Engineering Data Generators for Robust Experimental Evaluations

– Planar Graphs, Artificial Road Networks, and Traffic Information –

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Sascha Meinert

aus Neunkirchen-Seelscheid

Tag der mündlichen Prüfung:	21. Dezember 2011
Erster Gutachter:	Frau Prof. Dr. Dorothea Wagner
Zweiter Gutachter:	Herr Prof. Dr. Michael Kaufmann

Acknowledgments

First of all, I thank my advisor Dorothea Wagner for the opportunity to work in her great research group with its friendly atmosphere, her constant encouragement and support, as well as the freedom to choose and trust in the projects I invested time into. I also want to thank Michael Kaufmann for giving me the opportunity to start my PhD studies in his research group and for reviewing this thesis.

I thank my office mates Frank Schulz, Daniel Delling and Ignaz Rutter (in chronological order) for the enjoyable time and great atmosphere we had. In particular, I thank Frank for the warm welcomes when I sporadically visited Karlsruhe in the beginning. Thank you Daniel for the funny and 'sporty' time we had. I deeply thank Ignaz, who showed me how to step back and consider details in the context of the bigger picture. I also thank you for putting the right pressure on me in the right moments even though I moaned from time to time. It was always a great pleasure to work with you — keep going!

Many thanks to all my colleagues for the enjoyable and 'sporty' time I had in the research group. My favor for delicious dishes is well known and hence I thank the 'Freunde des guten Geschmacks' of our research group with whom I shared this passion; unfortunately, their number decreased over time, which made me visit the canteen more often than I wanted to. I want to thank all those who contributed to my work. Especially, I thank Marcus Krug for our discussions on planar graphs and Reinhard Bauer for sharing his knowledge on route-planning techniques with me. I thank Robert Görke for proofreading every single piece of work of mine and improving my spelling style. Besides going Leif, Bastian Katz and Isabelle Jellen found the time to read this thesis and to give important feedback even on short notice. Thank you so much! The most constant contributor has not been mentioned yet — it made this thesis finally possible: A special thank for its fine and everlasting support goes to our coffee machine.

Throughout my years at the university I organized a lot of events and was involved in several projects that would have been much more challenging without the help of Lilian Beckert, Anja Blancani and Elke Sauer. They made each problem at least an order of magnitude smaller than it was before asking them for help. Thank you for backing me up. I also enjoyed the trust set in me when it came to questions about up-to-date information technology. I thank Bernd Giesinger for our discussions about possible improvements to our technical infrastructure and the associated possibility to spend hundreds and thousands of euros. Also, I thank the numerous student workers who helped me deal with the realization of the GD'06, the final colloquium of SPP'1126 or the ALGO'08 and to perform projects like the GraphDB or the RoutePlanning@KIT.

I thank my parents Gritta and Günter Meinert for supporting me all the time in numerous ways. Finally, I want to thank Nina Hämmerle, who believed in me and encouraged me to finish this thesis: Thank you Nina for your love and patience.

Deutsche Zusammenfassung (German Summary)

Das *Algorithm Engineering* beschreibt ein modernes Verfahren des Algorithmenentwurfs. Dabei ist das erklärte Ziel, nicht ausschließlich theoretische Aspekte, sondern insbesondere auch die praktische Anwendung entwickelter Methoden zu betrachten. Diesem Verfahren liegt ein Kreislauf aus vier Phasen zugrunde. Ausgehend vom *Entwurf* folgt die *theoretische Analyse*. Die darauf folgende *Implementierung* wird dann im *Experiment* ausgewertet. Die experimentelle Auswertung erlaubt dann wieder Rückschlüsse auf den Entwurf; der Kreislauf ist geschlossen.

Elementare Voraussetzung für eine aussagekräftige Bewertung der Praxistauglichkeit in diesem Kreislauf ist das Experiment an Echtweltinstanzen. Allerdings sind solche Instanzen oft nicht oder nur begrenzt verfügbar. Die Gründe hierfür sind vielfältig. So sind Firmen oft nicht bereit, ihre Daten zur Verfügung zu stellen, da Konkurrenten Einblicke und somit Wettbewerbsvorteile gewinnen könnten. Manchmal ist aber auch die Anzahl an Instanzen begrenzt, wie zum Beispiel die Straßennetze kontinentaler Größe. Um dennoch die Effizienz und Robustheit von Algorithmen experimentell zu evaluieren, werden typischerweise künstliche Daten erzeugt. Diese sollten ausgewählte strukturelle Eigenschaften besitzen, die die Algorithmen entsprechend fordern, um signifikante Rückschlüsse auf deren Verhalten im Anwendungsfall zu ermöglichen.

An diesem Punkt setzt diese Arbeit an und leistet Beiträge zur geeigneten Generierung künstlicher Daten, die die Aussagekraft künftiger experimenteller Studien in zwei wichtigen Forschungsgebieten stärken. Dies sind zum einen Algorithmen für planare Graphen, insbesondere im Hinblick auf die sogenannte Festparameter-Algorithmik, und zum anderen Algorithmen zur Routenplanung in Straßennetzen. Zur Generierung künstlicher Eingaben für diese zweite Klasse von Algorithmen werden nicht nur Verfahren zur Erzeugung künstlicher Straßennetze vorgestellt, sondern auch Verfahren zur Generierung systematischer zeitabhängiger Änderungen des Verkehrsflusses in Straßennetzen. Auch die zu diesem Zweck vorgestellten Verfahren folgen einem Kreislauf, ähnlich dem Paradigma des Algorithm Engineering. Ausgehend von Echtweltdaten wird ein realistisches Modell erstellt, beispielsweise anhand der Repräsentation eines Straßennetzwerks. Dann wird ein Generator entworfen, der Daten mit Eigenschaften erzeugt, die dem Modell entsprechen. Dieser wird dann implementiert und es wird ausgewertet, inwieweit die hieraus generierten Daten den Ursprungsdaten ähneln. Der Rückschluss auf notwendige Anpassungen am Generator oder gar dem Modell vervollständigt dann den Kreislauf. Dies führt letztlich zu einem Modell inklusive *Generator*, das Instanzen ähnlich den Echtweltdaten erzeugt.

Daten zur Evaluierung von Algorithmen für planare Graphen

Der Prozess des Algorithm Engineering wird häufig in der *Festparameter*-Algorithmik angewendet. Dabei werden NP-schwere Probleme bezüglich Ihrer Instanzgröße polynomiell und haben dann eine Laufzeit von $f(k) \cdot n^{O(1)}$, wobei f eine berechenbare Funktion ist, deren Laufzeit nur vom Parameter k abhängt. Beispielsweise lässt sich die Frage, ob ein sogenanntes *Vertex Cover* der Größe k existiert, in Laufzeit $O(c^{\sqrt{k}}n)$ entscheiden, wobei c eine Konstante ist und n der Größe der Eingabe entspricht [AFN04]. Die Laufzeit hängt also in erster Linie von der Größe des Parameters k ab, der nicht zu groß werden sollte. Hierbei spielen oft planare Graphen, eine Unterklasse der allgemeinen Graphenklasse, eine wichtige Rolle. Ihre speziellen Eigenschaften ermöglichen es oft, stärkere theoretische Aussagen zu beweisen und effizientere Algorithmen anzugeben, als es für allgemeine Graphen möglich wäre. Konnte für ein Problem ein Festparameter-Algorithmus für planare Graphen angegeben werden, wird oft zusätzlich in einer experimentellen Studie gezeigt, dass die Algorithmen praktikabel sind, also die O-Notation keine hohen Konstanten versteckt. Allerdings hängt diese Evaluierung stark von den gewählten Instanzen ab. Ein Fakt, der in manchen Forschungsarbeiten vernachlässigt wurde und belegbar dazu führte, dass falsche Schlüsse aus experimentellen Studien gezogen wurden. Um dies zu verhindern, müssen planare Graphen repräsentativ generiert werden. Grundlegend betrifft dies die Wahl der zu nutzenden Generatoren. Besitzen Generatoren zudem den Freiheitsgrad, die Anzahl zu generierender Kanten über Parameter extern zu setzen, muss dieser ebenfalls geeignet gewählt werden.

Diese Arbeit untersucht eine Reihe ausgewählter Generatoren zur Erzeugung planarer Graphen. Die Analyse der Generatoren erfolgt auf ihre Vollständigkeit, ihre Laufzeit und Verteilungen von lokalen sowie globalen Maßen. Dazu werden unter anderem die Core-Zahl, der Durchmesser, die Größe des minimalen Dominating Set, die Baumbreite und der Clustering Coefficient betrachtet. Dabei können signifikante Unterschiede zwischen den Generatoren festgestellt werden, die in algorithmischen Studien oft ignoriert wurden. Aus diesen Ergebnissen resultieren Empfehlungen, welche Generatoren in einer experimentellen Studie genutzt werden sollten, um repräsentative Schlussfolgerungen ziehen zu können.

Daten zur Evaluierung von Algorithmen zur kürzeste-Wege-Suche

Die Beschleunigung von Punkt-zu-Punkt-kürzeste-Wege-Anfragen in sehr großen Straßennetzen kann als eine Paradedisziplin des Algorithm Engineering angesehen werden. So ist der klassische Algorithmus von Dijkstra zwar exakt, liefert die Antwort auf Straßennetzen kontinentaler Größe aber erst nach Sekunden. Daher wurden zahlreiche Ansätze entwickelt und iterativ verbessert, so dass die Antwortzeiten heute um das millionenfache beschleunigt sind. Diese Algorithmen wurden darauf zugeschnitten, die speziellen strukturellen Eigenschaften von Straßennetzen auszunutzen. Daher ist nicht zu erwarten, dass sie sich auf beliebigen Netzen ähnlich verhalten.

Künstliche Straßennetze

Forschern stehen zur Evaluierung ihrer Algorithmen im wesentlichen drei teilweise öffentliche Instanzen zur Verfügung: *Tiger/Line* umfasst die USA und wird vom U. S. Census Bureau zur Verfügung gestellt. Den Teilnehmern der 9. Dimacs Challenge (2005) wurde von der Firma *PTV* ein aus 14 europäischen Ländern bestehender Datensatz zugänglich gemacht. Freiwillige Mitarbeiter kartographieren beim offenen Internetprojekt *OpenStreet-*

Map (OSM) mit Hilfe handelsüblicher GPS-Geräte die Welt. Da das Projekt noch relativ jung ist, beschränkt sich die Verfügbarkeit detaillierter Karten auf Teile Europas und der USA. Dies ist auch der Grund, warum die bisherigen Algorithmen hauptsächlich auf die Eigenschaften von Tiger/Line und PTV-Daten hin optimiert und an ihnen evaluiert wurden. Es ist aber zu erwarten, dass weitere kontinentale Netzwerke hinzukommen, die dann vielleicht auch weitere, bislang nicht beobachtete Charakteristika aufweisen. Letztlich werden Algorithmen in der Lage sein müssen, Kürzeste-Wege-Anfragen auf dem globalen Straßennetz zu beantworten.

Um Forschern schon heute zu ermöglichen, ihre Algorithmen an vielfältigen und sehr großen Straßennetzen zu testen, muss ein entsprechender Generator entwickelt werden. Dabei stehen zwei Ziele im Vordergrund. Zum einen sollen die künstlichen Netze den Echtweltnetzen optisch ähnlich sein. Zum anderen sollen Beschleunigungstechniken in beiden Netzwerkarten ähnliches algorithmisches Verhalten aufweisen. Die Analyse der verfügbaren Straßennetze offenbart, dass Unterschiede in den Skalierungseffekten auftreten. Dies gilt sowohl für die Instanzen Tiger/Line und PTV als auch deren kontinentalen Entsprechungen im OSM-Datensatz. Daraus folgt einerseits, dass sich die Repräsentationen desselben Straßennetzes unterscheiden und andererseits, dass Straßennetze je nach geographischer Lage unterschiedliche Charakteristika aufweisen.

Um künstliche Straßennetze zu erzeugen, wurde der bislang ungetestete Vorschlag eines Generators von Abraham et al. [AFGW10] implementiert, wo nötig angepasst und mit den Echtweltinstanzen verglichen. Obwohl mit diesem Generator Instanzen generiert werden konnten, die oft ein ähnliches Verhalten bei Anwendung von Beschleunigungstechniken aufwiesen, entsprach deren Aussehen nicht dem der Echtweltinstanzen.

Daher wurde ein neuer Algorithmus entwickelt, der beide Ziele besser vereinbart. Die zentrale Herausforderung dabei ist eine realistische Erfassung folgender Eigenschaften: Städte weisen eine höhere Knotendichte als ländliche Regionen auf. Städtische Straßen sind relativ kurz und dürfen nur mit geringer Geschwindigkeit befahren werden. Benachbarte Städte sind direkt durch Landstraßen verbunden. Weit entfernte Städte werden zusätzlich durch Autobahnen untereinander angebunden. Dieses Modell der inhärenten Hierarchie von Straßennetzen wurde im neuen Algorithmus durch die rekursive Generierung von Voronoidiagrammen nachgebildet. Durch diesen Ansatz erzeugte Graphen übertreffen Abrahams Algorithmus sowohl in der optischen Ähnlichkeit als auch bezüglich des algorithmischen Verhaltens.

Zusammenfassend ermöglicht dieser Teil der Arbeit Forschern, ihre Algorithmen an mehr, größeren und, bei entsprechender Streuung der Parameter, vielfältigeren Instanzen zu evaluieren. Ferner könnte dieses neue Modell dazu beitragen, eine weitere Methodik zur theoretischen Untersuchung von Beschleunigungstechniken zu entwickeln, ähnlich dem System der *Highway Dimension* von Abraham et al. [AFGW10].

Systematische zeitabhängige Verkehrsflüsse in Straßennetzen

Ein weiterer aktueller Punkt in der Forschung ist die Beschleunigung der Suche nach schnellsten Punkt-zu-Punkt-Verbindungen unter Berücksichtigung von Informationen zum zeitabhängigen Verkehrsfluss. Werden diese nicht berücksichtigt, führt beispielsweise der schnellste Weg von Mannheim nach Karlsruhe immer über die Autobahn. Zieht man aber den zeitabhängigen Verkehrsfluss in Betracht, ist die Nutzung dieser Strecke im Morgen-

und Feierabendverkehr ungünstig, da aufgrund des hohen Verkehrsaufkommens mit deutlich reduzierter Durchschnittsgeschwindigkeit zu rechnen ist. Zu diesen Zeiten lohnt sich der Weg über die Landstraße, da hier eine höhere Durchschnittsgeschwindigkeit erreicht wird und das Ziel schneller erreicht werden kann. Das hier betrachtete Szenario sind also systematische zeitabhängige Verkehrsflüsse auf Strecken, wie sie täglich auftreten. Mangels öffentlich zugänglicher Echtweltdaten sind Forscher in diesem Bereich voll auf die Verwendung künstlicher Daten angewiesen.

Bislang existierte nur ein Ansatz von Nannicini et al. [NDLS08], der weder auf die Eigenschaften von Straßennetzen optimiert ist, noch auf seine Realitätsnähe getestet wurde und trotzdem in zahlreichen Arbeiten Verwendung fand [Del11, KLSV10, BGNS10, DN08]. Darüber hinaus ist fraglich, ob experimentell gewonnene Erkenntnis basierend auf diesen Daten aussagekräftig ist. So stellten Nannicini et al. fest, dass von ihnen getestete Algorithmen unter Inkaufnahme approximativer Lösungen um das zehnfache beschleunigt werden können, und somit eine attraktive Option sind [NDLS08]. Dahingegen stellte Delling in seiner Arbeit fest, dass unter gleichen approximativen Bedingungen, jedoch an Echtweltdaten getestet lediglich eine Beschleunigung um den Faktor 2 erreicht wird [Del09]. Daher wäre in einem Echtweltszenario die vorgeschlagene approximative Variante eine unwahrscheinliche Option.

In diesem Teil der Arbeit schließen wir obige Lücke durch die Entwicklung von Algorithmen, die in der Lage sind, realistischere zeitabhängige Instanzen zu generieren. Um dieses Ziel zu erreichen, analysieren wir einen vertraulichen zeitabhängigen Datensatz der Firma PTV. Hieraus ergibt sich, dass der Verkehrsfluss hauptsächlich innerhalb einer Stadt sowie zwischen der Stadt und einer gewissen Einflussumgebung entsteht. Daher teilen die entwickelten Ansätze das Straßennetz in städtische und ländliche Regionen. Daraus lassen sich Straßenabschnitte ableiten, die wahrscheinlich mit hohem Verkehr belastet sein werden. Dazu benötigen die Algorithmen zusätzliche Daten, die entweder Kanteninformationen zur Straßenkategorie oder die Koordinaten jedes Knotens beinhalten. Somit lassen sich für alle vorgestellten Echtwelt- und künstlich generierten Straßennetze Verkehrsflüsse erzeugen, da mindestens eine dieser Informationen im Netzwerk vorhanden ist. Experimentell werden die durch diese Verfahren generierten Daten mit dem vertraulichen Datensatz von Deutschland auf ihre lokalen, globalen sowie algorithmischen Eigenschaften hin verglichen. Bei geeigneter Parameterwahl werden hierbei Daten erzeugt, die den Echtweltdaten sehr ähnlich sind. Insbesondere gilt dies auch für die oben beanstandeten Verfahren, woraus wir schließen, dass unsere neuen Algorithmen deutlich realistischere Instanzen generieren als es bislang möglich war. Da die Verfahren parametrisiert und randomisiert sind, wird die Generierung vielfältiger Instanzen ermöglicht. In Zukunft können Forscher mit diesen Verfahren realitätsnahe Daten generieren, unabhängig von der Herkunft ihrer Straßennetze, um die Qualität Ihrer Evaluierung zu verbessern und robustere Algorithmen zu entwickeln.

Contents

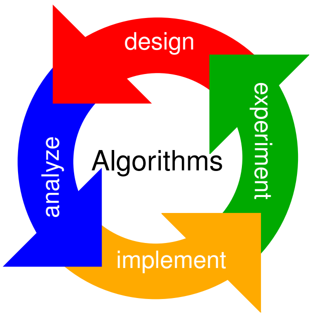
Chapter 1

Introduction and Outline

In *algorithmics*, we usually want to *construct* and *analyze* algorithms, which, in general, allow for solving problems, and in particular, for solving problems in an *efficient* way. For a long period of time, researchers working in this field were split into two fractions — theoreticians and practitioners. Without doubt, the following description of these two is exaggerated, however it does contain a grain of truth.

To solve fundamental problems, theoreticians work with simplified machine models and often consider a simplified version of a problem such that algorithms allow for a mathematical treatment. Algorithms that are analyzed this way admit a guaranteed running time for all possible inputs, which includes worst cases. On the other hand, practitioners consider problems that originate from applications, which typically have the goal to quickly solve real-world problems on real-world data. These problems are solved on physical machines, which are much more complex than classical machine models. They often facilitate parallelism and an inherent memory hierarchy, which enables cache effects. In addition to the complex machine model, real-world problems often consist of chains of problems, and hence, the structure as a whole can hardly be treated mathematically. Possibly, this is unnecessary from a practitioner's point of view, as real-world problems often do not adhere worst cases, and the goal to quickly solve a problem takes priority over theoreticians' goals.

Common practice in both directions led to a gap between both of them. This gap grew over time and for some people led to the, certainly provoking, question whether theoretical insights are practical at all. Hence, to bridge this gap, a growing community postulates and exemplifies a merge of these two fields for now about two decades. The new field forged by this community is known as *Algorithm Engineering* [MS10]. In particular, this process aims at combining traditional theoretical methods with thorough experimental investigations. To this end, the core of the process of Algorithm Engineering is a circular flow through four phases. Starting with a *design* a *theoretical analysis* follows. Consecutive is the *implementation*, which is then evaluated in an *experiment*. The experimental evaluation allows for conclusions on the design, which closes the cycle. In fact, each phase does not only influence the consecutive one but rather all phases are geared together such that they can influence each other. Ideally, both views contribute. Theory brings forward development



Cycle in Algorithm Engineering. Emblem of the DFG Priority Programme 1307.

and intuition of practically efficient algorithms. Experiments influence design and theory in that they might shift the view on the problem or reveal new insights.

Similarly to the practitioner's point of view, often in Algorithm Engineering, a special focus lies on the practicability of algorithms, which is often measured in terms of *efficiency* and *robustness*. A primary requirement to support usefulness in real-world applications is the experiment on real-world data. However, often such instances are not available or only to a limited extent. The reasons for this are manifold. For example, companies often are not willing to share their proprietary data. They fear competitors could gain insights, and thus, an advantage in competition. Another reason is that sometimes the number of instances is limited, e. g., the number of road networks of continental size.

To conduct an experimental evaluation of practicability, typically, *artificial data* is generated. Of course, this has to be done in a meaningful way, e. g., instances of an artificial data set have to admit properties that adequately challenge the algorithms and allow for verifying their robustness as well as flexibility. Only this allows for drawing significant conclusions on an algorithm's behavior in a real-world application, but also further afield. As important as the creation of a challenging data set and a consecutive experimental evaluation of the algorithm is the accurate documentation of the experiment, which includes the description of data sets used. In the case of artificial data sets, a proper description should at least comprise used libraries, employed generation methods and selected parameters. This way, experiments become repeatable, and hence, verifiable.

At this point several publications fail. They either do not report on the experimental setup or at least neglect crucial parts. For instance, publications that verify fixed-parameter algorithms on planar graphs deduce general theoretical results from non-representative data sets [ADN05, ABN06, BHT06]. To exonerate these works, they do partly describe their experimental setup and this allowed for a verification that revealed a flaw in the selection of the generating methods. Keeping this in mind, it is remarkable that many publications completely omit a description of their experimental setup.

To subsume, we have identified two areas of problems that are both centered around facilitating valuable artificial data for the experimental phase within the cycle of Algorithm Engineering. First, artificially generated data has to be accurately described in order to be verifiable such that insights gained become valuable. In particular, we found examples in the field of fixed-parameter algorithmics on planar graphs, where publications failed in the proper selection of test instances, and hence, conclusions drawn therein are flawed to a certain extent. Second, in some fields of research only few real-world instances exist, which make it difficult to test algorithms for robustness and flexibility. In particular, this is the case in the field of route planning.

Exactly these two aspects of facilitating valuable artificial data for the experimental phase in the process of Algorithm Engineering inspired this thesis. Our goal is to offer solutions for both of the problems. To this end, the work conducted in this thesis follows the principles of and is guided by the cycle of Algorithm Engineering. In this thesis, we tackle each of the problems in a seperate part as follows.

In the first part, we are interested in providing reasonable artificial data for an experimental analysis of algorithms that work on planar graphs. To this end, we analyze several existing algorithms, which partly originate from libraries, such that experimenters can more easily compile a reasonable data set. In particular, alongside with a general analysis in terms of theory and an experimental evaluation of basic graph properties, we specially focus on experiments conducted in fixed-parameter algorithmics.

In the second part, we aim at enlarging the set of instances that are typically used to perform experimental analysis of route-planning algorithms for both important scenarios, i. e., static and dynamic scenarios. In the static scenario, segments of a road network have associated constant weights, e. g., the time to traverse a road segment or its Euclidean length. In the dynamic scenario, weights of road segments might change over the course of a day. As route planning is a real-world application, we also want the artificial data to be realistic with respect to algorithmical and possibly visual properties. To this end, we start with the analysis of an instance of real-world data, from which we derive a realistic model. Based on this, we develop a generator that is capable of creating artificial data according to our model. Then, we implement the generator and experimentally evaluate to what extent the artificial data corresponds with the source data. This analysis allows for conclusions on possibly necessary modifications of the generator or even the model. With this last step, we have closed the cycle of Algorithm Engineering. Finally, we obtain a model including a generator, which allows for generating artificial instances that admit properties similar to real-world instances.

Organization of this work

This work is organized as follows:

In Chapter 2, we briefly repeat concepts and algorithms we assume the reader to be familiar with. In particular, this comprises basics on graph theory, complexity theory, algorithms and route planning concepts.

In Chapter 3, we study several planar graph generators that were selected according to availability, theoretical interest, ease of implementation and efficiency. The reasons to do so are twofold. On the one hand, planar graph generators exists that exhibit good theoretical properties but, from a practical point of view, are only of limited use. On the other hand, several practical planar graph generators exist, however, some exhibit a considerable bias, which may significantly affect outcomes of experiments. Hence, we aim at a classification of the generators that allows for their reasonable selection such that conducted experiments result in reliable insights. To this end, we analyze the algorithms with respect to both, theoretical and practical aspects. In particular, we derive some new aspects of the studied algorithms by their theoretical analysis. On the other hand, we experimentally analyze their implementations and conclude on their practicability. The experimental evaluation comprises network analysis by means of graph properties and algorithmic behavior, in particular kernelization of fixed-parameter tractable problems. By this, we will see the major influence of instance selection on algorithmic behavior. In addition, we derive insights on the (un)likeliness of events occurring in practice that should theoretically be possible. Altogether, the performed study in this chapter helps experimenters to carefully select sets of planar graphs that allow for a meaningful interpretation of their results. Part of this chapter has previously been published [MW11a], and part of the work presented here is unpublished, which is joint work with Marcus Krug and Ignaz Rutter.

In Chapter 4, we focus on the generation of realistic instances of road networks, which can be employed for experiments in static route planning. We measure the realism of artificial instances compared to real-world instances with respect to three characteristics. In particular, these are structural properties, algorithmic behavior and visual similarity.

First, we review the only existing model with a generator, which until now has neither been implemented nor experimentally tested. It turns out that the existing generator does not fulfill all of our requirements on realistic artificial instances, which mainly results from the missing modeling of the Steiner property. Hence, we propose an alternative model that incorporates properties similar to the aforementioned model and, additionally, the Steiner property. The generator derived from the newly proposed model relies on the recursive computation of Voronoi-diagrams, and allows for generating graphs that are much more realistic with respect to their visual appearance. A scalability test shows that both generators perform well with respect to memory consumption and running time. To also assess similarity of structural properties and algorithmic behavior, we perform an experimental analysis and compare the real-world road networks to the synthetic road networks. It turns out that the real-world instances differ with respect to structural properties as well as algorithmic behavior. With respect to this variance, both generators approximate the real-world instances quite well at which our newly proposed generator gains slight advantages. However, our newly proposed approach behaves superior with respect to visual similarity. In summary, this chapter provides ready-to-use generators that are capable of generating realistic artificial instances of road networks, whose properties can be steered in order to challenge algorithms to test. Part of this chapter has previously been published [BKMW10]. The work presented here is joint work with Reinhard Bauer and Marcus Krug.

In Chapter 5, we develop algorithms capable of generating realistic, artificial, daily traffic information on top of road networks of continental size. In our context, traffic information refers to the daily expected delay for distinct times of a day in a static scenario, i. e., unexpected events like accidents, holiday traffic or weather influence are not considered. These instances are typically used to assess time-dependent shortest-paths algorithms. We rate artificial instances as realistic compared to a real-world data set, if they admit similar local as well as global statistical properties, and additionally, a similar behavior of time-dependent shortest-path algorithms. Although, to the best of our knowledge, no free data set is available to researchers, we have access to a confidential real-world data set. By its analysis, we derive a realistic model of how traffic flow emerges in large networks. Based on this model, we develop algorithms that utilize either road categories or coordinates to enrich a given road network with artificial traffic information. This chapter provides ready-to-use generators that allow for the generation of realistic daily traffic information in road networks that may originate from manifold sources like commercial, open source or artificial ones. This remedies the situation that no realistic data sets are publicly available. Part of this chapter has previously been published [MW11b].

In Chapter 6, we conclude this work with a summary.

Chapter 2

Preliminaries

This chapter deals with fundamental concepts used later in this work. These will be briefly repeated in addition to references for a more thorough treatment.

We expect the reader to be familiar with basic mathematical concepts. By $\mathbb{Z}$ and $\mathbb{R}$ we denote the set of integers and reals, respectively. We use $\mathbb{N} := \{x \in \mathbb{Z} \mid x \geq 0\}$ to denote the natural numbers including zero. To denote positive subsets we use $\mathbb{R}_+ := \{x \in \mathbb{R} \mid x > 0\}$; the symbol $\mathbb{N}_+$ is defined analogously.

First, we introduce our notation of graphs as well as several related graph properties. This is extended by a side note on planar graphs and some of their properties. In addition, we present some concepts of network analysis. Second, we give a brief introduction to complexity, in particular, fixed parameter tractability. Third, we describe the concepts of randomness used in our generator engineering. Fourth, we present data structures and basic algorithms which are used later in this work. Fifth and last, we briefly introduce models, concepts and algorithms for route planning in road-networks and their acceleration.

2.1 Graphs

Often when people are talking about problems at hand, they have their view on the problem and on the data in mind. If this data comes in the flavor of relations between objects, we speak of a *network*. Usually, a network denotes an instance that stems from a real-world application, e. g., protein-protein interactions in metabolic networks or road networks used in route planning. However, to avoid misunderstandings that may occur because of the many origins and ways to observe a real-world instance, we need a common language. Hence, we usually employ *graphs* to mathematically abstract from the view on the networks. In this formalization a *vertex* represents an object, and an *edge* represents a relation between two objects. In some literature vertices are also called *nodes* and edges are also called *arcs*. We do not make a distinction and, hence, intermix the naming of a graph's elements. Additionally, we will use the terms graph and network interchangeably, since the context clarifies whether we are speaking of a real-world instance or its mathematical formalization. Next, we give a brief introduction to the notation of graphs, for a thorough treatment of graph theory we refer the reader to the book by Diestel [Die00].

2.1.1 Basic Definitions

We use the common notation of a graph G as a tuple $G = (V, E)$, where V represents a finite set of *vertices* and E represents a finite set of *edges*. An edge e is an unordered pair $e = \{u, v\}$, which connects two vertices $u, v \in V$; the edge e is said to be *incident* to u and v. We call the vertices u and v of an edge $e = \{u, v\}$ the *endpoints* of e. We say a vertex $u \in V$ is *adjacent* to a vertex $v \in V$ if they are connected by an edge $\{u, v\} \in E$. We denote the cardinality $|V|$ of the set V of vertices by $n := |V|$, and the cardinality $|E|$ of the set E of edges by $m := |E|$. Note that we refer to the cardinality of the vertex set if we speak of the size of a graph. We call a graph *undirected* if the edges are unordered pairs $\{u, v\}$ of nodes $u, v \in V$, which denotes a symmetric relationship. In contrast, if the edges are ordered pairs (u, v), we call a graph *directed*. A directed edge $e = (u, v)$ is a non-symmetric relationship. Hence, a directed edge has as endpoints a distinct *source* and a distinct *target*. An undirected graph can be transformed into a directed graph by substituting each undirected edge $\{u, v\}$ by the two edges (u, v) and (v, u). A graph is called *simple* if the set of edges does not contain *selfloops*, i. e., $\{u, u\} \notin E$ for all nodes $u \in V$. Note that the definition of a set forbids a graph to contain *multiple edges*, i. e., a graph contains at most one edge $\{u, v\}$ for each pair of distinct endpoints $u, v \in V$, $u \neq v$. A graph with a function that maps vertices to labels is called a *vertex-labeled graph* and a graph with a function that maps edges to labels is called an *edge-labeled graph*, respectively. An *unlabeled* graph can be vertex-labeled with unique labels $1, \ldots, n$, where n is the number of vertices of the graph, in $n!$ ways, which simply corresponds to all permutations of the labels. An *isomorphism* of two unlabeled graphs $G(V, E)$ and $H(V', E')$ is a bijection $f\colon V \longrightarrow V'$ such that $\{u, v\} \in E$ if and only if $\{f(u), f(v)\} \in E'$. If an isomorphism exists between two graphs, we call them *isomorphic*.

For the rest of this work, we consider graphs to be vertex-labeled by distinct labels, simple and undirected if not mentioned otherwise. Most of the following definitions are extendible to directed graphs. Note that we use *maximal* (*minimal*) to specify that by adding (deleting) another element some property is violated. The cardinality of the edge set of such a graph is bound by

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

A graph is *complete* if equality holds for Equation 2.1. We also call a complete graph a *clique*. The shorthand notation K_i is used to denote the complete graph on i vertices, e. g., K_5 represents the complete graph of size 5, which has ten edges. A complete *bipartite* graph $G_{i,j} = (V_i \cup V_j, E)$ consists of two disjoint sets of vertices V_i and V_j, where each vertex of the set V_i is connected to every vertex of the set V_j, i. e., $E = V_i \times V_j$. The shorthand notation for a complete bipartite graph is $K_{i,j}$, where i and j denote the size of each of the node sets.

The *density* of an undirected graph is defined by $\rho(G) = m/\binom{n}{2}$. We further specify a graph depending on the cardinality of its edge set. Therefore, we call a family of graphs *sparse* if $m \leq c{\cdot}n$ and *dense* if $m \geq c{\cdot}n^2$, with some constant $c \in \mathbb{R}$. The *neighborhood* $\Gamma(v)$ of a vertex v consists of the set of vertices adjacent to v and is defined by $\Gamma(v) = \{u \in V \mid \{u, v\} \in E\}$. The *degree* $\deg(v)$ of a node v is the size of its neighborhood $\deg(v) = |\Gamma(v)|$. The *degree sequence* of a graph $G = (V, E)$ is a list of the degrees of its vertices sorted in non-increasing order. The *maximum degree* and *minimum degree* of a graph G are defined by $\deg_{\max}(G) = \max\{\deg(v) \mid v \in V\}$ and $\deg_{\min}(G) = \min\{\deg(v) \mid v \in V\}$, respectively. The equation $\sum_{v \in V} \deg(v) = 2m$ is known as the *Handshake Lemma*. It states that each edge is accounted twice when the degrees of the nodes are summed up. The *average degree* of a graph G is defined by $\text{avg}(G) = \sum_{v \in V} \deg(v)/n = 2m/n$. If the minimal degree of a graph equals its maximal degree, it is called *regular*. It is *k-regular* if every vertex has degree k.

Subgraphs, Operations and Constructs. A graph $H = (V', E')$ is a *subgraph* of a graph $G = (V, E)$ if its vertex set and edge set are subsets of the G, i. e., $V' \subseteq V$ and $E' \subseteq E$. The *vertex-induced subgraph* $H = (V', E')$ of a graph $G = (V, E)$ consists of a subset of vertices $V' \subseteq V$, and the subset of edges $E' = \{\{u, v\} \in E \mid u, v \in V'\}$.

An *edge subdivision* is an operation on a graph that splits an edge by the insertion of a node. More precisely, the edge $\{u, v\}$ is deleted, a new vertex w is added, which is then connected to the former endpoints of $\{u, v\}$ by the insertion of two new edges $\{u, w\}$ and $\{w, v\}$. A *subdivision* of a graph is obtained by a repeated application of edge subdivisions. A subdivision of an edge can be reverted by the operation *edge contraction*. An edge contraction is performed on an edge $\{u, v\}$ of a graph by deleting the edge $\{u, v\}$ and, simultaneously, merging the two formerly connected nodes into a single node $u' = u \oplus v$ such that $\Gamma(u') = \{(\Gamma(u) \cup \Gamma(v)) \setminus \{u, v\}\}$. However, note that an edge contraction is more powerful than a subdivision as not every edge contraction can be reversed by an edge subdivision. A graph $H = (V', E')$ is a *minor* of the graph $G = (V, E)$ if H can be obtained by a sequence of edge deletions, deletions of isolated vertices and edge contractions on the graph G.

A *partition* of a graph $G = (V, E)$ is a grouping of the vertex set V into disjoint subsets $P_1, \ldots, P_k$ such that $P_i \subseteq V$ with $1 \leq i \leq k$, $P_i \cap P_j = \emptyset$ for each $1 \leq i, j \leq k$ with $i \neq j$, and $P_1 \cup P_2 \cup \ldots \cup P_k = V$.

The *k-core* of a graph is the maximal vertex-induced subgraph in which each vertex has at least degree k. Note that the $(k + 1)$-core always is a subgraph of the k-core, and hence, can be computed iteratively for $k = 1, \ldots, \deg_{\max}(G)$. By the *core decomposition* of a graph, we denote the set of k-cores of a graph. We call the vertices not being part of the k-core but part of the $(k - 1)$-core the *(k-1)-shell nodes*. The vertices of a k-shell have *coreness k*.

Paths, Connectivity and Weights. A *walk* between two vertices v_1 and v_k in a graph G is an alternating sequence of nodes and edges of G in which each node is adjacent to its predecessor and its successor except the starting node v_1 and the end node v_k, which only have a successor and a predecessor, respectively. More formally, by a walk W we denote a sequence $W = v_1, e_1, v_2, e_2, \ldots, e_k, v_k$ with $e_i = \{v_i, v_{i+1}\}$ for all $1 \leq i < k$. A *path* is a walk that contains each edge at most once, and in shorthand, we omit the specific edges and write $P = (v_1, v_2, \ldots, v_k)$. If additionally, the vertices of a path are visited only once, we say a path is *simple*. A path $P = (v_1, \ldots, v_k)$ in a graph is called a *cycle*, if the endpoint v_k of the path equals its start point v_1. A cycle $C = (v_1, \ldots, v_k)$ is called *simple*, if the path $P = (v_1, \ldots, v_{k-1})$ is simple. A cycle that is simple and consists of three vertices is called a *triangle*.

A *connected component* of a graph $G = (V, E)$ denotes a maximal set of vertices $S \subseteq V$ such that a path between every pair of vertices $\{u, v\} \in S$ exists. A *connected* graph consists of a single connected component. A subgraph $H = (V', E')$ of a graph $G = (V, E)$ is *spanning* if $V' = V$ such that H is connected. If a connected graph is cycle-free, we call it a *tree*. We call the vertices of degree 1 the *leaves* of a tree, and non-leaves the *internal vertices* of a tree. In some applications it is important to designate a single vertex of a tree, which we call the *root*. Accordingly, a tree that contains a root is a *rooted tree*. A tree $T = (V, E')$ that is a spanning subgraph of a graph $G = (V, E)$ is called a *spanning tree*.

Often, in networks relations between objects come with an associated value, e. g., costs or lengths. We call the corresponding formalization a *weighted graph*. A weighted graph $G = (V, E, \omega)$ extends the definition of a graph given above by an *edge weight* function $\omega\colon E \longrightarrow \mathcal{D}$ with some domain $\mathcal{D}$. By $\omega(u, v)$, we refer to the weight of an edge $\{u, v\}$, which is a shorthand notation for $\omega(\{u, v\})$. In our work, the domain $\mathcal{D}$ is, in most cases, $\mathbb{R}_+$.

Given a weighted graph $G = (V, E, \omega)$ with an edge-labeling function $\omega\colon E \longrightarrow \mathcal{D}$ the *length* of a path $P(v_1, v_k) = (v_1, \ldots, v_k)$ is $\operatorname{len}(P(v_1, v_k)) = \sum_{i=1}^{k-1} \omega(v_i, v_{i+1})$. We call a path of minimum length between a starting node s and a target node t a *shortest s-t path* or in shorthand *shortest path*. The *distance* $d(s, t)$ between two nodes s and t is the length of a shortest path between them. Usually, if no such path exists, the distance is defined to be ∞. We call the maximum distance from a vertex v to all other vertices in a graph the *eccentricity* of v. The *diameter* of a graph G is the length of a longest shortest path between all pairs of vertices in G, or, accordingly, the maximum eccentricity over all vertices of G. A *shortest-path tree* $\mathcal{T}(v) = (V, E')$ of a node $v \in V$ in a connected graph $G = (V, E)$ is a spanning tree rooted at v that contains a shortest path in G for every pair of vertices v, t with $t \in V \setminus \{v\}$. Note that a shortest-path tree is unique if and only if the shortest path for every pair of vertices v, t with $t \in V \setminus \{v\}$ is unique. A *t-spanner* of a graph $G = (V, E)$ is a spanning subgraph $S = (V, E')$ such that the distance between any pair of vertices is at most t times their distance in G.

Sometimes, we work with weighted graphs but want to refer to the definition of distance in the context of unweighted graphs, i. e., we count the edges used. We denote this by speaking of the *graph theoretic distance* or, in short, *hop distance*.

In a rooted tree, the *depth* d of a node v is the graph-theoretic length of the path from the root to v. The *height* h of a tree is the maximum depth of all its leaves.

2.1.2 Planar Graphs

So far we looked at the properties of graphs that come with their combinatorial representation. Next, we see a subclass of general graphs, which can be defined by the way the graphs can be drawn. A *drawing* of a graph $G = (V, E)$ is a mapping of its vertex set V onto distinct points in the plane and its edge set E onto simple *Jordan curves* that connect the endpoints of each edge $e \in E$. If a pair of edges intersects in a drawing it is called a *crossing*. Graphs are *embeddable* on a surface if its vertices and edges can be arranged in a way such that no crossing exists, except possibly at common nodes. The graphs that are embeddable on the Euclidean plane are called *planar graphs*. Such an embedded graph is called a *plane graph*. The edges of a connected finite planar graph split the plane into disjoint connected regions, each bounded by a set of edges such that the size of the region is maximal. These regions of maximal size are called *faces*, i. e., a face is not further subdivided by an edge of the graph. Each face is bounded by a cycle of length at least 3. The single unbounded region is called the *outer* face. If f denotes the number of faces of a finite connected plane graph *Euler's formula* states $n - m + f = 2$, which bounds the number of edges in a planar graph to $m \leq 3n - 6$ and the number of faces to $f \leq 2n - 4$. A graph is called *maximally planar* if it is planar but adding any edge would destroy the planarity property. In a maximal planar graph all faces are bound by cycles of length three — triangles, which leads to the name of a *triangulated graph*.

Planar graphs can also be defined by a combinatorial characterization. The *Kuratowski* theorem states that a finite graph is planar if and only if it does not contain a subgraph that is a subdivision of the graphs K_5 or $K_{3,3}$ [Kur30].

2.1.3 Network Analysis / Basic Graph Properties

Later in our work, we want to reveal structural information of networks and compare them by means of basic graph properties. In network analysis, we are interested in *local* as well as *global* graph properties to grasp underlying structures. Often, these graph properties are represented by a *single value*, e. g., the average degree. However, for many properties a single value is not detailed enough to describe a specific graph property, and we need to describe *multiple values*. Therefore, we are interested in the *distribution* of specific values within the network. For example, we examine the *degree distribution*, which refers to a list of the number of nodes with certain degree. We analyze the networks by some of the already introduced properties, e. g., average degree, diameter or core decomposition of a graph, and some more properties, which we briefly introduce now. For a thorough treatment, we refer the interested reader to a book on network analysis [BE05].

Shannon Entropy In network analysis, we are usually interested in single values of graph properties as these can directly be compared with each other. However, some of the tested properties consists of more than one element, e. g., the degree distribution. Hence, it would be best to aggregate the elements in a single value possibly without loosing too much information. However, this is a difficult task, e. g., building the average is a lossy operation.

To this end, we aggregate the multiple values of an experiment into a single value using the *Shannon entropy* [Sha01], which is known from information theory, in the following way. First, we estimate the probabilities of the outcomes $\hat{p}_i$ using the *maximum-likelihood* estimation, which is defined by $\hat{p}_i = n_i/N$ with n_i the number of occurrences of outcome i and N the total number of observations. Second, using the computed probability distribution we can compute the Shannon entropy H_b of each outcome by $H_b = -\sum_{i=1}^{k} \hat{p}_i \log_b(\hat{p}_i)$ with k the number of distinct outcomes and b the base of the logarithm. Depending on the base of the logarithm two different information are obtained. The *bit entropy* H_2 is computed using the base $b = 2$ which determines how many bits per entry on average are necessary to encode the information. The *normalized entropy* H_l to the base $b = l$, where l is the number of distinct elements within the single outcome, informs on the amount of redundancy within the information.

Clustering Coefficient and Transitivity Networks often contain groups of elements that have an increased number of relations among each other compared to others, e. g., communities in social networks. In network analysis, we are interested whether a network exhibits such structures. The application of finding groups in networks is called *clustering* and a group of elements is called a *cluster*.

We already saw that graphs can be sparse or dense. In a more local view, we are interested how dense the neighborhood of a vertex is or, in other words, to which extent a vertex clusters. A measure to grasp this value of a vertex is the *clustering coefficient*. It counts how many pairs of neighbors of a vertex are adjacent to each other. More formally, the clustering coefficient can be defined using triangles and triples [SW05]. Given a graph $G = (V, E)$, the number of triangles of a node v is defined as $\delta(v) = |\{\{u, w\} \in E : \{v, u\} \in E \text{ and } \{v, w\} \in E\}|$. A *triple* τ at a node v is a path of length two for which v is the center node. Then, the number of triples of a node v is $\tau(v) = \binom{\deg(v)}{2} = (\deg(v)^2 - \deg(v))/2$. For nodes v with degree $\deg(v) \geq 2$ the clustering coefficient is defined as $c(v) = \delta(v)/\tau(v)$.

Usually, we are interested in a single measure for a graph. Hence, the clustering coefficient of the vertices of a graph is condensed into a single value. To this end, we compute the average over all nodes, i. e., the clustering coefficient of a graph is defined as

$$\zeta(G) = \frac{1}{\hat{n}} \cdot \sum_{v \in \hat{V}} c(v) \ ,$$

where $\hat{V}$ is the set of nodes v with $\deg(v) \geq 2$ and $\hat{n}$ the size of the set $\hat{V}$.

Related to the clustering coefficient of a graph G is the *transitivity* $\mathcal{T}$, which is also defined on the number of triangles and triples in a graph. The transitivity is defined by

$$\mathcal{T}(G) := \frac{\sum_{v \in G}(3 \cdot \delta(v))}{\sum_{v \in G} \tau(v)} \ .$$

Thus, it captures the relation of triangles to triples in a global view, whereas the clustering coefficient captures their local view.

Modularity So far, we examined to which extent the neighborhood of a single vertex is clustered. In a more global view, we want to identify communities of size larger than the neighborhood of a single vertex. To this end, our goal is to partition the graph into clusters such that the number of edges within each cluster is large and the number of edges between vertices of different clusters is small. More formally, a clustering $\mathcal{C}(G) = (C_1, \ldots, C_k)$ of a graph $G = (V, E)$ is a partition of its vertex set into k disjoint sets of vertices C_i such that $C_i \cap C_j = \emptyset$ for $1 \leq i, j \leq k$ with $i \neq j$ and $C_1 \cup \ldots \cup C_k = V$. Note that as a shorthand we write $\mathcal{C}$ instead of $\mathcal{C}(G)$. We quantify a division of a graph into clusters using the benefit function *modularity*, which was proposed by Girvan and Newman as a quality index for clusterings [NG04]. The modularity of a graph is the number of edges within dense regions minus the expected number of edges that would exist if the edges are placed randomly between the nodes. Using the notation of [Gae05], the modularity $\operatorname{mod}(\mathcal{C})$ of a clustering of a graph is defined by

$$\operatorname{mod}(\mathcal{C}) := \sum_{C_i \in \mathcal{C}} \left[\frac{|E(C_i)|}{m} - \left(\frac{\sum_{v \in C_i} \deg(v)}{2m} \right)^2 \right] \ .$$

If a clustering exhibits a high modularity, the nodes of each cluster are highly connected with each other whereas the nodes of pairwise distinct clusters are only loosely linked, and additionally, the size of the clusters is balanced.

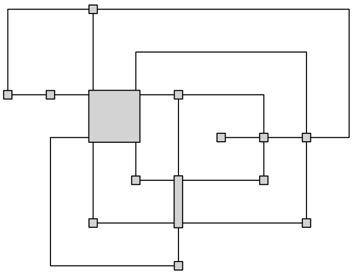


Figure 2.1: An orthogonal drawing of a graph. Nodes are drawn on a grid, edges are drawn along the grid, possibly alternating between horizontal and vertical segments. Each switch of a direction is called a bend. To allow nodes of degree larger than four, they are allowed to occupy multiple adjacent coordinates on the grid.

Usually, clustering algorithms are applied to detect groups within networks. However, in our work, we solely use modularity as a network index as we do not want to explicitly compute good clusterings. Unfortunately, optimizing the modularity of a network is NP-hard [BDG$^+$08]. Hence, we employ the standard greedy algorithm of Newman [New04] to compute a clustering with high modularity. By the modularity of a graph we refer to the clustering of a graph the greedy algorithm computes.

Orthogonal Drawings Another application of graphs is the visualization of the inherent relational information of a network [DBETT98]. One way to realize a clear scheme of the data is to use *orthogonal drawings*, which have the property that all edges solely consist of chains of *horizontal* and *vertical* segments. Figure 2.1 shows an example of an orthogonal drawing of a graph. In such a drawing, the vertices are placed on integer coordinates and the edges are drawn along the grid. Each switch from a vertical to a horizontal segment of an edge and vice versa is called a *bend*. If a vertex has degree larger than 4, we cannot draw edges along the grid without overlaps, which would reduce the benefit of readability. Hence, we allow the nodes to occupy multiple adjacent integer coordinates by drawing them as rectangles. By the *occupied area* of a drawing, we refer to the minimal bounding box that contains the whole drawing. Usually, the quality of a drawing is measured in terms of visual quality, which can be quantified in terms of several measures. To allow for an automated computation, we focus on the total number of bends and the occupied area of a drawing.

2.2 Complexity

We will use some terms from computational complexity theory, which we will now introduce briefly. Based on this, we present basic complexity classes, and finally, two classical problems that are difficult to solve. For a thorough treatment of computational complexity we refer the reader to a book by Garey and Johnson [GJ79].

2.2.1 Basic Terms

Often in algorithmics, we are given a computational problem consisting of an *instance* and a *question* to answer. The problems we encounter come in two flavors: *decision problems* and *optimization problems*. Assume the instances are 'reasonably' encoded using some finite alphabet. Then, by n we denote the size of an instance I, i. e., the number of symbols needed to encode I.

Decision Problem In a *decision problem* $\Pi = (D_\Pi, Y_\Pi)$, we are given a set of *instances* D_Π and a subset of *yes-instances* $Y_\Pi \subseteq D_\Pi$. The instances in $\mathcal{D}_\Pi \setminus Y_\Pi$ are called *no-instances*. An algorithm $\mathcal{A}$ solving a decision problem Π has to find the correct answer for all instances $I \in \Pi$, i. e., whether I is a yes-instance or not.

Optimization Problem In an *optimization problem*, we want to find a solution that either *maximizes* or *minimizes* an *objective function*. More formally, an optimization problem Π is a quadruple $\Pi = (\mathcal{I}, F, w, g)$, where

- $\mathcal{I}$ is a set of *instances*
- F is a mapping that assigns each instance $I \in \mathcal{I}$ a set of feasible solutions $F(I)$
- $w\colon (I, y) \longrightarrow \mathbb{R}_+$ with $I \in \mathcal{I}$ and $y \in F(I)$, is an objective function
- $g \in \{\max, \min\}$, specifies whether to minimize or maximize the problem

An *optimal solution* of an instance $I \in \mathcal{I}$ is a feasible solution $y \in F(I)$ that optimizes the objective function $w(I, y)$ with respect to g. If y is such an optimal solution, we denote $w(I, y)$ the *optimal value*.

Optimization problems can be cast into decision problems by introducing an additional parameter k. Then, an instance $I \in \mathcal{I}$ of a maximization problem (minimization problem) is a *yes* instance if feasible solutions $y \in F(I)$ exist such that $w(I, y) \geq k$ $(w(I, y) \ \leq \ k)$.

Asymptotic Analysis We use Landau's so called *Big-O Notation* to describe the asymptotic running time of algorithms. The symbols O, Ω, Θ denote sets of functions that exhibit specific upper or lower bounded asymptotic behavior. Let f and g be two functions $f, g\colon \mathbb{N} \longrightarrow \mathbb{R}$. We say $O(g)$ is an *upper bound* on f and write $f \in O(g)$ if and only if there exists $c \in \mathbb{R}_+$ and $n_o \in \mathbb{N}$ such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. Informally, the function f grows at most as fast as g. The opposite behavior, i. e., f grows at least as fast as g, is called a *lower bound* and is denoted by $f \in \Omega(g)$. More formally, we say $f \in \Omega(g)$ if and only if there exists $c \in \mathbb{R}_+$ and $n_o \in \mathbb{N}$ such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0$. In the case the growth of a function f is bounded from above and below we denote this by $f \in \Theta(g) = O(g) \cap \Omega(g)$, i. e., f grows equivalently to g. When we only want to roughly sketch the asymptotic running times of algorithms, we omit the Landau notations. Instead, we indicate the computation time that is needed to solve the problem by the following two classes. If an algorithm has a running time bounded by a polynomial in the input size, i. e., $f(n) \in O(n^k)$ for $k \in \mathbb{R}$, we call it a *polynomial-time algorithm*. In the literature, these algorithms are also said to be *efficient*. However, note that efficiency does not necessarily come with practicality, in particular, for large k it is unlikely to compute the answer if the instance exceeds a certain size. We use the term *exponential-time algorithm*, if its running time grows at least *exponentially* in its input size, i. e., $f(n) \in \Omega(k^n)$ for some fixed $k \in \mathbb{R}_{>1}$, and additionally, if it is bounded from above, i. e., $f(n) \in O(\hat{k}^n)$ for some fixed $k \leq \hat{k} \in \mathbb{R}_{>1}$.

2.2.2 Basic Complexity Classification

We already saw one kind of a classification of algorithms by virtue of asymptotic running times. Hence, it would be worthwhile to obtain a classification of problems in terms of the running time of a fastest algorithm that solves a problem. However, problems exist for which it is unknown how quickly they can possibly be solved. To somehow grasp the difficulty of solving problems in a greater view, problems are cast into classes according to their complexity. In the following, we present some of the classical complexity classes.

P and NP The complexity class P contains all decision problems that admit a polynomial-time algorithm. Hence, given an instance of such a problem we can efficiently compute a solution, which can also be efficiently verified. The complexity class NP drops the requirement of computing a solution efficiently. Therefore, the class NP contains all decision problems for which a solution can be verified efficiently. Hence, the class NP is a superclass of P and we have P $\subseteq$ NP. However, it is still unknown and one of the important questions in computational complexity whether P = NP or P $\neq$ NP, although it is widely believed that P $\neq$ NP.

NP-completeness The question whether P $\neq$ NP attracted much research interest, which lead to a further specification of complexity classes. In particular, this involved the development of the concept of a *polynomial-time reduction*. Given two problems Π_1, Π_2, we say a problem Π_1 is polynomial-time reducible to a problem Π_2 if an efficient algorithm exists that transforms every instance $I_1 \in \Pi_1$ into an instance of $I_2 \in \Pi_2$ with the same answer. We denote this polynomial-time reduction by $\Pi_1 \leq_{\mathrm{P}} \Pi_2$. This notation indicates that if we solve Π_2 in polynomial running time we can automatically solve Π_1 in polynomial running time too. In this sense, the problem Π_2 is at least as difficult as the problem Π_1, which allows for establishing an order of the difficulty of problems.

We call a problem Π *NP-hard* if $\Pi' \leq_{\mathrm{P}} \Pi$ for all $\Pi' \in$ NP. If it additionally holds that $\Pi \in$ NP, we say the problem Π is *NP-complete* and denote the complexity class containing exactly these problems by NPC. Hence, the NP-complete problems represent the hardest problems in NP. This is why the question whether P $\neq$ NP centers around NP-complete problems. If any NP-complete problem can be solved in polynomial time, then all problems in NP can be solved efficiently and P = NP. Conversely, if any NP-complete problem can be shown to admit no polynomial-time algorithm, then all NP-complete problems lack an efficient algorithm and P $\neq$ NP. Assuming that P $\neq$ NP, we say a problem is *intractable* if it is NP-complete.

To prove that a problem Π is NP-hard, it suffices to reduce an NP-hard problem Π' to the problem Π at hand, i. e., to show $\Pi' \leq_{\mathrm{P}} \Pi$. Recall that a problem is in NPC if it is NP-hard and an element of NP. At this point, we lack a proof for the existence of such a problem class. Therefore, Stephen Cook, who developed the concept of NP-completeness, also formally proved that the *circuit-satisfiability problem* in conjunctive normal form, which is often referred to as the *SAT* problem, is NP-complete [Coo71]. In particular, he showed that the instances of every problem in NP can be encoded as a SAT formula of polynomial size. This work of Cook was a major breakthrough in complexity theory and lead in short time to a bunch of NP-complete problems [Kar72], which is still extended today. The sheer number of NP-complete problems lets many researchers believe that P $\neq$ NP.

Fixed-Parameter Tractability Was that all? Are NP-complete problems so hard to solve that researches may either prove that problems are intractable or show an efficient algorithm? The answer is no. Indeed, many of those intractable problems arise in problems companies have to deal with on a daily basis, and hence, have to be tackled in one way or another.

Often, intractable problems are solved by applying an ad-hoc approach delivering solutions that work well on the instances at hand but omit any proofs on the running time and the quality of the solution. We call these approaches *heuristic*. Another way to tackle optimization problems are *approximation algorithms*. These algorithms admit a proof on the running time as well as the quality of the solution. The latter point gives the name of these algorithms as the approximated solutions are guaranteed to be at most a specific amount away from the optimal solutions. Note that this is an informal sketch of the idea of approximation, for details we recommend the book by Vazirani [Vaz03].

A third way is to solve the problem exactly, despite the fact that all the time, we were speaking about intractable problems. Of course we cannot hope for solving all those hard problems. The approach taken here is to further classify the complexity of problems in terms of a *parameter* k and the size of the input n. Such a problem is called a *parameterized problem*. An algorithm is called *fixed-parameter tractable* if it has a running time arbitrary in a fixed parameter k but only polynomial in its input size n. If the size of the parameter k is small enough, we can efficiently solve the problem despite the fact that the problem is intractable with respect to its input size. More formally, a parameterized problem Π_K is fixed-parameter tractable if for every tuple $(I, k) \in \Pi_K$, which consists of an instance I and a parameter k, an algorithm $\mathcal{A}$ applied to (I, k) has a running time of $f(k) \cdot |I|^{O(1)}$, where f is some computable function depending only on k. The corresponding complexity class is denoted by FPT. The concept of polynomial-time reductions has been extended to a parameterized reduction that works similarly to show whether a problem is in FPT. For more details concerning the theory of parameterized complexity, we refer the reader to the book of Downey and Fellow [DF99].

When we look at instances of intractable problems, we can see that some of them can be solved efficiently. Of course, this cannot be decided automatically. Otherwise, we could further split the problem in either easy or hard instances. However, we can look at the properties that allow some instances to be solved efficiently. In particular, we want to deduce rules that allow for pruning (and solving) the easy parts of all instances of a problem. Hence, we crystallize the hard part of the instances at hand, which is called the *kernel* of the instance. This kernel then has to be solved using an exponential-time algorithm. More formally, we call an algorithm $\mathcal{A}$ a *kernelization* for a parameterized problem Π_K if $\mathcal{A}$ transforms each instance $(I, k) \in \Pi_K$ in polynomial time in the size of $|I|$ and k to an instance $(I', k') \in \Pi^{'}_{K'}$ such that

- $(I, k) \in \Pi_K \Leftrightarrow (I', k') := \mathcal{A}(I, k) \in \Pi_K$,
- $k' \leq k$, and
- $|I'| \leq g(k)$ for some fixed function g depending only on k.

