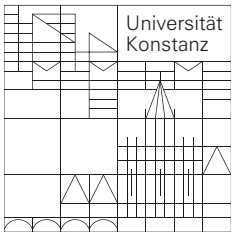
Listing all Maximal Cliques of Large Sparse Graphs

Bachelorarbeit

vorgelegt von
Sebastian Fichtner



Fachbereich Informatik

1. Gutachter: Prof. Dr. Ulrik Brandes
2. Gutachter: Dr. Andreas Karrenbauer
Betreuer: Prof. Dr. Ulrik Brandes

Konstanz, 2012

Contents

Abstract

Listing maximal cliques is a basic task of data analysis. This bachelor thesis investigates methods applicable to large sparse graphs. It is especially focused on the 2010 publication of Eppstein et al. [13], who improved the most practically relevant algorithm. We discuss their approach, its predecessors and several variants in theory and propose a modification to accelerate listing maximal cliques of a minimal size. The algorithms are evaluated through a systematic empirical comparison on synthetic and real graphs with regard to clique related graph parameters. As a result, we illustrate and explain the behavior of different algorithms on different graph types and are able to show that our modification can be the dominating choice in practice.

Chapter 1

Introduction

1.1 Cliques in Graphs

Graphs may be the most fundamental mathematical concept next to numbers. At its core, a graph is a binary relation on a set of arbitrary objects. In graph theory, we think of it as a set of vertices (or nodes) together with a set of edges (connections) between them.

Definition 1. *$G = (V, E)$ is a graph with vertices $V = \{v_1 ... v_n\}$ and edges $E \subseteq V \times V$. E is symmetric and irreflexive. The number of edges is $m = |E|$. G is complete iff $\forall u, v \in V : u \neq v \Rightarrow (u, v) \in E$. G is a subgraph of graph U iff $V(G) \subseteq V(U)$ and $E(G) \subseteq E(U)$.*

Due to the simplicity of these elements, most real world scenarios can be translated into an abstract graph representation. Therefore, it comes as no surprise that a solution to a graph problem is a solution to countless related problems in practice, even those that yet have to be discovered.

This work is dedicated to the essential class of graph problems that descend from the famous CLIQUE problem, which is known to be NP-complete. We are particularly interested in algorithms that list all maximal cliques. Here, a clique is referred to as the vertex set of a complete subgraph of G. It is maximal if it isn't the proper subset of another clique. This is not to be confused with the maximum clique, which is just a clique of maximum size, in the sense that no other one contains more vertices.

Definition 2. *U is a clique iff it is the vertex set of a complete subgraph of G. U is a maximal clique (MC) iff U is a clique and no clique W exists with $U \subset W$. The number of MCs of minimal size s is $\mu_s = |\{U | U$ is a MC $\wedge |U| \geq s\}|$. We write $\mu_1 = \mu$.*

Listing all MCs is a fundamental graph problem of general applicability. MCs specify the largest distinct subgraphs that are perfectly dense. They indicate local density or clusters, which makes them inevitable for basic data analysis. Furthermore, many clique related problems can be transformed into one another. The MCs of a graph, for instance, are equal to the maximal independent sets of the inverse graph. An algorithm that solves one clique problem can easily be adjusted to such different points of view, making it applicable to the wide range of related problems that arise in practice.

Many applications attest to the practical relevance of clique detection. A popular example is social network analysis, where such algorithms are used to identify communities [2, 10]. Nowadays, the most prominent domain is bioinformatics, where – by finding cliques in graphs – protein structures are predicted [36], clustered [31, 32], searched for frequent patterns [17, 25, 24] and compared (for instance, to find docking regions) [15, 20]. Many more applications have been reported, including document clustering [1], approximating depth from images [21], discovering frequent item sets [40] and solving problems of computational topology [41]. In the following section, we will see why finding cliques is quite a challenge.

1.2 Complexity

1.2.1 General Graphs

The problem to decide whether or not a given graph contains a clique of size k or larger is known as CLIQUE. It is one of the 21 basic decision problems that Richard Karp in 1972 proved to be NP-complete [23]. So, unless $P = NP$, there is no polynomial algorithm that could solve it. In 1991, Feige et al. [14] showed that the closely related maximum clique problem can not even be approximated in polynomial time. Our focus lies on listing all maximal cliques, which is not a decision problem and can certainly not be easier than finding a maximum clique since every maximum clique must also be maximal. Thus, our problem is NP-hard. And as if that wasn't enough, we have to recognize that just to count all MCs would take exponential time since the maximum number of MCs is $3^{n/3}$, as Moon and Moser have shown [33]. This maximum is realized in a so called "Moon-Moser Graph", in which nodes are divided into non-connected triplets while every two nodes from different triplets are connected. We select a MC from such a graph by picking a node from each of the $n/3$ triplets, resulting in $3^{n/3}$ possible choices. To understand why 3 of all numbers must be the size of non-connected sets, let

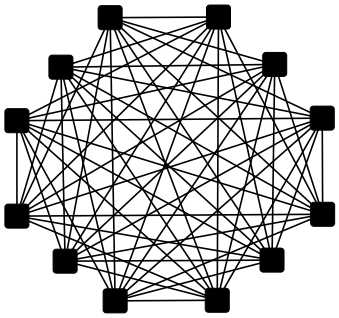


Figure 1.1: Moon-Moser graph containig 4 triplets and 81 MCs

x be that size, and we get

$$\arg\max_{x}(x^{n/x}) = \mathrm{e}$$

with "e" being Euler's number. The Moon-Moser graph depicted in Figure (1.1) contains 12 nodes and, therefore, $3^{12/3} = 81$ MCs. Just for the sake of argument, $n \; mod \; 3 = 0$ in typical Moon-Moser graphs. If $n \; mod \; 3 = 1$, the graph would have to contain 2 non-connected pairs to maximize the number of MCs, and one such pair if $n \; mod \; 3 = 2$.

1.2.2 Density Indicators

If we would not further specify the type of input graph, any algorithm would need exponential time to list all MCs, but looking at Figure (1.1) again, it should become clear that the real networks we want to analyze usually don't look anything like that. Their number of edges is said to be in $O(n)$ instead of $O(n^2)$, which is why they are called "sparse". So, in order to gain a complexity bound of higher practical relevance, we need to consider graph density. Its classic measure is the edge probability

$$p = \frac{2m}{n^2 - n}$$

Here, we understand it as the probability that two different arbitrary nodes are connected, rather than the probability with which edges were created. For the graph from Figure (1.1), p is about 0.82.

However, overall density does not limit the connectedness of subgraphs, which is why degeneracy δ and arboricity α are the most widely used indicators of graph density in our context. To be more accurate: They indicate the maximum connectedness of a graph and thereby also limit its density

(Proposition 3 in [26]) while p, on the other hand, does not provide a good limit for δ or α since edges could still be concentrated on a highly connected subgraph. We will briefly explain both indicators in the following.

There are many ways to approach degeneracy. Eppstein and Strash [13] give us a very compact definition:

Definition 3. *The degeneracy of a graph G is the smallest number δ such that every subgraph of G contains a vertex of degree at most δ.*

For further clarification of δ, we will outline the algorithm that not only leads to the number itself but also generates a δ-order of all nodes. Let there be a graph. If we repeatedly remove nodes of a degree lower than k until no such node is left, the remaining subgraph is the k-core of the original graph. Of course, the $(k+1)$-core is a subgraph of the k-core. The core number of a node is the biggest k for which it belongs to the k-core. δ-order only requires nodes to be sorted by their core numbers and can be computed in linear time by repeatedly removing a node of lowest degree until the graph is empty, while remembering the highest degree (that has occurred) as the current minimal core number. The biggest core number that is reached is the graph's degeneracy δ.

Every clique of size $k+1$ is a subgraph of the k-core because right before the algorithm removes the first node from the clique, that node has a degree of at least k. Therefore, a graph cannot contain cliques larger than $\delta + 1$. Another key observation about δ-order is that the number of a node's neighbours that may follow the node in the order is minimized and equal to δ. In other words, there is no order that provides a lower limit to the number of a node's higher ranked neighbours.

Arboricity α is the minimum number of edge-disjoint forests that together would cover all edges of the graph. We explicitly do not mean spanning forests because those only need to cover nodes. Like δ, this number is increased by highly interconnected subgraphs. Actually, both indicators are closely related and can at most differ by a constant factor:

$$\frac{\delta+1}{2} \leq \alpha \leq \delta$$

1.2.3 Sparse Graphs

One way to get a more specific time complexity is to narrow its validity down to a certain graph type. Planar graphs, for example, cannot have a degeneracy greater than 5 [26]. For planar- and low arboricity graphs, the number of cliques is in $O(n)$, and there even exist algorithms that list all MCs in $O(n)$ [8].

A more general approach to acknowledge how the complexity of our problem depends on graph density is to use density indicators as complexity parameters. Social networks, for instance, have a low degeneracy, as shown empirically by Eppstein and Spiro [12], which also limits maximum clique size σ. CLIQUE is widely believed to be fixed-parameter intractable for σ, meaning that probably no function f exists so that it could be solved in $f(\sigma)n^{O(1)}$ [9, 16]. So, most parameterized complexity bounds that have been found still have some exponent depend on σ or n.

So far, we have always described complexity in terms of the input size n. It is also possible to denote it with respect to the output, thereby directly scaling with the analyzed structural property instead of indirectly hinting to it through fixed parameters. Output sensitive complexity is specified as the time $O(f(\mu))$ that the algorithm needs to find all μ MCs. Since the maximum number of MCs decreases with decreasing density, output sensitive complexity bounds are especially meaningful for sparse graphs. Yet another way to declare complexity is the polynomial delay $O(f(n))$, which is the time that the algorithm needs to find one MC depending on the graph size.

1.3 Related Work

The term "clique" was introduced to our context in 1949 by Luce and Perry [29], who analyzed social networks from the perspective of social science. In 1957, the first algorithm for listing all MCs emerged from the same area [18].

Pardalos [35], Bomze et al. [3] and Harley [19] conducted surveys on how MC listing algorithms were further researched. We may identify two main streams. One is based on the algorithm that Augustson and Minker introduced in 1970 [1], which provides output sensitive complexity bounds. The other one builds on the famous algorithm presented by Bron and Kerbosch in 1973, who specified complexity in terms of the size of the input graph [6]. We will now give a brief overview on the development of both traditions.

Augustson and Minker [1] proposed the first notable approach to specify the complexity of MC enumeration. They proofed that their algorithm's runtime is in $O(\mu^2)$. Still, it is empirically slower than many following algorithms.

In 1977, a significant milestone was achieved by Tsukiyama et al. [39]. Their algorithm is quite comprehensible and has undergone many modifications by other authors. It is based on the the vertex sequence method of Augustson and Minker but is also inspired by Bron and Kerbosch. Its worst case time is in $O(nm\mu)$, worst case storage in $O(n+m)$ and polynomial delay in $O(nm)$.

Most other output sensitive algorithms have their roots in Tsukiyama and challenge its empirical performance. Loukakis et al. [28, 27] not only ensure lexicographic output of MCs but also claim to be even faster than Bron-Kerbosch. Chiba and Nishizeki [8] improved the worst case time by replacing n with α: $O(am\mu)$. Also building on Tsukiyama but using fast matrix multiplication, Makino and Uno [30] applied the maximum degree Δ as complexity parameter: $O(\Delta^4\mu)$. Their algorithm has been considered to be especially fast on sparse graphs.

Despite empirical improvements and interesting theoretic time bounds, algorithms of the Tsukiyama tradition do not compete well against those descending from Bron-Kerbosch. Tomita et al. clearly fortified that fact with their landmark publication in 2006 [38]. What they proposed is still the most widely used MC listing algorithm in practice. It is easy to implement and empirically very fast. Later on in this work, we will explain and benchmark Bron-Kerbosch and Tomita together with the algorithm of Eppstein et al. [11, 13], which is a modification of Tomita. For further research on algorithms based on Bron-Kerbosch and Tomita, we strongly suggest the comparison of Bron-Kerbosch variants by Johnston [22] as well as the clarifying note by Cazals and Karande [7].

1.4 About this Work

Our effort is motivated by the algorithm that Eppstein et al. recently introduced and evaluated [11, 13]. Because their method, as well as its predecessors, is mainly focused on theoretic aspects while no systematic empirical study has been done that would incorporate more algorithm variants, we want to find out how combinations of different methods compete under clean comparable conditions. Also, this work is geared to real world graphs, which are typically large and sparse. We imagine the algorithms being applied to social networks, for example. Our goal is to systematically analyze how they perform on different types of graphs and explain their behaviour. To be able to really operationalize graph types, we generate MC related parameterized random graphs.

The following Chapter will explain the selected algorithms, their modifications and implementation in detail. The synthetic graphs, as well as the general framework and output of our benchmark, will be discussed in Chapter 3. We conclude our work in Chapter 4, where we summarize the gained insights, allude to constraints and give an outlook on possible future work.

Chapter 2

Algorithms and Implementation

2.1 Bron and Kerbosch

It is crucial to understand the working principle of the Bron-Kerbosch algorithm [6] because it is the foundation of all variants that we will benchmark and discuss. To meet its importance, we will explain this algorithm in detail. Luckily, it is really quite simple to implement. As Algorithm (1) shows, the difficulty lies in its recursive nature rather than its definition.

Algorithm 1 Bron-Kerbosch (as outlined by Eppstein and Strash [11])

Input Variables: P, R, X

1: **if** $P \cup X = \emptyset$ **then**
2: **print** R
3: **end if**
4: **for all** $v \in P$ **do**
5: Bron-Kerbosch$(P \cap \Gamma(v), R \cup \{v\}, X \cap \Gamma(v))$
6: $P \leftarrow P \setminus \{v\}$
7: $X \leftarrow X \cup \{v\}$
8: **end for**

Let's start at the beginning: The input variables are sets of nodes. R is most important, it always contains a clique. The function call's job is to report all MCs that contain R. It does that in two basic steps. First, it checks if R itself is maximal. In that case R is printed out (lines 1 - 3). Only if R is not maximal, does the function call reach its second step (lines 4 - 8). Here, we know that there exist nodes v that are not in R but together

with R would form a clique $R \cup \{v\}$. In other words, these nodes are in the neighbourhood of our current clique: $v \in \Gamma(R)$.

