there exists a path P that connects u and v and every edge in P is contained in a triangle. Algorithm 5.1 uses this observation to compute the equivalence classes of $\mathbf{V}^3$. It is based on generic triangle listing, which also bounds its running time.

Corollary 10 *The equivalence classes of* $\mathbf{V}^3$ *can be computed in* $\mathcal{O}\left(m^{3/2}\right)$ *time.*

Algorithm 5.1: EQUIVALENCE CLASSES OF $\mathbf{V}^3$

Input: graph $G = (V, E)$;
Output: connected components $C_{1,\ldots,i}$ of G, each containing equivalent nodes;
forall $\Delta_{uvw} \in G$ **do**
  `markEdge`$(\{u, v\})$; `markEdge`$(\{v, w\})$; `markEdge`$(\{w, u\})$;
forall $e \in E$ **do**
  if e *is not marked* **then**
    $G \leftarrow G[E \setminus e]$;
$C_{1,\ldots,i} \leftarrow$ connected components of G;
return $C_{1,\ldots,i}$;

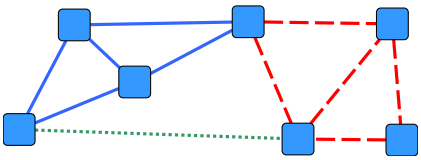


Figure 5.1: An example of triangular connectivity. There is only one equivalence class of the nodes induced by $\mathbf{V}^3$ but three equivalence classes of the edges induced by $\mathbf{E}^3$.

Triangular Edge Connectivity

The triangular edge connectivity induces equivalence classes of edges similar to the node connectivity. Two edges $\{u, w\}$ and $\{x, y\}$ are *triangularly connected* if there exists a sequence of triangles $(\Delta_1, \ldots, \Delta_s)$ with $\{u, w\} \in \mathcal{E}(\Delta_1)$, $\mathcal{E}(\Delta_i) \cap \mathcal{E}(\Delta_{i+1}) \neq \emptyset$ for $1 \leq i < s$, and $\{x, y\} \in \mathcal{E}(\Delta_s)$. The triangular edge connectivity defines an equivalence relation $\mathbf{E}^3$ on the set of the edges. See Figure 5.1 for an example of induced equivalence classes given by the two relations.

Lemma 18 ([Batagelj and Zaveršnik, 2003]) *The equivalence classes of the relation* $\mathbf{E}^3$ *can be computed in* $\mathcal{O}(d_{\max} \cdot m)$ *time.*

Algorithm 5.2: EQUIVALENCE CLASSES OF $\mathbf{E}^3$

Input: graph $G = (V, E)$;
Output: sets $S_{1,\ldots,i}$, each holding equivalent edges
forall $\{u, w\} \in E$ **do**
  `makeSet` $(S_{\{u,w\}})$;
forall $\Delta_{uvw} \in G$ **do**
  `union`$(S_{\{u,v\}}, S_{\{u,w\}})$;
  `union`$(S_{\{u,v\}}, S_{\{v,w\}})$;
return $S_{1,\ldots,i}$;

The algorithm in [Batagelj and Zaveršnik, 2003] depends on processing the triangles in a specific order. Unfortunately, this cannot be guaranteed by generic triangle listing. Our Algorithm 5.2 based on generic triangle listing uses basic set creation, finding, and merging operations. This multiplies the complexity of such operations to the running time complexity of triangle listing. However, they can be performed asymptotically in $\alpha(m)$, see [Cormen et al., 2001] for example. The function α is the inverse of the Ackermann

function. Note that α grows very slowly. It is a constant not bigger than four for any practical considerations.

Corollary 11 *The equivalence classes of the relation* $\mathbf{E}^3$ *can be computed in* $\mathcal{O}\big(\alpha(m) \cdot m^{3/2}\big)$ *time.*

We note that this result is more of theoretic value. The required data structures are generally not available in standard programming libraries, and the *BOOST-graph-library* is the only exception we are aware of. However, a running time in $\mathcal{O}\big(\log m \cdot m^{3/2}\big)$ can be achieved by the more commonly used dynamic tree data structures.

5.2 Small Clique Connectivity

In the two publications [Derenyi et al., 2005; Palla et al., 2005] connectivity by overlapping of small cliques is considered. The basic idea is similar to the short cycle connectivity in [Batagelj and Zaveršnik, 2003] which we discussed in in the previeous section. Indeed, Batagelj and Zaveršnik [2003] generalize the concept briefly and the connectivity considered by Palla et al. [2005] can be seen as a special case of this. Note that the contributions differ in principle. Batagelj and Zaveršnik consider methods whereas [Derenyi et al., 2005; Palla et al., 2005] primarily contain analytical results on networks.

In the small clique connectivity the authors define two nodes u and w equivalent if they are connected by a path of k cliques, where the overlap of the cliques is $k-1$ nodes at least. Let us give a more formal definition. Let $G^k = (V^k, E^k)$ consist of the node set V^k containing all k-cliques of G as nodes. Two nodes v_i^k and v_j^k in V^k are adjacent in G^k if and only if there exist nodes $v_1, \ldots, v_{k-1} \in V$, all of them contained in $\mathcal{V}\big(v_i^k\big)$ and in $\mathcal{V}\big(v_j^k\big)$, i.e. $\big|\mathcal{V}\big(v_i^k\big) \cap \mathcal{V}\big(v_j^k\big)\big| \geq k-1$. The connected components in G^k are the basis of the analytical consideration in [Palla et al., 2005]. Clearly this approach does not define an equivalence relation on V as the cycle connectivity does. A node $v \in V$ might be contained in several nodes $v_i^k \in V^k$ which can belong to different connected components. The number of the different components for a node $v \in V$ is actually one of the considered properties.

Depending on what kind of analysis is done, one can either construct G^k explicitly or construct sets, where each of those contains the nodes of V that can be found in one connected component of G^k. However, these methods are not subject of this work and in the following we will focus on listing all of the k-cliques and in particular on the 4-cliques. The authors set $k =$

4 for the larger networks they consider. Let us note that they performed their computation in a different way, which is only vaguely described in the publication (quote [Palla et al., 2005]: "we use an algorithm which is exponential").

Algorithm 5.3 is essentially a compact presentation of Algorithm 3.7 on page 35. We reproduce it here to conveniently compare the difference to Algorithm 5.4 which is an extension to K_4 listing. Now, let us consider the

Algorithm 5.3: 3-Clique Lister

Input: undirected graph $G = (V, E)$;
nodes $(1, \ldots, n)$ in order of non increasing degrees;
adjacencies $(\gamma_1(v), \ldots, \gamma_{|\Gamma|}(v))$ for each $v = (1, \ldots, n)$ ordered in same order as the nodes;
Output: all 3-cliques as node-sets;

```
1 for ℓ = (1, ..., n) do
2     for k ∈ Γ(ℓ), k < ℓ do
3         for j ∈ Γ(k) ∩ Γ(ℓ), j < k do
              output {j, k, l};
```

Algorithm 5.4: 4-Clique Lister

Input: undirected graph $G = (V, E)$;
nodes $(1, \ldots, n)$ in order of non increasing degrees;
adjacencies $(\gamma_1(v), \ldots, \gamma_{|\Gamma|}(v))$ for each $v = (1, \ldots, n)$ ordered in same order as the nodes;
Output: all 4-cliques as node-sets;

```
1 for ℓ = (1, ..., n) do
2     for k ∈ Γ(ℓ), k < ℓ do
3         for j ∈ Γ(k) ∩ Γ(ℓ), j < k do
4             for i ∈ Γ(j) ∩ Γ(k) ∩ Γ(ℓ), i < j do
                  output {i, j, k, l};
```

time complexity. Recall from Section 3.1.3 that line 1 together with line 2 require m-steps. Line 2 can be performed in $2\sqrt{m}$ steps, because of $k < l$ (see also the proof of Lemma 5 on page 33). Now, by similar arguments line 4 multiplies $3\sqrt{m}$ more steps to the total running time and we can conclude the following corollary.

Corollary 12 *Algorithm 5.4 lists all 4-cliques of an undirected graph. It can be implemented to run in $\mathcal{O}(m^2)$ time.*

As in the case of the triangles we cannot expect any better result since an n-clique contains $\Theta(n^4)$ 4-cliques in terms of n, or $\Theta(m^2)$ 4-cliques in terms of m respectively. We can easily extend the algorithms to list all k-cliques. The total number of steps of such an algorithm is given asymptotically by $1/k \prod_{i=1}^{k} (i\sqrt{m})$ for $k \geq 2$ and we can bound the running time of the derived k-clique listing algorithm by $\mathcal{O}\left(k!\, m^{k/2}\right)$ for $k \geq 2$.

We tested an implementation of Algorithm 5.4 on the "Movie Actor Network 2004" (see page 38 in Section 3.2). The execution time to list all 4-cliques turns out to be about 70 minutes. Let us recall that the execution time for listing all triangles by an implementation of Algorithm 5.3 was about 1 minute. We did not try to list all 5-cliques. We can estimate very roughly that the computation of those for the same graph should take somewhere between 4 and 2000 days. The first number is a linear extrapolation from triangle and 4-clique listing. The upper bound is due to the running time analysis from above.

5.3 Neighborhood Density

Neighborhood Density or Clustering Coefficient of a Node

Let us recall from Section 4 that the neighborhood density (also called the clustering coefficient) of a node v with degree at least 2 is given by the quotient of triangles and wedges $\varrho(v) = \delta(v)/\tau(v)$. A wedge of v is a simple path of length 2 for which v is the center node. Therefore, the number of wedges can be easily computed as $\binom{d(v)}{2}$. This leaves the determination of the number of triangles as the main problem in computing the neighborhood density. We briefly discussed triangle counting algorithms and their complexity in Section 3.1 on page 23. Up to date the most efficient counting algorithms achieve running time in $\mathcal{O}(n^\gamma)$ respectively in $\mathcal{O}\left(m^{2\gamma/(\gamma+1)}\right)$, where γ is the matrix multiplication coefficient.

An algorithm based on generic triangle listing is given in Algorithm 5.5.

Corollary 13 *Algorithm 5.5 computes the neighborhood density (i.e. clustering coefficient of a node) for all nodes of a graph. It can be implemented to run in $\mathcal{O}\left(m^{3/2}\right)$ time.*

Algorithm 5.5: NEIGHBORHOOD DENSITIES

Input: graph G with $\forall v : d(v) \geq 2$
Output: neighborhood density $\varrho(v)$ for all nodes of V
for $v \in V$ **do**
  $\delta(v) \leftarrow 0$;
forall *triangles* $\Delta_{u,v,w}$ *in* G **do**
  $\delta(u) \leftarrow \delta(u) + 1$;
  $\delta(v) \leftarrow \delta(v) + 1$;
  $\delta(w) \leftarrow \delta(w) + 1$;
for $v \in V$ **do**
  $\varrho(v) \leftarrow \delta(v) / \binom{d(v)}{2}$;

The neighborhood densities are rarely considered for themselves. However, the clustering coefficient which is the average value over all neighborhood densities of a graph is very popular. We will discuss it in Section 5.4. A variation of the node density that takes edge weights into consideration will be discussed in the following.

5.3.1 Edge Weighted Neighborhood Density

We extend the neighborhood density in a natural way to take weights on edges into account. A very general definition is given that includes the commonly used varieties by which the weighted neighborhood desity can be defined. We will give an efficient algorithm to compute these. We will discuss some restricted cases and develop a faster algorithm for those.

Generalized Presentation

Let μ denote a function that maps each edge of the graph to a strictly positive real, i.e. $\mu : E \to \mathbb{R}_+$. We write shorthand μ_{uv} for $\mu(\{u, v\})$ in the following. Recall that the neighborhood density $\varrho(v)$ of a node v is defined as the quotient of triangles divided by wedges $\varrho(v) = \delta(v)/\tau(v)$. We rewrite this expression first in Equation 5.1, in which we replace $\delta(v)$ with the sum over all triangles that contain node v. Likewise, we replace $\tau(v)$ with the sum of all wedges for which v is the center node.

$$\varrho(v) = \frac{\sum\limits_{\Delta_{uvw} \in \{\Delta_v \subset G\}} 1}{\sum\limits_{\Upsilon_{uvw} \in \{\Upsilon_v \subset G\}} 1} \tag{5.1}$$

We place functions of the participating edge weights into the sums instead of simply adding up the number 1. This extended formulation is given in Equation 5.2. Note that we rather use a shorthand notation here, e.g. the arguments of f and g are the edge weights of the triangles and not the subgraphs themselves.

$$\varrho_\mu(v) = \varphi(v) \frac{\sum\limits_{\Delta_{uvw} \in \{\Delta_v \subset G\}} f(\mu_{vu}, \mu_{uw}, \mu_{vw})}{\sum\limits_{\Upsilon_{uvw} \in \{\Upsilon_v \subset G\}} g(\mu_{vu}, \mu_{vw})} \tag{5.2}$$

In this general setup we require the functions $\varphi : V \to \mathbb{R}$, $g : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, and $f : \mathbb{R}^3 \to \mathbb{R}$ to be computable in $\mathcal{O}(1)$ time. Further, μ and g must be chosen such that the denominator does not take the value zero. We already mentioned, that the unweighted version of the neighborhood density is not defined for nodes with degree less than two. This caries over to the weighted case. Table 5.1 lists a few functions for f and g that are self-evident. Some combinations of those are shown in Table 5.2. Note that we omitted any constant factors in Table 5.2, for this reason the values may actually differ in the referenced publications.

Matrix Formulation and Computation

Before we list algorithms for computing the weighted coefficient we will review an alternative formulation based on matrices that has been used for example in [Barrat et al., 2004], [Onnela et al., 2005], and [Zhang and Horvath, 2005].