Whether a problem is kernelizable is related to fixed parameter tractability. In particular, a problem is fixed-parameter tractable if and only if it is kernelizable. One direction can been seen easily: we apply a kernelization algorithm to an instance, this takes time $O(|I|^c)$, for some constant c, and solve the resulting kernel in an arbitrary running time $O(f(k'))$. However, the other direction is more complicated and we refer the reader to the book of Downey and Fellows [DF99].

2.2.3 Basic NP-Hard Problems

In the following, we present two classical NP-hard optimization problems and their counterpart formulations as NP-complete decision problems. At least for restricted graph classes, both problems admit a kernelization algorithm, which we apply in our experiments later.

Vertex Cover A *vertex* cover of a graph $G = (V, E)$ is a subset $C \subseteq V$ of vertices such that $C \cap \{u, v\} \neq \emptyset$ for all edges $\{u, v\} \in E$. The optimization variant of this problem asks for a *minimum vertex cover* in a given graph. There also exists the decision variant of this problem, which is known as the *k-vertex cover* problem. Here, we are given a graph G and a positive integer k and ask whether there exists a vertex cover of size at most k. In our work, we concentrate on the problem of k-vertex cover, which is known to be NP-complete [Kar72]. However, k-vertex cover is fixed parameter tractable and can be solved for planar graphs in time $O(c^{\sqrt{k}}n)$ [AFN04], where c is a constant and n is the size of the input.

Dominating Set Given a graph $G = (V, E)$, a *dominating set* is a subset $D \subseteq V$ of vertices such that each vertex in $V \setminus D$ has a neighbor in D. The *domination number* $\gamma(G)$ of a graph denotes the size of the smallest dominating set of G. The decision problem *k-dominating set* asks whether $\gamma(G) \leq k$, which is a classical NP-complete decision problem [GJ79]. When restricted to planar graphs, the k-dominating set problem remains NP-hard but is fixed parameter tractable [AFN04]. Similar to k-vertex cover, the running time is in $O(c^{\sqrt{k}}n)$, albeit with a larger constant c.

2.3 Randomness

In network analysis we are interested in specific properties of a network. Such networks may exhibit characteristics that cannot be grasped in terms of provable properties but need to be evaluated experimentally. However, experimental studies have to be performed on a meaningful data set to not flaw insights gained. Hence, an exhaustive experimental study would be best, i. e., the experiments should be performed on all instances that are contained in the graph class of interest. However, this is often impossible. For example, the number of possible triangulations for planar graphs of size $n = 20$ is more than 50 billions, which is unlikely to be testable in reasonable time. Hence, we are interested in selecting a representative sample of instances that enables us to perform experiments with a meaningful statement afterward. The process of selecting a representative sample of an initial set of instances is often called *sampling*. However, sampling requires an initial set of instances to exist, which in our example is also unlikely. Hence, we are also interested in *generating* a representative sample of instances instead of generating all possible instances.

Hence, for the generation and sampling of data sets, we use the following terms from probability theory. We denote the probability of some event E by $\mathbb{P}[E]$, and the expected value of some random variable X by $\mathbb{E}[X]$.

Generators Having no instance at hand, a *generator* is capable of creating artificial instances using a random process. Usually, generators base on a specific model, which allows the instances to exhibit specific properties, e. g., planarity. Later, we will see generators that can be roughly classified by means of their capability to generate the full range of possible elements and whether they do this with a certain probability. In the case a generator is capable of generating all graphs of a given class with positive probability, we call the generator *complete* and otherwise *incomplete*. If the generator additionally generates each graph with the same probability, then we call it *uniform* or able to generate graphs *uniformly at random*.

Common Distributions In our experiments, we have to generate random values, which we want to be documented. Thus, our experiments are reproducible to a certain extent. We use the following probability distributions; for details see a book of Daly et al. [DHJ$^+$95]. We refer to the *uniform distribution* on $[a, b]$ by $\mathcal{U}(a, b)$ with density function $\mathcal{U}_{[a,b]}(x) = \frac{1}{b-a}$ for all $x \in [a, b]$ with $a < b \in \mathbb{R}$. We refer to the *normal distribution* with mean μ and standard deviation σ by $\mathcal{N}(\mu, \sigma^2)$ with density function

$$\mathcal{N}_{\mu,\sigma^2}(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$$

for all $x \in \mathbb{R}_0^+$, $\mu, \sigma \in \mathbb{R}$ and $\sigma \neq 0$. And we refer to the *exponential distribution* by $\mathcal{E}(\lambda)$ with density function $\mathcal{E}_\lambda(x) = \lambda e^{-\lambda x}$ for all $x \in \mathbb{R}_0^+$.

Markov Chain A *Markov chain* can be used to model stochastic processes, which works in *discrete time steps* [GS97]. We consider Markov chains with *finite state space S*. In each step a Markov chain *transits* from one state $i \in S$ to a next state $j \in S$ according to a transition probability distribution $\pi_{ij} \colon S \longrightarrow [0, 1]$. Markov chains have the property that their current state depends only on the previous state, i. e., the current state is independent of the whole history or future states. More formally, let X_t denote the random variable at time t that describes the state of a Markov chain, and let π_{ij} denote the probability that a Markov chain transits from state $i \in S$ to state $j \in S$. Then, the state of a Markov chain is $\mathbb{P}[X_{t+1} = j \mid X_t = i] = \pi_{ij}$. We consider Markov chains that admit a *stationary distribution*, i. e., the probability distribution is invariant under an arbitrary number of transitions of the Markov chain. More formally, in a stationary distribution we have probabilities $\pi \colon S \longrightarrow [0, 1]$ with

$$\pi(s) = \sum_{s' \in S} (\pi(s')) \cdot \mathbb{P}[X_{t+1} = s \mid X_t = s'] \ .$$

Sometimes, a Markov chain starts with an arbitrary probability distribution different from the stationary distribution. Under certain conditions and a certain number of transitions of a Markov chain, an arbitrary probability distribution can become a stationary distribution. By *mixing rate*, we denote the number of necessary transitions of a Markov chain to reach the stationary distribution from an arbitrary starting state.

2.4 Data Structures and Basic Algorithms

In this work, we handle huge amounts of data, which needs to be organized for an efficient usage. The usual way is to employ efficient *data structures*, which are also a key ingredient for designing algorithms that perform well. Hence, in the following we briefly introduce basic concepts of the data structures and the algorithms used later in our work.

2.4.1 Data Structures

In our algorithms we use several data structures. In the following, we briefly describe the concepts and give references for a more thorough treatment.

Disjoint Sets Later, we need a data structure that maintains a partition of a set of elements into disjoint subsets; see [CLRS01]. Each subset is identified by its representative. The data structure supports three operations, namely, *makeset*, *union* and *find*. The operation makeset is used to initialize the data structure by creating a subset for each of the elements, i. e., each element is the only member and representative of its subset. A union operation merges two subsets into a single one. The operation find returns the representative information of an element, which can be used for a fast check whether two elements are in the same set. The running time of our implementation for a sequence of n operations is $O(n \cdot \alpha(n))$ with $\alpha(n)$ the inverse Ackermann function.

Queue A *queue* is a data structure that can be used to store and retrieve elements in a specific order related to the insertion time of the elements; see [CLRS01]. Usually, the queue supports at least the operations *store* and *extract*. As the name of the operation indicates, store is used to add an element to the data structure. Opposed is the function of the operation extract, which returns the first element according to the order that the queue maintains, and removes it from the data structure. Note that in implementations the extract operation often is split into separate return and remove operations. The running time of all queue operations is $O(1)$. The two most common orders how the elements are retrieved are first-in first-out (FIFO) and last-in first-out (LIFO).

Priority Queue An extension of queues are *priority queues*, which do not return the elements in a specific order that is related to the insertion time. Instead, the priority queue works on elements $E = (k, v)$, where k is some key, which specifies the element's priority, and v is the contained value. The priority queue maintains the order of its elements according to their associated keys. In our case, the operations supported by a priority queue are *extract min*, *decrease key* and *insert*. The inserted elements are maintained by an internal data structure of the priority queue such that the element with minimal key can be extracted. The key value of an element can be updated by the operation decrease key, which also maintains the correct order of the elements. Priority queues come in several flavors. However, we are always using a *binary heap* in our implementations, which has a running time of $O(\log n)$ for any of the above operations [CLRS01].

Quadtree A way to quickly find the nearest point in a region in the two dimensional space $\mathbb{R}^2$ is to use a *quadtree* [dBCvKO08]. This data structure recursively subdivides a bounding box in the plane into four regions of equal size. These regions can be seen as buckets, which come with a specific maximum capacity. If this maximum capacity is reached, the quadtree further subdivides the region and the content of this bucket is moved

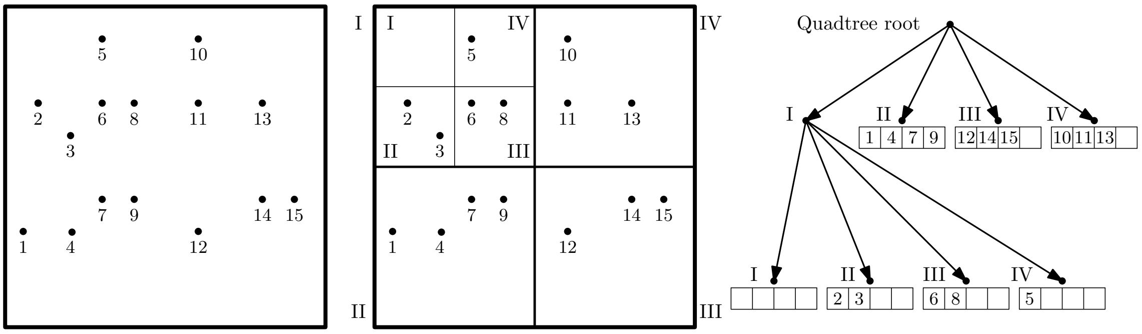


Figure 2.2: On the left: point set in the plane. In the middle: division of the point set according to a quadtree with a bucket size of 4. On the right: the corresponding quadtree with a height of 2.

into its subdivisions.

A quadtree is realized as a rooted tree, where each inner vertex has four children. In this tree, each vertex represents a rectangle. The leafs of that tree are a partition of the plane. Let h be the height of the quadtree. Then, the running time to build a quadtree is $O((h+1)\cdot n)$ with n the number of points to insert. In a quadtree, the neighbor of a point can be found in running time $O(h+1)$.

Figure 2.2 shows an example of a point set in the plane we want to process, the quadtree division of the plane and the corresponding quadtree. The bucket size of the quadtree is set to 4. In the first step, too many vertices are in the area and the quadtree splits. Then, the size of the buckets of regions II, III, and IV suffice. However, region I still contains too many nodes and is split further. The nodes are sorted into these new regions without violations of the size of the buckets. Hence, we are done and obtain a quadtree of height 2.

2.4.2 Basic Algorithms

In the following chapters, we assume the reader to be familiar with several algorithms. Hence, we now give a brief description and refer to further literature.

Breadth-First Search The *breadth-first search* (BFS) is a very basic algorithm which is used to traverse a graph from a starting node in order of growing graph-theoretic distance [CLRS01]. In this way, the visited nodes form concentric circles, which expand similar to the waves that originate from a stone thrown into water. Figure 2.3 on page 22 shows an example of a breadth-first search starting at node s. The algorithm visits the nodes in non-decreasing hop distance. The markings show the visited nodes after all nodes at hop-distance 1, 2 and 3 are visited.

More technically, given a graph $G = (V, E)$, a FIFO queue Q and a start point $s \in V$, the start point s is added to the queue $Q \leftarrow s$. Then, the main routine starts. The first element of the queue is extracted $Q \leftarrow Q \setminus v$ and all its neighbors $\Gamma(v)$ that have not been visited yet get enqueued $Q \leftarrow Q \cup (\Gamma(v) \setminus \mathrm{vis}(v))$. This is done until the queue is empty. The algorithm stops, once all elements that are within the same connected component of s have been visited.

K-means In this work, we want to compress data to a certain extent. To achieve this, we employ the *k-means* algorithm introduced by Lloyd [Llo82]. The goal is to partition a set of elements E into k clusters, such that each element is member of a *cluster*. Additionally, we require a function that allows for computing the distance between any pair of the elements, e. g., the Euclidean distance for points in $\mathbb{R}^n$. Then, the algorithm works as follows. First, $k \subseteq E$ elements are chosen randomly. They form the *centroids* of the k clusters. Then the main routine starts. Each element is assigned to the nearest centroid found by the distance function. After each step, the centroid of each cluster is replaced by the mean of its members. Then, for each element it is checked whether it needs to be reassigned to another cluster according to the distance function. The main routine is repeated until no further changes occur. In summary, applying k-means to a set of elements E, we gain k clusters, each represented by its centroid. Additionally, we derive a mapping of the elements of the original set to the cluster they belong to.

Sweep Line Paradigm Usually, *sweep line algorithms* are employed in *geometric applications*, where a graph is embedded in the plane. In this setting, we speak of *points* instead of the vertices of a graph, and associate them with coordinates.

An algorithm based on the *sweep line paradigm* usually consists of two phases. In the first phase, the points p of an input point set $p \in P$ are sorted lexicographically, e. g., increasingly by x- and then by y-coordinate, such that $p_1, \ldots, p_n$ denotes the sorted order. In the second phase, the algorithm performs operations during a single sweep over the sorted points $p_1, \ldots, p_n$. This can be seen as a vertical line that sweeps over the plane, e. g., from left to right.

2.5 Route Planning

In the past two decades route-planning techniques gained much attention from researchers all over the world. This has led to many remarkable results. As a large part of this work is dedicated to this field of research, we now briefly introduce the models and techniques used in our experiments later. For a detailed overview, we refer the reader for the time-independent scenario to a work by Delling et al. [DSSW09a] and for the time-dependent scenario to another work by Delling et al. [DW09], respectively.

2.5.1 Modeling of Road Networks

A common technique to represent road networks are directed weighted graphs. The vertices of such a graph are used to model junctions or to approximate bends. An edge connects two vertices if there is a drivable lane segment of a road between them.

As already mentioned, the graph representations of road networks are directed, which we use to model several details of the underlying network. For example, this allows modeling of one-way roads or properties of road lanes, e. g., speed limitations or differing weights for opposite directions. Depending on the scenario, *time dependent* or *time independent*, different length functions are assigned to the edges.

Time-Independent Scenario In the time-independent scenario, we use $\omega \colon E \longrightarrow \mathbb{R}_+$. Usually, we are interested in two metrics. First, the *Euclidean space* metric to specify the physical length of each edge. Second, the *time* metric to specify how long it takes to traverse a given edge.

Time-Dependent Scenario In the *time-dependent* scenario functions instead of constants are used to specify edge weights, i. e., $f \colon E \longrightarrow \mathbb{F}$, where $\mathbb{F}$ are periodic functions $t \colon \Pi \longrightarrow \mathbb{R}_+$, $\Pi = [0, p]$, $p \in \mathbb{N}$ such that $t(0) = t(p)$ and $t(x)+x \leq t(y)+y$ for any $x, y \in \Pi$, $x \leq y$. The value p is called the *period* of the function Π. We use $\operatorname{len} \colon E \times [0, p] \longrightarrow \mathbb{R}_+$ to evaluate an edge for a specific departure time τ. We model these edge weights by the use of *piecewise linear functions*, where each point depicts the travel time at a specific time of the day. In this work the piecewise linear functions represent the travel times of a whole day with a period of 96. More precisely, the function contains a sample point every 15 minutes. The evaluation of the function at time τ is then done by linear interpolation between the points left and right to τ.

2.5.2 Basic Route-Planning Techniques

For a long period of time, the algorithm of Dijkstra was the fastest algorithm that guaranteed to find the shortest path in road networks [Dij59]. Nevertheless, the running time on an instance of Germany that consists of 78 417 nodes and 241 516 edges, took about 40-60 seconds in 1998 [Ert98]. The only other method was to apply heuristics that run quick in practice but lack a proof of correctness. Of course, people could drive using maps or live with an approximation instead of losing time by asking one's way and possibly going the wrong way. Anyway, with increasing speed of computers, finding the exact shortest path became attractive again. Using bidirectional Dijkstra, a shortest-path computation on Europe, which consists of about 18 million nodes and 42.6 million edges, took on average 2.7 seconds in 2009 [DSSW09b]. Although, this was a big jump in computational performance it is still not sufficient for some applications such as providing an Internet online route-planning service.

Shortest Path Computations When thinking of route planning, we usually have in mind to find a route from some starting point to some destination point. Additionally, we require this route to fulfill certain properties, e. g., being as fast as possible.

However, in algorithmics the problem is more diverse. We present three basic problems, which are ordered increasingly with respect to their difficulty. Given some graph $G = (V, E, \operatorname{len})$, where len is some metric function, e. g., travel time or euclidean distance, we want to find a shortest s-t path from a node s to a node t. We call this problem the *single-source single-target shortest-path problem* or *point-to-point shortest-path problem*. Likewise, we denote by *single-source all-targets shortest-path problem* the problem of finding a shortest s-t paths for all $t \in V \setminus \{s\}$. This problem is equivalent to computing a shortest-path tree for the node s. The third problem is to compute all shortest s-t paths between all pairs of nodes in the graph. Accordingly, we call this problem the *all-pairs shortest-path problem*.

Scenario and Shortest-Path Computation The difficulty of computing shortest paths in road networks also depends on the road-network models we work on, as some require us to incorporate further constraints. Recall the road-network models from above. Depending on the model used, we call the problem of computing shortest paths in a time-independent road network the *static scenario* and in a time-dependent road network the *dynamic scenario*. The computation of shortest paths in the static scenario is rather carefree. However, in the dynamic scenario we are interested in shortest paths computation for some departure time τ. Without further requirements on the network model, this may lead to race conditions, i. e., for two given points in time τ_1, τ_2 with $\tau_1 < \tau_2$, if a car leaves node u at time τ_1 via the edge (u, v) before another car leaves node u at time τ_2 via the edge (u, v), the car departing at τ_2 can still arrive at v before the car departing at τ_1 arrives. Hence, we restrict the graph to fulfill the *FIFO property*, which is also known as the *non-overtaking property*. If a network exhibits the FIFO property, we can compute shortest paths in polynomial time [KS93]. In the case a network lacks the FIFO-property, the complexity of computing shortest paths depends on whether we are allowed to wait at nodes. The problem remains polynomially solvable if waiting is allowed and becomes NP-hard otherwise [OR90]. Note that in this thesis, we will only consider models that fulfill the FIFO-property.

Dijkstra For a long time, the algorithm of Dijkstra [Dij59] was the state of the art to solve the problem of finding shortest paths in networks with *non-negative* edge weights. The algorithm maintains an array of *interim distances* $d(s, u)$, from the starting point $s \in V$ to all nodes $u \in V$. Hence, in an initialization step these distances are set to ∞. Note that in an actual implementation this step is skipped and instead, a time stamp mechanism is used to verify whether a node has been visited already. The algorithm locally explores the network, i. e., nodes $u \in V$ are visited in non-decreasing order with respect to the distance $d(s, u)$. To quickly retrieve the node with the shortest distance to s, a priority queue Q is used. In particular, the node u that is nearest to s is extracted from Q. Then, the newly discovered neighbors of u are stored in Q and the interim distance of all nodes found is updated if necessary. The update of an edge is called *relaxation*. Whenever a node u is extracted from the priority queue, the distance $d(s, u)$ is *settled*. We call these nodes the *settled nodes*. By the time the algorithm settles node t, we have found the *distance* $d(s, t)$. If we are interested in the point-to-point shortest path from node s to t, we can stop now. Otherwise, the algorithm runs until all reachable nodes are settled.

Note the subtle difference, we were interested in the shortest path but computed only the shortest distance at this point. Hence, we have to extract the shortest path. Using Dijkstra's algorithm, we can easily maintain a *shortest-path tree*, i. e., each node stores its predecessor. Note that storing only the predecessor of a node is not yet a tree in our sense, as we can only traverse the edges towards the root. Nevertheless, we can quickly extract the shortest path from any node to the root by following the predecessor information.

The asymptotic running time of Dijkstra's algorithm is highly dependent of the running time of the priority queue used. Using the algorithm of Dijkstra in combination with a Fibonacci heap [FT87] to compute shortest paths in general graphs yields a worst-case time complexity of $O(m+n \log n)$. In contrast, using a binary heap as priority queue leads to a worst-case running time of $O((n+m) \log n)$ [CLRS01]. However, our application is to compute shortest paths in road networks, which are sparse, and thus roughly have similar asymptotic running time compared to Fibonacci heaps. Hence, we use binary heaps as their implementation is rather easy compared to Fibonacci heaps, and they are efficient in practice [CGR96, Sch08].

The set of settled nodes grows in concentric circles of growing distance around the starting node s during the execution of the algorithm of Dijkstra. This is due to the

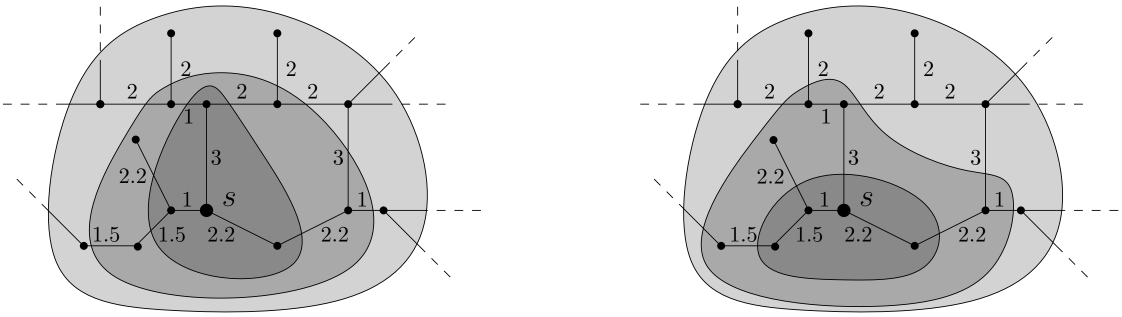


Figure 2.3: On the left: visited nodes of the breadth first search (BFS) algorithm starting at node s after it visited all nodes at hop-distance 1, 2 and 3. On the right: Dijkstra's algorithm, which is also started at node s, settles nodes in a order different to the order in which BFS visits the nodes. For comparison, the number of settled nodes marked equals the number of visited nodes by BFS.

non-decreasing distance in which nodes are settled. In this way, an ordering of the nodes with respect to s is established. By *Dijkstra rank*, we denote the position of a node u within this ordering. Figure 2.3 shows the visited nodes of a BFS and the settled nodes of Dijkstra's algorithm applied to an example network, whose edge weights are Euclidean distances. Both algorithms are started at node s in the network. We can observe that Dijkstra's algorithm settles nodes in an order different to the order in which BFS visits nodes.

2.5.3 Basic Speed-Up Techniques

As already mentioned, the classical algorithm of Dijkstra was for a long time the fastest approach to solve the shortest-path problem in road networks exactly. Despite the improvements in computational power, the running time on instances of continental size is still in the seconds for a single query. This is much too slow for two reasons. First, users lack the patience to wait for several seconds to get a solution. Second, in a server-based scenario we additionally have many parallel requests that would require large expensive clusters to provide a high quality service. For these reasons, a two-phase approach has been established. In the first phase, which we call the *preprocessing phase*, auxiliary data is computed. In the second phase, which we call the *query phase*, the auxiliary data of phase 1 is exploited to accelerate the shortest-path computation. In particular, in almost all cases the query phase is implemented using a modified variant of the algorithm of Dijkstra, which has been adapted to exploit the auxiliary data computed in phase 1.

Hence, it is natural to compare the running time of a new approach $\mathcal{A}$ to the running time of the algorithm of Dijkstra $\mathcal{D}$. In particular, we are interested how much faster $\mathcal{A}$ performs than $\mathcal{D}$ for the same shortest-path computation between fixed nodes s and t. To make the outcomes of two algorithms comparable, they have to be applied to the same set of (s, t) pairs. Usually, the size of the networks forbids to compare the algorithm on all pairs of nodes. Hence, a set of (s, t) pairs is sampled, which we refer to as *demand set* $\mathcal{S} = \{(s, t) \mid s \in V, t \in V \text{ and } s \neq t\}$. The *speed-up* of a technique $\mathcal{A}$ compared to Dijkstra

$\mathcal{D}$ is then

$$\frac{1}{|\mathcal{S}|} \cdot \sum_{(s,t) \in \mathcal{S}} \frac{\mathcal{D}(s,t)}{\mathcal{A}(s,t)}.$$

As we have already seen, the algorithm of Dijkstra has to search and settle many nodes. Hence, the common approach to improve the performance of an algorithm is to reduce the number of nodes to settle. The settled nodes are called the *search space* of a shortest-path algorithm. Using the search space of a shortest-path algorithm for a comparison has two reasons. First, the running time depends on the computer language as well as the computer architecture and speed. Second, comparing the search space is mostly independent of implementation skills. Hence, the search space and, possibly, more details on the operations that an algorithm performs, are used for comparisons of shortest path algorithms.

Historical Overview In Figure 2.4, we present an overview of important shortest-path techniques and when they emerged. Note that the techniques presented in the timeline is by far not complete and only selected representatives are shown.

Prior to our depicted start of Phase I research already focused on shortest-paths. However, Dijkstra's algorithm had the best guaranteed running time when applied to practical instances, i. e., networks not containing cycles of negative length. Hence, we can say Phase I started with the publication of Dijkstra's algorithm and was centered around improving it for a long time. In particular, research focused on the development of faster priority queues to gain improved running times.

Phase II started when researchers dropped the constraint of improving the worst-case running time and turned towards heuristic improvements that allows techniques to return *exact* results faster. Additionally, with the emerging Internet a demand for route-planning sites grew that provide their service to multiple users in parallel. However, the high running-time of Dijkstra's algorithm on large networks prohibited its usage in high-quality services, which raised the demand for faster solutions. To this end, it was allowed to spend a long time for preprocessing that will quickly get amortized by fast query answers. In this time, most of the basic speed-up techniques were developed.

Usually, the techniques are compared with each other by the running-time they need for preprocessing, the running-time of the query and the occupied memory. In Phase III researchers began to refine and combine approaches to develop techniques further. At the end of this phase, techniques had been greatly improved. In terms of the above measures, a preprocessing of networks of continental size takes only minutes. At the same time, networks, which includes the auxiliary preprocessing data, fit in the main memory of a desktop computer, which can answer queries within microseconds.

Having achieved these impressive results, in Phase IV researchers turned their focus to the more realistic dynamic scenario. Several techniques developed for the static scenario have been augmented and refined to adapt to the complex dynamic scenario. However, techniques developed for the dynamic scenario have not yet achieved similar speed-ups as for the static case. Hence, research in this field is still ongoing. Next, we present a brief introduction to the techniques highlighted in the timeline.

Basic Concepts The majority of available speed-up techniques aims at reducing the search space in the query phase. The algorithms incorporate at least one of the following four basic concepts.

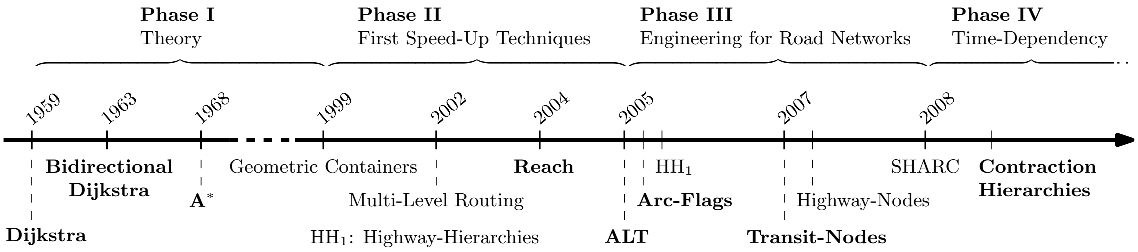


Figure 2.4: The four phases of research in the field of shortest-path computation and their duration. The timeline is by far not complete but depicts important dates and techniques.

First, techniques benefit from the knowledge in which direction the target vertex is located. Hence, the algorithms are capable of expanding the search space towards the goal. This way, we can omit vertices not leading in the right direction. We say such a technique works in a *goal-directed* way, e. g., A^*.

Secondly, a basic concept used in many speed-up techniques is to add *shortcuts* to the network $G = (V, E)$ [SS05]. Shortcuts are used to skip rather unimportant nodes in a query. Take for example a simple path $(u_1, \ldots, u_k)$, with $u_i \in V$ and $k \geq 3$, of nodes, which are considered to be unimportant. A shortcut from a vertex u_1 to a vertex u_k corresponds to adding an edge $e = (u_1, u_k)$ to the network $G = (V, E \cup e)$ and we say the nodes $u_2, \ldots, u_{k-1}$ are shortcut. To preserve the distances within G, the weight of e is set to the distance $d(u_1, u_k)$. During a query, the shortcut vertices can be omitted, which reduces the search space. However, techniques adding *shortcuts* to the network have to invest greater effort than plain Dijkstra to extract the shortest path.

Thirdly, techniques precompute so called *transit nodes* and the distance between each pair of them [BFM$^+$07, BFSS07]. This approach is based on the observation that all sufficiently long shortest-paths have to pass through a rather small set of nodes, which are the transit nodes. Hence, long-range queries consist of finding the nearest transit nodes and a couple of table look-ups between them.

Fourthly, road networks exhibit an inherent hierarchy, which speed-up techniques can exploit [SS05]. Hence, we say such a technique works in a *hierarchical* way. The idea behind this approach is that usually, a road network consists of several hierarchy levels, each being a smaller subset of the previous level. Take for instance a network that admits three hierarchy levels, i. e., slow city streets, intermediate country roads and fast expressways. Intuition says that if we start in a city and want to drive to a city far away, we have to drive on the slowest level until we leave the city. Then, we travel on faster roads until we reach the top level of the hierarchy, the expressway. When we close in to our target destination, we descend again from the top level of that hierarchy until we reach the target, possibly on the slowest road again. The observation behind this approach is that the graph to consider on higher levels in the hierarchy is significantly smaller, and hence, faster to traverse. Thus, a query tries to climb up in the hierarchy as fast as possible to consider only small subgraphs.

These basic concepts led to several approaches how to best prune the search-space to speed up shortest-path queries, not to mention the plethora of combinations to gain even superior speed-ups. However, in this work, we concentrate on approaches that first

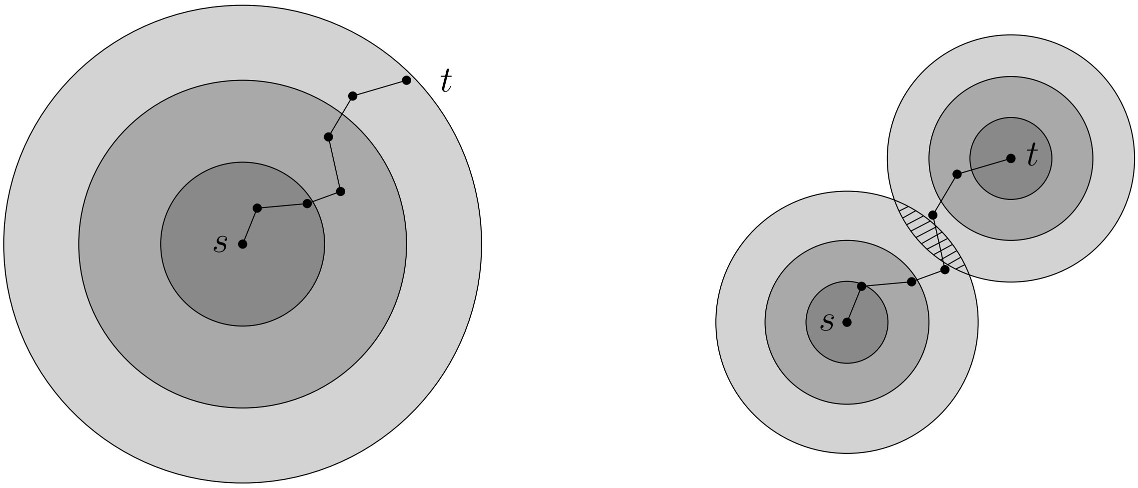


Figure 2.5: On the left: schematized search space of a query from s to t using Dijkstra's algorithm. On the right: schematized search space of the bidirectional approach, i. e., a Dijkstra query is started at s and a backward Dijkstra query is started at t. The search space of the bidirectional approach is about the half of the size of that of plain Dijkstra.

realized the basic concepts and laid the foundations for the later works. Hence, in the following, we briefly describe those representatives of speed-up techniques that will be used later in our work.

Bidirectional Dijkstra One of the first ideas to accelerate the algorithm of Dijkstra $\mathcal{D}$ is to perform two searches, one for the source node $\mathcal{D}(s,t)$ and one for the target node $\overleftarrow{\mathcal{D}}(t,s)$ [Dan62]. Note that the *backward search* $\overleftarrow{\mathcal{D}}$ explores the graph along incoming edges instead of following the outgoing edges in the classical algorithm of Dijkstra $\mathcal{D}$. Both searches are stopped as soon as the search spaces overlap to a certain extent. Figure 2.5 shows schematized search spaces of plain Dijkstra and the bidirectional approach for a shortest path query on the same pairs of nodes s and t . The search space roughly grows in concentric circles. Using bidirectional Dijkstra the radius of each search space around s and t is approximately the half of plain Dijkstra. This way only a marginal speed-up of about 2 can be achieved. However, the base idea of the bidirectional approach lays the foundation for many of the ingenious methods that were introduced later.

A* The classical algorithm A^* [HNR68] is a representative of the class of algorithms that work in a *goal directed* way. In a goal-directed algorithm the search space is stretched in the direction of the target, e. g., the search space grows elliptically toward the goal instead of growing in concentric circles around the source node. The algorithm A^* achieves this by modifying the edge weights $\operatorname{len}(u,v)$ by some lower bound function $\pi\colon V \longrightarrow \mathbb{R}_+$. The length of an edge is then $\operatorname{len}'(u,v) := \operatorname{len}(u,v) - \pi(u) + \pi(v)$, where $\pi(v)$ is a lower bound on $d(v,t)$. A common function to estimate the distance to the target node is the Euclidean distance $\parallel v - t \parallel$. This way, the edges toward t are shortened and, thus, preferred. However, using the Euclidean distance for a lower bound estimation leads to a slowdown of a factor of 2 [GH05]. For this reason, we omit this technique in our experiments. Though, the algorithm can be improved using other lower-bound functions as described

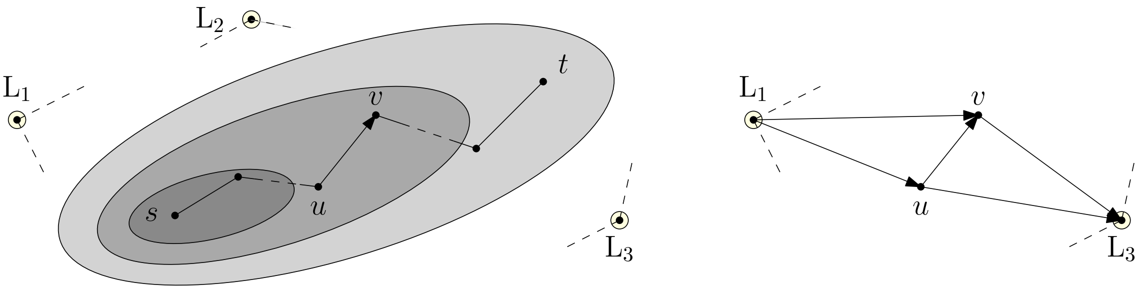


Figure 2.6: On the left: the schematized search space of a query from s to t using the ALT algorithm. Additionally, three landmarks $\mathrm{L}_1, \ldots, \mathrm{L}_3$ are shown. The search space of ALT is guided by the landmarks and hence, stretched towards the target t. On the right: triangle inequality for an edge (u, v) to the landmarks L_1 and L_3.

next.

ALT The algorithm A^* comes with a slowdown in performance when used in road networks for two reasons [GH05]. First, the algorithm introduces a significant overhead, and second, the travel-time metric turned out to be a poor lower bound, which reduces the search space only slightly. Hence, the idea was to employ designated nodes of the graph to improve lower bounds. These points are called *landmarks*. In particular, each node of the graph stores the shortest-path distance to the whole set of landmarks. Now, when we consider whether to visit a node, we can estimate the gain using the shortest-path distance to well-selected landmarks and the triangle inequality, i. e., for all three vertices $u, v, w \in V$ in a graph $G = (V, E)$ holds $\operatorname{dist}(u, w) \leq \operatorname{dist}(u, v) + \operatorname{dist}(v, w)$. The concepts of A^*, landmarks and the triangle inequality were condensed into the algorithm ALT [GH05]. Of course, the performance of the algorithm depends on the number and quality of the landmarks [GW05]. Additionally, during the search we have to select a set of appropriate landmarks. Figure 2.6 shows the schematized search space of ALT and an exemplary evaluation of the triangle inequality for an edge (u, v) and two landmarks. Using the landmarks, we can see that the search space is stretched towards the designated target. Using reasonable landmarks, a speed-up of up to 28 can be gained.

Reach-Based Routing The *reach value* of a node v measures how central v's position within the longest shortest paths that go through v is. A high reach value correlates with a high centrality in a long shortest path. More formally, the reach value of a shortest path of a node v is $R_{st}(v) := \min(d(s, v), d(v, t))$. The reach value of a node v is then defined as $R(v) := \max_{s,t \in V} R_{st}(v)$. Hence, the reach values of the nodes define a hierarchy within the network. In the shortest-path search, we can skip the exploration of nodes whose reach value is too small to reach the source or target from this node. Note that reach can also be defined on edges, and in terms of geometric distance instead of shortest-path distance. The concept of reach has independently been developed by Gutman [Gut04] and Ertl [Ert98] using the geometric distance. The preprocessing of reach values involves computing all-pairs shortest paths. Hence, it is not practical on graphs of continental size. However, we use an improved version that incorporates shortcuts and performs much better due to an approximate computation of the contained all-pair shortest-paths problem [GKW09]. Note that even though part of the preprocessing is approximated, the

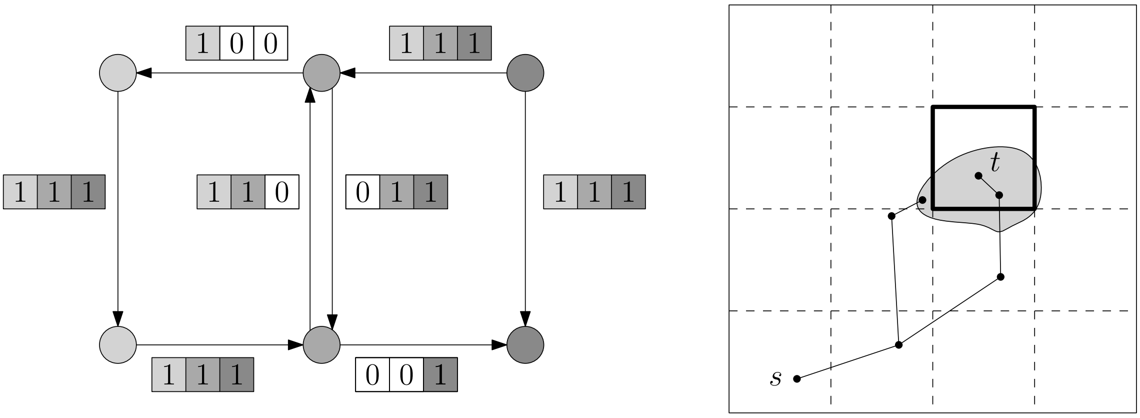


Figure 2.7: On the left: a network partitioned into three cells and the corresponding arc-flags. On the right: schematized search space of a shortest-path query from s to t using the arc-flags algorithm. The search space roughly expands along the shortest path until the algorithm explores edges near the target cell. Having set most arc-flags to true the search space then grows largely.

query remains correct and returns exact solutions. The improved approach can achieve a speed-up of up to 3 932.

Arc-Flags The basic idea of the *arc-flag algorithm* [Lau04] is to partition the road network into k disjoint regions of about equal size. Then, we store at each arc whether it is important for reaching a specific region. In particular, an arc is important for a region i if it is part of some shortest-path that leads into i. We store this information in a bit vector of size k, which is attached to each edge (u, v). If a shortest path leads into region $1 \leq i \leq k$ that contains the edge (u, v), then, the ith bit is set to true. In the query phase, we determine in which region j the target is located. Then, we explore only those edges that have the jth flag set to true. In Figure 2.7, we show an example network preprocessed by the arc-flag algorithm. The network is partitioned into three cells. Additionally, we present a schematized search space of a shortest-path computation from s to t using the arc-flags algorithm. The arc-flag algorithm roughly follows the main routes towards t until the network around the target cell is explored. From there on, the search space largely expands. The reason is that all arc-flags within that cell are implicitly set to true. Additionally, the edges around a cell are likely to also have the arc-flag for the target cell set to true. Note that a naïve implementation involves a very high preprocessing time. Hence, several specific characteristics have to be considered [HKMS09]. Doing so, we can achieve a speed-up of up to 3 951.

Contraction Hierarchies In our descriptions so far, we omitted techniques that explicitly exploit the inherent hierarchy that road networks adhere. Corresponding algorithms are rather complicated. Hence, it is quite surprising how successful the rather easy technique of *contraction hierarchies* is, although it drives the concept of hierarchy to an extreme end [GSSD08].

Like the other techniques, the basic idea is to enable the query algorithm to skip unimportant nodes in its search. To this end, contraction hierarchies uses a highly detailed hierarchy. In particular, the hierarchy consists of n levels, which is one for each node.

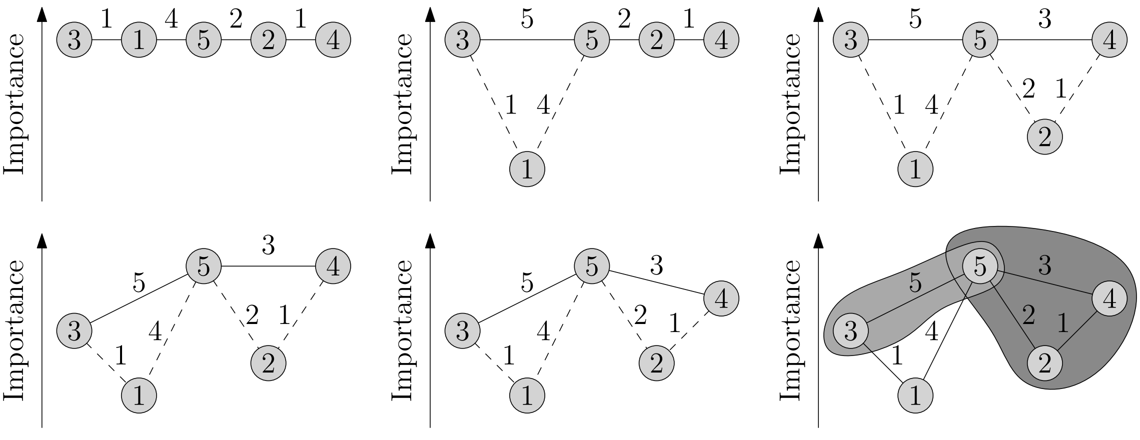


Figure 2.8: Five contractions applied onto a network. Labels in nodes denote the importance of the node, labels on the edges represent weights. Order of the contraction steps is on the top from left to right and then, on the bottom from left to the middle. Dashed lines represent shortcut edges. The height of the nodes depicts their importance. Bottom right: search space of a query from node 3 to node 2.

The preprocessing phase of contraction hierarchies is split into two phases. First, the importance of the nodes are estimated. Second, one of the most unimportant nodes is contracted and shortcuts are inserted that preserve the distance between the remaining nodes. The *contraction* of a node v corresponds to the deletion of v and its incident edges. Additionally, shortcuts are added between the neighbors of v such that their shortest path distance is preserved. Depending on the method used to measure the importance of nodes, an update of the affected nodes has to be performed. Note that the order of the contractions defines a total order on the set of nodes. However, computing an optimal order of the nodes that minimizes the number of shortcuts to insert is NP-hard [BCK$^+$10].

The query phase uses the bidirectional approach. Hence, a forward and a backward search are started similar to the bidirectional algorithm of Dijkstra. However, the basic (backward) algorithm is adapted to relax only those edges that lead to (come from) more important levels.

In Figure 2.8, we present an example of the contraction routine of a contraction hierarchies preprocessing as well as an example of the query. In the example, the importance of the nodes has already been computed. The order of the contractions is on the top from left to right and then, on the bottom from left to the middle. The node with lowest importance is processed first. It is pulled to its level and in this case, the dashed incident edges are temporarily removed and substituted by a shortcut. This is iteratively done until all nodes have been processed. The last picture shows the search space of a query from node 3 to node 2. We can observe, that only edges towards higher levels are relaxed. Furthermore, the node 1 has not been settled although it is part of the shortest path from node 3 to node 2. However, note that at this point, we only computed the distance between the nodes and possibly need to unpack the shortest path.

In our work, we use the original code provided by Geisberger et al. using the parameter set of the *aggressive* variant of contraction hierarchies [GSSD08]. This way, speed-ups of up to 41 051 can be achieved.

Further Information In this work, we only briefly described a small subset of the plethora of available speed-up techniques. For each technique several improvements are available as well as combinations of them, which allow for even superior acceleration; for an overview see the work by Delling et al. [DSSW09a]. Most of the abovementioned techniques can be adapted to work in time-dependent scenarios, which is a challenging task; for an overview see the work by Delling et al. [DW09].

Part I

Artificial Planar Graphs in Experiments

Chapter 3

Planar Graph Generators

This chapter deals with a proper selection of planar graph generators such that experiments conducted with data sets created therewith result in reliable insights. To this end, we analyze a set of selected planar graph generators with respect to both, theoretical and practical aspects. This analysis is guided by the cycle of Algorithm Engineering. In particular, we derive some new aspects of the studied algorithms by their theoretical analysis. On the other hand, we experimentally analyze their implementations and conclude on their practicability and on theoretical aspects. The experimental evaluation comprises network analysis by means of graph properties and algorithmic behavior, in particular kernelization of fixed-parameter tractable problems. In summary, we gain a classification of the studied graph generators that allows for a more reasonable selection, according to the requirements of an experiment.

3.1 Introduction

Planar graphs are an important class of graphs for at least two reasons. On the one hand, the possibility to visualize information in a crossing-free drawing supports grasping the big picture of contained information in a network. On the other hand, many algorithms are tailored to and thus perform much better on this subclass than in the general case. Additionally, planar graphs often serve as a stepping stone for augmenting algorithms to also perform well on larger graph classes. As a result, planar graphs are among the best studied graph classes and researchers often have to deal with this class of graphs.

For an experimental analysis, experimenters have to set up data sets that attest a broad applicability to algorithms. These data sets should be described precisely. However, many publications fail by either sketchy descriptions or inappropriate data-set selection. For instance, works exist that describe the use of the LEDA library [MN99] for random planar graph generation [ADN05, ABN06, BHT06]. They speak either of *the* random planar graph generator or *a specific representative* planar graph generating function of LEDA. In fact, LEDA offers at least five functions to randomly generate planar graphs. They are named similar or, even worse, identical and their behavior is completely different, depending on the parameters used.

Obviously, experimenters want easy-to-use planar graph generators and do not want to care about inherent generator details. They use one generator in the hope that a representative data set is created. As we will see later this is a crucial misestimation that may either completely flaw or at least bias experimental insights. As a result the cycle of Algorithm Engineering can get compromised.

Related Work Ideally, an algorithm should be tested on all graphs of a given class to conclude that it performs well. On several classes of planar graphs *Plantri* [BM07] would be a good choice, as it is capable of generating all *isomorphism-free* graphs of certain graph size. However, we excluded Plantri in our experimental study as it is infeasible to generate all non-isomorphic planar graphs of large size, e. g., the number of 3-connected planar graphs having a node degree of at least 3 of graph size 10 is 32 300 and of graph size 15 already 23 833 988 129. As the sheer number of planar graphs is way too large, it is obvious to ask for a representative sample. Thus, planar graphs that are generated uniformly at random [Fus09, DVW96, BGK07] may be an appropriate choice. The implementation of such a generator is a challenging issue. Hence, non-uniform but easier-to-implement generators gain importance. These can often be found in algorithmic libraries, e. g., the LEDA library [MN99]. Part of this chapter has been previously published [MW11a].

Contribution We empirically study eight selected planar graph generators by means of running time, network analysis with respect to graph properties, and algorithmic behavior, in particular kernelization of fixed-parameter tractable problems. Additionally, we report on the *completeness* of each generator, i. e., whether it is able to generate all planar graphs with positive probability. This work gives an overview of several planar graph generators by means of theoretical background and algorithmic behavior, which enables experimenters to compile representative data sets that allow for a meaningful interpretation of algorithmical experiments and their reproducibility.

Outline First, the functionality and theoretical background of each planar graph generator is described and partially extended in Section 3.2. Additionally, this chapter provides information about the implementations, which includes abbreviated library function calls, and their modifications where it became necessary. Then, the planar graph generators are experimentally surveyed in Section 3.3. There, we first describe our generated data sets and our recommended parameter selection. To classify the graph generators, we perform a network analysis on the generated graphs by means of graph properties in a local as well as a global scope. Afterwards, we sharpen this classification by means of algorithmic behavior, in particular, with a special focus on *fixed-parameter tractable* kernelization algorithms. Finally, in Section 3.4 we conclude with a recommendation which generators should be chosen to create a compilation of data sets that allows for meaningful interpretations of experiments.

3.2 Graph Generators

In this chapter, we consider a graph $G = (V, E)$ to be simple, labeled, undirected and planar, where V is the set of vertices $\{1, \ldots, n\}$ and E is the set of edges. Recall that a planar graph can be embedded in the plane without *edge crossings*. We call a planar graph together with a planar embedding a *planar map*. The selected graph generators presented here are either available in libraries, have low running times, are of theoretical interest, or are based on ideas that are easy to implement. In our opinion this collection represents the important planar graph generators available. We separate the generators into two groups according to the input parameters. The first group are *(n)-generators* that take as a parameter the number of nodes n. Clearly, the number of edges of graphs generated by such a generator is random. The second group consists of the generators additionally taking the number of edges m as a parameter, denoted by *(n,m)-generators*. They allow for an arbitrary number of edges. As always, a degree of freedom implies the difficult decision for the experimenter on how parameters have to be selected reasonably.

For our evaluation we created benchmark sets with each generator. Each planar graph generator had to generate graphs of size 5, 10, 25, 50, 75, 100, 250, 500, 1 000, 2 500 and 5 000 nodes n in increasing size. Since some graph generators have a high running time we set a cut-off time to 14 days. For each graph size n we generated $4n$ graphs. If a graph generator failed to create the requested number of graphs within the time limit we report this behavior in the detailed generator descriptions later on. In the case of the (n,m)-generators, for each node count, graphs were generated having $m = in/4$ edges for $i = 1, \ldots, 11$. The number of graphs generated for each of the n, m combinations is $4n$. All (n,m)-graphs were generated within the time limit. Note that the (n,m)-generators presented here first generate a maximal planar graph and then randomly delete edges until m edges remain.

3.2.1 (n)-Generators

The generators presented here can be subdivided into two groups according to their generation process. The first group consists of *combinatorial generators*, whereas the *geometric generators* form the second group. All of the presented (n)-generators are complete. Two of them are expected to draw planar graphs uniformly at random from the set of all planar graphs with the given vertex set $V = \{1, \ldots, n\}$.

Fusy The planar graph generator developed by Fusy [Fus09] is based on the principles of a Boltzmann Sampler [DFLS04]. Labeled connected graphs of size n are drawn uniformly at random. The running time is in $\mathrm{O}(n^2)$ if the exact number of nodes is sampled or $\mathrm{O}(\frac{n}{\epsilon})$ for an approximated graph size within $[n(1-\epsilon), n(1+\epsilon)]$. The available implementation [Fus05] is the linear-time version of the algorithm. The sampler is based on probabilities described using generating functions, which have to be evaluated laboriously for every number of nodes in a preprocessing step.

Note that the Fusy generator cannot be used as an out-of-the-box generator for two reasons. First, the implementation allows the graphs only to consist either of 1 000, 10 000 or 100 000 nodes. For other graph sizes, the branching probabilities have to be separately computed, e. g., using Maple. Second, the implementation lacks the possibility to specify the size of ϵ, which in general is a drawback as no guarantee on the size of the graph is given. In particular, we aimed at a graph size that deviates 5% from the target size, i. e., within the interval $[0.95n, 1.05n]$ with $\epsilon = 0.05$, and were surprised of the large variance of the size of the generated graphs. The ratio of all generated graphs that have inappropriate size $\hat{n}$ was on average 77% for $3 \leq \hat{n} < 0.95n$ and 21% for $25n \geq \hat{n} > 1.05n$, depending on the targeted graph size. Hence, only 2% of the generated graphs were within the targeted interval.

We extended the generator to additionally accept two parameters to overcome the limited influence on the target size of the generated graph and to allow for an automated generation of multiple instances. First, the parameter ϵ can be specified, which restricts the size of the generated graphs to be within the interval $[n(1-\epsilon), n(1+\epsilon)]$. Our modification simply rejects graphs of inappropriate size. Note that our parameter ϵ does not correspond to the parameter of Fusy, and hence, does not affect the running time of Fusy. Second, the algorithm accepts as parameter the number of graphs to create. This is both, more convenient to use and ensures the control of the random source over several iterations of graph generations. This is a problem not limited to the Fusy generator but affects all fast generators, see Section 3.2.3 for more details.

Markov Chain The planar graph generator by Denise et al. is based on a simple *Markov chain* [DVW96]. The algorithm chooses a pair of nodes u, v at random. Now the transitions are as follows. If edge $e = \{u, v\}$ exists, it is deleted. If not, a check is done whether the graph $G = (V, E \cup \{e\})$ is planar. In the case of planarity the edge e is inserted. Otherwise it is discarded and the Markov chain remains in the current state. The stationary distribution of this Markov chain is uniform over all subgraphs of a given graph. Unfortunately, the mixing rate of the Markov chain is unknown, but the authors expect that the stationary distribution should be reached after $3n^2$ iterations [DVW96]. We verified this behavior in a preliminary test with respect to average degree and degree distribution; see Section 3.3.2 for details. Note that the preliminary test is not part of the experimental study and only briefly described next. We performed the preliminary tests on 40 000 graphs of size 50 generated by $3n^2$, n^3 and n^4 iterations. Generation of larger graph sizes seems infeasible because of the long running time. The outcomes of our preliminary tests were similar, independent of the number of iterations. Hence, we have to perform at most $3n^2$ planarity tests. Checking whether a graph is planar can be done in running time $O(n)$ [HT74]. Thus, the Markov generator has a running time of $O(n^3)$.

Kuratowski The Kuratowski generator is based on the Kuratowski theorem, which states that a graph is planar if and only if no subgraph is present that is a subdivision of K_5 or $K_{3,3}$. The generator starts with a non-planar random graph. Iteratively, the algorithm searches for a K_5 or $K_{3,3}$ subdivision and removes one of its edges until no more Kuratowski violations exist.

The Kuratowski generator can be applied to the complete graph on n vertices. Hence, the generator is complete as its outcome solely depends on the order of Kuratowski violations to remove. However, our experiments, presented later, indicate that the generator is incapable to generate graphs uniformly at random.

Clearly, the running time of the Kuratowski generator strongly depends on the density of the initial graph. To possibly speed up the generation process we analyzed graph properties of graphs that were created from random graphs with edge size m equal to $m = n \log n$ and $m \in \Theta(n^2)$ in a preliminary experiment. On these two sets of graphs, we empirically compared basic graph properties, i. e., the average degree, the degree sequence, the clustering coefficient and the diameter. The outcomes of the preliminary analysis were similar. Thus, graphs with $m = n \log n$ edges were generated as source for applying the Kuratowski algorithm, which yields a running time of $O(n^2 \log n)$. In addition, using sparse random graphs allows us to directly use the LEDA library function `KURATOWSKI` [MN99], which returns a single Kuratowski subdivision. The Kuratowski algorithm, which is to iteratively find a Kuratowski subdivision and remove it, works deterministically, i. e., applying the Kuratowski algorithm twice to the same graph yields identical orders of Kuratowski violations. By using sparse random graphs we overcome this limitation and ensure the generator to be complete. The correspondingly used LEDA subroutine is `KURATOWSKI(graph, V, E, deg)`.

Intersection This geometric (n)-generator is provided by the LEDA library [MN99]. It is based on the intersection of line segments. First, the generator randomly places n line segments in the plane. Then, an intersection finding algorithm constructs the corresponding arrangement. The endpoints of each line segment and their intersection points are interpreted as nodes of which the algorithm keeps the first n nodes. Clearly, this results in a planar graph, which is possibly unconnected. Hence, in the last step missing edges are inserted to make the graph connected. The documentation of LEDA omits the exact running time of the Intersection algorithm and refers to the general section of line segment intersection. However, our experiments indicate the running time to be roughly in $O(n \log n)$. This would correspond to the segment intersection algorithm of Mulmuley [Mul90], which has an asymptotic running time of $O(m + s + n \log n)$ with s the number of line segments.

Note that the Intersection generator is complete as it is possible to randomly generate a corresponding segment arrangement. However, according to our experiments, presented later, it is unlikely that Intersection is capable of generating graphs uniformly at random. The correspondingly used LEDA subroutine is `random_planar_graph( graph, xcoords, ycoords, n)`.

3.2.2 (n,m)-Generators

Similar to the (n)-generators, (n,m)-generators rely either on combinatorial or on geometric approaches. Combinatorial generators triangulate the graph while adding nodes to it. Geometric generators first place points at random in a uniformly sized rectangle. Afterwards the point set is triangulated. Note that the combinatorial generators create maximal planar graphs whereas the geometric ones generate *planar triangulations*, i. e., each face is bounded by cycles of length 3, except possibly the outer face. No (n,m)-generator is known that creates graphs uniformly at random.