Definition 4. *The neighbourhood* Γ *of a vertex* $u \in V$ *(set of vertices* $U \subseteq V$*) consists of all vertices that are connected to* u *(all vertices in* U*):*

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

$$\Gamma(U) = \{v \in V | U \subseteq \Gamma(v)\}$$

$|\Gamma(u)|$ *is the degree of* u*.*

For each of the cliques $R \cup \{v\}$, a recursive call is invoked to report all MCs containing $R \cup \{v\}$. So, with each recursion level, the size of R increases by 1 while the number of nodes in the neighbourhood of R cannot increase and will most likely decrease. It is important to understand that one call, through its recursion, considers all cliques containing R and therefore really reports all MCs that contain R.

The previous paragraph may have provoked this question: How does the function call know which nodes are in $\Gamma(R)$? As one might have guessed from Algorithm (1), the other two parameters provide for that since they constitute a partition of the neighbourhood: $X \cap P = \emptyset$ and $X \cup P = \Gamma(R)$. Note how the neighbourhood of R is intersected with the neighbours of the added node v for each recursive call (line 5).

What still needs to be explained is why the function call, in its second step, only looks at the neighbours in P while $P \cup X$ is the whole neighbourhood of R. Imagine, for a moment, the function call would actually consider all nodes from the neighbourhood as possible extensions of R. On the next recursion level, all considered cliques would contain one more node, and they would all be different because the one added node would be different. Yet another level deeper in the recursion tree, another node is added. On this level, two function calls for the same clique could be executed because they could have added the same two nodes to R if they did it in opposite orders. To prevent MCs from being reported several times, the algorithm keeps track of which nodes have already been added to R in previous calls higher in the recursion tree. Only nodes in P potentially lead to new MCs. After each invocation of a recursive call, the node v that was added to R is moved from P to X because we know that every MC containing $R \cup \{v\}$ has already been found (lines 6, 7). This way, each MC is reported exactly once.

We start the whole algorithm by calling Bron-Kerbosch$(V, \emptyset, \emptyset)$. It means that initially every node is a potential extension of the empty clique R, and no node has been moved to X, yet. If we define this initial call to be on

recursion level 0, we can easily deduct that the cliques considered on level r are exactly the cliques of size r.

To avoid dispensable calls that would be made when the selected node has no neighbours in P at all, we rearranged the algorithm so that it tests the termination condition before making a recursive call. Algorithm (2) better depicts our implementation. In spite of avoiding some recursion depth, it does exactly the same as Algorithm (1).

Algorithm 2 Bron-Kerbosch

Input Variables: P, R, X

```
1: for all v ∈ P do
2:    P ← P \ {v}
3:    if P ∩ Γ(v) = ∅ then
4:       if X ∩ Γ(v) = ∅ then
5:          print  R ∪ {v}
6:       end if
7:    else
8:       Bron-Kerbosch*(P ∩ Γ(v), R ∪ {v}, X ∩ Γ(v))
9:    end if
10:   X ← X ∪ {v}
11: end for
```

Bron and Kerbosch proved that the worst case storage requirement of their algorithm is $O(\frac{1}{2}n(n+3))$. They did not specify its time complexity. Their experiments suggest that time consumption per MC is almost constant – independent of the graph size. This observation was made on tiny graphs and is disputable from today's perspective.

2.2 Tomita et al.

2.2.1 The Algorithm

In 2006, Tomita et al. [38] added a technique to the Bron-Kerbosch algorithm that heavily prunes the recursion tree. It is called "Pivoting". A function call reduces the number of its recursive calls by letting its loop skip all neighbours of a so called pivot node. To understand why still all MCs are being reported, consider pivot node u and its neighbourhood $\Gamma(u)$. A clique that contains only neighbours of u cannot be maximal because we could extend it with u. So, every MC must contain at least one node that isn't a neighbour of u (u isn't a neighbour of itself). Since a recursive call for $\{v\} \cup R$ reports all

remaining MCs containing $\{v\} \cup R$, we can skip all recursive calls in which v would be a neighbour of u. Tomita et al. choose their pivot node so that it has the maximum number of neighbours in P and therefore allows to skip the maximum number of recursive calls (Algorithm 3, line 1).

Algorithm 3 Tomita[2]

Input Variables: P, R, X

1: choose $u \in P \cup X$ so that $|P \cap \Gamma(u)| = \max_{v \in P \cup X}(|P \cap \Gamma(v)|)$
2: **for all** $v \in P \setminus \Gamma(u)$ **do**
3: $P \leftarrow P \setminus \{v\}$
4: **if** $P \cap \Gamma(v) = \emptyset$ **then**
5: **if** $X \cap \Gamma(v) = \emptyset$ **then**
6: **print** $R \cup \{v\}$
7: **end if**
8: **else**
9: Tomita$^2(P \cap \Gamma(v), R \cup \{v\}, X \cap \Gamma(v))$
10: **end if**
11: $X \leftarrow X \cup \{v\}$
12: **end for**

2.2.2 Complexity

One important contribution of Tomita et al. is that, with the help of their pivoting strategy, they were able to proof the worst case time bound of the algorithm to be $O(3^{n/3})$. Considering the finding of Moon and Moser that $3^{n/3}$ is actually the maximum number of MCs for a graph with n vertices, the Tomita algorithm can be said to be worst case time optimal, meaning that for a worst case number of MCs the algorithm's time consumption is linear in this number.

The authors denote the time complexity of one call, without its recursive calls but including the selection of u, to be in $O(|P \cup X|^2)$ ([38], chapter 4, definition 3). Darren Strash, a co-author of David Eppstein in [11, 13], indicated that the selection of u alone is in $O((|P| + |X|)|P|)$ [37].

To be able to proof these general time bounds, Tomita and ELS must keep track of a dynamic graph that is passed down and adapted through recursion to provide recursive calls with parameters X and P. Our implementation satisfies the same worst case time bounds if we assume our neighbour tests via hash maps to be in $O(1)$, like they would be with an adjacency matrix. There are several reasons why we make that assumption in the following:

- We are dealing with sparse graphs, where the average node degree can be considered a constant.

- Hash maps with a small, previously known maximum capacity (Δ) and small utilization ($\bar{d}$ elements) are especially good at providing constant access.

- Fast neighbour tests are further supported by another optimization. The neighbour hash map of each node only stores those neighbours that follow the node in storage order. When testing two nodes for neighbourship, we use the hash map of the node with the lower index, thereby shrinking worst case time of neighbour tests by a constant factor.

- We intend to classify algorithms by their empirical performance in practice and are more interested in the comparability of implementation standards and good average case complexities.

- If a practical application is time critical and, by any means, needs a bound to its runtime, data structures can be adjusted without interfering the algorithm in principle.

One might still argue that, although a neighbour test can be considered to be in $O(1)$, it might still be slower by some constant factor, and that thereby the measured runtimes may be dominated by the total number of neighbourship tests. We consider that true and not true:

- Tomita without an adjacency matrix might be a little slower. What is relevant, though, is that we use the best Tomita implementation applicable to large graphs because doing so meets the precondition of this work and is necessary to its consistency.

- The neighbourship test is the core operation about which all the loops of the algorithm rotate. What one function call (without its recursion) has to do is to extend clique R with all nodes $v \in P$. R is extended with v by testing which neighbours of v are in $P \cup X$. So, the total number of tests characterizes the complexity of the algorithm, anyway. Neighbourship tests are really the atoms of which that complexity is constituted. The whole point of improving Bron-Kerbosch is to reduce the number of neighbourship tests. The longer one test takes, the better can we measure the success of different approaches.

2.2.3 Variants

Aside from the theoretical concerns discussed in the previous section, "quadratic pivoting" seems quite expensive, anyway. So, we further implemented two simplifications of Tomita[2]:

- Tomita[1] (linear pivoting) chooses $u \in P \cup X$ so that $|\Gamma(u)| = \max_{v \in P \cup X}(|\Gamma(v)|)$ as a heuristic to maximize $|P \cap \Gamma(u)|$. Its runtime is in $O(|P| + |X|)$.
- Tomita[0] (constant pivoting) chooses $u \in P \cup X$ without any constraint. Its runtime is in $O(1)$.

Note that these variants only differ in the inner calls because, in the outer call, Tomita[2] is as cheap as Tomita[0]. Since all nodes are in P at the beginning of the outer call, we just have to choose the one with the highest degree there. We can remember that node from the initialization step, so pivoting in the outer call is always in $O(1)$.

It is remarkable that the authors, indeed, evaluate their algorithm empirically but do not compare it to the one it is based on. We couldn't find empirical verification that their original pivoting strategy is actually faster than the Bron-Kerbosch algorithm. Anyhow, the experiments of Koch [24] indicate at least that simplified pivoting strategies are supposedly faster than not to use pivoting at all. We hope to shed some light on the performance issue in the evaluation chapter.

2.3 Eppstein et al.

2.3.1 The Algorithm

The latest algorithm that we analyse in this work was introduced by Eppstein, Löffler and Strash in 2010 [11] and empirically verified in 2011 [13]. We will simply call it "ELS". It is based on the fact that the order in which nodes are processed within the initial call effects the maximum size of those sets that are passed to the recursive calls as parameter P, thereby also setting that limit for all recursion steps. The idea is, now, that limiting $|P|$ would improve performance, especially on sparse graphs, where the limit might be brought down to $O(1)$. We will have to give a more in-depth explanation of the performance effects, when looking at the results in Chapter 3.

Algorithm (4) hardly looks any different from Algorithm (3). The only modification is the introduction of a node order $v_1 ... v_n$, that is applied in the outer call (lines 2, 3). A variant of the Tomita algorithm is then invoked for

Algorithm 4 ELS^t

Input Variables: P, X

```
1: choose u ∈ P so that |Γ(u)| = max_{v∈P}(|Γ(v)|)
2: for i = 1 → n do
3:    if v_i ∉ Γ(u) then
4:       P ← P \ {v_i}
5:       if P ∩ Γ(v) = ∅ then
6:          if X ∩ Γ(v) = ∅ then
7:             print  R ∪ {v}
8:          end if
9:       else
10:         Tomita^t(P ∩ Γ(v_i), {v_i}, X ∩ Γ(v_i))
11:      end if
12:      X ← X ∪ {v_i}
13:   end if
14: end for
```

inner calls (line 10). The important difference is really that we now control the node order, which was previously random. In fact, the original storage order of nodes gets shuffled for all our algorithms.

What Listing (4) does not show is how nodes are sorted during the initialization step, so the crucial part of ELS happens before recursion even starts. With δ-order, the number of a node's higher ranked neighbours is limited by δ, as we explained in the introduction. The algorithm exploits that property, when it invokes recursive calls in line 10. Since all nodes in P have a higher rank than i, the intersection $P \cap \Gamma(v_i)$, which becomes parameter P within the recursive call, cannot contain more than δ nodes.

2.3.2 Complexity

Similar to the theoretic contribution of Tomita et al., the modification Eppstein and Strash introduced allowed them to proof a tighter worst case time bound, which is $O(\delta n 3^{\delta/3})$, thereby revealing that listing al maximal cliques is fixed parameter tractable for parameter δ. They consider it the "near-optimal worst case time bound for graphs with low degeneracy" [13] and also show that for degeneracy δ, the maximum number of MCs is $(n - \delta)3^{\delta/3}$.

The ELS algorithm, as presented by Eppstein et al. [13], uses a dynamic graph data structure $H_{P,X}$ to represent subproblems and relevant adjacencies for pivoting. We will now briefly compare our implementation of ELS^2 against the one of Eppstein et al., with regard to how the complexities of

outer- and inner call depend on function arguments and other parameters. Note that Δ is the maximum degree.

	Eppstein et al.	our implementation
outer call	$O(n\delta\Delta)$	$O(n\Delta)$
inner call	$O(\|P\|^2(\|P\| + \|X\|))$	$O(\|P\|(\|P\| + \|X\|))$

In the outer call, Eppstein et al. go through all n vertices v_i. For each v_i, they need time $\delta\Delta$ to build the dynamic graph that provides the intersections to recursive calls. See Lemma (3) in [11], where their specification $O(|P| + |X|)$ translates to $O(|\Gamma(v_i)|)$ in our context. The complexity of their inner call is denoted in Lemma (4).

In our implementation of ELS^2, the outer call also goes through all n vertices v_i, but for each v_i we only have to check whether its up to Δ neighbours $w_j \in \Gamma(v_i)$ are in X. In the outer call, this membership test is in $O(1)$ because $w_j \in X \iff j < i$, with the index being the node's position in processing order. In our inner call, we check for each $v \in P$, which neighbours $w \in P \cup X$ of the current clique are also neighbours of v. We, however, don't need to iterate through the neighbours of v.

As we reasoned in the section about the Tomita algorithm, our pivoting is also in $O(|P|(|P| + |X|))$, but with δ-order, that reasoning can be fortified even more. We mentioned that neighbour hash maps store only higher ranked neighbours. To check whether two nodes are connected, we only need to request the node w with the higher rank from the neighbour hash map of the other node v. Now that v can at most have δ neighbours following it in the order, the limit for the hash map size drops from Δ to δ, which is a huge improvement. To do better than that, you need to use an adjacency matrix, which Eppstein et al. don't. Δ is usually much bigger than δ. Even in sparse graphs, there can be a central node whose degree is in $O(n)$. Setting δ as the maximum capacity for those Java hash maps may further save main memory.

2.3.3 Assumptions

The δ-order concept is useful to theoretically prove certain complexity bounds with respect to δ, but still, we need to find out how it compares to other algorithms in practice, partly because Eppstein et al. build on some explicit and implicit assumptions we may not want to adopt completely. We will now discuss two of them and how they were translated into our implementation.