Let the nodes of G be indexed $(v_0, \ldots, v_n)$. Let A be the adjacency matrix: $A \in \{0,1\}_{n \times n}$ with $a_{ij} = 1$ if $\{v_i, v_j\} \in E$, and $a_{ij} = 0$ otherwise. Quite similar, let M be a real quadratic $n \times n$ matrix $M \in \mathbb{R}_{n \times n}$ with entries $m_{ij} = \mu(\{v_i, v_j\})$ if $\{v_i, v_j\} \in E$ and $m_{ij} = 0$ else. With this notation Equation 5.2 can be equivalently expressed by Equation 5.3.

$$\varrho_\mu(v_k) = \varphi(v_k) \frac{\sum\limits_{1 \le i \le n} \sum\limits_{i < j \le n} a_{ik} a_{ij} a_{jk} \cdot f(m_{ik}, m_{ij}, m_{jk})}{\sum\limits_{1 \le i \le n} \sum\limits_{i < j \le n} a_{ik} a_{jk} \cdot g(m_{ik}, m_{jk})} \tag{5.3}$$

	(a) $f(\mu_{vu}, \mu_{uw}, \mu_{vw})$	(b) $g(\mu_{vu}, \mu_{vw})$
1.	1	1
2.	μ_{uw}	$\frac{\mu_{uv}+\mu_{vw}}{2}$
3.	$\frac{\mu_{uv}+\mu_{vw}}{2}$	$\mu_{uv} \cdot \mu_{vw}$
4.	$\frac{\mu_{uv}+\mu_{uw}+\mu_{vw}}{3}$	$\sqrt{\mu_{uv} \cdot \mu_{vw}}$
5.	$\mu_{uv} \cdot \mu_{vw}$	
6.	$\sqrt{\mu_{uv} \cdot \mu_{vw}}$	
7.	$\mu_{uv} \cdot \mu_{uw} \cdot \mu_{vw}$	
8.	$\sqrt[3]{\mu_{uv} \cdot \mu_{uw} \cdot \mu_{vw}}$	

Table 5.1: Some functions for the edge weighted neighborhood density.

reference	$f(\mu_{vu}, \mu_{uw}, \mu_{vw})$	$g(\mu_{vu}, \mu_{vw})$
[Lopez-Fernandez et al., 2004]	μ_{uw}	1
[Barrat et al., 2004]	$\mu_{uv} + \mu_{vw}$	$\mu_{uv} + \mu_{vw}$
[Onnela et al., 2005]	$\sqrt[3]{\mu_{uv} \cdot \mu_{uw} \cdot \mu_{vw}}$	1
[Grindrod, 2002], [Zhang and Horvath, 2005], [Kalna and Higham, 2006]	$\mu_{uv} \cdot \mu_{uw} \cdot \mu_{vw}$	$\mu_{uv} \cdot \mu_{vw}$

Table 5.2: Used combinations for the edge weighted neighborhood density.

In the case of multiplicative functions like $f = \mu_{uv} \cdot \mu_{uw} \cdot \mu_{vw}$ and $g = \mu_{uv} \cdot \mu_{vw}$ this reduces to a simplified version as in Equation 5.4.

$$\varrho^*_\mu(v_k) = \varphi(v_k) \frac{\sum_{1 \le i \le n} \sum_{i < j \le n} m_{uv} \cdot m_{uw} \cdot m_{vw}}{\sum_{1 \le i \le n} \sum_{i < j \le n} m_{uv} \cdot m_{vw}} \tag{5.4}$$

Kalna and Higham [2006] give their version in the form of Equation 5.4. They also give an equivalent of Equation 5.4 in algebraic formulation using matrix power operations and basic vector operations. They conclude that $\varrho^*_\mu(v)$ for all nodes can be computed in $\mathcal{O}(n^3)$ time. The theoretic bound based on fast matrix multiplication is in $\mathcal{O}(n^{2.37})$ [Coppersmith and Winograd, 1990]. The algebraic representation used by Kalna and Higham does not impose any disadvantages in their case since the input is a dense matrix with $m_{ij} > 0$ for $i \neq j$.

Grindrod [2002] uses the weighted neighborhood density in conjunction with a network model, no real networks are considered. Other publications [Lopez-Fernandez et al., 2004; Barrat et al., 2004; Onnela et al., 2005; Zhang and Horvath, 2005] do not go into details of computation. We note that the algorithm directly gained from Equation 5.3 has a running time in $\Theta(n^3)$ and due to the matrix representation a space consumption in $\Theta(n^2)$. For the usual case of sparse networks the running time is not optimal and the space requirement makes such an approach intractable for larger networks.

Algorithms

We will give an algorithm to compute the weighted neighborhood density by generic triangle listing that is based on Equation 5.2. We will further give an improved version for restricted functions g.

A realization directly derived from Equation 5.2 is listed in Algorithm 5.6. It computes the edge weighted neighborhood density for all nodes v of the input graph. We handle the general case of arbitrary functions $f(\mu_{vu}, \mu_{uw}, \mu_{vw})$, $g(\mu_{vu}, \mu_{vw})$, and $\varphi(v)$. We require that these functions are computable in $\mathcal{O}(1)$ time. We further note that depending on the function G and on the graph structure, the result of Algorithm 5.6 might not be defined in the case of a division by zero in Equation 5.2.

The correctness of Algorithm 5.6 is obvious due to the direct translation from Equation 5.2. The loops of line 1 and 4 can be implemented to run in linear time with reasonable data structures. The loops of line 2 and 3 are critical

Algorithm 5.6: EDGE WEIGHTED NEIGHBORHOOD DENSITIES

Input: graph G with $\forall v : d(v) \geq 2$, edge weights $\mu : E \to \mathbb{R}_+$
Output: weighted neighborhood density $\varrho_\mu(v)$ for all nodes of V

1 **for** $v \in V$ **do**
  $\delta(v) \leftarrow 0$, $\tau(v) \leftarrow 0$;
2 **forall** *triangles* $\Delta_{u,v,w}$ *in* G **do**
  $\delta(u) \leftarrow \delta(u) + f(\mu_{uv}, \mu_{vw}, \mu_{uw})$;
  $\delta(v) \leftarrow \delta(v) + f(\mu_{vu}, \mu_{uw}, \mu_{vw})$;
  $\delta(w) \leftarrow \delta(w) + f(\mu_{wu}, \mu_{vu}, \mu_{wv})$;
3 **forall** *wedges* $\Upsilon_{u,v,w}$ *in* G **do**
  $\tau(v) \leftarrow \tau(v) + g(\mu_{vu}, \mu_{vw})$;
4 **for** $v \in V$ **do**
  $\varrho_\mu(v) \leftarrow \varphi(v) \cdot \delta(v)/\tau(v)$;

for the running time. Recall from Chapter 3 that listing all triangles of a graph can be done asymptotically in the number of wedges. The wedges are conveniently listed by a loop over all pairs of neighbors of a node and we conclude with the following Lemma.

Lemma 19 *Algorithm 5.6 can be implemented to run in* $\mathcal{O}(m)$ *space and* $\mathcal{O}(\tau)$=$\mathcal{O}(\sum_V d^2(v))$ *time.*

Let us remark that listing the wedges and for many cases computing g are "cheap" operations, i.e. they involve only small constants. However, listing the triangles may be not so "cheap" and despite the above observation the line 2 can practically determine the execution time.

We now consider the case $g(\mu_{vu}, \mu_{vw}) = c \cdot (\mu_{uv} + \mu_{vw})$, with c being a constant. We can then reformulate the denominator of Equation 5.2 since an edge incident with v is part of $d(v) - 1$ wedges:

$$\sum_{\Upsilon_{uvw} \in \{\Upsilon_v \subset G\}} g(\mu_{vu}, \mu_{vw}) = \sum_{\Upsilon_{uvw} \in \{\Upsilon_v \subset G\}} c \cdot (\mu_{uv} + \mu_{vw}) = c \sum_{u \in \Gamma(v)} (d(v) - 1)\mu_{vu}$$

An adapted version is listed in Algorithm 5.7, in which we replaced line 3 of Algorithm 5.6. The computation of g for all nodes can now be performed in $\mathcal{O}(m)$ steps. Consequently the part requiring the most steps is now the listing of the triangles and computing function f. This implies an effort in $\mathcal{O}\left(m^{3/2}\right)$, see Section 3.1 for details.

Algorithm 5.7: EDGE WEIGHTED NEIGHBORHOOD DENSITIES WITH ADDITIVE g

... (as in Algorithm 5.6) ...

3 **forall** $v \in V$ **do**
  forall $u \in \Gamma(v)$ **do**
    $\tau(v) \leftarrow \tau(v) + (d(v) - 1)\mu_{vu}$;

... (as in Algorithm 5.6) ...

Corollary 14 *The edge weighted neighborhood density can be computed in* $\mathcal{O}\left(m^{3/2}\right)$ *time if* g *is restricted to* $g(\mu_{vu}, \mu_{vw}) = c \cdot (\mu_{uv} + \mu_{vw})$ *or* $g = 1$.

Usage of the Weighted Neighborhood Density

As in the case of the standard neighborhood density it is very popular to compute the average over all nodes and argue for or against the *small world property* of the considered network. But there are also cases, in which the distribution of the weighted neighborhood densities are considered like in the following example.

Kalna and Higham [2006] analyze gene expression correlation data from normal and tumor tissues. They consider the distribution of weighted degrees and edge weighted neighborhood density. The latter is given in the form of

$$\varrho_\mu(v) = \frac{\sum\limits_{\Delta_{uvw} \in \{\Delta_v \subset G\}} \mu_{vu} \cdot \mu_{uw} \cdot \mu_{vw}}{\sum\limits_{\Upsilon_{uvw} \in \{\Upsilon_v \subset G\}} \mu_{vu} \cdot \mu_{vw}}.$$

They found that the cancer tissue has more pronounced features such as sharper pikes in both the degree and neighborhood density distribution. The authors point out that in the weighted neighborhood density distribution these differences are more clearly visible than in the degree distribution.

5.4 Clustering Coefficient and Transitivity

We have already introduced the clustering coefficient and the transitivity in the Preliminaries, see Section 4. We first recall some definitions briefly. Then we further investigate the properties of the indexes and the relation of the both.

The Clustering Coefficient

The clustering coefficient was introduced by Watts and Strogatz [1998] in the context of social network analysis. Given three actors u, v and w with mutual relations between v and u as well as between v and w, it represents the likeliness that u and w are also related. We introduced this concept formally as the neighborhood density $\varrho(v) = \delta(v)/\tau(v)$ (Equation 2.10 on page 16). The clustering coefficient of a graph is the average over the neighborhood densities of its nodes. To compute the clustering coefficient of existing networks with nodes of degree one we define

$$c\,(G) = \frac{1}{|V'|} \sum_{v \in V'} \varrho(v), \tag{5.5}$$

where V' is the set of nodes v with $d(v) \geq 2$. Alternatively to Equation 5.5 the term $\varrho(v)$ is in some cases defined to be either zero or one for nodes v with degree at most one and included in the sum. The computation of the index is straight forward. It essentially relies on computing the neighborhood densities which we discussed in Section 5.3.

Transitivity

We defined the transitivity of a graph G as used in [Newman et al., 2002] as

$$t(G) = \frac{3\,\delta(G)}{\tau(G)},$$

see Equation 2.12 on page 17. It is essentially the *index of transitivity* [Harary and Paper, 1957] or the *transitivity ratio* [Harary and Kommel, 1979] restricted to undirected graphs. Like the clustering coefficient it involves wedges and triangles and it is also between zero and one by Equation 2.9. The computation of this index is straight forward and its efficiency is bounded by the efficiency of counting all triangles in a graph. Note that there is no known algorithm that is more efficient in counting the triangles for the whole graph compared to counting the number of triangles for all nodes.

Let us consider a series of sparse graphs G_n with $m \in o\big(n^{4/3}\big)$ and at least one high degree node v with $d(v) \in \Theta(n)$. Because of the latter $\tau(G) \in \Omega(n^2)$ holds. By $\delta \in \mathcal{O}\big(m^{3/2}\big)$ (Equation 2.6) we get $\delta \in o(n^2)$ and we can conclude with the following.

Corollary 15 *The transitivity t approaches zero for a series of growing sparse graphs with high degree nodes.*

This leads to some important consequences. Skewed or even power law degree distributions have been found in many real work networks and these have commonly nodes with relatively high degree. By Corollary 15 the transitivity does not seem to be a meaningful network index for those kinds of networks. Bollobás and Riordan [2002] proved that the value of t drops with $(\log^2 n)/n$ for constructed graphs by the *LCD model*. In practice this model is equivalent to the so called preferential attachment generators, which will be briefly discussed in this work in Section 6.1 on page 95.

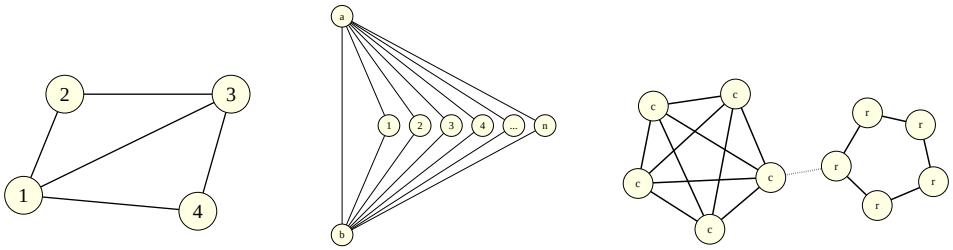
Notes on Naming Conventions

We follow historical conventions in the way we use the names "clustering coefficient" and "transitivity" for a graph. The latter derives from transitive relations and the equivalent property in directed graphs, see page 51 in Section 4.2 where the transitive orientation is introduced. A priori the term doesn't make much sense in undirected graphs. The word "clustering" itself is awfully overloaded in network analysis. For the case of the clustering coefficient of a node we rather prefer to use the term neighborhood density. In the case of the index for the whole graph we keep the name "clustering coefficient". We can thus also easily distinguish between the corresponding value for nodes or the whole graph. What we call a wedge is called a *triple* in some publications that deal with both the clustering coefficient and the transitivity. The term "triple" is also used with other meanings in literature, e.g. in [Buriol et al., 2006]. Therefore, we decided not to adopt this term.

Extending the Clustering Coefficient with Node Weights and Formal Relation to the Transitivity

The basic results of the following were given by Bollobás and Riordan [2002]. We give the contents in a unified presentation and extend them.

The transitivity as in Equation 2.12 was claimed to be equal to the clustering coefficient as in Equation 2.11 in [Newman et al., 2002]. However, this is clearly not true and the complete graph of four nodes with one edge removed is the smallest counterexample, see Figure 5.2(a). It is even possible to construct a series of graphs, such that $c \to 1$ and $t \to 0$ for $n \to \infty$, see the *double star* [Bollobás and Riordan, 2002] in Figure 5.2(b). The reverse case can be constructed by a graph G_{c+r} consisting of a c-clique and an r-ring, see Figure 5.2(c). We get $t(G) \approx \binom{c}{3} / \left(\binom{c}{3} + r/3\right)$ and hence $t(G) \to 1$ for



(a) $c \neq t$ (b) $c \to 1$ and $t \to 0$ (c) $c \to 0$ and $t \to 1$

Figure 5.2: Examples for the inequality of clustering coefficient and transitivity.

$r \in o(c^3)$. On the other hand we get $c(G) \approx c/(r+c)$ and hence $c(G) \to 0$ for $r \in \omega(c)$.

A formal relation between the two indices will follow after introducing the following convenient extension. The definition of the clustering coefficient does not consider the fact that, depending on the network, some nodes might be more important than others. This can be specified by a weight function that is either induced by the graph structure or given explicitly. For a weight function $w : V \to \mathbb{R}^+$ we define the *node weighted clustering coefficient* to be

$$c_w(G) = \frac{1}{\sum\limits_{v \in V'} w(v)} \sum_{v \in V'} w(v)\varrho(v). \tag{5.6}$$

Two implicit weight functions are immediate, the degree-based weight $w(v) = d(v)$, and the weight given by the number of wedges $w(v) = \tau(v)$. In the second case, $\tau(v)$ simply cancels out in each additive term of the numerator, and we get $c_\tau(G) = \sum \delta(v) / \sum \tau(v)$ which can be rewritten by equation Equation 2.5 and Equation 2.12 to

$$t(G) = c_\tau(G). \tag{5.7}$$

We recognize the transitivity herewith as the wedge weighted clustering coefficient. The following properties can be drawn directly from this observation.

