UNIVERSITÀ DI PISA

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

Tesi di Laurea

LISTING LARGE CLIQUES IN REAL-WORLD GRAPHS

Relatore:
Prof. ROBERTO GROSSI

Relatore:
Dott. ALESSIO CONTE

Candidato:
DAVIDE RUCCI

ANNO ACCADEMICO 2019 - 2020

Abstract

In undirected graphs, a clique is a subset of its vertices which are all pairwise connected. The problem of detecting all cliques of a graph has received extensive study due to its various fields of applications, ranging from detecting social communities, through the development of integrated circuits, to extracting information from protein interactions and even detecting communities of dolphins in the ocean. The clique problem is also of relevant interest from the theoretical point of view, since it is one of the first 21 problems to be classified as NP-Complete by Karp in 1972. For these reasons many different strategies were adopted throughout the years to solve it as fast as possible, including heuristics and approximation algorithms.

The version of the clique problem that we attack in this thesis is the enumerative one, as we want to list and count every inclusion-maximal clique found in large graphs. Nowadays graphs are used to formalize a huge variety of contexts and their size is growing at a fast pace, thus the need to design faster algorithms that are capable to process them in a reasonable amount of time.

One of the first algorithms able to enumerate all maximal cliques of a graph was that by Bron and Kerbosch, a recursive backtracking algorithm which performs a depth-first visit of the search tree that it explores, adding one vertex at a time to the clique being formed. While the authors gave no time bounds for their algorithm, it has later been proven that with a particular strategy for pruning, it achieves $\mathcal{O}(3^{|V|/3})$ time, which is worst-case optimal. More recently Eppstein et al. proposed a further enhancement on this algorithm, making it fixed-parameter tractable and very fast in practice by exploiting a particular ordering of the vertices.

The goal of this thesis is to specialize the version of the algorithm by Eppstein et al. to find cliques of at least k vertices, motivated by the fact that larger cliques often carry more significant information about communities rather than the small ones. We design new fast methods to further reduce the number of recursive calls made by the algorithm, practically expanding the pool of affordable graphs to process. We plug our strategy into the existing implementation of the algorithm by D. Strash and we do extensive testing both on real and randomly generated datasets with different properties to show its practical efficiency with respect to Eppstein's starting algorithm, and how this relates to the structure of the graphs.

Table of Contents

Introduction **iii**

1 Preliminaries **1**
1.1 Graph Degeneracy 2

2 The Bron-Kerbosch Family of Algorithms **5**
2.1 The Bron-Kerbosch Algorithm 5
2.2 The Heuristic by Tomita et al. 7
2.3 The Algorithm by Eppstein et al. 10
2.3.1 Parametrized Complexity 12
2.3.2 Output-Sensitivity 12

3 Extending Eppstein's Algorithm **15**
3.1 Size Heuristic 15
3.2 Edge Count Heuristic 17
3.3 Other Approaches 20
3.3.1 Chromatic Number 20
3.3.2 Clique Number 23
3.3.3 Degeneracy and Second Edge Count Heuristic 23
3.3.4 Erdös and Turán Theorems 24

4 Evaluation and Experiments **27**
4.1 Implementation Details 27
4.2 Dataset 30
4.3 Experimental Study and Discussion 32
4.3.1 Recursive Calls Analysis 33
4.3.2 Individual Datasets Analysis 34
4.3.3 Results Visualization 38

5 Conclusions and Future Work **42**

Bibliography **44**

Introduction

The concept of *network* has always been fundamental in its various contexts and fields of application. Computer science is no exception, in fact the popularity of networks in this area has been ever increasing as they are able to model a huge number of situations such as computer networks, social networks [26], road networks or even protein to protein interactions [27]. These networks can be easily formalized using *graphs*: a graph $G = (V, E)$ consists in a set of entities called vertices, V, and a set E of pairwise connections between them, called edges. There are various types of graphs depending on the properties of their edges: they can be ordered, labelled or weighted for instance, or they can even be subsets of more than two vertices (hypergraphs). If we allow E to be a multiset (i.e. to have repeated elements inside) we are dealing with multigraphs.

The majority of real-world networks, nowadays, can be modeled with simple undirected graphs, that is graphs with no self loops and no directed or weighted edges. Most of these networks do not appear to be random, in the sense that they cannot be represented by an Erdös-Renyi graph [10], traditionally considered the model of choice for random graphs. For this reason one is interested in finding the substructures that compose these graphs in order to extract as much information as possible from them. One possible aim is to find *communities*, groups of vertices that share a high number of links on their "inside" and a significantly smaller number of connections with the rest of the graph. The interpretation given to the communities found varies according to the field from which comes the network: groups of friends on social networks can be used to suggest new people to become friends with, a community of routers can recover a lost link more easily if it is aware of the structure of the net [14] and a cluster of web clients can be served by a dedicated mirror server with less time latency. Customers that share the same interests on e-commerce sites can be grouped together and targeted with specific product recommendations while, in a more computer science related context, clusters in graphs can be used to summarize the structure of huge graphs to help speeding up search, navigation queries and more. More information and details about general community detection can be found in [11, 12].