Assumption 1: Tomita needs $O(n^2)$ storage space

The authors proclaim it as their goal to "rival the speed of Tomita while having linear storage cost". But what if Tomita can already do that in our context? On one hand, they assume that the Tomita algorithm needs quadratic storage space because it relies on an adjacency matrix. On the other hand, they implemented a version of Tomita that makes use of adjacency lists. They state that this "simple" version would be slower because it cannot test the adjacency of two nodes in constant time. Obviously, they don't consider adjacency hash maps as an addition and by not doing so confine the fairness of the empirical comparison with Tomita. Even so, for nearly half of the real world graphs on which Eppstein and Strash measure runtimes, at least one version of Tomita is faster than both versions of ELS. They could not run the matrix variant of Tomita at all on the 13 largest graphs of their real world data set. Since we aim for such large networks, whose adjacency matrices might not fit into main memory, our implementation only uses data structures that enable linear space complexity.

Assumption 2: When δ-order is applied in the outer call, no pivoting can be done there

What Eppstein et al. propose to do in the outer call is nothing else than the Bron-Kerbosch algorithm with a specified node order. They explicitly avoid pivoting there without giving a reason for that. Actually, pivoting would have the lowest cost and strongest effect there because choosing the pivot node is in $O(1)$, and we are on the highest level of recursion. To discard pivoting on this level, causes problems for the evaluation as well because the comparison between Tomita and ELS can only reveal the potential of δ-order if it comes as an addition to- instead of a replacement of pivoting, especially when the implementation of Tomita might be slower than plain old Bron-Kerbosch in some cases.

The authors may have prevented pivoting in the outer call for the following reason: If we skip some nodes, when processing all of them in δ-order, the skipped ones must remain in P, making it impossible to decide on a node's membership in P or X based on its δ-order position. Building the intersections (Algorithm (4), line 10) would be slower by a constant factor and involve something like a hash map for X. This argument, however, would apply to the Tomita algorithm as well because there the nodes are also processed in some order, even though that order may not be specified.

It remains unclear why Eppstein et al. substantially altered the Tomita algorithm, but we found that the benefits of outer call and pivoting don't ex-

clude each other, anyway. Our implementation makes a distinction between degeneracy- and processing order. In processing order, the neighbours of the pivot node (here: node of highest degree) are ranked higher than all others to reflect that they are neither processed nor moved to X. Based on that order, we can still test in constant time whether a neighbour of v is in X. To apply pivoting, we have to test for all $|P|$ nodes whether they belong to the $O(\Delta)$ pivot neighbours (Algorithm (4), line 3). When a neighbour of the pivot node is skipped in δ-order, it just gets pushed to the end of the processing order, thereby avoiding any additional hashing or complexity.

2.3.4 Variants

Persistent δ-Order

Although Eppstein et al. state that δ-order in the outer call can have a positive effect, the authors do not demand to keep it for all inner calls. This would even come very cheap. We just have to order all adjacency lists one time before we start, and δ-order can be kept throughout recursion. In our initial experiments, we compared δ-order against reversed δ-order for those lists, in terms of runtime and loop iterations. We found that it might make a small but significant difference to the performance. That is why we also included ELS variants in the benchmark that order their adjacency lists as an addition. All other algorithms shuffle their neighbour lists during initialization to ensure these lists are not ordered in any systematic manner.

We already optimized pivoting a little. A node cannot have more neighbors in P than its degree. When the maximum number of neighbors in P that has already been found is bigger than the degree of the current node, that node is skipped. This is especially effective for quadratic pivoting, where we avoid counting the current node's neighbours in P.

Now, the neighbour sorting algorithms have their pivoting improved even more. For linear and quadratic pivoting, we additionally demand to go through P and X backwards, when iterating through the whole neighbourhood $P \cup X$ of the current clique. Nodes with higher core numbers tend to have higher degrees. Going from higher to lower degrees leads to more nodes being skipped.

Be aware that, while we can keep X and P ordered for inner calls, X does not necessarily precede P as it does in the processing order of the outer call. This is a consequence of pivoting. We would have to apply the same strategy as in the outer call, but it isn't worth the effort because $|P|$ in the inner calls is limited by δ and quickly shrinks with recursion depth.

Since the nodes skipped by pivoting tend to be a minority among all nodes

in P, and the pivot node and its neighbours tend to have higher degrees, the nodes in X still tend to precede the nodes in P. That is why we start with P, when searching for the pivot node.

The most important optimizations then happen, when iterating through X. As soon as the current node precedes the first node of P, we know that all remaining nodes precede P as well because we iterate through X in reverse δ-order. From that node on, we apply a modified iteration:

- We request the degree of the current node from the hash map which stores only following neighbours because only those can be in P. This leads to more nodes being skipped.

- When a better pivot node is found that has at least δ neighbours in P, iteration stops because the nodes preceding P cannot surpass δ.

Remember Tomita[0], where pivoting is done in $O(1)$ by choosing an arbitrary node from $P \cup X$. What we really want is a pivot node with many neighbours in P. Since P is now in δ-order, we know that the first node in P cannot have more than δ neighbours in P. The last one might be a much better pick because it can have $|P| - 1$ neighbours in P.

Minimal Size

In practical scenarios, we often don't care about the vast amount of small MCs. Instead, we are interested in the largest ones or those that we consider significant. With degeneracy order, it might be possible to speed up the algorithm if we only wanted to find MCs of a minimal size k. Lets call this modification "k-ELS".

All vertices in a MC of size k have core numbers of at least $k-1$. After we have generated the degeneracy order, nodes are sorted by their core number, and through the buckets we have constant access to the position of the first node that has at least core number $k-1$. We would still find all MCs if we started the loop of the outer call at that position because all previous nodes have core numbers lower than $k-1$ and thus cannot be in a MC of minimal size k.

To fully exploit this idea, it is important that all nodes preceding the start position are actually in X, when the outer call starts its loop because we don't need to consider them in recursive steps either. So, we don't put any of these nodes to an unreachable rank in our processing order. Only nodes at our start position and beyond may still be skipped by pivoting and, therefore, be saved from ending up in X. Note that we cannot do something

like using two pivot nodes in the same function call whereas pivoting and this optimization for minimal size k do not constrain each other.

Now, when starting the outer call, P doesn't contain all nodes of the graph anymore, so we cannot choose the best pivot node in constant time. Instead, P contains all nodes from the start position to the end, which makes the optimizatons explained in the previous section about persistent δ-order even more effective here. Of course, all of them are applied to the outer call of our k-ELS implementation.

We will evaluate the possible performance gain through a separate comparison against regular ELS in Chapter 3. Bron-Kerbosch and Tomita can not take advantage of a minimal size k because they retrieve no prior knowledge about the graph structure.

Chapter 3

Evaluation

3.1 Random Graphs

3.1.1 Parameters

To systematically evaluate how each algorithm performs on different types of graphs, we need a parameter space with a small number of strongly decoupled dimensions, in which the various types can be located. The parameters should be decoupled in the sense that each one describes a distinct property of the graph and is not predetermined by the others.

We choose n as our first parameter. It is the main indicator of the sheer size of the problem. In addition to its size, we want to classify the graph's density. We already mentioned the most common measure of graph density, which is the edge probability p. However, p doesn't fit our needs because it quadratically depends on n. In most real graphs, however, the number of edges is rather in $O(n)$ than in $O(n^2)$. In other words, the average number of neighbours is more like a constant that depends on the type of graph. Think of social networks: The average number of friends does not grow endlessly in the way the network is expanding through new memberships. In order to reflect this faithfully, we regard the average node degree as our second parameter:

$$\bar{d} = \frac{2m}{n}$$

Finally, we want to have a measure on how many MCs are in the graph because even with predefined numbers of n and $\bar{d}$, there is a high degree of freedom left for the number of MCs. Obviously, we can't simply use the number of MCs per node because this number would still heavily depend on $\bar{d}$. Our measure of MC frequency has to scale with the maximum possible μ, given n and $\bar{d}$. The literature does not provide a formula for that number, but

we can deduct a tighter upper bound by assuming that there always exists a graph that maximizes μ for the given parameters and has nearly equal node degrees, meaning that the maximum difference between the degrees of two nodes would be 1. We will now briefly explain why we make that assumption.

In a Moon-Moser graph, the number of edges is $\binom{n}{2} - n$. With m growing beyond that, the possible number of MC shrinks again, until we can only have one complete graph with $m = \binom{n}{2}$. That case can be left out of consideration because nearly complete graphs are practically irrelevant and definitely not sparse. We're also less interested in the case where $2 < m \leq n$, because there, μ can be maximized by making one ring out of m edges and m nodes, thereby creating n MCs (counting isolated nodes as well).

Now, if $m = \binom{n}{2} - n$, the graph with maximized μ implies equal node degrees of $n - 3$, and if $m = n$, all node degrees can be 2. So, even if node degrees would temporarily move away from equal distribution, with m shifting from one of both extremes to the other, they start and end being equally distributed, and the deducted upper bound would still roughly scale with the possible number of MCs, paying respect to graph density.

However, we conjecture that maximizing the number of MCs for given n and m would, indeed, always lead to nearly equal node degrees. The following example is just one way to approach the evidence. Two different MCs A and B can share many nodes, but each of both has at least one node that is not contained in the other MC. Otherwise, one would completely be contained in the other and would, therefore, not be maximal. If A contains one more node than B, it has two nodes v, w that are not in B. Since it would only need one such node v to distinguish itself from B, the edges that connect w with all other nodes in A can be considered "waste", with respect to A and B because they do not contribute to the distinction of both MCs. Even if there was a MC C of size $|A|$ that distinguishes itself from A by not containing w, the argument would be shifted to B and C. Another kind of waste can result from nodes that are contained in too many MCs since nodes in $A \cap B$ do not contribute to the distinction of A and B either. In Moon-Moser Graphs, the intersection of all MCs is empty while every node is contained in one third- and every edge in $\frac{1}{9}$ of all MCs. It should now be clearer that the concentration of edges always eliminates some combinatorial freedom to constitute MCs.

Theorem 1. *For n and equal node degrees $\bar{d} = 2m/n$ with $n < m \leq \binom{n}{2} - n$,*

$$\frac{n3^{\bar{d}/3}}{\bar{d}/3 + 1}$$

is an upper bound to the number of MCs.

Proof. Because there are at most as many edges as in a Moon-Moser graph, m limits the number of MCs, and $\bar{d}$ limits the number of MCs per node.

1. We can assume the degree of every node to be $\bar{d}$.

2. By definition, all MCs containing node v are subsets of $\{v\} \cup \Gamma(v)$. From (1) and the findings of Moon and Moser follows that v can at most be in $3^{\bar{d}/3}$ different MCs.

3. If node v is in its maximum number of MCs, these MCs are of size $\bar{d}/3 + 1$ because the neighbours of v would constitute a Moon-Moser graph containing $\bar{d}/3$ triplets.

4. From (2) and (3) follows that the number of MCs per node is bounded above by
$$\frac{3^{\bar{d}/3}}{\bar{d}/3 + 1}$$
.

□

Consider the Moon-Moser graph of Figure (1.1) again. If we would force every node to have 6 neighbours instead of 9, the number of MCs would be bounded by 36 instead of 81, so the theorem really gives a better upper bound with respect to its assumption.

Now that we can estimate the maximum number of MCs, we can normalize the frequency of MCs and express it as the quantitative portion of possible MCs that is realized in a graph:

$$q = \frac{\mu_2(\bar{d}/3 + 1)}{n3^{\bar{d}/3}}$$

Note that for this measure, we ignore MCs that only contain one isolated node because the performance processing these artificial MCs would only depend on n. Another reason is that it helps our graph generation algorithm dealing with low values of $\bar{d}$. Also be aware that this is not a MC probability, by any means, because with $q < 1$, the graph may contain certain MCs that it cannot contain with $q = 1$.

We determined the parameters of the real world graphs on which Eppstein and Strash ran their experiments and chose a realistic interval for each of them:

- $n \in [500 \ldots 10000]$

- $\bar{d} \in [2.5 \ldots 11.5]$
- $q \in [0.03 \ldots 0.3]$

These limits define our "parameter cube", in which we measure the performance of all algorithm variants at regularly distributed sample points.

3.1.2 Generation

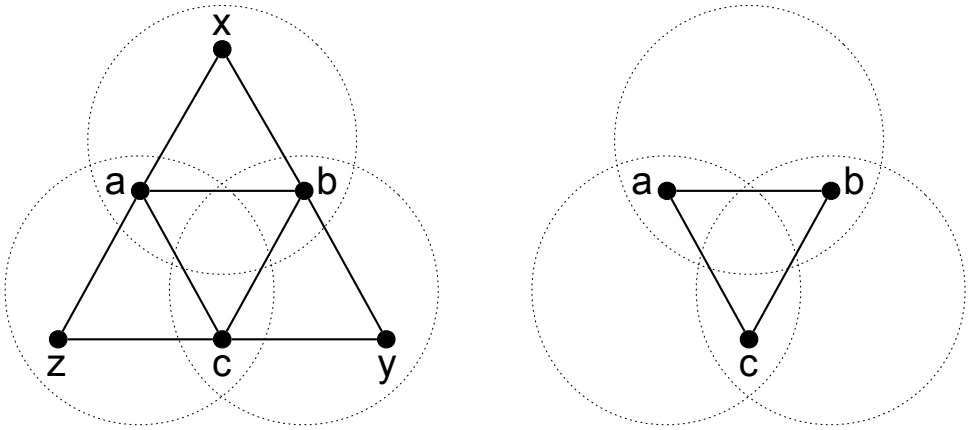
It is, of course, no big deal to generate a random graph for predefined values of n and $\bar{d}$. What makes our generation algorithm a bit trickier is the need to additionally control the number of MCs in order to satisfy parameter q:

$$\mu_2 = \frac{qn3^{\bar{d}/3}}{\bar{d}/3 + 1}$$

The idea of our approach is to randomly assign nodes to MCs, while keeping track of how many edges are added with each such assignment (knowing that the added node might already be connected to some nodes in the MC). To ensure every node is assigned to at least one MC, and every MC is assigned at least one node, we first do exactly that: assign each node to a MC and then one node to each empty MC. This procedure gives perfect control over n and m but often leads to a different number of MCs than intended, so let's call the set of nodes that we assign to a MC a "potential MC" (PMC).