Corollary 16 *The indices c and t are equal for graphs where*

- *all nodes have the same degree, or*

- *all nodes have the same clustering coefficient.*

An equivalent of Equation 5.7 was given in [Bollobás and Riordan, 2002].

Historical Notes on Confusing the Clustering Coefficient with the Transitivity

As shortly mentioned previously the indices c and t are not uncommonly confused with each other. We list some literature on the history of this in chronological order.

Harary and Paper [1957] introduced the term index of transitivity in an analytical sense. As the name suggests it is defined for directed graphs. However, simplified to undirected graphs it is essentially what we defined as the transitivity.

Watts and Strogatz [1998] coined the term clustering coefficient. The index was introduced in context of studying what the authors refer as high local "clustering" and "cliquishness" of networks. On the one hand it is introduced as a measure on the nodes which is equivalent with what we defined as the neighborhood density. Watts and Strogatz define the clustering coefficient of a graph as the average of the neighborhood densities over all nodes. They show that the value of the clustering coefficient is relatively high for many real world networks compared to the random graph models of Erdős and Rényi [1959] and Gilbert [1959]. They also give a random graph "model" based on random modification of a torus that achieves high clustering coefficients on the one hand but also a short average path length for a random pair of nodes. The latter behavior is known as the "small world" or "six degrees of separation" result of Milgram [1967]. With the work [Watts and Strogatz, 1998] a "small world" network becomes known as a network that has high clustering coefficient and short average path length.

Barrat and Weigt [2000] give a "simplified" version of the clustering coefficient. They write:"a simple redefinition" ... "without altering its physical signification". This redefinition is actually equivalent to the transitivity. However, Barrat and Weight use their version of the index only with respect to certain graphs that have little variation in their degree distribution from the average degree. We have seen that both definitions are practically equivalent for these networks, see Corollary 16. Barrat and Weight actually support the equivalence with some mathematical arguments and "numerical checks", i.e. computational experiments on small world networks.

In 2001 and 2002 confusion manifest in a series of papers by Newman [2001]; Newman, Strogatz, and Watts [2001]; Newman, Watts, and Strogatz [2002]. The last of the three publications contains the quote "Consider the following expression for the clustering coefficient (which is one of a number of ways it can be written):"

$$\frac{3 \times \text{ number of triangles on the graph}}{\text{number of connected triples}^* \text{ of vertices}} \tag{5.8}$$

Clearly, Equation 5.8 is equivalent to the definition of the transitivity in Equation 2.12. The publications [Newman et al., 2001] and [Newman et al., 2002] contain also computational results on real world networks where the clustering coefficient is considered. They compute a value of 0.199 for the movie actor collaboration network. We compute $t = 0.166$ and $c = 0.785$ for the data set we have available for the movie actor collaboration of the year 2002. While some variation might be explained by slightly different data sets (the movie actor network is updated frequently) it seem clear that Newman, Watts and Strogatz actually computed the transitivity. In the original work [Watts and Strogatz, 1998] the clustering coefficient is computed to 0,79 for the movie actor collaboration network. Our result 0.785 agrees with this.

Bollobás and Riordan [2002] and independently Ebel, Mielsch, and Bornholdt [2002] state that the both terms c and t are not equivalent.

Values for Real World Networks

We consider values for c and t of real world networks to test of agreement with Corollary 16 and Corollary 15. These are presented in Table 5.3 along with some other properties. Some of these networks have a longer description along with extended properties in Section 3.2.

The first network is based on the German roads, see also page 37. It is almost planar, has little maximum degree and little variation in the degree, see Figure 3.3(a) on page 39. The values of c and t are quite close. The following three networks are based on projections of bipartite networks. The "DBLP Trier" is based on collaborative publications within the computer science community. The "actor" networks are based on actors playing together in a movie, see also page 38. All of these networks have considerable high values of c and comparatively lower values of t. The actor network has an increase in t from the year 2002 to the year 2004. Note that this is not a contradiction

*wedges, in our definition

	roads	DBLP Trier	actor 2002	actor 2004	google 2002	WWW Notre Dame
n	$4.8 \cdot 10^6$	$3.1 \cdot 10^5$	$3.8 \cdot 10^5$	$6.7 \cdot 10^5$	$3.9 \cdot 10^5$	$3.3 \cdot 10^5$
m	$5.9 \cdot 10^6$	$8.3 \cdot 10^5$	$1.5 \cdot 10^7$	$2.7 \cdot 10^7$	$4.8 \cdot 10^5$	$1.1 \cdot 10^7$
$d_{\max}$	7	248	3956	4605	1160	10721
d_{avr}	2.5	5.4	78.7	82.6	2.4	6.7
t	0.048	0.341	0.166	0.262	0.007	0.088
c	0.050	0.760	0.785	0.796	0.228	0.466

Table 5.3: Clustering coefficient and transitivity in real networks.

to Corollary 15. The maximal degree did only increase by a factor of 1.16 while the size of the network almost doubled.

The last two networks share some characteristics. The "WWW Notre Dame" is based on hyper links within a domain. See page 38 for a description and Figure 3.5(a) on page 41 for further properties. The "google 2002" network is based on the google contest from the year 2002. We do not exactly now how it is constructed. By its properties and origin we might speculate on a web graph background, too. Nevertheless, both have high degree nodes and quite small values for the transitivity. The values for "WWW Notre Dame" are higher for those of "google 2002", this can be explained by large number of edges in the first one. In general we see good agreement with our conclusions from Corollary 15 and Corollary 16 for the considered real networks.

Chapter 6

A Graph Generator with Adjustable Clustering Coefficient and Increasing Cores

Chapter Introduction

The creation of random networks is important to understand existing networks and to test algorithms. Seminal work exploring properties of random networks dates back to 1959, see e.g. [Erdős and Rényi, 1959], [Gilbert, 1959], [Austin et al., 1959].

After the discovery of so called power-laws [Faloutsos et al., 1999] or at least heavy tailed degree distributions [Chen et al., 2001] in Internet related graphs and other networks; generators were proposed that mimic this behavior. A very popular approach functions by the so called *linear preferential attachment* [Barabási and Albert, 1999; Albert et al., 1999]. Starting with a prime graph a newly inserted vertex connects to existing nodes with probability linear to their current degree. The concept was introduced in [Barabási and Albert, 1999] and refined to a strict mathematical model in [Bollobás et al., 2001]. Efficient algorithms for various generators are discussed in [Batagelj and Brandes, 2005].

However, generators only based on preferential attachment were found not to achieve various properties of existing graphs, see e.g. [Bu and Towsley, 2002]. In this section, we consider generators that on the one hand work

according to the preferential attachment rule but also achieve high values of the clustering coefficient. Our proposed generator achieves also an increasing core number κ and increasing size of the cores. See page 18 in Section 2.4 for an introduction to the core structure of a graph.

Contribution and Related Work. Holme and Kim proposed a generator with "tunable clustering coefficient" based on preferential attachment in [Holme and Kim, 2002]. However, we show that their approach gives the desired high clustering coefficients only for a limited set of parameters. On the other hand the method in [Li, Leonard, and Loguinov, 2004] achieves high clustering coefficients in a more general sense. However, it does so by post-processing an existing graph and hence does not model a growing process.

We give a very simple and well defined extension of preferential attachment and we show experimentally, that our approach can achieve any value of the clustering coefficient up to 0.68 for a wide range of input parameters. We also consider the core structure of the generated graphs. We reason that traditional preferential attachment as well as the approach in [Holme and Kim, 2002] does not achieve any non trivial core structure. However, we validate that our approach does so. More precisely we experimentally show, that our generator produces increasing core numbers κ and increasing size of the cores $\mathfrak{C}_{k \leq \kappa}$ for a series of growing graphs.

A simplified version of our generator was used in [Schank and Wagner, 2004, 2005a] to test the efficiency of an algorithm for approximation of the clustering coefficient. These publications contain the results with respect to the achieved clusterings coefficient. However, these are presented in a very compact manner. The results with respect to the evolution of the core structure are new in this work. We are not aware of any other generators that are designed to produce increasing core structures for evolving graphs. Actually, results on the evolution of the core structure in real world networks appeared only very recently, for example in [Aggarwal et al., 2006].

Organization. The remaining of this chapter is organized as follows. We start with the general linear preferential attachment generator in Section 6.1 where we give a well defined generator in pseudo code. We review the generator by Holme and Kim [2002] in Section 6.2. We show experimentally that it fails to generate high clustering coefficients for general parameters and give an explanation for this behavior. We give the new generator in Section 6.3 and experimentally verify some properties of the generated graphs.

We first give a well defined pseudo code of our generator in Section 6.3. In the following we consider the achieved clustering coefficients and the basic core structure of the generated graphs. In the last part we give two variants that address undiscretized clustering coefficient values and a simplified implementation.

6.1 A Linear Preferential Attachment Generator

The basic idea behind linear preferential attachment is to connect in each step one new vertex v with μ new edges to existing vertices. Algorithm 6.1 depicts a linear preferential attachment generator, where a part of it is outsourced in Function 6.2. Linear preferential attachment is characterized by choosing an existing vertex u to be connected to v with probability $d(u)/\sum_{v\in V} d(v)$ as in line 1 of Function 6.2. The return statement in line 2 of function `PA-Step` is not used in Algorithm 6.1, but it is required for the algorithm in the next section when we use Function 6.2 again.

Algorithm 6.1: A Preferential Attachment Generator

Input: integer n, $\mu \geq 1$;
Output: graph $G^{\mathrm{PA}}_{n,\mu}$
Data: initially empty node array A, node v
1 $G(V, E) \leftarrow \max\{\mu, 2\}$-clique ;
forall $v \in V$ **do**
  for $(1, \ldots, d(v))$ **do**
    append $v \rightarrow A$;
for $(\mu + 1, \ldots, n)$ **do**
2 $v \leftarrow$ `NewNode()` ;
  $V \leftarrow V \cup \{v\}$;
  for $(1, \ldots, \mu)$ **do**
    `PA-Step(`v`)`;

Function 6.2: `PA-Step`

Input: node v
Data: Node: u;
repeat
1 $\quad r \leftarrow$ uniformRandomInteger($\{0, \ldots, |A|\}$);
$\quad u \leftarrow A[r]$;
until $u \neq v$ `and` $\{u, v\} \notin E$;
$E \leftarrow E \cup \{u, v\}$;
append $u \rightarrow A$, $v \rightarrow A$;
2 **return** u;

Note that we differ in constructing a clique as a prime graph (line 1) from many "traditional" descriptions of preferential attachment generators, including [Barabási and Albert, 1999]. Often a single node or a graph with an empty edge set in the beginning is used. However, if there are no edges the operation in line 1 of Function 6.2 is not defined. Further Function `PA-Step` is easier to describe this way since it is always possible to find μ nodes that have no link to v.

At first glance the running time appears to be a critical point. The $\mu + 1$-th node connects to all currently existing nodes. Choosing neighbors randomly in the loop of Function 6.2 seems to be quite inefficient. However, it is actually equivalent to the so called *coupon collector problem* and for connecting the $\mu + 1$-th to all other nodes the expected running is in $\mathcal{O}(\mu \log \mu)$.

Corollary 17 *Algorithm 6.1 is well defined. For a fixed value of μ it can be implemented to use linear space and to run in expected linear time.*

6.2 The Holme-Kim Generator

Holme and Kim give a variation of a preferential attachment generator that is also supposed to achieve a high clustering coefficient in [Holme and Kim, 2002]. They try to achieve this by adding a so called "triad formation step". Algorithm 6.3 lists an equivalent of Holme's and Kim's algorithm and

Function 6.4 lists the triad formation step.

Algorithm 6.3: HOLME-KIM GENERATOR

Input: integer: n, $\mu \geq 1$; real: $0 \leq p \leq 1$;
Output: graph $G^{\mathrm{HK}}_{n,\mu,p}$
Data: initially empty node array A; nodes: v, w
1 $G(V, E) \leftarrow \max\{\mu, 2\}$-clique ;
forall $v \in V$ **do**
  for $(1, \ldots, d(v))$ **do**
    append $v \rightarrow A$;
2 **for** $(\mu + 1, \ldots, n)$ **do**
  $v \leftarrow$ `newNode()` ;
3 $w \leftarrow$ `PA-Step`(v) ;
  for $(2, \ldots, \mu)$ **do**
4 with probability p: `TF-Step`(v,w) ;
5 else: $w \leftarrow$ `PA-Step`(v) ;

In each loop (line 2) of Algorithm 6.3 the `PA-Step` is called at least once (line 3). Then for the remaining 2 to μ new edges of v either `TF-Step` is called with probability p (line 4) or `PA-Step` is called again with probability $1-p$ (line 5). In function `TF-Step` a triangle $\Delta_{v,w,u}$ between the new node v, the node from the last `PA-Step` w and a random neighbor u of w is formed.

Function 6.4: `TF-Step`

Input: node v,w
repeat
  $u \leftarrow$ uniformRandonAdjacentNode(w);
until $v \neq u$ `and` $\{v, u\} \notin E$;
$E \leftarrow E \cup \{u, v\}$;
append $u \rightarrow A$, $v \rightarrow A$;

We applied a subtle change to the original description of Holme and Kim, which is not given in pseudo code and leaves some room for interpretation. The difference is that we start with a μ- or at least 2-clique as in Algorithm 6.1 of the previous section. As already noted this makes function `PA-Step` easier to describe but also simplifies Function 6.4. Any node in $V \setminus \{v\}$ has at least degree μ and in course of executing `TF-Step` node v has less than μ neighbors. Hence it is always possible to find a neighbor of u of w that has no link to v. In the original description this is not necessarily the case and `PA-Step` is performed instead. We remark that for large n and for higher numbers in the loop of line 2 this case rarely happens and both specifications

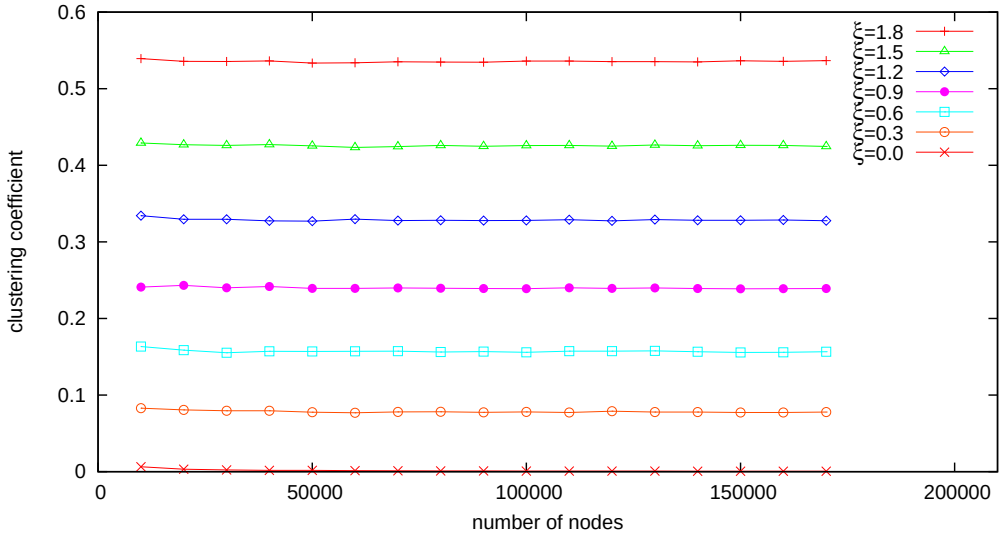


Figure 6.1: Clustering coefficients for the Holme-Kim generator with $\mu = 3$.

are practically equivalent.