The traditional and most intuitive formalization of communities in graphs is that of *cliques*, subsets of vertices that are all pairwise connected, the strongest form of aggregation in a network. We put our focus on the enumeration of maximal cliques in simple and undirected graphs, a traditional NP-Hard problem [15] in computer science that has received extensive study both in past and recent years [17].

The Bron-Kerbosch algorithm was the first enumeration algorithm to be able to handle sufficiently large graphs in reasonable time in practice, despite not having an optimal worst-case theoretical complexity [5]. Later in 2006 and then in 2011, two improvements were described for this algorithm, the latter one pushing the algorithm to its best known performance [9], while the first served also as a proof for its newly-achieved worst-case optimality [24]. In this thesis we will discuss all these three versions of the Bron-Kerbosch algorithm to understand how they work and where their efficiency comes from, then we will proceed to present some techniques to further enhance its performances by exploiting the properties of a little variation of the problem to be solved.

Our aim is to enumerate only those cliques of size larger than a given parameter, motivated by the fact that we expect large cliques to carry more information than small ones like 3-, 4- or 5-cliques that often cover the vast majority of these types of community in real-world graphs, especially those that come from social networks. The heart of this work is therefore the deep understanding of the Bron-Kerbosch algorithm and its most recent variation, and the development of reasonable and fast strategies to adapt it to our problem by cutting away useless branches of the search space to be explored and thus reduce the overall running time of the algorithm.

This thesis is organized as follows. Chapter 1 formalizes our problem and gives precise definitions of the tools we will need throughout our dissertation. Chapter 2 presents the three aforementioned versions of the Bron-Kerbosch algorithm, namely the Tomita et al. variant and the Eppstein et al. variant, in increasing order of efficiency, and we discuss their correctness and their running times in a literature review fashion. Chapter 3 constitutes the theoretical heart as it explains the strategies and heuristics that we developed in order to shrink the search space of the algorithms according to our need to enumerate only large cliques. We give formal proofs of necessary conditions that must hold during its execution and we show how we can skip useless work if these conditions are not met. We also give an insight on other conditions, the sufficient ones, that cannot be exploited directly in our setting but can be useful to conduct future and related work on this topic. Chapter 4 shows the extensive experimental validation that we have done on both real-world and synthetic graphs to better understand how the algorithm performs given different structural properties of the networks and to asses the quality of our heuristics. These two chapters represent the core of this work and contain almost entirely original material. Finally, Chapter 5 concludes the dissertation and gives some ideas on how to proceed and what could be done in the future for this problem.

Chapter 1

Preliminaries

In this chapter we give some basic definitions, fix the notation and formally state the problem that we are going to face.

Let $G = (V, E)$ be a graph, where V is the set of vertices and E is the set of edges. The graphs that we deal with in this thesis are all simple and undirected, meaning that self-loops are not allowed and that the pairs of vertices $\{a, b\} \in E$ are not ordered, so that the edge $\{a, b\}$ is the same as $\{b, a\}$ and can be traversed in both directions as needed. We assume that G is connected, otherwise the algorithms presented here can be applied to every connected component separately. Figure 1a shows an example of simple, undirected and connected graph.

The neighborhood of a node $v \in V$ is $N(v) = \{u \in V \mid \{u, v\} \in E\}$, and the degree of a node is defined as $\delta(v) = |N(v)|$. The next definition serves as a basis for our work.

Definition 1.1 (Induced Subgraph)**.** Let $G = (V, E)$ be a graph and let $S \subseteq V$. The graph $G[S] = (S, E' = \{\{u, v\} \in E \mid u, v \in S\})$ is called the *subgraph of G induced by S* (or just *induced subgraph* when G and S are clear from the context).

Figure 1b shows the subgraph induced by the set of vertices $\{B, D, E\}$ of the main graph.

A clique in a graph is a subset of its vertices that are all pairwise connected and it is formally defined as follows.

Definition 1.2 (Clique)**.** A *clique* in a simple graph $G = (V, E)$ is a subset of vertices $K \subseteq V$ such that for each $u, v \in V, u \neq v$, $\{u, v\} \in E$. In other words, $G[K]$ is the complete graph on $|K|$ vertices. A *k-clique* is a clique made by exactly k vertices.

In this thesis we are interested in cliques that are maximal under the inclusion operator.

Definition 1.3 (Maximal Clique)**.** A clique K is *inclusion-maximal* (or just *maximal*) when there is no $K' \subseteq V$ such that $K' \supsetneq K$ and K' forms a clique in G.

Figure 1b shows that the subgraph induced by vertices $\{B, D, E\}$ is actually a clique of three vertices, and it is also maximal as there no other neighbors common to all