Convex Hull Triangulation The LEDA library [MN99] provides a geometric convex-hull triangulation algorithm (CHT) that is based on the *sweep-line paradigm*. To generate a graph $G = (V, E)$, the CHT generator starts by placing n points at random in the plane. Then, the actual sweep-line algorithm starts. First, the points $p \in P$ are sorted lexicographically, i. e., by x- and then by y-coordinate, such that $p_1, \dots, p_n$ denotes the sorted order. In the second phase, the generator maintains an ordered circular list of nodes that in each step contains the nodes of the current convex hull H. In advance, the first two vertices $v_1 = p_1, v_2 = p_2$ are added to the vertex set and the convex hull. Additionally, the first edge (v_1, v_2) is added to the graph. Then, in each step a vertex $v_i = p_i$ is added to the vertex set of $G = (V \cup v_i, E)$ for $3 \leq i \leq n$. The newly inserted vertex v_i sees a subset of the nodes $h_i, \dots, h_j$ of the nodes of the convex hull, where h_i is the leftmost and h_j is the rightmost visible vertex according to the ordering of the nodes of the convex hull. Now, edges are inserted to each of the visible vertices $h_i, \dots, h_j$. Then, the vertices in between h_i, h_j, i. e., $h_{i+1}, \dots, h_{j-1}$ for $j \geq i+2$, are deleted from the convex hull and replaced by v_i, i. e., the subset of nodes $h_i, \dots, h_j$ becomes h_i, v_i, h_j. In this way, after step i, the current graph consists of a planar triangulation of the first i points. The next vertex is connected to a subset of the previously inserted points lying on the convex hull. This is iteratively done until all points have been processed. The CHT algorithm has a running time of $O(n \log n)$ [MN99]. The correspondingly used LEDA subroutine is `random_planar_map( graph, xcoords, ycoords, int n, int m)`.

Theorem 1. The generator CHT is not complete.

Proof. By construction, when node v_i is processed, node v_{i-1} is part of the current convex hull and can be seen by node v_i. Thus, an edge (v_i, v_{i-1}) is added and v_i becomes part of the new convex hull. Since edges are never removed, the sequence $v_1, \dots, v_n$ is a Hamiltonian path in the output triangulation. Thus, CHT cannot generate Non-Hamiltonian triangulations, which are known to exist [Hel07]. □

Delaunay The LEDA library also contains a Delaunay triangulation algorithm, which can be used to generate planar graphs [MN99]. A Delaunay triangulation of a point set P has the property that the interior of the circumcircle of any triangle of the triangulation does not contain a point of P. Often, a Delaunay triangulation of a point set is preferred over other triangulations as it tends to balance the interior angles of the triangles, which makes them appear to be rounder. In particular, in a Delaunay triangulation the smallest interior angle of any triangle of the triangulation is maximized.

The Delaunay generator first places a set of points P at random in a uniform sized rectangle. Then, these points are triangulated using the Delaunay algorithm provided in LEDA. The implementation provided by LEDA is based on the divide-and-conquer algorithm of Dwyer [Dwy87], which has a running time of $O(n \log n)$. The correspondingly used LEDA subroutine is `DELAUNAY_TRIANG( list L, GRAPH DT)`.

Theorem 2. The generator Delaunay is not complete.

Proof. Not every maximal planar graph is Delaunay realizable [Dil90]. □

Expansion The Expansion generator has been developed by Marcus Krug and Jochen Speck within the scope of a lab course on graph generators in the year 2006 [KS06], however, to us, it is unknown whether this generator has been published prior to this date. The only reference we found is related to the operations behind Expansion, which were first presented by Bowen et al. [BF67], who also showed that planar graphs generated using these operations are complete. However, Expansion uses these operations not directly but subsumes them in a single operation called *expansion step*. The following work on Expansion is part of an unpublished manuscript [KMR11], which is joint work with Marcus Krug and Ignaz Rutter.

To generate a planar graph with n vertices and m edges Expansion starts out with a planar embedding of K_4. It performs $n-4$ expansion steps, each of which adds a vertex to the current graph. Moreover, Expansion adds edges in each step to ensure that the graph is a triangulation.

We now give a description of the expansion step, exemplified in Figure 3.1. Let G be a planar graph with a fixed planar embedding and let $v \in V(G)$ be a vertex with distinct incident edges $e_1 = \{v, u\}$ and $e_2 = \{v, w\}$. Let $N_1(v)$ denote the sequence of neighbors of v as they occur when starting at e_1 and traversing its neighbors in counter-clockwise

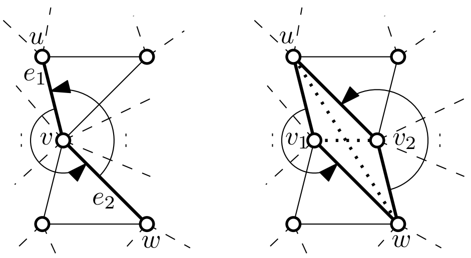


Figure 3.1: Example of an expansion step at v with respect to the thick edges e_1 and e_2, the insertion step randomly chooses one of the dotted lines in the quadrangular face resulting from the split operation.

order until e_2. Similarly, let $N_2(v)$ denote the sequence of neighbors of v in counter-clockwise order, starting from e_2 and ending at e_1. Note that we assume that the edges e_1 and e_2 are considered in both orders, and hence we have that $N_1(v) \cap N_2(v) = \{u, w\}$.

Now the expansion step works as follows. We first remove the node v and create two new nodes v_1 and v_2. We connect v_1 to all vertices in $N_1(v)$ (in the same counter-clockwise order), and analogously, v_2 to all vertices in $N_2(v)$. The cyclic order at the nodes u is such that v_1, v_2 occur exactly in this order at the prior position of v, for w the order is reversed; see Figure 3.1 for an example. Note that this results in a quadrangular face uv_1wv_2. To regain a triangulation, the expansion step finally randomly inserts one of the two diagonals $\{u, w\}$ or $\{v_1, v_2\}$. We now analyze the running time of the Expansion generator.

Theorem 3. The Expansion generator has worst-case running time $\Theta(n^2)$.

Proof. Recall that the generation of an n-vertex graph takes $O(n)$ expansion steps. It is not hard to see that a single expansion step takes time proportional to the degree of the node that is expanded. Since the maximum degree is bounded by $O(n)$, each expansion step takes at most $O(n)$ time, and hence the worst-case running time is bounded by $O(n^2)$. We now turn to the lower bound.

A double wheel graph is a graph that consists of a cycle and two additional *central vertices* that are connected to all vertices of the cycle; see Figure 3.2a. Suppose we wish to generate a graph with n vertices, and as an intermediate step we arrive at a double-wheel graph with $n/2$ vertices, i.e., its central vertex v has degree $n/2 - 2$. If all of the following $n/2$ expansion steps choose to expand the node with the highest degree with respect to two consecutive edges, then the maximum degree never decreases. Depending on the choice of the diagonal that is inserted it may even increase by 1; see Figure 3.2b. Hence, we have a linear number of steps, each taking linear time, thus resulting in a running time of $\Omega(n^2)$. □

Fortunately, the running time is not quite as bad as it seems. In particular, we can show that the expected running time is in fact linear.

Theorem 4. The node expansion generator has expected running time $\Theta(n)$.

Proof. Let X be a random variable describing the running time of the generation process taking n expansion steps. Let further $X_1, \ldots, X_n$ be random variables describing the running time of the ith expansion step. Clearly, by linearity of expectation we have $\mathbb{E}[X] =$

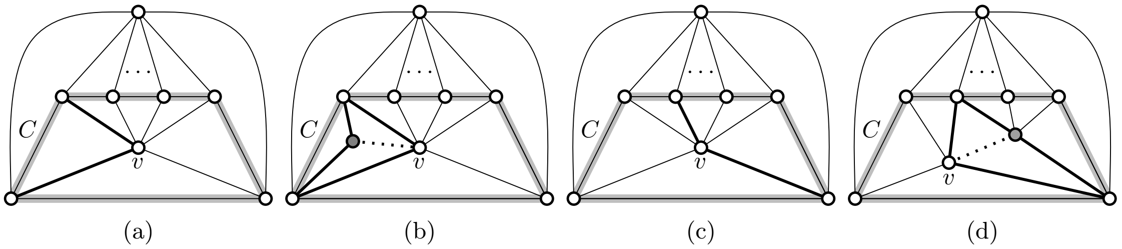


Figure 3.2: Worst-case example for the Expansion generator: A wheel graph with an additional vertex connected to all vertices of the gray marked cycle C.

$\sum_{i=1}^{n} \mathbb{E}[X_i]$. Recall that the running time of an expansion step is proportional to the degree of the expanded node. Since at each step, the node to be expanded is chosen uniformly at random, this running time is thus proportional to the expected degree of a randomly chosen node. However, the average degree of a planar graph is bounded by 6, and thus we have $\mathbb{E}[X_i] \leq 6$ for $i = 1, \ldots, n$. This proves the theorem. □

Efficient Expansion We now propose an alternative implementation of Expansion that keeps the same favorable properties but additionally has an improved worst-case running time of $O(n \log n)$.

There are two main issues that keep the running time high. First, we need to be able to quickly choose two random edges incident to a common vertex. Second, we need to perform the actual expansion step. In a standard circular adjacency list representation of a planar graph it is not possible to quickly find two random edges incident to a common vertex. We therefore store for each vertex an additional array that contains its incident edges in arbitrary order. We assume that the entries are equipped with pointers so that from an element of an access array of a vertex v, we can obtain the corresponding edge in the incidence list of v in $O(1)$ time, and vice versa. Thus, we can update the array in $O(1)$ time when an edge is added or removed. Using this array, we can now quickly choose two random incident edges at a vertex, simply by choosing random indices into the array, which can be done in $O(1)$ time. This resolves the first issue.

To improve the running time of expanding a node v, we note that it is not strictly necessary to replace v by two new nodes v_1 and v_2. Instead, we add only a single new node v_2 and let v take the role of v_1. We then only have to remove the edges of v that should not be incident to v_1 after the expansion step. If the degree of v_2 is low, this may give a substantial speedup. Clearly, a random diagonal can be inserted in $O(1)$ time. To further improve the running time, we do not always reuse v as v_1 but rather reuse it for the vertex of v_1 and v_2 that has higher degree, as this minimizes the number edges that need to be removed. Thus, expanding the node v with respect to two edges e_1 and e_2 takes time $O(\min\{|N_1(v)|, |N_2(v)|\})$. Note that the size of the smaller set can be computed in the same time as follows. We perform two traversals t_1 and t_2 of the incident edges of v in counterclockwise direction starting at e_1 and e_2, respectively. We take steps of these traversals alternately and terminate both traversals once t_1 hits e_2 or t_2 hits e_1. The traversal that terminates first corresponds to the smaller of the two intervals (e_1, e_2) and (e_2, e_1).

Intuitively, the work we have to perform in an expansion step of v is either small (if the degrees of the resulting nodes are rather imbalanced) or large, but the degrees of the resulting nodes are balanced and thus significantly smaller than that of the original vertex v, and hence such a step cannot occur too often. More precisely, we have the following theorem.

Theorem 5. The implementation of Expansion described above has expected running time $O(n)$ and worst-case running time $\Theta(n \log n)$.

Proof. The argument for the expected running time is the same as in Theorem 4. We now concentrate on proving the worst-case running time.

We use an accounting scheme, where we have an account for each vertex v, and we keep the invariant that each vertex v has at least $\lceil \deg(v) \log(\deg(v)) \rceil$ units on its account, where log denotes the logarithm with respect to base 2. First, we show that when inserting an edge $\{u, v\}$, it is sufficient to add $O(\log n)$ units to the accounts of u and v. To this end, assume that the degree of u is k, and thus adding the edge uv raises the degree to $k + 1$. To ensure the account invariant, we may have to add $\lceil (k+1) \log(k+1) \rceil - \lceil k \log k \rceil$ units. A simple computation shows that $\lceil (k+1) \log(k+1) \rceil - \lceil k \log k \rceil \leq (k+1) \log(k+1) + 1 - k \log k = \log(k+1) + 1 + k \log(1 + 1/k)$. Using the fact that $\log(1+x) = \ln(1+x)/\ln 2 \leq x/\ln 2$ for $x \in \mathbb{R}^+$, we see that the whole expression is in $O(\log k)$, and hence in $O(\log n)$ since $k \leq n$. We can thus add and remove single edges at an amortized cost of $O(\log n)$ while maintaining the account invariant. It remains to show that the additional costs paid by the insertion operations are enough to pay the expansion operations. More precisely, we claim the following.

Claim. The cost of splitting a node v of degree k and filling the accounts of the resulting two nodes v_1 and v_2 is bounded by $(k+2) \log(k+2) + 2$.

Let $e_1 = \{v, u\}$ and $e_2 = \{v, w\}$ be two edges incident to v. Without loss of generality assume that $|N_1(v)| \leq |N_2(v)|$, i. e., we reuse v as v_2. Let $k_1 = |N_1(v)|$ and $k_2 = |N_2(v)|$, then we have $k_1 + k_2 = k + 2$ due to the common neighbors u and w of v_1 and v_2. As shown above, the expansion step takes $O(k_1)$ time, and additionally, we have to make sure that the account invariant of v_1 and v_2 is satisfied.

The cost of the operation thus sum to $\lceil k_1 \log k_1 \rceil + \lceil k_2 \log k_2 \rceil + k_1 \leq k_1(\log k_1 + 1) + k_2 \log k_2 + 2 = k_1 \log 2k_1 + k_2 \log k_2 + 2$. Clearly, it holds that $k_2 \leq k+2$, and since $k_1 \leq k_2$ we also have $2k_1 \leq k+2$. By applying these observations, we get $k_1 \log 2k_1 + k_2 \log k_2 + 2 \leq k_1 \log(k+2) + k_2 \log(k+2) + 2 = (k+2) \log(k+2) + 2$. This proves the claim.

We now compare the cost with the account of v, containing at least $k \log k$ units. The cost that cannot be paid from the account of v thus is $(k+2) \log(k+2) + 2 - k \log k = k(\log(k+2) - \log k) + 2 \log(k+2) + 2 = k \log(1 + 2/k) + 2 \log(k+2) + 2$. Again we have that $\log(1 + 2/k) \leq 1/\ln 2 \cdot 2/k$, and thus $k \log(1 + 2/k) + 2 \log(k+2) + 2 \leq 2 \log(k+2) + 2/\ln 2 + 2$, which is in $O(\log n)$. Hence, the cost of each expansion operation can always be paid from the account of the expanded node plus an additional cost of $O(\log n)$. Since there are at most $O(n)$ operations, and each is charged at most $O(\log n)$ units, the total amount of work performed during all operations is in $O(n \log n)$.

To see that this bound is tight, consider again a double wheel graph where the center vertex has degree $n/2 - 2$; see Fig. 3.2c. If the generation process splits in half this center vertex (Figure 3.2d shows an example of this), and recursively each of the resulting

Table 3.1: Summary of the studied generators. Results marked with an Asterisk can be found in this work.

generator	parameter	complexity	complete	uniform
Fusy [Fus09]	n	$O(n)$	yes	yes
Intersection [MN99]	n	$O(n \log n)$	yes	no*
Kuratowski [MN99]	n	$O(n^2 \log n)$	yes	no*
Markov [DVW96]	n	$O(n^3)$	yes	yes
CHT [MN99]	n,m	$O(n \log n)$	no*	no*
Delaunay [MN99]	n,m	$O(n \log n)$	no [Dil90]	no
Expansion [BF67]	n,m	$O(n^2)^*$, expected $O(n)^*$	yes	open
Efficient Expansion*	n,m	$O(n \log n)^*$, expected $O(n)^*$	yes	open
Insertion [MN99]	n,m	$O(n)$	no	no

vertices, the running time can be described by the recurrence $T(n/2) \geq 2 \cdot T(n/4) + n/2$, which solves to $\Omega(n \log n)$. This concludes the proof. □

Insertion The combinatorial Node Insertion algorithm is available in the LEDA library [MN99]. It starts out with a triangle and iteratively picks a face f uniformly at random, inserts a new vertex and connects it to all vertices of f. The running time of the Insertion generator is in $O(n)$. The correspondingly used LEDA subroutine is `random_planar_map( graph, int n, int m)`.

Theorem 6. The generator Insertion is not complete.

Proof. Graphs generated by the Insertion generator always contain a node of degree at most 3. Hence, *k-regular graphs* with $k > 3$ cannot be generated. However, k-regular planar graphs with $k = 5$ are known to exist [DKS09]. □

3.2.3 Summary

Table 3.1 gives an overview of the origin, complexity, completeness and possible uniformness of generation. We do not report on uniform generation of subclasses of planar graphs. Hence, an incomplete generator implies non-uniform generation. Due to their complexity Markov and Kuratowski were not capable of generating graphs larger than 500 and 1000 nodes, respectively. Hence, these generators cannot be recommended for large scale planar graph generation. In contrast, the other generators perform quite well. An exception is Delaunay; it generated all requested graph sizes but due to its complexity it may not be capable of generating really large graphs. Note that the running time of LEDA's Intersection implementation is not specified but experiments indicate a running time of $O(n \log n)$. Additionally, note that the completeness of Intersection and Kuratowski follows from their definitions and that uniform generation is quite unlikely, which is shown by our experiments.

Implementation Side Notes One of the crucial points of random generation is to employ reasonable random sources. Often, these are provided by programming languages or by libraries, e. g., LEDA. However, we have to initialize them in a proper way. A common approach is to use the the current time of the system clock. This was our first approach, too. We wanted to generate graphs one by one giving necessary parameters by command line scripts. Doing so, each of the fast graph generators produced many identical graphs. It turned out they were so fast that the random source in our generators were started several times with the same initialization value. The reason was, that the resolution of the system clock queried is one second, and thus, too low. In other words, graphs generated within one second using the same parameters were identical. This problem can be overcome in at least two ways. First, the quick solution is to initialize the random source with the system time + process id. Second, the generator can be extended to accept either a parameter file or the number of graphs to generate and a seed for a random source, which is used to initialize the random source of the generator. Thus, the generation can be repeated with similar outcomes.

3.3 Experiments

We already described the performance of the generators, which together with the theoretical background allows to decide whether a generator is capable of generating graphs of favored size. In this chapter we further classify the presented planar graph generators using concepts of *network analysis* [BE05]. In particular, we analyze the generated graphs by means of basic graph properties in a local as well as a global scope. Additionally, we strengthen the basic classification with respect to algorithmical behavior. Altogether, this result will help experimenters to reasonably compile data sets. But first, we report on the generation of our data-sets, which are then used in our experimental studies.

3.3.1 Dataset Generation

Usually, in experimental studies a broad applicability of newly developed algorithms should be confirmed. In our case it is not feasible to generate all planar graphs of a certain size n, especially when algorithms should be tested on larger instances. Obviously, a representative sample should be chosen. The (n)-generators sample graphs by their underlying behavior, which includes the number of generated edges. In contrast, the (n,m)-generators' number of edges can be set arbitrarily. This raises the difficult question how to set the number of edges in a representative way. A first idea is to set the number of nodes and edges the (n,m)-generators have to create to the number of nodes and edges created by an (n)-generator, which allows to compare them with each other.

As aforementioned, Fusy and Markov are expected to generate graphs uniformly at random but Markov was not capable of generating graphs of larger size. Thus, we generate a data set using the number of nodes and edges predefined by Fusy as input for the (n,m)-generators. This distribution is referred to as *Fusy distribution*. Each generator created 40 000 graphs whose number diverged by ±5% from 1 000 nodes as described in Section 3.2.1.

Giménez and Noy [GN09] found that the average degree of a randomly selected planar graph is asymptotically normal. A graph of node count n is expected to contain $\mu \approx 2.21n$ edges on average with a standard deviation of $\sigma \approx 0.656\sqrt{n}$. Fusy and Markov exactly fit the theoretical distribution. A consequence of this distribution is that with increasing node count the average degree of a randomly generated planar graph gets closer to its

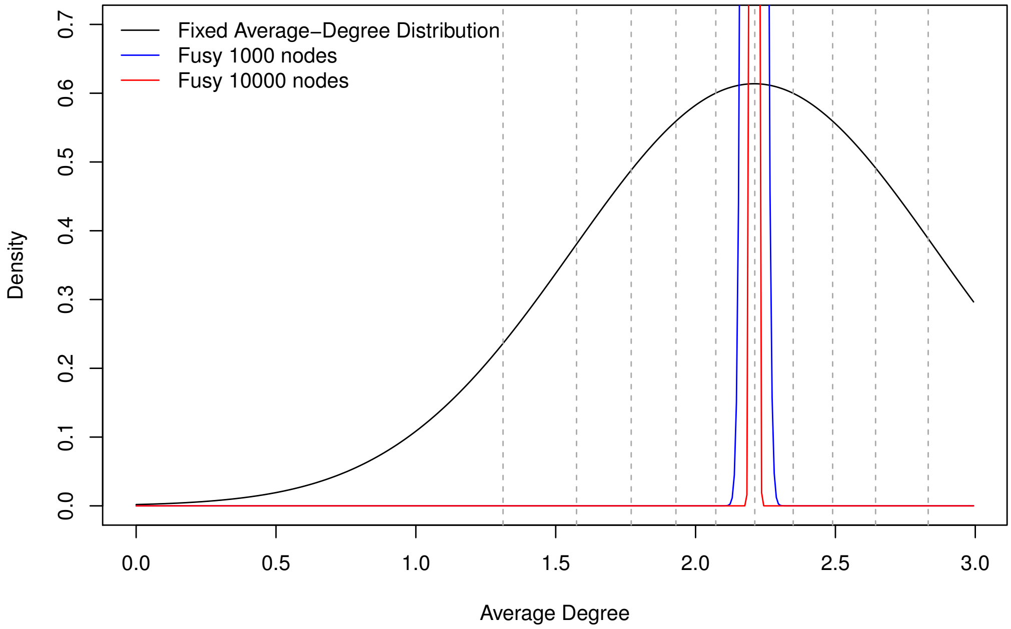


Figure 3.3: Density of the Fixed Average-Degree distribution and partly of Fusy distribution for 1 000 and 10 000 nodes. Vertical dotted lines split the number of graphs into eleven groups of equal size, which is used later in plots to demonstrate how values of properties behave for different parts of the whole distribution. The maximum density for $\text{Fusy}_{1\,000}$ is 19.16 and for $\text{Fusy}_{10\,000}$ is 52.06.

expected value. In particular, the standard deviation of the average degree of a planar graph containing 1 000 nodes is $\sigma_{1000} \approx 0.0207$, which states with high probability only a small subset within the interval of possible edges $m \in [0, 3n - 6]$ is generated. Thus a fourth data set was created by the (n,m)-generators whose average degree correspond to a normal distribution with mean $\mu = 2.21$ and standard deviation $\sigma = 0.65$. This data set of 40 000 values was chosen to span a larger part of the interval $[0, 3n - 6]$, which should give a better overview how (n,m)-generators behave on a spread distribution. This distribution is referred to as *Fixed Average-Degree distribution*. In Figure 3.3, we show the density of the Fixed Average-Degree distribution and partly of the Fusy distribution, which has a maximum density of 19.16 for $\text{Fusy}_{1\,000}$ and 52.06 for $\text{Fusy}_{10\,000}$. Additionally, the Fixed Average-Degree distribution is split into eleven groups of equal size. These groups are used in plots of the experimental section to demonstrate how values of properties change over the whole distribution.

Note that from here on we name the data sets, which are generated according to the above distributions, by the respective generator name and refer by it to these data sets. Additionally note that all plots showing the (n)-generators base on a graph size of 500 for Intersection, Kuratowski and Markov (lowest common denominator). The smallest size of Fusy graphs is 1 000 so this value was taken. The reason for letting Fusy be the exception was that Kuratowski was able to generate only very few graphs of size 1 000.

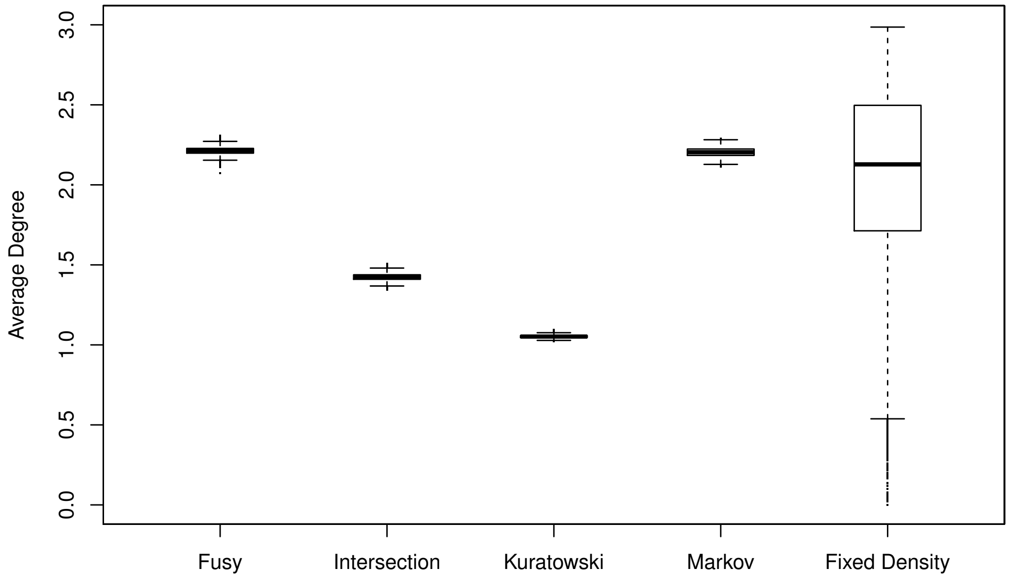


Figure 3.4: Inherent distributions of the average degree of the (n)-generators and the Fixed Average-Degree distribution.

3.3.2 Basic Network Analysis

In the following, we perform a network analysis [BE05] on the generated graphs by means of basic, local and global graph properties. We will see how the basic graph properties directly influence the other properties, and with that, experiments in general. Thereby, a classification of the generators is done. This gives a first sketch which generators should be used for a representative data-set compilation.

Average Degree One of the very basic questions answered recently by Giménez and Noy [GN09] was the expected average degree of a planar graph drawn uniformly at random; see Section 3.3.1. According to the theoretical results the distributions of the average degree of Fusy and Markov match the expected distribution, see Figure 3.4. Intersection and Kuratowski can each be clearly separated from the other generators. For comparison, the Fixed Average-Degree distribution used by the (n,m)-generators is shown.

Degree Distribution The average degree describes a graph only roughly. A better way to reflect the structure of a graph is its degree distribution. Hence, in Figure 3.5 boxplots of the degree distribution of graphs generated by the (n,m)-generators according to the Fusy distribution are shown. Note that for an better overview, we cut off nodes with a degree larger than 10 and removed the outliers. For an easier comparison, we also show the average values of the degree distribution and cut off nodes with a degree larger than 13. All graphs except those generated by the Fusy generator consist of more than one connected component. Thus, Fusy lacks nodes with degree zero. The plot shows that CHT, Expansion and Fusy behave very similar and can hardly be separated from each other. In particular, this holds for Expansion and Fusy. Insertion has an increased number of nodes of degree 2 and the peak at degree 3. This slope then decreases rapidly. The Delaunay generator has a larger mean. Its number of nodes having a degree less than 4 is much smaller compared to the other generators. The peak is at degree 4. More than 50% of the nodes have either degree 4 or 5. The slope then decreases very quickly. Although the boxplots of the degree distributions given in Figure 3.5 have similar trends, the average values differ quite much. However, in averaging a value we might loose much information. Hence, we also analyzed the entropy of the degree sequence and present the outcomes in Figure 3.6. Again, Delaunay and Insertion can be separated clearly from the others. The normalized entropy separates Insertion from the other generators, whereas the bit entropy of Delaunay is much lower than that of the others. Additionally, the bit entropy as well as the normalized entropy separates Expansion from Fusy to a certain extent, which was not possible when comparing the raw and the average values of the degree distribution.

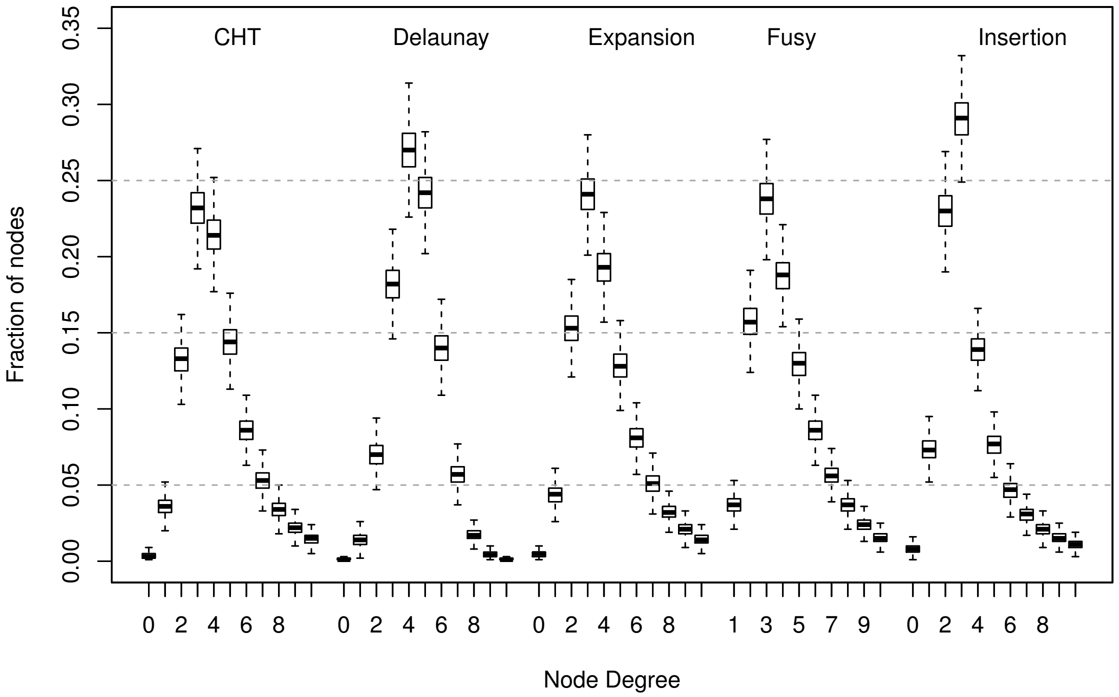
We omit plots of the entropies of the (n)-generators as they do not reveal new insights but briefly describe their properties. The bit entropy of Fusy and Markov are very similar, whereas the normalized entropy shows some minor differences between them. Similar to the average-degree distribution, Intersection and Kuratowski can clearly be separated from each other as well as from Fusy and Markov.

We further study the degree distributions by the number of entries that are necessary to describe a graph. Our assumption is that it correlates with the graphs' structural complexity. Note that we do not expect this to correlate with algorithmic complexity. In Figure 3.7 the number of degree sequence entries of the $(n{,}m)$-generators using the Fusy distribution as well as the Fixed Average-Degree distribution are shown. Again, Delaunay and Insertion can be separated clearly from the other generators, which have similar distributions of the number of entries of the degree distribution. The Delaunay generator has nearly a constant number of entries for all graphs and thus, exhibits a regular structure. In contrast, the Insertion-generated graphs show the largest variance.

Connected Components In theory a planar graph created uniformly at random is connected with high probability [GN09]. The graphs produced by the (n)-generators are all connected, partly by construction. On the contrary, the $(n{,}m)$-generators are usually not connected. Recall that the $(n{,}m)$-generators first generate a maximally planar graph and then remove the desired number of edges. Hence, in this process the graph can get disconnected. Nevertheless, the $(n{,}m)$-generators generate a graph that contains a single, large connected component if the average degree exceeds a certain value. In Figure 3.8 the ratio of the largest connected component's size and the total size of the graphs generated by the $(n{,}m)$-generators according to the Fixed Average-Degree distribution are shown. The data points are sorted increasingly from left to right according to the average degree of the represented graphs. With increasing average degree of a graph, a giant component emerges, which consists of a large fraction of the graph. The giant component has a size of at least 90% at an average degree of 1.6 for CHT, 1.45 for Delaunay, 1.65 for Expansion and 1.8 for Insertion graphs. The size of this large connected component grows further with increasing average degree.

Diameter So far we analyzed elementary properties that were easy to obtain. By this, we got an idea which generators could complement each other. This classification will now be extended by a global property, namely the *diameter* of a graph. Recall that the diameter of a graph is the length of the longest shortest path between any pairs of nodes in a graph. Thus, their values describes how compact a graph is.

Figure 3.9 shows the diameters of graphs generated by the (n)- and $(n{,}m)$-generators



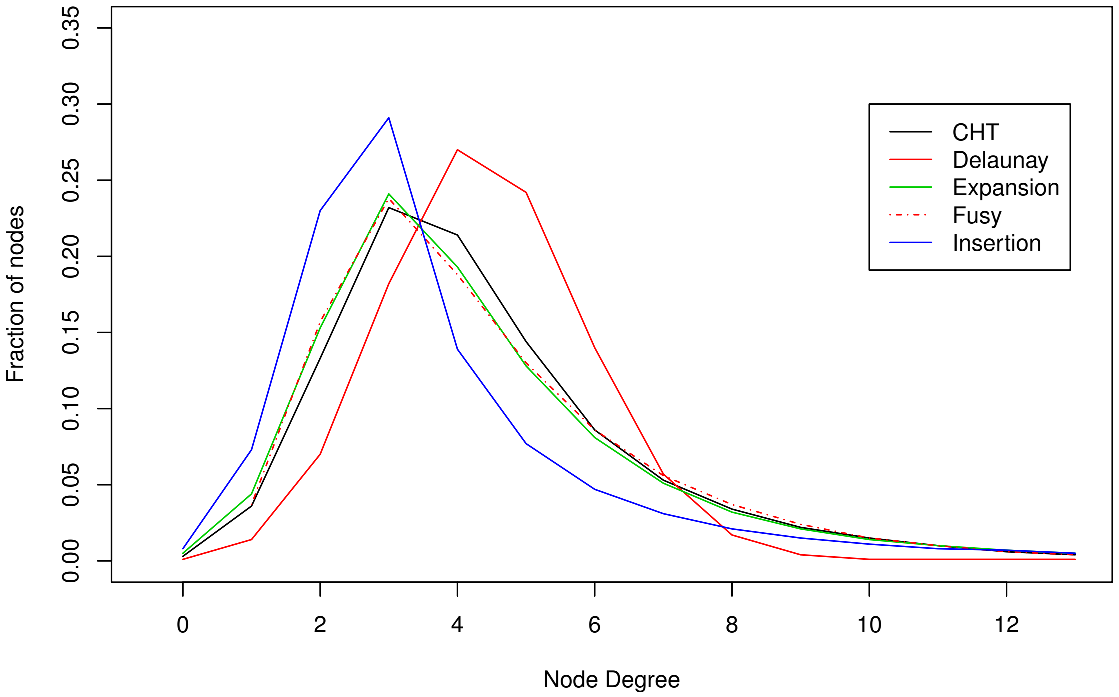


Figure 3.5: On the top, boxplots of the degree distributions of graphs generated by the (n,m)-generators using the Fusy distribution. For a better overview, outliers have been removed and the degree distribution has been cut off at nodes of degree 10. On the bottom, lineplots of the average values of the above boxplots, which now have been cut off at nodes of degree 13.

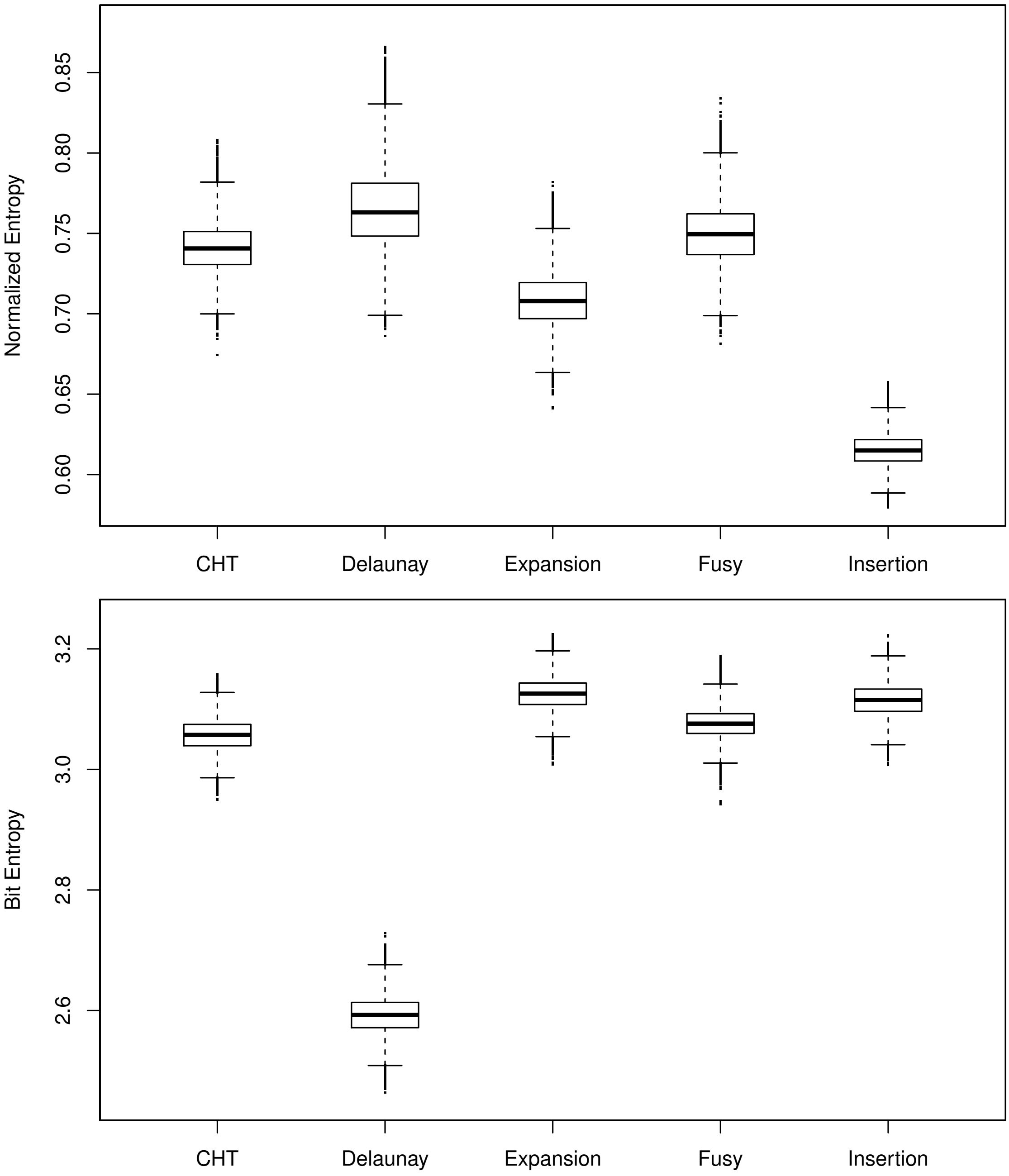
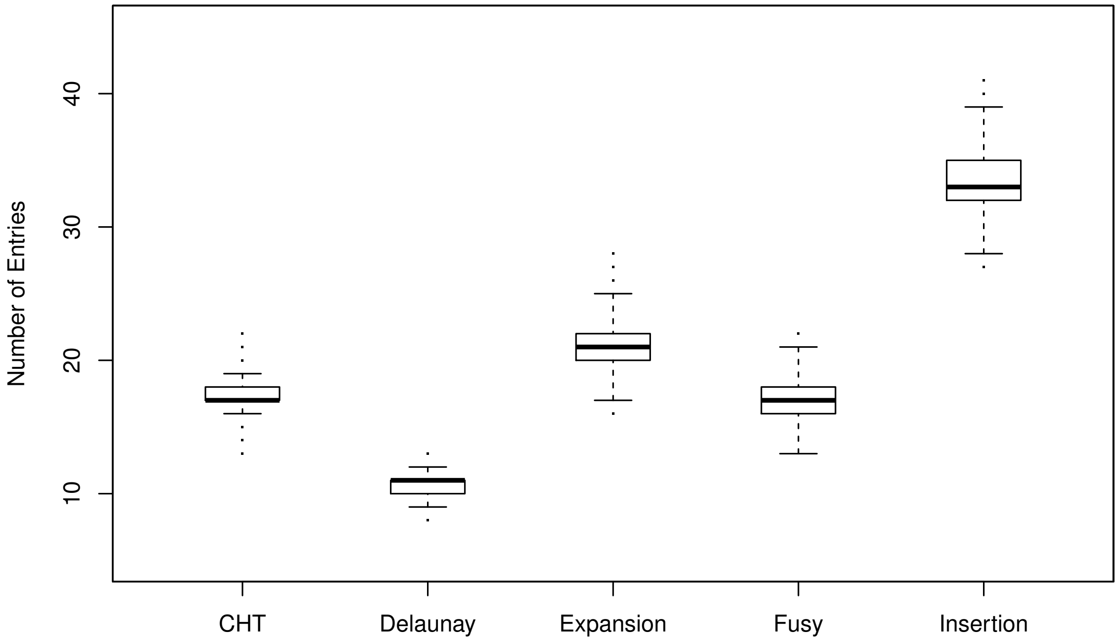


Figure 3.6: The entropy of the degree sequence of graphs is shown that are generated by the (n,m)-generators according to the Fusy distribution. On the top, the normalized entropy clearly separates Insertion from the other generators. On the bottom, the bit entropy of Delaunay is less than that of the others.



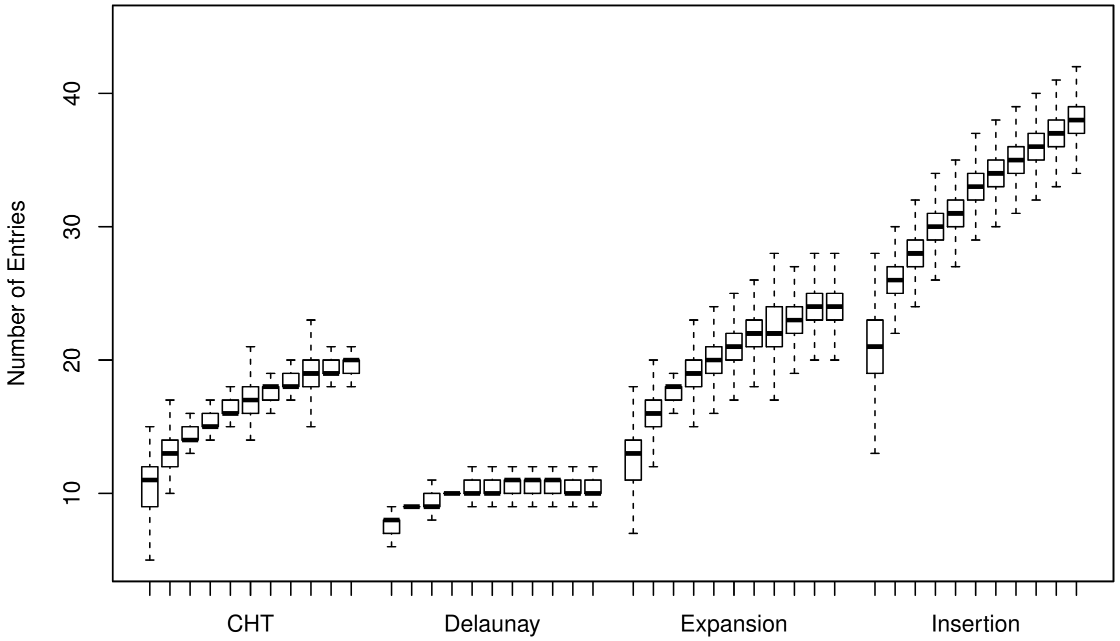


Figure 3.7: The number of values of the degree sequence is shown. Both show graphs generated with (n,m)-generators. On the top, graphs underlying the Fusy distribution are shown. The graphs on the bottom are generated using the Fixed Average-Degree distribution, which have been split into eleven groups of increasing edge count for each generator, outliers have been removed.

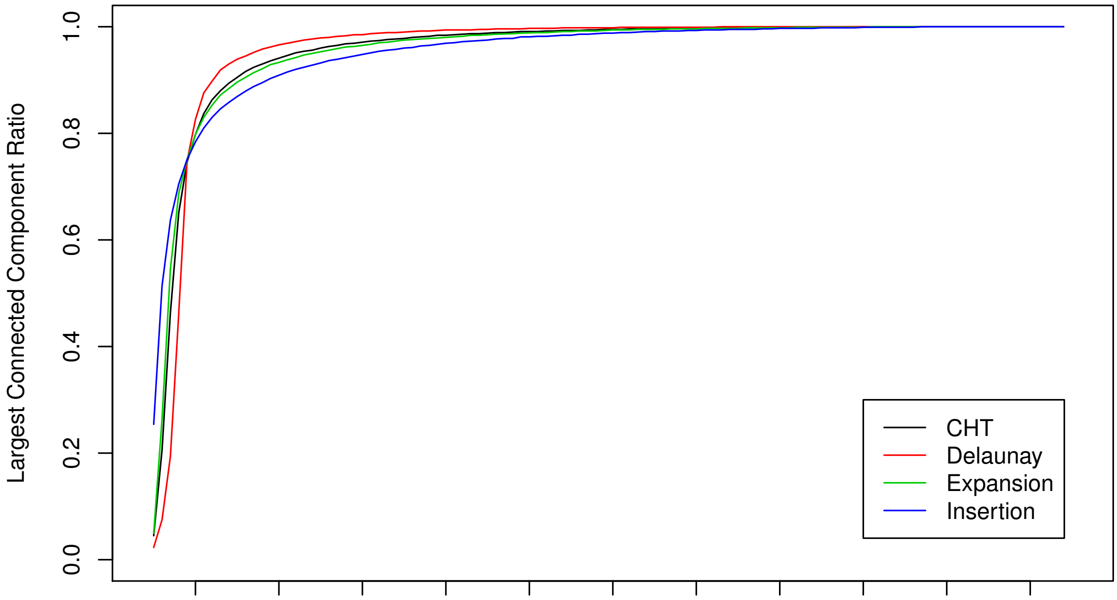


Figure 3.8: The size of the largest connected component in relation to the total size of the graph. Shown are CHT-, Delaunay-, Expansion- and Insertion-generated graphs according to the Fixed Average-Degree distribution. The data points are sorted increasingly from left to right according to the average degree of the represented graphs.

according to the Fusy distribution. The diameters of Intersection and Kuratowski are very large compared to the other generators. The graphs of both generators are connected and simultaneously have very low average degrees. Thus, the structure is more tree-like, which results in high diameters. The (n,m)-generators exhibit some peculiarities. For example, Insertion has a very small diameter whereas Delaunay has the largest mean diameter. Additionally, Insertion and CHT have small spreads. This difference can be explained by their generation process. Delaunay generates a comb-like regular structure whereas Insertion and CHT insert shortcuts, i. e., long edges, which reduce both the diameter and its spread. Altogether, the studied diameter of graphs confirms the basic classification on a global scale.

Clustering Coefficient We advance in our basic classification of the planar graph generators by the analysis of a local property. Namely, we measure the density of each vertex' neighborhood, in particular we compute the *clustering coefficient* of a graph [SW05]. Recall the definitions of Section 2.1.3. The clustering coefficient of a vertex is defined in terms of triples and triangles. A complete subgraph of three nodes is called a *triangle*. A *triple* at a node v is a path of length two for which v is the center node. The clustering coefficient of a node is the number of triangles this node is part of divided by the number of triples at this node.

Figure 3.10 shows the clustering coefficients of the (n)-generators and the (n,m)-generators using the Fusy distribution. Clearly, Kuratowski and Intersection can be separated from the other generators. This can be explained with the average degree being very small for Kuratowski. The graph is tree-like and thus its clustering coefficient is very small. The Intersection graphs have a slightly larger average degree that allow for a more

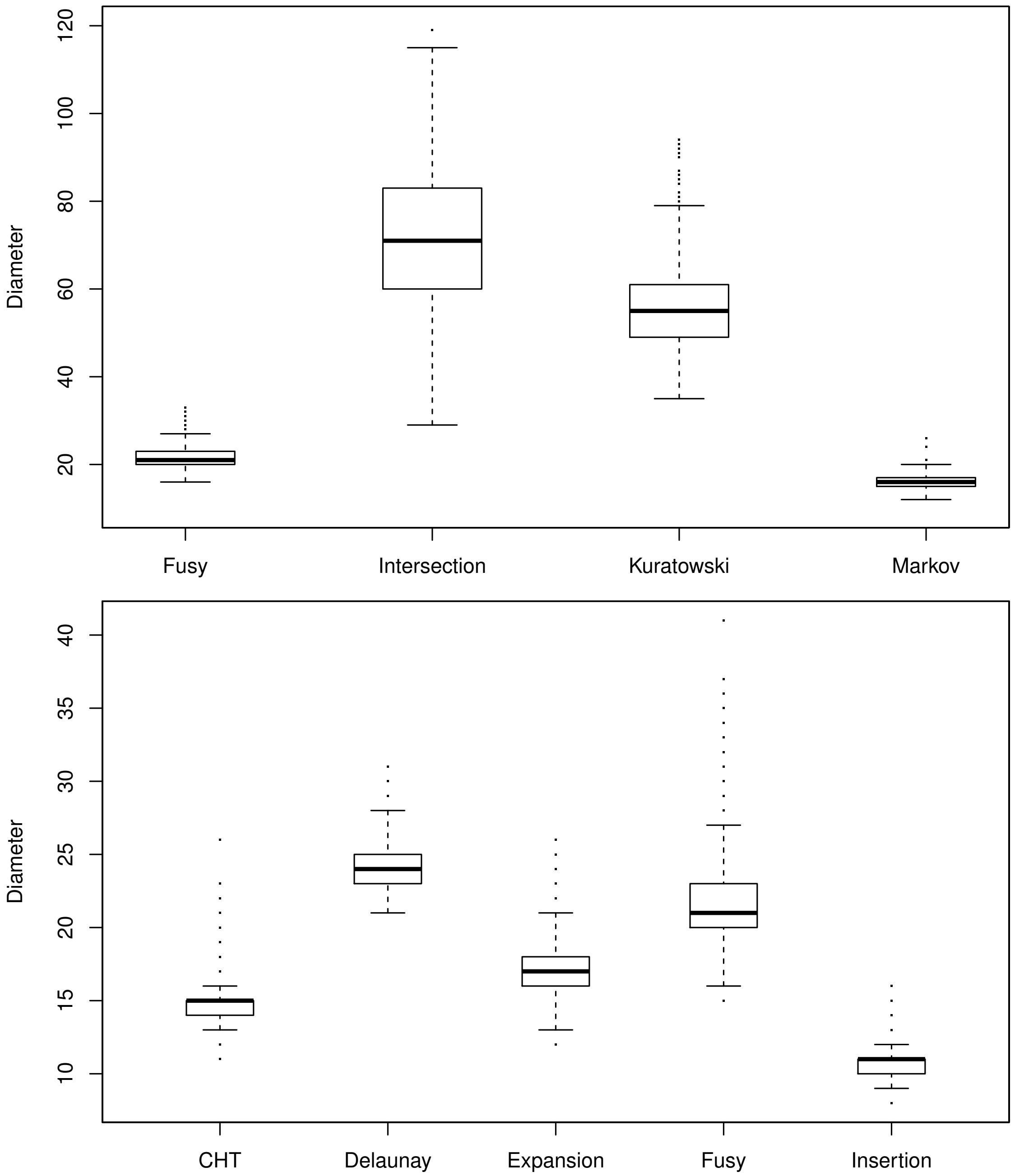


Figure 3.9: On the top, the diameters of the (n)-generators. On the bottom, the diameter of the (n,m)-generators using the Fusy distribution.

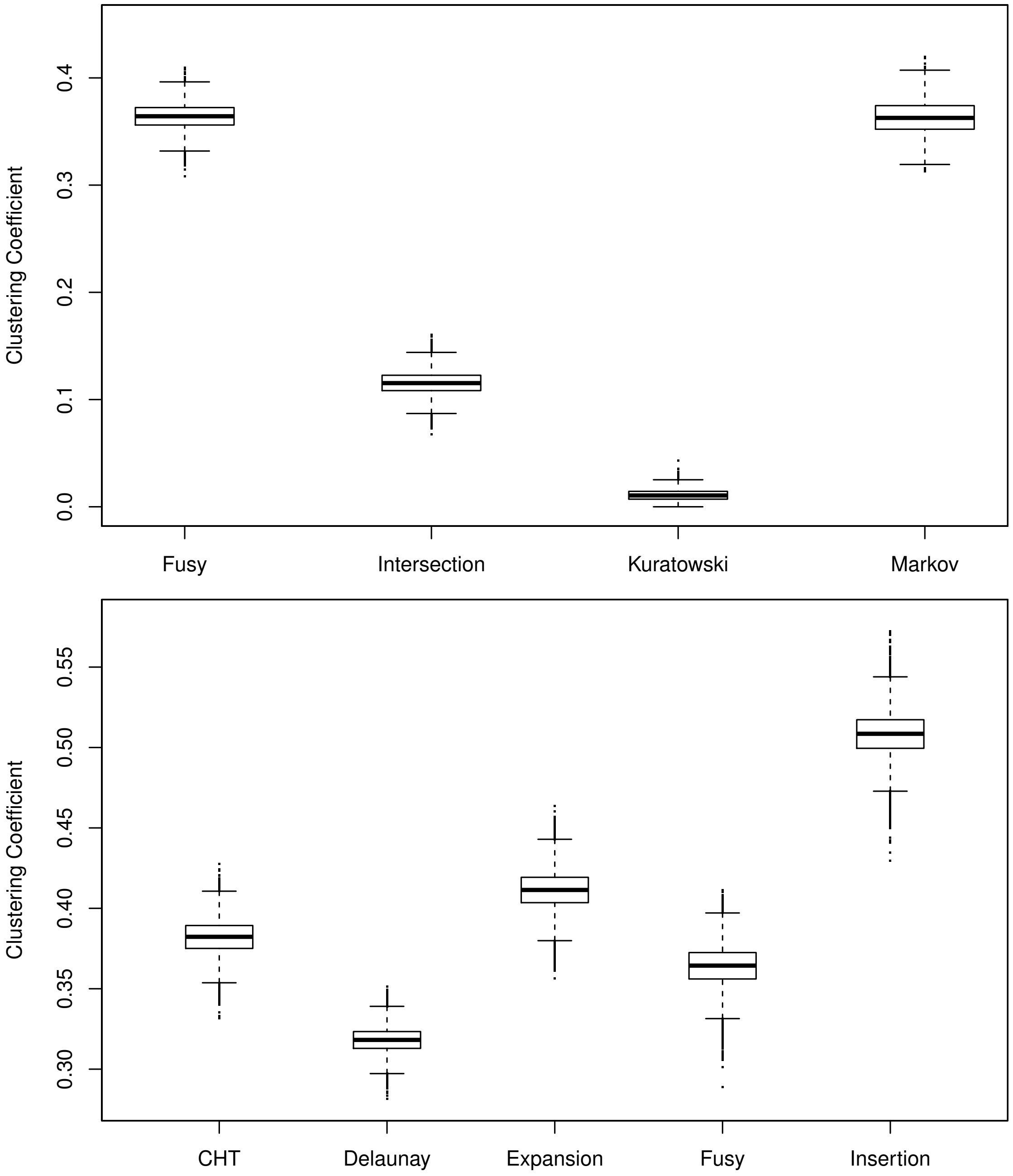


Figure 3.10: On the top, clustering coefficients of the (n)-generators. On the bottom, clustering coefficients of the (n,m)-generators using the Fusy distribution.

complex graph than a tree. Hence, it exhibits a slightly larger interconnection. In the case of the (n,m)-generators three groups can be separated. The first consists of the Delaunay-generated graphs. Due to the generation process, the clustering coefficient is less than for the other groups. In particular, the number of nodes having a degree of 4 or higher is significantly larger than those of the other generators. Hence, it is rather unlikely that all possible triangles are realized. The second group consists of CHT, Expansion and Fusy. Minor differences within this group can be seen but they cannot be separated clearly. The Insertion-generated graphs represent the third group. Their generation process leads to a high interconnection of the neighbors, which results in a very large clustering coefficient. In particular, most of the nodes have a small degree and, thus, it is more likely that the possible triangles are realized.

Figure 3.11 gives an overview of the clustering coefficients of the graphs generated by the (n,m)-generators according to the Fixed Average-Degree distribution. Trends similar to the graphs generated according to the Fusy distribution can be seen. The clustering coefficient of Insertion grows faster with increasing average degree and, thus, can be clearly separated from the others. In contrast, the clustering coefficient of Delaunay grows much slower than the others. In summary, for the Fixed Average-Degree distribution we can observe a behavior similar to the Fusy distribution.

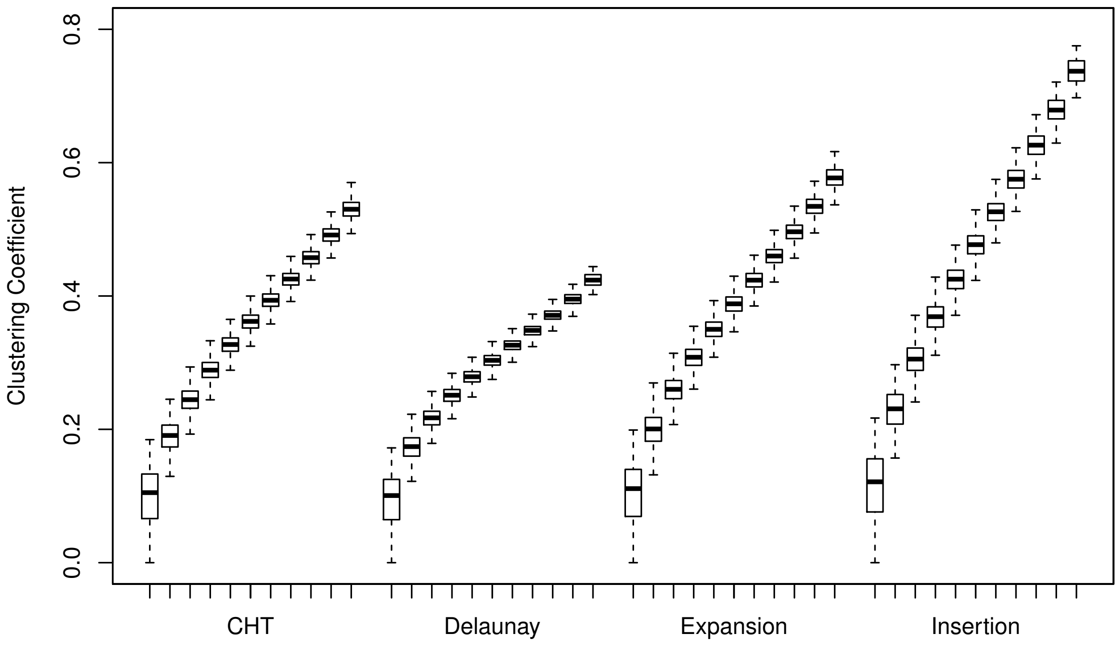


Figure 3.11: Clustering coefficients of the (n,m)-generators using the Fixed Average-Degree distribution. The total number of generated graphs has been divided into eleven groups of increasing edge count, outliers have been removed.