Now, let us analyze Algorithm 6.3 with respect of the achieved clustering coefficient. On average the `TF-Step` is called $(\mu - 1)p$-times for each new node v and Holme and Kim define*

$$\xi = (\mu - 1)p. \tag{6.1}$$

Figure 6.1 shows the results for $\mu = 3$ and various values of ξ. This agrees with the result of Holme and Kim shown in Figure 3(a) in [Holme and Kim, 2002]. As to be expected the clustering coefficient is highest for $p = 1$. The authors have carried out only experiments with $\mu = 3$, and they write: "We only focus on the case of $\mu = 3$ with expectation that other values of μ should give qualitatively the same behavior."†

We will check this claim experimentally in the following. Figure 6.2 shows the results for $p = 1$, which will give the highest clustering coefficients, and various values of μ. We find that the clustering coefficient falls significantly with higher values for μ. We will try to give an explanation for this in the following.

Our hypothesis is, that the clustering coefficient of the Graph is "not far" from the neighborhood density of the most recently added node. Now, the

*The original notation in is different: μ is denoted with m and ξ with m_t.

†We allowed us to change the symbols in the quotation to the ones we use.

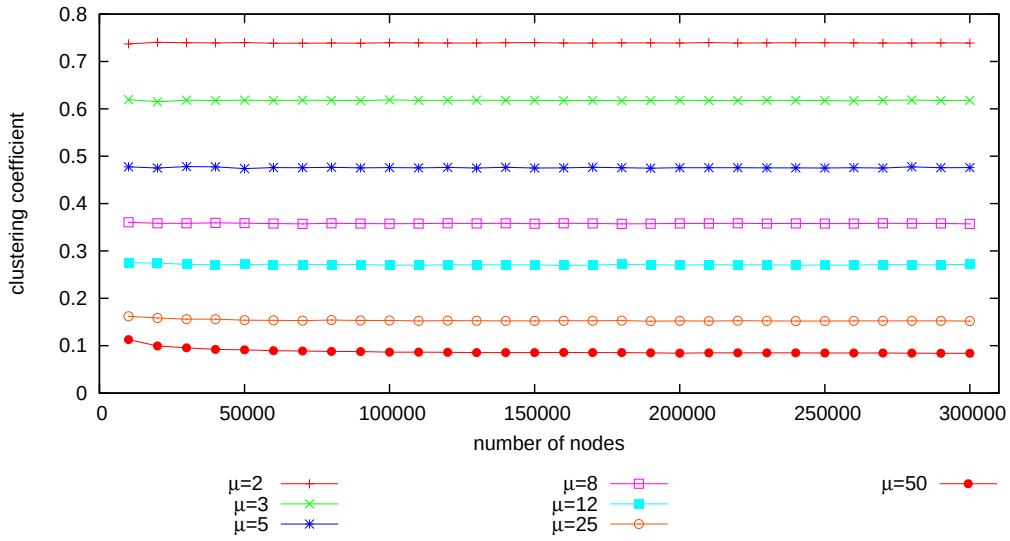


Figure 6.2: Highest clustering coefficients achieved with the Holme-Kim generator for various μ.

μ	2	3	5	8	12	25	50
$\frac{(\mu-1)p}{\binom{\mu}{2}} = \frac{2}{\mu}$	1	0.67	0.4	0.25	0.17	0.08	0.04
$c_{\text{exp}}\left(G^{\text{HK}}_{n,\mu,p}\right)$ for $n > 3 \cdot 10^5$	0.74	0.62	0.47	0.35	0.27	0.15	0.08

Table 6.1: Experimental results for the clustering coefficient $c_{\text{exp}}\left(G^{\text{HK}}_{n,\mu,p}\right)$ from Figure 6.2 compared with $c\left(G^{\text{HK}}_{n,\mu,p}\right)$ as in Equation 6.2.

most recently added node v has ξ expected triangles created by the function `TF-Step`. We ignore other triangles that may be induced by existing edges between the neighbors of v for now. Since v has degree μ our claim is

$$\lim_{n\to\infty} c\left(G^{\text{HK}}_{n,\mu,p}\right) \approx \frac{(\mu-1)p}{\binom{\mu}{2}} = \frac{2p}{\mu}. \tag{6.2}$$

This is generally in agreement with Table 6.1 where we compare the achieved clustering coefficient with our prediction. We see two discrepancies:

- The observed coefficients are lower than our prediction for high values. We explain this by the fact that new arriving nodes introduce higher degrees of existing nodes and therefore effectively reduce the coefficient of existing nodes.

- For low values we get slightly higher coefficients. This can be explained by "existing", i.e. not by function `TF-Step` purposely created triangles. For low values they superimpose the effect mentioned above.

However, our general conclusion is that the clustering coefficient of the Holme-Kim generators falls with the inverse of μ.

6.3 The New-Generator

In the last section we observed, that the clustering coefficients of graphs generated by the Holme-Kim generator fall significantly with increasing values of μ, i.e. the number of added edges in each step. In the following we propose a new generator that is based on linear preferential attachment, but does also achieve high clustering coefficients as well as increasing core size for a family of growing graphs.

The New-Generator listed in Algorithm 6.5 is similar to standard preferential attachment. However, in each round of the main loop we call an additional `CC-Step` (Function 6.6) after each `PA-Step`. In the `CC-Step` edges between the neighbors of the most recently added node v are inserted until the desired neighborhood density $\varrho(v)$ is reached.

Algorithm 6.5: New-Generator

Input: integer n, $\mu \geq 1$, $\epsilon \leq \binom{\mu}{2}$;
Output: graph $G^{\texttt{NEW}}_{n,\mu,\epsilon}$
Data: initially empty node array A, node v
$G(V, E) \leftarrow \max\{\mu, 2\}$-clique ;
forall $v \in V$ **do**
  for $(1, \ldots, d(v))$ **do**
    append $v \rightarrow A$;
for $(\mu + 1, \ldots, n)$ **do**
  $v \leftarrow$ `newNode()` ;
  $V \leftarrow V \cup \{v\}$;
  for $(1, \ldots, \mu)$ **do**
    `PA-Step`(v);
  `CC-Step`(v, ϵ);

Function 6.6: `CC-Step`

Input: node v, int ϵ;
Data: node u;
while $\delta(v) < \epsilon$ **do**
  repeat
    $u \leftarrow$`randomAdjacentNode(`v`)`;
    $w \leftarrow$`randomAdjacentNode(`v`)`;
  until $u \neq w$ `and` $\{u, w\} \notin E$;
  $E \leftarrow E \cup \{u, w\}$;
  append: $u \rightarrow A$, $w \rightarrow A$;

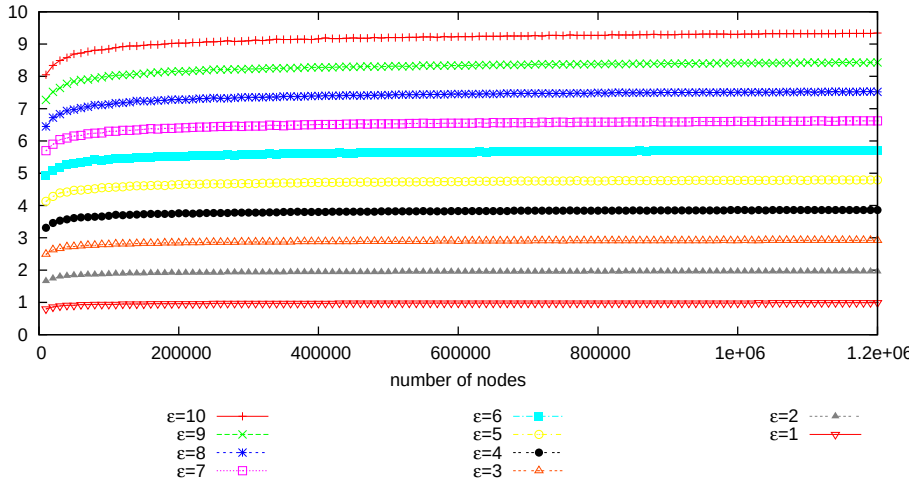


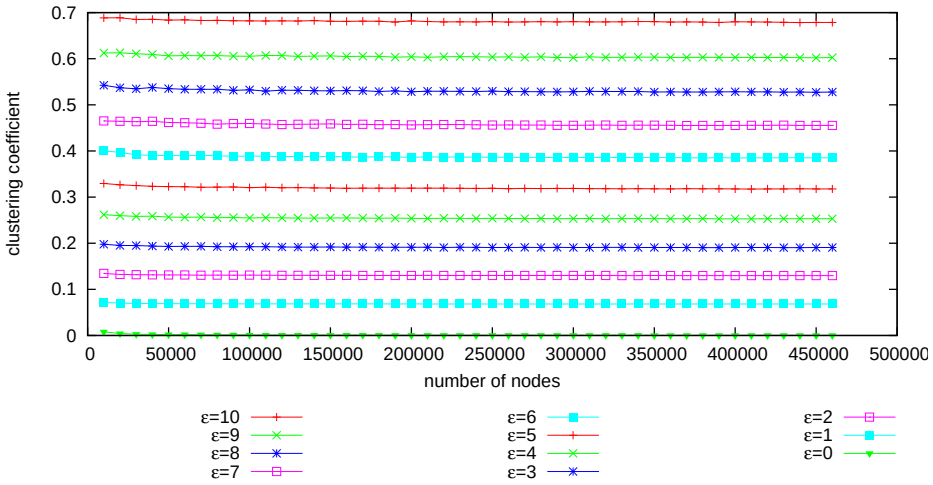
Figure 6.3: Number of added edges on average in `CC-Step` for $\mu = 5$ and various ϵ.

6.3.1 Clustering Coefficient

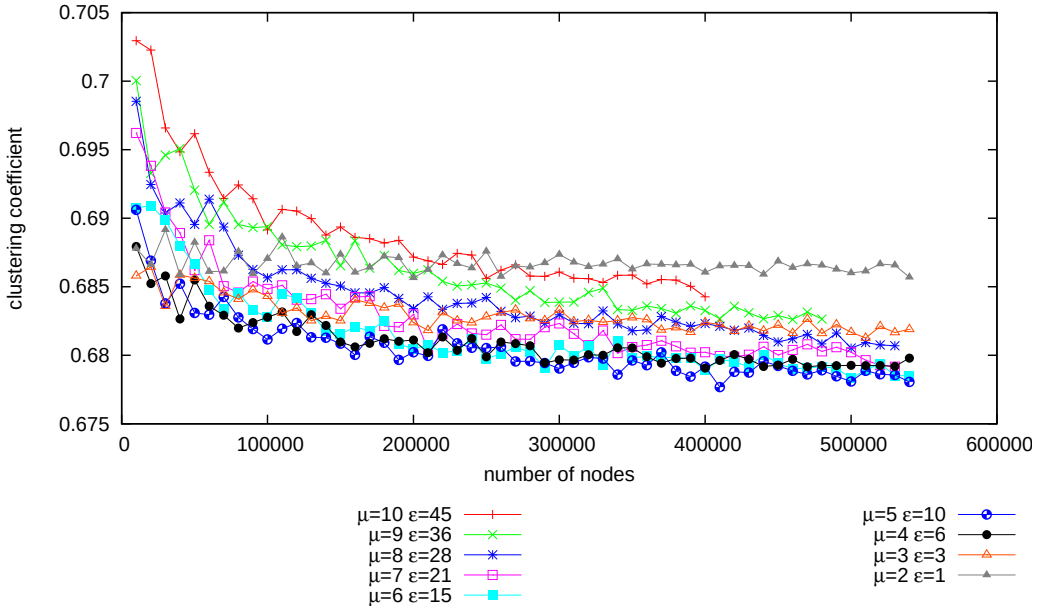
We assume again that the clustering coefficient of a graph is mainly influenced by the last nodes that were added. Hence, we adjust the neighborhood density of the last added node v, by increasing the number of edges between neighbors of v until the number ϵ is reached, see Function 6.6. Note that the neighborhood density of the last added node is then bounded by

$$\varrho(v) \geq \frac{\epsilon}{\binom{\mu}{2}}. \tag{6.3}$$

Compared to standard preferential attachment and also to the Holme-Kim generator, it is not exactly known how many edges $\overline{\mu}$ are inserted in each



(a) $\mu = 5$, various ϵ



(b) hightest clustering coefficients: $\epsilon = \binom{\mu}{2}$

Figure 6.4: Clustering coefficients for Algorithm 6.5.

$\frac{\epsilon}{\binom{\mu}{2}} = \frac{\epsilon}{10}$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
$c_{\text{exp}}\left(G^{\texttt{NEW}}_{n,5,\epsilon}\right)$	0.07	0.12	0.19	0.25	0.32	0.39	0.46	0.52	0.60	0.68

Table 6.2: Experimental results from Figure 6.4 compared with Equation 6.3.

round by our method $\mu \leq \overline{\mu} \leq \mu + \epsilon$. Figure 6.3 shows how many edges are created on average in Function `CC-Step` for $\mu = 5$ and various values of ϵ. We can see that $\overline{\mu} \geq 9/10 \cdot \epsilon + \mu$ and also that there is little fluctuation.