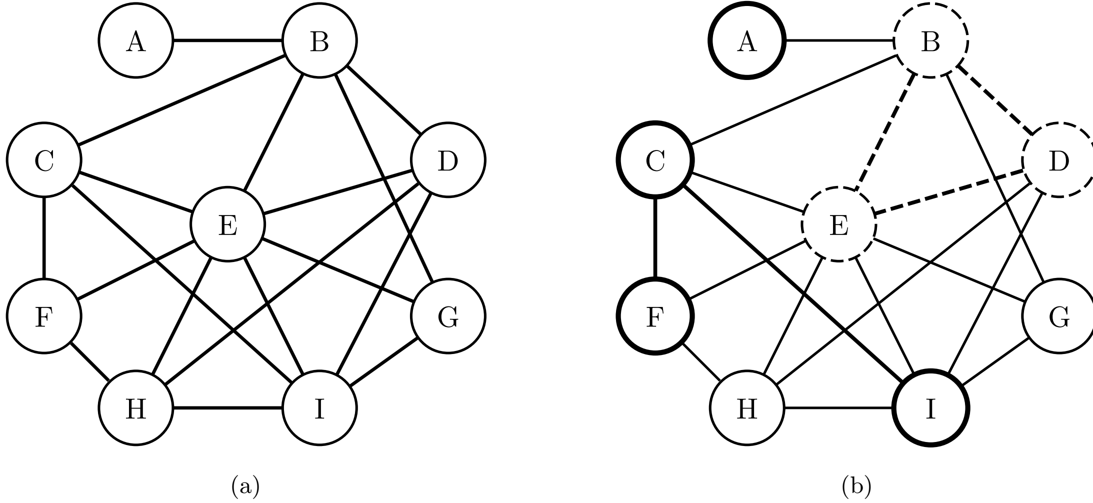


Figure 1: (a) A simple, undirected graph. (b) Two induced subgraphs, one by the subset $\{B, D, E\}$ (dashed), the other by the subset $\{A, C, F, I\}$ (bold). This graph has been rebuilt from the one found in [9].

three vertices at the same time. The induced subgraph $G[\{A, C, F, I\}]$, instead, is not a complete subgraph and it is not even connected.

Now that we have the basic definitions, we can state the main goal of this thesis. The *clique problem* known in the literature, given $k \in \mathbb{N}$, asks if there exists k-clique in G. We consider an extension of this problem consisting in the enumeration of all maximal cliques with at least k vertices involved. Formally,

Definition 1.4 ($k_{\geq}$-Clique Enumeration Problem)**.** The problem that we attack is:
INSTANCE: A simple, undirected graph $G = (V, E)$ and $k \in \mathbb{N}, k \leq |V|$.
QUESTION: Enumerate all maximal *t-cliques* of G, where $t = k, k+1, \ldots, |V|$.
We call this problem the *enumeration of* $\mathbf{k}_{\geq}$*-cliques*.

Since the Clique Problem is known to be NP-Complete [15], it follows that our version is at least NP-Hard. It has also been proven by Moon and Moser [21] that there exist graphs having a number of maximal cliques that is exponential in the number of their vertices, thus requiring any algorithm to take at least exponential time in the worst case.

1.1 Graph Degeneracy

One of the most common properties of graphs is that of *sparsity*, an indicator of how much "connection" there is among their vertices. More formally, a graph $G = (V, E)$ is usually said to be *sparse* if $|E| = \mathcal{O}(|V|)$, or dense if $|E| = \mathcal{O}(|V|^2)$. One of the possible measures of this property is the *edge density*, defined as the ratio of the number of edges with respect to the maximum possible number of edges in a given graph, while other

Algorithm 1.1 Linear time greedy algorithm to compute a degeneracy ordering of a graph

```
1: function DegeneracyOrder(G = (V, E))
2:     D ← array of empty lists of length |V|
3:     L ← empty list
4:     d ← 0
5:     for all v ∈ V do
6:         Append v to the list D[δ(v)]
7:     end for
8:     for i = 1 to |V| do
9:         j ← 0
10:        while D[j] is empty do
11:            j ← j + 1
12:        end while
13:        d ← max{d, j}
14:        Remove a vertex u from D[j]
15:        Append u to L
16:        Move all vertices of N(u) from their respective D[x] to D[x − 1]  ▷ Consider
   only vertices of N(u) not already in L
17:    end for
18:    return ⟨d, L⟩              ▷ L contains the vertices in the degeneracy order found
19: end function
```

measures include *arboricity*[1] and *thickness*[2]. Another, more refined, sparsity measure which is a constant factor away from the latter two and will be crucial throughout all the dissertation is that of *degeneracy*, defined as follows.

Definition 1.5 (Degeneracy, Lick and White [18])**.** *The degeneracy of a graph $G = (V, E)$ is the least $d \in \mathbb{N}$ such that every induced subgraph of G contains a vertex v with $\delta(v) \leq d$.*

The value of d for any graph can be computed in linear time, using an algorithm that repeatedly removes a vertex v such that $\delta(v)$ is minimum [20, 3, 9]. An instance of such a procedure is shown in Algorithm 1.1: it follows a greedy strategy that uses two simple data structures like an array of lists and a list, to iteratively pick and remove a vertex of minimum degree. Using this strategy it is possible to get the degeneracy value for $G = (V, E)$ in time $\mathcal{O}(|V| + |E|)$, as line 16 of Algorithm 1.1, the most expensive of all operations, takes time proportional to the degree of u, the vertex being removed.

As a side effect of this removal procedure we can get a particular ordering of V (represented by L in Algorithm 1.1), the so-called *degeneracy ordering*, obtained by orienting the edges of G from u to v if u has been removed before v during the algorithm

[1] The arboricity of a graph is the minimum number of forests into which its edges can be partitioned.

[2] The thickness of a graph is the minimum number of planar subgraphs into which its edges can be partitioned.

execution; the graph obtained is a *Directed Acyclic Graph* (DAG). Since this procedure will give all vertices an outdegree (i.e. the number of edges exiting a node) of at most d, the following definition of degeneracy is equivalent to Definition 1.5.