Transitivity The clustering coefficient of a graph is the average over the clustering coefficient of each node in the graph. Hence, it is the average of a local property. Similar to the clustering coefficient, the *transitivity* of a graph is defined in terms of *triangles* and *triples*. However, these are set in relation in a global scope. The transitivity of a graph is three times the number of triangles of the graph divided by the number of triples of the graph, see Section 2.1.3 for details. In other words, this is the ratio between the number of realized triangles in a graph and the number of possible triangles.

Recall that the number of triples of a node v depends on its degree, i. e., the number of triples is $(\deg(v)^2 - \deg(v))/2$. How does the transitivity evolve with increasing degree of a node? Take for example the subgraph that has all triples realized. A node of degree 3 implies a K_4. For a node of degree 4 this would already be a K_5, which is not planar any more. In particular, we observe that the number of triples grows quadratic in terms of the degree of a node, while the number of triangles grows only linear due to the planarity constraints. We conclude that nodes with large degree have a major impact on the transitivity. Hence, we expect the transitivity of a graph to be smaller than its clustering coefficient, especially, if it contains nodes that have a large degree.

Indeed, Figure 3.12 shows that the transitivity of the graphs generated by the (n,m)-generators according to the the Fixed Average-Degree distribution is much smaller than the corresponding values of the clustering coefficient. Interestingly, the values of the transitivity of the graphs show an inverted behavior compared to those of the clustering coefficient. Again, the graphs generated by CHT and Expansion have similar values. However, Insertion-generated graphs exhibit a much smaller transitivity than the other generators, while the transitivity of Delaunay-generated graphs exceeds those of the others. In particular, despite the Delaunay-generated graphs have an increased number of nodes of degree 4 to 7 compared to the other generators the amount of nodes with degree 8 and larger seem to easily compensate this. Notably, the Insertion-generated graphs contain nodes of larger degree than graphs generated by the other generators. In particular, this holds for nodes of degree 13 and larger. For these two reasons, the transitivity of Insertion-generated graphs is lower than the ulterior generated graphs.

Modularity An important question in network analysis is whether a graph exhibits community structures, i. e., whether subgraphs have more internal than external relations. Tackling this question originates in the research field of *clustering*. A clustering of a graph is a partition of its vertex set into disjoint subsets, which are called *clusters*. Several approaches and measures exist to perform a clustering of a graph and then rate it according to a measurement. Usually, a clustering is rated good, if the number of edges within each cluster is much higher than between the clusters and the clusters are balanced in terms of size. In the following, we will concentrate on the index *modularity* as it is one of the most popular measures. Modularity relates the number of edges within the clusters to the expected value of akin number, see Section 2.1.3 for details. The formula of modularity implicitly penalizes nodes of high degree as these are likely to lead to unbalanced clusters.

Figure 3.13 shows the modularity of the (n,m)-generators using the Fixed Average-Degree distribution. Note that the values shown base on the clustering that is computed using the greedy algorithm postulated by Newman [New04]. We can see, that CHT- and Expansion-generated graphs exhibit similar modularity. Notable is the rather large span of values that Delaunay shows. In particular, with increasing degree the modularity is significantly lower than the others. We explain this by the rather regular structure of Delaunay- generated graphs, which leads to a lower number of edges within the clusters and, hence, a lower value of modularity. In contrast to Delaunay the Insertion-generated graphs exhibit a smaller span of modularity, which additionally has smaller values than

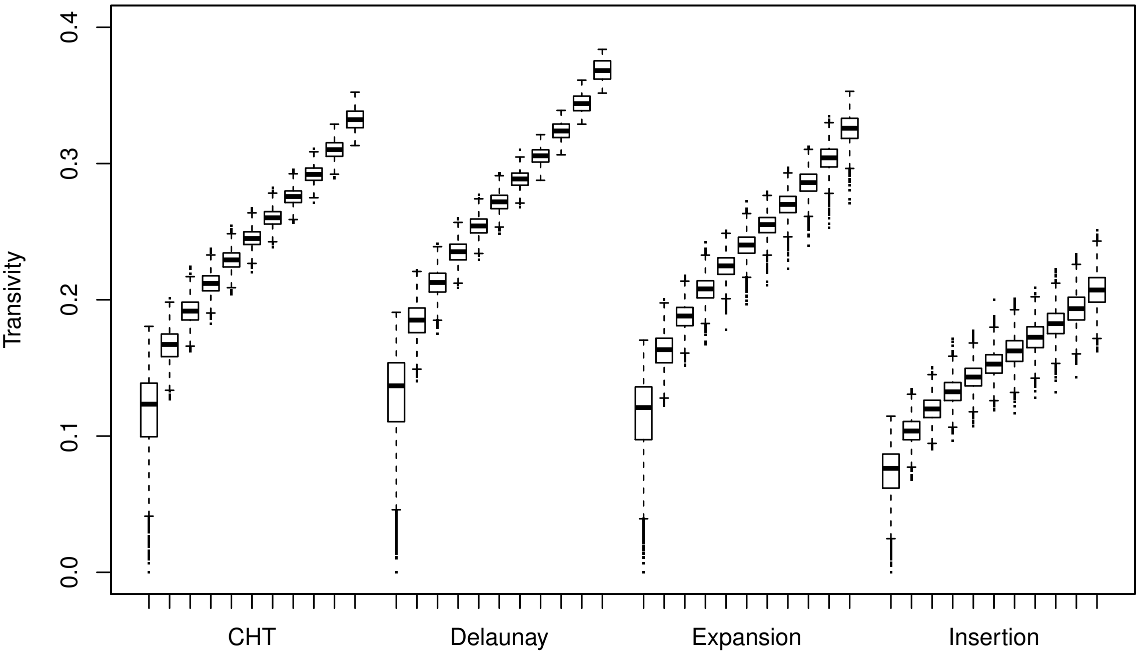


Figure 3.12: Transitivity of the (n,m)-generators using the Fixed Average-Degree distribution. The total number of generated graphs has been divided into eleven groups of increasing edge count.

CHT- or Expansion-generated graphs. Here, clusters contain considerably more edges than Delaunay. However, the graphs also contain several nodes of high degree, which influence the penalty term of modularity quite significantly. Note that we used modularity to reveal structural information instead of finding good clusterings.

Face-Size Distribution So far, we examined several measures typically used in network analysis. However, planar graphs can be characterized in terms of a possibly embedding in the plane that has to be crossing free. Therefore, we look at the structure of the faces that an embedding creates. In particular, we are interested in the distribution of the size of the created faces. The size of a face is the number of edges being part of a face. Note that we are aware of the fact that the face-size distribution depends on an embedding and a planar graph usually admits several embeddings. Nevertheless, we performed an analysis on the embeddings generated by the LEDA functions `make_planar_map` and `compute_faces`. In particular, we looked at graphs generated by the (n,m)-generators according to the Fusy distribution. We anticipate that the generators can be hardly distinguished, and, hence, we omit plots of our analysis but briefly describe the outcome.

The largest fraction of the faces of the generated graphs have a size within the interval $[3, 9]$, i. e., more than 99.5% of the faces. The peak is at size 3 and decreases fast with growing face size. The face-size sequence seems to be approximately equal among all generators as CHT, Delaunay, Expansion and Insertion can hardly be distinguished from each other. Only Fusy can be separated. It has a slightly smaller number of faces of size 3 and a slightly increased number of faces of size 4 and 5, compared to the (n,m)-generators. The only difference among the generators is their maximal face size. In particular, this is for CHT 55, for Delaunay 65, for Expansion 33, for Fusy 22 and for Insertion 32. However,

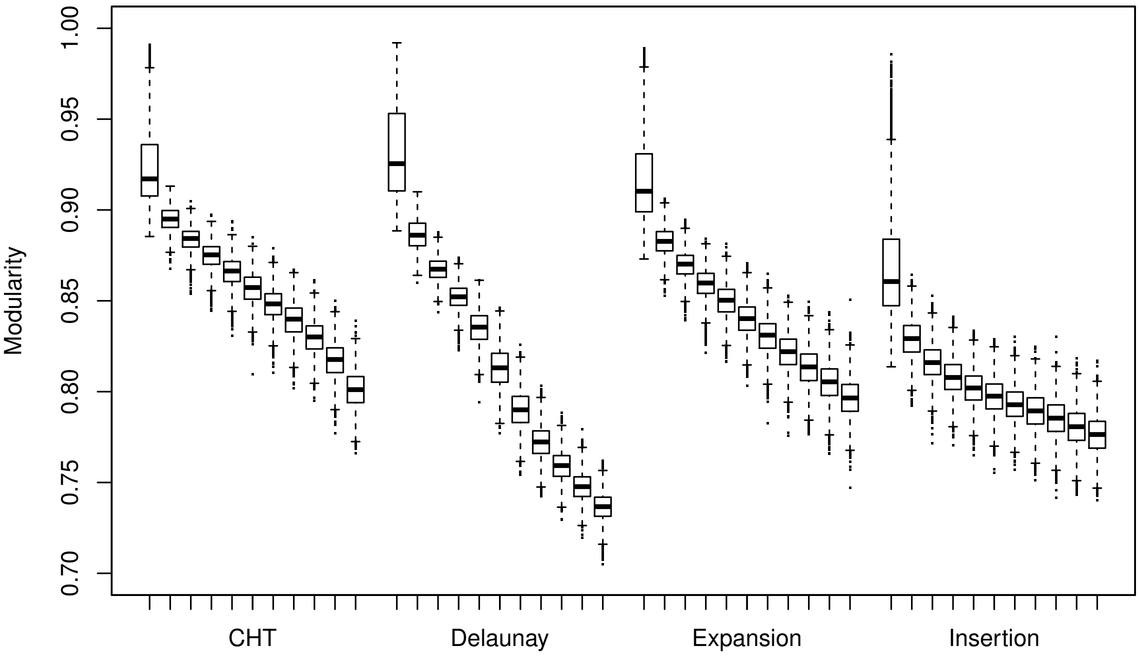


Figure 3.13: Modularity of the (n,m)-generators using the Fixed Average-Degree distribution. The total number of generated graphs has been divided into eleven groups of increasing edge count.

the number of faces being that large is not significant, which rules out any statement that a graph generated by a certain generator always contains a face of specific size.

Orthogonal Drawings Our attempt to analyze planar graphs in terms of their face-size distribution did not reveal significant differences. In a sense, an embedding of a planar graph is arbitrary if it does not admit certain properties, i. e., being 3-connected. Hence, we look at some characteristics of systematic graph drawings, in particular, *orthogonal drawings*. Planar graphs are favorable because relations can be drawn without crossings, which may distract viewers from grasping the big picture. In a similar direction aim orthogonal drawings, which are widely used in applications because they allow for a good readability. Recall that in an orthogonal drawing nodes are placed on integer coordinates and edges consist of a series of alternating horizontal and vertical segments, which are drawn on a grid. To allow nodes of degree larger than 4, we allow the nodes to occupy multiple adjacent integer coordinates, see Section 2.1.3 for details. In summary, the readability is greatly enhanced as a viewer can easily follow edges, especially if the algorithm tries to minimize the number of bends. Additionally, a viewer can get more easily an overview of the graph, if the drawing is compact, i. e., the area of the drawing is minimized.

Figure 3.14 shows the occupied area and the number of bends of (n,m)-generators using the Fixed Average-Degree distribution. Delaunay generated graphs catch our eye as the occupied area as well as the number of bends is significantly smaller than of the other generators. In contrast, CHT, Expansion and Insertion generated graphs show quite similar behavior. In a sense, Delaunay exhibits a certain *locality*, which for us consists of the following three properties. First, the structure of the graph is rather regular. Second, the maximum degree of the nodes is rather small compared to the others. Third, the

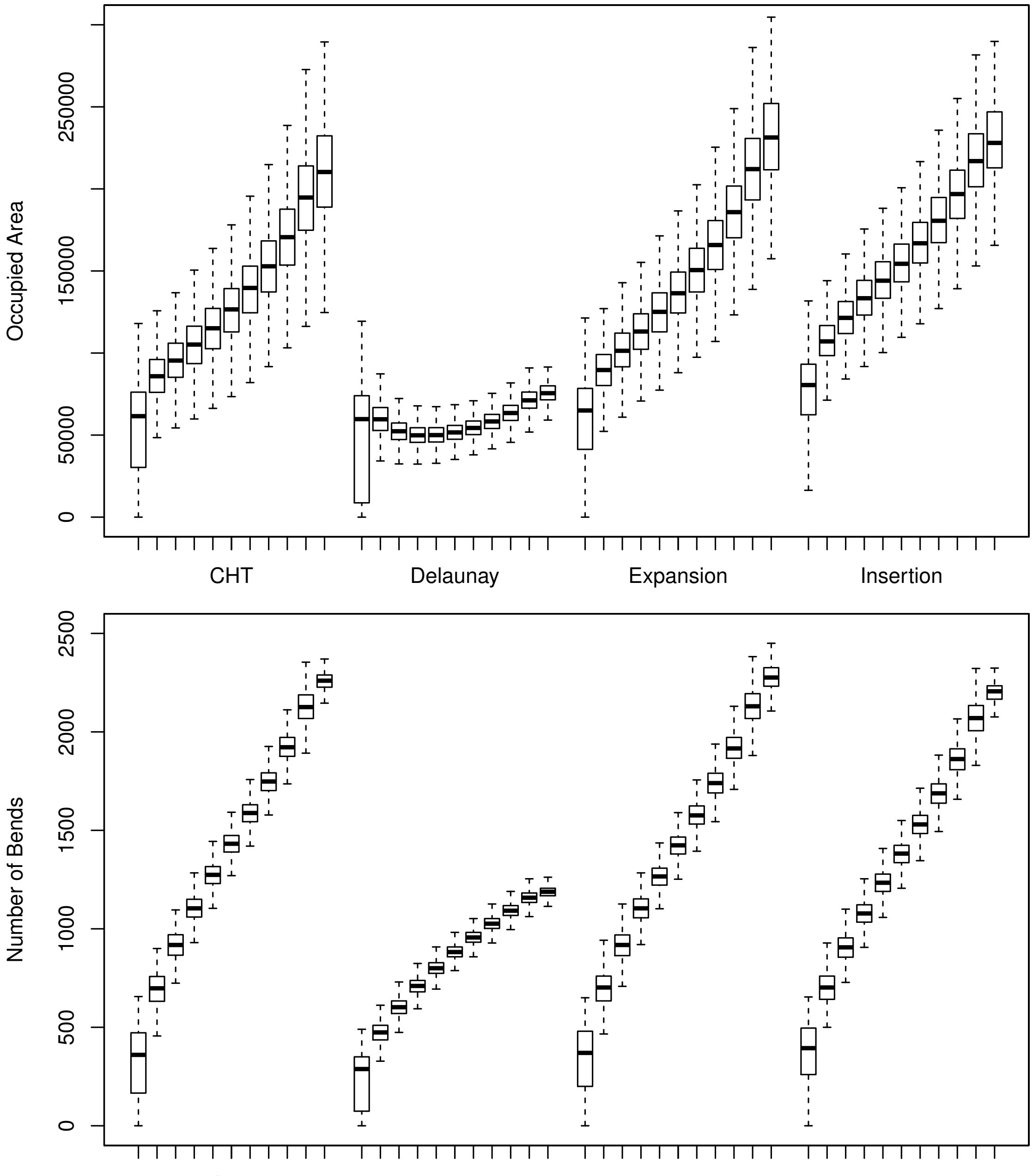


Figure 3.14: Occupied area (top) and number of bends (bottom) of orthogonal drawings of the (n,m)-generators using the Fixed Average-Degree distribution. The total number of generated graphs has been divided into eleven groups of increasing edge count, outliers have been removed.

diameter is quite large compared to the others. This locality allows the drawing algorithm to place the nodes in a more space efficient way, and at the same time, to avoid bends.

3.3.3 Algorithmical Behavior

So far, we performed a network analysis of the planar graph generators in terms of basic, local and global properties. Additionally, we analyzed the behavior of planar embeddings and orthogonal drawings. The outcomes of our analysis would already allow for a classification of the generators with respect to a complementary compilation. However, we now want to strengthen our proposition by a study of advanced algorithmical behavior, in particular by applying *fixed parameter tractable* algorithms [Nie06].

***K*-Core Decomposition** First, we study topological properties of the generated graphs. To this end, we use the *k-core decomposition* [Sei83]. This method destructively simplifies the graph. Iteratively, nodes are removed from the graph by increasing residual degree k. This procedure is applied until all nodes have been pruned off the graph. Note that in the case of adjusting the degree of a node u to a value smaller than k, u is also removed and has coreness k.

Recall the definition of Section 2.1.1. The *k-core* of a graph is a maximal subgraph in which each vertex has at least degree k. A node has *coreness k* if it is part of the k-core but not of the $k+1$-core. The k-core decomposition cannot be very large for planar graphs. A k-core implies the graph to have a *k-regular* subgraph. Because every planar graph contains a vertex with degree at most 5, no 6-regular subgraph can occur in a planar graph. Although planar 5-regular graphs exist [DKS09] their random generation seems to be unlikely as none of our generated graphs contains a 5-core.

In Figure 3.15 the coreness distribution of graphs generated by the (n)- and (n,m)-generators using the Fusy distribution are shown. The top figure shows the corresponding boxplots to visualize the coreness distribution and its spread. Note that outliers have been removed to give a better overview. The bottom figure shows the average values to allow for an easier comparison of the different generators. Evidently, connected graphs do not have nodes with coreness zero. The Intersection and Kuratowski graphs only consist of nodes with coreness one and two. Again, the group consisting of CHT, Expansion, Fusy and Markov can hardly be distinguished. Due to its generation process, Insertion contains at most a 3-core as by construction an Insertion-generated graph contains always a node of degree at most 3, and if this node is removed, we again obtain an Insertion graph by induction. Delaunay graphs are dominated by the amount of nodes with coreness 3. Thus, Delaunay and Insertion can be clearly separated from the other generators.

For completion we also report on the coreness entropy but omit to show the plots. Similar to the entropy plots of the degree distribution, Insertion as well as Delaunay can be clearly separated from each other and from the remaining graphs.

***K*-Vertex Cover** We now take a first glance at a *fixed parameter tractable* problem, see Section 2.2.2 for details on complexity. A problem is fixed parameter tractable if it admits an algorithm with running time $f(k) \cdot n^c$, where f is an arbitrary function depending only on a parameter k and c some constant. The NP-complete problem we examine is *k-vertex cover*.

A *vertex cover C* of a graph $G = (V, E)$ is a subset of its vertex set $C \subseteq V$ such

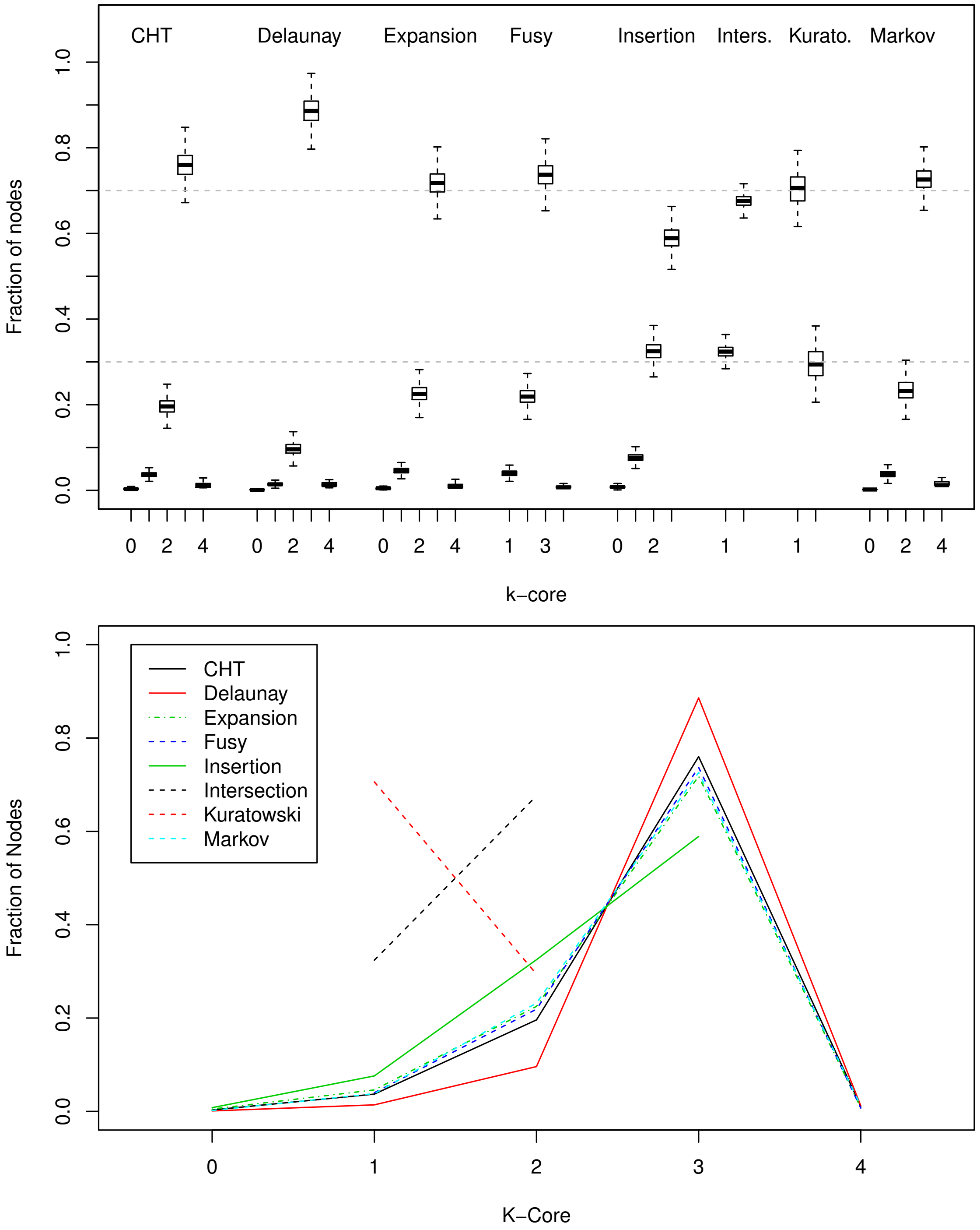


Figure 3.15: On the top, boxplots of the coreness distribution of (n)- and (n,m)-generators using the Fusy distribution, outliers have been removed for a better overview. On the bottom, lineplots of the average values of the above data set.

that each edge is incident to at least one vertex of C, see Section 2.2.3 for details. In the optimization variant of this problem, we search for a vertex cover of minimal size, which is known to be NP-hard. Though, we are looking into the decision variant *k-vertex cover* of this problem, i. e., we ask whether a vertex cover of size $|C| \leq k$ exists. The decision variant of this problem is known to be NP-complete [Kar72]. However, k-vertex cover is fixed parameter tractable with respect to the parameter k [AFN04].

To solve k-vertex cover a *kernelization* algorithm is applied in a preprocessing step, which reduces the initial instance to solve in polynomial time to its problem *kernel*, see Section 2.2.2 for details on kernelization. We now briefly explain kernelization rules that explain how to handle nodes of degree 0, 1 and 2; see the work by Abu-Khzam et al. [AKCF$^+$04] for details. Note that the rules are iteratively applied until the instance contains no more nodes of specific degree.

R0 Trivially, nodes of degree 0 do not contribute in terms of covered edges and can be removed.

R1 For nodes u of degree 1, either the node u or v of the incident edge $\{u, v\}$ needs to be in the vertex cover. Hence, we include the adjacent node v in the vertex cover, which allows for covering possible neighbors of v too. Then, both nodes u and v as well as the edges incident to u and v are removed from the graph.

R2a For nodes u of degree 2 and *adjacent* neighbors v, w, two nodes of the corresponding triangle need to be in the vertex cover. Instead of placing u in the vertex cover, we can include v, w and, thus, cover possible neighbors of v and w. Hence, we remove u, v, w and their incident edges from the graph.

R2b For nodes u of degree 2 and *non-adjacent* neighbors v, w, we cannot yet decide which nodes to include in an optimal vertex cover. Hence, we merge the nodes u, v and w into a single vertex u'. Additionally, we update the edges incident to v and w to the new vertex u'. Additionally, we update the edges incident to v and w to have the new vertex u' as endpoints. After solving the kernel optimally, u' is expanded, and then, processed with respect to the optimal solution found so far. This is only a brief description, and details are much more complicated; for details see [AKCF$^+$04].

The topological information we gained can be very useful. For instance, it can be used to estimate the size of the kernel when k-vertex cover kernelization is applied to graphs originating from a certain generator. Thus, the problem reduction equals the sum of the number of the nodes having coreness 0, 1 and partly 2. Recalling the outcomes of the coreness distribution in Figure 3.15, we see that Intersection and Kuratowski can be solved optimally in polynomial time. The best reduction of the remaining generators is achieved by Insertion. In contrast to that, Delaunay has the smallest reduction of all generators. The remaining generators behave very similarly. Figure 3.16 shows the (n,m)-generators using the Fixed Average-Degree distribution, and indeed, the generators exhibit the behavior estimated on base of their coreness distribution.

Planar *k*-Dominating Set So far, we used topological information, in particular the coreness distribution, to estimate algorithmical complexity, which we confirmed in terms of k-vertex cover kernelization. Now, we strengthen the algorithmical classification on the

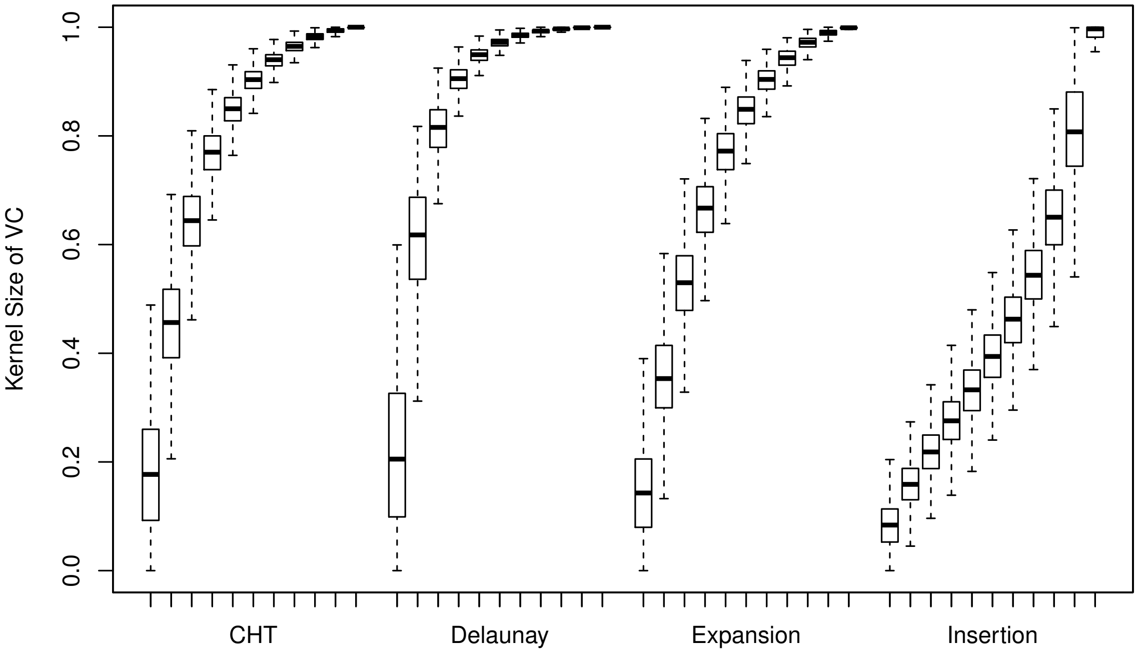


Figure 3.16: Size of the vertex cover kernel of (n,m)-generators using the Fixed Average-Degree distribution after applying preprocessing kernelization rules. The total number of generated graphs has been divided into eleven groups of increasing edge count, outliers have been removed.

generated graphs solving *planar k-dominating set*, which is also fixed parameter tractable for planar graphs [AFN04]. Recall Section 2.2.3, a *dominating set* is a set $D \subseteq V(G)$ of vertices such that each vertex in $V(G) \setminus D$ has a neighbor in D. The planar k-dominating set problem asks whether a dominating set of size $|D| \leq k$ exists in a planar graph.

Similar to k-vertex cover, first a problem kernel is computed. However, the problem reduction of planar dominating set does not rely on the degree of the nodes but the neighborhood of an examined node. The algorithms described in [AFN02] and [AFF$^+$05] have been implemented. The rules are rather complicated, and hence, we omit a description here. The reduction is based on two kernelization and seven search tree rules. Alber et al. report impressive kernelization results [ABN06] on several real world instances as well as artificially generated graphs. The authors were mainly interested in showing the algorithmical performance on real world problems, but the data set also contained planar graphs, which seem to not be selected in a representative way.

Figure 3.17 presents the size of the planar dominating set kernel. The large reduction reported was only partly achieved. CHT, Delaunay and Expansion show a large variance in the reduction, whereas Insertion can be reduced in most times by an amount of about 85%. Thus, the selection of the graph generator has a crucial impact on the results achieved by this algorithm and a wrong selection would greatly bias the results gained [ABN06].

Treewidth So far, the topological information and two fixed-parameter tractable algorithms confirm our classification. But we aim at establishing a broad base for a classification. We achieve this by studying the *treewidth* of a graph, which is the number of nodes that are mapped to the tree nodes of a tree decomposition of the graph. It is NP-hard to compute the treewidth of a graph [ACP87]. Nevertheless, if the treewidth of a graph class

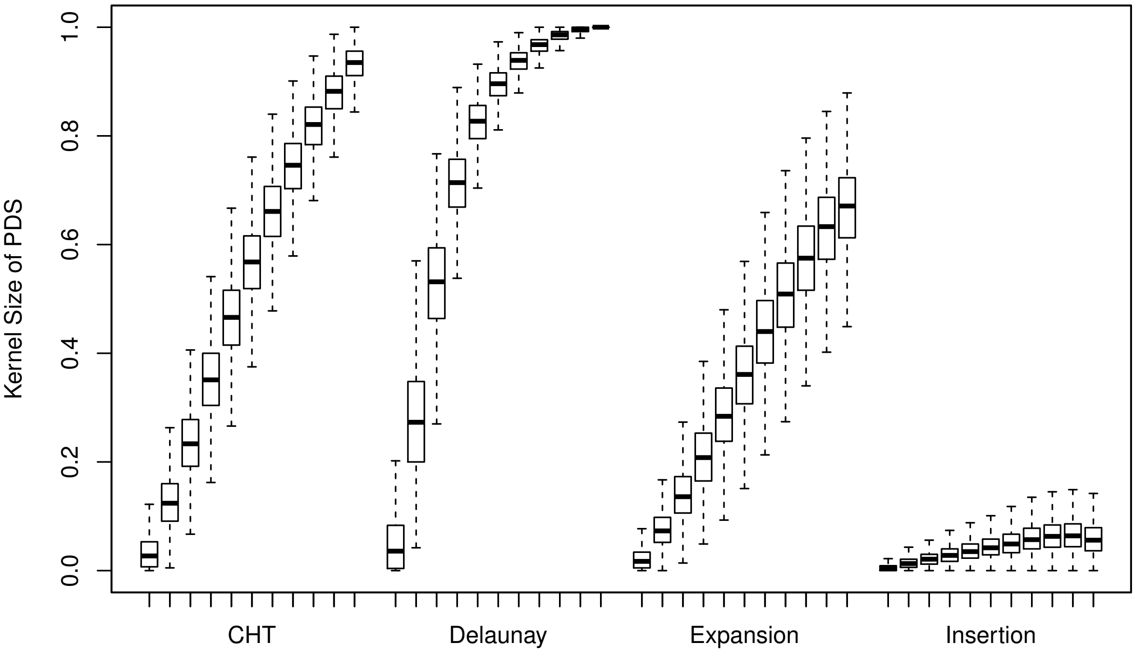


Figure 3.17: Size of the planar dominating set kernel of (n,m)-generators using the Fixed Average-Degree distribution. The total number of generated graphs has been divided into eleven groups of increasing edge count, outliers have been removed.

is bounded, several NP-hard combinatorial problems can be solved in polynomial time or even linear time [AP89].

A *k-tree* is defined recursively as follows. The complete graph K_k on k vertices is a k-tree. Given a k-tree G on $n \geq k$ vertices, a k-tree on $n+1$ vertices is obtained by adding a new vertex u and edges connecting u to every vertex of a K_k subgraph in G. A graph is a *partial k-tree* if it is a subgraph of some k-tree. Partial k-trees are exactly the graphs with treewidth up to k.

Theorem 7. The generator Insertion has a treewidth of at most 3.

Proof. The recursive definition of a k-tree is exactly the way the Insertion generator creates planar graphs. □

As we have seen, computing the exact treewidth is intractable. Hence, we employ heuristics provided by the library LibTW [vDvdHS06]. In general, we followed their recommendations although we performed a comparison of the provided algorithms on a sample subset of the (n,m)-generators on our own. It turned out that the running time of the provided exact algorithms was partly much too high. Hence, we solely selected a number of lower- and upper-bound heuristics, which we chose according to balance quality and running time. In particular, we use the minimum of the heuristics `GreedyDegree` and `GreedyFillIn` to find an upper bound on the treewidth [BK10]. To find a lower bound, we use the maximum of the heuristics `MinorMinWidth` as well as `AllStartMinorMinWidth` [GD04] and `MaximumMinimumDegreePlusLeastC` [BWK06].

Figure 3.18 shows the outcomes of the selected LibTW upper- and lower-bound heuristics, which were applied to graphs generated by the (n,m)-generators according to the

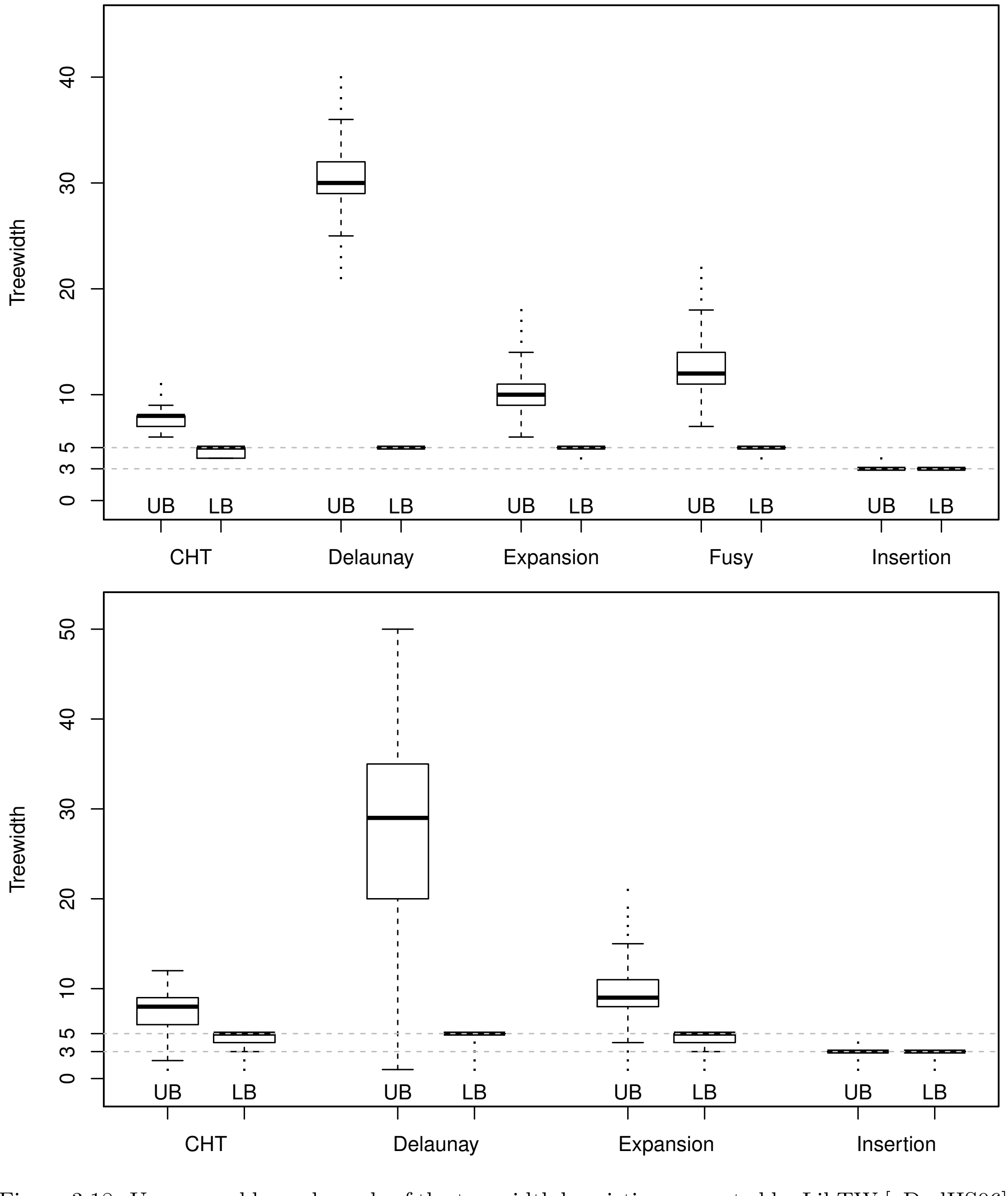


Figure 3.18: Upper- and lower bounds of the treewidth heuristics computed by LibTW [vDvdHS06] of the largest connected component of (n,m)-generated graphs according to the Fusy- (top) and the Fixed Average-Degree distribution (bottom). For each generator the left boxplot shows the upper bound (UB) and the right boxplot the lower bound (LB).

Fusy- and Fixed Average-Degree distributions. According to Theorem 7, Insertion generated graphs have treewidth at most 3 and are often not appropriate instances when dealing with NP-hard problems, which is confirmed by the LibTW heuristics. The treewidth of CHT is larger than Insertion but is rather small compared to the other generators. The treewidth of Expansion partially equals the treewidth of Fusy. Delaunay generated graphs seem to be very hard instances since the span of upper- and lower bound is largest.

3.4 Concluding Remarks

In the cycle of Algorithm Engineering, the experimental phase plays a crucial role in that it can show the practicality of newly developed or improved algorithms. To this end, the experiment has to be accurately described to make it repeatable, and hence, reliable. However, in many experimental works we can find ambiguous descriptions of experiments or inappropriately selected data sets. In this chapter, we improved this situation for experiments that involve evaluations of planar graphs. Our work is not limited to improving the situation within the experimental phase in the cycle of Algorithm Engineering but is also guided thereby.

To this end, we systematically analyzed known planar graph generators, particularly with regard to assembling a complementary compilation of planar graph generators that allow for a meaningful interpretation of experimental work. Hence, this study allows for such a classification of a selection of planar graph generators, namely four (n)-generators (Fusy, Markov, Intersection, Kuratowski) and four (n,m)-generators (CHT, Delaunay, Expansion, Insertion). The (n)-generators accept as parameter the number of nodes to create. Clearly, these generators admit an inherent distribution of edges. All (n)-generators are *complete*, i. e., they are capable of generating all possible planar graphs with positive probability. Additionally, Fusy and Markov are capable of generating planar graphs uniformly at random. However, with growing graph size n the expected average degree tends to a fixed value of 2.21, which by no means represents the possible interval $[0, (3n-6)/n]$. In contrast, the (n,m)-generators allow for an arbitrary average degree. Hence, we recommend a distribution of the average degree that spans this interval much better.

In the context of the theory phase, we showed that CHT is not complete, and additionally reason, why Delaunay and Insertion are also not complete. Hence, the only (n,m)-generator that is complete is Expansion. The Expansion generator is a good example how the experimental phase in the cycle of Algorithm Engineering can influence the viewpoint on theoretical aspects. From a first look, we expected Expansion to admit a linear running time. Although we did not perform large-scale running-time tests, the overall time to generate planar graphs using Expansion significantly exceeded those of the other fast generators. Being skeptical about our view on this generator, we returned to the theory phase to analyze the running time. As a matter of fact, our first view was partly wrong and, from an expectation point of view, partly right. In particular, we showed that Expansion has a running time in $\Theta(n^2)$ and an expected running time in $\Theta(n)$. Based on the observations of the running time of Expansion and guided by its theoretical analysis, we developed an improved version of the Expansion algorithm, from which we showed that its worst-case running time is in $O(n \log n)$.

In the implementation phase, we modified the stand-alone Fusy generator to accept our parameters. We implemented the remaining generators using LEDA. Markov and Expansion were developed using the graph framework of LEDA, whereas the remaining generators can be directly or indirectly computed using methods provided by the library. Unfortunately, the internal generators of LEDA are confusingly named, i. e., for identical or similar function calls totally different algorithms are used to generate graphs depending on the parameters passed. In an overview, we cleaned up this situation.

In the context of the experimental phase, we studied the eight selected planar graph generators by means of running time, network analysis with respect to graph properties, and algorithmic behavior, with a particular focus on fixed-parameter tractable kernelization algorithms. It turned out that Kuratowski and Markov are not efficient enough to run large scale tests. Although Intersection and Kuratowski are complete, we conclude from our experiments that it is highly unlikely they generate graphs uniformly at random. The efficient generator Fusy is capable of drawing graphs uniformly at random but cannot be used as an out-of-the-box generator. According to our network analysis, most of the graph generators can be classified into groups. Thus, Delaunay and Insertion can clearly be distinguished from each other and from the group consisting of CHT, Expansion and Fusy. The latter group shows small differences at various tests but none allows for a clear separation. This classification of the graph generators is strengthened by the analysis of the algorithmic behavior on generated instances with respect to both, basic algorithms and fixed-parameter tractable kernelization algorithms.

As a basic principle, experimental works should precisely describe the origin of the used data sets, which for generated graphs includes their distribution of the average degree. Experimenters should keep in mind which structural properties their algorithms exploit and ensure that the used data sets exhibits well distributed and representative structural properties to allow for significant empirical results. Because of the manifold properties of graphs that may be of interest, it is hard to present a general recommendation which of the studied generators to use. Nevertheless, theoreticians verifying practicability in a small experiment should rely on one of both uniform generators. The Markov generator is not practical for a generation of larger instances. However, on small instances it performs well enough, and its implementation is rather easy. Uniform generation comes with a restricted average degree of the graphs. If a small experiment should be conducted on a spread distribution of the average degree, we recommend Expansion. The naïve implementation is rather straightforward and has an expected running time of $\Theta(n)$. For detailed experimental works, experimenters should compile data sets that at least consist of Expansion-, Delaunay- and Insertion-generated graphs. Expansion overlaps CHT and Fusy to a certain extent. Nevertheless, both CHT and Fusy complement the data set.

Future Work Often in experimental algorithmics, graphs with a predefined number of nodes and edges are of interest, which Fusy cannot create due to its restrictions. Thus, future work might be a planar (n,m)-generator capable of generating labeled planar graphs uniformly at random. Expansion could be a good starting point as it is complete and compared to Fusy exhibits a similar behavior.

In this chapter, we dealt with labeled planar graphs. Of course a natural extension of these labeled approaches is to find similar algorithms for the unlabeled case. With the work by Fusy, the generation of labeled planar graphs uniformly at random is considered to be solved. Hence, the focus already changed from the generation of labeled planar graphs to the unlabeled case, which seems to be much more complicated, and thus, to to the best of our knowledge, is still open.

Part II

Artificial Data in Experiments for Route-Planning Techniques

Chapter 4

Artificial Road Networks

This chapter deals with the generation of realistic artificial instances of road networks. In doing so, we enlarge the set of available instances with respect to both, variety and magnitude. Typically, the performance of algorithms for static route planning is experimentally verified, and hence, an extended data set will certainly improve the reliability of such experiments. We consider a synthetic instance to be realistic, if it exhibits similar characteristics in terms of structural properties, algorithmic behavior and visual appearance. In the development of a graph generator that is capable of creating realistic instances, we are guided by the cycle of Algorithm Engineering. First, we review the only existing model with appendant generator, which has neither been implemented nor experimentally tested until now. This generator aims at creating networks that exhibit a small so-called 'highway dimension', which is a structural property conjectured to also be small for road networks. Indeed, thereby generated graphs exhibit properties that can be found in road networks. However, they also show atypical properties leading to a visual appearance that can hardly be recognized as a road network. Hence, we propose an alternative model, which incorporates structural properties found in road networks in a way that additionally supports the visual appearance to be more realistic. For this newly proposed model, we present a corresponding generator, which we discuss in detail, and report on a possible implementation. The implementations of both generators are evaluated and compared with each other and with real-world road networks. In this comparison, we found that real-world instances differ from each other to a certain extent with respect to both, structural properties and algorithmic behavior. Within this variance, both generators behave rather similar to real-world instances, however, the newly proposed generator gains advantages in most tested properties. In addition, our alternative generator is clearly superior compared to the existing generator with respect to a realistic visual appearance. In summary, this chapter provides an alternative model with a ready-to-use generator that is capable of generating realistic artificial instances of road networks.

4.1 Introduction

During the last two decades, advances in information processing led to the availability of large graphs that accurately represent the road networks of whole continents in full detail. Today, these networks are omnipresent in applications like route-planning software, geographical-information systems or logistics planning. While there is a vast amount of research on algorithms that work on — and often are tailored for — road networks, the natural structure of these networks is still not fully understood.

Aims In this work we aim to synthetically generate graphs that replicate real-world road networks. The motivation of doing so is manifold: First, the existing data is often commercial and availability for research is only restricted. In those situations, graph generators are a good and established way to obtain test data for research purposes. Additionally, it seems likely that data sets that represent the road network of the entire world will be available in a few years. It will be shown later that the size of the road network has a crucial, nontrivial and nonlinear influence on the performance of algorithms applied to it. Hence, using graph generators that are able to generate graphs of appropriate size and structure is essential when doing algorithmic research on such networks.

Second, we want to improve the understanding of the structure within road networks. This may support a theoretical analysis of algorithms that have been tailored for road networks. A well known example is the development of route-planning techniques during the last decade that yield impressive running times by exploiting a special 'hierarchy' in road networks [DSSW09a]. Many of these algorithms have recently been analyzed by Abraham et al. [AFGW10]. There, the intuition of hierarchy has been formalized using the notion of the so-called 'highway dimension' and a generator for road networks. In the work of Abraham et al., no evidence is given that this generator is a good model for road networks, which we evaluate in this chapter.

Furthermore, there is not only one 'road network'. Hence, we compare graphs originating from different sources with each other. This helps practitioners to assess the value of a given experimental work for their specific data and problem. Finally, as generators usually involve some tuning parameters, experimentalists can use them to generate interesting instances and steer the properties these instances have. This can help to better understand tailored algorithms.

Related Work There is a huge amount of work on point-to-point route-planning techniques. These are mostly tailored for road networks. An overview can be found in a work by Delling et al. [DSSW09a]. Abraham et al. propose a model and an idea of a graph generator, which has neither been implemented nor experimentally verified [AFGW10].

A work by Eppstein and Goodrich studies properties of real-world road networks [EG08]. Therein, road networks are characterized as a special class of geometric graphs. This characterization is evaluated on a real-world road network of the USA, namely the Tiger/Line road network provided by the administration. It could be a starting point for a possible graph generator, but too many degrees of freedom are left open to use it directly. Furthermore, road networks are analyzed concerning planarity: The typical number of crossings of the embedding given by the GPS-coordinates of an n-vertex road network is reported to be $\Theta(\sqrt{n})$.

Part of this chapter is joint work with Reinhard Bauer and Marcus Krug and has previously been published [BKMW10].

Contribution and Outline This is the first work that experimentally generates synthetic road networks and tests their practical applicability. In Section 4.2, we survey existing networks that are partly available to the scientific community. Then, we present two models with generators in Section 4.3, which aim at generating road-networks. In particular, we first introduce the model and the idea of a generator proposed by Abraham et al. [AFGW10]. Additionally, we describe our implementation and how we filled the degrees of freedom therein. As a last step for the generator by Abraham et al., we compare the visual appearance of generated networks to those of the real-world instances. From this analysis, we derive an alternative model, which aims at an improved modeling of structural and visual properties observed in real-world networks. For this newly proposed model, we also present a generator, report on its implementation, and show that hereby generated graphs exhibit a superior visual appearance compared to the former introduced generator. Finally, we briefly describe two standard generators, namely those for Unit-Disk and Grid-graphs, which are included in the experimental evaluation in order to compare the quality of the tailored road-network generators. In Section 4.4, we first report on the parameter sets used by our implementations to create our benchmark set. In addition, we analyzed the performance of both generators in terms of possible graph sizes, their memory consumption and time taken to create these instances. Afterwards, we compare the artificially generated instances with the real-world instances with respect to structural properties. In particular, we consider *connectedness*, *directedness*, *degree distribution*, *density* and *distance distribution*.

With respect to the properties, we observed that the tailored generators are good models for the real-world data, however, the generator of Abraham et al. [AFGW10] produces graphs that are too dense and incorporate nodes of too high degree. Having found that both generators mimic structural properties of real-world road networks quite well, we analyze their algorithmic behavior in Section 4.5. In particular, we focus on point-to-point shortest-path computations as this area uses techniques that have been highly tailored for road networks. We evaluate their behavior on a large scale, i. e., within the entire network, and on a small scale, where we sampled graphs of increasing size from the originating network. We conclude in Section 4.6. One unexpected outcome is that the real-world graphs significantly differ in their algorithmic behavior. Furthermore, both standard generators approximate road networks only to a limited extent. In summary, the proposed models with generators seem to be reasonably good approximations for the real-world instances, albeit the generator of Abraham et al. does generate instances that exhibit atypical structural properties and lack realistic visual appearance, which reduces its acceptability as an alternative to real-world road networks.

4.2 Real-World Road Networks

Graphs that represent road networks are typically constructed from digital maps. Digital mapping is expensive and mostly done by companies like Google, Teleatlas or NAVTEQ. Hence, up-to-date data is hard or expensive to obtain. We are aware of only three sources for real-world road networks that are (almost) free for scientific use.

PTV The data set PTV is commercial and property of the company PTV-AG[1]. It is based on data of the Company NavTeq and not fully free, but has been provided to participants of the 9th Dimacs Implementation Challenge [DGJ09]. We use slightly updated data from the year 2006.

[1] `http://www.ptv.de/`

Table 4.1: Overview of origin and size of the available real-world data.

data set	origin	represents	#nodes	#edges
TIGER	U.S administration	USA	24 412 259	58 596 814
PTV	commercial data	Europe	31 127 235	69 200 809
OSM-EU	collaborative project	Europe	48 767 450	99 755 206
OSM-USA	collaborative project	USA	158 774 110	326 893 667

TIGER The U.S. Census Bureau publishes the Tiger/Line data sets[2] TIGER. We use the version available at the 9th Dimacs Implementation Challenge[3].

OSM OpenStreetMap[4] is a collaborative project that uses GPS-data gathered by volunteers to assemble a digital map of the whole world. We use Europe-data of December 2009 and USA-data of January 2010, from which we removed all items that do not correspond to roads. In particular, we proceeded as follows.

To distill the OSM-data set, we used data available from a service offered by the company Geofabrik. Their download server[5] contains excerpts and derived data from the official OSM data set. Our distilled graph includes all elements for which the highway-tag equals one of the following values: `residential`, `motorway_link`, `trunk`, `trunk_link`, `primary`, `primary_link`, `secondary`, `secondary_link`, `tertiary`, `unclassified`, `road`, `residential`, `living_street`, `service` and `services`. We refer to the osm data set of Europe by OSM-EU and to the osm data set of the USA by OSM-USA.

In Figure 4.1, we present two example pictures of excerpts of continental real-world road networks.

4.3 Graph Generators

In this chapter, we develop graph generators capable of creating large graphs that capture the properties of real-world road networks. By the term large, we aim at graphs of continental size or larger. We call artificially generated road networks also *synthetic road networks* interchangeably. To allow for the generation of synthetic graphs of large size, we present two approaches. First, we introduce a model and an idea of a generator developed by Abraham et al. [AFGW10]. This generator had not been implemented before. Hence, we also present details on a possible efficient implementation. A drawback of the model of Abraham et al. is that it does not incorporate the Steiner property. This results in untypical structural properties compared to road networks, e. g., nodes having a high degree or the existence of many crossings in the network. Hence, we develop an alternative model whose improvements are based on observed shortcomings of the existing generator. Based on this model, we develop a novel generator, which employs Voronoi diagrams, and describe a possible efficient implementation. Additionally, we outline two

[2] `http://www.census.gov/geo/www/tiger/`

[3] `http://www.dis.uniroma1.it/~challenge9/`

[4] `http://www.openstreetmap.org/`

[5] `http://download.geofabrik.de/osm/`

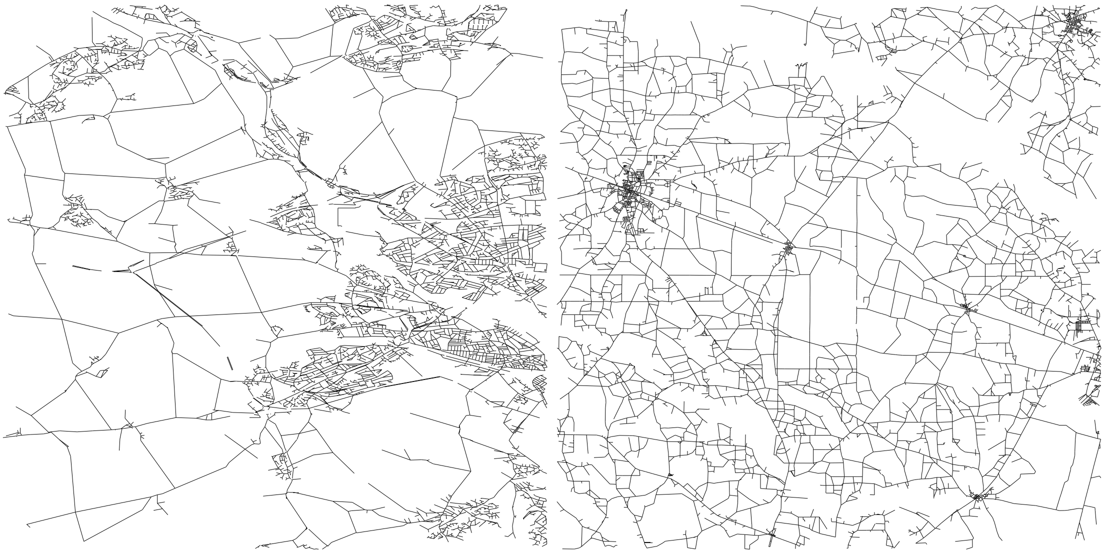


Figure 4.1: Both pictures show subgraphs extracted from real-world road networks. On the left, a subgraph of the data set PTV, and on the right, a subgraph of the data set TIGER.

standard generators, which are used to assess the quality of both road-network generators.

4.3.1 The Generator of Abraham et al.

Recently, Abraham et al. presented the framework of *highway dimension* which for the first time allowed to formally prove good query performance of several shortest-path techniques if the underlying network exhibits a small highway dimension [AFGW10]. The concept of highway dimension has its source in the observation of *transit nodes* in road networks, recall Section 2.5.3 on page 24. We do not directly work with the concepts of highway dimension, and hence, refer the interested reader to the work by Abraham et al. [AFGW10]. Note that for convenience we sometimes abbreviate the group's work and simply refer to the generator of Abraham or in short ABR.

In their work, Abraham et al. also presented a model with a generator, which is capable of generating networks that exhibit a small highway dimension. Hence, graphs generated by Abraham's generator are expected to exhibit properties similar to road networks. However, the generator had not been implemented so far. Hence, in the following, we first introduce the model and generator presented by Abraham et al. [AFGW10]. Then, we report on our implementation of this generator, and finally, summarize its pros and cons.

Model The model by Abraham et al. does not directly aim at generating realistic road networks but to generate graphs that admit a low highway dimension. However, this generator captures some properties we can find in road networks. The model behind this approach consists of the following three observations:

(1) Usually, roads are built incremental over time. Additionally, decisions are typically done locally, and not necessarily by a central planning instance. Hence, a *decentralized* and *on-line* process is considered to form a road network.

(2) Road networks exhibit a certain hierarchy within a road network, and additionally, there is a certain trade-off of costs between the levels of the hierarchy. On a faster road, we can travel faster but it is more expensive to build it. Hence, for a specific number of slow roads, a specific number of faster roads are built. This is propagated over all hierarchy levels in a road network. To capture this property, the underlying geometric space is required to have low doubling dimension. A metric has doubling dimension $\log \alpha$, if every ball of radius r can be covered by α balls of radius $r/2$.

(3) On roads that are in a higher level within the hierarchy, we are allowed to drive faster. Hence, the travel speed corresponds to the length of roads. To capture this property, the traversal time $\tau(u, v)$ of a road segment $\{u, v\}$ is set to $d(u, v)^{1-\delta}$ with a speedup parameter $0 < \delta < 1$.

In summary, the model bases on an underlying metric space (M, d) with diameter D. In this chapter, M will always be a rectangle in the Euclidean plane with $d(u, v)$ being the Euclidean distance between points u and v.

Random Point Source A generator based on the model of Abraham et al. also requires a random generator that distributes points in M. However, Abraham et al. do not specify such a distribution in their work [AFGW10]. Hence, we will now describe the distributions used later.

In our implementation we make two random point sources available, to which we refer by RPS in general. Both are implemented as online processes, which randomly generate points in an incremental fashion within M by rejecting outliers. Our routine relies on randomness, for which we use standard random distributions, recall Section 2.3 on page 16.

Firstly, we sample points uniformly at random, which we denote by RPSUNIF. However, typically vertices in road networks are not distributed uniformly at random as the network usually consists of sparse or dense regions. Hence, we do not expect a network generated using this point source to be similar to road networks with respect to its visual appearance.

Secondly, we developed an improved random point source that mimics city-like structures, denoted by RPSCITY. The improved point source bases on a 2-phase approach. In the preparation phase, we iteratively compute special points within M called *city centers*. This is done as follows. A new city center c is chosen uniformly at random within M. We assign a value r_c to c which is chosen from an exponential distribution $\mathcal{E}(\lambda)$ with parameter $\lambda = 1/(0.05 \cdot s)$ where s is the length of the longer border of M. We then assign a population p_c to c which is $r_c^{1.1}$. The preparation-phase stops, when the overall population exceeds the number of requested nodes n. In a preliminary experiment, the values given above turned out to be reasonable, and hence, are used in our later experiments without further tunings.