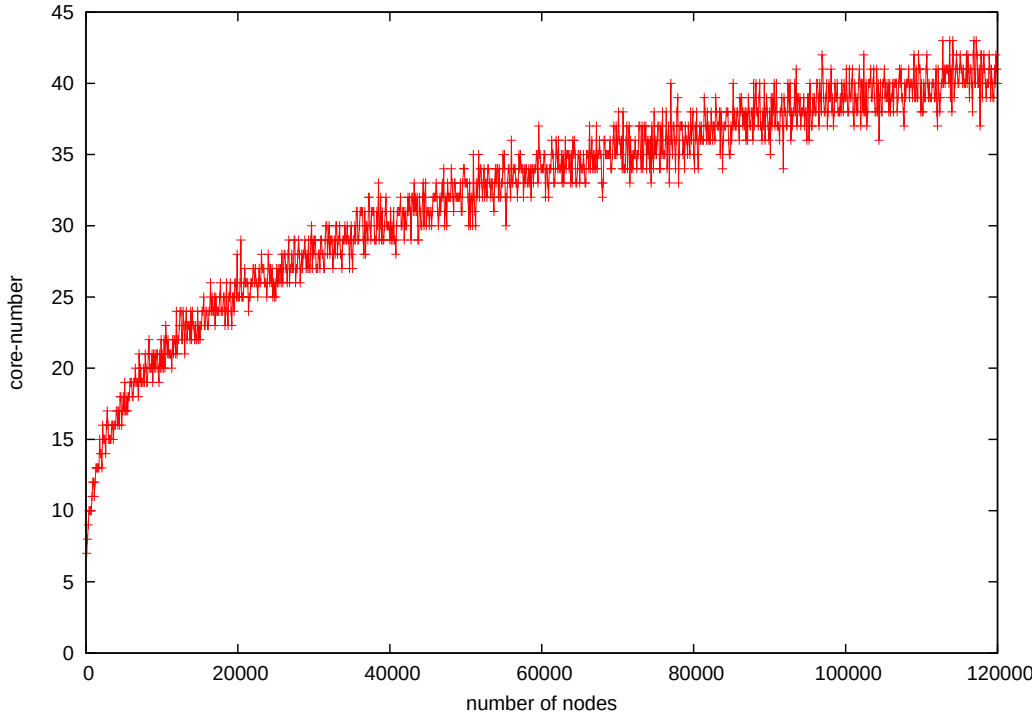
Figure 6.4(a) and Table 6.2 show the resulting clustering coefficients of Algorithm 6.5 for $\mu = 5$ and various values of ϵ. Figure 6.4(b) shows the result for μ and the corresponding highest values of $\epsilon = \mu(\mu-1)/2$. We first notice that we can achieve high clustering coefficients. If we compare with Equation 6.3, the neighborhood density of the last added node, we get approximately

$$\lim_{n\to\infty} c\left(G^{\texttt{NEW}}_{n,\mu,\epsilon}\right) \approx 0.7\frac{\epsilon}{\binom{\mu}{2}}. \tag{6.4}$$

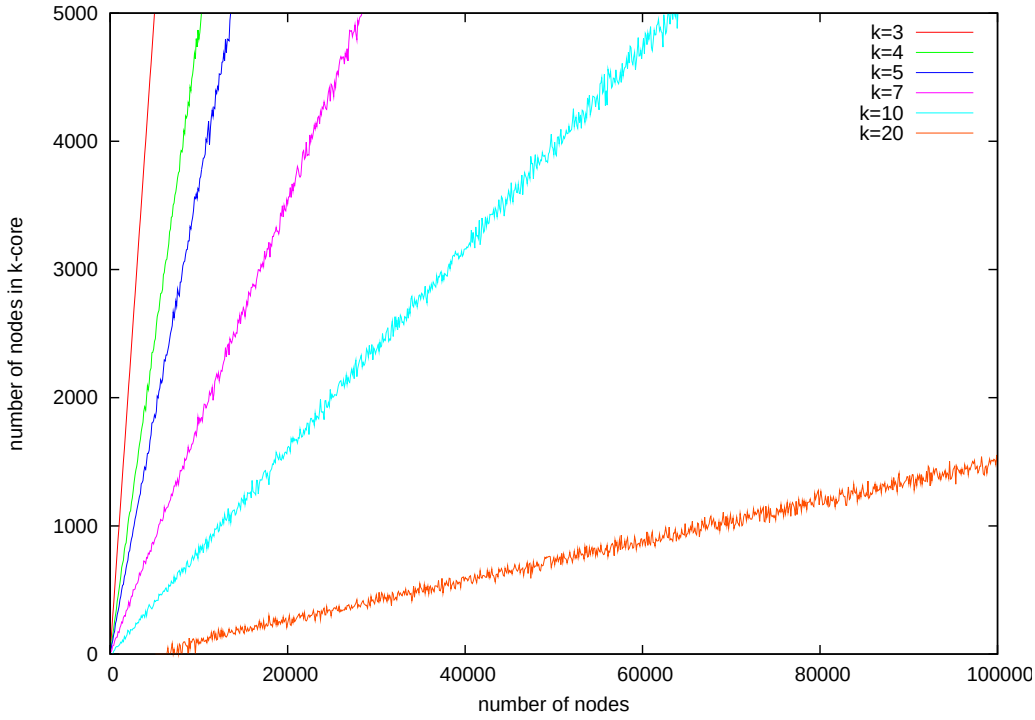
6.3.2 Core Structure

We introduced the core concept in Section 2.4. Table 6.3 lists some real world networks with their corresponding core number and network size. Some of these networks have been introduced in Section 3.2, the others are described in [Schank and Wagner, 2005c]. While the core number differs for the networks, it is considerably high in many cases. The movie actor network shows a notable increase in the core number between the two given instances from 2002 and 2004. As the movie actor network origins from a projection of a bipartite network (see page 38 in Section 3.2), one might expect a relative high core number. The projection results in a number of larger cliques. Each of them induces a core of its size minus one and the total core number can be even larger due to overlaying of cliques.

The evolution of the core structure in the Autonomous System Graph of the Internet was studied in [Aggarwal et al., 2006]. This network is based on routing paths between admistrative collections of Internet router subnetworks. This recent study shows that the core number actually increased during the years from 2001 to 2004. For the same network a layout method which highlights the pronounced core structure was proposed in [Baur et al., 2005].



(a) core number for $\mu = 3$ and $\epsilon = 2$



(b) core sizes for $\mu = 3$ and $\epsilon = 2$

Figure 6.5: Resulting cores for Algorithm 6.5.

	n	m	κ
Road Network of Germany	4799875	5947001	3
Movie Actor Network 2002	382219	15038083	365
Movie Actor Network 2004	667609	27581275	1005
Movie Actor Network 2004 (no porn)	366131	7711904	173
DBLP Authors	307971	831557	114
Google Contest 2002	394510	480259	13
High Energy Physics Publications	27240	341923	37
Internet Router Network	192244	609066	32
Notre Dame WWW	325729	1090108	155
US Patents	3774768	16518947	64

Table 6.3: Core numbers for some real world networks.

In standard preferential attachment as well as in the Holme-Kim generator new edges are always inserted with the node added at last, which has then a degree of μ. Therefore the empty graph remains when all nodes with degree μ or less are removed iteratively. We can draw the following corollary from this observation.

Corollary 18 *The core number of a graph generated by standard preferential attachment or generated by the Holme-Kim generator is not higher than the number of added edges in each round:* $\kappa\left(G^{\mathrm{PA}}_{n,\mu}\right) \leq \mu$ *and* $\kappa\left(G^{\mathrm{HK}}_{n,\mu,p}\right) \leq \mu$.

We want to emphasize that both the core number and the core size only depend on the input parameter μ but not on the size of the graph.

Let us now consider the proposed generator Algorithm 6.5. Additionally to adding μ edges to a new node v we also add ϵ edges between the neighbors of v. Theses neighbors have been selected by preferential attachment and we can therefore hope that we also place a considerable fraction of these edges between high degree nodes. This should increase the density within the subgraph of the high degree nodes and we expect that the core size $\kappa(G)$ and also the number of nodes in the k-cores are increasing for growing graphs. This can only be true if $\epsilon > 0$ holds, otherwise the New-Generator is identical to the preferential attachment generator.

Figure 6.5 shows the core number, and some k-core sizes for graphs generated with our algorithm. As we predicted both are increasing over time, i.e. for growing sizes of the graph. We therefore argue that our generator models real world networks also with respect of the core structure much better than previous algorithms.

Variations of the New-Generator

Undiscretizing the Clustering Coefficient

The clustering coefficients of Algorithm 6.5 between consecutive values of ϵ might differ from the desired value. Algorithm 6.7 lists a variation with an additional input parameter $0 \leq p \leq 1$. It combines the clustering coefficient of the two consecutive values $\epsilon - 1$ and ϵ linearly such that

$$c\left(G^{\texttt{NEW}}_{n,\mu,\epsilon,p}\right) = c\left(G^{\texttt{NEW}}_{n,\mu,\epsilon-1}\right) + p\left[c\left(G^{\texttt{NEW}}_{n,\mu,\epsilon}\right) - c\left(G^{\texttt{NEW}}_{n,\mu,\epsilon-1}\right)\right]$$

with high probability. Now it is possible to achieve any number between 0 and 0.68 for clustering coefficient c.

Algorithm 6.7: Variation of Algorithm 6.5

Input: integer n, $2 \leq \mu$, $1 \leq \epsilon \leq \binom{\mu}{2}$;
real $0 \leq p \leq 1$;
Output: graph $G^{\texttt{NEW}}_{n,\mu,\epsilon,p}$
Data: initially empty node array A; node v;
$G(V, E) \leftarrow \max\{\mu, 2\}$-clique ;
forall $v \in V$ **do**
  for $(1, \ldots, d(v))$ **do**
    append $v \rightarrow A$;
for $(\mu + 1, \ldots, n)$ **do**
  $v \leftarrow$ `NewNode()` ;
  $V \leftarrow V \cup \{v\}$;
  for $(1, \ldots, \mu)$ **do**
    `PA-Step`(v);
  with probability p: `CC-Step`(v, ϵ);
  else: `CC-Step`$(v, \epsilon - 1)$;

Simplifying the Implementation

Function 6.6 was given such that it is convenient to analyze the results. However, computing the neighborhood and keeping track of it is a computational burden, though it can be done efficiently, see Chapter 3. Moreover, it complicates an implementation significantly. Therefore we give in Function 6.8 an alternative of `CC-Step` that is simpler but looses some accuracy. However, from the results shown in Figure 6.3 we can assume that this variation behaves well, i.e. the added number of edges between neighbors is stable in expectation.

Function 6.8: VARIATION OF `CC-Step`

```
Input: node v, int ε
Data: node: u;
for (1, ..., ε) do
    u ←RandomAdjacentNode(v);
    do
        w ←RandomAdjacentNode(v);
    while u = w ;
    if {u, w} ∉ E then
        E ← E ∪ {u, v};
        append: u → A, v → A;
```

Chapter Conclusion

We gave a new graph generator that is based on preferential attachment and creates graphs with adjustable clustering coefficient, in which any value up to 0.68 can be achieved. The value of the clustering coefficient does not depend on the total number of edges added in each round. The total number of edges added in each round is not deterministic but it can be estimated well. Our experiments show that core number and the size of the cores of the generated graphs grow with the size of the graph. This behavior is reasonable for existing evolving networks as it has been shown recently. The dynamic analysis of the core structure seems to be at its very beginning. Our generator is a first step in this area with qualitative reasonable behavior.

Chapter 7

Approximating Clustering Coefficient, Transitivity and Neighborhood Densities

Chapter Introduction

The consideration of very large graphs in network analysis [Kumar et al., 2000; Abello et al., 2002; Eubank et al., 2004] requires or makes it at least very desirable that the analysis can be performed in linear or preferably even in sub-linear time. We have seen in Chapter 3 that triangle listing and triangle counting can be performed very efficiently. However, the algorithms do still have super-linear running times with respect to the input size. It is even impossible to achieve running time in the size of the input for the case of triangle listing. Consequently applying these algorithms to compute network indices like the clustering coefficient results in super-linear running times, too.

This chapter is devoted to algorithms for the efficient approximation of the following indices in sub-linear time with respect to the size of graph. We consider the case of approximating the clustering coefficient c and its node weighted version c_w for which the transitivity is a special case. We also consider the neighborhood densities of the nodes.

Contribution and Related Work. Approximation algorithms based on sampling are well known and have been used extensively for various applications. Eubank et al. [2004] use such a method to approximate the clustering

coefficient c by sampling nodes and compute the average neighborhood density of the samples. They mention that by applying general probabilistic bounds it suffices to take $\mathcal{O}(\log n)$ samples.

Our main contribution is a $\mathcal{O}(1)$ time algorithm for approximating the clustering coefficient c. We further give a $\mathcal{O}(n)$ time algorithm for the case of the node weighted clustering coefficient c_w of a graph. These results and the algorithms on which they are based on were published in [Schank and Wagner, 2005a]. These work here extends our publication by the following. We review the method based on sampling nodes and also show why our method to approximate c is favorable. We will further give an algorithm to approximate all neighborhood densities of a graph in $\mathcal{O}(n \log n)$ time, along with some consideration of practical applications.

Peripherally related to our work is the approximation of the number of triangles by streaming algorithms [Bar-Yosseff et al., 2002; Jowhari and Ghodsi, 2005; Buriol et al., 2006].

Organization. The rest of this chapter is organized as follows. We give some definitions and other prerequisites first. We review the method of approximating c by sampling the neighborhood density of several nodes in Section 7.1. We will then introduce a method based on sampling wedges in Section 7.2. In Section 7.3 we consider the approximation of the neighborhood density of nodes of a graph. A conclusion closes this chapter.

Preliminaries

Approximation and Randomized Algorithms

We introduced the concept and especially the running time of an algorithm on page 17 in Section 2.3. An *approximation algorithm* is an algorithm that transforms the input to an output that is by some measure close to the desired output. The full meaning for our purpose will become clear within this chapter. *Randomized algorithms* appear frequently in the context of approximation algorithms. A randomized algorithm uses a number of random bits besides its usual input. In general either the output or the running time or both will depend also on the random input bits. The main interest will be in the expected output or the expected running time and the behavior of the distribution "around" the expected value. Actually, we have already seen randomized algorithms in Chapter 6. The probabilistic context there was

quite clear without giving any additional explanation. However, this chapter requires some further knowledge in this area which we will introduce now.

Probability

We will extend the basic concepts of probability which we presented on page 12 in Section 2.1. Two random variables X and Y are *independent* if for all $x, y \in \mathbb{R}$

$$\mathbb{P}\left[\{X = x\} \cap \{Y = y\}\right] \quad = \quad p(x) \cdot p(y).$$

For a real number r and independent random variables X, Y the *linearity of expectation*

$$\mathbb{E}\left[r \cdot X \cdot Y\right] = r \cdot \mathbb{E}\left[X\right] \cdot \mathbb{E}\left[Y\right] \tag{7.1}$$

is a useful property. There exist several results that bound the likeliness to deviate from the expectation of a random variable. The very general Markov inequality

$$\mathbb{P}\left[X \geq r\mathbb{E}\left[X\right]\right] = \frac{1}{r}. \tag{7.2}$$

is valid for non-negative random variables X and r a positive real. For sums of non-negative independent random variables the much stronger Chernoff bound can be derived from Equation 7.2. The most general version of this bound can not be applied with ease in many cases and therefore many variants exist.

A very convenient bound for our purpose is due to Hoeffding [Hoeffding, 1963]. Since it is similar to the Chernoff bound it is also referred to as the Chernoff-Hoeffding bound. Let X_i be independent real random variables bounded by $0 \leq X_i \leq \beta$ for all i. Then, Hoeffding's bound [Hoeffding, 1963] states for $k \in \mathbb{N}_{>0}$

$$\mathbb{P}\left(\left|\frac{1}{k}\left(\sum_{i=1}^{k} X_i\right) - \mathbb{E}\left[\frac{1}{k}\left(\sum_{i=1}^{k} X_i\right)\right]\right| \geq \epsilon\right) \leq 2e^{\frac{-2k\epsilon^2}{\beta^2}}. \tag{7.3}$$

Hoeffding's Bound in Example

We give an example on approximating an average value by Hoeffding's bound first. Assume that we have m boxes. Each box is either empty or contains one item. We can compute the average number of items in a box in $\mathcal{O}(m)$

time by looking in each box. Alternatively we can pick k-times a box uniformly at random and sum up the found items. At the end we divide the summed up items by k and have an approximation of the average number of items in one box. Note that the expected number of items found in one box sampled uniformly at random is exactly the average value we are looking for. Hoeffding's bound tells us how many boxes k we should pick to have the desired result, i.e. the deviation from the average and the probability that this result is correct.