Definition 1.6 (Degeneracy [6])**.** A graph $G = (V, E)$ has degeneracy d if and only if its edges can be oriented to form a Directed Acyclic Graph (DAG) with each node having outdegree less than or equal to d.

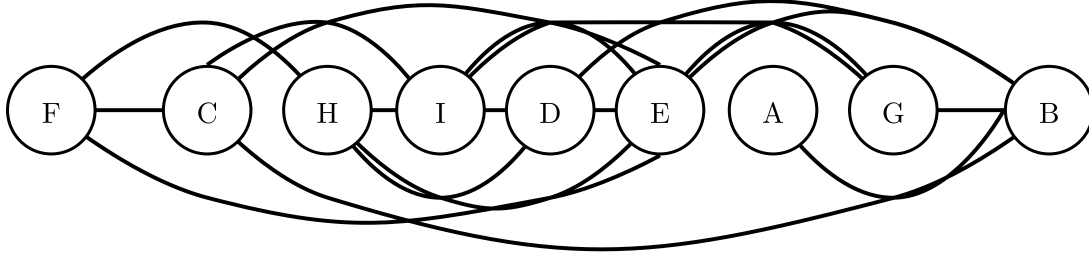


Figure 2: A possible degeneracy ordering of the graph in Figure 1. Figure rebuilt from [9].

Note that there is not a unique degeneracy order for a given graph, instead there are many orderings with the same outdegree: what matters in this context is that every vertex has a number of neighbors later in the ordering bounded by d. Figure 2 shows a possible output of Algorithm 1.1 on the graph from Figure 1a. We can see that every vertex has three or less neighbors that come after it in the ordering, thus we can conclude that this graph has degeneracy 3, according to Definition 1.6.

The following Lemma, additionally, establishes a bound on the number of edges of a graph in function of its degeneracy.

Lemma 1.1 (Lick and White [18])**.** *For a graph $G = (V, E)$ with degeneracy d it holds that $|E| \leq d\left(|V| - \frac{d+1}{2}\right)$.*

It is clear that the number of edges grows linearly with the number of nodes when the value of d is sufficiently small, i.e., $d = o(|V|)$. If this happens we can then say that G is sparse. Notice how the degeneracy is often a better measure of sparsity than others like, for instance, the maximum degree of the graph: Figure 1 has maximum degree equal to 6, but it is actually more sparse as it contains only cliques of up to four vertices, a fact well captured by the degeneracy since we have the following result.

Lemma 1.2. *In a graph $G = (V, E)$ with degeneracy d, the maximum clique size possible is $d + 1$.*

Proof. Suppose that there exists a clique in G made of $t > d + 1$ vertices. Then this clique forms a subgraph in G where each node has degree $\delta(v) \geq t - 1 > d + 1 - 1 = d$ and, by Definition 1.5, G has degeneracy greater than d, a contradiction. ■

Given the basics we may now proceed to describe one of the most used techniques to enumerate all maximal cliques in graph.

Chapter 2

The Bron-Kerbosch Family of Algorithms

In this chapter we present one the first algorithms able to efficiently enumerate all maximal cliques, discovered by Bron and Kerbosch in 1973 [5], along with two of its most recent enhancements. While the algorithm has evolved over the years, the heart of its process has remained practically unchanged and has spawned a whole family of algorithms used to enumerate cliques and other type of communities such as k-plexes (e.g. [7]).

We first review the idea of the BK algorithm, then we discuss its aforementioned variations.

2.1 The Bron-Kerbosch Algorithm

The Bron-Kerbosch algorithm discovers all maximal cliques of a graph by exploiting recursion and backtracking, and it works by constructing cliques one vertex at a time picking it from a list of candidates. If the clique being constructed eventually achieves maximality it is output, otherwise the algorithm backtracks and selects another candidate to be added to the partial clique. This corresponds to a depth-first visit of the search tree spanned by the algorithm itself. The set of candidates will be called P, the clique under construction will be contained in the set R, while the set X keeps track of vertices which cannot be added to R for reasons that will be clear later. These three sets constitute the heart of the algorithm, and the strategies adopted to populate them or to extract elements from them are what defines the running time of the whole process.

Before describing the pseudocode of the algorithm we remark the roles of the three aforementioned sets:

- **P**: The set of candidates from which to pick a vertex at each recursive step to expand the current clique.
- **R**: The clique being constructed.

Algorithm 2.1 Bron-Kerbosch Algorithm

```
1: function BRON-KERBOSCH(P, R, X)
2:     if P = ∅ and X = ∅ then
3:         return                                  ▷ R is a maximal clique
4:     end if
5:     for all v ∈ P do
6:         BRON-KERBOSCH(P ∩ N(v), R ∪ {v}, X ∩ N(v))
7:         P ← P \ {v}
8:         X ← X ∪ {v}
9:     end for
10: end function
```

- **X**: The set of vertices that cannot be added to R at a given time during the recursion.