In the on-line phase, a new point x is generated on request as follows: Firstly, a center c with positive population p_c is chosen uniformly at random. We then reduce the population by one, i. e., $p_c = p_c - 1$. The location of x is determined in polar-coordinates with center c by choosing an angle uniformly at random and by choosing the distance from a normal distribution $\mathcal{N}(\mu, \sigma^2)$ with mean $\mu = 0$ and standard deviation $\sigma = r_c$. Whenever we sample points not lying in M these get rejected.

Algorithm The intuition behind the generator is similar to how road networks are built. Consider the following example. Typically, the road network of residential areas consists

Algorithm 4.1: Generator of Abraham et al. [AFGW10]

```
input : number of vertices $n$, random point source RPS, distance factor for
        neighborhood relation $k$
output: Graph $(V, E)$
initialize $V = C_0, \dots, C_{\log D}, E$ to be $\emptyset$ ;
for $t = 1$ to $n$ do
    $v_t \leftarrow$ RPS() ;
    $V \leftarrow V \cup \{v_t\}$ ;
    for $i = \log D$ to $1$ do
        if $d(v_t) > 2^i$ for each $w \in C_i$ then
            $C_i \leftarrow C_i \cup \{v_t\}$ ;
            for $w \in C_i$ do
                if $d(v_t, w) \leq k \cdot 2^i$ then $E \leftarrow E \cup \{v_t, w\}$
        $w \leftarrow$ closest point of $v_t$ in $C_{i+1}$ ;
        if $v_t \neq w$ then $E \leftarrow E \cup \{v_t, w\}$
set edge weights such that $\text{len}(u, v) = d(u, v)^{1-\delta}$ for $\delta = 1/8$
```

of sets of urban roads segments, i. e., segments are rather small and can only be traversed slowly. To reach nearby residential areas faster, these are interconnected by country roads that can be traveled faster. In a larger network it would be rather long-winded to only move on country roads if we want to drive to farther locations. Hence, on top of this network expressways are built that interconnect regions. This is done in a rather fair way such that travelers of each region can quickly reach the other regions. In this example, we built a network similar to the typical construction of road networks in a bottom-up manner. In particular, we maintained a hierarchy of road segments. We can say if enough road segments are within a specific area this area gets connected by a faster road, which corresponds to a higher level within a hierarchy. The generator of Abraham et al. has a similar intuition but works in a top-down fashion. To this end, the generator also maintains a hierarchy within the network to be generated but starts to place road segment in the highest level of hierarchy first. With growing size of the network, the generator descends within the hierarchy while it adds road segments.

More precisely, the generator starts with the empty graph $G = (V, E) = (\emptyset, \emptyset)$ and iteratively adds new vertices to V. Their location in M is distributed according to RPS. A 2^i-cover is a set C_i of vertices such that for $u, v \in C_i$, $d(u, v) \geq 2^i$ and such that for each $u \in V$ there is a $v \in C_i$ with $d(u, v) \leq 2^i$. During the process of adding vertices, the generator maintains for each i with $1 \leq i \leq \log D$, a 2^i-*cover* C_i: After a vertex v_t has been added, the lowest index i is computed such that there is a $w \in C_i$ with $d(v_t, w) \leq 2^i$. Then v_t is added to all C_j with $0 \leq j < i$. If no such i exists, v_t is added to all sets C_j. Then, given a tuning-parameter k, for each $C_i \ni v_t$ and each $w \in C_i$ an edge (w, v_t) is added if $d(w, v_t) \leq k \cdot 2^i$. Further, for each $C_i \ni v_t$ with $i < \log D$ such that $v_t \notin C_{i+1}$, an edge from v_t to its nearest neighbor in C_{i+1} is added.

Finally, edge lengths are fixed such that $\text{len}(u, v) = d(u, v)^{1-\delta}$ for $\delta = 1/8$. The pseudocode of the generator of Abraham et al. can be found in Algorithm 4.1.

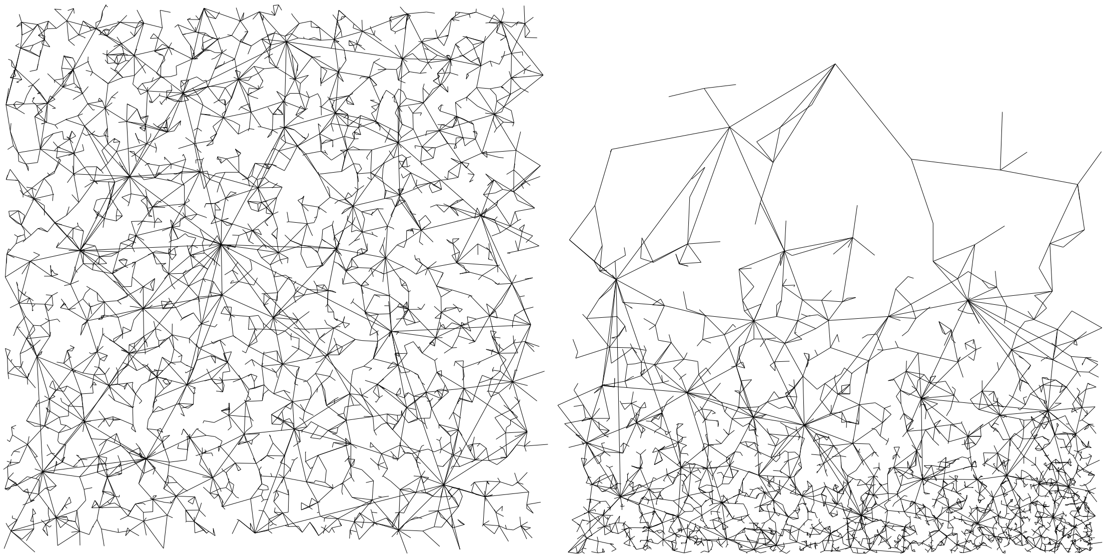


Figure 4.2: Both sides show graphs generated by the generator of Abraham. On the left side, the point source corresponds the uniform random point source rpsUnif, and on the right side, to the city random point source rpsCity, respectively. Both graphs were automatically extracted from large graphs, each consisting of 15 Million nodes.

Implementational Issues As already mentioned, Abraham et al. did not implement the generator. In the spirit of Algorithm Engineering, we report on our implementation, which includes data structures and tunings.

Above, we already reported on the random point sources rps. For drawing random numbers, we relied on the Boost library [boo].

The algorithm has to maintain covers C_i of size 2^i, with $1 \leq i \leq \log_2 D$. To maintain a cover, we need two fast queries: finding the nearest neighbor and finding the set of nearest neighbors within the range 2^i. Quadtrees allow for quickly answering both types of queries, recall Section 2.4.1 on page 17. Additionally, due to the cover property, we can assume that regions do not contain arbitrarily many points. Hence, we decided to realize and maintain each cover C_i by a single *quadtree* L_j, with $0 \leq j \leq \lceil \log_2 D \rceil$ and $i = j$. In maintaining these covers, we also maintain a hierarchy within the network. By naming a quadtree L_j we also refer to the *level* within that hierarchy. Hence, we decided to realize and maintain each cover C_i by a single *quadtree*. In maintaining these covers, we also maintain a hierarchy within the network. Hence, we name a quadtree L_j to recall the *level* within that hierarchy. In particular, we have covers C_i and quadtrees L_j, with $0 \leq j \leq \lceil \log_2 D \rceil$ and $i = j$. Note that level L_0 maintains no covers but contains all points of the network.

The running time of this algorithm is mainly influenced by the neighborhood search to maintain the covers. We need to maintain a constant number of quadtrees, which would result in a running time in $O((h+1) \cdot n)$. However, finding the neighbors in a covered area takes considerably more time but it turns out that the running time for generating these networks remains reasonable as we show in a scalability experiment later; see Table 4.3 on page 86 for details.

We let the generator accept the size of the metric space D as parameter, and additionally allowed for setting its aspect ratio. Another tuning parameter is k, which allows to steer the size of the area a node covers. In particular, a node in level L_i covers an area of size $k \cdot 2^i$. This deviates from the original description, where this area was fixed to $6 \cdot 2^i$.

Summary As already mentioned the model by Abraham et al. does not directly aim at generating realistic road networks [AFGW10]. Instead, the goal was to generate graphs that exhibit a low highway dimension. Nevertheless, we will now take a glance on its visual properties and defer the experimental analysis of the graph properties and algorithmic behavior to later sections.

In Figure 4.2, we present pictures of two randomly extracted subgraphs of large graphs generated by Abraham's generator using the random point source rpsUnif and rpsCity, respectively. Examining the distribution of the points, we can immediately observe the expected distributions of both random point sources. However, both artificial graphs exhibit properties that catch our eye, which are usually not observed in real-world road networks. In general, both graphs do not exhibit a typical visual appearance. However, due to the occurrence of sparse and dense regions, the graph generated using rpsCity looks slightly more realistic. In more detail, the structure of both graphs looks frayed and contains many crossings. Additionally, the graphs contain a couple of parallel or nearly parallel edges and nodes of large degree, which typically do not occur in road networks. We explain their occurrence with the hierarchy that is maintained by the cover property. Nodes that are on a high level within the hierarchy have neighbors on several lower hierarchy levels. This results in both, nodes of high degree, and long edges, which are then parallel to other edges and cause many crossings in the network. We think, this behavior could be avoided if the model of Abraham et al. would incorporate the *Steiner property*, which they point out was not their aim. If a model for generating road networks admits the Steiner property, a newly inserted vertex can connect to a point on an already existing edge by splitting the edge at this point and inserting a second vertex. This corresponds to creating an intersection on an existing road segment, which is typically done in real-world road networks, especially if it is cheaper to connect a new road to an already existing one instead of building a new farther road.

4.3.2 Voronoi-Based Generator

We have seen that Abraham et al. mainly focus on generating networks that have a small highway dimension, and that thereby generated graphs exhibit a visual appearance that can hardly be recognized as real-world road networks. However, our aim is to generate realistic artificial road networks, which also includes similar visual characteristics. To this end, we first present an alternative model that incorporates observations that are motivated by real-world instances and that allow to overcome the shortcomings of Abraham's generator. Based on this model, we then develop an appropriate generator and report on our implementation of it. Finally, we take a glance at the visual appearance of graphs generated by our new generator and compare it to both, graphs generated by Abraham et al. and real-world road networks.

Model Our model captures properties partly similar to Abraham's model. In particular, our model consists of the following assumptions about road networks, which are well motivated from real-world road networks:

(1) Road networks are typically built so as to interlink a given set of resources, such as, for instance, natural resources or industrial agglomerations of some sort.

(2) In the presence of two or more sites of resources it is best to build road segments along the bisectors of these sites. In doing so, any two sites incident to a road segment can access this part of a road at the same cost.

(3) To save costs, road networks are typically built taking the Steiner property into account. A newly built road is connected to the road network by the construction of an intersection at an existing nearby road.

(4) Usually, roads are built incremental over time where decisions are typically done locally. This originates from the fact that sites containing many resources will attract even more resources.

(5) Road networks exhibit a hierarchical, nested structure: The top level of this hierarchy is defined by the highways, which form a grid-like pattern. Each cell of this grid-like network is subdivided by a network of smaller roads, which again exhibits a grid-like structure. In this top-down manner, we ensure to fairly connect each region to the road network.

(6) Shortest paths in road networks do not typically exhibit a large dilation with respect to the Euclidean distance: Although dilation may be very large in the presence of long and thin obstacles, such as rivers and lakes, it is rather low for the larger part of the road network.

Our model differs from the model of Abraham et al. in some points. For example, we model the aggregation of resources directly, where for Abraham's generator we developed the random point source rpsCity, which mimics city like structures. Another point is the support of the Steiner property, from which we expect to lead to an improved visual appearance of the generated graphs.

Algorithm Our generator is a generic framework which works in two phases using the following two random distributions. First, by $\mathcal{E}_\lambda(x)$ we refer to the exponential distribution, and second, by $\mathcal{U}_{[a,b]}(x)$ we refer to the uniform distribution. For details on these distributions recall Section 2.3 on page 16. In the first phase, we generate a random graph based on recursively defined Voronoi diagrams in the Euclidean plane. The random process of creating points is not separated as in the generator by Abraham et al. but is incorporated in the algorithm. In the second phase, we then compute a sparser subgraph since the graph computed in the first phase is rather dense as compared to real-world networks.

A *Voronoi diagram of a set of points P* is a subdivision of the plane into convex Voronoi regions $\text{vreg}(p)$ for all $p \in P$. The Voronoi region $\text{vreg}(p)$ contains all points whose distance to p is smaller than their distance to any other point $q \in P \setminus \{p\}$. The union of all points which are equally far from at least two points form a plane graph, which we call the *Voronoi graph* $\mathcal{G}(P)$. The vertices of this graph are exactly the set of points which are equally far from at least three points in P. Each face f of this graph

Algorithm 4.2: Subdivide-Polygon

```
Input: polygon $P$, number of centers $n$, density distribution $\mathcal{D}$,
       radius distribution $\mathcal{R}$
$C \leftarrow \emptyset$;
for $i = 1$ to $n$ do
    $x \leftarrow$ choose uniform point inside $P$;
    $\alpha \leftarrow$ random value chosen according to $\mathcal{D}$;
    $r \leftarrow$ random value chosen according to $\mathcal{R}$;
    $m \leftarrow \lceil r^{\alpha} \rceil$;
    $C \leftarrow C \cup \{x\}$;
    for $j = 1$ to $m$ do
        $p \leftarrow$ choose random point in $R(x, r)$;
        $C \leftarrow C \cup \{p\}$;
    compute Voronoi diagram of $C$ in $P$;
```

Algorithm 4.3: Voronoi-Road-Network

```
Input: polygon $P$, number of levels $\ell$, fraction of smallest faces $\gamma_i$, center
       distribution $\mathcal{C}_i$, density distribution $\mathcal{D}_i$, radius distribution $\mathcal{R}_i$, $1 \leq i \leq \ell$
$\ell_0 \leftarrow 1$;
$S \leftarrow P$;
for $i = 1$ to $\ell$ do
    $m \leftarrow \gamma_{i-1}|S|$;
    $S \leftarrow$ smallest $m$ faces in $S$;
    $S' \leftarrow \emptyset$;
    for $f \in S$ do
        $n \leftarrow$ choose according to distribution $\mathcal{C}_i$;
        $S' \leftarrow S' \cup$ Subdivide-Polygon $(P(f), n, \mathcal{D}_i, \mathcal{R}_i)$;
    $S \leftarrow S'$;
```

corresponds to the Voronoi region of some point p. By $P(f)$ we denote the simple polygon which forms the boundary of f. The Voronoi diagram for a set of n points can be computed in $\mathcal{O}(n \log n)$ points by a simple sweep line algorithm [PS85].

Phase I The first phase of our generator is a recursive procedure whose core is a routine called Subdivide-Polygon($P, n, \mathcal{D}, \mathcal{R}$).

The pseudocode of Subdivide-Polygon is listed in Algorithm 4.2. Invoked on a simple polygon P this routine computes a Voronoi diagram inside P from a set of points which are chosen as follows: First, we choose a set of n uniformly distributed points in P, which we call *center sites*. For each of the center sites x we choose a density parameter α according to the distribution $\mathcal{D}$ as well as a radius r according to the distribution $\mathcal{R}$. Then we choose $\lceil r^{\alpha} \rceil$ points in the disc centered at x with radius r by choosing radial coordinates uniformly at random. Thus, we create a set of points as agglomerations around uniformly distributed centers.

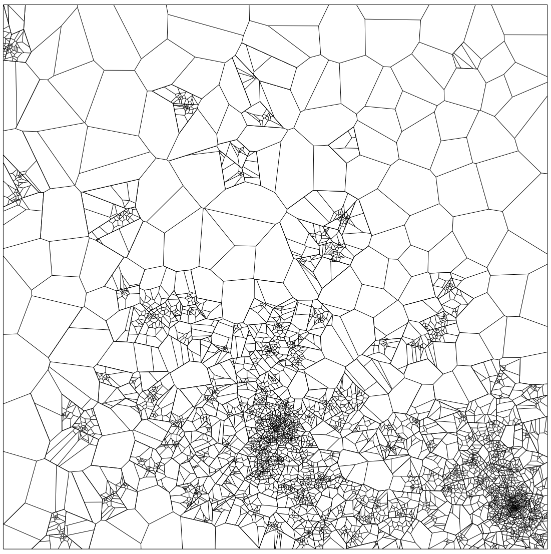


Figure 4.3: Graph generated by Phase I of the Voronoi graph generator. The graph exhibits already properties similar to road networks as it contains spare and dense regions. However, it is still too dense and has to be thinned out.

The recursive procedure that facilitates the first phase of our generator is called Voronoi-Road-Network. Its pseudocode is listed in Algorithm 4.3. The input consists of an initial polygon P, a number ℓ of levels for the recursion and for each recursion level $1 \leq i \leq \ell$ a fraction γ_i of cells to be subdivided along with distributions $\mathcal{C}_i$, $\mathcal{R}_i$ and $\mathcal{D}_i$.

We then proceed as follows: First, we choose a set of Voronoi regions to subdivide among the regions which were produced in the previous iteration of the algorithm. Let S be the set of Voronoi regions which were produced in the previous iteration, then we subdivide the $\gamma_i|S|$ smallest regions in the current iteration of the algorithm. Hence, the distribution of points will be concentrated in areas with many points. Therefore, we simulate the fact that sites with many resources will attract even more resources.

For each Voronoi region $f \in S$ that has been chosen to be subdivided, we first choose an associated number of centers n according to the distribution $\mathcal{C}_i$. Then, we call Subdivide-Polygon on input $P(f)$, n and the distributions $\mathcal{D}_i$ and $\mathcal{R}_i$ corresponding to the current level in the recursion.

The running time of Voronoi-Road-Network depends mainly on three operations. The two first are the sorting of the faces according to their size and the computation of Voronoi diagrams, which can be both done in $O(n \log n)$. The last operation is the random generation of points within a polygon. Due to the rejection sampling, this can take arbitrary running time, however, the running time remains reasonable as we show in a scalability experiment later; see Table 4.3 on page 86 for details.

Figure 4.3 shows a picture of a graph generated by Phase I of the Voronoi graph generator. We can observe that the graph resulting from the first phase is too dense, and hence, has to be thinned out.

Phase II In the second phase, we greedily compute a sparse graph spanner of the graph computed in Phase I using routine Greedy-t-Spanner. Given a graph G a *t-spanner* H

Algorithm 4.4: Greedy-t-Spanner

```
Input: Graph $G = (V, E)$, $t \in \mathbb{R}$
$H \leftarrow$ empty graph  for $\{u, v\} =: e \in E$ with non-increasing length do
    if $u, v$ are in different components or $\mathrm{dist}_H(u, v) > t \cdot \mathrm{len}(u, v)$ then
        $G \leftarrow G + e$;
```

of G with stretch t is a subgraph of G such that for each pair of vertices u, v in G we have $\mathrm{dist}_H(u, v) \leq t \cdot \mathrm{dist}_G(u, v)$. Usually, we would like H to contain as few edges as possible. However, determining whether a graph G contains a t-spanner with at most m edges is NP-hard [PS89].

In order to compute a sparse graph spanner greedily, we iterate over the edges sorted by non-increasing length, and add only those edges to the graph whose absence would imply a large dilation in the graph constructed so far. The pseudocode is listed in Algorithm 4.4. Note that the order in which we consider the edges differs from the greedy algorithm for graph spanners discussed, e. g., in the work by Bose et al. [BCF$^+$08]. Let H be the graph we obtained after considering m edges. Then, we insert the $(m + 1)$-st edge $\{u, v\}$ if and only if $\mathrm{dist}_H(u, v)$ is larger than $t \cdot \mathrm{len}(u, v)$. We assume $\mathrm{len}(u, v)$ to be ∞ if u and v are not in the same component. Hence, at each step H is a t-spanner for all pairs of vertices which are connected in H. At the end we will obtain a connected graph, and therefore, a t-spanner for G.

In order to determine $\mathrm{dist}_H(u, v)$, we use Dijkstra's algorithm for computing shortest paths, which can be stopped whenever we have searched the complete graph-theoretic ball of radius $t{\cdot}\mathrm{len}(u, v)$. Since we consider the edges in sorted order with non-increasing length, we will heuristically consider only few edges, as long edges are considered at the beginning, when the graph is still very sparse. Since the larger part of the edges in the graph is short compared to the diameter, the running time for this algorithm is reasonable as it takes about 1/5 of the total running time of the Voronoi graph generator; see scalability experiments in Table 4.3 on page 86. In order to speed up computation for the cases in which u and v are not connected, we use a *union-find* data structure to keep track of the components of H.

Implementational Issues Our generator relies on basic algorithms, and hence, can be easily implemented. The more difficult part is finding reasonable parameter sets. However, we now report on our implementation.

In the first phase, we compute Voronoi diagrams of a set of points using the CGAL library [cga]. We already described how points are distributed within a simple polygon P in routine Subdivide-Polygon. Since, points can possibly be placed outside P, we reject points not in P and generate additional points until the desired number of points has been created.

In the second phase, we rely on our own implementation of the algorithm of Dijkstra and a binary heap to greedily compute a t-spanner of the graph.

The running time of our sequential implementation is within reasonable time, i. e., a network of about 96 million nodes can be generated within less than 10 hours. However, for generating really large graphs this could possibly take too long and one might ask for a parallel version. The first phase of our algorithm can easily be parallelized and, hence, can be used to compute huge networks. In order to parallelize the second phase, however, we must give up on the idea of computing a spanner greedily, since the spanner property is global. This problem could be tackled by computing a local spanner.

Summary We presented an alternative model to generate realistic road networks. Similar to the model of Abraham et al., we incorporate a hierarchy within the network that allows for a fair distribution of fast roads. However, we do not explicitly model different traversal speeds for road segments in the hierarchy as this could be easily integrated in the recursive algorithm. Analogously to Abraham's model, we assume road networks are built incremental over time. However, we think this is a consequence of the fact that sites with many resources will attract even more resources, which have to be connected to the road network. For this reason, we do not utilize an on-line process for the growth of the network but incorporate the attraction of resources in our recursive approach. A major difference between both models is that in our model, we do not connect inserted points directly by roads but interpret them as resources. This allows for employing Voronoi diagrams which, together with the offline distribution of resources, obey the Steiner property. Hence, we expect graphs generated by our new algorithm to be more realistic. Again, we defer an experimental analysis to later sections.

Figure 4.4 shows pictures of artificial and real-world road networks. At the top, pictures of graphs generated by the Voronoi generator are shown. Both pictures show the same graph, on the left after Phase I, and on the right after Phase II, respectively. For a comparison of the visual appearance, the networks already described in Figure 4.1 and Figure 4.2 are shown. In the middle row, both real-world samples, and on the bottom row, both samples generated by Abraham. We can observe that the graph generated by Phase I of our newly proposed generator already looks much more realistic than graphs generated by Abraham's generator, but its overall density is still too high. However, after the graph is thinned out by the second phase the graph can hardly be distinguished from real-world instances on a coarse-grain scale. In particular, the graph does not contain nodes with high degree or crossings that attract attention. Additionally, similar to graphs generated by Abraham et al., the Voronoi generated graph contains edges of length varying from very short to very long without violating the overall picture. In summary, graphs generated by our newly proposed generator exhibit a visual characteristic that can be rated realistic.

4.3.3 Standard Generators

In order to allow for conclusions on the quality of our tailored generators, we compare them to the following two standard graph generators:

Grid Graphs These graphs are based on two-dimensional square grids. Each crossing in the grid corresponds to a node in the graph. Between each two neighbors on the grid an edge is inserted. We set the weights of the edges to randomly chosen integer values between 1 and 1000 [BDD+10].

Unit-Disk Graphs Given a number of nodes n, a unit-disk graph is generated by randomly assigning each of the n nodes to a point in the unit square of the Euclidean plane. There is an edge $\{u, v\}$ in case the Euclidean distance between u and v is below a given radius r. We adjusted r such that the resulting graph has approximately $7n$ edges. These

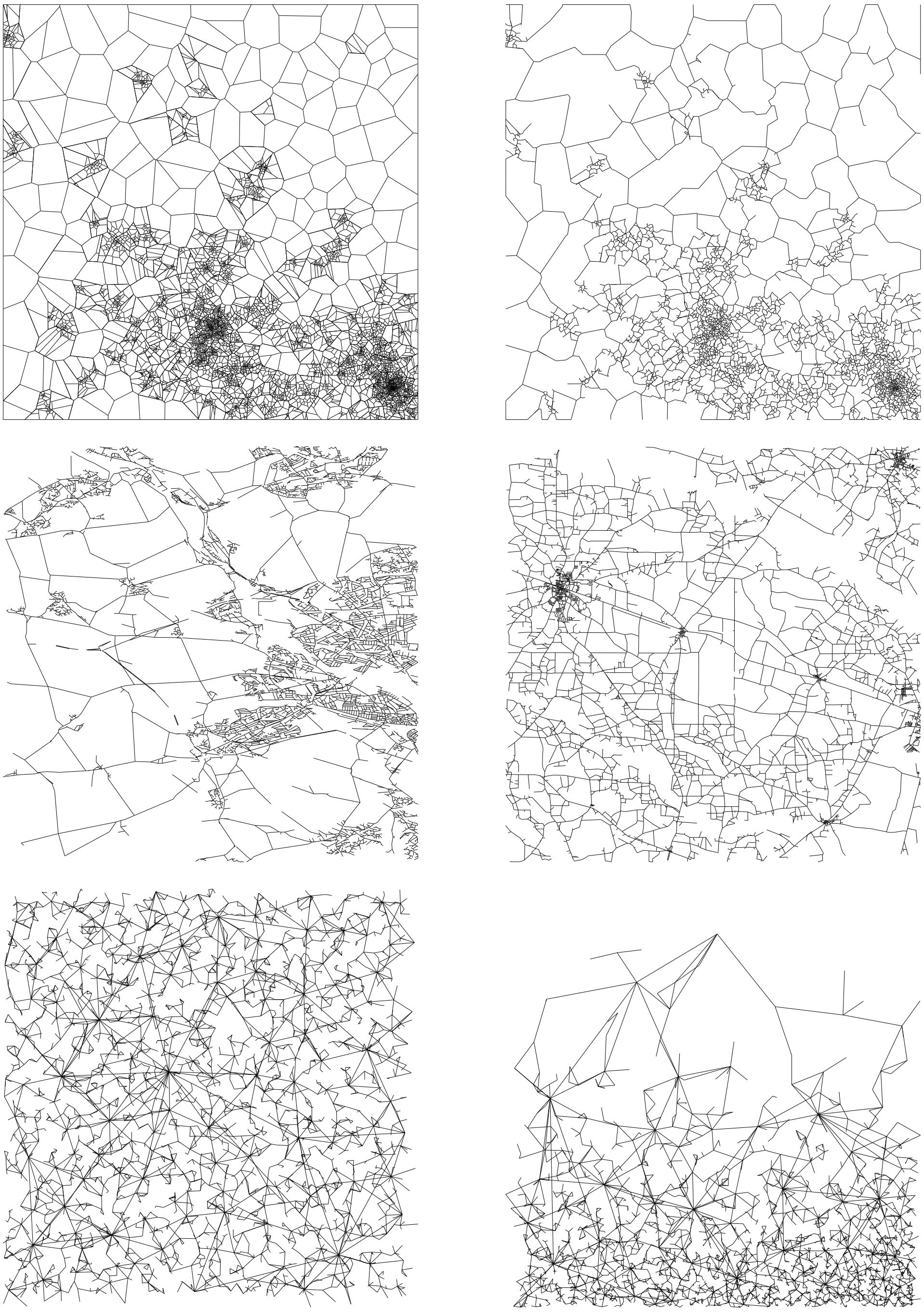


Figure 4.4: Shown are different real-world and artificial road networks. On the top row, graphs generated by the Voronoi graph generator. On the left, a graph after Phase I, and on the right, identical graph after Phase II. Note that the bounding box is part of the input polygon, and hence is thinned out. In the middle row, two subgraphs extracted from real-world road networks. On the left, a subgraph of the data set PTV, and on the right, a subgraph of the data set TIGER. On the bottom row, subgraphs extracted from graphs generated by the generator of Abraham. The used point source corresponds on the left to rpsUnif and on the right to rpsCity.

values were selected to balance connectedness and the number of edges, which should be close to road networks. Often, for a number of edges below $7n$ the graph gets disconnected. We use the Euclidean distances as edge weights.

4.4 Data Generation and Graph Properties

Our goal is to artificially generate realistic road networks. Therefore, we presented two approaches that are capable of doing so. Yet, we have not verified whether the artificial instances exhibit similarities beyond their visual appearance. Hence, in this section we experimentally assess the quality of the proposed generators with respect to structural graph properties.

To this end, we first report on the generated data sets used in our comparison. Afterwards, we assess the scalability of both generators in terms of size, running time and memory consumption. Finally, we compare artificially generated instances to real-work road networks by means of basic graph properties.

4.4.1 Data Generation

This work incorporates the first implementation of Abraham's generator and of our newly proposed generator. We will now report on the parameter sets used to create the benchmark set, which will be used later in our experimental evaluation, and thus, these will be named alongside with the presentation of the parameters. In addition, we perform a scalability test on both generators where we measure their performance with respect to both, memory consumption and running time. We omit the detailed parameter sets of the the standard generators as these are already described in Section 4.3. To assess the quality of the proposed generators, we generated the following three instances:

Abraham et al. In a preliminary test, we evaluated several parameter combinations to find reasonable parameters, which are used to generate the following two graphs by Abraham's generator. We choose the number of levels to be 25 and therefore set $D = 2^{25}$. The aspect ratio of the rectangle representing M is 0.75. Finally, we set the covered area modifier to $k = \sqrt{2}$. For our benchmark set, we generated two graphs using identical parameters sets but utilized as random point source either RPSUNIF or RPSCITY. The first network was created using the uniform random point source, and hence, we refer to this graph by ABR-UNI. The graph ABR-UNI consists of 15 million nodes and 43.5 million edges. The second network was generated using the random point source RPSCITY. We refer to this graph by ABR-CITY, which also consists of 15 million nodes and 43.5 million edges. For the scalability test, we use the above parameters but adapt the number of nodes the graph should contain.

Voronoi-Based generator By VOR, we denote the graph generated by the Voronoi generator, which is part of our benchmark set. In particular, the graph VOR is a 4-level graph computed using the parameters listed in Table 4.2. In the second phase, we greedily computed a 4-spanner subgraph. The final graph contains about 42 million nodes and 93 million edges. The number of nodes in a graph generated by the Voronoi generator strongly depends on the parameters used. For the scalability test, we also generated 4-level

Table 4.2: Parameters of the Voronoi road network.

Level i	# of centers $\mathcal{C}_i$	density $\mathcal{D}_i$	radius $\mathcal{R}_i$	fraction γ_i
1	1 700	.2	$\mathcal{E}_{.01}$	.95
2	$\mathcal{U}_{[2,40]}$	.5	$\mathcal{E}_{.1}$	.9
3	$\mathcal{U}_{[2,70]}$	.9	$\mathcal{E}_{2}$	.7
4	$\mathcal{U}_{[4,40]}$	.0	0	0

graphs and computed a 4-spanner subgraph of each, but slightly varied the parameters shown in Table 4.2. We will not report on the details as we do not use these graphs in our experimental analysis.

Performance In addition to the presented data sets used to assess the quality of the generators, we generated instances of increasing size to also assess the scalability of the generators. To this end, we report on our implementation and the measurements of the number of nodes and edges in the graph as well as memory consumption and running time.

The generator by Abraham et al. is written in C++ using the STL, and for drawing random numbers the Boost library [boo]. The Voronoi generator was also written in C++ using the STL, and additionally using the CGAL library [cga]. The code of both generators was compiled with GCC 4.5 using optimization level 3. The scalability experiments were run on a machine that consists of four AMD Opterons 6172 CPUs having twelve cores each. The cores are clocked at 2.1 GHz. Each of the CPUs has an associated *local* memory bank that it can access fast. In addition, a CPU can access the memory banks of the other CPUs with a time penalty, which originates from crossbar communication. In total, the machine has 256 GB of memory at its disposal, which is 64 GB of local memory per CPU. We will refer to this computer by *Machine I*. The generators were run using OpenSuse 11.3 on a single core of the machine. We did not ensure the implementations to primarily use the local memory of the core it runs on, and thus, allocated memory might be split over the memory banks. Hence, certain fluctuations in running time occur, which in our opinion does not affect the general outcomes of our scalability experiments for two reasons. First, we deal with long-run computations. Second, for huge instances the local memory of a CPU would not be sufficiently large, and hence, it is distributed over all memory banks anyway.

In Table 4.3, we present the outcomes of our scalability experiment. In general, we can observe that both generators perform reasonably well. When comparing the Voronoi generator and Abraham's generator, we observe that the Voronoi generator consumes less memory. This is due to the fact that vertices are possibly contained within several covers in the hierarchy of Abraham's generator. Comparing both types of Abraham's graphs, we can see that ABR-CITY consumes less memory than ABR-UNI, which originates from the agglomerate point source of ABR-CITY. The reason for this is that newly inserted points tend to be covered already, and hence, are contained within a smaller number of covers. Note that the size of the output graphs generated by the Voronoi generator depends on the chosen parameters, and hence, can only roughly be estimated. To generate the shown graphs, we had to adapt the parameters of Table 4.2, which have a certain influence on the running time of the Voronoi generator as can be seen in the fluctuations of the running

Table 4.3: Outcomes of the scalability experiments on artificially generated graphs. Three graph types of increasing size were generated using the Voronoi generator and Abraham's generator using either RPSUNIF or RPSCITY. Besides the number of nodes and edges of the synthetic networks, the memory consumption and the total running time used during their generation are shown.

algorithm	nodes	edges	memory	time
VOR	931 714	2 330 488	451 MB	30 s
VOR	6 295 922	13 859 308	2 594 MB	806 s
VOR	20 960 761	46 471 514	9 420 MB	3 024 s
VOR	43 265 545	95 920 306	17 752 MB	8 578 s
VOR	96 877 872	214 773 074	39 748 MB	33 080 s
VOR	369 666 928	825 470 874	151 646 MB	132 646 s
ABR-UNI	1 000 000	2 907 490	764 MB	107 s
ABR-UNI	5 000 000	14 529 014	3 439 MB	655 s
ABR-UNI	15 000 000	43 582 908	10 690 MB	1 596 s
ABR-UNI	50 000 000	145 253 186	36 952 MB	6 337 s
ABR-UNI	100 000 000	293 351 468	52 199 MB	10 970 s
ABR-UNI	200 000 000	583 934 624	137 606 MB	21 914 s
ABR-CITY	1 000 000	2 909 828	569 MB	86 s
ABR-CITY	5 000 000	14 529 458	3 107 MB	508 s
ABR-CITY	15 000 000	43 579 060	8 894 MB	1 453 s
ABR-CITY	50 000 000	145 231 954	30 507 MB	5 182 s
ABR-CITY	100 000 000	293 390 344	58 825 MB	10 638 s
ABR-CITY	200 000 000	583 860 202	116 551 MB	21 271 s

time. The time it takes to compute the t-spanner of the graph computed in Phase I is about 1/5 of the total running time. In summary, besides a slight overhead, the generators exhibit a good scalability with respect to both, memory consumption and running time.

4.4.2 Graph Properties

In this section, we analyze structural network properties of the artificially generated graphs and the real-world road networks. By comparing them with each other, we assess whether similarities between the networks can be found. We compare four instances of two real-world road networks of continental size, which originate from three different sources. In doing so, we can analyze whether the modeling process of the real-world road networks influence graph properties or the behavior of algorithms applied. In particular, we establish the following comparison chain: TIGER$\longleftrightarrow$ OSM-USA$\longleftrightarrow$ OSM-EU$\longleftrightarrow$ PTV. Later in the algorithmic analysis we will not use the data set OSM-USA but the data set OSM-S-USA, which is a geometrical square-sized subgraph sampled from the network OSM-USA. Hence, we included this graph in our analysis of the structural network properties.

In Table 4.4, we present the origin, represented area and basic structural properties of the networks. We always count the number of edges as directed, i. e., edges (u, v) and (v, u) both contribute to the overall number of edges. We first observe that all real-world graphs have an almost equal density of 2.05 to 2.4, which is strongly related to the highly detailed modeling of bends. The density of VOR is similar to the real-world networks. However,

Table 4.4: Overview of origin, represented area, size, density and fraction of directed edges of the available real-world data sets and the artificially generated data sets.

data set	origin	represents	#nodes	#edges	density	% directed
TIGER	U.S administration	USA	24 412 259	58 596 814	2.40	.0%
PTV	commercial data	Europe	31 127 235	69 200 809	2.22	4.9%
OSM-EU	collaborative	Europe	48 767 450	99 755, 206	2.05	3.5%
OSM-USA	collaborative	USA	158 774 110	326 893 667	2.07	6.6%
OSM-S-USA	collaborative	USA	24 268 971	50 131 139	2.06	7.3%
ABR-CITY	Abraham et al.	synthetic	15 000 000	43 573 536	2.90	.0%
ABR-UNI	Abraham et al.	synthetic	15 000 000	43 585 606	2.90	.0%
VOR	Voronoi generator	synthetic	42 183 476	93 242 474	2.21	.0%

Table 4.5: Relative sizes of the k biggest strongly connected components (SCC). Their size is measured in number of nodes of the network.

	k					
data set	1	2	5	20	100	total # of SCCs
OSM-EU	80%	91%	94%	96%	97%	541 264
OSM-USA	89%	89%	90%	91%	92%	10 578 020
PTV	97%	97%	97%	97%	97%	924 561
TIGER	98%	98%	99%	99%	99%	89 796

the density of both graphs generated by Abraham's generator is slightly too high, which cannot be adjusted to a lower value using the available parameters but increased by the parameter k. Furthermore, we observe that all real-world graphs are undirected or almost undirected. Typically, a certain number or edges within road networks are directed, which originates from the modeling of one-way roads and road segments of different length. The data set TIGER is undirected although the data set OSM-USA is directed. Hence, we conclude that the TIGER data might not be perfectly realistic. By construction the synthetic graphs are undirected.

Strongly Connected Components In Table 4.5, we give an overview of the *strongly connected components* of the real-world instances. They consist of one huge and many tiny strongly connected components. We can observe that the OSM-data deviates in the size of the biggest strongly connected component. An explanation for this is the unfinished state of the underlying map and we expect the deviation to decrease with increasing level-of detail in the OSM-data in the future. Our extraction routine returns a single strongly connected component, and hence, we do not report on the components of the data set OSM-S-USA. By construction, the synthetic graphs are strongly connected.

Degree Distribution In Table 4.6, we present the degree distributions of the road-network instances. We can observe, that the real-world instances have in common that almost all nodes have a degree of 4 or less, and only a marginal number of nodes is of degree 5 or larger. We observe that the distribution of the node-degrees is very similar for PTV and TIGER but deviates significantly for OSM-EU or OSM-USA whereas the

Table 4.6: Degree distribution of the data sets: relative frequency according to degree.

	degree						
data set	0	1	2	3	4	5	≥6
OSM-EU	.001	.097	.773	.114	.015	0	0
OSM-USA	.058	.059	.788	.078	.016	0	0
OSM-S-USA	0	.051	.853	.076	.019	0	0
PTV	.030	.247	.245	.428	.049	.001	0
TIGER	0	.205	.301	.386	.106	.001	0
ABR-UNI	0	.323	.254	.160	.094	.056	.113
ABR-CITY	0	.324	.254	.160	.093	.056	.113
VOR	0	.100	.601	.292	.005	.002	0

latter two are quite similar to each other. Obviously, the modeling process has a major influence on the degree distribution. We conclude that the instances originating from the OSM-project model bends in a much higher detail than both, TIGER and PTV. A reason might be that PTV as well as TIGER are used in products, where the size of the data is important, and thus, the data is processed to balance quality and size. The artificial instances are strongly connected by construction, and hence, they do not contain nodes of degree 0. On the one hand, we can observe that the distribution of VOR is quite similar to OSM-EU and rather similar to OSM-USA. On the other hand, we see that the degree distribution of both instances generated by Abraham's generator significantly deviate from the others as both instances contain considerably more nodes of large degree. In particular, about 16.9% of the nodes have a degree of 5 or larger. In addition, the maximum degree is 47 and 58 for ABR-UNI and ABR-CITY, respectively. The choice of the random point source used in Abraham's generator seems not to affect its degree distribution at all. Most notable, for the first time, we observe a considerable spread of a property of the real world instances, and we see that VOR fits best in between.

Degree Distribution Variance In Figure 4.5, we give an overview of the variance of degree distributions with the road-network instances. In order to show the variance within the graphs, we sampled 100 geometrically square-sized subgraphs, each containing 100 000 nodes. The box-plots show the results for these samples, and the squares represent the numbers of the original instances. On these samples, we observe average degree distributions similar to those of the large instances. However, the real-world samples exhibit a quite large variance whereas the artificial samples show only slight variance of degree distributions. We can now see, that the random point source influences at least the distributions of points in that ABR-CITY exhibits a larger variance of the degree distribution than ABR-UNI. Again, the data generated by the Voronoi generator exhibits the best behavior in that it exhibits the largest variance of degree distribution among the artificial data sets.

Distance Distribution In Figure 4.6, we show the distribution of the distances between pairs of nodes in the networks. The distributions are hard to interpret but can be used as a fuzzy fingerprint for the structure within the graphs, e. g., to separate them from different graph classes like small-world graphs. We observe that the data set OSM-USA corresponds the data set TIGER, and that the data set OSM-EU is similar to the PTV-

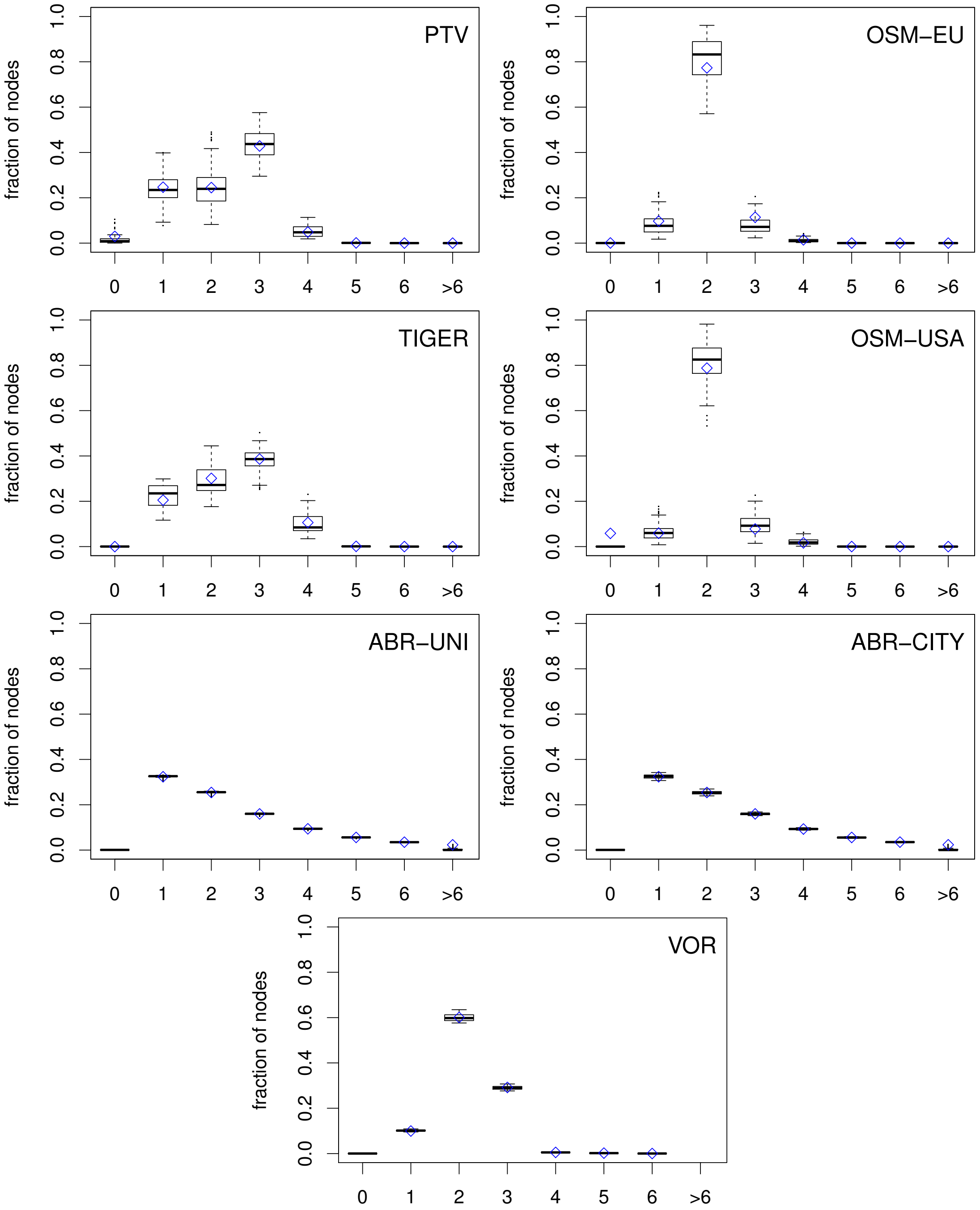


Figure 4.5: Degree distribution and its variance of the given road networks. In order to show the variance within the networks, we sampled 100 geometrically square-sized subgraphs of size 100 000, and show their degree distribution. The blue squares give the numbers for the original networks. Note that the x-axis refers to the node degree.

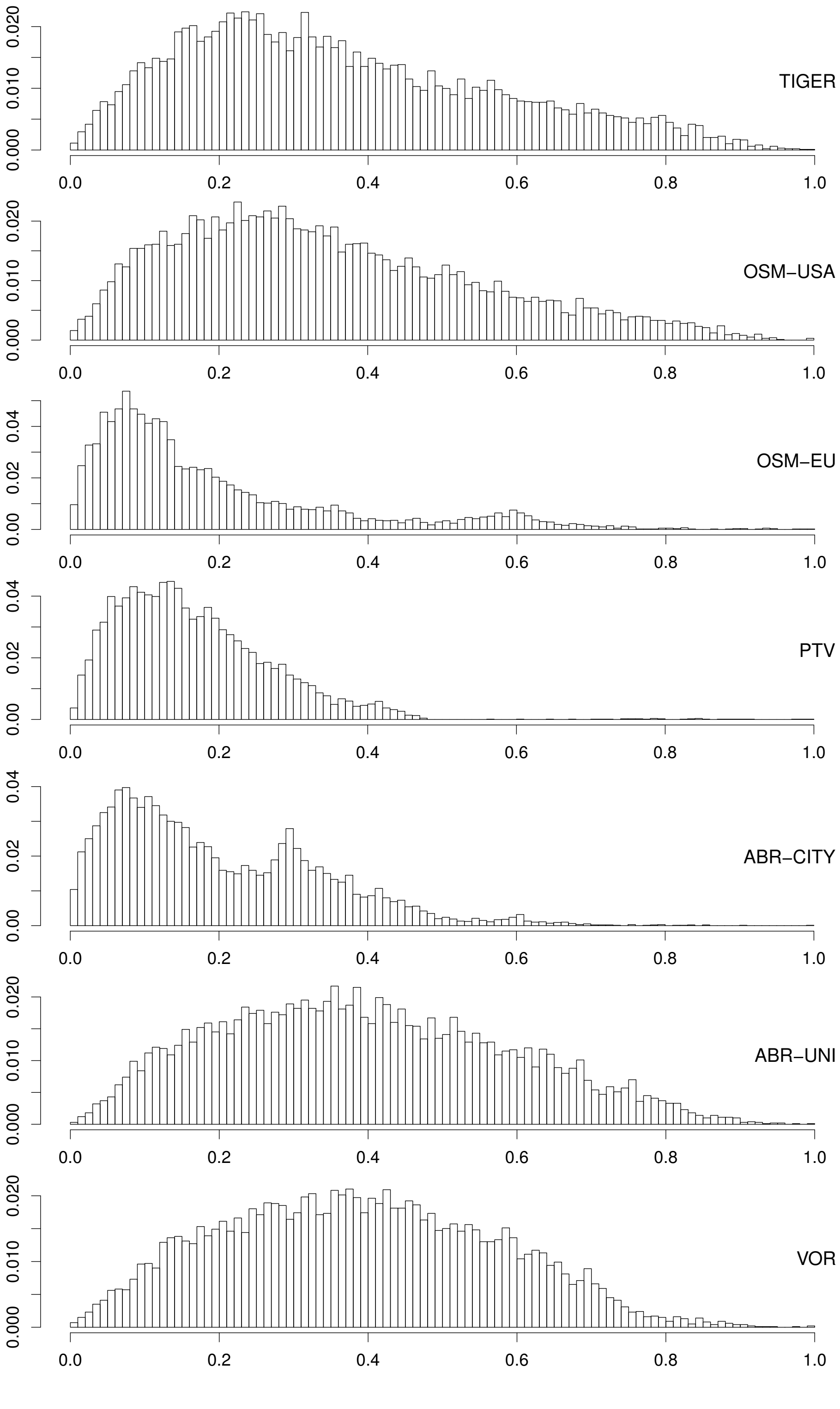


Figure 4.6: Distance distribution sampled by considering 1000 source nodes and, for each source node, 1000 target nodes. Unconnected pairs have been removed, values have been normalized.

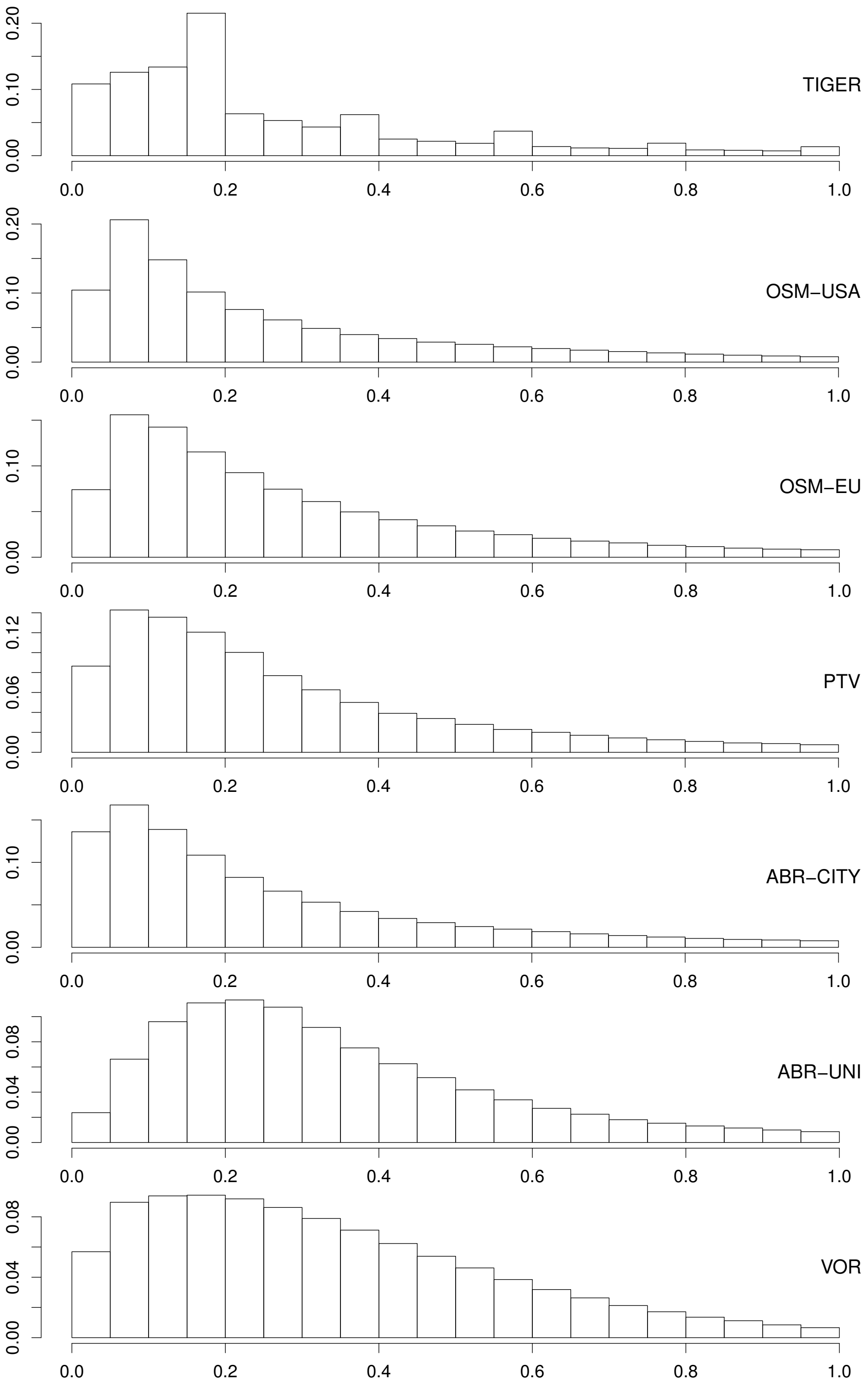


Figure 4.7: Edge-weight distributions sampled by considering 1000 source nodes and, for each source node, 1000 target nodes. Unconnected pairs have been removed, values have been normalized. For a better comparability, edge-weights have been replaced by Euclidean distances.

data. However, the latter both do not correspond to the first two data sets. We explain the difference of the data sets TIGER and OSM-USA to the data sets OSM-EU and PTV by their geographical origin. The distance distributions of the artificial instances cover well those of the real-world instances. In particular, the ABR-CITY-data set is a good approximation for the instances of OSM-EU and PTV. On the other hand, the data sets VOR and ABR-UNI approximate the OSM-USA and TIGER-data.

Edge-Weight Distribution For the sake of completeness, we also report the distributions of the according edge weights in Figure 4.6. Note that the edge-weight distribution has only a small impact on the considered algorithms [BDW07]. Hence, for better comparability, we always applied Euclidean distances instead of the original weights. We can observe that the group of real-world data sets consisting of PTV, OSM-EU and OSM-USA have similar edge-weight distributions. However, the amplitude of the OSM-USA-data is slightly larger than those of the others. The data set TIGER has a distribution different to the others. As the distance distribution of OSM-USA and TIGER were similar, we believe that the TIGER-data contains characteristics different from the others. In particular, these could possibly originate from the larger density of the TIGER-data, which indicates that bends are not modeled in high detail, and thus, the TIGER-data contains edges longer than the other real-world data sets. From the artificial instances, the data set ABR-CITY approximates the real-world instances best, whereas the data set VOR fits them only to a limited extent. The data set ABR-UNI has an edge-weight distribution different from the others, which is shifted towards long edges.

Summary In this section, we first described the parameter sets used in our generation process, and experimentally assessed scalability of the generators in terms of memory consumption and running time. It turns out the generator of Abraham et al. and our newly proposed Voronoi generator perform reasonable well in that they are capable of generating large networks that consists of hundreds of millions of nodes. Then, we analyzed structural properties of artificial and real-world instances and compared them with each other. In particular, we studied the networks with respect to density, directedness, connectedness, degree distribution, and distance distribution. Regarding these properties, we found out that the real-world instances are quite uniform in general, although on a more detailed level, they exhibit a certain amount of variance. Hence, we cannot speak of the single real-world road network but have to consider both, the underlying physical networks and their possibly different modelings. Finally, regarding the structural properties both generators are good models for the real-world data with the exception that the generator of Abraham et al. produces graphs that are too dense and incorporate nodes of too high degree.

4.5 Algorithmic Behavior

In this section, we analyze the algorithmic behavior of shortest-path speedup techniques applied to both, real-world and synthetically generated graphs. These techniques have specifically been designed for the task of heuristically speeding up Dijkstra's algorithm on real-world road networks, and thus, we will use their performance as an indicator for the similarity of the networks. In Section 2.5.3, we briefly introduced the speedup techniques used in this section but did not cover the tuning possibilities that we employ now. For a more thorough treatment on details see the work by Delling et al. [DSSW09a].

To assess similarity of algorithmic behavior, we analyze the speedups which can be achieved using the following speedup techniques for point-to-point shortest path queries as compared to Dijkstra's standard algorithm:

- bidirectional Dijkstra [Dan62],
- ALT with 16 landmarks computed by the max-cover strategy [GW05],
- approximate Reach-Based Routing with parameters $\epsilon = 0.5$ and $\delta = 16$ [GKW09],
- Arc-Flags with a partition of the graph into 128 Voronoi-cells [HKMS09] and
- Contraction Hierarchies using the original code and the parameter set of the aggressive variant [GSSD08].

In the experimental evaluation, we always use Euclidean distances as edge weights in the graphs. This results in better comparability and is well justified as changing to other metrics like travel time has only low impact on the considered algorithms [BDW07].

All the speedup techniques are written in C++ using the STL library but no other additional library. An exception is Contraction Hierarchies, which for in- and output of files additionally relies on the Boost-Iostreams library [boo]. The code was compiled with GCC 4.5 using optimization level 3. The preprocessing of the speedup techniques was conducted on Machine I, which is described in detail in Section 4.4.1 on page 85. However, we performed the experimental analysis of the query phase on a machine different from Machine I to overcome the memory latency effects, and refer to this second computer by *Machine II*. In particular, Machine II consists of one AMD Opteron 2218 CPU and 16 GB of memory. Each CPU contains two cores, which are clocked at 2.6 GHz and have an L2-cache of size 1 MB. The experiments were run using OpenSuse 11.3 on a single core of Machine II.

Benchmark Set As a benchmark set we analyze the real-world instances PTV, TIGER, OSM-EU and OSM-S-USA as well as the synthetically generated graphs VOR, ABR-UNI and ABR-CITY. In our analysis of the algorithmic behavior, we will not use the data set OSM-USA but the data subset OSM-S-USA as described in Section 4.4.2. Some of the speedup techniques were not economical with respect to memory consumption, and hence, they ran out of memory not completing the preprocessing. We did not adjust the size of the other graphs in that the sampling may have an influence on the structural properties as we have seen for OSM-S-USA in Table 4.6. In order to assess the quality of the synthetic data, we also included data from generators that are not tailored for road networks, namely grid-graphs and unit-disk graphs, which are both described in Section 4.3.3. The respective data sets are named GRID and UD.

4.5.1 Small-Scale Behavior

Since the speedup of the various techniques is non-trivially correlated with the size of the graph, we sampled random snapshots of increasing size from our benchmark set in order to be able to capture the underlying correlation. In particular, we extracted subgraphs of increasing size $s = 2^i \cdot 1000$ for $i = 0, \ldots, 9$. The sampling was performed exploiting the given geometrical layout in that we extracted square-sized subgraphs around random points. Thereby, the diameter has been adapted, such that the resulting graph had the desired size.