Of course, all PMCs end up being cliques because their nodes get connected, but neither are they guaranteed to become maximal, nor are they guaranteed to become the only MCs generated. Consider both graphs of Figure 3.1. Each PMC is enclosed by a dotted circle. In graph (a), the set $\{a, b, c\}$ constitutes an unintended additional MC whereas in graph (b), the PMCs merge into one MC $\{a, b, c\}$, thereby producing two MCs less than intended.

The good news about the number of PMCs is that it still influences the number of actual MCs. The more MCs we attempt to create, the more we get, but there is no simple formula for this relation. First of all, both other parameters n and $\bar{d}$ have a strong impact on how many MCs we get. Second, the more MCs we want to create, the more PMCs we need for each one of them. Our algorithm deals with these issues by iteratively converging to the desired number of MCs, meaning that each iteration generates a graph, counts MCs and then makes a better guess on how many PMCs are needed. When the error decreases to 1% or lower, the graph is accepted and iteration stops.



(a) Additional MC $\{a, b, c\}$ (b) PMCs merge into one MC

Figure 3.1: Potential MCs (dotted circles) don't equal the resulting MCs

In order to benchmark many parameter points in reasonable time, we had to optimize this error minimization in several ways, but these aspects as well as implementation details do not effect the principle or result of the algorithm, so we don't further elaborate on them.

3.2 The Testbed

All implementation and evaluation was done on a mid 2011 Mac Mini with a 2.3 GHz Intel Core i5 processor and 8 GB main memory. Algorithms and benchmark were implemented and run in the graph drawing and analysis software Visone [5], using Java.

We benchmarked one parameter point on 3 different graphs that match the parameter values. On each graph, all algorithms were measured. Some algorithms execute very fast for certain parameters. Such short runtimes have a relative high variance. In order to get reliable results, runtime must be accumulated over many repeated executions.

As Boyer [4] explains in a web article, the optimization behaviour of the JVM is hard to predict and can heavily interfere with benchmarking in Java. Inspired by that article, we took some precautions, and though we had to be careful not to elongate our benchmark runtime too much, we found that the quality of our measurements heavily improved:

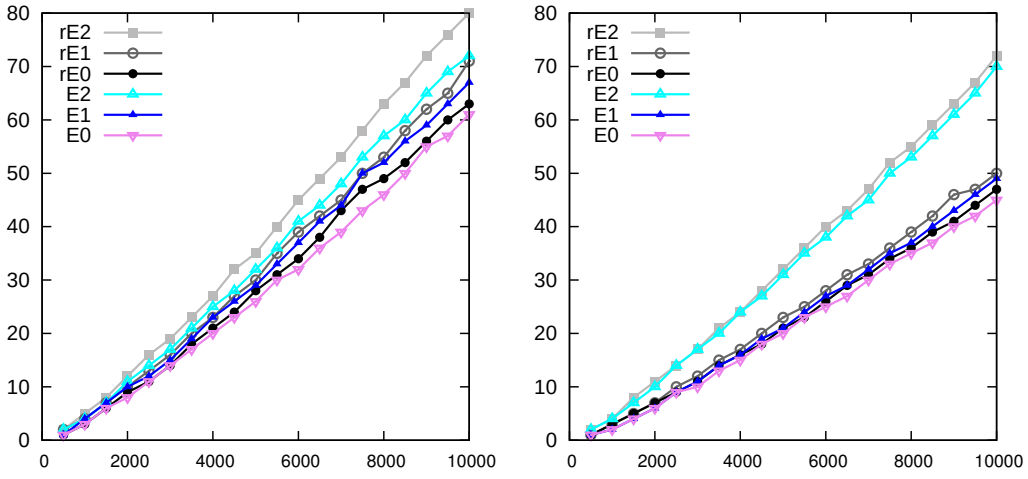
- Before the actual measurement, a warm up phase executes the algorithm in a loop for at least 2 seconds and at most 20 times.

- The measurement phase executes the algorithm in a loop for at least 2 seconds and at most 40 times.
- To calculate the accumulated runtimes, we use the CPU time that the current thread has spent instead of the time that has passed.
- Requesting thread time takes much longer than the standard command to request elapsed time. We use the thread operation to calculate the actual runtime but not for the break condition.
- Initialization times turned out to be too short to be gauged, so the initialization step is excluded from the measurement. That helps with the JVM as well because if only the pure algorithm is repeated in warm up and measurement phase, the JVM can better adjust to it. Anyhow, initialization is mainly concerned with basic data preparation which is almost the same for all algorithms.
- Warm up- and measurement phase are preceded by advising Java to run the garbage collector and complete outstanding finalizations.
- While benchmarking, the detected MCs are not written to any output device like console or hard disk.

3.3 Results

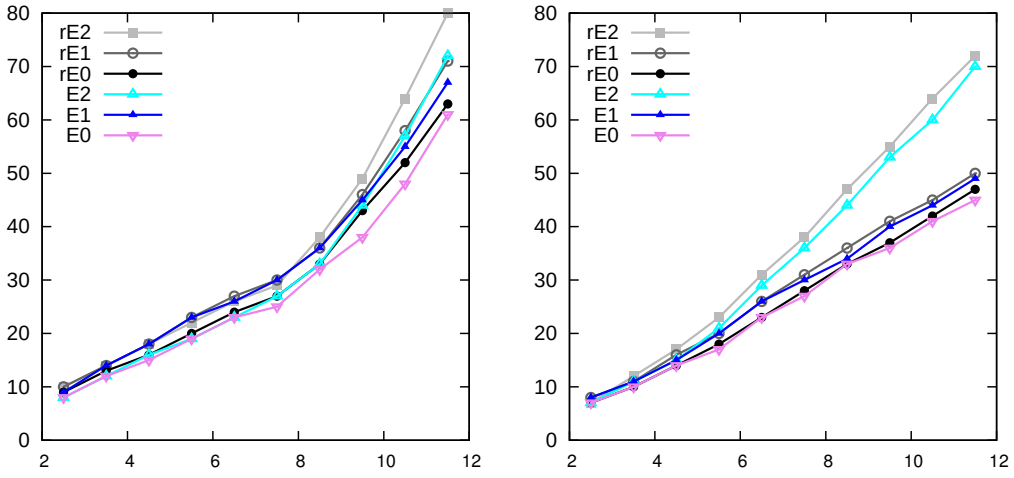
Before we give the starting signal, we have to make sure Eppstein et al. have their most promising horses in the race. Is it worth sorting adjacency lists to keep δ-order for inner calls? That is the question we want to get out of the way in order to prevent the plots from overcrowding. In Figures (3.2), (3.3) and (3.4), the neighbour sorting modification is compared against standard ELS. The labels of the standard variants (random neighbours) start on "r". Note that in the diagrams of this chapter, algorithms are labeled by abbreviations: BK = Bron-Kerbosch, Tomita^2 = T2, ELS^2 = E2 and so on. All runtimes are in milliseconds. On some plots, they where even shorter than in Figure (3.4 a). Those are not shown here because they don't discriminate well.

Persistent δ-order dominates the standard version by a small margin. This is especially obvious at Figure (3.4 b), where graph size and density are both high at all points of measurement. We found that the advantage of persistent δ-order is even more significant on real graphs. In later comparisons, all ELS variants, even k-ELS, will implicitly apply the neighbour sorting optimization, without being additionally labeled as such.



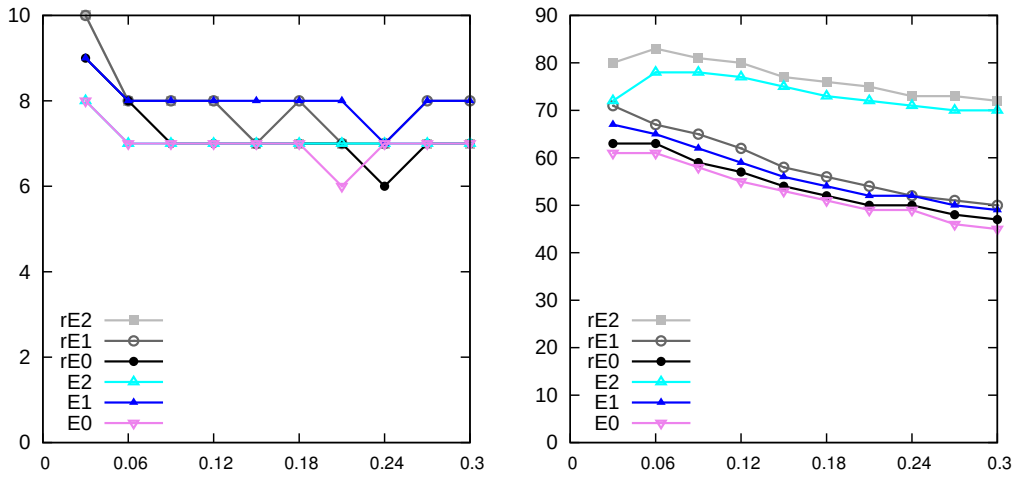
(a) $\bar{d} = 11.5$, $q = 0.03$ (b) $\bar{d} = 11.5$, $q = 0.3$

Figure 3.2: Runtimes over n for $\bar{d} = 11.5$ and different values q



(a) $n = 10,000$, $q = 0.03$ (b) $n = 10,000$, $q = 0.3$

Figure 3.3: Runtimes over $\bar{d}$ for $n = 10,000$ and different values q



(a) $n = 10,000$, $\bar{d} = 2.5$ (b) $n = 10,000$, $\bar{d} = 11.5$

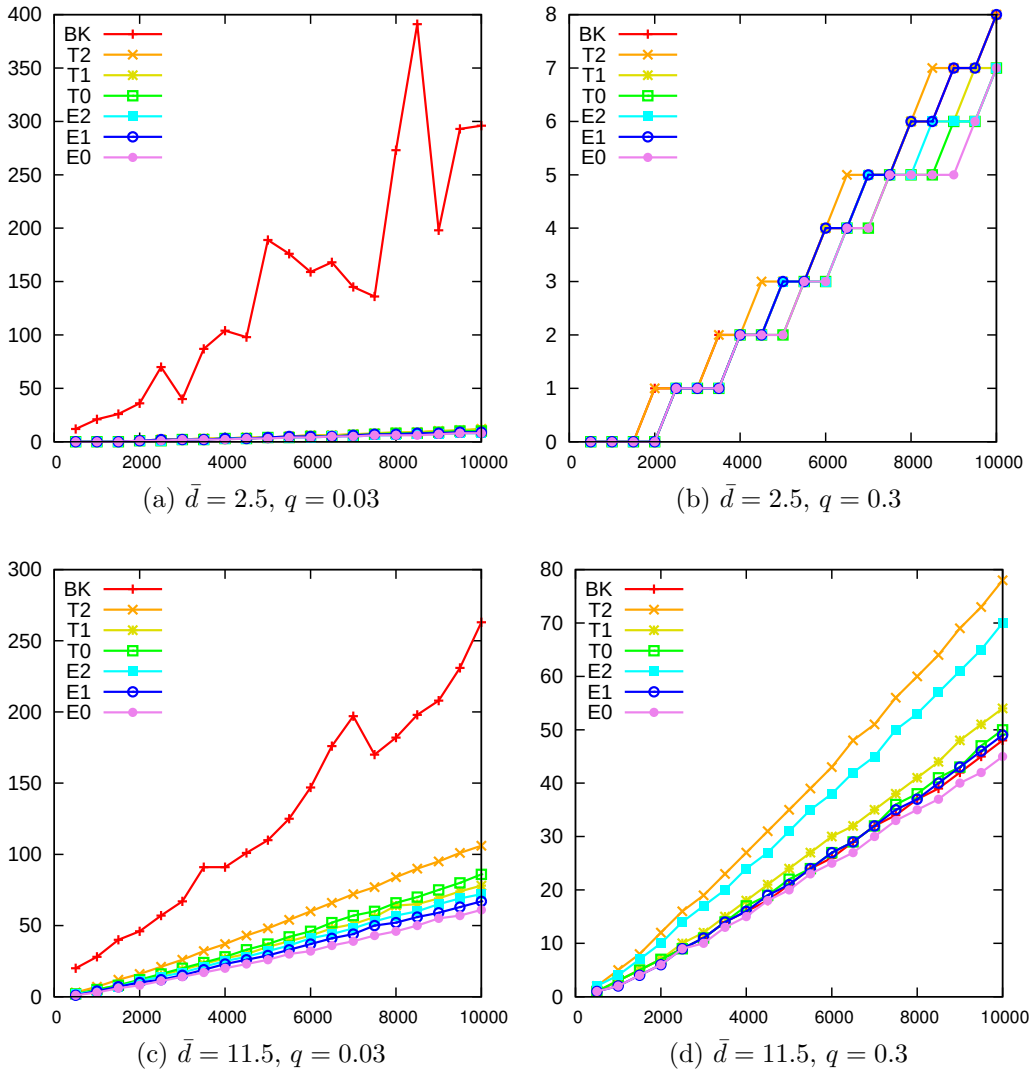
Figure 3.4: Runtimes over q for $n = 10,000$ and different values $\bar{d}$

3.3.1 The Competition

For an overview on how all the algorithms compete, we look at the plots that correspond to the 12 edges of our parameter cube. All other possible 2D plots would be some mixture of those. Figures (3.5), (3.6) and (3.7) illustrate time consumption over parameters n, $\bar{d}$ and q.

What immediately catches the eye is the overall outsider position of Bron-Kerbosch. Not only is BK at times by magnitudes slower, but it is also the algorithm to which parameter q really makes all the difference (Figure (3.7)). For $q > 0.1$, BK even beats quadratic pivoting, making it best suited for graphs in which the edges are expected to be more equally distributed, forming many small MCs. The reason for that is simple: As we discovered before, the size of the unpruned recursion tree corresponds to the number of cliques in the graph. A MC of size n contains $\binom{n}{k}$ cliques of size k. Since the number of cliques grows so rapidly with n, a few large MCs, as induced by low values of q, lead to a much higher number of cliques in the graph and thereby to a much bigger recursion tree. All that being said, Bron-Kerbosch is still dominated by other algorithms. Now, we take it out of the plots because it stretches the scale of some of them out of proportion.