We can now discuss Algorithm 2.1. The procedure builds and explores a search tree in which nodes represent an instance of the R set, and edges correspond to the vertices being added from P to R before recurring. To enumerate all maximal cliques of a graph $G = (V, E)$ one must issue the call BRON-KERBOSCH(V, $\emptyset$, $\emptyset$). Each iteration of the for loop adds a vertex v to R and updates P and X to keep only the vertices in the neighborhood of v. Then the algorithm proceeds to recursively enumerate all cliques containing $R \cup \{v\}$ and when it finishes the vertex v is moved to X. By doing this at each recursive call we can get to a point where $P = X = \emptyset$ and $R \neq \emptyset$ which means that R is maximal as no other vertex is a neighbor of all R, otherwise P would have been non-empty. The set X contains vertices that have already participated to the enumeration of cliques with the current configuration of R, and are now explicitly excluded. The reason for this is given in the proof of the following Proposition.

Proposition 2.1. *A necessary condition for R to be a maximal clique is that P is empty. If X is empty at the same time, then R is maximal and has not been previously discovered.*

Proof. At each level of recursion, the set R is populated by picking one vertex from P, and P itself is updated with an intersection operation. At level i, the corresponding P_i set can be written as $P_i = P_0 \bigcap_{j=0}^{j=i} N(v_j)$, while $R_i = \{v_i, v_{i-1}, \dots, v_0\}$. If R_i is maximal, P_i must be empty by its definition, in fact P_i corresponds to all vertices that are adjacent to R_i.

Suppose now that at some stage we reach $P = \emptyset$ but $X = \{x_1, x_2, \dots x_t\}$. Recall that a vertex u joins X after the recursive call in which it has been added to R. If u is still in X when P is empty, it means that u is in the neighboorhood of all R and thus it could be used to expand that clique, but u was already used with this R at previous stages, so the clique contained in R is not maximal.

■

Proposition 2.1 is a basis for proving the correctness of the Bron-Kerbosch algorithm, i.e., that it will enumerate all and only maximal cliques of G without duplication.

Theorem 2.1 (Correctness of BRON-KERBOSCH [5])**.** *Algorithm 2.1 enumerates all and only maximal cliques of* $G = (V, E)$ *without duplication.*

The authors gave no detailed time complexity analysis, but noted that an implementation without any particular strategy to choose vertices from P would take up to $\mathcal{O}(4^{|V|/3})$ steps, while their proposed method to select a candidate benefits the whole algorithm bringing it to $\mathcal{O}(3.14^{|V|/3})$. It is also known that this version of the algorithm is not output sensitive, i.e., its time complexity cannot be written as a function of its output size. However, as we shall see in the next section, it is nearly worst-case optimal and with a good pivoting heuristic it can reach optimality in the worst case.

2.2 The Heuristic by Tomita et al.

In 2006, Tomita, Tanaka and Takahashi proposed a pivoting strategy to reduce the number of recursive calls made by the Bron-Kerbosch algorithm [24]. This mechanism makes the algorithm worst-case optimal with a $\mathcal{O}(3^{|V|/3})$ time complexity. We discuss the details in this section.

We begin by clarifying the definition of *pivoting* in this context.

Definition 2.1 (Pivot and Pivoting)**.** A *pivot* is a particular vertex chosen before the loop through P in the Bron-Kerbosch algorithm. The method used to select the pivot and the modification of the loop according to some strategy involving the pivot is called *pivoting.*

The number of iterations of the for loop will vary according to the pivoting strategy and to which pivot is selected. The Bron-Kerbosch algorithm revised according to the Tomita strategy is presented in Algorithm 2.2. We can see that the heart of the procedure is unchanged except for lines 6-7 in which a pivot u is chosen and then the for loop is on the elements of $P \setminus N(u)$. Let us informally remark how this works: assuming that these changes preserve the correctness of the algorithm it is clear that the for loop will make fewer iterations compared to the "pure" version of Algorithm 2.1, because $|P \setminus N(u)| \leq |P|$ where equality holds only in the case that $N(u)$ is the empty set. Thus we can say that Algorithm 2.2 will always make less recursive calls or at most the same number as without the pivoting strategy, the optimal case being when $|N(u)|$ is maximum at each stage, which is what we will ensure with the selection of the pivot. Clearly choosing u must be done as efficiently as possible, otherwise the benefits of reducing the recursion levels will be balanced by the overhead introduced by line 6 of the pseudocode. Now we shall see the details of the strategy proposed by Tomita et al. to choose the pivot u in order to maximize $|N(u)|$ and in turn minimize the number of subsequent recursive calls.

Suppose we are at a generic step of the algorithm, with $P, X, R \neq \emptyset$ and let $q \in P$ be the vertex chosen for expanding R to $R_q = R \cup \{q\}$.

Algorithm 2.2 Bron-Kerbosch with Tomita et al. Pivoting Heuristic

```
1: function TOMITA(P, R, X)
2:     if P = ∅ and X = ∅ then
3:         return                                          ▷ R is a maximal clique
4:     end if
5:     u ← a vertex in P ∪ X with maximum |P ∩ N(u)|
6:     for all v ∈ P \ N(u) do
7:         TOMITA(P ∩ N(v), R ∪ {v}, X ∩ N(v))
8:         P ← P \ {v}
9:         X ← X ∪ {v}
10:    end for
11: end function
```

Observation 2.1. When R_q is formed, only vertices in $P_q = P \cap N(q)$ can be used for expanding R_q, since for any $x \in X_q = X \cap N(q) \subseteq X$, the cliques $R_q \cup \{x\}$ have already been listed by definition of X.