In the following, we present for some techniques an overview of the measured speedups achieved by applying the technique to samples of our benchmark set. The values represent the means of the acceleration gained. These tables are intended to give a rough overview on the behavior of the speedup techniques and to allow for a discussion instead of thumbing through this thesis in the search for the corresponding plots. However, these are also contained later and are referred to properly. Note that we renamed some of the data sets to preserve readability, i. e., O-EU represents OSM-EU, OS-USA represents OSM-S-USA, A-UNI represents ABR-UNI and A-CITY represents ABR-CITY.

Bi-Directional Dijkstra First, we start with the analysis of bidirectional Dijkstra but omit a presentation of the values in an overview table as the mean values do not contribute to an analysis. The more interesting point here is that road networks typically contain rather dense and rather sparse regions. Hence, depending on a selected area the speedup of bidirectional Dijkstra should exhibit a certain variance. By our sampling method, we can evaluate whether the artificial networks exhibit a similar behavior. The measured speedups range from 0.9 to 2.2 for the real-world instances and are rather concentrated around 1.5 for all networks and graph sizes, which corresponds to the estimated behavior. However, in Figure 4.9 on page 104, we can see that the artificial instances ABR-UNI, GRID and UD are highly concentrated around their mean value whereas the other artificial data sets exhibit a much larger spread. We explain this behavior with a rather uniform distribution of the nodes and edges in the geometrical layout. This assumption is strengthened as we can observe that employing the random point source rpsCity to generate the ABR-CITY-data lead to a larger variance of the measured speedups than in the ABR-UNI-data but is still significantly smaller than those of the instance VOR. Hence, we can conclude that the VOR-data sets seem to contain sparse and dense regions in a similar fashion like the real-world networks as the speedups of bidirectional Dijkstra approximate the behavior of the real-world instances best. In general, these small-spread effects of the variance with the measured acceleration holds for most of the speedup techniques, and therefore, will not be discussed there again.

(Unidirectional) ALT We now analyze a first speedup technique tailored for road networks, namely unidirectional ALT, whose measured speedups are presented in Figure 4.10 on page 105. We can observe that applying this speedup technique to the real-world instances results in a acceleration of about 10 to 15 for the largest sizes of graphs. An exception is the TIGER-data, where we can observe a slight decrease for graphs of size 128 000 and 256 000 nodes. On the other hand, for each of the synthetic-data sets the speedup grows faster than for the real-world instances. In particular, the observed speedup for GRID and UD is almost identical but almost twice as large as for the real-world graphs. The data sets ABR-UNI and ABR-CITY exhibit a curve progression of the speedup similar to GRID and UD until the graph size exceeds 128 000 nodes. From there on the speedup remains almost constant. The acceleration gained for instances of the VOR-data set is well in between the other artificial and real-world instances.

Typically, we can gain a larger acceleration of a technique if we employ its bidirectional variant. Hence, in Table 4.7, we show the speedups of bidirectional ALT and the detailed

Table 4.7: Selected outcomes of the small-scale experiment for the technique bidirectional ALT. Shown are the mean values of speedups measured on instances of increasing size, which have been sampled of the benchmark set. Note that k abbreviates a thousand.

technique	#nodes	O-EU	OS-USA	TIGER	PTV	VOR	A-CITY	A-UNI	UD	GRID
ALT	2k	1.9	3.8	3.5	5.1	4.5	8.3	8.8	5.9	6.9
ALT	32k	6.2	11.2	11.3	12.5	11.3	20.1	23.0	16.4	17.4
ALT	128k	13.3	18.0	13.5	21.4	17.5	31.5	34.4	25.1	25.5
ALT	256k	15.3	20.0	14.3	28.2	19.1	32.0	36.1	28.4	27.1
ALT	512k	16	25.0	20.1	30.3	24.1	30.9	34.4	29.3	27.2

outcomes in Figure 4.11 on page 106. In general, we can observe curve progressions similar to the unidirectional variant with an observed speedup that is about two times larger. In more detail, PTV and OSM-USA seem to result in a slightly larger speedup compared to the other real-world instances whereas OSM-EU does not gain a doubling of speedup. Again the curve progressions of ABR-UNI and ABR-CITY are similar to each other exhibiting again the nearly constant curve progression starting at a graph size of 128 000 nodes. The instances of UD and GRID behave similar to each other but do not gain a speedup that is twice as large as for the unidirectional case. For instances of smaller size the gained speedup exceeds those of the real-world instance but is in between for instances of size 256 000 nodes. For bidirectional ALT, the data set of VOR exhibits a speedup well in between the real-world instances with a similar curve progression. In summary, we can observe that the real-world instances exhibit a large variance of their speedups, and that the data sets ABR-UNI and ABR-CITY respond too well on this technique. From the set of artificial instances, the VOR-data approximates the behavior of the real-world networks best.

(Unidirectional) Arc-Flags We now look at Arc-Flags, which is another goal directed speedup technique. In Figure 4.12 on page 107, we show the speedups measured using the unidirectional variant. For the largest size of real-world instances, speedups for this technique are in the range of 22 to 47. Arc-Flags applied to the following groups of instances results in a similar behavior: TIGER equals OSM-EU, PTV equals OSM-USA and VOR, GRID, equals UD, and ABR-CITY equals ABR-UNI. We can observe that for ABR-UNI and ABR-CITY the curve progression of the speedup is almost constant as soon as the instance size exceeds 64 000 nodes. The speedup of VOR is well in between PTV and OSM-USA, whereas the speedup of UD and GRID is slightly below the observed speedups of the real-world instances.

Similar to ALT, we will now look at the bidirectional variant of Arc-Flags. The brief overview is given in Table 4.8 and detailed plots can be found in Figure 4.13 on page 108. We first observe that the gained speedups exceed those of ALT both, in general and in their order compared to the unidirectional approach. In particular, for the largest size of instances speedups range from 60 to 460. We can further separate the data sets into three groups according to their observed speedup behavior. The first consists of ABR-CITY and ABR-UNI, which exhibit a constant speedup for graphs of size 64 000 nodes and

Table 4.8: Selected outcomes of the small-scale experiment for the technique bidirectional Arc-Flags. Shown are the mean values of speedups measured on instances of increasing size, which have been sampled of the benchmark set. Note that k abbreviates a thousand.

technique	#nodes	O-EU	OS-USA	TIGER	PTV	VOR	A-CITY	A-UNI	UD	GRID
Arc-Flags	2k	3.4	7.7	8.2	12.7	11.8	24.1	25.2	19.3	18.35
Arc-Flags	32k	18.7	39.1	45.5	63.3	68.5	142.3	148.4	53.6	46.7
Arc-Flags	128k	52.7	78.7	61.4	137.8	124.4	187.4	219.0	70.4	56.3
Arc-Flags	256k	60.5	86.3	65.3	205.2	160.7	183.2	227.0	75.3	73.9
Arc-Flags	512k	80.0	154.2	98.5	292.7	229.7	194.0	236.6	80.3	76.5

larger, which is a behavior similar to the unidirectional case. The second group consists of PTV and VOR, whose curve progressions of the speedup is much larger than those of the third group, which consist of the remaining generators. In summary, we can say the artificial-data sets GRID and UD are good approximations of the TIGER-, OSM-USA- and OSM-EU-data, whereas VOR-data is a good approximation of PTV-data.

Reach-Based Routing We now examine speedups for Reach-Based Routing, which in our comparison is the first technique that exploits the hierarchy in a network. We present the overview of the measured acceleration in Table 4.9 and the detailed plots in Figure 4.14 on page 109. For the largest size of instances, the observed speedups range from 4 to 107. The group consisting of the data sets ABR-UNI, ABR-CITY, GRID and UD exhibit a similar and at the same time smaller speedup behavior compared to the other instances. The data set TIGER exhibits the smallest speedup among the real-world instances and roughly equals PTV- and OSM-EU-data. The last of the real-world instances, namely OSM-USA, behaves similarly to the others but achieves a larger acceleration, especially for larger sizes of the instances. The behavior of Reach-Based Routing does not seem to be very well approximated by any of the synthetic generators. Up to 256 000 nodes, the behavior is best approximated for all real-world instances by the VOR-data. However, for 512 000 nodes the speedup achieved by Reach-Based Routing on instances of VOR is significantly larger than that achieved on the real-world graphs. On the other hand, all other synthetic generators do not seem to capture the behavior of Reach-Based Routing very well. Most remarkably, the data-sets ABR-UNI and ABR-CITY do not respond to the hierarchical speedup technique as the real-world networks do, although they should exhibit a hierarchy similar to road networks. In summary, we conclude that VOR-data exhibits a strong hierarchy within the network, whereas the artificial data sets GRID and UD do not capture this property at all and ABR-UNI and ABR-CITY only to a limited extent.

Contraction Hierarchies Finally, we analyze the behavior of Contraction Hierarchies, which in our comparison is the latest developed technique. We present the particular outcomes in Figure 4.15 on page 110 and the overview in Table 4.10. For the largest size of real-world instances, mean speedups are in the range of 494 to 1 056. We can observe that the real-world data sets TIGER and OSM-EU behave rather similar to each other except for the variance found in OSM-EU, which is much larger for graphs

Table 4.9: Selected outcomes of the small-scale experiment for the technique Reach-Based Routing (Reach). Shown are the mean values of speedups measured on instances of increasing size, which have been sampled of the benchmark set. Note that k abbreviates a thousand.

technique	#nodes	O-EU	OS-USA	TIGER	PTV	VOR	A-CITY	A-UNI	UD	GRID
Reach	2k	3.5	3.7	2.7	3.3	2.8	3.8	4.0	1.6	1.2
Reach	32k	12.3	19.8	9.7	13.6	11.5	7.8	8.1	3.1	2.1
Reach	128k	24.2	25.3	14.9	23.4	22.9	7.7	8.7	3.7	2.9
Reach	256k	36.6	46.6	20.5	26.8	33.2	8.2	8.7	4.0	4
Reach	512k	31.0	44.6	15.0	23.5	49.2	8.8	9.6	4.6	4.7

Table 4.10: Selected outcomes of the small-scale experiment for the technique Contraction Hierarchies (CH). Shown are the mean values of speedups measured on instances of increasing size, which have been sampled of the benchmark set. Note that k abbreviates a thousand.

technique	#nodes	O-EU	OS-USA	TIGER	PTV	VOR	A-CITY	A-UNI	UD	GRID
CH	2k	24.3	33.6	22.4	28.9	20.4	18.9	19.7	13.3	9.4
CH	32k	143.4	253.4	124.85	207.8	202.9	94.6	94.2	72.7	47.6
CH	128k	323	371.4	322.7	611.3	633.7	209.9	216.3	140.7	86.3
CH	256k	633.2	802.1	496.9	810.4	1 139.4	294.5	312.4	188.9	132.0
CH	512k	522.5	1 056.1	494.9	823.1	1 979.6	433.9	453.7	254.7	182.2

of size 256 000 nodes. The remaining real-world instances PTV and OSM-USA roughly exhibit similar behavior of the speedup although a certain variance can be observed. Note that the speedup measured for this group is almost twice as large as for the aforementioned real-world instances. The group consisting of the artificial instances ABR-UNI and ABR-CITY exhibit almost no spread in their speedups. However, their speedups roughly match the mean values of the less accelerated real-world instances. On the other hand, the speedups of the VOR-data set by far exceed those of the others. In contrast, GRID- and UD-data exhibit similar but smallest speedups, and hence, do not approximate any of the real-world data sets.

Summary Small-Scale Behavior In this section, we analyzed the algorithmic behavior of samples of the benchmark set with respect to shortest-path speedup techniques. The results of the small-scale analysis can be summarized as follows. We observed that real-world graphs significantly differ in their algorithmic behavior: Although the data sets OSM-EU and TIGER behave similarly with respect to the speedup techniques ALT, Arc-Flags and Contraction Hierarchies, the two speedup techniques differ by almost a factor of two with respect to Reach-Based Routing. Even worse, the OSM-USA- and PTV-data sets achieve similar speedups compared to each other, whereas the speedup is almost twice as large as OSM-EU or TIGER with respect to Contraction Hierarchies. However, with respect to Reach-Based Routing, the OSM-USA-data allows for an acceleration almost twice as large as for PTV-data but with respect to Arc-Flags, we can observe a behavior the other way round. The observation of algorithmic variance is strengthened by our comparison chain. Although the data sets OSM-EU and OSM-USA both originate from the same modeling process, we observe different algorithmic behavior. Additionally, the instances of the continental counterparts TIGER versus OSM-USA and PTV versus OSM-EU exhibits different behavior. Hence, we conclude that the networks on the one hand differ in terms of structural properties, and on the other hand, the algorithmic behavior also depends on the modeling process.

The data sets GRID and UD approximate road networks only to a limited extent: Both GRID and UD behave similarly for all speedup techniques we considered. Although both graphs seem to be rather good approximations for PTV with respect to ALT and for both OSM-EU and TIGER with respect to Arc-Flags, they seem to be rather bad estimates for Contraction Hierarchies and Reach-Based Routing. This may be explained by the lack of hierarchy inherent to both generators and the fact that both techniques are based on hierarchical decompositions of the underlying graphs.

The data sets ABR-UNI, ABR-CITY and VOR are reasonably good approximations: Contrary to GRID and UD, the aforementioned seem to capture both the magnitude and the trend exhibited by real-world instances quite well. In particular, the speedups measured for ABR-UNI and ABR-CITY are slightly too large for ALT and Arc-Flags and they are too small for Reach-Based Routing. For Contraction Hierarchies, on the other hand, they are very close to the speedups measured for OSM-EU and TIGER. Except for Contraction Hierarchies, the speedups measured for VOR are well in between the speedups measured for the real-world instances. For Contraction Hierarchies, the speedups are close to the speedups for PTV and OSM-USA until the graph size exceeds 128 000 nodes, and rises then faster than for all other data sets.

Note that parameters of the synthetic data have not been fine-tuned for approximating the given results. Since there is considerable amount of deviation in the behavior of the real-world data we consider doing so to be over-fitting.

4.5.2 Large-Scale Behavior

Typically, for several speedup techniques the achieved acceleration grows with the distance between the start and destination point. Hence, in addition to the analysis of the sampled data set, we also analyzed the speedup achieved for selected techniques applied to the whole instances of the benchmark set. In our comparison, we did not include GRID and UD graphs as it already turned out that they are no good models for road networks. To gain an overview on how the speedup techniques behave with growing distance, we performed *Dijkstra rank* queries for 1 000 randomly selected source nodes; recall Section 2.5.2. The use of Dijkstra rank queries for measuring the performance of speedup techniques in large road networks is typically preferred over purely random queries as the latter usually favors mid- to long-range queries over short-range queries. The reason for also taking short range queries into account is the fact that people typically travel locally with respect to road networks of continental size, and hence, speedups measured only for mid- to long range distances possibly turn out to be overrated. For a source node target nodes of increasing Dijkstra rank are randomly selected, which results in a total number of 25 000 up to 28 000 queries depending on the size of the graph. By performing these queries, we analyze the behavior of ALT, Contraction Hierarchies and Reach-Based Routing.

We also planned to incorporate Arc-Flags, however, we were not able to preprocess the data within reasonable time. A reason for this can be found in the running time of the preprocessing phase. In particular, this involves for each cell the computation of a shortest-path tree for each of its boundary nodes. From our computation of Dijkstra ranks, we know that the running time for a single-source-all-targets Dijkstra query roughly ranges from 15 to 30 seconds depending on the graph and possibly on its size. Our partition of the instances resulted in the following number of boundary nodes: PTV: 100 490, OSM-EU: 43 007, OSM-USA: 25 742, TIGER: 92 398, VOR: 118 451, ABR-CITY: 513 925 and ABR-UNI: 578 549. Although we did not analyze Arc-Flags, we can see that the behavior of the VOR-data equals those of the real-world instances with respect to the number of boundary nodes of the partitions. On the other hand, the instances ABR-UNI and ABR-CITY seem not to allow for a good partitioning of the graphs as it is possible for road networks.

In Figure 4.8, we present boxplots of the measured speedups achieved on our benchmark set. The plot is our basis for discussion and we refer to the more detailed Dijkstra rank plots shown later.

ALT We first discuss the behavior of uni- and bidirectional ALT presented in Figure 4.16 and in Figure 4.17 on page 111 et seq., respectively. We can observe that the speedup of bidirectional ALT is not twice as large as for the unidirectional case but yields only slight improvements. For both approaches, the general curve progression of the measured speedups is similar and differs only by a small amount. The real-world instances behave rather similar although PTV-data allows for a larger acceleration for long distance queries. The instances ABR-UNI and ABR-CITY exhibit a larger speedup for short- to mid-range queries, which results in a larger mean speedup. The speedup of the VOR-data is almost similar to the real-world data with respect to both, mean values and curve progression.

Reach-Based Routing Regarding Reach-Based Routing, we can observe that the real-world instance behave similar except the TIGER-data. In particular, the TIGER-data gains a smaller speedup in terms of mean value as well as curve progression, which can be seen in Figure 4.18 on page 113. In particular, the curve progression of the speedups for the instances PTV, OSM-EU and OSM-USA are almost similar except for very long distance queries. On the other hand, the instances ABR-UNI and ABR-CITY exhibit almost all the time a smaller curve progression, although their mean value of the speedup is identical to the real-world instances. However, the upper interquartile range is much smaller than those of the real-world networks. In contrast, the VOR-data exhibits a much larger upper interquartile range albeit its speedup mean value is also similar to the real-world graphs. When looking at the detailed curve progression, we can see that the VOR-data has an increased speedup for mid- to long-range queries. To a certain extent, the instance ABR-UNI and ABR-CITY approximate the behavior of real-world instances from below, whereas VOR-data exhibits speedups slightly larger than those of the real-world networks.

Contraction Hierarchies The detailed measured speedups of Contraction Hierarchies are presented in Figure 4.19 on page 114. We can again observe that the speedup of Contraction Hierarchies is orders of magnitude larger than the previously analyzed techniques. In particular, the speedups of TIGER-data rather equals those of OSM-EU-data and PTV-data equals OSM-USA. In the detailed curve progressions, we can

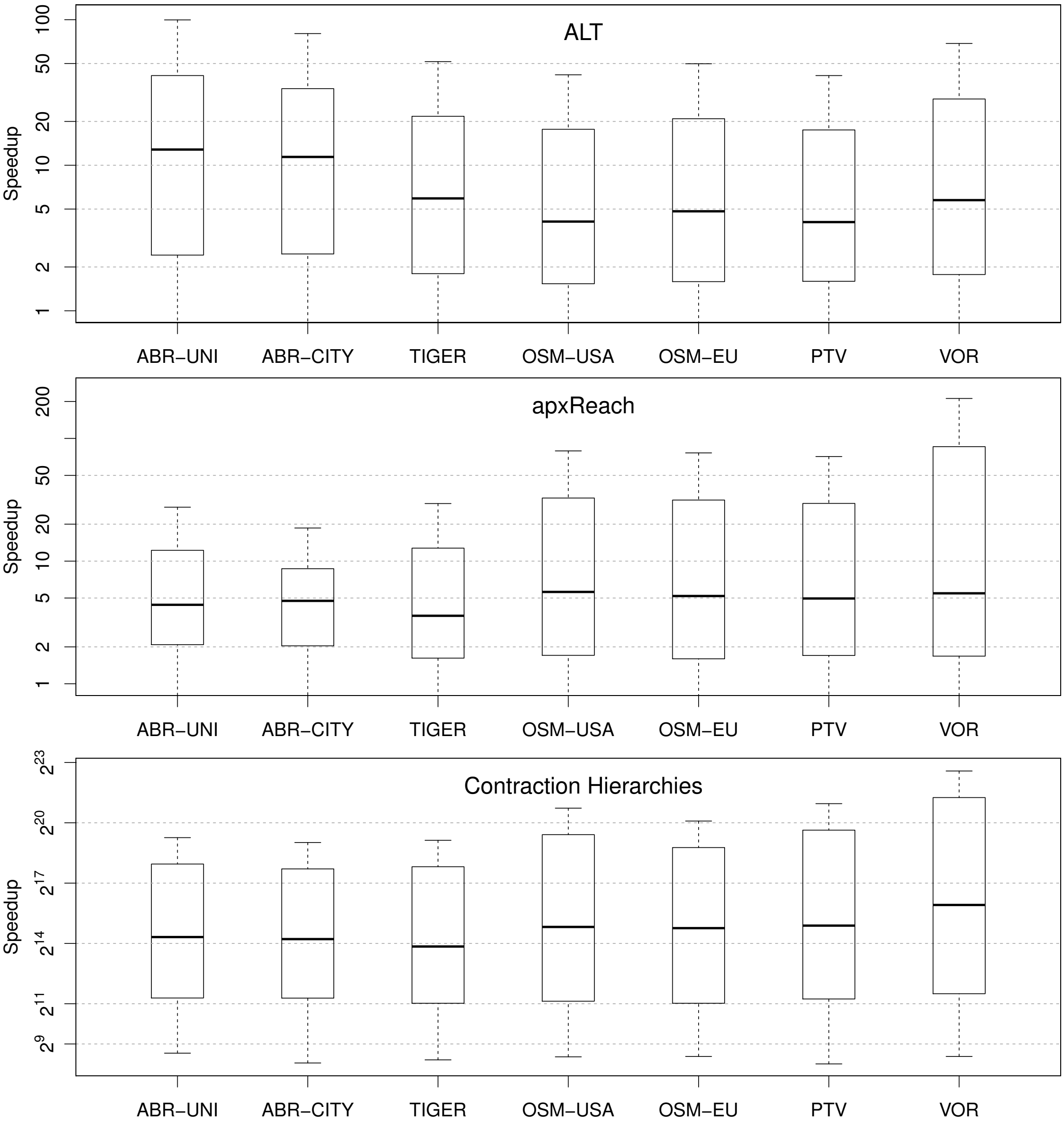


Figure 4.8: Overview of the speedups measured applying different shortest-path techniques on the benchmark set. From top to bottom, ALT, Reach-Based Routing (apxReach) and Contraction Hierarchies. The boxplots show the values of 25 000 to 28 000 queries (depending on the graph size) that were performed between 1 000 randomly selected start nodes and for each a set of target nodes selected according to increasing Dijkstra rank. selected Note that the measured speedups are shown on a logarithmic scale.

observe that the speedups of PTV-data are slightly increased compared to OSM-USA. We can also see that the speedups of the instances ABR-UNI and ABR-CITY are well in between those of TIGER and OSM-EU, possibly except for very long distance queries. In contrast, the VOR-data exhibits always larger speedups than PTV-data can achieve, however, the curve progression is rather similar.

Summary Large-Scale Behavior In summary, we can say that also on large scale, the real-world data exhibit a certain variance in algorithmic behavior with respect to shortest-path speedup techniques. The analyzed artificial instances capture this behavior quite well in general, although we cannot determine a specific generator that is suited best to approximate the behavior of all real-world instances. The different techniques exploit various properties of the networks and the artificial generators mimic those diversely. On a more detailed level, the VOR-data exhibits an improved approximate behavior as the mean values and the curve progression for the techniques ALT and REACH-BASED ROUTING are closer to those of the real-world instances compared to the instances ABR-UNI and ABR-CITY. For the technique CONTRACTION HIERARCHIES, the measured speedups of the real-world instances are mostly in between those of VOR and both, ABR-UNI and ABR-CITY.

4.6 Concluding Remarks

A most successful example for the application of the cycle of Algorithm Engineering is the field of research that deals with the acceleration of shortest-path algorithms. The goal here is to highly speed up shortest-path queries compared to classical Dijkstra while returned answers remain correct, i. e., they are no approximate solution. These speedup techniques are heuristically motivated and tailored for road networks. To assess the performance of newly developed techniques, their behavior is typically evaluated experimentally. However, the number of available road networks is limited, and hence, such evaluations leave the question for a broad applicability or robustness open. It could be satisfactory to remain in the current state, however, freely available commercial road networks are rather old and collaborative road networks might be inaccurate as they typically lack a quality process. Additionally, the modeling gets more and more accurate and we expect road networks that cover the entire world to be available in the next years. This raises the additional question to which extent a specific speedup technique scales.

In this chapter, we aimed at overcoming the currently limited availability of road networks, especially with respect to improving the situation for experimental evaluations of speedup techniques. In particular, we wanted to extend the variety and increase the absolute size of road networks. To this end, we followed a common approach, which in this case is to generate realistic artificial instances that hopefully exhibit properties typically found in real-world road networks. We assume an artificial instance to be realistic if it has similar characteristics compared to real-world instances, where we rate the characteristics in terms of visual appearance, structural properties and algorithmic behavior.

We gave an overview of the three sources known to us where scientists can find instances of road networks, which are PTV, a commercial data set of Europe from 2005, Tiger/Line, provided by the administration of the USA, and OpenStreetMap (OSM), a collaborative project aiming at covering the world. From these sources, we gained four instances of real-world road networks, two covering Europe and two covering the USA. In doing so, we establish a comparison chain that allows comparing the road network of the USA with that of Europe, where the OSM-data acts as an intermediary.

Having these real-world road networks at hand, we introduce a first model and an idea of a generator, both proposed by Abraham et al. [AFGW10]. The focus of the work by Abraham et al. was the development of a concept called highway dimension, which allows for proving running times of speedup techniques on graphs that exhibit a small highway dimension. Hence, the generator by Abraham et al. primarily aims at the generation of networks that have low highway dimension and not directly in generating realistic road networks. However, this generator had never been implemented, and hence, not experimentally verified whether it exhibits properties similar to road networks at all. In the spirit of Algorithm Engineering, we implemented the generator and report on both, our implementation and how we filled the degrees of freedom therein. From a first look, the hereby generated graphs have no visual appearance similar to real-world road networks. In our opinion, a main reason for this is that the model does not incorporate the Steiner property, which Abraham et al. point out was not their intention.

Based on these observations, we derive an alternative model that incorporates both, the Steiner property and an inherent hierarchy. Another major difference is that we interpret points as resources, which are not directly connected by road segments but can be connected fairly to nearby roads. This allows for a recursive algorithm that relies on Voronoi diagrams, which we used in our implementation. Compared to graphs generated by the model of Abraham et al., graphs generated by the newly proposed model exhibit an improved and much more realistic visual appearance.

For both generators, we presented the parameter sets used by our implementations to create the benchmark set for the evaluations. We also analyzed the scalability of the generators with respect to the size of the instances, memory consumption and running time, which turns out to be reasonable. To assess the quality of both models, we additionally employed grid- and unit-disk graph generators, and added thereby generated graphs to our benchmark set.

We then compared the artificially generated instances with the real-world instances with respect to structural properties, like connectedness, directedness, degree distribution, density and distance distribution. In doing so, we observed a certain variance among the real-world instances, which cannot be explained by different modeling procedures alone. Hence, we concluded that the underlying road-networks also exhibit different structures. With respect to the structural properties, we found out that the tailored generators are good models for the real-world data, despite the generator of Abraham et al. [AFGW10] produces graphs that are too dense and incorporate nodes of too high degree.

As the structural properties of the artificially generated instances mimicked those of the real-world instances, we next focused on the analysis of the algorithmical behavior. In particular, we concentrated on point-to-point shortest-path computations, as techniques developed there have been highly tailored for road networks, and hence, they exploit properties found therein. At first, we analyzed the shortest-path behavior of speedup techniques on small-scale, where we sampled graphs of increasing size from the originating network. The acceleration techniques we examined were ALT, Arc-Flags, Reach-Based Routing and Contraction Hierarchies. By the analysis of their behavior, we found that grid- and unit-disk graphs approximate the real world instances only to a very limited extent. Additionally, the real-world instances significantly differ in their algorithmic behavior. On the other hand, the artificial instances generated by both models exhibit an algorithmic behavior within the variance found in the real-world instances. We also evaluated the speedup behavior on large-scale, i. e., within the entire network. In particular, we analyzed the algorithmic behavior of ALT, Reach-Based Routing and Contraction Hierarchies. With respect to the measured speedups, the artificial instances behave similarly to the real-world instances.

In summary, we passed through the cycle of Algorithm Engineering and end up with two models and for each a ready-to-use generator, which is capable of generating huge artificial instances within reasonable time. The thereby generated instances seem to be reasonably good approximations for the real-world instances, albeit the generator of Abraham et al. does not generate instances that exhibit a visual appearance similar to road networks. On a more detailed view, our newly proposed generator gains advantages over the generator of Abraham et al. with respect to both, structural properties and algorithmic behavior. From the point of view to enlarge the variety within instances of specific size, we emphasize that the parameter sets of the artificial generators were not fine-tuned to approximate a particular real-world network or its behavior. Hence, the generators still contain degrees of freedom, which allow for steering their inherent properties, without losing the property to generate graphs similar to road networks.

Future Work To allow for the creation of even larger road networks, both generators can be engineered further. In particular, the generator of Abraham et al. consumes much memory due to the fact that it stores points in multiple levels. A reduction would allow for the generation of larger graphs. On the other hand, our newly proposed generator involves a rather large running time compared to the aforementioned generator, and hence, an acceleration would improve its usability. Another interesting question targets on the theoretical foundation of road networks, which is yet not fully understood. Possibly, our Voronoi generator allows for establishing a framework similar to that of highway dimension.

4.7 Appendix of Plots

In this section, we present figures that show details of the experimental analysis of the speedup techniques conducted in Section 4.5. We deferred these plots as their number would have reduced the readability of the previous sections.

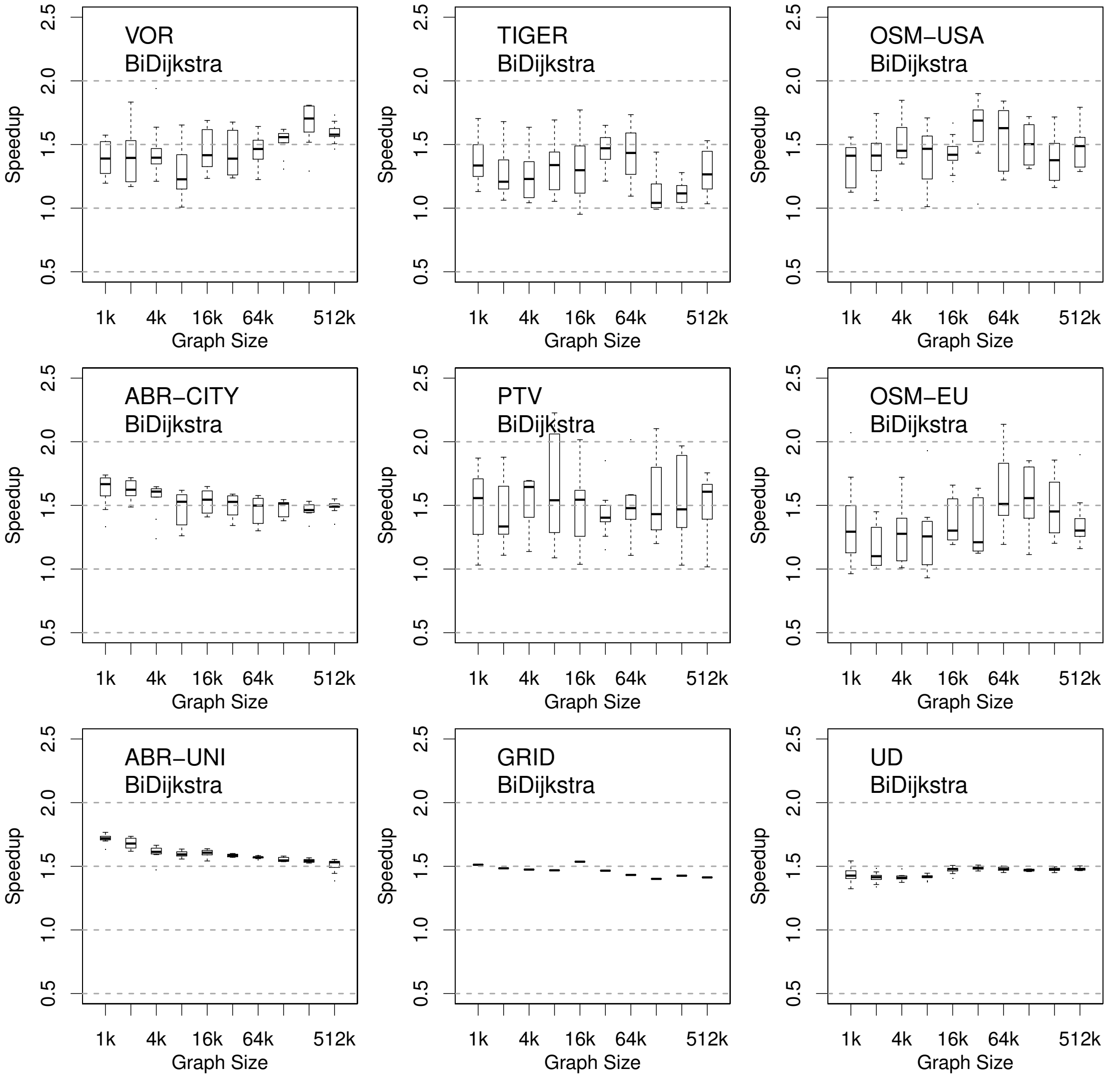


Figure 4.9: Boxplots of the speedup achieved by the technique bidirectional Dijkstra applied to samples of the benchmark set. The samples have size $2^i \cdot 1\,000$ for $i = 0, \ldots, 9$. For each size, 10 geometrical square-sized subgraphs have been sampled. To allow for a comparison of the speedup, 1 000 queries between randomly selected nodes are performed for each sample.

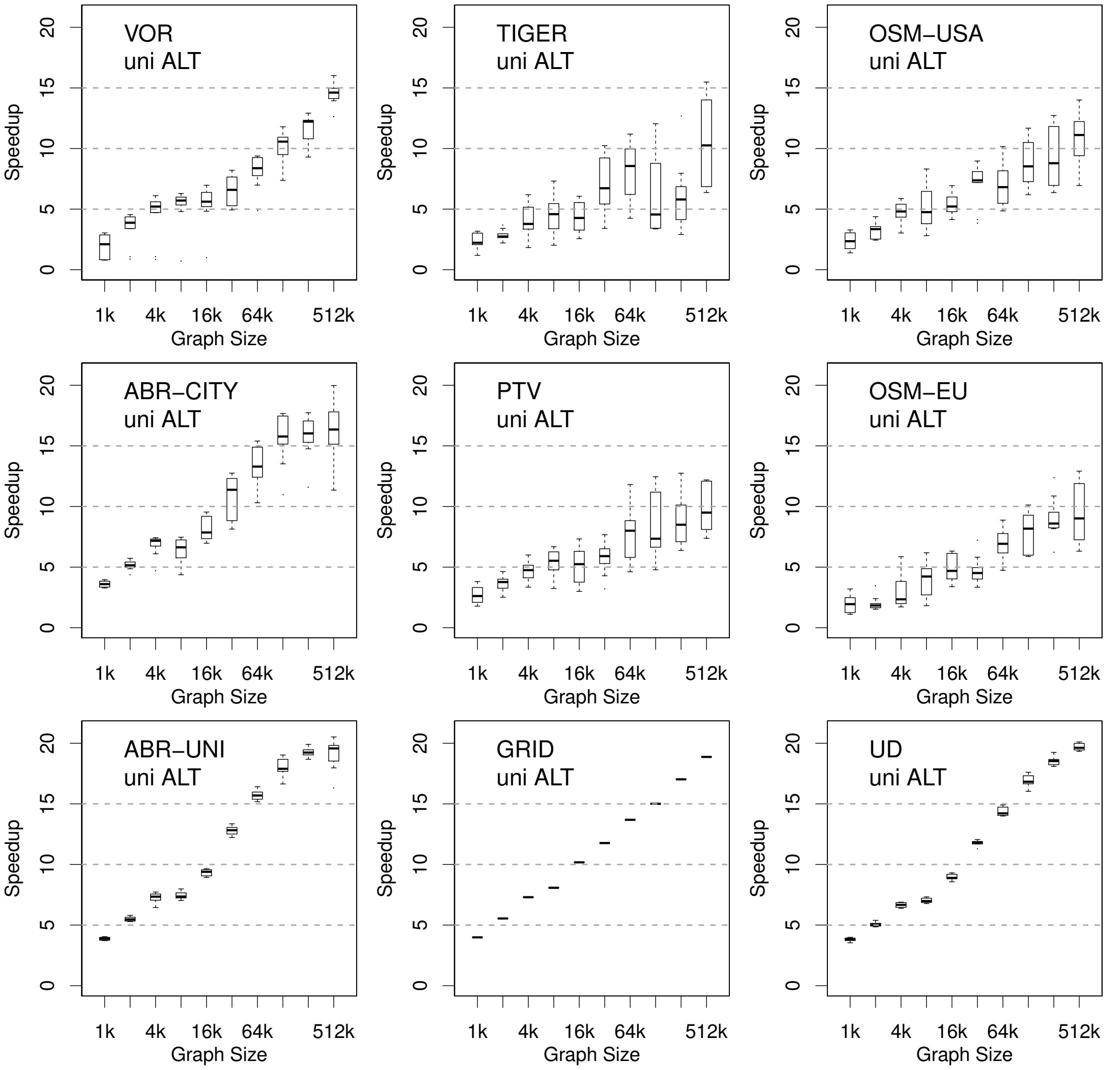


Figure 4.10: Boxplots of the speedup achieved by the technique unidirectional ALT (uni ALT) applied to samples of the benchmark set. The samples have size $2^i \cdot 1\,000$ for $i = 0, \ldots, 9$. For each size, 10 geometrical square-sized subgraphs have been sampled. To allow for a comparison of the speedup, 1 000 queries between randomly selected nodes are performed for each sample.

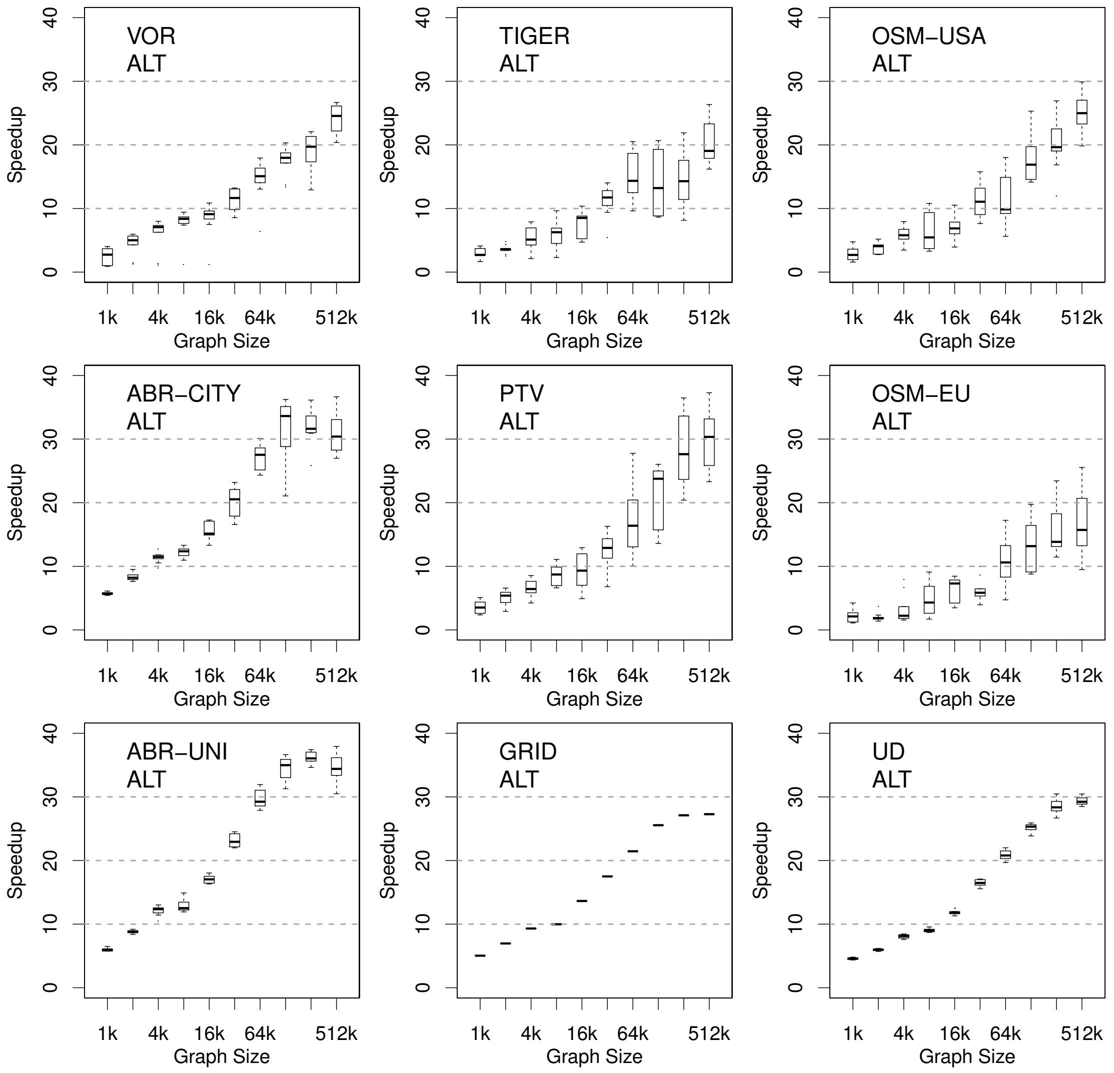


Figure 4.11: Boxplots of the speedup achieved by the technique bidirectional ALT (ALT) applied to samples of the benchmark set. The samples have size $2^i \cdot 1\,000$ for $i = 0, \ldots, 9$. For each size, 10 geometrical square-sized subgraphs have been sampled. To allow for a comparison of the speedup, 1 000 queries between randomly selected nodes are performed for each sample.

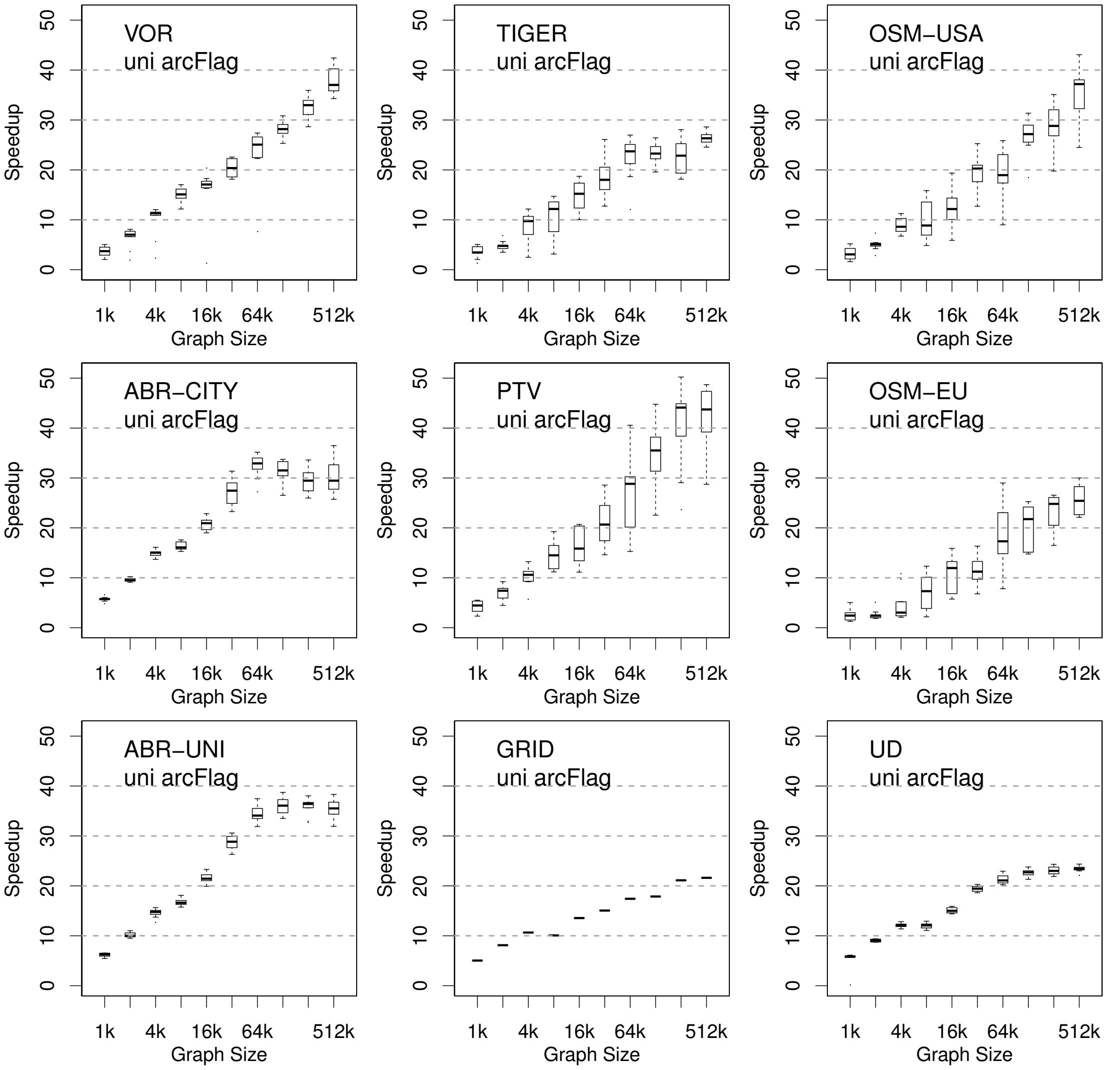


Figure 4.12: Boxplots of the speedup achieved by the technique unidirectional Arc-Flags (uni arcFlag) applied to samples of the benchmark set. The samples have size $2^i \cdot 1\,000$ for $i = 0, \dots, 9$. For each size, 10 geometrical square-sized subgraphs have been sampled. To allow for a comparison of the speedup, 1 000 queries between randomly selected nodes are performed for each sample.

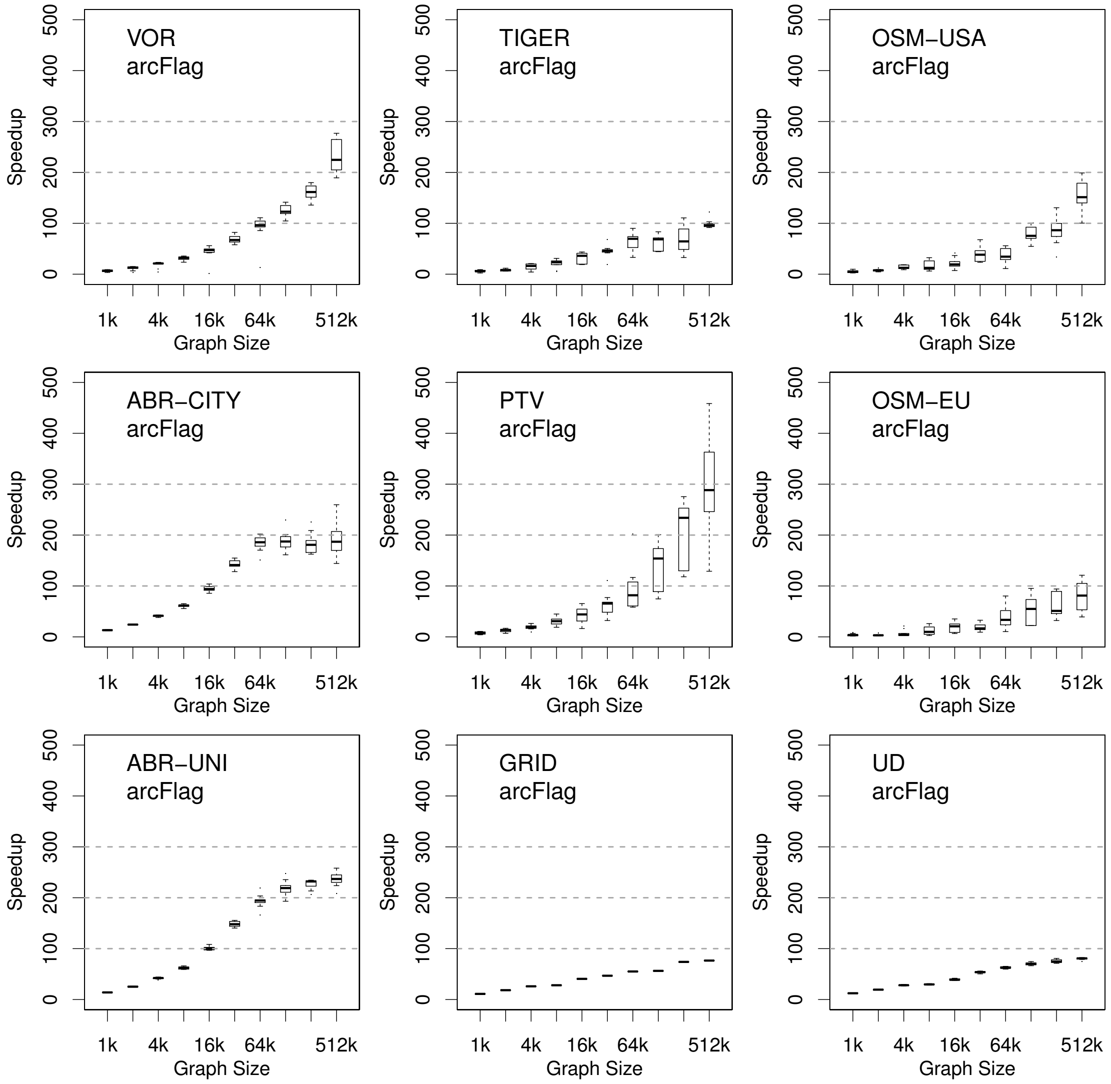


Figure 4.13: Boxplots of the speedup achieved by the technique bidirectional Arc-Flags (arcFlag) applied to samples of the benchmark set. The samples have size $2^i \cdot 1\,000$ for $i = 0, \ldots, 9$. For each size, 10 geometrical square-sized subgraphs have been sampled. To allow for a comparison of the speedup, 1 000 queries between randomly selected nodes are performed for each sample.

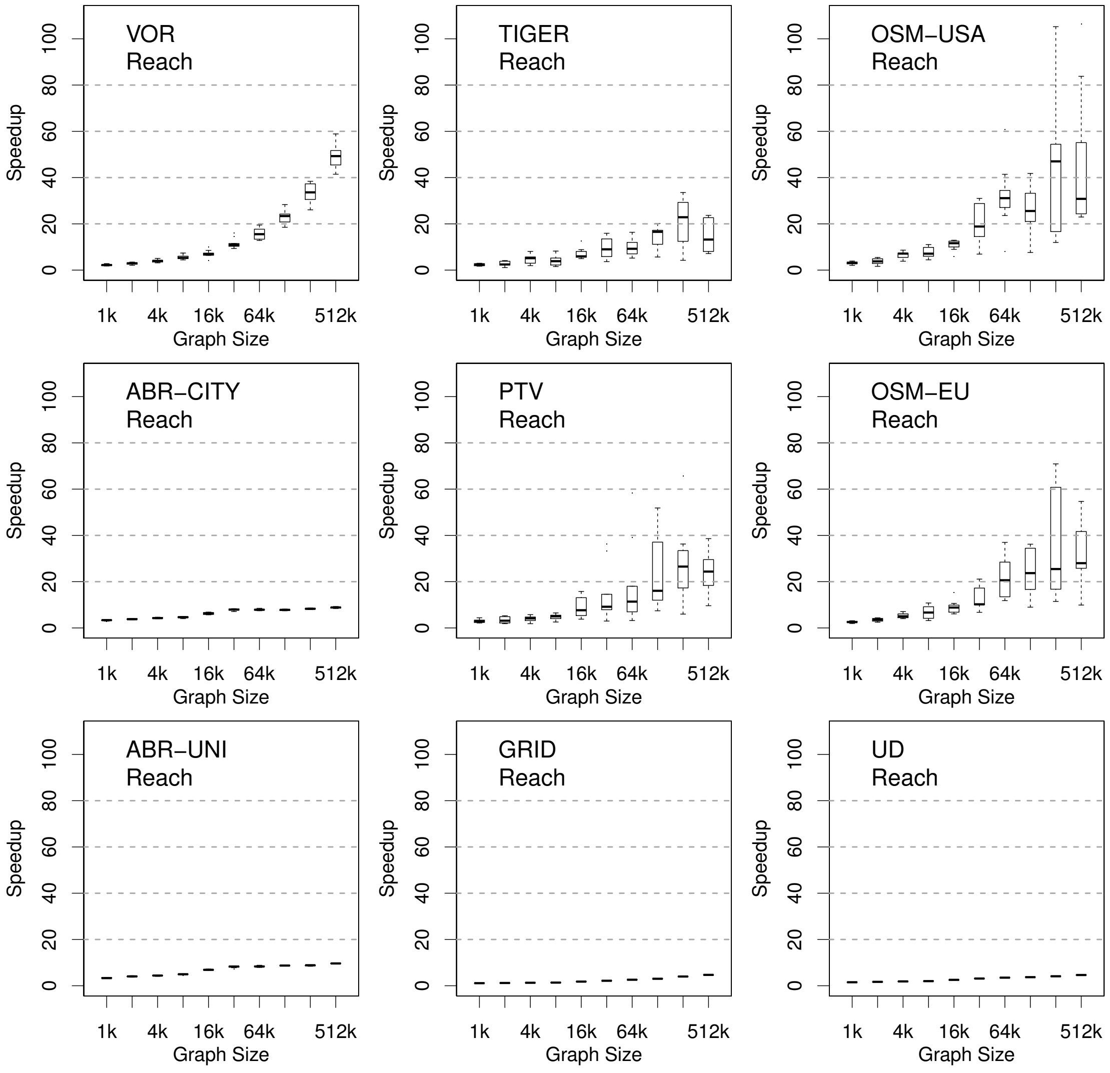


Figure 4.14: Boxplots of the speedup achieved by the technique Reach-Based Routing (Reach) applied to samples of the benchmark set. The samples have size $2^i \cdot 1\,000$ for $i = 0, \ldots, 9$. For each size, 10 geometrical square-sized subgraphs have been sampled. To allow for a comparison of the speedup, 1 000 queries between randomly selected nodes are performed for each sample.

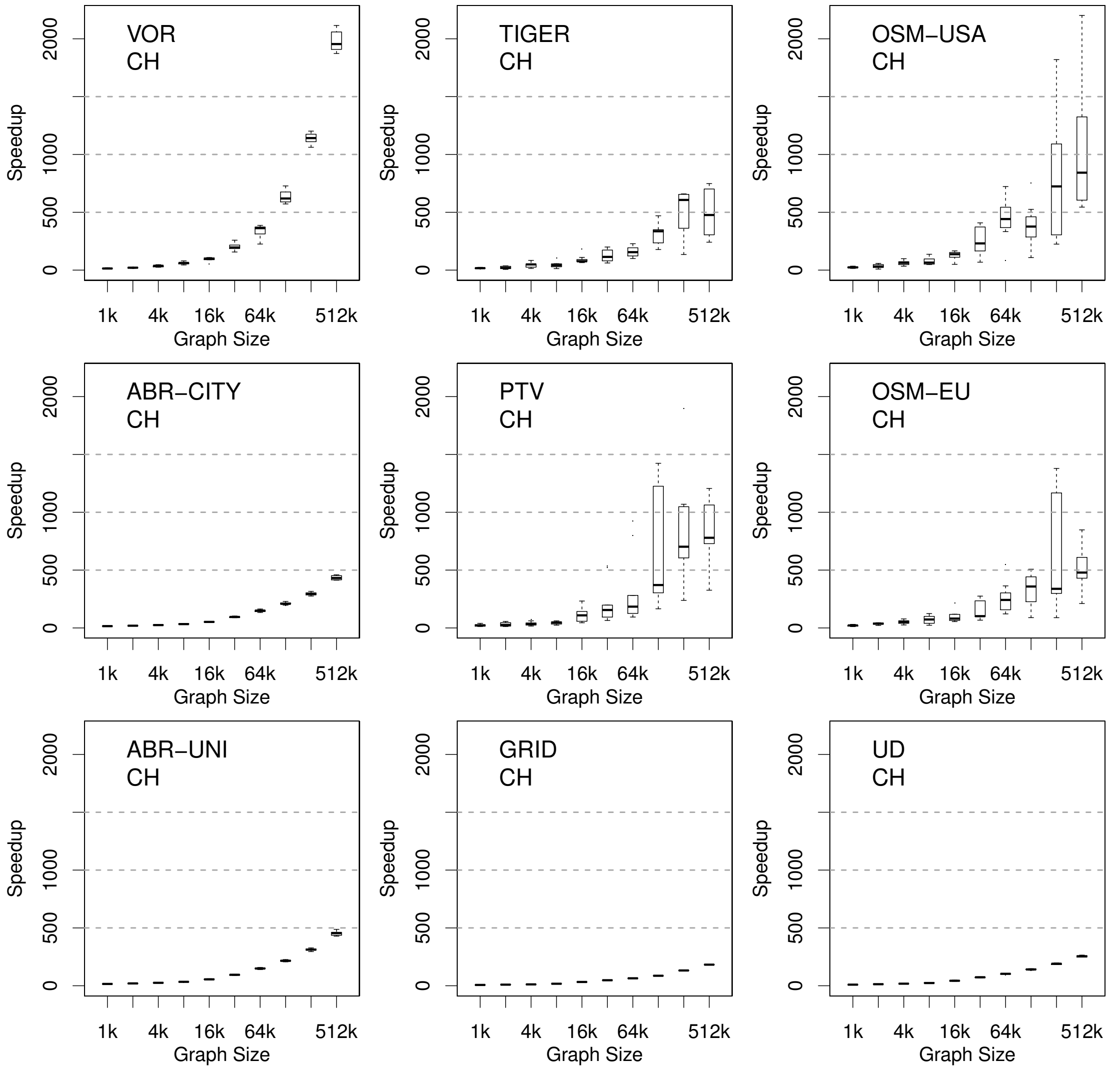


Figure 4.15: Boxplots of the speedup achieved by the technique Contraction Hierarchies (CH) applied to samples of the benchmark set. The samples have size $2^i \cdot 1\,000$ for $i = 0, \ldots, 9$. For each size, 10 geometrical square-sized subgraphs have been sampled. To allow for a comparison of the speedup, 1 000 queries between randomly selected nodes are performed for each sample.