For example we want to be off from the average by at most $\epsilon = 0.01$ items and the value should be correct in all but one of $\nu = \binom{49}{6}$ cases*, i.e. the normalized probability of getting a correct result is $p = \frac{\nu-1}{\nu}$. The random variable X_i corresponds to picking up a box and count the including items which are either zero ore one, therefore $\beta = 1$. Now, rearranging Equation 7.3 generally yields

$$k = \left\lceil \frac{\beta^2}{2\epsilon^2} \ln 2\nu \right\rceil \tag{7.4}$$

and with our parameters we compute $k \approx 8.6 \cdot 10^4$ for the number of samples we should take. In asymptotic notation we get $k \in \Theta(1/\epsilon^2 \log \nu)$. We want to emphasize the following two points:

- the number of samples, and thus also the running time, is independent of the number of total items, and
- raising the probability of correctness (logarithmic factor) is very cheap compared to improving the error bound (quadratic factor).

Note that it is not uncommon to include the input size nevertheless, e.g. for $P(m)$ a polynomial in m one can get increasing probabilities of correctness with $\nu \in \Theta(P(m))$ whilst achieving running times in $\mathcal{O}(\log m)$. However, in such cases as in our example there is a priori no reason why a larger instance should be handled with higher probability of correctness than a smaller instance.

*$1 / \binom{49}{6} = 1/13983816$ corresponds to the likeliness of winning the main price in the weekly German lottery with one single trial

7.1 Approximating the Clustering Coefficient by Sampling Nodes

Let us recall that the clustering coefficient of a graph G is the average neighborhood density

$$c(G) = \frac{1}{|V|} \sum_{v \in V} \varrho(v),$$

in which $\varrho(v) = \delta(v) / \binom{d(v)}{2}$. We assume, that $\varrho(v)$ is well defined for all nodes, i.e. $d(v) \geq 2$ for all $v \in V$.

A straightforward method to approximate the clustering coefficient of a graph $c(G)$ would be to take samples from the neighborhood densities of the set of nodes. Let S be a multiset of nodes chosen uniformly at random from V. Then

$$\operatorname{apx}(c(G)) = \frac{1}{|S|} \sum_{v \in S} \varrho(v)$$

is an approximation of $c(G)$ and Algorithm 7.1 lists a direct realization with $k = |S|$.

Algorithm 7.1: c-Approximation by Node Sampling

Input: graph $G = (V, E)$ with $\forall v \in V$: $(v) \geq 2$;
number of samples k;
Output: approximation of $c(G)$;
Data: real r;
$r \leftarrow 0$;
for $(1, \ldots, k)$ **do**
    $v \leftarrow$ uniformRandomNode of all nodes V;
1   $r \leftarrow r + \varrho(v)$;
return $\operatorname{apx}(c(G)) = r/k$;

This method has been used by Eubank et al. [2004]. They mention that choosing k in $\Theta(\log n)$ "is easily shown via a Chernoff bound to give an accurate estimate".

Actually, if we directly apply Hoeffding's bound (Equation 7.3) it suffices to choose $k \in \mathcal{O}(1)$ for a fixed error bound ϵ and fixed probability p of approximating within the range induced by this bound. See the example on page 111 for details. In the following we will assume $k \geq 1$ to be bounded by a constant.

Let us investigate the expected running time of Algorithm 7.1. The crux clearly lies in line 1 in which the neighborhood density of the sampled node v is computed. Without any knowledge of the structure of G and especially the neighborhood $\Gamma(v)$ a direct algorithm has no choice but performing $\binom{d(v)}{2}$ tests for edges in the neighborhood of v. For $k \in \mathcal{O}(1)$ the probability of a node v being sampled in Algorithm 7.1 is asymptotically $1/n$. Therefore the following equation holds for the expected running time $\mathbb{E}[T]$ of a direct implementation of Algorighm 7.1

$$\mathbb{E}\left[T_{\text{Alg7.1}}\right] \in \Theta\left(\frac{1}{n}\sum_{v\in V} d^2(v)\right) \tag{7.5}$$

Now, let us consider a family of graphs with exactly one node v of degree $d(v) \in \Omega(n)$ and all other nodes with degrees bounded by a constant. We can then directly conclude from Equation 7.5 that

$$\mathbb{E}\left[T_{\text{Alg7.1}}\right] \in \Omega(n)\,.$$

Note that we are still considering worst case running times with respect to the input of graphs. The expected running time does not depend on various graph instances, but is due to the randomized sampling of the nodes.

We have seen that a constant number of samples leads to an at least linear expected running time. Let us have a closer look at the running times depending on the random input. The whole point of applying tail inequalities as the Chernoff bounds is to show that is is very unlikely to deviate far from the expected value. This has been achieved for the computed approximation $\text{apx}(c)$. However, the distribution of the running times of Algorithm 7.1 itself doesn't behave well in this sense. Let us consider again a graph with one single high degree node as above. For this case the expected running time is in $\Theta(n)$, but we get running time in $\Omega(n^2)$ with probability as high as k/n

$$\mathbb{P}\left[T_{\text{Alg7.1}} \in \Omega\big(n^2\big)\right] \geq \frac{1}{n}.$$

To amend this problem one could also approximate the neighborhood densities $\varrho(v)$ for the nodes itself. However, we will propose a more elegant solution in the following section.

7.2 Approximating the Clustering Coefficient by Sampling Wedges

We will give approximation algorithms for the node weighted clustering coefficient c_w (including the transitivity) and for the unweighted clustering coefficient c. Before getting to the algorithms we will first extend the transitivity $t(G)$ in a similar manner as the clustering coefficient $c\,(G)$ in Equation 5.6 on page 88.

Extending the Transitivity

Let Π be the set of all wedges in a graph G, then $\tau(G) = |\Pi|$. Further consider the mapping $X : \Pi \to \{0, 1\}$, where $X(\Upsilon)$ equals one if there is an edge between the outer nodes of the wedge Υ, and zero otherwise. Then we can rewrite the transitivity as

$$t(G) = \frac{1}{|\Pi|} \sum_{\Upsilon \in \Pi} X(\Upsilon).$$

This equation is similar to the definition of the clustering coefficient in Equation 2.11 on page 17. We define

$$t_\varpi(G) = \frac{1}{\sum\limits_{\Upsilon \in \Pi} \varpi(\Upsilon)} \sum_{\Upsilon \in \Pi} \varpi(\Upsilon) X(\Upsilon). \tag{7.6}$$

with a weight function $\varpi : \Pi \to \mathbb{R}^+$.

Lemma 20 *The weighted clustering coefficient is a special case of the weighted transitivity.*

Proof: For a node weight function w, let $\varpi(\Upsilon) = \frac{w(v)}{\tau(v)}$ where v is the center of wedge Υ. Further, let Π_v be the set of wedges with center v. Then accordingly $\tau(v) = |\Pi_v|$ and $\sum_{\Pi_v} X(\Upsilon) = \delta(v)$. The following transformation completes the proof:

$$\begin{aligned}
c_w &= \frac{1}{\sum\limits_{V'} w(v)} \sum_{V'} w(v)\varrho(v) &&= \frac{1}{\sum\limits_{V'} \frac{w(v)}{t(v)} t(v)} \sum_{V'} w(v) \frac{m(v)}{t(v)} \\
&= \frac{1}{\sum\limits_{V'} \frac{w(v)}{t(v)} \sum\limits_{\Pi_v} 1} \sum_{V'} \frac{w(v)}{t(v)} \sum_{\Pi_v} X(\Upsilon) &&= \frac{1}{\sum\limits_{V'} \sum\limits_{\Pi_v} \frac{w(v)}{t(v)}} \sum_{V'} \sum_{\Pi_v} \frac{w(v)}{t(v)} X(\Upsilon) \\
&= \frac{1}{\sum\limits_{\Pi} \varpi_v} \sum_{\Pi} \varpi_v X(\Upsilon) &&= T_\varpi
\end{aligned}$$

□

The proof above will be useful for the upcoming approximation algorithms. Let Υ be a wedge with center node v. For $w(v) = \tau(v)$, i.e. $\varpi \equiv 1$ we get the unweighted transitivity $t_\varpi = t$. Likewise for $w \equiv 1$, i.e. $\varpi(\Upsilon) = 1/\tau(v)$ we get the unweighted clustering coefficient $t_\varpi = c$.

7.2.1 Approximating the Weighted Clustering Coefficient

We will show the following theorem in this Section.

Theorem 9 *The node weighted clustering coefficient c_w and particularly the transitivity t can be approximated in $\mathcal{O}(n)$ time.*

Roughly speaking our approximation algorithm samples wedges with appropriate probability. It then checks whether an edge between the non center nodes of the wedge is present. Finally, it returns the ratio between the number of existing edges and the number of samples. The pseudo code is presented in Algorithm 7.2. For simplicity we restrict the node weights to strictly positive integers.

Lemma 21 *For a graph G with given node weights $w(v)$, a value $\operatorname{apx}(c_w(G))$ that is in $[c_w(G) - \epsilon, c_w(G) + \epsilon]$ with probability at least $\frac{\nu-1}{\nu}$ can be computed in $\mathcal{O}\left(n + \frac{\log \nu}{\epsilon^2} \log n\right)$ time.*

Proof: We prove that Algorithm 7.2 has the requested properties. Let us first consider the time complexity. The running time of the first for-loop (starting at line 1) is obviously in $\mathcal{O}(n)$. For the second for-loop (line 2), the `findIndex` function (line 4) can be executed in $\log(n)$ steps asymptotically by performing binary search. Choosing two adjacent nodes (line 5 up to line 6) is expected to be in constant time. Testing of edge existence (line 6) is expected to be in constant time as well if for example a hashed data structure is used for the edges. Finally, defining $k = \lceil \ln(2\nu)/(2\epsilon^2) \rceil$ gives total expected running time of $\mathcal{O}\left(\frac{\log \nu}{\epsilon^2} \log n\right)$ for the second for-loop.

In order to prove the correctness for our choice of k, we make use of Hoeffding's bound [Hoeffding, 1963]. See Equation 7.3 on page 111 and in general Section 7 starting at page 111. We will now prove that the expectation $\mathbb{E}[l/k]$ is equal to c_w and that the bounds on ϵ and $\frac{\nu-1}{\nu}$ are fulfilled for our choice of k. The proof of Lemma 20 on page 115 implies that c_w can be computed by testing for each wedge whether it is contained in a triangle or not. With the

Algorithm 7.2: NODE WEIGHTED CLUSTERING COEFFICIENT APPROXIMATION

Input: graph $G = (V, E)$ with $\forall v \in V$: $d(v) \geq 2$;
arbitrary indexed order of nodes $(v_1, \ldots, v_n)$;
node weights $w : V \to \mathbb{N}_{>0}$;
number of samples: k;
Output: approximation of c_w
Data: node variables: u, w;
integer variables: r, l, j, $\phi_{0,\ldots,n}$;
$\phi_0 \leftarrow 0$;
1 **for** $i = (1, \ldots, n)$ **do**
  $\phi_i \leftarrow \phi_{i-1} + w(v_i)$;
$l \leftarrow 0$;
2 **for** $(1, \ldots, k)$ **do**
3 $r \leftarrow$ `uniformRandomNumber(`$\{1, \ldots, \phi_n\}$`)`;
4 $j \leftarrow$ `findIndex(` $i : \phi_{i-1} < r \leq \phi_i$`)`;
5 $u \leftarrow$ `uniformRandomAdjacentNode(`v_j`)` ;
  repeat
    $w \leftarrow$ `uniformRandomAdjacentNode(`v_j`)` ;
  until $u \neq w$;
6 **if** $\{u, w\} \in E$ **then**
    $l \leftarrow l + 1$;
return $\text{apx}\,(c_w(G)) = l/k$;

same notation as in Section 7.2 and particularly as in the proof of Lemma 20, we get

$$c_w(G) = \frac{1}{\sum\limits_{u \in V'} w(u)} \sum_{v \in V'} \sum_{\Upsilon \in \Pi_v} \frac{w(v)}{\tau(v)} X(\Upsilon)$$

where $w(v)/\tau(v)$ is the weight of the corresponding wedge. This corresponds to the probability of $w(v)/\left(\tau(v)\sum w(u)\right)$ that a wedge is chosen in one single loop starting at line 2. Hence, by linearity of the expectation $\mathbb{E}\left[l/k\right] = c_w$ holds. The random variable $X(\Upsilon)$ is a mapping from Π to $\{0,1\}$, and consequently $\beta = 1$ in Equation 7.3. We can now immediately see that the bounds on ϵ and the probability $\frac{\nu-1}{\nu}$ are fulfilled for our choice of k. □

One may regard the error bound ϵ and the probability ν as fixed parameters. The number of wedges $\tau(v)$ for a node v can be computed in $\mathcal{O}(1)$ time if e.g. adjacency arrays are used. Theorem 9 follows from these observations.

7.2.2 Approximating the Clustering Coefficient

The central statement of this section is the following Theorem.

Theorem 10 *The clustering coefficient c of a graph G can be approximated in constant time.*

To show it we will use a simplified version of Algorithm 7.2 which is listed in Algorithm 7.3. The main difference is that the node v, which corresponds to node v_j in Algorithm 7.2, is now sampled uniformly at random from all nodes in V. Thus, all lines dealing with the node weights are not required and removed.

To see the correctness one verifies that a wedge at center node v is sampled with probability $1/\left(\tau(v)|V|\right)$ which corresponds to the correct weight $w \equiv 1$ or $\varpi = 1/\tau$ for obtaining c, compare to the proof of Lemma 20 on page 115 and the paragraph following that proof. The rest follows analogously as in the proof of Lemma 21 on page 116.

Corollary 19 *For a graph G a value $\operatorname{apx}\left(c\left(G\right)\right)$ that is in $[c(G)-\epsilon, c(G)+\epsilon]$ with probability at least $\frac{\nu-1}{\nu}$ can be computed in $\mathcal{O}\left(\frac{\log \nu}{\epsilon^2}\right)$ time.*

Algorithm 7.3: CLUSTERING COEFFICIENT APPROXIMATION

Input: graph $G = (V, E)$ with $\forall v \in V$: $d(v) \geq 2$;
number of samples: k;
Output: approximation of c_w
Data: node variables: u, w;
integer variables: l;
$l \leftarrow 0$;
for $(1, \ldots, k)$ **do**
  $v \leftarrow$ `uniformRandomNode`(V);
  $u \leftarrow$ `uniformRandomAdjacentNode`(v) ;
  repeat
    $w \leftarrow$ `uniformRandomAdjacentNode`(v) ;
  until $u \neq w$;
  if $\{u, w\} \in E$ **then**
    $l \leftarrow l + 1$;
return $\operatorname{apx}(c(G)) = l/k$;

Performance in Practice. We implemented Algorithm 7.3 to approximate the clustering coefficient. We set the parameters to $\epsilon = 0.01$ and $\nu = \binom{49}{6}$. It takes between 0.8 seconds for a very small graph of 4 nodes up to 1.4 seconds for the larger "Movie Actor Network 2004" (see page 38 in Section 3.2). This difference could be explained by the cache of the CPU. Note, that the execution times do not contain the creation of the used hash data structures for the edges.