Now suppose that all cliques associated to $R \cup \{u\}$, for a given $u \in P \cup X$ have been generated.

Observation 2.2. Every *new* maximal clique containing R but not $R \cup \{u\}$ must contain at least one vertex that is in $P \setminus N(u)$. Otherwise R could be expanded to $R \cup S$ with $S \subseteq P \cap N(u)$, leading to a contradiction because $R \cup S \cup \{u\}$ would have formed a larger clique, hence $R \cup S$ was not maximal.

Following these two observations we can conclude that the only subtrees to be expanded during the algorithm execution are only those corresponding to nodes in $P \setminus N(u)$, for a vertex $u \in P \cup X$. It follows that the most appropriate u is the one that maximizes $|P \cap N(u)|$ and thus minimizes $|P \setminus N(u)|$, to expand the least subtrees possible. Figure 3 shows the search forest of this algorithm for an example graph. Notice how it expands just a few nodes, in contrast to the standard procedure that would have expanded every node of every tree.

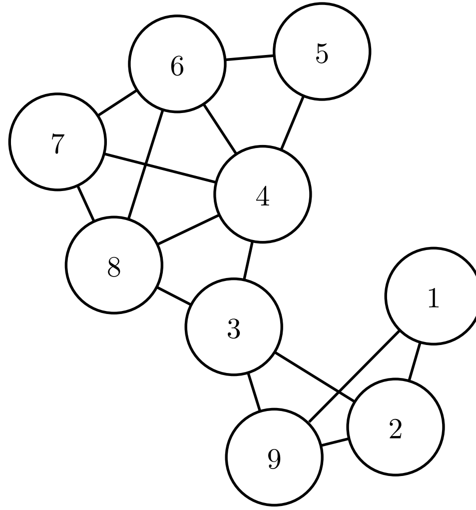
The next theorem, proven in [24], ensures the correctness of this modified version of the Bron-Kerbosch algorithm.

Theorem 2.2 (Correctness of Tomita et al. [24])**.** *Given a graph $G = (V, E)$, Algorithm 2.2 generates all and only maximal cliques without duplication.*

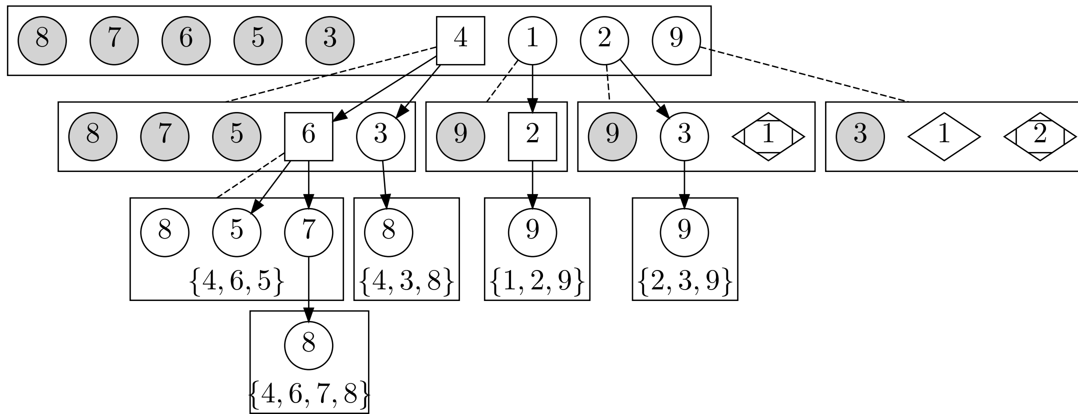
What remains then is to prove the worst-case running time.

Theorem 2.3 (Time Complexity of Tomita et al. [24])**.** *The Bron-Kerbosch algorithm with the Tomita et al. pivoting strategy, takes $\mathcal{O}(3^{|V|/3})$ time, which is optimal for a graph $G = (V, E)$.*

While the proof can be found again in [24], we want to make some remarks about this complexity.



(a) A simple undirected graph.



(b) White background represents non-neighbor vertices of the pivot, which is marked by a square. Vertices in $P \cap N(pivot)$ are shaded. A diamond represents a vertex in X, while vertices in P are represented by circles. Squared diamonds are vertices in X chosen as pivot.

Figure 3: (a) An example graph with five maximal cliques. (b) A search forest for Tomita et al. algorithm on the graph in (a). Leaves with a label indicate that a clique has been found. Both figures were recreated from the ones found in [24].

Remark 2.1. The algorithm is worst-case optimal because there exist a class of graphs, namely the Moon-Moser graphs [21], that have $\Omega(3^{|V|/3})$ maximal cliques.

Remark 2.2. For the algorithm to correctly achieve the $\mathcal{O}(3^{|V|/3})$ bound, the cliques found must not be explicitly listed.