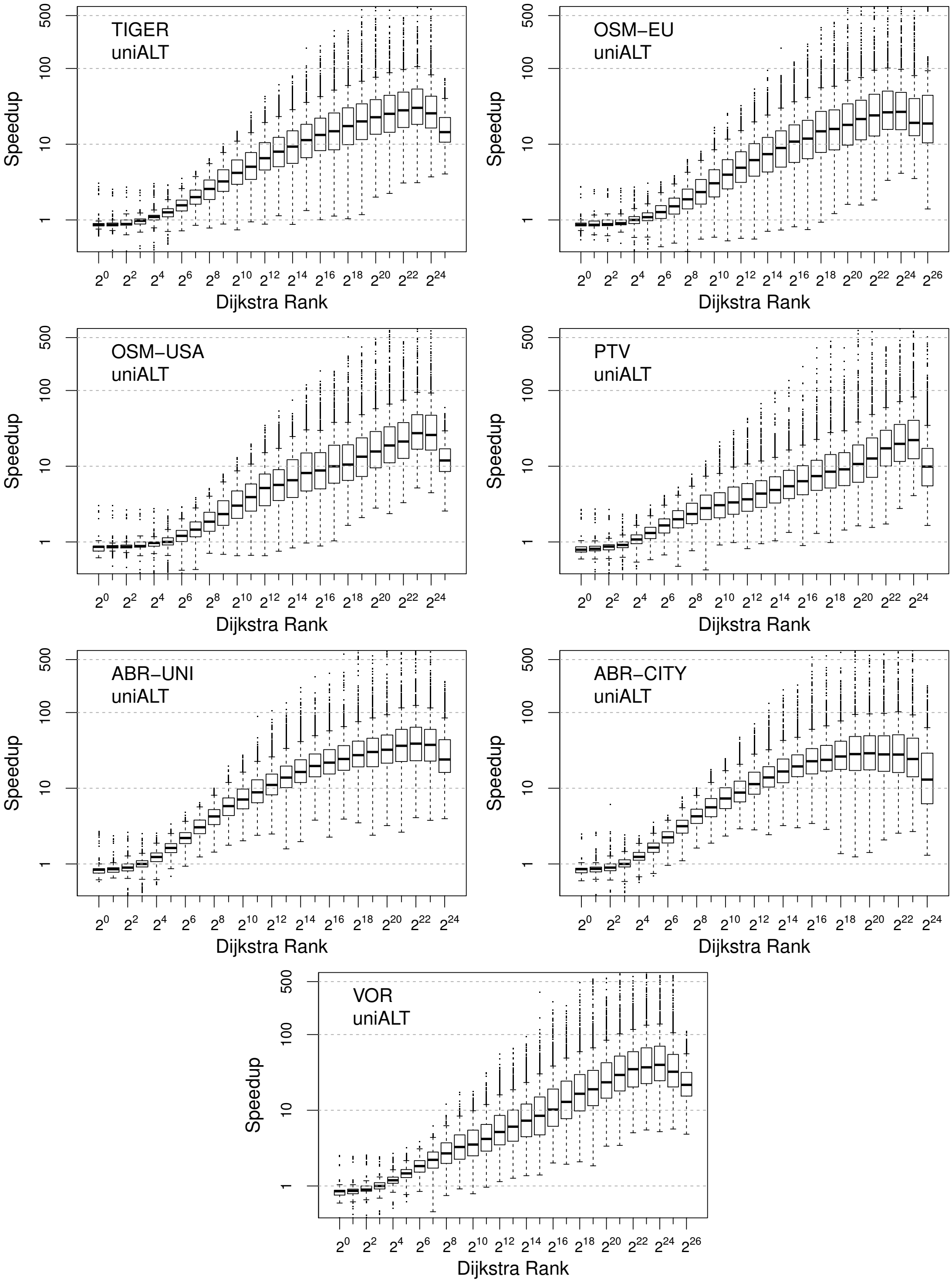


Figure 4.16: Boxplots of the speedups achieved by the technique unidirectional ALT (uni ALT) against Dijkstra rank. For each instance of the benchmark set 1 000 randomly selected starting nodes were selected, from which a query to a randomly selected node of increasing Dijkstra rank was performed.

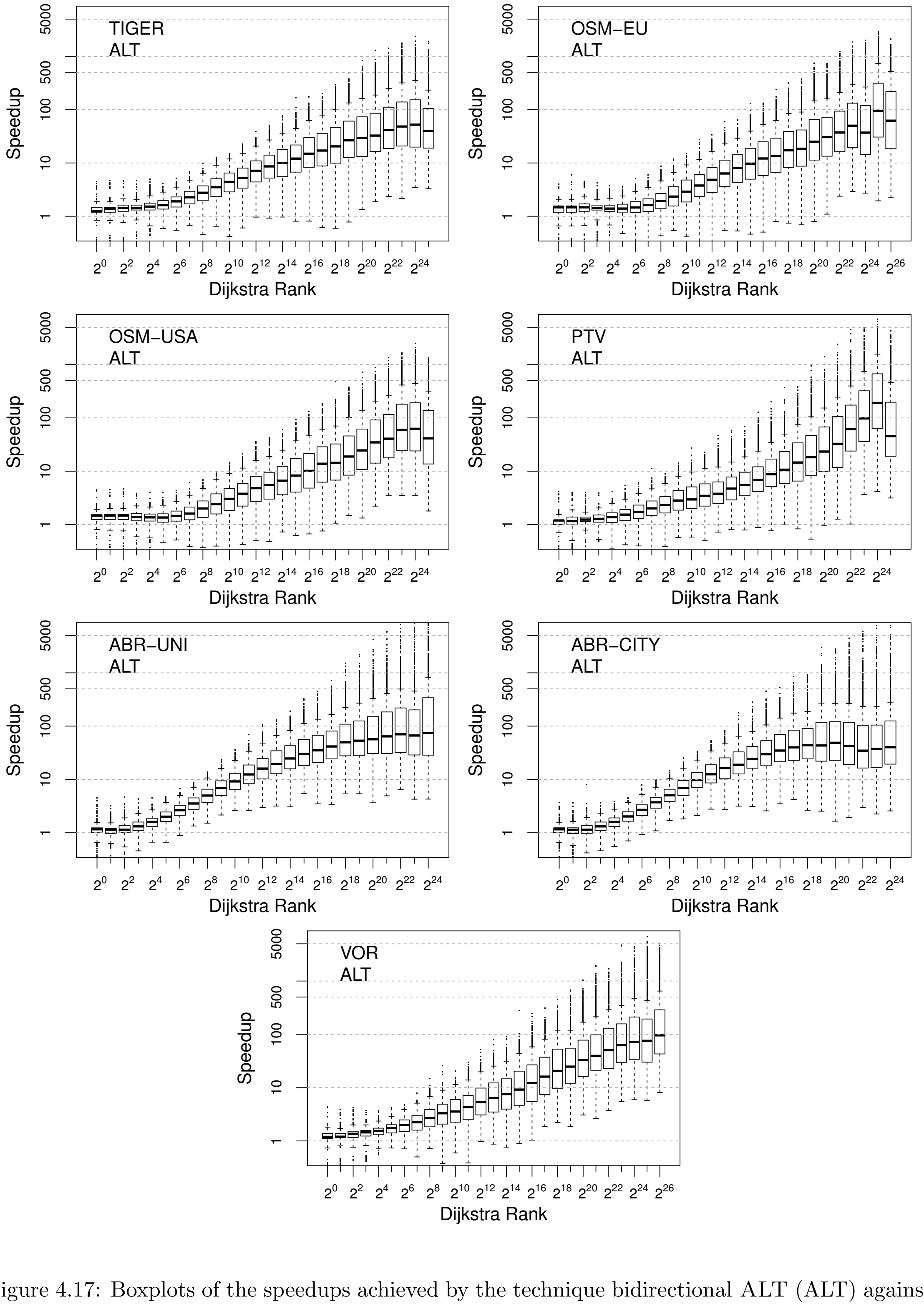


Figure 4.17: Boxplots of the speedups achieved by the technique bidirectional ALT (ALT) against Dijkstra rank. For each instance of the benchmark set 1 000 randomly selected starting nodes were selected, from which a query to a randomly selected node of increasing Dijkstra rank was performed.

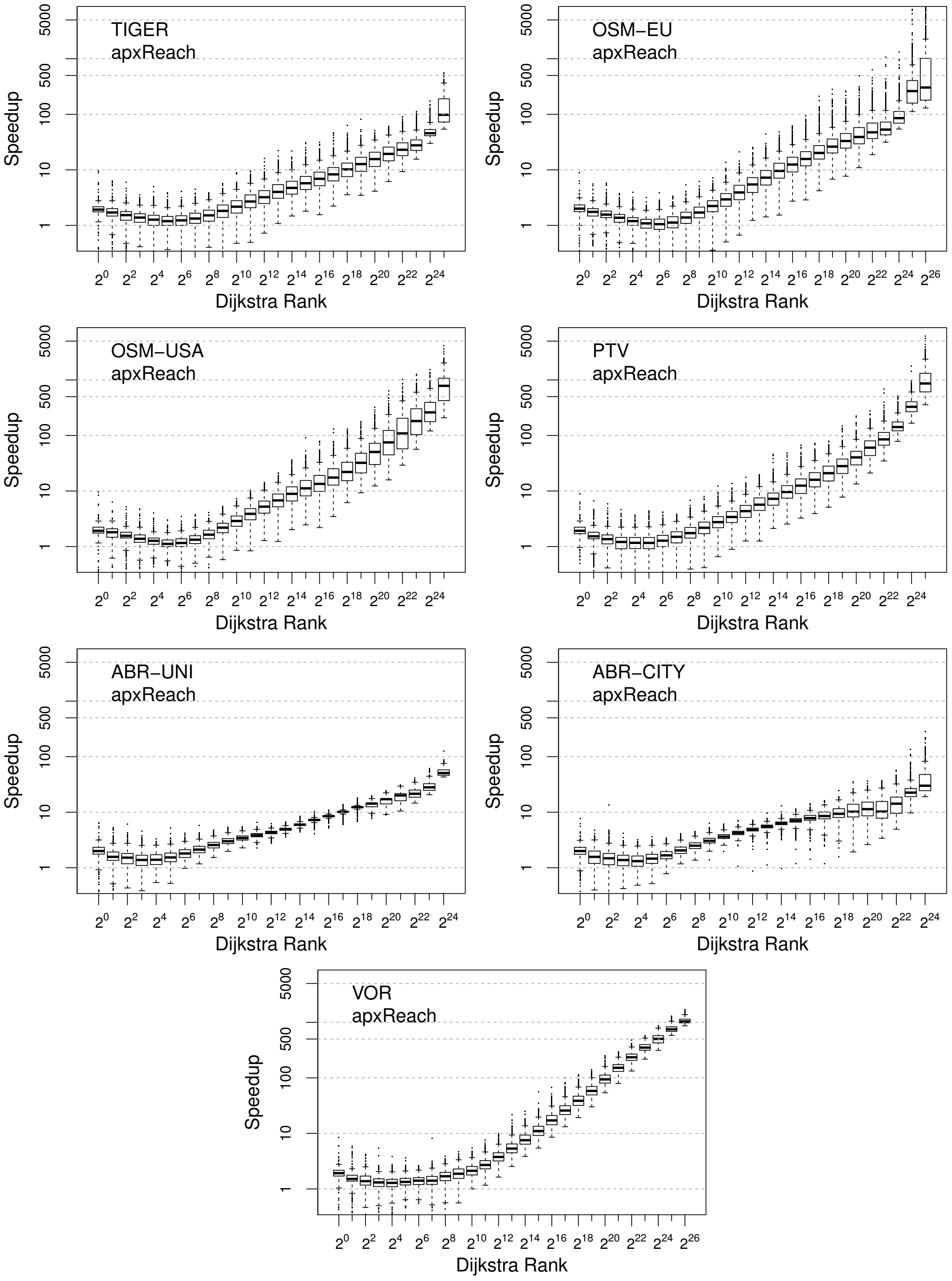


Figure 4.18: Boxplots of the speedups achieved by the technique approximate Reach-Based Routing (apxReach) against Dijkstra rank. For each instance of the benchmark set 1 000 randomly selected starting nodes were selected, from which a query to a randomly selected node of increasing Dijkstra rank was performed.

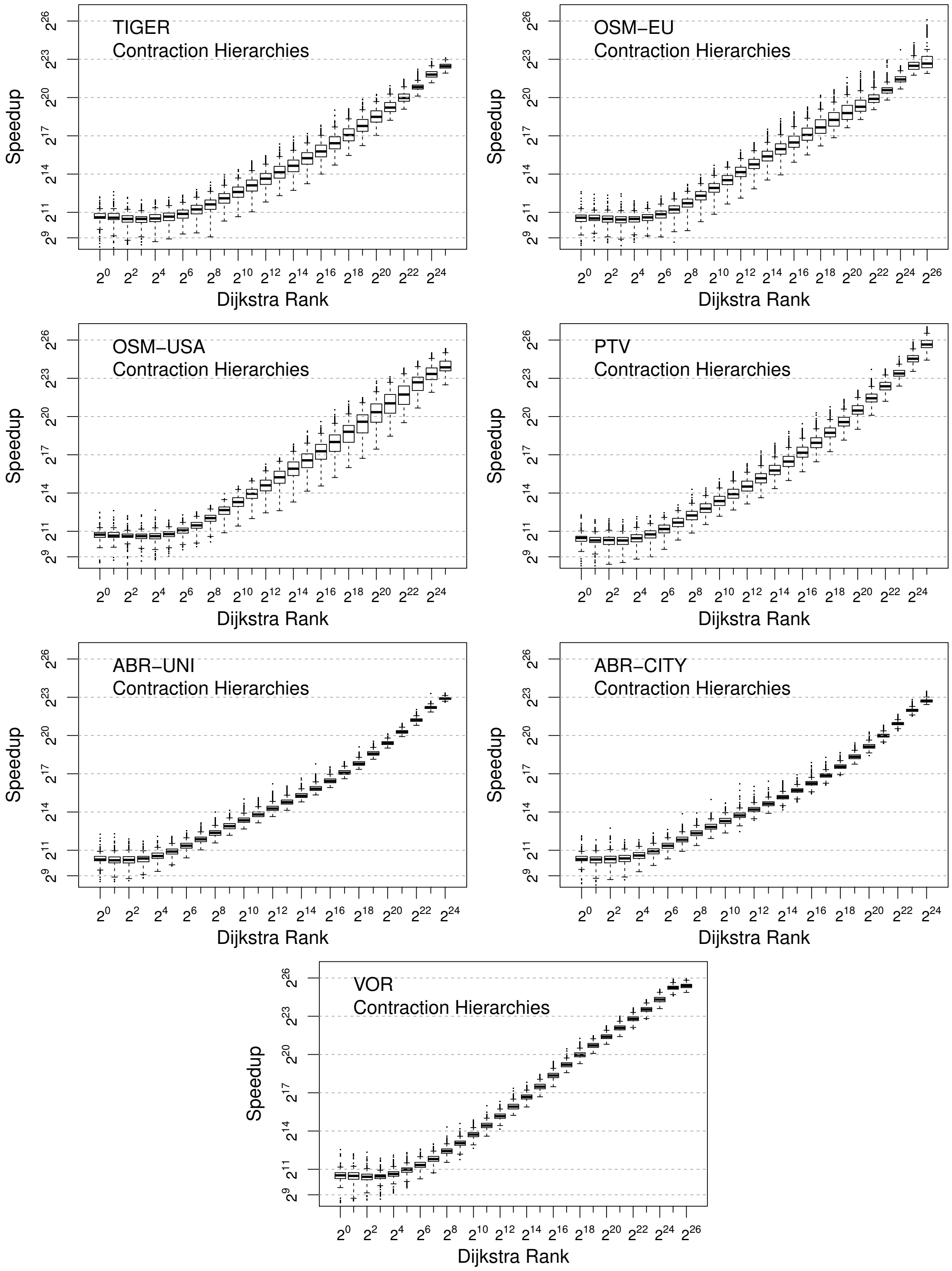


Figure 4.19: Boxplots of the speedups achieved by the technique Contraction Hierarchies against Dijkstra rank. For each instance of the benchmark set 1 000 randomly selected starting nodes were selected, from which a query to a randomly selected node of increasing Dijkstra rank was performed.

Chapter 5

Artificial Traffic Information

In the second part of this thesis, our general aim is to improve the situation of a missing broad base of instances, which experimenters can use to assess algorithms developed in the field of route planning. Having provided generators that are suitable for the static case in the previous chapter, we now focus on the time-dependent scenario. In particular, we develop algorithms capable of generating realistic, artificial, daily traffic information on top of road networks of continental size. In doing so, we enlarge the set of available instances, which are typically used to experimentally assess the quality of speedup techniques developed for time-dependent route planning. However, until now, the set of available time-dependent instances was rather limited. In particular, to the best of our knowledge, no real-world data set that contains daily traffic-flow information of a large road network is publicly available to researchers. On the other hand, the only existing generator is of limited use in that it involves undocumented procedures and seems not to be geared towards real-world instances. In our scenario, we rate artificial instances as realistic, if they admit similar properties compared to a real-world data set. In particular, artificial instances should exhibit similar local as well as global statistical properties. Additionally, shortest-path algorithms should behave similar on artificial and on real-world instances. In the development of our algorithms, we were guided by a confidential real-world data set we have access to. By its analysis, we derive a realistic model of how traffic flow emerges in large networks. Based on this model, we develop algorithms that utilize either road categories or coordinates to enrich a given road network with artificial traffic information. This chapter provides ready-to-use generators that allow for the generation of realistic daily traffic information on top of road networks that may originate from manifold sources like commercial, open source or artificial ones. This remedies the situation that no realistic data sets are publicly available and opens the door to assess and compare shortest-path algorithms in time-dependent road networks.

5.1 Introduction

In recent years the focus of accelerating shortest path queries switched from static to time-dependent scenarios; see Section 2.5.3 for a brief overview. One reason is that the time-dependent scenario is more realistic in the sense that the time needed to travel a certain distance depends highly on the traffic and thus changes significantly over the day. Yet, the speedups for route calculations gained in this field are not nearly as impressive as they are for static scenarios. Hence, for many research groups, this field is still very attractive, as they strive for similar results in time-dependent scenarios. To prove the applicability of new algorithms they are usually evaluated on large-scale, i. e., time-dependent road networks of continental size. Ideally, this is done using real-world data, as most of the algorithms are tailored for road networks. Unfortunately, to the best of our knowledge, no publicly available real-world data set of time-dependent road networks exists. The reason may be that observing traffic and maintaining statistics is an expensive and tedious task [Flö08]. Hence, companies do not share their valuable data. Thus, experimental evaluations are done using artificial data sets [Del11, NDLS08, BGNS10]. The test data used there is generated by an ad-hoc method for randomized delay assignment and it is not geared towards any properties of real-world data. Even worse, it turns out that the relative speedups observed thereon are significantly increased [NDLS08] compared to those found on real-world instances [Del09]. We aim at closing this gap by artificially generating realistic time-dependent road networks by exploiting properties of an underlying static road network.

Related Work In recent years, many publications on point-to-point route-planning techniques have appeared. Good overviews for the static [DSSW09a] as well as the time-dependent scenario [DW09] exist. Time-dependent data is usually modeled by a mapping of functions to edges that determine for each time of the day how fast one can traverse a certain distance in relation to the default time when no delay occurs. These functions are called *profiles*.

To our knowledge, the only known approach for artificially assigning time-dependent information on edges in a large scale road-network is to randomly assign delays on specific levels within a precomputed highway hierarchy of a road network [NDLS08]. This work does neither report on details nor provide a systematic analysis of the generated data. In a personal discussion with the authors it turned out that way too many profiles are generated, which have to be removed by an unreported process. The information on the shape of profiles used in the above work is taken from the research field of traffic and transportation prediction [Ker04]. The simulations used there are not applicable to large-scale road networks of continental-size, e. g., Europe, for two reasons. First, no data set exists that contains the travel behavior of all people living on a continent. Second, no algorithm is capable of simulating all individuals' behavior within a road-network of that size [BG10]. Part of this chapter has previously been published [MW11b].

Contribution This is the first work providing algorithms which allow researchers to enrich static road networks with time-dependency information and conduct more reasonable validations of speedup techniques for the time-dependent scenario. Our scenarios are systematic time dependencies in road networks that occur on a daily basis. We explicitly do not cover *dynamic* time dependencies, i. e., delays occurring because of unexpected events like accidents, holiday traffic or weather conditions.

The analysis of a confidential small time-dependent subset of the European road-network[1], namely Germany, gives general insights on the properties of delays. Taking these results into account, we derive a model which we take as a base for the further development of our algorithms.

In order to generate meaningful time-dependency, we require a road network to offer

[1] Provided by the company PTV (`http://www.ptv.de.`)

one of two kinds of additional data, road-category information attached to edges or coordinates of the nodes stored therein. Thus, our algorithms can be applied to graphs of manifold origin, e. g. commercial, open source or artificial. We use the required additional data to compute structural information, in particular, areas of urban type. Based on this auxiliary data, we find edge candidates for assigning time-dependency information. In an extensive computational study, we show that the generated data has statistical properties similar to those of the real-world road networks in a global as well as in a local scope. Regarding algorithmic properties, we find that the inherent shortest-path structure exhibits similar characteristics on artificial and on real-world data. Additionally, we show that speedup techniques applied to our artificial instances exhibit the same relative acceleration as on the real-world instances, which was not true for the ad-hoc generator. In doing so, we come to the following two conclusions. On the one hand, a shortest-path technique that performs well on our generated data sets is likely to behave similar on real-world data. On the other hand, our algorithms contain several degrees of freedom that allow for assessing the scalability of time-dependent shortest-paths algorithms with respect to both, running time and memory consumption.

Outline In Section 5.2, we establish the basis of our work by analyzing confidential time-dependent road-network data[1]. We present statistical data as well as representative extracts of the time-dependent data and derive a systematic model of how delays occur. Based on our observations, we first develop a generic five-phases approach in Section 5.3, which we then transfer into algorithms that generate time-dependency within static road networks according to both scenarios, where we either have coordinates of nodes or road-category information of the edges available. In doing so, we derive two algorithms for the scenario where we can rely on road categories, and one algorithm that exploits the geometric layout of the underlying graph. In Section 5.4, we first report on the parameter sets used to generate the benchmark set. Then, we compare the thereby generated instances and the available real-world data with respect to global and local statistical properties. It turns out that the parameter sets of all three algorithms can be tuned to approximate the statistical properties of the real-world data set accurately. We demonstrate the usefulness of our algorithms by comparing the shortest-path behavior on artificially generated instances with those on the real-world data set. In addition, we analyze the behavior of two speedup techniques on instances of the benchmark set and compare it with the outcomes on real-world instances. It turns out that the speedup techniques exhibit a similar behavior on all instances, which was not true for the ad-hoc generator. Hence, we finally conclude that our algorithms are well suited to generate realistic time-dependent instances in a way superior to what was possible before.

5.2 Analysis of Real-World Time-Dependent Data

The time-dependent road-network data set of Germany[1] we use is confidential and part of commercial products. On the technical side, the data set consists of two parts. The basis is a static network relying on data of the company NavTeq. The data is not fully free for scientific use, but has been provided to participants of the 9th Dimacs Implementation

[1] Provided by the company PTV (`http://www.ptv.de`.)

Table 5.1: Overview on the statistical properties of the time-dependent data set of Germany, whose static graph consists of ~ 11 million edges. For different days of the week, the number of distinct time-dependent profiles and the fraction of edges that is affected by these profiles is shown. The number of affected edges has been broken down into their respective road category.

	Mo	Tu-Th	Fr	Sa	Su
#Profiles	406	436	420	255	153
% affected edges	4.60%	4.73%	4.30%	2.67%	1.74%
Expressway	14.05%	14.24%	12.90%	7.43%	10.12%
Non-urban	60.52%	60.44%	60.38%	66.75%	63.49%
Urban	25.43%	25.32%	25.72%	25.82%	26.39%

Challenge [DGJ09]. We have access to a slightly updated version of the year 2006.

The second part consists of the time-dependency information. Delays are modeled by assigning profiles to edges. A widely applied solution is to use *piecewise linear functions* (PWLF) to model the delay of specific intervals of the day. Intermediate points are interpolated linearly. The real-world data set provides sets of PWLFs for different days of the week. A mapping assigns edges of the static data set that are affected by a delay to the corresponding PWLF. Each day is split up into 96 time intervals of 15 minutes each. Accordingly, a PWLF consists of 96 support points. Each point represents the delay of traversing this edge as a factor in relation to the mean speed usually assigned. For the remaining part of this chapter we refer to these PWLFs as *profiles*.

The data set contains *systematic* time-dependency profiles, i. e., it contains historical data of a larger, specific period of time that represents time-dependency information that is used for daily routing. It explicitly does not cover dynamic time-dependency information, which may be caused by unpredictable events like accidents, holiday traffic or weather conditions. The time-dependency data set of Germany covers only a fraction of the static graph. In Table 5.1, we give a brief overview on statistical information of the data set. The data covers different days of a week, and shows for each the fraction of edges that have a time-dependency profile assigned. Additionally, the number of affected edges is broken down into their assigned road categories. The first row of Table 5.1 shows that each of the daily time-dependent data sets consists of a few hundred profiles, each of which represents traveling times of hundreds of thousands of edges. Evidently, the data is already highly compressed. We push the compression idea to the limit to see whether a small set of representative profiles emerges that could assist us in the interpretation of traffic flows.

As already mentioned, each profile is represented by a PWLF having 96 supporting points. The idea is now, to find similar profiles using the *k-means* algorithm of Lloyd [Llo82], and to use the thereby computed centroids as representatives, recall Section 2.4.2. In order to cluster the profiles, we normalize the values to $[0, 1]$ and interpret them as points in $\mathbb{R}^{96}$. Since the outcome of k-means depends on the randomly chosen profiles in the initialization phase, the algorithm is restarted several times and the result with the lowest total distance is used. The results were stable over several executions of the k-means algorithm.

Figure 5.1 shows all profiles of the days Tuesday to Thursday on the top and the normalized, k-means compressed profiles with $k = 4$ on the bottom. As we can see in the top figure, the profiles can hardly be distinguished but seem to follow certain patterns.

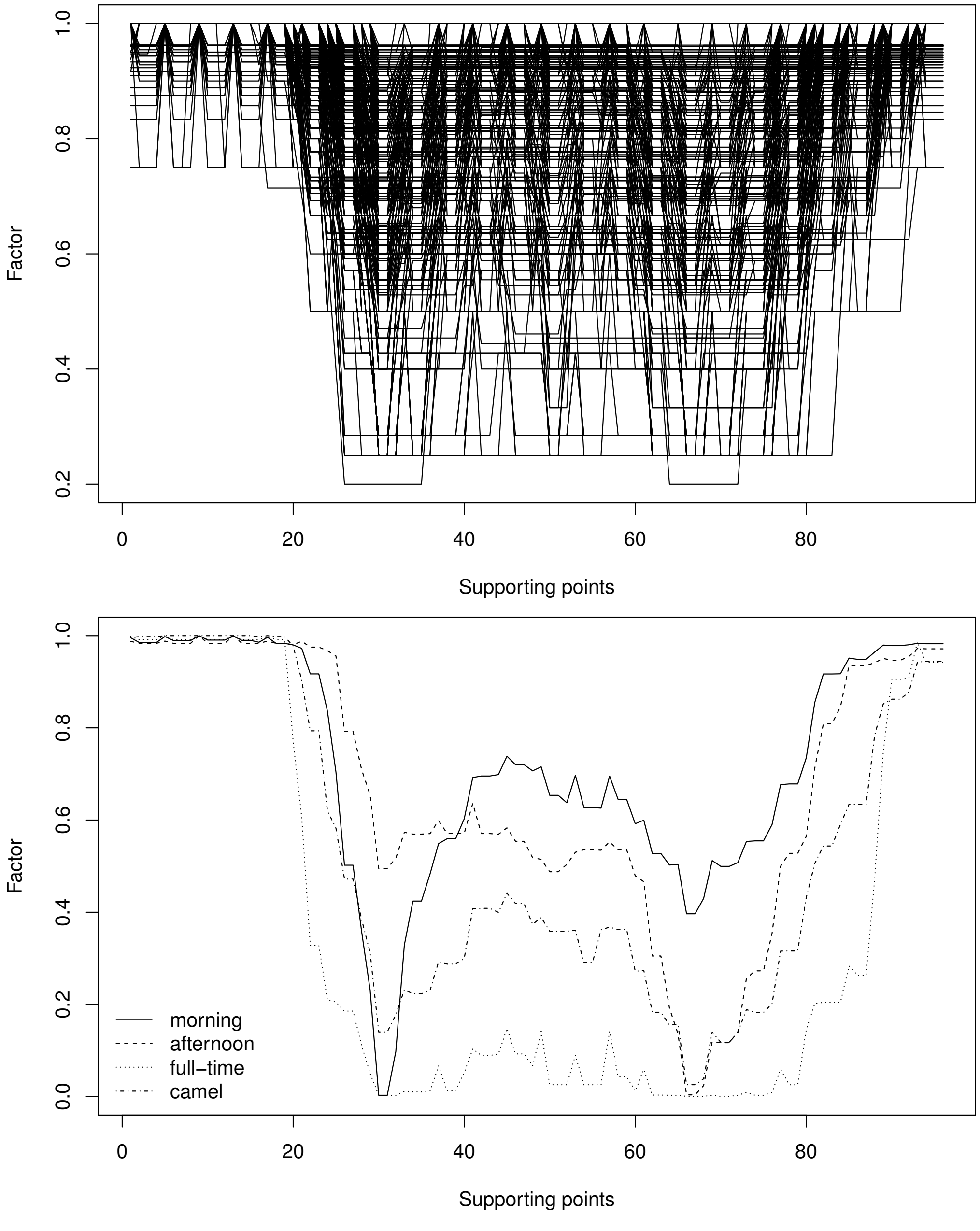


Figure 5.1: On the top, all profiles for the days Tuesday to Thursday of the PTV data set. On the bottom, the centroids of the first normalized ([0.0, 1.0]) and then compressed set of profiles for the days Tuesday to Thursday using the *k-means* algorithm with $k = 4$.

The bottom figure shows the four computed centroid profiles. By using larger values of k, functions are computed that exhibit a similar curve progression but differ by an offset. The following four types of profiles can be distinguished:

- **full-time**: the travel speed is constantly reduced on a large part of the day (FUL).
- **camel**: the travel speed is reduced in the morning and in the afternoon. In between the edge can be traversed faster (CAM).
- **morning**: the travel speed is mainly reduced in the morning. Probably it is reduced in the afternoon too, but not as strong as in the morning (MOR).
- **afternoon**: the travel speed is opposite to the case in the morning, thus, the main reduction is in the afternoon (AFT).

An explanation for the distinction between morning as well as afternoon profiles on the one side and camel profiles on the other side can be found in the structure of the underlying graph. To reduce space consumption, roads are typically modeled using undirected edges where possible, e. g., small roads that can be traversed in both directions at the same costs. However, in the case of roads whose directional lanes are separated or the costs differ, directed edges are used. Thus, segments of roads that would have a camel profile assigned have a morning profile assigned for the one direction and an afternoon profile for the other direction.

To see where these profiles may originate from, we take a further, more detailed look at the road-network infrastructure. We can observe that road-networks usually contain dense and sparse regions, with a change from one to the other by a rather smooth transition. Typically, these dense regions correlate to *urban* areas, whereas the sparse regions match *rural* areas. These urban areas seem to attract a certain region around them, where the attractiveness gets reduced with growing distance. We expect this attraction to originate from the available resources provided there, e g., services or work places, which let individuals travel from their homes to those resources and back. Now, we bring in the compressed profiles of the real-world road network and observe their statistical occurrence within the network's infrastructure. In particular, a large part of the full-time profiles are assigned to urban roads, which do not include residential roads. In the case of separated lanes, morning profiles are assigned to roads leading into urban regions whereas afternoon profiles are assigned to roads leaving urban regions. In the case that roads are in between two attractive regions or are not modeled by separated lanes, they have camel profiles assigned.

In Figure 5.2, we show the compressed time-dependent data of an excerpt of the real-world data set, which represents Munich. We can see that most of the full-time profiles (yellow) are assigned to parts of urban centers. The other profiles mainly occur on the radial roads leaving these centers and on the roads that interconnect them. On many of the streets that have separated lanes, we can observe that the one direction has morning profiles (green) assigned and that on the opposite direction afternoon profiles (blue) occur. On the most heavily used road segments, we find the camel profiles (red) attached. This supports our observations on the distribution of profiles to edge categories from above.

Admittedly, the matching from a road category of road segments to its assigned profile is simplified and varies from region to region. However, the scenario is highly complicated,

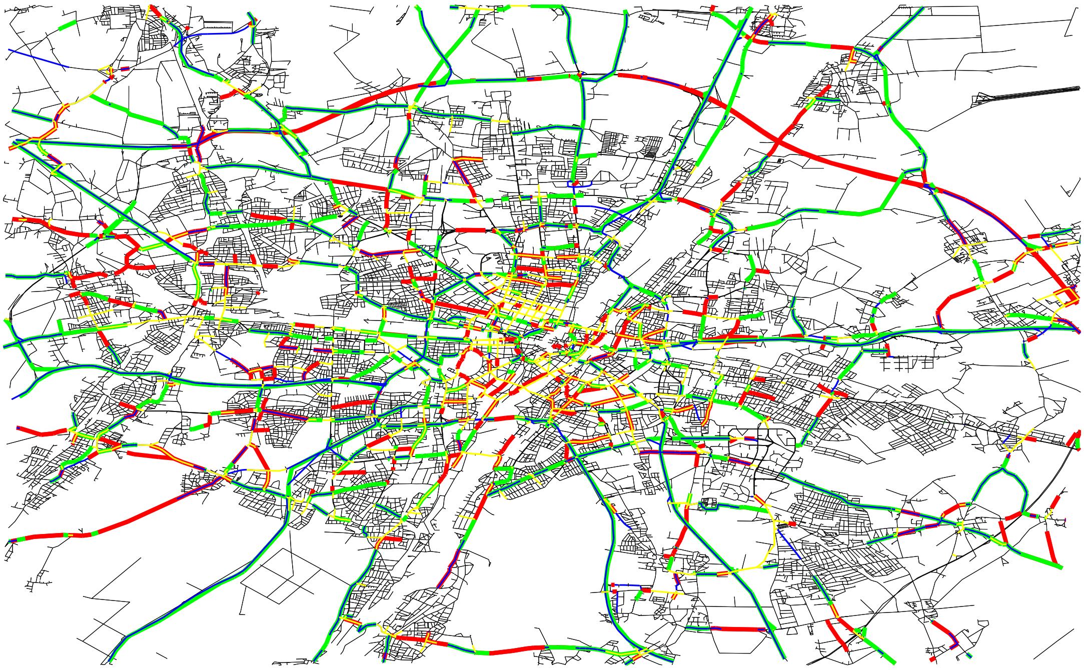


Figure 5.2: Excerpt of the compressed time-dependent real-world data set of PTV, which shows the region around the city of Munich. Road segments are represented by edges, which are colored according to the profile assigned to them (camel: red, morning: green, afternoon: blue, full-time: yellow, unaffected: black).

which makes a generalization necessary to gain a comprehensible model. In addition, we should not stick too close to the real-world data as it contains considerable amount of uncertainty. First, except the technical specification of the data format, the data set is rather undocumented. For example, the data collection process itself is unreported, and it is unknown whether the data has been altered before we got access to it. Moreover, the time-dependency information is dated to the year 2006, and hence possibly outdated. Finally, some information is lost as the data set seems to have been compressed prior to our usage. Obviously this is now definite as we pushed the compression idea to the limit to see whether distinct profile types emerge. Nevertheless, we are guided by this data in our observations and see the necessity for a generator strengthened that incorporates degrees of freedom to broaden the set of benchmark instances.

Summary By the analysis of the network's infrastructure, global statistics and a visual example, we made three observations that are now condensed in the following model to create time-dependency data on top of static road networks.

(1) Delays originate from traffic between urban regions and a certain area of influence. This originates from the fact that urban regions contain many resources, e. g., services or places of work, that attract individuals.

(2) The profiles assigned to edges are different, based on their location within, leading into or leading out of town. This models the behavior of commuters according to their location within the network.

(3) Profiles have a similar curve progression and are therefore well compressible. That way, we can store a few representative profile types $P = \{\textsc{ful}, \textsc{cam}, \textsc{mor}, \textsc{aft}\}$ and use them for the assignment. This is due to the fact that most of the traffic stems from daily routine.

A fourth observation can be made looking at the modeling process. Usually, roads have bends and crossings. Thus, they are often modeled having many edges and nodes. On the one hand, it is important to know which edges are affected by time-dependency in general. On the other hand, it is unlikely that profiles of adjacent nodes differ significantly.

(4) Similar profiles should be assigned to paths of adjacent nodes.

This completes our model that is based on observations of structural delays and their possible origin in a real-world road network. In the next section, we derive algorithms that compute artificial delays on top of static road networks according to this model.

5.3 Algorithms for Assigning Profiles

As already stated, real-world road networks are hard to obtain. Nevertheless, some sources for *static* scenarios are available, e. g., commercial data of PTV AG [DGJ09], artificially generated data, compare Chapter 4, or open source data like the OpenStreetMap project[2]. Usually, these data sets not only contain nodes, edges and travel times of the underlying road network, but contain additional information. We exploit this information to locate urban regions within a given road network.

Real-world data sets often contain *road categories* that roughly correspond to common street categories, e. g., urban, express- or motorway, but are much more detailed; see Section 4.2 on page 72 for an example. Each edge $e \in E$ has a single road category from the set of all road categories ζ assigned by a function $cat : E \longrightarrow \zeta$. In some cases, the data sets may not contain road-category information in high detail, the data is flawed or is completely absent, e. g., when dealing with artificial road-networks. However, in many cases these data sets have in common that they rely on an embedding in the plane and thus contain *coordinates* for nodes. Each of the vertices $v \in V$ has coordinates assigned by a function $p : V \longrightarrow \mathbb{R}^2$. In the following, we develop algorithms that exploit these pieces of supplemental information on the underlying network.

5.3.1 General Five-Phases Approach

As described above, we require the network to contain one of both, road-categories and coordinates data. This additional information is used to compute structural information, which is then exploited to locate edges that are likely to be affected by time dependency. Note that we do not assign a profile to an edge immediately but rather attach profile types and their quantity to delayed edges. This allows edges to be affected by different types of categories, e. g., if a road can be traversed in both directions it may get a morning as well as an evening profile type assigned. Incorporating the model derived from our observations in Section 5.2, we develop the following approach that works in five stages.

[2] `http://www.openstreetmap.org/`

Preprocessing Phase First, in the *preprocessing phase* structural data is computed to split the set of nodes V of a road network into two disjoint sets. Namely, *urban nodes* $U \subseteq V$ and *rural nodes* $R \subseteq V$. *Boundary nodes* B are defined by the subset of rural nodes that are adjacent to at least one urban node, i. e., $B := \{v \in R \mid \exists(v,u) \in E, u \in U\}$. The following phases rely on this separation for profile type assignment. In some cases, the urban regions can be classified to form a set of disjoint towns.

Urban Phase Next, the *urban phase* starts. We assume that people who live within an urban region will travel mostly within an urban region. Thus, edges of paths that are used often will get a full-time profile type assigned.

Rural Phase In the *rural phase*, we determine the edges that are likely to have a morning or evening profile assigned. The basic idea is that commuters drive from rural into urban regions in the morning and back in the evening. This is not done arbitrarily. Each urban region has a surrounding region of influence denoted by *urban catchment*.

Filtering Phase In the case that too many edges are affected by profile types, these can be filtered to fit statistical properties, e. g., the profile distribution of the commercial data. We follow a simple rule: If less than a filter limit F profiles types are assigned to an edge, the edge is considered to be not affected by any delay.

Postprocessing Phase So far profile types $p \in P$ have been assigned to edges. In a more realistic scenario, these could now be created with random offsets or overlaid with each other. To preserve comparability we will omit this randomization part. Instead, each edge is examined and the profile type occurrences will determine the resulting profile type of the edge using the following mapping, where *!* indicates no occurrence:

1. $(AFT \wedge !MOR) \longrightarrow AFT$;
2. $(!AFT \wedge MOR) \longrightarrow MOR$;
3. $(AFT \wedge MOR) \longrightarrow CAM$;
4. $(FUL \wedge !AFT \wedge !MOR) \longrightarrow FUL$.

Note that other goals of this phase could be to make the transitions from exterior profiles to interior profiles smooth by interpolation, e. g., where the boundary nodes are crossed. Furthermore, it may be interesting to overlay profiles on affected edges, e. g., to reduce harsh changes in the profiles. Problems arising when dealing with both of these problems are not covered here.

In the following, we present three heuristics based on this approach to find edges that are likely to be affected by time-dependent profile types without having any traffic information available. The phases of each algorithm are described in detail. Note that all of the algorithms use the filtering and postprocessing phase as described above to ensure comparability.

5.3.2 Algorithm I: Affected-By-Category

Our algorithm Affected-By-Category relies solely on road-category information to identify edges that are likely to be affected by delays. We make two assumptions. First, exploiting road categories leads to a good classification of the nodes into urban and rural

Algorithm 5.1: Reduce-Fragmentation

```
Input: $G(V, E)$, hop limit $hLim$, neighbor ratio $nRatio$, connected components
       information $CC$
Output: Towns $T$
$U \longleftarrow \emptyset$;
foreach $v \in V$ do
    neighbors $\longleftarrow$ NeighborhoodSearch(v, hLim);
    $maxfrac_i(v) \longleftarrow$ largest fraction of neighbors in same CC $i$;
    if $size(maxfrac_i(v)) \geq nRatio$ then
        $U \longleftarrow U \cup pair(v, i)$;
foreach $u \in U$ do  $CC(u) = i$;
```

regions. Second, most people use faster roads to travel between rural and urban regions. Thus, delays mainly occur on those road categories that allow for a fast traveling.

Preprocessing Phase We locate *urban regions* within the network by assuming that they are connected by edges e of a certain road category, i. e., $cat(e) \in \zeta_{urban}$, where $\zeta_{urban} \subseteq \zeta$ denotes a set of road categories that is considered to be urban. Additionally, the length of those edges e that are of urban category must not exceed a maximal length $len(e) \leq mDist$. The urban regions are defined by the connected components of the subgraph containing only these edges. They are computed by a modified *breadth-first search* algorithm (BFS), which follows only the constrained edges. In addition, we maintain a *union-find data structure* that keeps track of the nodes' belonging. Using this data structure, we can quickly determine the corresponding towns and check whether a node has already been visited. The urban regions of a road network identified this way are candidates for being towns.

In reality a town is not only connected by slow urban roads. Often faster roads are built to speed up travel times within, out of and into towns. In the above described first step of the preprocessing phase, our algorithm omits these roads as otherwise, by following them, only few quite big towns are found. We reduce the generated fragmentation in the next step. The pseudo-code of the Reduce-Fragmentation procedure is listed in Algorithm 5.1. Again, we use a local neighborhood search (BFS) that is limited to a specific number of hops *hLim* to visit nearby nodes and look for their town-candidate membership. If for a node v the number of neighbors belonging to the same town candidate exceeds a specific ratio *nRatio*, we consider node v to be part of that town candidate too. We store the update information, but defer the update itself for a later batch update to prevent towns from growing arbitrarily large due to cascading.

After the fragmentation has been reduced, we compute the size of each town candidate. If the size exceeds a limit *tThresh*, we consider it to be a town t. Additionally, we compute and store the set of boundary nodes by checking whether nodes of a town t are adjacent to nodes outside that town. We end up with a set of towns T, which are used in the consecutive steps of this algorithm.

Urban Phase In the urban phase, we expect people who live in urban regions to travel mainly within that area. This is done all over the day and thus, full-time profiles have to be assigned to edges. To find intensely used roads inside a town, we perform shortest path queries between all boundary nodes B of each town $t \in T$ using Dijkstra's algorithm, and assign full-time profiles to the edges of the shortest paths found.

Rural Phase The regional influence of a town $t \in T$ is the basis of the rural phase. We expect that each town attracts people from a rural region around it. From this rural area, commuters will drive in the morning into and in the evening out-of the town. During their travel they have to pass at least one boundary node of the town. We assume that the regional attraction of a town decreases with the distance people have to travel to reach the town, and additionally, we expect travelers to use faster roads to quickly reach their destination. Thus, the idea is to push a certain amount of delay along roads of a specific category starting from the boundary nodes of each town. In each step the delay decays until the process stops.

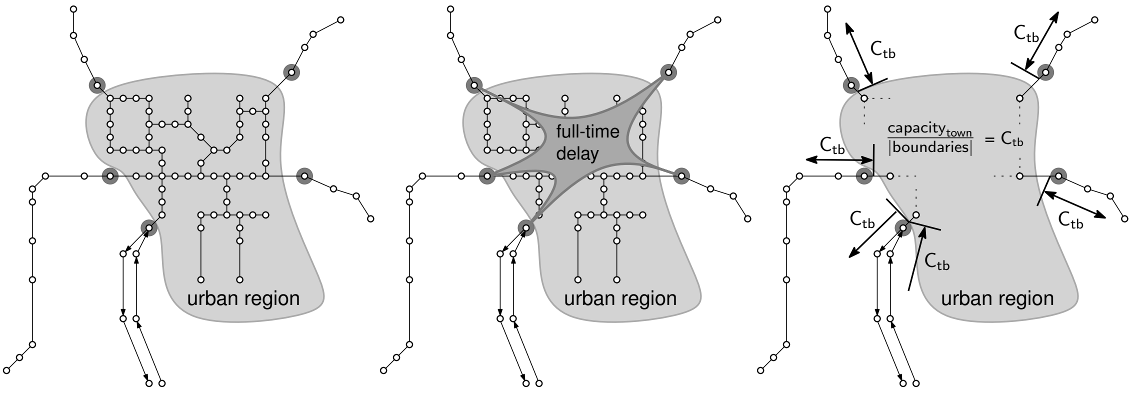
The set of delay categories ζ_{delay} used for pushing delays is a subset of non-urban road categories, i. e., $\zeta_{delay} \subseteq \zeta$ and $\zeta_{delay} \cap \zeta_{urban} = \emptyset$. In our generation process, we used all road categories that have a faster traveling speed than the urban road categories.

First, the so called *capacity* $C_t = 10 \cdot \mathit{size}(t)$ of each town is computed, which models the amount of people traveling from and into town. Then, for each town t, the subroutine DampeningBFS[Morning/Afternoon] (t, ζ_{delay}, C_t, *cDamp*, *rDamp*) is executed. It first sets the interior nodes of the town to be visited to not accidentally run into the interior part. Then, the capacity C_t is equally distributed among the boundary nodes B of t. Depending on the subroutine's type (morning/afternoon), DampeningBFS follows edges (incoming/outgoing) of a given category ζ_{delay} and assigns profile types to the visited edges (morning/afternoon). Every assignment of a profile type to an edge uses up a constant factor $\mathit{cDamp} = 1.0\%$ of the remaining capacity as well as a dynamic part $\mathit{rDamp} = 0.5\% \cdot \mathit{edgeLength}$. The subroutine ends when the capacity is depleted or falls below the threshold $\mathit{cLim} = 100$. In our experiments, we used the constants given above.

Summary In Figure 5.3, we depict the important phases of the algorithm Affected-By-Category. In the preprocessing phase, we find connected components that we assume to correspond to the urban region of towns if they exceed a certain threshold. For each town hereby found, we additionally compute the boundary nodes not part of this town. In Figure 5.3a, we show an example of an urban region with marked boundary nodes computed by the preprocessing phase. The goal of the urban phase is to find delays within towns. To this end, we compute all-pair shortest paths between the boundary nodes of a town, and assign full-time profiles along the shortest paths found; see Figure 5.3b. In the rural phase, we realize the travel behavior from an urban catchment of a town towards it and back by first computing a capacity for a town and equally distributing this capacity to its boundary nodes; see Figure 5.3c. Starting at these boundary nodes, we explore incoming (outgoing) category constrained edges and assign a morning (afternoon) profile to each edge encountered, which uses up some capacity. The exploration stops when the capacity is depleted.

The advantage of the algorithm Affected-By-Category is that individual capacities and categories can be chosen. Thus, it is possible to model diverse behavior, e. g., short, mid or long distance commuters.

This approach solely relies on structural information of the underlying network, which involves the following drawbacks. Typically, roads do not have the same category from



(a) result of preprocessing phase (b) urban phase (c) rural phase

Figure 5.3: Phases of the algorithm Affected-By-Category. On the left, an urban region with marked boundary nodes after the preprocessing phase that finds components connected by road-category constrained edges. In the urban phase, all-pairs shortest-path computation between the boundary nodes to find edges delayed with a full-time profile. In the rural phase, starting at each boundary node a certain capacity C_{tb} is pushed away from the town on incoming (outgoing) edges that get morning (afternoon) profiles assigned, where each assignment consumes a specific amount of capacity.

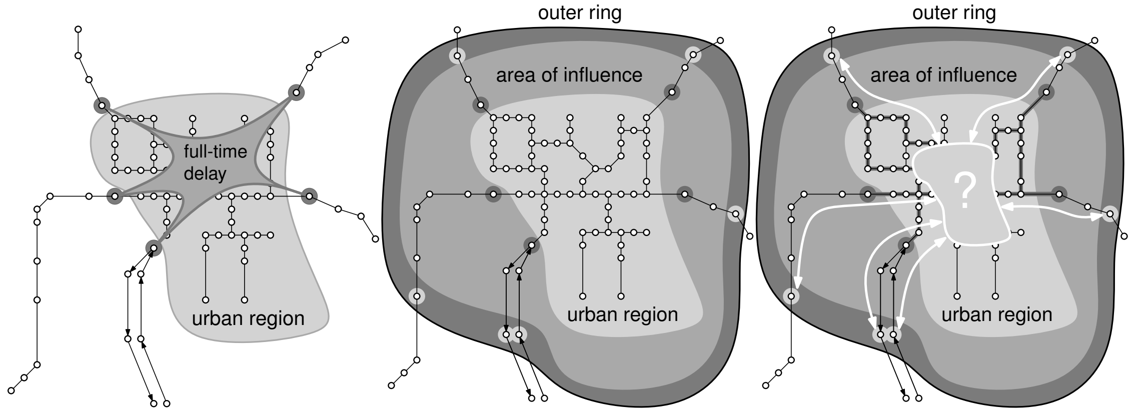
their start to their end. Thus, the algorithm might not be able to use up a chosen capacity. Additionally, a harsh changeover of the assigned profiles at the boundary nodes of towns can be observed. We tackle both drawbacks in the following algorithm.

5.3.3 Algorithm II: Affected-By-Region

To overcome the limits of Affected-By-Category, we propose the alternative algorithm Affected-By-Region that involves random shortest path queries between the urban catchment of a town and its interior. In particular, we also rely on road-category information but propose an alternative approach for the rural phase, which assigns delays without explicitly considering road-categories. Hence, the preprocessing and urban phase are similar to Affected-By-Category presented in Section 5.3.2, and only the rural phase differs. The initial situation and the rural phase of the algorithm Affected-By-Region are depicted in Figure 5.4.

Rural Phase The rural phase is split into two steps. In the first step, we identify an *area of influence* around a town that we expect to be its corresponding urban catchment. The size of this area ℓ around a town t is specified by the parameter *urban catchment ratio UCR* in relation to a town's size, i. e., $\ell = \mathit{UCR} \cdot \mathit{size}(t)$. Using a breadth-first search that is initialized with the boundary nodes of the urban region, we find ℓ nodes around it that form the area of influence. We let this local exploration run a little bit longer, until we find the neighbors of the nodes located in the area of influence, which form the *outer ring*. Figure 5.4b shows a possible area of influence and an outer ring after step 1 of the rural phase is completed.

In the second step, we again assume that commuters travel from the area of influence of a town into the town in the morning and back in the afternoon. To model this behavior,

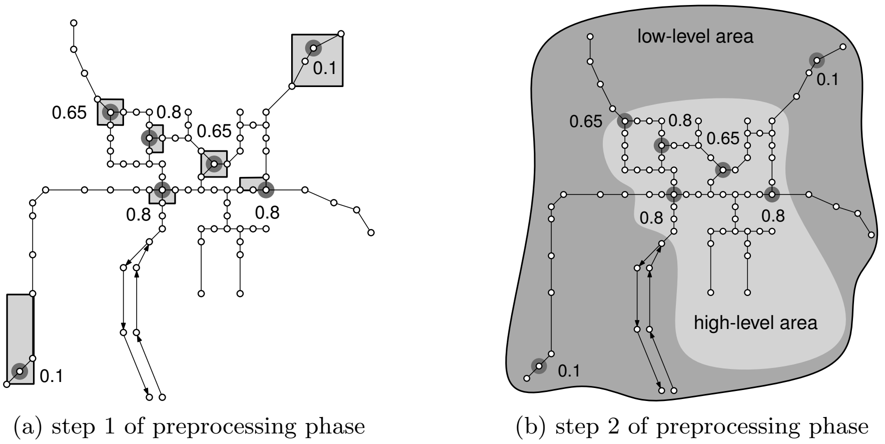


(a) urban phase (b) step 1 of rural phase (c) step 2 of rural phase

Figure 5.4: Phases of the algorithm Affected-By-Region. On the left, result of the preprocessing and the urban phase, which is similar to Affected-By-Category. In step 1 of the rural phase, computation of the area of influence around the town and an outer ring, containing the neighbors outside the area of influence. In step 2 of the rural phase, on shortest paths from a fraction of outer ring nodes to a set of random destinations within the urban area morning profiles are assigned, and afternoon profiles are assigned for the way back, respectively.

we perform Dijkstra queries from a fraction RQF of randomly selected nodes of the *outer ring* to randomly selected urban nodes of the town and back; see Figure 5.4c. Along these shortest paths, we assign morning profiles if they lead into town and afternoon profiles if they lead back, respectively.

Summary The algorithm Affected-By-Category had two main drawbacks, namely, a harsh changeover of the profiles at the boundary of an urban region and its restriction to certain road categories for assigning morning and afternoon profiles. Both drawbacks have their origin in the fact that solely structural information of the underlying network is used. To overcome this situation, we propose an alternative rural phase, which is incorporated in the Affected-By-Region algorithm. Similarly to Affected-By-Category, urban regions are computed, which we associate with towns. Additionally, within these urban regions, full-time delays are assigned on shortest paths between the boundary nodes. In the alternative rural phase, we explicitly compute the area of influence around a town and perform shortest-path queries from outside this region into the urban part of a town and back. In doing so, we find road segments that are likely to be heavily used and assign morning profiles on paths leading into the town, and assign afternoon profiles on paths that lead out of town, respectively. In doing so, we overcome the limitations of Affected-By-Category, and additionally, cover much better behavior of individuals in a network. However, the Affected-By-Region algorithm does not model long-distance commuters' behavior as it relies on local subroutines. Nevertheless, this can easily be overcome by computing random town-to-town shortest paths.



(a) step 1 of preprocessing phase

(b) step 2 of preprocessing phase

Figure 5.5: Two-step preprocessing phase of Affected-By-Level. In step 1, computation of the level of each node, which is the normalized reciprocal value of the bounding-box area occupied by a number of neighbors. In step 2, the levels are sorted to split the set of nodes into high-level and low-level areas according to a ratio parameter.

5.3.4 Algorithm III: Affected-By-Level

A general limitation of the algorithms Affected-By-Category and Affected-By-Region is the required road-category information. This data is a major ingredient as it is used to compute urban regions, full-time delays therein, and in the case of Affected-By-Category, additionally, the edges affected by morning as well as afternoon profiles. However, road-category information may be faulty or in the case of artificially generated instances completely absent. Hence, we propose a third algorithm called Affected-By-Level that solely uses coordinate information of the underlying road network to compute artificial delays therein.

Our algorithm relies on the observation that towns are well connected and modeled in large detail. Thus, the bounding boxes of urban nodes of a limited local neighborhood search are expected to be small, whereas the bounding boxes of rural nodes are rather large. We use the normalized reciprocal value of the occupied area and call it *level* of the node, which is a function $\mathit{level}\colon V \longrightarrow [0.0, 1.0]$. Thus, a high level corresponds to a vertex in an urban region whereas a low value is likely to lie in a rural region.

Preprocessing Phase In the preprocessing phase we compute the level of each node of the road network to allow for a classification of the nodes into urban or rural ones in two steps; exemplified in Figure 5.5.

In the first step, we compute the level of each node. To this end, the local neighborhood of each node $v \in V$ is explored until a limited number *BBL* of *neighbors* is found. Afterwards, the bounding box of the neighbors is computed and its size stored, which we access by a function $\mathit{area}\colon V \longrightarrow \mathbb{R}_{+}$. Based on this, we compute the level of each node by $\mathit{level}(v) = 1/\mathit{area}(v)$, and finally, normalize the level values to $[0.0, 1.0]$; see Figure 5.5a.

In the second step, we split the set of nodes V into two distinct subsets $R \subseteq V$ and $U \subseteq V$, where R contains the rural nodes and U contains the urban nodes, respectively. Our algorithm requires a parameter *urbRatio* that specifies the ratio of how many nodes are classified into the urban set of nodes. To assign the data set to either U or R, we sort the data in non-decreasing order, and determine the *urban threshold* uT, which is the entry found on index $urbR \cdot |V|$ in the sorted data set. Next, for each node $v \in V$, $level(v) \leq uT$ is evaluated. If true, the node is assigned to the rural set R of nodes, otherwise to the urban set U of nodes; see Figure 5.5b. Note that in the preprocessing, we do not explicitly model towns, but allow for arbitrary agglomerations of resources.

Urban- / Rural Phase In the algorithm Affected-By-Level, we incorporate the urban and the rural phase in a single process in which nodes are randomly selected, and according to its level either a rural step or an urban step is performed; see Figure 5.6. In this phase, we follow the intuition that commuters in most cases travel locally depending on their level. In particular, commuters starting from low-level nodes choose as target destinations nodes that have a higher level. This way, we can assign morning profiles on the way towards the target and afternoon profiles on the way back to the starting nodes. The full-time profiles are assigned by commuters that travel from high-level nodes to nodes of a similar level. This realizes our model of the travel behavior of individuals. In particular, we proceed as follows; see Algorithm 5.2 for pseudo-code.