7.3 Approximating the Neighborhood Densities

In this section we give algorithms to approximate the neighborhood density $\varrho(v)$, i.e. the clustering coefficient of the nodes of a graph. The main result is compactly given by the following theorem.

Theorem 11 *All the neighborhood densities $\varrho(v)$ of a graph G can be approximated in $\mathcal{O}(n \log n)$ time.*

The main algorithm is listed in Algorithm 7.4. We will discuss it in the following.

First note that there is a trivial bound for high degree nodes in otherwise sparse graphs. The neighborhood $G[\Gamma(v)]$ of a node v can not contain more than $m - d(v)$ edges and therefore $\varrho(v) \leq (m - d(v))/\binom{d(v)}{2}$ holds. Hence, the neighborhood density of nodes with $\epsilon > (m - d(v))/\binom{d(v)}{2}$ can be set to zero without any further computational effort for an allowed deviation of ϵ. This happens in line 1 of Algorithm 7.4. In the case of the "Notre Dame WWW" network (see also page 38 in Section 3.2) three nodes fall into this category for $\epsilon = 0.05$. However, these 3 out of 325729 nodes reduce the overall number of edge tests, and thereby also the execution time, to 1/3-rd (for Algorithm 7.4 without the approximation of line 2).

Algorithm 7.4: NEIGHBORHOOD DENSITIES APPROXIMATION

Input: graph $G = (V, E)$ with $d(v) \geq 2$ for all $v \in V$;
subset U of V; parameter k, ϵ;
Output: approximation $\varphi(v)$ of $\varrho(v)$ for all $v \in U$;
forall $v \in U$ **do**
  if $\epsilon > (m - d(v))/\binom{d(v)}{2}$ **then**
1 $\varphi(v) \leftarrow 0$;
  else if $k < \binom{d(v)}{2}$ **then**
2 $\varphi(v) \leftarrow k$-sampling-apx $(\varrho(v))$;
  else
3 $\varphi(v) \leftarrow \varrho(v)$;

The next step is to include the k-sampling, i.e. test for k randomly chosen pairs of neighbors whether they are connected by an edge. This method is integrated with line 2 to Algorithm 7.4. If we want to ensure that each node has the same probability of being approximated correctly, we need to change Equation 7.4 to

$$k = \left\lceil \frac{\beta^2}{2\epsilon^2} \ln\left(2\nu\,|U|\right) \right\rceil, \tag{7.7}$$

from which k can be computed.

For $k \geq \binom{d(v)}{2}$ sampling does not make sense and therefore those remaining nodes are handled with the standard method of testing edges between all neighbors (see algorithm *node-iterator* on in line 3. See page 26 in Section 3.1.2 for a discussion of this method.

Let us note that the approximation in line 2 is essential for the following corollary, the cases handled by the lines line 1 and line 3 give improvements

in the running time by a factor that depends on the graph structure and the choice of k.

Corollary 20 *Algorithm 7.4 can be implemented such that it approximates the neighborhood density of all nodes in $U \subset V$ of a graph G in $\mathcal{O}\left(|U| \frac{\log(\nu|U|)}{\epsilon^2}\right)$ time. For each node v a value $\operatorname{apx}(\varrho(v))$ is computed, which is in $[\varrho(v) - \epsilon, \varrho(v) + \epsilon]$ with probability at least $\frac{\nu-1}{\nu}$.*

Discussion of the Practical Relevance. We tested how small k has to be set to beat the execution time of algorithm *forward* in an implementation. The result for the graph "Movie Actor Network 2004" (see Section 3.2 on page 38) is $k \approx 190$ which does not give terribly good bounds on ϵ. For example we would get $\epsilon \approx 0.2$ for the rather low probability of correctness that exactly one node out of all nodes is not approximated correctly in expectation.

Therefore, we have to draw the following conclusion: it is not beneficial to approximate the neighborhood density for all nodes in the case of sparse graphs. For reasonable bounds on the deviation there exist exact algorithms with at least equivalent execution times, i.e. algorithm *forward*.

This leaves two mentionable cases where Algorithm 7.4 should be used in practice. Algorithm 7.4 can be applied if one is interested in the neighborhood densities of a small subset of V. This is actually the reason, why we listed Algorithm 7.4 such that it processes a subset U of V. In this context note that algorithm *forward* for example can only be applied to process the whole graph. Further, the running time of the approximation algorithm is sub-linear in the case of dense graphs. We performed an experiment where we compared the execution time of *forward* with the approximation algorithm on dense graphs, which were generated $G_{n,m}$ graphs (see Section 3.2 on page 42) with density $\rho = 0.5$. The number of samples was set to $k = 3429$. The break even point between the two algorithms turned out to be at about $m = 2.2 \cdot 10^6$ edges, and the execution time at this point was about 20 seconds. Certainly these numbers are specific to the implementation and the used hardware. However, they show that the graph sizes are not exorbitant when it is favorable to use Algorithm 7.4.

Chapter Conclusion

We have seen various results on the approximation of the clustering coefficient. We highlight the possibility of approximating the clustering coefficient c of a graph in constant time. We have shown that it is feasible to apply this methods in practice. However, especially in the case of approximating the clustering coefficient of all nodes it has to be carefully considered whether it is worth applying an approximative method instead of a fast exact algorithm.

The algorithms might have a further impact on the approximation of the coefficients for very large networks that do not fit into main memory. For example, it is known that the exact number of triangles can not be counted by streaming algorithms with reasonable models [Bar-Yosseff et al., 2002]. However, it can be approximated within some bounds [Bar-Yosseff et al., 2002; Jowhari and Ghodsi, 2005; Buriol et al., 2006]. This is a field for future research.

Chapter 8

Conclusion

We introduced a triangle listing algorithm which is very efficient in practice. Its execution time is competitive to the previously considered practical algorithms. However, our algorithm also achieves optimal running time with respect to the input size of the network. Due to the improved running time, our algorithm performs much better on large graphs with high degree nodes. These types of networks are quite commonly considered in network analysis. Our algorithm achieves execution times in minutes on recent hardware for graphs that fill up all the main memory of an adequately equipped computer. The bottleneck is now rather memory consumption, and techniques that avoid storing the graphs in central memory will be the focus in the future.

We considered a variety of graph classes with respect to triangle listing and triangle counting. We presented triangle listing algorithms that achieve a linear running time in the input (the size of the graph) and the output (the number of triangles). For the case of triangle counting we developed algorithms that have a running time linear in the number of edges or even in the number of nodes. These algorithms do not use techniques that involve high constants in the running times. Therefore, they can be used in practical applications. They also exploit the triangular structure of the graph methodologically, which gives insights into the respective triangular structures.

We reviewed a variety of network analysis related problems that benefit from efficient triangle listing algorithms. We discussed the neighborhood densities and different weighted variants thereof. We gave a unified notation of those weighted versions and a general algorithm to compute them by generic triangle listing. We discussed the clustering coefficient and the related index of transitivity. We gave improved algorithms to compute equivalence classes

of triangular connected nodes and edges. In the context of connectivity by cliques we showed that our triangle listing algorithm can be extended to a 4-clique listing algorithm that achieves acceptable execution times even for larger graphs.

We reviewed the approach to approximate the clustering coefficient of a graph by computing the average of the neighborhood densities of randomly sampled nodes. We have observed that this approach leads to an expected running time at least linear in the number of nodes. We gave an approximation algorithm that achieves a constant running time. It is based on sampling wedges and can be extended to approximate weighted versions of the clustering coefficient. Similar techniques can be used to approximate the neighborhood density of a node. In the case of considering all nodes of a graph, these algorithms have a strong competitor based on the very efficient triangle listing algorithm as discussed above. However, we have shown cases where the approximation nevertheless shows considerable advantages.

We gave a graph generator that is based on preferential attachment. Unlike previously presented approaches, our generator achieves high clustering coefficient values for a wide range of parameters. Additionally, we achieve a nontrivial core structure of the generated networks that increases with the graph size.

Acknowledgments

First of all I would like to thank my supervisor Dorothea Wagner for giving me the opportunity to work with her, for the advice and help all over the years, and for providing the very nice environment in her algorithmics group. I'm very thankful to Ulrik Brandes, who provided many times the help of a second supervisor and particularly let me work with him and his group in Konstanz.

I enjoyed the time with my former office colleague Sabine Cornelsen and my current Christian Pich. Sabine gave me excellent guidance at the beginning of this work. Christian was always very helpful and had a lot of patience with me. Special thanks to Christian and Sabine as well as Michael Baur, Martin Hoefer, Martin Holzer and Svetlana Mansmann for their critical proofreading of parts of this manuscript. I warmly thank Sabine Cornelsen for hinting to and discussing split decomposable graphs, as well as Ulrik Brandes for hinting to and discussing chordal graphs. Many administrative duties would have cost me much more time without the help of Marco Gaertler.

Some work done during recent years did not find its way into this thesis, simply because it did not fit so well to the main topic. Nevertheless I would like to thank the people that were involved with it. Thomas Erlebach and Alexander Hall let me join their interesting work during my time at the ETH Zurich. Michael Baur's help was and is very valuable for the recent work in the CREEN project.

I also enjoyed incredible support and patience from Monika Kulartz during the compilation of this work. I'm very thankful to my parents for their support and trust during my whole education.

Last but not least let me thank again Dorothea Wagner and Ulrik Brandes for taking their time as referees for this thesis.

Zusammenfassung

Verschiedene Methoden in der Netzwerkanalyse basieren auf dem effizienten *Abzählen* oder *Auflisten* aller Dreiecke eines Graphen. Die *Nachbarschaftsdichte* (engl. clustering coefficient) ist eine der zur Zeit populärsten Kenngrößen der Netzwerkanalyse. Ihre schnelle Berechnung hängt von dem effizienten Abzählen aller Dreiecke eines Graphen ab. Diese Arbeit beschäftigt sich mit algorithmischen Problemen in der Netzwerkanalyse, die in engem Bezug zu der Anzahl und der Struktur der Dreiecke in einem gegebenen Netzwerk stehen.

Wir betrachten zunächst Algorithmen für das effiziente Auflisten aller Dreiecke eines Graphen. Im Gegensatz zum Abzählen der Dreiecke ist das Auflisten ein universelleres Werkzeug, da sich beispielsweise aus einem Algorithmus zum Auflisten aller Dreiecke sehr einfach ein Algorithmus zum Abzählen aller Dreiecke ableiten lässt. Beide Probleme wurden in der Vergangenheit weitgehend nur theoretisch betrachtet. Es sei hierzu insbesondere auf [Alon et al., 1997] verwiesen, die den bis dato effizientesten Algorithmus zum Abzählen aller Dreiecke vorstellt. Dieser Algorithmus basiert auf schneller Matrizen-Multiplikation, eine Technik, die zwar weitreichende theoretische Konsequenzen hat, in der Praxis aber nicht verwendet wird. Wir konzentrieren uns jedoch insbesondere auf die praktische Anwendbarkeit von Algorithmen für das Auflisten aller Dreiecke. Die bisher einzige Arbeit, die dieses Problem auch im Hinblick auf praktische Anwendungen beleuchtet ist [Batagelj and Mrvar, 2001]. Der dort betrachtete Algorithmus dient uns als Grundlage für die Entwicklung eines neuen Algorithmus mit besserer Laufzeit. Wir erweitern ihn um einige sehr einfache zusätzliche Operationen und Datenstrukturen und beweisen, dass seine asymptotische Laufzeit bezüglich der Anzahl der Kanten optimal ist. Zudem zeigen wir experimentell, dass dieser Algorithmus im allgemeinen sehr schnelle Ausführungszeiten erreicht und insbesondere auch bei Eingaben mit hohen Knotengraden sehr effektiv ist.

Weiterhin befassen wir uns mit dem Auflisten und Aufzählen aller Dreiecke in *speziellen Graphklassen*. Hierbei steht nicht die praktische Anwendung, sondern die angewandte Methodik im Vordergrund. Wir beginnen mit Graphen, deren *Kernzahl* durch eine Konstante beschränkt ist. Dazu gehören *planare* Graphen und auch Graphen die durch die Methode *lineare Bevorzugung*, besser bekannt unter dem englischen Begriff “linear preferential attachment”, erzeugt werden. Im weiteren behandeln wir *transitiv orientierbare* und *chordale* Graphen. Als letzte Klasse betrachten wir sogenannte *distanzvererbende* Graphen, bei denen in jedem durch einen Pfad induzierten Untergraph die gleichen Knotenabstände wie im Originalgraph gelten. Die vorgestellte Me-

thode zum Auflisten und Zählen der Dreiecke basiert bei distanzvererbenden Graphen allerdings auf einer weiteren, durch Zerlegung charakterisierten Eigenschaft. Falls der Graph in einer impliziten Zerlegung kodiert ist, erreichen wir für das Zählen der Dreiecke eine Laufzeit, die linear in der Anzahl der Knoten ist. Insgesamt erreichen wir für alle erwähnten Graphklassen Laufzeiten, die linear in der Größe von Eingabe und Ausgabe sind. Im Falle des Aufzählens bedeutet dies, dass die Laufzeit linear in der Größe des Graphen ist, und im Falle des Auflistens, dass sie linear in der Größe des Graphen plus der Anzahl der im Graphen vorhanden der Dreiecke ist.

Im Bezug auf Anwendungen in der Netzwerkanalyse beschränken wir uns auf Algorithmen, die als Eingabe eine Liste der Dreiecke des Graphen verwenden. Dies ist zum einen aus den schon besprochenen Inhalten und zum anderen aus der praktischen Wiederverwendbarkeit des immer gleichen Algorithmus zur Auflistung aller Dreiecke motiviert. Als erste Anwendung behandeln wir die Berechnung der Äquivalenzklassen, die durch Wege sich überlappender kurzer Zyklen induziert sind. Dieser Verallgemeinerung von Zusammenhangskomponenten in Graphen wurde in [Batagelj and Zaveršnik, 2003] vorgestellt. Als Spezialfall werden auch Zusammenhangskomponenten, die auf kurzen Zyklen der Länge drei - also Dreiecken beruhen, betrachtet und Algorithmen zur Berechnung der entsprechenden Äquivalenzklassen angegeben. Basierend auf dem generischen Auflisten von Dreiecke können wir Algorithmen mit verbesserter Laufzeit aufzeigen. Ähnlich zu dem Zusammenhangsbegriff, der auf Wegen sich überlappender kurzer Zyklen beruht, lässt sich Zusammenhang auf Basis der Überlappung von kleinen Cliquen definieren. In diesem Kontext zeigen wir die Erweiterbarkeit unseres Algorithmus zum Auflisten aller Dreiecke auf das Auflisten aller k-Cliquen und belegen experimentell dessen Anwendbarkeit, indem wir alle 4-Cliquen eines größeren Netzwerkes auflisten.