This can become of crucial importance especially when the cliques found are large. More precisely, if one wants to explicitly list all cliques, the time complexity becomes $\mathcal{O}((d+1)3^{|V|/3})$, where we recall $d+1$ is the maximum possible size of any clique in a

Algorithm 2.3 Eppstein et al. algorithm for maximal clique enumeration

```
1: function EPPSTEIN(G)
2:     for all v_i ∈ V in a degeneracy ordering of G do
3:         P ← N(v_i) ∩ {v_{i+1}, v_{i+2}, ..., v_{|V|-1}}
4:         X ← N(v_i) ∩ {v_0, v_1, ..., v_{i-1}}
5:         TOMITA(P, {v_i}, X)
6:     end for
7: end function
```

graph with degeneracy d. For the optimal bound to be achieved, once a maximal clique has been found, only $\mathcal{O}(1)$ work should be done, e.g. by incrementing a counter.

As a final remark, we note that the selection of the pivot can be done in $\mathcal{O}(|P|^2)$.

2.3 The Algorithm by Eppstein et al.

So far we have discussed enumeration algorithms valid for any graph, that exploit none or few properties of their input. Tomita et al. pruning technique alone, however, is not sufficient for handling very large graphs like the ones behind popular social networks or, for example, like the Internet graph. In 2010, Eppstein, Löffler and Strash developed a new variation of the Bron-Kerbosch algorithm (we refer to the paper published later in 2013, see [9]) that exploited the sparsity of the input graph to speed up the computation. As outlined in Chapter 1, the measure of graph sparsity that we are going to use is that of degeneracy, which we recall from Definition 1.5 to be the least d such that every subgraph in $G = (V, E)$ has a vertex of degree at most d.

The key change of this algorithm is, in fact, to loop through the vertices of G following their degeneracy order rather than an arbitrary (e.g. lexicographic) fashion. Algorithm 2.3 gives the pseudocode of the new procedure. This time it consists in an even simpler change made at the outermost recursion level, before calling the pivoted version of Bron-Kerbosch. In particular, the algorithm serves as a preprocessing phase for the TOMITA one, by appropriately setting P and X to the vertices that follow and that precede each $v_i \in V$, respectively. Then the algorithm falls back to TOMITA, letting it do the actual enumeration in the same way as before. It turns out that this simple preprocessing step makes the algorithm fixed-parameter tractable, with a time complexity of $\mathcal{O}(|V|d3^{d/3})$, losing the exponential dependency on $|V|$.

We now report the results that prove the correctness of Algorithm 2.3, i.e., we show that the reorganization done on P and X before calling TOMITA does not hinder the correctness of whole technique.

Theorem 2.4 (Correctness of Eppstein et al. [9])**.** *Algorithm 2.3 generates all and only maximal cliques of $G = (V, E)$ without duplication.*

Proof. By Theorem 2.2, line 5 of Algorithm 2.3 generates all and only maximal cliques involving vertex $\{v_i\}$, some vertices in P and no vertex of X (without duplication). Now

consider a maximal clique C and let $v \in C$ be the earliest of its nodes in the degeneracy order of G. By Theorem 2.2, C will be reported only once when processing v, which will then be moved to X. When processing any other $u \in C$, v will be in X, thus C will not be reported again. ■

The following Lemma gives a more precise bound on the time needed to perform the pivot selection of TOMITA, and is important for proving the total running time of Eppstein et al. algorithm.

Lemma 2.1 (Lemma 3.4 of [9]). *A call to* TOMITA *issued by line 5 of. Algorithm 2.3, takes $\mathcal{O}(|P|^2(|P|+|X|))$ total extra time for the pivot selection.*

We note that this is the total running time for the pivot selection for each first-level call. In other words it is the total time spent for the pivot search for each $v \in V$. Another thing to be noted is that the degeneracy order have to be computed only once at the beginning of the procedure, and this can be done using Algorithm 1.1 described in Chapter 1 with a running time of $\mathcal{O}(|V|+|E|)$, a negligible overhead with respect to the total cost, formally stated in the following theorem.

Theorem 2.5 (Eppstein et al. Running Time [9]). *Algorithm 2.3 reports all and only maximal cliques of a given graph $G=(V,E)$ with degeneracy d, in $\mathcal{O}(d|V|3^{d/3})$ time.*

This bound resembles the one of Tomita et al., but this time the exponential dependency is on d rather than $|V|$, a remarkable improvement on the previous bounds since, for many real-world graphs, $d \ll |V|$. From a theoretical point of view, however, this bound is still non-optimal as proven by the authors themselves with the following theorem.

Theorem 2.6 (Theorem 3.8 of [9]). *Let $G=(V,E)$ be a graph with degeneracy d and $|V| \geq d+3$. The largest number possible of maximal cliques contained in G is $(|V|-d)3^{d/3}$.*

The proof gives an upper bound and then shows that there exists a graph with a number of cliques matching that bound, making it tight. We know from Chapter 1 that a graph with degeneracy d has maximum possible clique size equal to $d+1$, so a lower bound for enumerating all maximal cliques is $\Omega(d(|V|-d)3^{d/3})$. The authors also show that there exists an optimal algorithm matching this time bound by a simple modification of Algorithm 2.3: its first phase consists in a plain execution of the same algorithm, until it has processed the first $|V|-d$ vertices in the degeneracy ordering, spending $\mathcal{O}(d(|V|-d)3^{d/3})$ time. The second phase launches TOMITA on the subgraph D induced by the last d vertices of the ordering. This will then find all cliques which are maximal in D, but not necessarily in G, thus the need to spend $\mathcal{O}(d(|V|-d))$ extra time for each clique to check its maximality in G. Putting it all together, the two phases will have a total cost of $\mathcal{O}(d(|V|-d)3^{d/3})$ steps, therefore this algorithm is theoretically optimal. However, as the authors point out, this is highly unlikely to be faster in practice, especially for relatively small values of d. This is because of the second phase which in practice consists in a scan of the entire graph G for each clique that can be very costly with respect to the non-optimal algorithm.