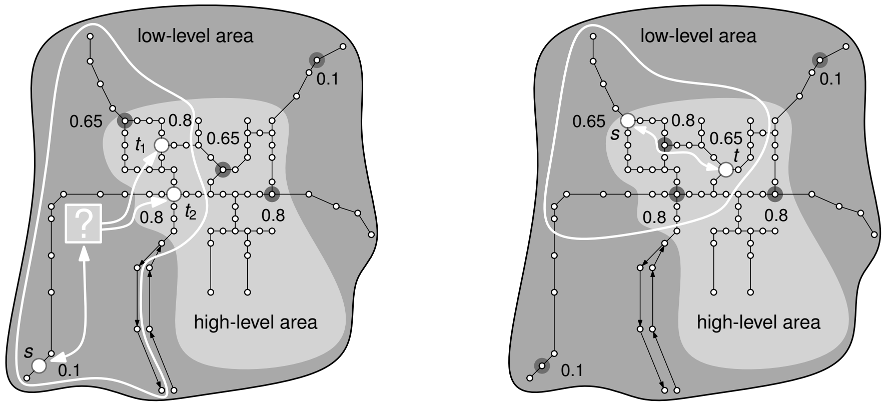
First, the fraction QF of local queries in relation to the number of nodes in the graph is determined. Then, a random node $u \in V$ is chosen until the necessary amount of queries to perform is exceeded. Starting from the node u, the subroutine LevelDijkstra(u, SNL) returns the shortest-path tree S after SNL nodes have been settled. Now depending on the level of the node either a rural or an urban step is performed.

If the node u is of rural type, a random target node v of high level is chosen from the nodes contained in S, and a morning profile type is attached to the edges of the shortest path from u to v. Afterwards, the edges of the shortest path from v to u get an afternoon profile assigned; see Figure 5.6a.

In the case that the level of node u is of urban type, a random node v of approximately equal level is chosen from the set of nodes in S. Afterwards, the edges of the shortest paths from u to v as well as from v to u get a full-time profile assigned; see Figure 5.6b.

Summary The algorithm Affected-By-Level solely depends on coordinate information of the underlying road network, which we exploited to incorporate the behavior of individuals for all delay types. Hence, the approach can be rated as all-purpose as it is independent of specific structural information, which has the following clear benefit. If a road network does contain faulty structural information, e. g., road categories, or in the case that they are completely absent, e. g., in the case of artificially generated data, this algorithm can still be applied. Exploiting the geometric layout of the network also has the benefit that we do not explicitly locate towns but find dense regions, which we assume to be of interest. This way, we do not rely on a specific threshold for a town's size, and hence, also create delays for areas smaller than the category-based algorithms allow for.

A drawback of this approach is that quite a large fraction of the road network has to be explored locally to achieve a representative profile assignment. As a result quite a lot of edges are affected, which have to be filtered out to fit statistical properties, e. g., those of the available real-world data set. Another possible drawback is the assumption on a highly detailed modeling of agglomerations of resources. Hence, we have to ensure that a network exhibits this feature, which is true for the real-world data sets as well as instances generated by the algorithms presented in Chapter 4.



(a) rural step (b) urban step

Figure 5.6: Combined urban and rural phase of Affected-By-Level. For a randomly selected node s a local shortest-path tree (marked in white) is computed. Depending on the level of node s either a rural step or an urban step is performed. Rural step: select a high-level node t from S, assign morning profiles to the shortest path from s to t, and assign afternoon profiles on the shortest path back. Urban step: select a node t of approximately similar level from S, assign full-time profiles along the shortest paths from s to t and from t to s.

Algorithm 5.2: Affected-By-Level

```
Input: $G(V,E)$, $\forall v \in V: level(v) \in [0,1]$, rural nodes $R$, urban nodes $U$, query
       fraction $QF$, settled nodes limit $SNL$
Output: $G(V,E)$, $e \in E: p(e) \in P$
individuals ⟵ $QF \cdot |V|$; //number of individuals
count ⟵ 0;
while count ≤ individuals do
    count ⟵ count + 1;
    u ⟵ randomNode(V); //draw a random node in the network
    S ⟵ LevelDijkstra(u, SNL); //shortest path tree of first SNL nodes from u;
    if $u \in R$ then
        v ⟵ select random high level node from S;
        Dijkstra(u, v, MOR); //assign morning profiles
        Dijkstra(v, u, AFT); //assign afternoon profiles
    else
        v ⟵ select random node with similar level from S;
        Dijkstra(u, v, FUL); //assign full-time profiles
        Dijkstra(v, u, FUL); //assign full-time profiles
```

5.3.5 Summary

In this section, we presented a generic five-phases approach, which guided our development of three algorithms that allow for assigning artificial delays on top of static road networks, namely Affected-By-Category, Affected-By-Region and Affected-By-Level. The generic approach allows for incorporating all aspects of our model, which briefly described are that people travel locally, that towns attract individuals from a certain area around it, and that delays originate from daily routine. In general, we compute structural information that splits the network into a rural and an urban part. To compute this auxiliary data, we require the network to contain one of both, road categories or coordinates, and first, presented algorithms that rely on road-category information.

The algorithms Affected-By-Category and Affected-By-Region exploit road-category information to compute urban regions and are identical except the rural phase. Within an urban region full-time delays are assigned to heavily used roads found by all-pair shortest-path computations between the boundary nodes of the urban region. Starting at the boundary nodes of an urban region, the algorithm Affected-By-Category assigns morning (afternoon) profiles along category-constrained road segments leading into (out of) this region. In contrast, the algorithms Affected-By-Region explicitly computes the region of influence around a town and assigns morning profiles along shortest paths from the exterior part of this region of influence into the urban region and evening profiles on the way back.

The algorithm Affected-By-Level solely uses coordinate information of the underlying network. It determines the level of a node v by computing the bounding box of a limited number of neighbors and assigning the normalized reciprocal value of the size of the bounding box to v. We assume that a node having a high level value is likely to be in an urban region, whereas a low level value corresponds a rural region. The algorithm Affected-By-Level randomly selects nodes in the network, and computes for each node u a shortest path tree S limited to a specific number of settled nodes. If node u is of high level, a randomly selected node v with similar level is selected in S, and full-time profiles assigned along the shortest path from u to v and back from v to u. On the other hand, if node u is of low level, a randomly selected node v of high level is chosen in S and a morning profile assigned on the shortest path from u to v and an afternoon profile on the shortest path from v to u.

In all three algorithms we do not directly assign a profile to the edges but count their occurrence. This allows us to determine less heavily used road segments and to possibly filter them out before profiles gets assigned. In addition, this allows for a postprocessing of these profiles, e. g., an overlay of profiles on an edge or between consecutive road segments. However, we omit this randomization part to preserve comparability between the algorithms. This is realized by using a fixed mapping in the case that several profile types occur on an edge.

In the domain of enriching a static road network with time-dependent data possible approaches are in between two far ends. On the one side, solely structural properties of a network are exploited, e. g., the inherent hierarchy. On the other side, the travel behavior of individuals is simulated in a network, which is typically done on different levels of details due to its computational complexity. In our development we started at the structural approach and incorporated more and more simulation aspects. In particular, the Affected-By-Category algorithm solely uses structural data in that it exploits road-category information of edges, which to a certain extent corresponds the inherent hierarchy of the network. Then, we incorporated some simulation aspects in the Affected-By-Region algorithm as we altered the rural phase to incorporate travel behavior of individuals from the urban catchment of a town towards its center and back. Finally, we left the area that uses structural information and developed the Affected-By-Level algorithm, which solely simulates traffic behavior according to the geometric embedding of the network. Having these algorithms at hand, we next assess their quality.

5.4 Experiments

In this section, we experimentally assess the usefulness of our proposed algorithms. First, we report on the parameter tuning done for each of our algorithms to generate instances that exhibit properties similar to those of the real-world instance. These 'realistic' instances are used as a benchmark set in our further analysis. In that context, we first compare their global statistical properties to those of the real-world instance. Afterwards, we examine the visual appearance of the generated data sets and check whether their intended behavior can be observed. Then, we compare the instances of the benchmark set with the real-world data set by means of statistical properties in a local scope. Finally, we analyze the shortest-path structure of the data sets.

Our implementations are written in C++ using the STL and no other additional library. The code was compiled with GCC 4.5 and optimization level 3. If not stated otherwise, the experiments were run using OpenSuse 11.3 on one core of an AMD Opteron 6172, clocked at 2.1 GHz, having 512KB L2 cache and 256GB RAM at its disposal.

Input In our experiments, we used the road network of Germany that is provided by the company PTV AG. The graph consists of about 4.69 million nodes and 11.18 million edges. An additional data set contains time-dependency information, which refers to a subset of the affected edges of the graph. We further compressed the time-dependent data and substituted each profile by its representative profile, as identified in Section 5.2. To preserve comparability, this compressed profile set for mid-week days is also used for the artificially generated instances. We neglected the profile sets for the remaining days of the week for two reasons. First, the profile sets have almost no influence on the relative performance of speedup techniques [Del09], and second, we were mainly interested in the most congested days of the week, which serve as a worst case scenario. However, we feel certain that the parameters available in our algorithms allow for a tuning that approximates the other days.

Parameters The algorithms Affected-By-Category, Affected-By-Region and Affected-By-Level use as input a static road network and compute edges therein that are likely to be affected by delays. These algorithms can be fine-tuned by a set of parameters. To find a suitable parameter set for each algorithm that fits best the global statistical properties of the profile distribution of the real-world data set, we performed an extensive experimental study. In doing so, we derived for each algorithm an instance that we assigned to our benchmark set. In particular, the benchmark set contains the instance Category computed by Affected-By-Category, the instance Region computed

Table 5.2: Parameter sets used to compute the artificial time-dependency instances of the benchmark set on top of the static real-world road network of Germany. The parameter sets are named by abbreviations of the corresponding algorithm. Affected-By-Category and Affected-By-Region use parameters: maximal edge length *mdist*, town size threshold *tThresh*, hop limit of neighborhood search*hLim*, ratio of common neighborhood *nRatio*. Affected-By-Region additionally uses parameters: ratio of urban catchment *UCR* and fraction of region queries *RQF*. Affected-By-Level uses parameters: ratio of urban nodes *urbR*, bounding box limit *BBL*, fraction of random queries *QF*, settled nodes limit of search tree *SNL* and edge filer *F*.

data set	mDist	tThresh	hLim	nRatio	UCR	RQF	urbR	BBL	QF	SNL	F
Category	300	500	5	75%	-	-	-	-	-	-	0
Region	300	750	5	75%	3.0	5%	-	-	-	-	0
Level I	-	-	-	-	-	-	45%	200	20%	400	8
Level II	-	-	-	-	-	-	45%	200	40%	400	8

by Affected-By-Region and the instances Level I as well as Level II computed by Affected-By-Level. However, an exception from the similarity of global statistics is the instance Level II, which we additionally added as it best matches the outcomes of the time-dependent Dijkstra experiment, which we show later. The detailed parameter sets used to create these benchmark instances are listed in Table 5.2. Parameters that are not listed here are treated as constants and are specified in the corresponding algorithm in Section 5.3. When speaking of *generated data sets* we refer to the instances of this benchmark set. In our discussion, we use the following abbreviations for each of the data sets: PTV, Category (CAT), Region (REG), Level I (L1) and Level II (L2).

Global Statistical Properties Table 5.3 shows the distribution of the representative profile sets of the real-world data set as well as of the generated data sets. CAT best fits the properties of PTV but contains slightly more FUL profiles. REG contains only a third of FUL profiles compared to CAT and three times the amount of MOR and AFT profiles. An explanation is that REG works similarly to CAT but randomly selects nodes inside of towns instead of the towns boundary nodes. Thus, by the specification of our postprocessing step many FUL profiles are overridden with profiles of type MOR and AFT. A similar overriding behavior can be observed for L1. Noticeable is the difference of the occurrences of MOR and AFT profiles for L1 and L2, which originates from different shortest paths for nodes (u, v) and (v, u) in combination with the simple filtering rules applied. L2 does not aim at fitting best to PTV but we can see that the ratio of profile categories is preserved. Additionally, the running time of each of the generated data sets is given. Most of the time is consumed by performing point-to-point shortest path queries, especially the all-pairs shortest path computations in the context of Affected-By-Category and Affected-By-Region. This becomes apparent when we compute artificial delays on the continental road-network of Europe, which consists of about 31 million nodes and 72 million edges. Using the parameter sets of Table 5.2, the generation took the following times: Affected-By-Category — 30.4 hours, Affected-By-Region — 66.0 hours, Affected-By-Level on parameter set Level I — 1.8 hours and Affected-By-Level on parameter set Level II — 3.3 hours. Our algorithms can be accelerated using speed-up techniques where possible.

Table 5.3: Percentage of the profile types assigned to edges of the time-dependent instances of the benchmark set. Additionally, their creation time is given in minutes.

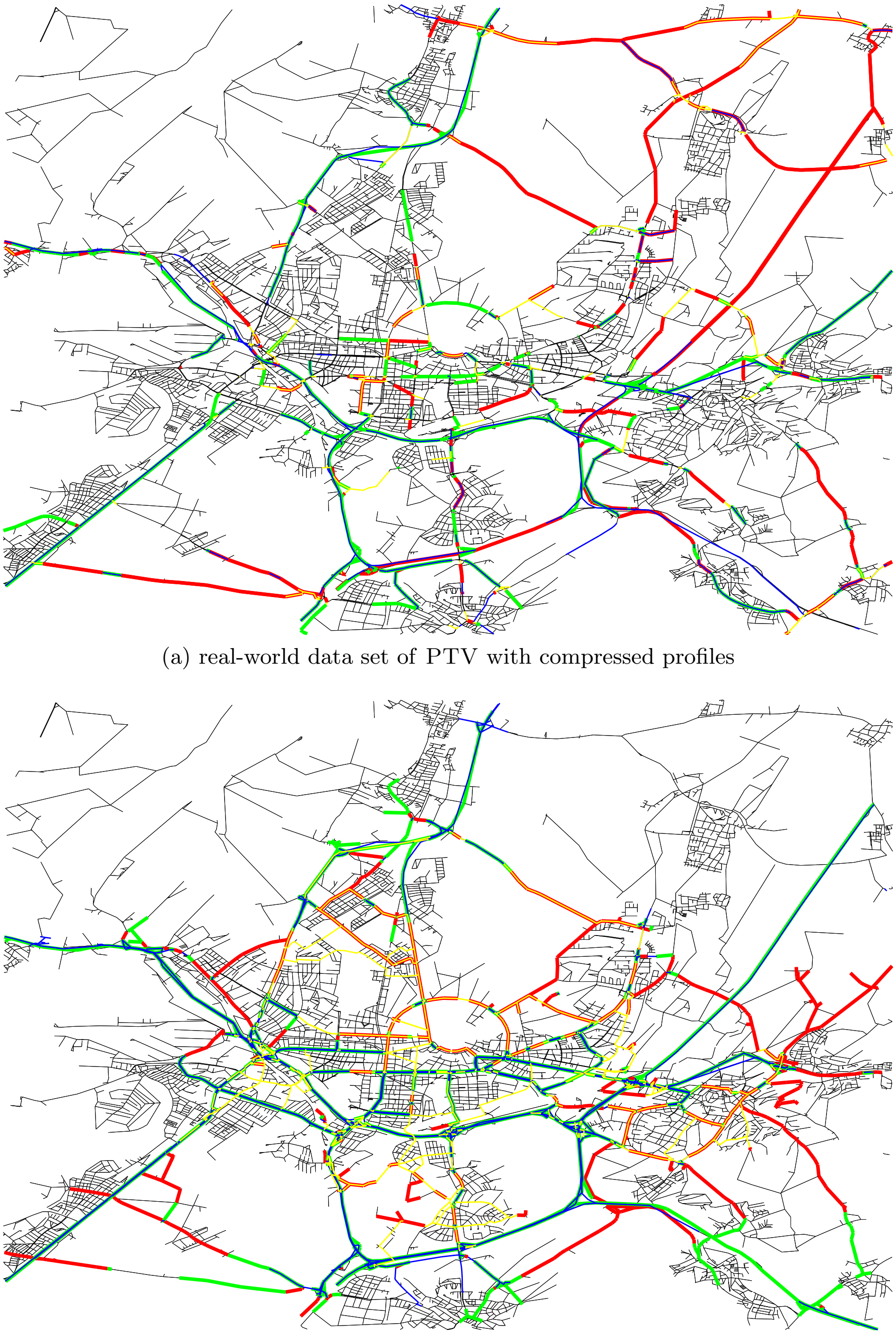
profile type	PTV	Category	Region	Level I	Level II
none	93.00%	92.60%	90.73%	92.13%	80.55%
camel	2.73%	2.19%	1.53%	3.60%	9.33%
morning	1.21%	1.22%	3.40%	2.44%	5.30%
afternoon	1.53%	1.22%	3.40%	1.30%	3.30%
full-time	1.50%	2.74%	0.92%	0.50%	1.52%
time in min	-	55	72	21	26

Visual Comparison We now take a look at excerpts of the data sets around the city of Karlsruhe, where the compressed profiles are highlighted in different colors. In particular, the colors correspond to profiles in the following way: red to camel, green to morning, blue to afternoon and yellow to full-time profiles. Note that lanes are printed closely together, and hence, colors may overlay to a certain extent. The excerpts we look at are shown in Figure 5.7 and Figure 5.8. For comparison, we present the real-world instance of PTV in Figure 5.7a. Note that the delays shown here are a local subset of a large instance, which is not particularly tuned for a high similarity in the visual appearance but for global statistics. In general, we can see that the overall picture is rather similar possibly despite individual differences, which are related to the behavior of the corresponding algorithm.

In Figure 5.7b of instance Category, we can observe that the urban roads are affected by full-time profiles as intended, and that the radial as well as the interconnection roads are affected by the remaining profile types. However, in the top right of the figure, we can observe that Affected-By-Category did not distribute profiles as the road categories on a section of a road are urban, and hence, the algorithm did not use up its capacity. Due to the increased simulation behavior of Affected-By-Region, we can see that the instance Region distributes profiles in every direction; see Figure 5.8a. We already mentioned the drawback that the behavior of long distance commuters is not modeled in this approach. Hence, we can observe that most of the road segments are affected by morning profiles in the one direction and afternoon profiles in the opposite direction. In Figure 5.8b of instance Level II, we can observe that the Affected-By-Level algorithm also considers smaller agglomerations as important, e. g., it assigns full-time profiles to rather small villages like on the top or on the right of the Figure. Note that a larger number of edges is affected in this instance compared to the other instances shown, which originates from the fact that almost 20% of the edges are affected by delays.

Local Statistical Properties The statistical properties given in Table 5.1 only reflect the global view on the distribution of time-dependent edges in the generated data sets. To compare this information on a local level, we created a *query set* consisting of 20 000 randomly chosen source-destination pairs. According to this query set, we performed Dijkstra queries and compared the occurrences of profile types on the shortest path found for each of the data sets.

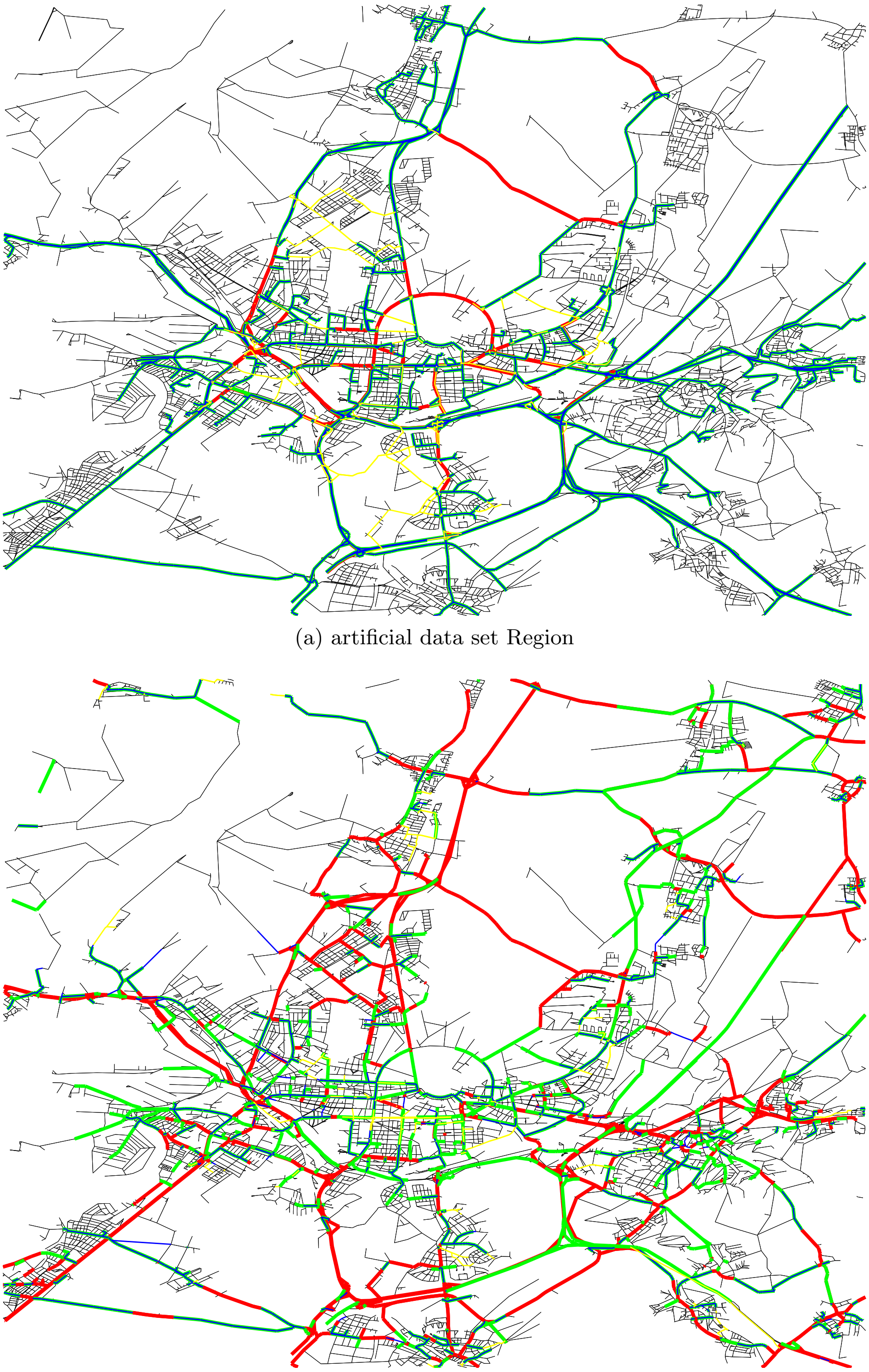
In Table 5.4 the average number of the encountered profile types during each query are shown. For a better overview, the total number of edges affected by time dependency are summed in the column $\sum TDE$. Generally speaking, despite minor outliers, the relations



(a) real-world data set of PTV with compressed profiles

(b) artificial data set Category

Figure 5.7: Excerpts of the time-dependent road networks around the city of Karlsruhe, road segments are colored according to their profile type (camel: red, morning: green, afternoon: blue, full-time: yellow, unaffected: black). Table 5.2 shows the parameters of the artificial instance.



(a) artificial data set Region

(b) artificial data set Level II

Figure 5.8: Excerpts of the time-dependent road networks around the city of Karlsruhe, road segments are colored according to their profile type (camel: red, morning: green, afternoon: blue, full-time: yellow, unaffected: black). Table 5.2 shows the parameters of the artificial instances.

Table 5.4: Number of profile types assigned to edges of shortest paths computed by the algorithm of time-dependent Dijkstra on the instances of the benchmark set and the real-world instance with compressed profiles. For each instance, queries were performed according to a query set that contains 20 000 randomly chosen source–destination pairs. The results have been averaged, and $\sum$TDE refers to the sum of time-dependent edges along the shortest paths.

data set	#full-time	#afternoon	#camel	#morning	#$\sum$TDE	#not-set
PTV	6.14	26.49	28.36	18.79	79.78	188.99
Category	8.16	35.36	7.73	40.18	91.43	177.33
Region	1.09	32.28	31.63	33.60	98.60	170.16
Level I	2.23	4.80	17.66	13.15	37.84	230.93
Level II	5.89	10.07	41.03	31.93	88.92	179.85

between time-dependent and time-independent edges of the data sets seem to be of equal size, except the instance Level I, however, this one fits the global statistical properties. The reason could be that the Affected-By-Level algorithm also considers smaller agglomerations of resources, and hence, distributes the profiles in the network, whereas the other algorithms concentrate on cities. Hence, for the same global statistics it is unlikely to encounter the same number of time-dependent edges in the case of rather long distance queries. The detailed distribution of the distinct profile types does not seem to follow a certain pattern despite Level I and Level II, which scale with respect to the total number of time dependent edges. From an experimenter's point of view, all instances behave similarly good, and hence, there is no best approximate instance, while Region exhibits the smallest sum of one-on-one distances. Therefore, we suggest to take all instances as a benchmark set.

Shortest-Path Behavior So far we assessed the generated data in terms of similar structural properties in global and to a certain extent in local scope. Next, we focus on the algorithmic behavior in order to show that an evaluation of shortest-path algorithms on the generated data sets gives similar results as for the real-world data set. To this end, we created a *query set* that consists of 5 000 distinct source nodes and randomly chosen target nodes of increasing Dijkstra rank, which results in a total number of 115 000 requests. In Table 5.5, we present the shortest-path properties of Dijkstra queries that were performed according to this query set. In particular, these are the number of settled nodes, touched edges and time-dependent edges (tdEdges). We can observe that the number of settled nodes and touched edges is almost similar for all tested instances. The number of time-dependent edges visited on average roughly corresponds to the ratio of time-dependent to time-independent edges. The remaining deviation might originate from local fluctuations. Additionally, the absolute error rate (errorRate) is given, which specifies the number of queries that had a different length compared to the referential distance computed on PTV. We can observe that about one quarter of the shortest-paths in the artificial instances had a length different to those of the real-world data set. However, the relative average difference (rel-avg) between the actual computed distance and the reference distance of the real-world graph is smaller than 0.6% for all instances. Additionally, we can observe that the relative maximal difference (rel-max) is less than 6% for Category, Region as well as Level I, and smaller than 10% for Level II, which is due to the increased number of

Table 5.5: Algorithmic properties of time-dependent Dijkstra-rank queries for 5 000 source nodes performed on the real-world and the benchmark instances. Abbreviated properties are: time dependent edges (tdEdges), absolute error rate (errorRate), relative average length difference (rel-avg) and relative maximal length difference (rel-max). The results show averaged values, except errorRate and rel-max.

data set	#settled nodes	#touched edges	#tdEdges	errorRate	rel-avg	rel-max
PTV	364 722	436 399	27 608	-	-	-
Category	364 700	436 362	32 684	22.77%	0.39%	5.88%
Region	364 705	436 364	37 853	26.07%	0.45%	5.88%
Level I	364 721	436 400	29 788	22.62%	0.43%	5.95%
Level II	364 725	436 407	73 641	21.88%	0.56%	9.70%

time dependent edges therein. The experiment indicates that all of our time-dependency generation algorithms lead to a similar behavior of the shortest path algorithm applied. However, we have yet not verified whether this holds for speedup techniques too, which we focus on next.

Time-Dependent Speedup Techniques So far, we tested statistical properties and algorithmic behavior with respect to time-dependent Dijkstra on instances generated by our algorithms. However, we aim at enlarging the set of reasonable benchmark sets for experimental algorithmics in the field of speedup techniques for time-dependent routing planning. Hence, we now focus on their behavior when applied to our benchmark set. In particular, we assess our benchmark set, and thus our algorithms, using the speedup techniques time-dependent (tdALT) by Nannicini et al. [NDLS08] and time-dependent Core-based ALT (tdCALT) by Delling and Nannicini [DN08].

In the work by Nannicini et al. [NDLS08], the only other known approach to generate artificial delays in road networks is proposed, which has been used in several other works [BGNS10, KLSV10, Del11], to name a few. This data set is also used for the experimental verification of tdALT and tdCALT [NDLS08, DN08]. A particular interest of the latter both works was to check whether incorporating a tolerable approximate ratio k results in further speedups of the algorithms. The experiments conducted on their artificially generated data set indicate that an approximation value of 15% yields the best trade-off in terms of speedup and incorrectness as it leads to a further speedup of about a factor of 10 in query time by reducing the number of settled nodes in the same order of magnitude [DN08]. However, this behavior cannot be observed on the real-world data set of Germany as it turns out that the speedup is rather limited to a factor of about 2 for both, the reduction in the number of settled nodes and the speedup of the query time [Del09]. Hence, we take this as an important indicator that our algorithms generate more realistic data, which exhibit a behavior of speedup techniques closer to that observed for real-world instances.

Similar to the other implementations, the code is written in C++ using the STL and no other additional library. The code was compiled with GCC 4.5 and optimization level 3. This particular experiment was conducted using OpenSuse 11.3 on one core of an AMD Opteron 2218, clocked at 2.6 GHz, having 1MB L2 cache and 16GB RAM at its disposal.

The tuning parameters we chose for the algorithm were for tdALT: 16 landmarks

Table 5.6: Settled nodes (#SN) and running time of the algorithms time-dependent ALT (tdALT) and time-dependent Core-based ALT (tdCALT), which incorporate an approximate factor k, where $k = 1.00$ equals the exact version and $k = 1.15$ an approximation of 15%. The algorithms were applied to the real-world data set with original profiles ($\mathrm{PTV}_{\mathrm{real}}$), the real-world data set with compressed profiles ($\mathrm{PTV}_{\mathrm{comp}}$) and the instances of the benchmark set.

	tdALT, k=1.00		tdALT, k=1.15		tdCALT, k=1.00		tdCALT, k=1.15	
data set	#SN	time [ms]	#SN	time [ms]	#SN	time [ms]	#SN	time [ms]
$\mathrm{PTV}_{\mathrm{real}}$	286 040	208.94	152 077	95.13	9 174	12.45	4 311	5.26
$\mathrm{PTV}_{\mathrm{comp}}$	184 808	124.59	107 904	68.66	12 019	15.51	5 853	6.93
Category	236 180	161.26	150 729	98.75	14 832	18.86	7 511	8.74
Region	241 442	164.65	146 618	94.91	15 665	20.97	7 921	9.72
Level I	157 266	104.37	102 821	65.88	9 915	12.72	5 258	6.27
Level II	182 184	130.30	106 381	71.10	18 871	24.91	9 349	11.27

computed by the avoid strategy [GW05], and for tdCALT: the contraction parameter $c = 2.5$, the hop bound limit $h = 30$, 32 landmarks computed by the avoid strategy and the limitation of the number of interpolation points on shortcuts to 300; for details see the work by Delling and Nannicini [DN08]. We applied the algorithms to the real-world instance with the profile set Tuesday to Thursday ($\mathrm{PTV}_{\mathrm{real}}$), the real-world instance with compressed profiles ($\mathrm{PTV}_{\mathrm{comp}}$) and the instances of the benchmark set. The algorithms were applied to each of these instances in the exact version ($k = 1.00$) and with an approximation factor of 15% ($k = 1.15$). We analyze the average number of settled nodes (#SN) and the average running time of 10 000 queries for randomly selected source-destination pairs. The outcomes of this experiment are shown in Table 5.6.

In general, we can observe that the algorithms tdALT and tdCALT exhibit a rather similar behavior on all instances. In more detail, we can find that the ratio of speedup achieved when switching from the exact to the approximate version of the algorithm is also preserved. On average, the compression of the profiles has only a rather limited influence on the performance of an algorithm. In particular, the performance of tdALT was slightly increased whereas it was slightly decreased with respect to tdCALT. In addition, we can see that the algorithm tdCALT applied to Level II results in an increased number of settled nodes as well as query time, which is a consequence of the increased number of affected edges of Level II compared to Level I. At the same time it preserves the speedup factor of about 2. Hence, the algorithms seem suitable for scalability tests with respect to the number of time-dependent edges. In summary, we note that instances generated by our algorithms approximate the behavior of real-world instances quite well, and in that, superior compared to the solely existing approach by Nannicini et al. [NDLS08].

5.5 Concluding Remarks

In this chapter, we presented the first algorithms to generate realistic time-dependency information for large-scale road networks of continental-size. The scenario we deal with consists of systematic delays on a daily basis within road-networks, which omits unexpected events like accidents, holiday traffic or weather conditions. These instances are typically used to verify the performance of speedup techniques in a time-dependent scenario. However, to the best of our knowledge no real-world instances are publicly available to researchers. Hence, the only known synthetic generator by Nannicini et al. is used [NDLS08], which is neither fully documented nor geared towards the properties found in real-world instances. Nevertheless, it is used in several other works [BGNS10, KLSV10, Del11].

Using such a generator may flaw experimental results as the following example shows. Some speedup techniques allow for a larger gain in acceleration if we settle for approximate results. In particular, an additional speedup of a factor of 10 was achieved for an approximate ratio of 15% when algorithms were applied to instances generated by Nannicini et al. [DN08], which seems to be a good trade-off. However, it turns out that the same algorithms using the same approximate ratio applied to real-world instances only have a marginal additional speedup of about 2 [Del09], which now seems to be a rather bad trade-in. This led to some uncertainty when using this generator, which we saw as a good reason to develop algorithms that are capable of generating more realistic instances.

In this chapter, we focus on enlarging the set of available instances for experimentally verifying speedup techniques in the time-dependent scenario. We rate artificial instances as realistic if they admit similar statistical local and global properties compared to a real-world data set. Additionally, time-dependent shortest-path algorithms should behave similarly on artificial and on real-world instances. Hence, we assessed our algorithms with respect to these criteria.

In a preliminary step, we strengthened our intuition on the origin and structure of traffic flows by the analysis of a confidential time-dependent road network of Germany[3]. This data set is two-part and consists of a static road network and time-dependent information assigned to edges, which in our case are piece-wise linear functions that describe the increase of travel time for specific points of a day, where the values in between are interpolated linearly. We refer to these functions as *profiles*. By our analysis, we found a set of four representative profiles that are related to daily routine and the modeling of a network. We further observed that people mostly travel locally and that towns attract individuals from a certain area around it. We condensed these observations in a model that assumes commuters behave in predictable ways, depending on their location within a road network. Hence, we had to compute the edges that are likely to be affected by time dependency.

Based on this model, we developed a generic five-phases approach, which consists of the following steps. In the *preprocessing phase* auxiliary data is computed, which splits the network into urban and rural areas by the utilization of either road categories or coordinates. In the *urban phase*, we determine heavily used roads within towns, and in the *rural phase*, we compute congested roads that lead into and out of towns. In general, we do not assign profiles directly but count them and determine their type. This allows for adjustments in the *filtering phase*, e. g., removing congestions on less heavily used roads. Assigning profiles to road segments is performed in the *postprocessing phase*, which in our case is done according to a fixed mapping to preserve comparability.

This five-phases approach guided our development of three algorithms that allow for artificially assigning delays on top of static road networks. The algorithms AFFECTED-

[3] Provided to us by the company PTV AG.

By-Category and Affected-By-Region both use road categories to compute this auxiliary data. The algorithm Affected-By-Category solely uses the road-category information to assign profiles to edges. On the other hand, the algorithm Affected-By-Region uses this information only to compute urban areas. To assign profiles to heavily used roads, Affected-By-Region then simulates traffic from the urban catchments into their urban area and back. The algorithm Affected-By-Level solely uses coordinate information to compute the level of each node, where a high level corresponds to urban areas and a low level corresponds to rural regions, respectively. This information is then used to simulate traffic from and to different levels within the network. Roads used heavily thereby are identified, which then get according profiles assigned. In the development of our algorithms, we shifted from solely using structural properties in Affected-By-Category over partly incorporating simulative aspects in Affected-By-Region to a full simulation of the traffic in Affected-By-Level. Note that albeit we use the term *simulation*, our approaches are by no means some kind of a traffic simulation encountered in the field of traffic and transportation prediction.

Having these algorithms at hand, we performed an extensive computational study to assess the quality of thereby generated data sets. It turns out, the parameters of our generators can be fine-tuned to approximate the real-world data set with respect to statistical properties on global and on local scope. We then used these instances to analyze the algorithmic behavior of time-dependent shortest-path techniques applied to the artificial and real-world road networks. Using the algorithm of Dijkstra, we confirmed that all instances exhibit similar characteristics. Even more important, it turns out that this is also true for the time-dependent speedup techniques we applied, which was not the case for instances generated by Nannicini et al. [NDLS08].

Hence, we conclude that our work allows experimenters to validate algorithms for time-dependent shortest-path queries on more realistic data than it was previously possible. The algorithms incorporate many degrees of freedom that allow for an adaption to various scenarios, of which one could be to approximate the real-world data set of PTV. Furthermore, our general five-phases approach allows for improving or replacing a single phase as we did in the development of Affected-By-Region. The running times of our algorithms are practical in the sense that using reasonable parameters, continental-sized graphs can be handled within a few hours up to three days. This is already acceptable, given that in our implementations we compute many shortest-paths solely using the algorithm of Dijkstra, whose replacement by speedup techniques where possible allows for a great acceleration.

Throughout this chapter, we were guided by the cycle of Algorithm Engineering. In particular, we developed a model by the analysis of real-world data, derived three algorithms, implemented those, evaluated them experimentally and adapted the algorithms where necessary. Even if not reported on, we have gone through this cycle several times, improved algorithms or came up with a new one.

Future Work In a first step, the performance of the algorithms can be increased by incorporating speedup techniques where possible. In addition, the filtering and the preprocessing phase can be improved to be more realistic, which includes to take heavy utilization of road segments into account, to make transitions of adjacent profiles smooth by interpolation, and to preserve similar traffic behavior along paths.

We already mentioned that our kind of simulations is not related to those conducted in the field of traffic and transportation prediction as their computations are too complex to be applied to road networks of continental size. However, we were inspired by and still see certain touching points that would definitely improve the quality of the model behind our generators. Therefore, we think closing this gap could be seen as a most challenging task, which is right now wishful thinking.

Chapter 6

Conclusion

This thesis addressed the problem of meaningful selecting artificial instances when conducting experimental evaluations of algorithms in two important fields of research: algorithms for planar graphs and algorithms for route-planning. Experimentally assessing algorithms is a crucial step in the cycle of Algorithm Engineering, of which the core is a circular flow through four phases — design, analysis, implementation, and experimental evaluation.

A particular goal of Algorithm Engineering is to not only consider worst cases but to also assess practicality of evaluated algorithms. Hence, it is natural to perform such an evaluation on real-world data. However, often such instances are not available or only to a limited extent. A common and established way is to use algorithms that artificially generate instances instead and to utilize these as a test-bed. It is most important to select these algorithms and thereby generated instances properly to prevent flawed conclusions derived from an experimental analysis conducted thereon. However, we have seen cases in the field of algorithm research for planar graphs that fail in doing so.

In the first part of this thesis, we dealt with the above problem by classifying a set of planar-graph generators that allows for their reasonable selection, such that conducted experiments result in reliable insights. In particular, we analyzed several planar-graph generators that were selected according to availability, theoretical interest, ease of implementation and efficiency. Our analysis of the algorithms covered both, theoretical and practical aspects. In particular, we derived some new aspects of the studied algorithms by their theoretical analysis. On the other hand, we experimentally analyzed their implementations and concluded on their practicability. We saw that some widely used algorithms solely generate instances that exhibit a considerable bias, which has a major influence on algorithmic behavior, in particular, in the field of fixed-parameter algorithmics. Altogether, this study helps experimenters to carefully select sets of planar graphs that allow for a meaningful interpretation of their results.

A further favorable goal of the experimental phase in the cycle of Algorithm Engineering is to assess scalability and robustness of algorithms. In particular, we refer to scalability in terms of running time as well as memory consumption, and to robustness in terms of a similar behavior on diverse instances. However, in the research field of route planning almost no reliable up-to-date real-world instance and, additionally, no assessed generators exist, which makes propositions on scalability and robustness of algorithms developed there almost impossible.

In the second part of this thesis, we tackled this problem by enlarging the set of instances that are typically used to experimentally assess route-planning algorithms for both important scenarios, i. e., time-independent and time-dependent scenarios. We dealt with each of them in separate chapters.

In Chapter 4, we focused on the generation of realistic instances of road networks. We measured the realism of artificial instances compared to real-world instances with respect to structural properties, algorithmic behavior, and visual similarity. We assessed the only existing model that has been realized by a theoretical generator, but which, until now, has neither been implemented nor experimentally tested. Thereby generated graphs fulfilled our requirements on realistic artificial instances only to a limited extent. Hence, we proposed an alternative model, whose derived generator relies on the recursive computation of Voronoi-diagrams, which allows for generating graphs that are much more realistic with respect to their visual appearance. We further analyzed the generators with respect to structural properties and algorithmic behavior, where it turned out that both generators approximate the real-world instances quite well, a task at which our newly proposed generator gains a slight advantage. A scalability test showed that both generators perform well with respect to memory consumption and running time. In summary, this chapter provides ready-to-use generators that are capable of generating realistic artificial instances of road networks, whose properties can be steered in order to challenge algorithms.

In Chapter 5, we developed algorithms that artificially generate realistic daily traffic information on top of road networks of continental size. In our context, daily traffic information specifies for distinct times of a day how long it takes to traverse a road compared to the uncongested case, where daily refers to the expected delay of each day, excluding accidents, holiday traffic or weather influence. These instances are typically used to assess time-dependent shortest-paths algorithms. However, the only existing generator is rather undocumented and it creates instances that do not allow for conclusions on the performance of algorithms applied to real-world instances. Hence, we aimed at closing the eminent gap in the cycle of Algorithm Engineering. By the analysis of a confidential real-world time-dependent data set, we strengthened our intuition on the sources of traffic, and condensed our observations in a realistic model. Based on this model, we developed a generic five-phases approach, which was used to formulate three algorithms that utilize either road categories or coordinates to artificially enrich a given road network with traffic information. On the possible scale to develop algorithms that enrich a static road network with time-dependent data, we started at solely exploiting structural information and ended at solely simulating traffic. We assessed instances that were generated by these algorithms to be realistic in the sense that they exhibit statistical properties and a behavior of time-dependent shortest-path algorithms similar to those observed for real-world instances. This chapter provides ready-to-use generators that allow for the generation of realistic daily traffic information in road networks that may originate from manifold sources like commercial, open source or artificial ones. This remedies the situation that no realistic data sets are publicly available.

Throughout this thesis, we were guided by the cycle of Algorithm Engineering. Therein, we closed significant gaps in the field of planar graphs and of route planning in static as well as time-dependent scenarios. We achieved this by paving the way for reliable, efficient and representative experimental evaluations. Hence, for these fields, we feel confident that we greatly improved the situation of researchers that experimentally assess algorithms.

Curriculum Vitæ

Name	Sascha Meinert
Date of Birth	7th of April 1977
Place of Birth	Neunkirchen-Seelscheid, Germany
Nationality	German

since 2/2011	external PhD student, chair *Algorithmics I*, Karlsruhe Institute of Technology (KIT). Advisor: Prof. Dr. Dorothea Wagner
04/2005-01/2011	PhD student and research assistant, chair *Algorithmics I*, Karlsruhe Institute of Technology (KIT). Advisor: Prof. Dr. Dorothea Wagner
02/2005-05/2005	PhD student and research assistant, chair *Paralleles Rechnen*, Eberhardt-Karls-Universität Tübingen. Advisor: Prof. Dr. Michael Kaufmann
10/1997-01/2005	Student in Informatics at Universität Tübingen. Finished with Diploma in Informatics.
07/1996-07/1997	Military service
06/1996	Abitur (university entrance qualification), Burghardt-Gymnasium Buchen

List of Publications

Book Chapters

[1] **Modeling**. In: *Algorithm Engineering*, pages 16–57. Springer, 2010. Joint work with Markus Geyer and Benjamin Hiller.

Articles in Refereed Conference Proceedings

[2] **An Experimental Study on Generating Planar Graphs**. In: *Proceedings of the 7th International Conference on Algorithmic Aspects in Information and Management (AAIM'11)*, volume 6681 of *Lecture Notes in Computer Science*. Springer, 2011. Joint work with Dorothea Wagner.

[3] **Generating Time Dependencies in Road Networks**. In: *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*. Springer, 2011. Joint work with Dorothea Wagner.

[4] **Synthetic Road Networks**. In: *Proceedings of the 6th International Conference on Algorithmic Aspects in Information and Management (AAIM'10)*, volume 6124 of *Lecture Notes in Computer Science*. Springer, 2010. Joint work with Reinhard Bauer, Marcus Krug and Dorothea Wagner.

Technical Reports

[5] **An Experimental Study on Generating Planar Graphs**. Technical report, ITI Wagner, Department of Informatics, Karlsruhe Institute of Technology (KIT), 2011. Karlsruhe Reports in Informatics 2011-13, joint work with Dorothea Wagner.

[6] **GraphArchive - An Online Graph Data Store**. Technical report 3, Wilhelm-Schickard-Institut, Eberhard-Karls-Universität Tübingen, 2011. Universitätsbibliothek Tübingen, WSI 2011-03, joint work with Philip Effinger, Michael Kaufmann and Matthias Stegmaier.

Thesis

[7] **Ein Softwareentwurf für ein Client/Server System zum Archivieren und Austauschen von Graphen im GraphML-Format basierend auf Web-Services**. Diplomarbeit Informatik, Universität Tübingen, January 2005.

Bibliography

[ABN06] Jochen Alber, Nadja Betzler, and Rolf Niedermeier. Experiments on data reduction for optimal domination in networks. *Annals of Operations Research*, 146(1):105–117, 2006.

[ACP87] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of Finding Embeddings in a k-Tree. *SIAM Journal on Matrix Analysis and Applications*, 8(2):277–284, 1987.

[ADN05] Jochen Alber, Frederic Dorn, and Rolf Niedermeier. Experimental evaluation of a tree decomposition-based algorithm for vertex cover on planar graphs. *Discrete Applied Mathematics*, 145(2):219–231, 2005.

[AFF$^+$05] Jochen Alber, Hongbing Fan, Michael R. Fellows, Henning Fernau, Rolf Niedermeier, Frances Rosamond, and Ulrike Stege. A refined search tree technique for Dominating Set on planar graphs. *Journal of Computer and System Sciences*, 71(4):385–405, 2005.

[AFGW10] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In Moses Charikar, editor, *Proceedings of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'10)*, pages 782–793. SIAM, 2010.

[AFN02] Jochen Alber, Michael R. Fellows, and Rolf Niedermeier. Efficient data reduction for Dominating Set: a linear problem kernel for the planar case. In Martti Penttonen and Erik Meineche Schmidt, editors, *Proceedings of the 8th Scandinavian Workshop on Algorithm Theory (SWAT'04)*, volume 2368 of *Lecture Notes in Computer Science*, pages 150–159. Springer, July 2002.

[AFN04] Jochen Alber, Henning Fernau, and Rolf Niedermeier. Parameterized complexity: exponential speed-up for planar graph problems. *Journal of Algorithms*, 58(1):26–56, July 2004.

[AKCF$^+$04] Faisal N. Abu-Khzam, Rebecca L. Collins, Michael R. Fellows, Michael A. Langston, W. Henry Suters, and Chris T. Symons. Kernelization Algorithms for the Vertex Cover Problem: Theory and Experiments. In ALENEX'04 [ALE04], pages 62–69.

[ALE04] *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*. SIAM, 2004.

[AP89] Stefan Arnborg and Andrzej Proskurowski. Linear Time Algorithms for NP-hard Problems Restricted to Partial k-Trees. *Discrete Applied Mathematics*, 23(1):11–24, April 1989.

[BCF$^+$08] Prosenjit Bose, Paz Carmi, Mohammad Farshi, Anil Maheshwari, and Michiel Smid. Computing the Greedy Spanner in Near-Quadratic Time. In *Proceedings of the 11th Scandinavian Workshop on Algorithm Theory (SWAT'08)*, volume 5124 of *Lecture Notes in Computer Science*, pages 390–401. Springer, 2008.

[BCK$^+$10] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. Preprocessing Speed-Up Techniques is Hard. In *Proceedings of the 7th Conference on Algorithms and Complexity (CIAC'10)*, volume 6078 of *Lecture Notes in Computer Science*, pages 359–370. Springer, 2010.

[BDD$^+$10] Reinhard Bauer, Gianlorenzo D'Angelo, Daniel Delling, Andrea Schumm, and Dorothea Wagner. The Shortcut Problem – Complexity and Algorithms . Technical Report 2010-17, ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH), 2010.

[BDG$^+$08] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Höfer, Zoran Nikoloski, and Dorothea Wagner. On Modularity Clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20(2):172–188, February 2008.

[BDW07] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. In Christian Liebchen, Ravindra K. Ahuja, and Juan A. Mesa, editors, *Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07)*, OpenAccess Series in Informatics (OASIcs), pages 209–225, 2007.

[BE05] Ulrik Brandes and Thomas Erlebach, editors. *Network Analysis: Methodological Foundations*, volume 3418 of *Lecture Notes in Computer Science*. Springer, February 2005.

[BF67] Rufus Bowen and Stephen Fisk. Generation of triangulations of the sphere. *Mathematics of Computation*, 21:250–252, 1967.

[BFM$^+$07] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In Transit to Constant Shortest-Path Queries in Road Networks. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 46–59. SIAM, 2007.

[BFSS07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007.

[BG10] Hillel Bar-Gera. Traffic assignment by paired alternative segments. *Transportation Research Part B: Methodological*, 44(8-9):1022–1046, 2010.

[BGK07] Manuel Bodirsky, Clemens Gröpl, and Mihyun Kang. Generating labeled planar graphs uniformly at random. *Theoretical Computer Science*, 379(3):377–386, 2007.

[BGNS10] Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders. Time-Dependent Contraction Hierarchies and Approximation. In Festa [Fes10], pages 166–177.

[BHT06] Endre Boros, Peter L. Hammer, and Gabriel Tavares. Preprocessing of unconstrained quadratic binary optimization. Technical Report 10, Rutgers Center for Operations Research (RUTCOR), 2006.

[BK10] Hans L. Bodlaender and Arie M.C.A. Koster. Treewidth computations I. Upper bounds. *Information and Computation*, 208(3):259–275, 2010.

[BKMW10] Reinhard Bauer, Marcus Krug, Sascha Meinert, and Dorothea Wagner. Synthetic Road Networks. In *Proceedings of the 6th International Conference on Algorithmic Aspects in Information and Management (AAIM'10)*, volume 6124 of *Lecture Notes in Computer Science*, pages 46–57. Springer, 2010.

[BM07] Gunnar Brinkmann and Brendan D. McKay. Fast generation of planar graphs. *MATCH - Communications in Mathematical and in Computer Chemistry*, 58(2):323–357, 2007.

[boo] Boost C++ Libraries.

[BWK06] Hans L. Bodlaender, Thomas Wolle, and Arie M.C.A. Koster. Contraction and Treewidth Lower Bounds. *Journal of Graph Algorithms and Applications*, 10(1):5–49, 2006.

[cga] Cgal, Computational Geometry Algorithms Library.

[CGR96] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms. *Mathematical Programming, Series A*, 73:129–174, 1996.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 2nd edition, 2001.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing (STOC'71)*, pages 151–158. ACM Press, May 1971.

[Dan62] George B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1962.

[dBCvKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark H. Overmars. *Computational Geometry: Algorithms and Applications.* Springer, 2008.

[DBETT98] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing - Algorithms for the Visualization of Graphs.* Prentice Hall, 1998.

[Del09] Daniel Delling. *Engineering and Augmenting Route Planning Algorithms.* PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2009.

[Del11] Daniel Delling. Time-Dependent SHARC-Routing. *Algorithmica*, 60(1):60–94, May 2011. Special Issue: European Symposium on Algorithms 2008.

[DF99] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity.* Monographs in Computer Science. Springer, 1999.

[DFLS04] Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann Samplers for the Random Generation of Combinatorial Structures. *Combinatorics, Probability and Computing*, 13(4-5):577–625, 2004.

[DGJ09] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book.* American Mathematical Society, 2009.

[DHJ^{+}95] F. Daly, D.J̃. Hand, M. C. Jonas, A.D̃. Lunn, and K. J. McConway. *Elements of Statistics.* The Open University,, 1995.

[Die00] Reinhard Diestel. *Graph Theory.* Graduate Texts in Mathematics. Springer, 2nd edition, 2000.

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[Dil90] Michael B. Dillencourt. Realizability of Delaunay triangulations. *Information Processing Letters*, 33(6):283–287, 1990.

[DKS09] Guoli Ding, Jinko Kanno, and Jianning Su. Generating 5-regular planar graphs. *Journal of Graph Theory*, 61(3):219–240, July 2009.

[DN08] Daniel Delling and Giacomo Nannicini. Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks. In Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, editors, *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08)*, volume 5369 of *Lecture Notes in Computer Science*, pages 813–824. Springer, December 2008.

[DSSW09a] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.

[DSSW09b] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Highway Hierarchies Star. In Demetrescu et al. [DGJ09], pages 141–174.

[DVW96] Alain Denise, Marcio Vasconcellos, and Dominic J. A. Welsh. The random planar graph. *Congressus Numerantium*, 113:61–79, 1996.

[DW09] Daniel Delling and Dorothea Wagner. Time-Dependent Route Planning. In Ravindra K. Ahuja, Rolf H. Möhring, and Christos Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 207–230. Springer, 2009.

[Dwy87] Rex A. Dwyer. A faster divide-and-conquer algorithm for constructing delaunay triangulations. *Algorithmica*, 2(1):137–151, 1987.

[EG08] David Eppstein and Michael T. Goodrich. Studying (non-planar) road networks through an algorithmic lens. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems (GIS '08)*, pages 1–10. ACM Press, 2008.

[Ert98] Gerhard Ertl. Shortest path calculation in large road networks. *OR Spectrum*, 20(1):15–20, 1998.

[Fes10] Paola Festa, editor. *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*. Springer, May 2010.

[Flö08] Gunnar Flötteröd. *Traffic State Estimation with Multi-Agent Simulations*. PhD thesis, Technische Universiät Berlin, 2008.

[FT87] Michael L. Fredman and Robert E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.

[Fus05] Éric Fusy. Implementation of a Boltzman Sampler for planar graphs, 2005.

[Fus09] Éric Fusy. Uniform random sampling of planar graphs in linear time. *Randoms Structures and Algorithms*, 35(4):464–522, December 2009.

[Gae05] Marco Gaertler. Clustering. In Brandes and Erlebach [BE05], pages 178–215.

[GD04] Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. In Max Chickering, Joseph Halpern, and Christopher Meek, editors, *UAI'04 Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 201–208. AUAI Press, 2004.

[GH05] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165. SIAM, 2005.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness.* W. H. Freeman and Company, 1979.

[GKW09] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Shortest Path Algorithms with Preprocessing. In Demetrescu et al. [DGJ09], pages 93–139.

[GN09] Omar Giménez and Marc Noy. Asymptotic enumeration and limit laws of planar graphs . *Journal of the American Mathematical Society*, 22(2):309–329, April 2009.

[GS97] Charles Miller Grinstead and James Laurie Snell. *Introduction to Probability.* American Mathematical Society, 1997.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In McGeoch [McG08], pages 319–333.

[Gut04] Ronald J. Gutman. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In ALENEX'04 [ALE04], pages 100–111.

[GW05] Andrew V. Goldberg and Renato F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 26–40. SIAM, 2005.

[Hel07] Guido Helden. *Hamiltonicity of maximal planar graphs and planar triangulations.* PhD thesis, RWTH Aachen, 2007.

[HKMS09] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Demetrescu et al. [DGJ09], pages 41–72.

[HNR68] Peter E. Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[HT74] John E. Hopcroft and Robert E. Tarjan. Efficient Planarity Testing. *Journal of the ACM*, 21(4):549–568, October 1974.

[Kar72] Richard M. Karp. Reducibility among Combinatorial Problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[Ker04] Boris S. Kerner. *The Physics of Traffic.* Springer, 2004.

[KLSV10] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. Distributed Time-Dependent Contraction Hierarchies. In Festa [Fes10], pages 83–93.

[KMR11] Marcus Krug, Sascha Meinert, and Ignaz Rutter. Efficient Node Expansion. 2011.

[KS93] David E. Kaufman and Robert L. Smith. Fastest Paths in Time-Dependent Networks for Intelligent Vehicle-Highway Systems Application. *Journal of Intelligent Transportation Systems*, 1(1):1–11, 1993.

[KS06] Marcus Krug and Jochen Speck. Praktikum Graphgeneratoren: Planare Graphen, 2006. Report on Lab Course, written in German.

[Kur30] Casimir Kuratowski. Sur le problème des courbes gauches en Topologie. *Fundamenta Mathematicae*, 15:271–283, 1930.

[Lau04] Ulrich Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230. IfGI prints, 2004.

[Llo82] Stuart Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[McG08] Catherine C. McGeoch, editor. *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science.* Springer, June 2008.

[MN99] Kurt Mehlhorn and Stefan Näher. *LEDA, A platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[MS10] Matthias Müller–Hannemann and Stefan Schirra, editors. *Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice*, volume 5971 of *Lecture Notes in Computer Science.* Springer, 2010.

[Mul90] Ketan Mulmuley. A fast planar partition algorithm. *Journal of Symbolic Computation*, 10(3-4):253–280, August 1990.

[MW11a] Sascha Meinert and Dorothea Wagner. An Experimental Study on Generating Planar Graphs. In Mikhail Atallah, Xiang-Yang Li, and Binhai Zhu, editors, *Proceedings of the 7th International Conference on Algorithmic Aspects in Information and Management (AAIM'11)*, volume 6681 of *Lecture Notes in Computer Science*, pages 375–387. Springer, 2011.

[MW11b] Sascha Meinert and Dorothea Wagner. Generating Time Dependencies in Road Networks. In Panos M. Pardalos and Steffen Rebennack, editors, *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 434–446. Springer, 2011.

[NDLS08] Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes. Bidirectional A* Search for Time-Dependent Fast Paths. In McGeoch [McG08], pages 334–346.

[New04] Mark E. J. Newman. Fast Algorithm for Detecting Community Structure in Networks. *Physical Review E*, 69:066133, 2004.

[NG04] Mark E. J. Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(026113):1–16, 2004.

[Nie06] Rolf Niedermeier. *Invitation to Fixed Parameter Algorithms.* Oxford University Press, 2006.

[OR90] Ariel Orda and Raphael Rom. Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length. *Journal of the ACM*, 37(3):607–625, 1990.

[PS85] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction.* Springer, 1985.

[PS89] David Peleg and Alejando A. Schäffer. Graph Spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

[Sch08] Dominik Schultes. *Route Planning in Road Networks.* PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, February 2008. `http://algo2.iti.uka.de/schultes/hwy/schultes_diss.pdf`.

[Sei83] Stephen B. Seidman. Network Structure and Minimum Degree. *Social Networks*, 5:269–287, 1983.

[Sha01] Claude E. Shannon. A Mathematical Theory of Communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, January 2001.

[SS05] Peter Sanders and Dominik Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

[SW05] Thomas Schank and Dorothea Wagner. Approximating Clustering Coefficient and Transitivity. *Journal of Graph Algorithms and Applications*, 9(2):265–275, 2005.

[Vaz03] Vijay V. Vazirani. *Approximation Algorithms.* Springer, 2003.

[vDvdHS06] Thomas van Dijk, Jan van den Heuvel, and Wouter Slob. Computing treewidth with LibTW, 2006.