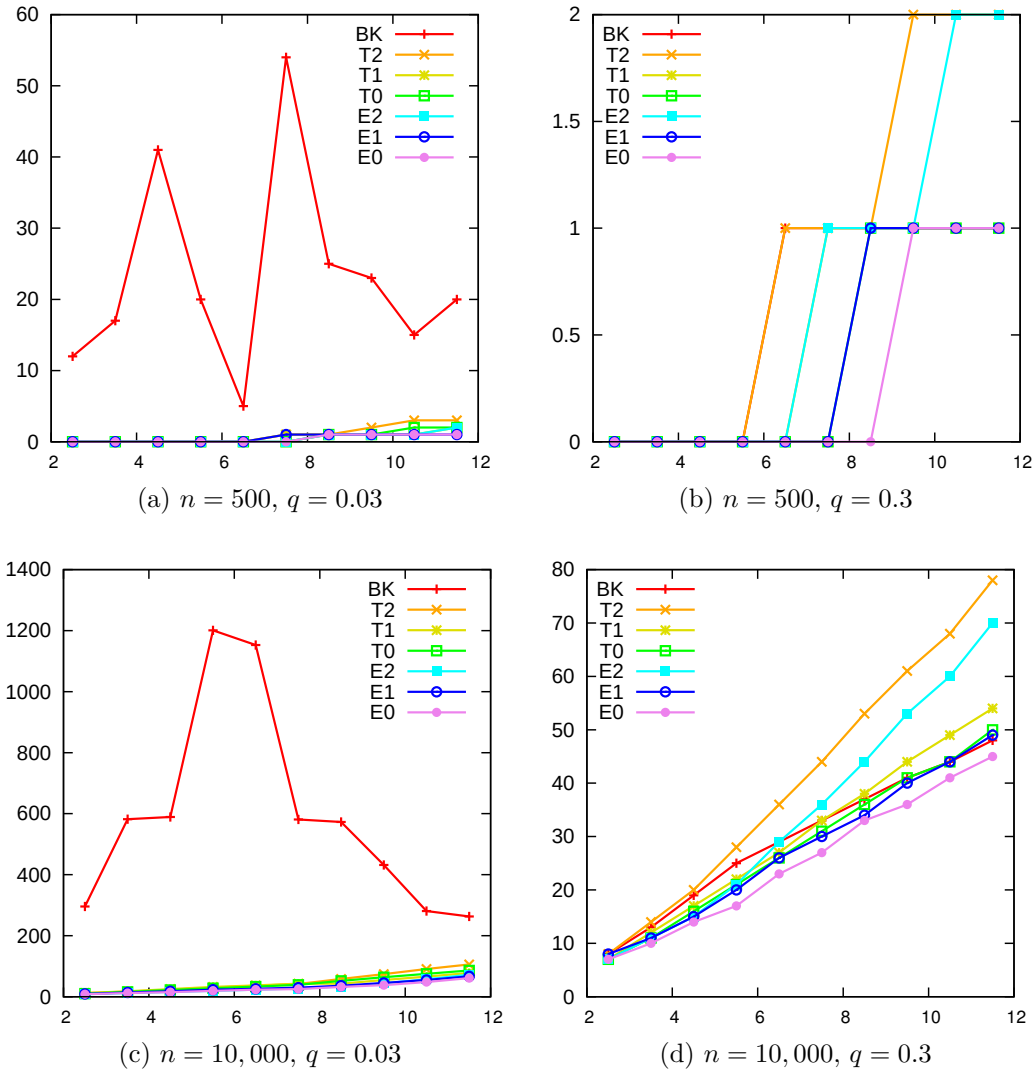
Figures (3.8), (3.9) and (3.10) enable a clearer comparison of the other algorithms. Again, we left out some parameter combinations where runtimes are too short. Apparently, the algorithms are all very fast. On some plots, we can hardly see any difference between them. We have to make this very clear:



(a) $\bar{d} = 2.5$, $q = 0.03$ (b) $\bar{d} = 2.5$, $q = 0.3$

(c) $\bar{d} = 11.5$, $q = 0.03$ (d) $\bar{d} = 11.5$, $q = 0.3$

Figure 3.5: Runtimes over n for different values $\bar{d}$, q



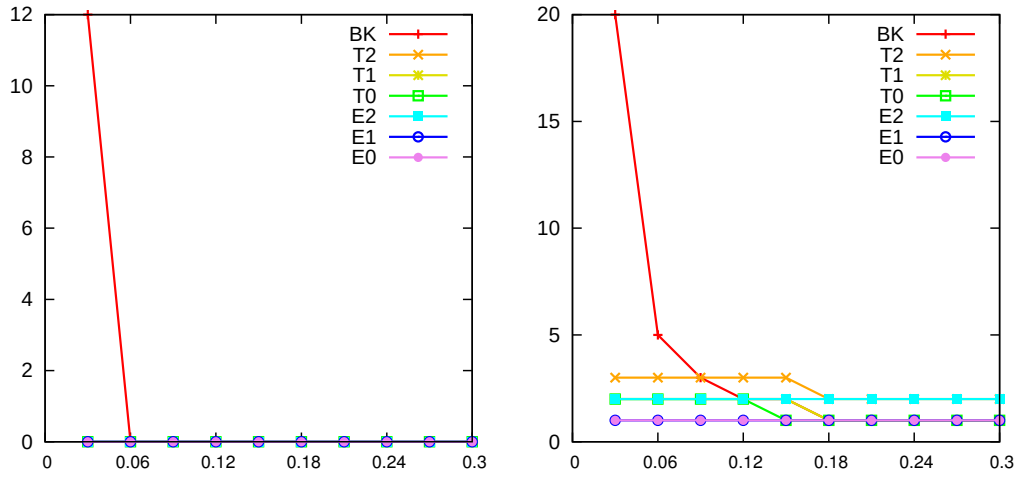
(a) $n = 500$, $q = 0.03$

(b) $n = 500$, $q = 0.3$

(c) $n = 10,000$, $q = 0.03$

(d) $n = 10,000$, $q = 0.3$

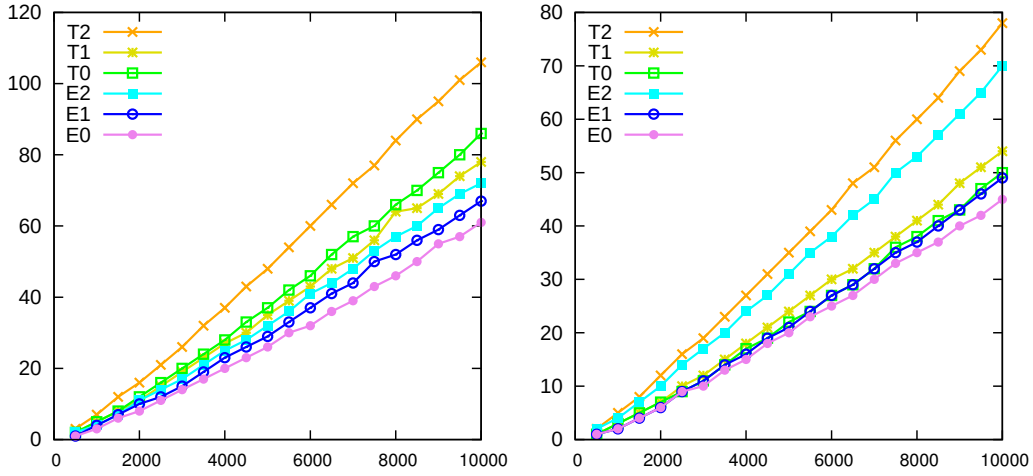
Figure 3.6: Runtimes over $\bar{d}$ for different values n, q



(a) $n = 500$, $\bar{d} = 2.5$ (b) $n = 500$, $\bar{d} = 11.5$

(c) $n = 10,000$, $\bar{d} = 2.5$ (d) $n = 10,000$, $\bar{d} = 11.5$

Figure 3.7: Runtimes over q for different values n, $\bar{d}$



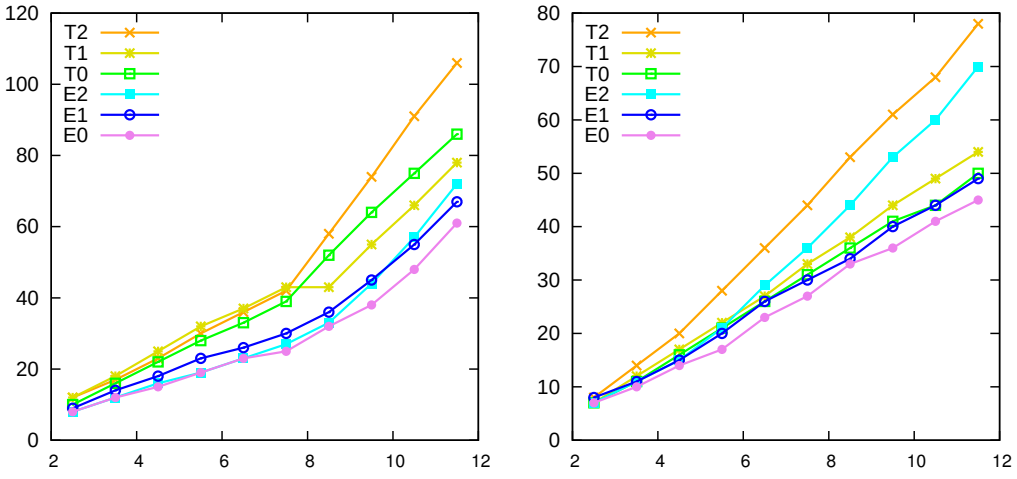
(a) $\bar{d} = 11.5$, $q = 0.03$ (b) $\bar{d} = 11.5$, $q = 0.3$

Figure 3.8: Runtimes over n for different values $\bar{d}$, q

Our measurements on synthetic graphs show that the remaining algorithms all do the trick in about the same time. The visible gradation wouldn't be essential in practice. Yet, there is some information hidden in the data that may be interesting for theoretical assessment.

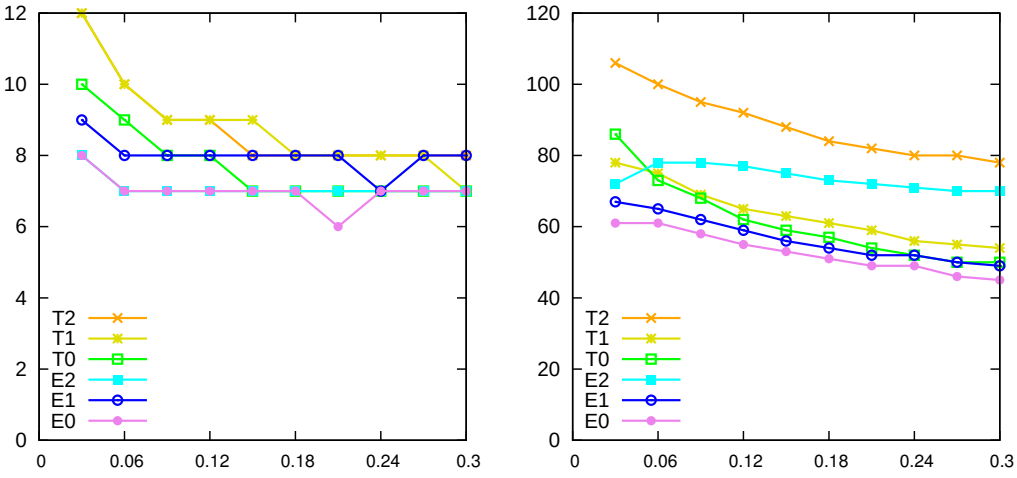
Comparing each ELS^t against its counterpart Tomita^t, we are happy to see some definite coherence: δ-order dominates pure pivoting, at least on our synthetic graphs. Among the ELS variants, the best pivoting strategy is the simplest: ELS^t dominates ELS^{t+1}. For Tomita, it's not quite the same because with increasing $\bar{d}$ and low q, Tomita^0 falls behind Tomita^1. In general, how the pivoting strategies relate to each other is very consistent between Tomita and ELS, which attests further significance to our benchmark.

Another striking revelation is that for high values of $\bar{d}$ and q, the pivoting strategy is even more important than δ-order because there, the quadratic pivoting, as originally proposed by Tomita et al., is dominated by the more straight forward approaches, regardless of the node order. Actually, we have already explained that. It is the same reason why Bron-Kerbosch is so slow for low values of q. Here, q is larger. If q alone was increased, it wouldn't have the same effect. The number of MCs that would have to be built, using the same amount of edges, would just about linearly increase, but since $\bar{d}$ is larger as well, the possible number of MCs rises exponentially. Therefore, the graph must contain so much more MCs that the size of each one of them has to shrink. Figure (3.14) also illustrates that. Thus recursion depth and the actual number of cliques to be considered by an algorithm can drop with



(a) $n = 10,000$, $q = 0.03$ (b) $n = 10,000$, $q = 0.3$

Figure 3.9: Runtimes over $\bar{d}$ for different values n, q



(a) $n = 10,000$, $\bar{d} = 2.5$ (b) $n = 10,000$, $\bar{d} = 11.5$

Figure 3.10: Runtimes over q for different values n, $\bar{d}$

$\bar{d}$ and q growing. Of course, with a less complex recursion tree, pruning it effectively becomes less important. Also consider the sag of performances for $6 \leq \bar{d} \leq 9$ in Figure (3.9 a). It is especially noticeable for quadratic pivoting and corresponds roughly to the peak that the runtime of BK makes in Figure (3.6). What is pure complexity to Bron-Kerbosch seems to be handled notably well by effective pivoting.

Considering the data and algorithms that we discussed so far, ELS^0 would be the dominating choice. Of course, there is still the uncertainty left that the topology of our graphs might not reflect the characteristics of real networks, even with the clustering effect of q, and that this difference might account for different performances in practice. That is why we ran some tests on real graphs, considering their position in the graph parameter space.

3.3.2 Degeneracy Order

We already found that δ-order offers a slightly superior algorithm, but its performance effect still needs to be explained.