Die Dichte des durch die Nachbarn induzierten Graphen eines Knotens bezeichnen wir als *Nachbarschaftsdichte*. Sie ist in der englischsprachigen Literatur als "*clustering coefficient*" eines Knotens bekannt. Algorithmisch lässt sich diese für alle Knoten sehr einfach durch das Abzählen und damit auch Auflisten aller Dreiecke berechnen. Wir konzentrieren uns bei der algorithmischen Besprechung auf den interessanteren Fall des mit Kantengewichten versehenen Graphen. Der Mittelwert der Nachbarschaftsdichten über alle Knoten wird als *Nachbarschaftsdichte* oder *Cluster-Koeffizient* des Graphen selbst bezeichnet. Seine Berechnung selbst ist algorithmisch weniger interessant, aber da in der Literatur wir der Cluster-Koeffizient und der ähnlich definierte *Transitivitäts-Index* manchmal fälschlicherweise synonym verwendet werden, diskutieren wir einige Eigenschaften dieser Indizes, insbesondere

die Unterschiede zwischen den entsprechenden Werte für bestimmte Graphklassen.

Eine wichtige Rolle für das Verständnis real existierender Netzwerke spielen Algorithmen zur Generierung von Graphen mit bestimmten Eigenschaften. Zudem werden Graphgeneratoren zum Testen von Algorithmen benötigt. Ein sehr populäres Modell für die Erzeugung von Graphen ist die sogenannte *lineare Bevorzugung*. Mit ihr lassen sich Graphen erzeugen, bei denen die Verteilung der Knotengrade gute Übereinstimmung mit der entsprechenden Verteilung bei vielen realen Netzwerken hat. Es hat sich allerdings gezeigt, dass der Cluster-Koeffizient eines auf diese Weise erzeugten Graphen deutlich niedriger ist als bei den in realen Netzwerken typischerweise gefundenen Werten. Eine entsprechend angepasster Generator, der angeblich einen hohen Cluster-Koeffizient erzielt, wurde in [Holme and Kim, 2002] vorgestellt. Wir zeigen experimentell, dass dieser Ansatz tatsächlich nur sehr eingeschränkt funktioniert. Wir führen einen neuen Algorithmus zur Generierung von Graphen ein, der diese Einschränkungen nicht zeigt. Weiterhin zeigen wir, dass die von unserem Algorithmus generierten Graphen, im Gegensatz zu den durch die erwähnten Methoden erstellen Graphen, eine nicht triviale *Kernstruktur* aufweisen.

Im letzten Kapitel befassen wir uns mit der näherungsweisen Berechnung des Cluster-Koeffizienten eines Graphen durch Stichprobenerhebung. Eine solche Methode mit einer erwarteten Laufzeit, die mindestens proportional zu der Anzahl der Knoten ist, wird in [Eubank et al., 2004] angewendet. Wir zeigen, dass sich der Cluster-Koeffizient durch Stichprobenerhebung sogar in konstanter Laufzeit approximieren lässt.

Bibliography

James Abello, Panos M. Pardalos, and Mauricio G. C. Resende, editors. *Handbook of massive data sets*. Kluwer Academic Publishers, Norwell, MA, USA, 2002. ISBN 1-4020-0489-3.

Vinay Aggarwal, Anja Feldmann, Marco Gaertler, Robert Görke, and Dorothea Wagner. Analysis of Overlay-Underlay Topology Correlation using Visualization. In *Proc. 5th IADIS International Conference WWW/Internet Geometry*, Murcia, Spain, 5–8 October 2006.

Réka Albert, Albert-László Barabási, and Hawoong Jeong. Mean-field theory for scale-free random networks. *Physica A*, 272:173–187, 1999.

Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

T.L. Austin, R.E. Fagen, W.F. Penney, and J. Riordan. The number of components of random linear graphs. *The Annals of Mathematical Statistics*, 30:747–754, 1959.

Ziv Bar-Yosseff, Ravi Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 623–632, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. ISBN 0-89871-513-X.

Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

Alain Barrat and Martin Weigt. On the properties of small-world network models. *The European Physical Journal B*, 13:547–560, 2000.

Alain Barrat, Marc Barthelemy, Romualdo Pastor-Satorras, and Alessandro Vespignani. The architecture of complex weighted networks. *PROC.NATL.ACAD.SCI.USA*, 101:3747–3752, 2004.

Vladimir Batagelj and Ulrik Brandes. Efficient generation of large random networks. *Physical Review E*, 71(036113), 2005.

Vladimir Batagelj and Andrej Mrvar. Pajek – A program for large network analysis. *Connections*, 21(2):47–57, 1998.

Vladimir Batagelj and Andrej Mrvar. A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Social Networks*, 23:237–243, 2001.

Vladimir Batagelj and Matjaž Zaveršnik. An $\mathcal{O}(m)$ algorithm for cores decomposition of networks. Technical Report Preprint Series, volume 40, number 798, Insitute of Mathematics, Physics and Mechanics, Department of Theoretical Computer Science, University of Ljublana, Ljubljana, Slovania, 2002.

Vladimir Batagelj and Matjaž Zaveršnik. Short cycles connectivity, 2003.

Michael Baur, Ulrik Brandes, Marco Gaertler, and Dorothea Wagner. Drawing the as graph in 2.5 dimensions. In *Proceedings of the 12th International Symposium on Graph Drawing (GD'04)*, Lecture Notes in Computer Science, pages 43–48, 2005.

Béla Bollobás. *Extremal Graph Theory*. Dover Publications, Incorporated, 2004. ISBN 0486435962.

Béla Bollobás. *Modern Graph Theory*, volume 184 of *Graduate Texts in Mathematics*. Springer-Verlag, 1998.

Béla Bollobás and Oliver M. Riordan. Mathematical results on scale-free random graphs. In Stefan Bornholdt and Heinz Georg Schuster, editors, *Handbook of Graphs and Networks: From the Genome to the Internet*, pages 1–34. Wiley-VCH, 2002.

Béla Bollobás, Oliver M. Riordan, Joel Spencer, and Gábor Tusnády. The degree sequence of a scale-free random graph process. *Randoms Structures and Algorithms*, 18:279–290, 2001.

Andreas Brandstädt, Van Bang Le, and Jeremy P. Spinrad. *Graph classes: a survey*. Society for Industrial and Applied Mathematics, Philadelphia , PA , USA, 1999. ISBN 0-89871-432-X.

Tian Bu and Don Towsley. On distinguishing between Internet power law topology generators. In infocom02 infocom02.

Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 253–262, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-318-2.

Qian Chen, Hyunseok Chang, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger. The origin of power laws in Internet topologies revisited. In infocom02 infocom02.

Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, 1985. ISSN 0097-5397.

Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

Derek G. Corneil. Lexicographic breadth first search - a survey. In Juraj Hromkovic, Manfred Nagl, and Bernhard Westfechtel, editors, *WG*, volume 3353 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2004. ISBN 3-540-24132-9.

W.H. Cunningham. Decomposition of directed graphs. *SIAM J. Alg. Disc. Meth.*, 3:214–228, 1982.

Elias Dahlhaus. Efficient parallel and linear time sequential split decomposition (extended abstract). In P. S. Thiagarajan, editor, *FSTTCS*, volume 880 of *Lecture Notes in Computer Science*, pages 171–180. Springer, 1994. ISBN 3-540-58715-2.

Roman Dementiev, Lutz Kettner, and Peter Sanders. Stxxl: Standard template library for xxl data sets. In Gerth Stølting Brodal and Stefano Leonardi, editors, *ESA*, volume 3669 of *Lecture Notes in Computer Science*, pages 640–651. Springer, 2005. ISBN 3-540-29118-0.

Imre Derenyi, Gergely Palla, and Tamas Vicsek. Clique percolation in random networks. *Physical Review Letters*, 94:160202, 2005.

Reinhard Diestel. *Graph Theory*. Graduate Texts in Mathematics. Springer-Verlag, 2nd edition, 2000.

Holger Ebel, Lutz-Ingo Mielsch, and Stefan Bornholdt. Scale-free topology of e-mail networks. *Phys. Rev. E*, 66(3):035103, Sep 2002.

Paul Erdős and Alfred Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

Stephen Eubank, V.S. Anil Kumar, Madhav V. Marathe, Aravind Srinivasan, and Nan Wang. Structural and algorithmic aspects of massive social networks. In *Proceedings of the 14th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'04)*, pages 718–727, 2004.

Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the Internet topology. In *Proceedings of SIGCOMM'99*, 1999.

Delbert R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pacific J. Math.*, 15(3):835–855, 1965.

Marco Gaertler and Maurizio Patrignani. Dynamic analysis of the autonomous system graph. In *IPS 2004 – Inter-Domain Performance and Simulation*, pages 13–24, March 2004.

Horst Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30 (4):1141–1144, 1959.

Martin C. Golumbic. The complexity of comparability graph recognition and coloring. 18(3):199–208, 1977.

Peter Grindrod. Range-dependent random graphs and their application to modeling large small-world proteome datasets. *Physical Review E*, 66(6): 066702, 2002.

Peter L. Hammer and Frédéric Maffray. Completely separable graphs. *Discrete Appl. Math.*, 27(1-2):85–99, 1990. ISSN 0166-218X.

Frank Harary and Helene J. Kommel. Matrix measures for transitivity and balance. *Journal of Mathematical Sociology*, 6:199–210, 1979.

Frank Harary and Herbert H. Paper. Toward a general calculus of phonemic distribution. *Language : Journal of the Linguistic Society of America*, 33: 143–169, 1957.

Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):713–721, 1963.

Petter Holme and Beom Jun Kim. Growing scale-free networks with tunable clustering. *Physical Review E*, 65(026107), 2002.

infocom02. *Proceedings of Infocom'02*, 2002.

Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978.

Hossein Jowhari and Mohammad Ghodsi. New streaming algorithms for counting triangles in graphs. In Lusheng Wang, editor, *COCOON*, volume 3595 of *Lecture Notes in Computer Science*, pages 710–716. Springer, 2005. ISBN 3-540-28061-8.

Gabriela Kalna and Des Higham. Clustering coefficients for weighted networks. volume 3, pages 132–138, 2006.

Ton Kloks, Dieter Kratsch, and Haiko Müller. Finding and counting small induced subgraphs efficiently. *Information Processing Letters*, 74(3–4): 115–121, 2000.

Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D. Sivakumar, Andrew Tomkins, and Eli Upfal. The Web as a graph. In *Proc. 19th ACM SIGACT-SIGMOD-AIGART Symp. Principles of Database Systems, PODS*, pages 1–10. ACM Press, 15–17 2000.

Matthieu Latapy. Theory and practice of triangle problems in very large (sparse (power-law)) graphs, 2006.

Xiafeng Li, Derek Leonard, and Dmitri Loguinov. On reshaping of clustering coefficients in degree-based topology generators. In Stefano Leonardi, editor, *Algorithms and Models for the Web-Graph: Third International Workshop, WAW 2004, Rome, Italy, October 16, 2004, Proceeedings*, volume 3243 of *Lecture Notes in Computer Science*, pages 68–79. Springer-Verlag, 2004.

Luis Lopez-Fernandez, Gregorio Robles, and Jesus M. Gonzalez-Barahona. Applying social network analysis to the information in cvs repositories. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 101–105, 2004.

Ross M. McConnell and Jeremy P. Spinrad. Linear-time transitive orientation. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 19–25, Philadelphia , PA , USA, 1997. Society for Industrial and Applied Mathematics. ISBN 0-89871-390-0.

Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn, editors. *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar,*

March 10-14, 2002], volume 2625 of *Lecture Notes in Computer Science*, 2003. Springer. ISBN 3-540-00883-7.

Stanley Milgram. The small world problem. *Psychology Today*, 1:61, 1967.

Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge Univ. Press, 1995. ISBN 0-521-47465-5.

Mark E. J. Newman. Scientific collaboration networks. i. network construction and fundamental results. *Phys. Rev. E*, 64(1):016131, Jun 2001.

Mark E. J. Newman, Steven H. Strogatz, and Duncan J. Watts. Random graphs with arbitrary degree distributions and their applications. *Phys. Rev. E*, 64(2):026118, Jul 2001.

Mark E. J. Newman, Duncan J. Watts, and Steven H. Strogatz. Random graph models of social networks. *Proceedings of the National Academy of Science of the United States of America*, 99:2566–2572, 2002.

Falk Nicolai. A hypertree characterization of distance-hereditary graphs, 1996.

J. P. Onnela, J. Saramaki, J. Kertesz, and K. Kaski. Intensity and coherence of motifs in weighted complex networks. *Physical Review E*, 71:065103, 2005.

Gergely Palla, Imre Derenyi, Illes Farkas, and Tamas Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435:814, 2005.

Donald J. Rose, Robert E. Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976.

Sheldon M. Ross. *Introduction to Probability Models*. Academic Press, 8th edition, 2003.

Thomas Schank and Dorothea Wagner. Approximating clustering-coefficient and transitivity. Technical Report 2004-9, Universität Karlsruhe, Fakultät für Informatik, 2004.

Thomas Schank and Dorothea Wagner. Approximating clustering coefficient and transitivity. *Journal of Graph Algorithms and Applications*, 9(2):265–275, 2005a.

Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Proceedings on the 4th International Workshop on Experimental and Efficient Algorithms (WEA'05)*, volume 3503 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005b.

Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. Technical report, Universität Karlsruhe, Fakultät für Informatik, 2005c.

Stephen B. Seidman. Network structure and minimum degree. *Social Networks*, 5:269–287, 1983.

Gopalakrishnan Sundaram and Steven S. Skiena. Recognizing small subgraphs. *NETWORKS: Networks: An International Journal*, 25, 1995.

Virginia Vassilevska, Ryan Williams, and Raphael Yuster. Finding the smallest -subgraph in real weighted graphs and related problems. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *ICALP (1)*, volume 4051 of *Lecture Notes in Computer Science*, pages 262–273. Springer, 2006. ISBN 3-540-35904-4.

Duncan J. Watts and Steven H. Strogatz. Collective dynamics of "small-world" networks. *Nature*, 393:440–442, 1998.

Bin Zhang and Steve Horvath. A general framework for weighted gene co-expression network analysis. *Statistical Applications in Genetics and Molecular Biology*, 4, 2005.

Index

Curriculum Vitae

Thomas Schank studied mathematics and physics at the University of Konstanz. He spent the third year of his studies at the Northern Arizona University in Flagstaff/USA. He graduated in 2002 with the degree "Wissenschaftliches Staatsexamen". From October 2002 to March 2003 he participated in the "Predoc Course: Combinatorics, Geometry and Computation" at the ETH-Zürich/Switzerland. In 2000, he started as a Ph.D. student in the Algorithms & Data Structures group in Konstanz. Since 2004, he works as a scientific assistant in the algorithmics group of Prof. Dr. Wagner at the University of Karlsruhe (TH).