Observation

Why do Eppstein and Strash, who exclusively compare ELS^2 against $Tomita^2$, report even stronger effects of δ-order? We previously discussed some problematic assumptions they make and suspected that they might not bring their implementation of $Tomita^2$ to its full empirical potential. Now, we need to discuss another aspect of that. Neither Eppstein and Strash nor Tomita et al. themselves verify that the pivoting strategy originally published by Tomita et al. is not only useful to proof its worst case time bound but also faster than plain old Bron-Kerbosch in practice. Eppstein and Strash just assume that this is the case, so they ascribe all performance gain of their algorithm to the ordering they have added. Now, what if their implementation of $Tomita^2$ was slower than not to use pivoting at all? Then, at least part of the benefit would result from avoiding some of the slower pivoting in the outer call of their implementation of ELS^2. Our benchmark results further support that concern. As we can see in Figures (3.5 d), (3.6 d), (3.7 b) and (3.7 d), Bron-Kerbosch is faster than quadratic pivoting for high values of q.

Eppstein and Strash [13] claim that

> "All other theoretically-fast algorithms for sparse graphs have been shown to be significantly slower than the algorithm of Tomita et al. (Theoretical Computer Science, 2006) in practice,"

referring to the 2006 publication of Tomita et al. [38]. However, in this publication Tomita et al. only compared their algorithm against 3 successors of Tsukiyama et al. [39], which are said to be fast on sparse graphs but according to the results, are all dominated by Tomita[2]. We implemented the algorithm of Tsukiyama et al. ourselves and confirmed by our experiments that it cannot even compete with Bron-Kerbosch and is, indeed, by magnitudes slower. To do a full benchmark of the Tsukiyama algorithm in reasonable time, would have been impossible. That is why we dropped it. The point we want to make here is that the performance effect of δ-order is an interesting phenomenon that demands a fair evaluation and hasn't been sufficiently explained, yet.

Analysis

It is, now, time to investigate how exactly δ-oder brings about its performance gain. Eppstein et al. state that "[...] the order in which the vertices of G are processed by the Bron-Kerbosch algorithm is also important." [11] and explain the advantage of their algorithm by the fact that δ-order limits the size of P:

> "The sets P passed to each of these recursive calls will have at most d elements in them, leading to few recursive calls within each of these outer calls."

They claim to reduce the number of recursive calls by processing the outer call in δ-order:

> "Below the top level of recursion we switch [...] to the pivoting algorithm [...] to **further** control the number of recursive calls."

They do not explain, however, how limiting the sizes of P reduces the sum of all recursive calls, or why it should be faster for other reasons. While it is a fact that pivoting actually reduces the number of calls (whatever that may cost), the same has not been proven for this ordering strategy. The explanation of Eppstein and Strash becomes even more baffling, when we consider the following: One function call runs its main loop as many times as there are nodes in P. As we previously explained, the algorithm in total runs its main loop exactly once for every clique. Its performance should, therefore, mainly depend on the number of cliques, which, of course, doesn't care about node order. When we look at Algorithm (2), we notice that not every loop (clique) results in a recursive call. To quickly estimate that effect, we actually let the algorithm count the number of all recursive calls for δ-order, different random orders and reversed δ-order (without any pivoting) and made two main observations:

1. Node order has a relatively small effect on the number of all calls.

2. Within this small range, δ-order seems to maximize the number of calls while reversed δ-order seems to minimize it, meaning that no random order exceeded these limits.

Note that when discussing the mechanisms of δ-order, we always think of it as applied to the Bron-Kerbosch algorithm instead of Tomita, so pivoting doesn't interfere.

The second observation comes as a bit of a surprise. To understand it, we have to imagine how the recursion tree looks like. Consider the graph of Figure (3.11):

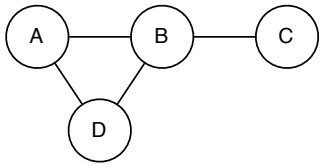


Figure 3.11: Example graph with $\delta = 2$

Figures (3.12) and (3.13) show standard (or random) and degeneracy orientation of the nodes together with the corresponding recursion trees of the algorithm. The trees are, of course, pictured as ordered. With the initial function call being the root on level 0, recursion nodes on level r represent the cliques of size r. For every clique in the graph, there is one node in the recursion tree. Leafs (grey) represent cliques that are considered by the algorithm but do not result in recursive calls while inner nodes represent cliques that actually invoke calls. So, the number of recursion nodes on each level is constant for the graph and independent of the node order.

An empty set R is passed to the initial call on level 0. In other words, by starting the algorithm, we "consider" the empty clique. Therefore, the root box in Figures (3.12) and (3.13) is empty. The set P that is passed to the initial call contains all $n = 4$ nodes of the graph, which is reflected by the children of the root. Inside the initial function call on level 1, the nodes in P are looped in order to consider all cliques of size 1. Here, δ-order comes into play by limiting the sizes of the sets P that are passed to level 2, which limits the number of children that nodes on level 1 can have. In our simple example, the numbers are not actually limited but still less concentrated because in Figure (3.13) only one recursion node on level 1 has two children.

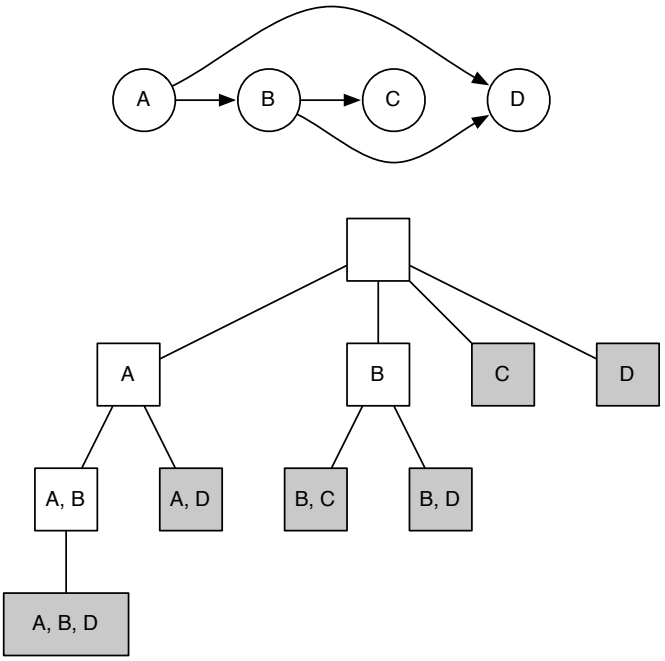


Figure 3.12: Standard orientation and corresponding recursion tree

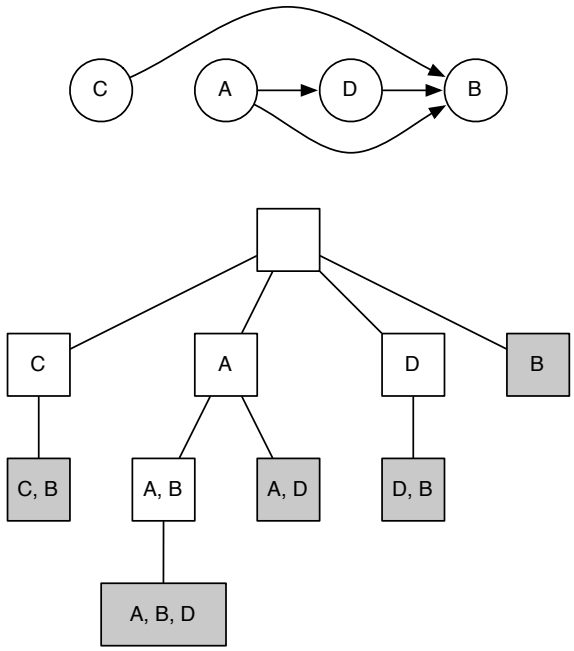


Figure 3.13: Degeneracy orientation and corresponding recursion tree

However, the sum of those sizes must be $m = 4$ because there are always m cliques of size 2 to be considered on level 2. Hence, with δ-order, the m loop iterations on level 2 are distributed to more different function calls, or in other words, the recursion nodes on level 2 have more different parents on level 1. The recursion tree has the same number of nodes but lesser leafs.

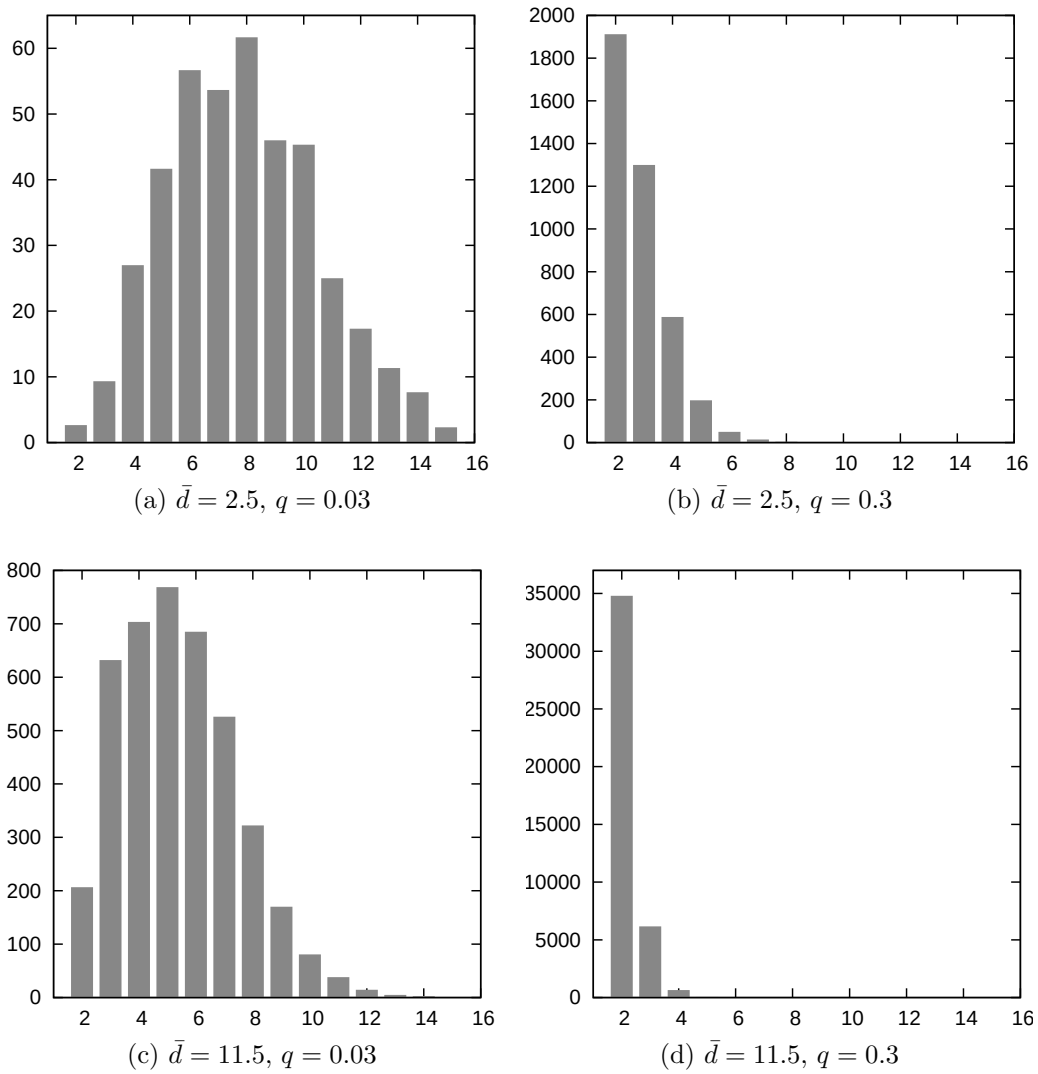
So, if δ-order increases the number of recursive calls, where does the performance gain come from? Remember that, in the implementation by Eppstein and Strash, the processing of one inner call without its recursive calls is in $O(|P|^2(|P| + |X|))$. From the cubic dependence on $|P|$, we can easily deduct that performance must improve if we distribute our constant work load of m recursion nodes on level 2 more equally to more recursive calls invoked from level 1. In our implementation, one such call is only in $O(|P|(|P| + |X|))$, so we benefit less from δ-order. The reason that Eppstein and Strash found their algorithm to be especially useful on sparse graphs is that there, the number of recursion nodes on layer 1 accounts for a greater portion of all nodes in the recursion tree.

Let's look at our example again and assume that one call without its recursive calls would need exactly time $|P|^2$. Applying standard order (Figure (3.12)), the algorithm would need $4^2 + 2^2 + 2^2 + 1^2 = 25$ time units in total whereas with degeneracy order (Figure (3.13)), it would need $4^2 + 1^2 + 2^2 + 1^2 + 1^2 = 23$.

Minimal Size

If we wanted to include the minimal size strategy in our benchmark, the question would be how to decide on k at each parameter point. The number of MCs matching that minimum depends heavily on the graph parameters, and when a graph contains no such MCs at all, a comparison might be considered unfair and irrelevant. Furthermore, it would be interesting how exactly performance depends on k, so we made a separate evaluation at the 4 corners of the parameter cube where $n = 10,000$. Because the main influence of n is that it linearly increases runtimes, the other 4 corners, where $n = 500$, don't reveal much more and are omitted here.

To put the measurements in perspective, we still need to have an idea of how many MCs exist for each size k. Figure (3.14) illustrates distributions of MC sizes for the examined graphs. Remember that all presented numbers that relate to a benchmarked synthetic graph are actually averages over 3 graphs measured at that parameter point. This applies to the frequencies in Figure (3.14) as well.

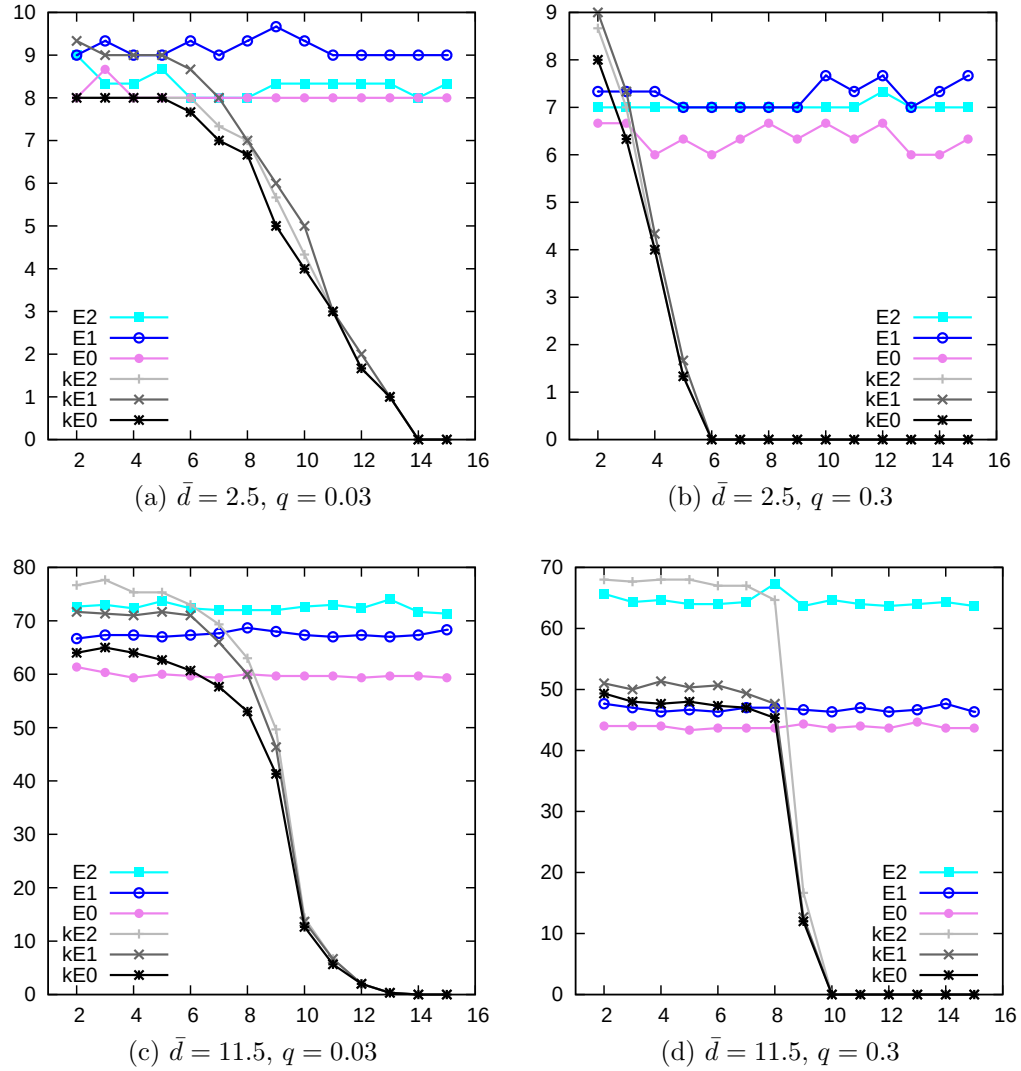


(a) $\bar{d} = 2.5$, $q = 0.03$ (b) $\bar{d} = 2.5$, $q = 0.3$

(c) $\bar{d} = 11.5$, $q = 0.03$ (d) $\bar{d} = 11.5$, $q = 0.3$

Figure 3.14: Histograms over MC size k for different values $\bar{d}$, q

The actual performances in dependence on k are shown in Figure (3.15), where k-ELS^t is labaled kEt. For $\bar{d} = 11.5$ and $q = 0.03$, as displayed in diagram (c), k-ELS^0 is 4% slower than ELS^0 at listing all 4156 MCs. However, if we are satisfied with the 61 MCs of minimal size 11, k-ELS^0 is 91% faster than ELS^0. This effect is clearly significant for lower values of q or $\bar{d}$. Only diagram (d), where both parameters are high, shows a different picture.



(a) $\bar{d} = 2.5$, $q = 0.03$ (b) $\bar{d} = 2.5$, $q = 0.3$

(c) $\bar{d} = 11.5$, $q = 0.03$ (d) $\bar{d} = 11.5$, $q = 0.3$

Figure 3.15: Runtimes over minimal MC size k for $n = 10,000$ and different values $\bar{d}$, q

The thing about density in random graphs is that with increasing $\bar{d}$, the core numbers of the nodes rise while the sizes of the MCs cannot keep up. If MCs are formed randomly, their frequency drops dramatically with their size, and we get many small MCs (Figure (3.14)). So, for most nodes, the size of the biggest MC that they belong to is much smaller than their core number. Thus, the algorithm cannot skip many nodes in the outer call because their core numbers rise too quickly at too early positions in δ-order.

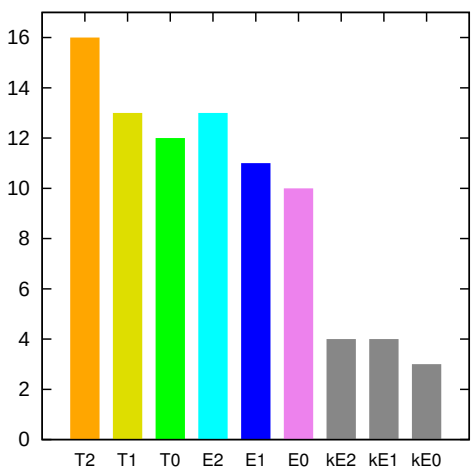
Even more important is q. If we decrease q, we get a less "random" graph that is more clustered and contains lesser but larger MCs. Now, if $\bar{d}$ and q both are high, that is really the worst case for our minimal size strategy but still the only scenario where that strategy appears to be worthless. Considering that real world graphs are most likely sparse and structured, we conclude that through our modification, the approach of Eppstein et al. makes an essential difference in practical applications.

3.4 Real World Validation

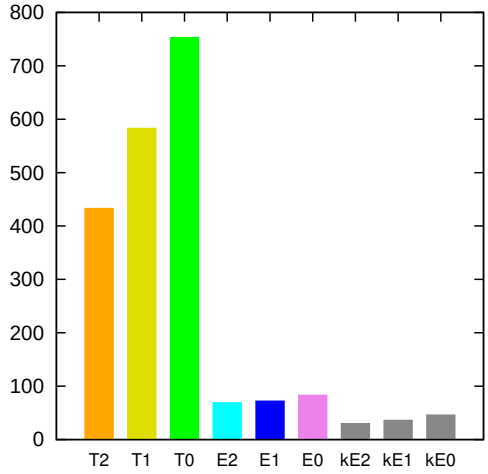
To validate our observations on real world data, we took 6 graphs from Mark Newman's data set [34]. Three of them were used by Eppstein and Strash as well [13]. Since Bron-Kerbosch is overwhelmingly dominated on all of them, we had to leave it out of the plots. In order get an idea of what potential the k-ELS variants have on real data, we chose a k for each graph such that at least 100 MCs were reported. Figures (3.16) and (3.17) show the results. Here are some general observations:

- ELS^t still dominates Tomita^t.
- ELS^0 is still an optimal general choice for listing **all** MCs.
- k-ELS^t still dominates ELS^t. The speed potential of k-ELS is still huge.
- Constant pivoting is less competitive on real graphs. In two cases, it is even much worse than quadratic pivoting.
- δ-order is more competitive on real graphs, especially on the internet graph, which has many MCs, regarding its sparsity.
- The performance patterns of all 3 Condensed Matter Collaboration graphs are very similar, which supports the significance of the presented numbers.

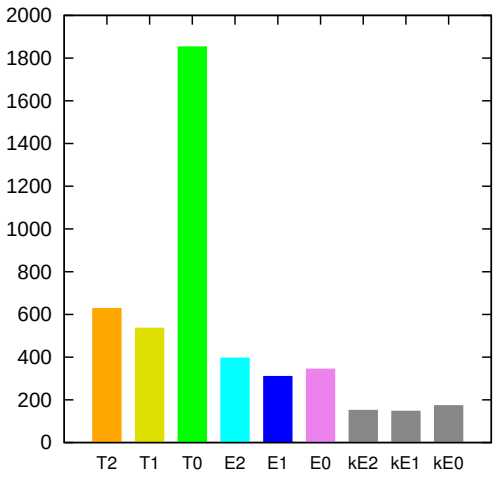
Because ELS^t is actually the same as 2-ELS^t, choosing an algorithm for the real graphs would be easy: k-ELS^0 with k set to our needs would be a very viable option. It may be an optimal k-ELS variant on these graphs and also dominates the synthetic benchmark. Since our experiments on random graphs are much more extensive and systematic, they still provide more reliable orientation.



(a) hep-th: $\bar{d} = 3.8$, $q = 0.41$



(b) internet: $\bar{d} = 4.2$, $q = 0.88$

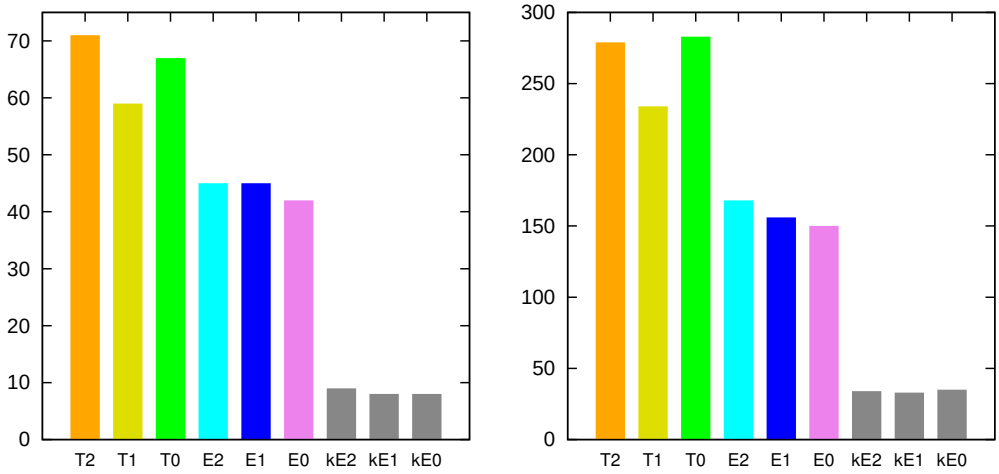


(c) astro-ph: $\bar{d} = 14.5$, $q = 0.026$

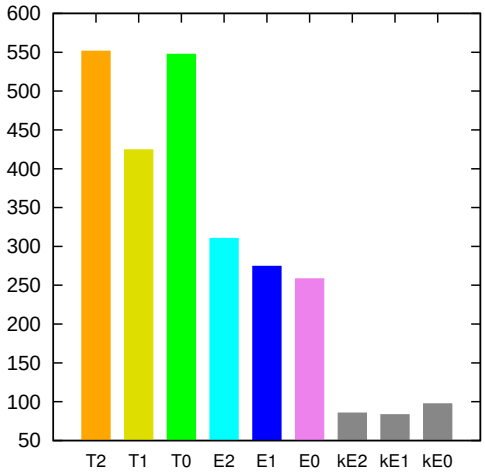
Figure 3.16: Runtimes on different (social) networks from Mark Newman's data set [34]

The internet- and the astro graph reveal some unexpected behaviour of our algorithms that needs to be mentioned. Both are positioned outside the parameter cube that we defined for our random graphs.

On the internet graph, $q = 0.88$ is much higher than on our random graphs while $\bar{d}$ is quite low. We might think that most of the many MCs must be very small. This is true to some degree, but still, 6.4% of all MCs are of minimal size 10, and the two maximum cliques are of size 17. The reason for this is a fundamental difference between our random- and real graphs.



(a) 1999: $\bar{d} = 5.7$, $q = 0.23$ (b) 2003: $\bar{d} = 7.7$, $q = 0.158$



(c) 2005: $\bar{d} = 8.7$, $q = 0.134$

Figure 3.17: Runtimes on the Condensed Matter Collaboration coauthorship networks from Mark Newman's data set [34]

Although, a high value of q might suggest the graph's MCs are small, their high number can only result from MCs sharing more edges. Just remember the Moon-Moser graphs, where no MC has even one exclusive edge to its own. That is a major difference in the way the graph could be generated. To reach such a high value of q, a generation process would have to use the available edges very systematically, at least in some parts of the graph. In real graphs, these regions emerge naturally. With MC sizes rising again for that purpose, complexity and the need to prune the recursion tree do so

as well, which explains why – in spite of its sparsity – effective (quadratic) pivoting suddenly dominates fast pivoting on the internet graph.

At the same time, sparsity and high q still lead to flat recursion and edges being more evenly distributed for most parts of the graph. 62% of all reported MCs are exactly of size 2, meaning that the recursion nodes on level 1 account for a very large portion of all recursion nodes. As we explained, when analyzing results of the synthetic benchmark, δ-order is especially useful under such conditions. That is why the otherwise small advantage of δ-order turns out to be a crucial property of Bron-Kerbosch based algorithms on sparse graphs with a very large q.

On the astro-ph graph, constant pivoting encounters problems previously unseen. The experiments on random graphs already suggested, that constant- falls behind linear pivoting for higher density and small q, as Figure (3.9 a) demonstrates. Here, both values are even out of the parameter cube, which is why Tomita^0 falls way behind even Tomita^2. Be aware that this does not significantly effect the ELS variants.

Chapter 4

Conclusion

4.1 Summary

Listing all MCs is a common task in all sorts of application domains. While it is a very complex problem theoretically, Tsukiyama et al. [39] and Bron and Kerbosch [6] published algorithms that solve it in reasonable time on small graphs. Both are dominated by the Tomita algorithm of 2006 [38], which is much faster on complex graphs. Eppstein et al. [11] improved the Tomita algorithm theoretically and empirically. Our modification k-ELS further exploits the minimal MC size k, that is given to the algorithm, and is several times faster, depending on k. In the sense that it cannot be slower than the pure ELS algorithm, and k can be set to 2, k-ELS^0 can be the dominating choice in practice.

It may occur that the complexity of listing all MCs of a certain graph is unknown, and running 2-ELS might take too long. In this case, we recommend to run k-ELS for a presumably high value of k to estimate graph structure, MC frequency and algorithm runtime. If no MC of size k or larger exists in the graph, the algorithm would just generate the δ-order, which is done in $O(n)$, and return. When we first were measuring initialization time separately, we recognized that even including δ-order generation, it takes an unmeasurable small fraction of the whole runtime. So, when k is chosen too high, it can be decreased iteratively until the algorithm reports some of the largest MCs of the graph.

When implementation effort is an issue, one should start with the Bron-Kerbosch algorithm, which is really easy to implement and also dominating the Tsukiyama algorithm empirically. All its discussed improvements can be added incrementally, with difficulty and the need to care for small details growing.

Often, papers do not provide enough evidence for practical decisions and explain the effects of their algorithms insufficiently. So, apart from analyzing performances, we also presented valuable insights into the nature of the problem and the inner workings of the algorithms, thereby empowering the reader to adjust them to his needs.

4.2 Outlook

4.2.1 Data

We would like to run benchmarks on even larger synthetic graphs for two reasons:

1. We observed that comparative measurements are more discriminating on large dense graphs with large MCs, where runtimes are longer. In fact, after all optimizations were applied, a lot of clarity had vanished from our plots because runtimes got very short.

2. The size of real world graphs knows no limit. Benchmarks should be as close to reality as possible.

Now, to come up to that, we have to overcome two problems:

1. Java limits our accessible main memory.

2. With more vertices, time- and memory demand of graph generation would explode while the size of the parameter cube that the generation algorithm can cover would be quite limited.

The first problem can easily be solved by translating our implementation to a language like C++, that gives us better control over memory usage. That would also avoid the uncertainties induced by the JVM's background activities. To tackle the second problem, we would need to adjust or rethink our graph generation approach.

Extending the real world validation might also reveal some new aspects. The graphs we used are the ones on which runtimes are neither too short nor too long to be be benchmarked. They were not selected for any special reason. Many real graphs, however, did not make it into the benchmark due to a variety of minor technical problems. Running algorithms on all sorts of real graphs would be a major priority of developing a stable, flexible and fast benchmarking system.

4.2.2 Parameters

As we developed our MC related graph parameters, we stumbled upon an interesting question: Given n and m, how many MCs can the graph contain at most? As simple as the question sounds, it seems to be non-trivial and unanswered. We reasonably estimated the maximum through a specific upper bound, but if the conjecture we presented was true, that upper bound would hold in general. To prove or disprove the conjecture would not only add to the informative value of our work but also be a significant contribution by itself. We already know that the specific bound, as given by Theorem (1), can actually be reached because it can easily be shown that $q = 1$ for Moon-Moser graphs. Now, the crucial step towards proving or disproving the conjecture would be to find out whether $q = 1$ is always realizable for $n < m < \binom{n}{2} - n$. There are two possible outcomes:

- No, it is not. We can probably show where the real maximum lies.
- Yes, it is. If the conjecture is false, a graph with $q > 1$ exists as a counter example. Else, Theorem (1) provides the precise general maximum.

Despite this question about the number of MCs, we learned in the evaluation that performances might depend much more on the number of cliques. A formula for the maximum number of cliques in a connected graph with n and $\bar{d}$ might lead to another interesting parameter and generation algorithm. Also, it might be desirable to systematically asses the interdependence between all possible parameters – those of the graphs and those of the algorithms. Algorithm parameters could be: empirical performance, number of neighbour tests, loop iterations, recursion depth and the like. Correlations between parameters could be computed statistically over many graphs or be deducted theoretically. If we knew, for example, that performance correlates very strongly with the number of neighbour tests, we could compare algorithms much more accurate and flexible based on these tests.

4.2.3 Generalisation

Now that we know which algorithms are very fast in practice, we may want to address a practical applicability problem: Because the definition of MCs is very strict, large MCs are very rare and clique sizes very limited. In a purely random graph, no clique of size k exists at all with probability

$$P(\text{"No clique of size } k \text{ exists"}) = P_k = \left(1 - p^{\binom{k}{2}}\right)^{\binom{n}{k}}$$

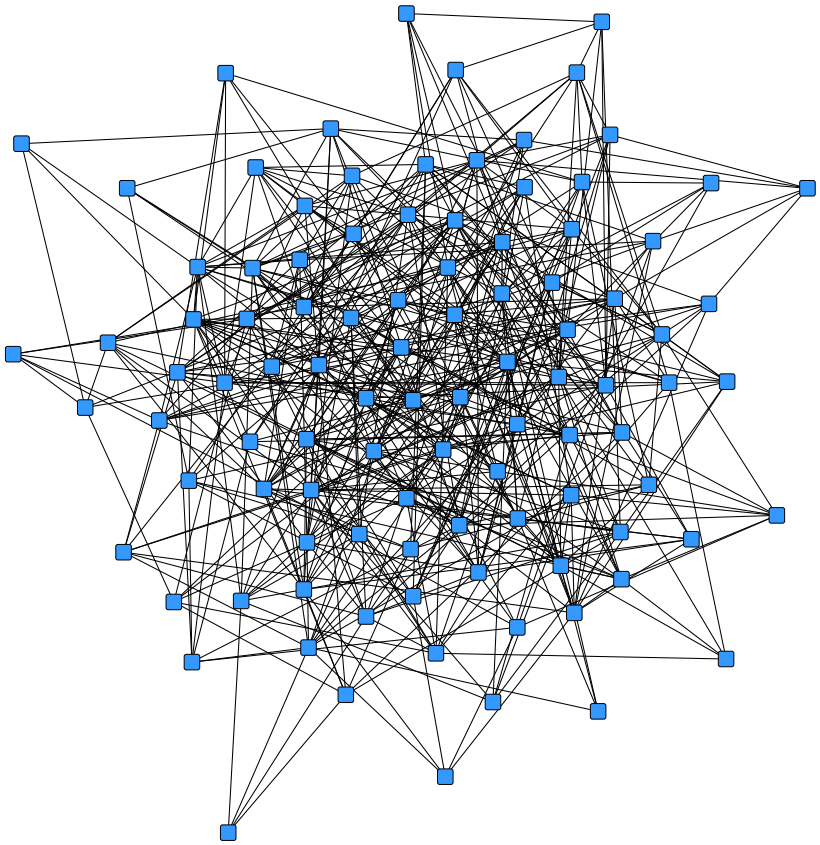


Figure 4.1: Random graph with $\bar{d} = 9.9$ and 3 maximum cliques of size 4

Also consider this simple fact: A graph contains a MC of minimal size k **if and only if** it contains a clique of size k.

Now, let's illustrate the meaning of all that by a small example: Let there be a graph with $n = 100$ and $\bar{d} = 9.9$, then $p = 0.1$, and we get two revealing probabilities $P_4 = 0.0189$ and $P_5 = 0.9925$, which tell us that the largest cliques are almost certainly of size 4. Figure (4.1) displays a graph that is typical for the example. It contains 495 edges, and its 3 maximum cliques are of size 4. Smaller MCs, however, are very frequent: We count 149 of size 3 and 180 of size 2.

Real graphs are not perfectly random, but the frequency of their MCs still drops exponentially with increasing minimal size k. That is why MCs often don't provide sufficient information for clustering related tasks. It gets even worse, when the graph has a systematic peculiarity like having no triangles. There, the discussed algorithms would be completely useless because they would just list all edges as MCs of size 2 while the graph may actually have an interesting density structure.

It would be exciting to investigate how the algorithms could be adjusted to more general MC definitions (dense regions) or to weighted graphs. Possibly, MCs can even be utilized for hierarchical graph clustering and visualization by repeatedly collapsing them into pseudo vertices. In data analysis tasks, we also have the issue that MCs might heavily overlap and have to be filtered or consolidated. Many paths lead on from here, yet it isn't the purpose of this work to go any further.

Bibliography

[1] J. G. Augustson and J. Minker. An analysis of some graph theoretical cluster techniques. *J. ACM*, 17:571–588, 1970.

[2] N. Berry, T. Ko, T. Moy, J. Smrck, J. Turnely, and B. Wu. Emergent clique formation in terorirst recruitment. In *Agent Organization: Theory and Practice Workshop*. AAAI Press, 2004.

[3] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo. The maximum clique problem. In *Handbook of Combinatorial Optimization*, pages 1–74. Kluwer Academic Publishers, 1999.

[4] B. Boyer. Robust java benchmarking, part 1: Issues, 2008.

[5] U. Brandes and D. Wagner. Visone – analysis and visualization of social networks. In *GRAPH DRAWING SOFTWARE*, pages 321–340. Springer-Verlag, 2003.

[6] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM*, 16:575–577, 1973.

[7] F. Cazals and C. Karande. A note on the problem of reporting maximal cliques. *Theoretical Computer Science*, 407(1-3):564–568, 2008.

[8] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM*, 14(1):210–223, 1985.

[9] R. G. Downey and M. R. Fellows. Fixed-parameter tractability and completeness ii: On completeness for w[1], 1995.

[10] N. Du, B. Wu, X. Pei, B. Wang, and L. Xu. Community detection in large-scale social networks. In *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*, WebKDD/SNA-KDD '07, pages 16–25, New York, NY, USA, 2007. ACM.

[11] D. Eppstein, M. Löffler, and D. Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *ISAAC (1)*, pages 403–414, 2010.

[12] D. Eppstein and E. Spiro. The h-index of a graph and its application to dynamic subgraph statistics. In F. Dehne, M. Gavrilova, J.-R. Sack, and C. Tóth, editors, *Algorithms and Data Structures*, volume 5664 of *Lecture Notes in Computer Science*, pages 278–289. Springer Berlin / Heidelberg, 2009.

[13] D. Eppstein and D. Strash. Listing all maximal cliques in large sparse real-world graphs. In *Proceedings of the 10th international conference on Experimental algorithms*, SEA'11, pages 364–375, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] U. Feige, S. Goldwasser, L. Lovasz, S. Safra, and M. Szegedy. Approximating clique is almost np-complete. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:2–12, 1991.

[15] E. J. Gardiner, P. Willett, and P. J. Artymiuk. Graph-theoretic techniques for macromolecular docking. *Journal of Chemical Information and Computer Sciences*, pages 273–279, 2000.

[16] J. Gramm, J. Guo, and R. Niedermeier. On exact and approximation algorithms for distinguishing substring selection. In A. Lingas and B. Nilsson, editors, *Fundamentals of Computation Theory*, volume 2751 of *Lecture Notes in Computer Science*, pages 963–971. Springer Berlin / Heidelberg, 2003.

[17] H. M. Grindley, P. J. Artymiuk, D. W. Rice, and P. Willett. Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm. *Journal of Molecular Biology*, 229(3):707 – 721, 1993.

[18] F. Harary and I. C. Ross. A procedure for clique detection using the group matrix. *Sociometry*, 20(3):pp. 205–215, 1957.

[19] E. R. Harley. Comparison of clique-listing algorithms. In *MSV/AMCS'04*, pages 433–438, 2004.

[20] M. Hattori, Y. Okuno, S. Goto, and M. Kanehisa. Development of a chemical structure comparison method for integrated analysis of chemical and genomic information in the metabolic pathways. *Journal of the American Chemical Society*, 125(39):11853–11865, 2003.

[21] R. Horaud. Stereo correspondence through feature grouping and maximal cliques. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11:1168–1180, 1989.

[22] H. C. Johnston. Cliques of a graph - variations on the bron-kerbosch algorithm. *International Journal of Parallel Programming*, 5:209–238, 1976. 10.1007/BF00991836.

[23] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[24] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1-2):1 – 30, 2001.

[25] I. Koch, T. Lengauer, and E. Wanke. An algorithm for finding maximal common subtopologies in a set of protein structures. *Journal of computational biology : a journal of computational molecular cell biology*, 3(2), 1996.

[26] D. R. Lick and A. T. White. k-degenerate graphs. *Canadian Journal of Mathematics*, 22:1082–1096, 1970.

[27] E. Loukakis. A new backtracking algorithm for generating the family of maximal independent sets of a graph. *Computers and Mathematics with Applications*, 9(4):583–589, 1983.

[28] E. Loukakis and C. Tsouros. A depth first search algorithm to generate the family of maximal independent sets of a graph lexicographically. *Computing*, 27:349–366, 1981. 10.1007/BF02277184.

[29] R. D. Luce and A. D. Perry. A method of matrix analysis of group structure. *Psychometrika*, 14(2):95–116, 1949.

[30] K. Makino and T. Uno. New algorithms for enumerating all maximal cliques. In *Scandinavian Symposium and Workshops on Algorithm Theory*, volume 3111 of *LNCS*, pages 260–272. Springer, 2004.

[31] S. Mohseni-Zadeh, P. Brézellec, and J. L. Risler. Cluster-c, an algorithm for the large-scale clustering of protein sequences based on the extraction of maximal cliques. *Computational Biology and Chemistry*, 28(3):211 – 218, 2004.

[32] S. Mohseni-Zadeh, A. Louis, P. Brézellec, and J. Risler. Phytoprot: a database of clusters of plant proteins. *Nucleic Acids Res.*, 32(1):D351–D353, 2004.

[33] J. Moon and L. Moser. On cliques in graphs. *Israel Journal of Mathematics*, 3:23–28, 1965. 10.1007/BF02760024.

[34] M. E. J. Newman. The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences of the United States of America*, 98(2):404–409, 2001.

[35] P. M. Pardalos. The maximum clique problem. 1992.

[36] R. Samudrala and J. Moult. A graph-theoretic algorithm for comparative modeling of protein structure. *Journal of Molecular Biology*, 279(1):287 – 302, 1998.

[37] D. Strash. Listing all maximal cliques in large sparse real-world graphs (symposium on experimental algorithms), 2011.

[38] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1):28 – 42, 2006. Computing and Combinatorics, 10th Annual International Conference on Computing and Combinatorics (COCOON 2004).

[39] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.

[40] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. Technical report, Rochester, NY, USA, 1997.

[41] A. Zomorodian. The tidy set: A minimal simplicial set for computing homology of clique complexes. In *Proc. ACM Symposium of Computational Geometry*, pages 257–266, 2010.