2.3.1 Parametrized Complexity

Algorithm 2.3 is very efficient in practice, although it is not worst-case optimal and, most of all, it is exponential. We can explain this phenomenon with the concept of *fixed-parameter tractability* mentioned at the beginning of the section.

Parametrized complexity is a branch of the vast computational complexity theory that have seen rising interest since the last years of the 90s. It works under the assumption that $\mathcal{P} \neq \mathcal{NP}$ and puts its focus on exponential time complexities, or more generally, on *superpolynomial* ones. Usually, complexity of algorithms is given as a function of the whole input size and, as we know, it can be exponential or worse but this intractability may come from a specific parameter of the algorithm and not directly from the size of its input. Therefore if that parameter, say k, is fixed to be a small number the algorithm can be efficient in practice because the "costly" part is still the polynomial one.

Formally, the complexity class FPT is defined as follows.

Definition 2.2. The complexity class **FPT** contains all the problems which admit a solution taking $f(k) \cdot |x|^{\mathcal{O}(1)}$ time, where k is a parameter of the algorithm and $|x|$ is the size of the input.

The function f is allowed to grow arbitrarily fast, while $|x|$ participates only polynomially.

Algorithm 2.3 belongs to a subclass of FPT called **FPL**, which stands for *Fixed-Parameter Linear* and it denotes the special case where the running time is $f(k) \cdot |x|$, i.e., *linear* in the size of the input $|x|$. In fact, recall that Eppstein et al.'s running time is $\mathcal{O}(d|V|3^{d/3})$ and thus linear in $|V|$, a dimension of the input graph. As we will see later in Chapter 4, the value of d for many real-world graphs is relatively small, hence we can expect the algorithm to be practically fast even for large graphs, provided they are sufficiently sparse.

2.3.2 Output-Sensitivity

Among all the properties that an algorithm can have, we find one that is particularly relevant when speaking of enumeration algorithms, and it is called *output-sensitivity*.

Definition 2.3 (Output-sensitive Algorithm)**.** An algorithm is *output-sensitive* if its running time can be expressed as a function of its output size rather than (or in addition to) the size of the input.

Taking into account the size of the output when analyzing an enumeration algorithm may help to give it more refined time bounds that can, in turn, differentiate its actual performances from other algorithms that have the same computational complexity in terms of the input size. Eppstein et al.'s algorithm significantly outperforms the other two variations of Bron-Kerbosch, despite having an exponential complexity like them. This could suggest that Eppstein et al. version of the algorithm is actually output-sensitive, but at the time of writing this is not known and it is currently subject of research.

What is known in the literature, however, is the so-called *delay* of these algorithms.

Definition 2.4 (Delay)**.** The *delay* of an enumeration algorithm is the time spent between the output of two consecutive solutions.

In our case this translates to giving a bound to the time between the output of two consecutive maximal cliques. It has recently been proven in [8] that Eppstein et al.'s algorithm has $\Omega(3^{|V|/6})$ delay for some class of graphs, as stated by the following Lemma.

Lemma 2.2 (Lemma 23 of [8])**.** *There exists an infinite family of graphs over n vertices for which the algorithm by Eppstein et al. has $\Omega(3^{n/6})$ delay.*

Speaking of the Tomita et al. variant, there are two results that are worth noting in this context. The first is a Lemma, proven in [8], about the time required by the algorithm to output a clique.

Lemma 2.3 (Lemma 24 of [8])**.** *The algorithm by Tomita et al. has cost $\Omega(|V|^3)$ on a complete graph $G = (V, E)$.*

This is a weaker result compared to the previous, as we can only say that a maximal clique will take at least cubic time to be enumerated, but nothing can be said on the actual delay of the algorithm. We also remark that Tomita et al.'s algorithm is designed to not explicitly output the cliques found but only a compressed representation of them, e.g. the number of cliques found, otherwise it would lose its worst-case optimality. There is, however, a very recent result for the algorithm by Tomita et al. that states that this algorithm has, like Eppstein et al.'s, exponential delay unless $\mathcal{P} = \mathcal{NP}$ [23].

One may then ask if there already exists an algorithm that is output-sensitive for maximal clique enumeration. The answer to this question is yes, in fact there exists an algorithm by Tsukiyama et al. [25] that is able to enumerate all maximal cliques in $\mathcal{O}(\alpha((|V|^2 - |E|)\,|E|))$ time, where α is the number of maximal cliques in a given graph $G = (V, E)$. This algorithm was born as a backtracking Maximal Independent Set[1] enumeration technique that requires $\mathcal{O}(|V|\,|E|)$ time per solution (in the same fashion of the bound we gave above for Tomita et al. algorithm), thus it can be applied to the complementary graph $\overline{G} = (V, \overline{E} = \{\{u, v\} \mid u, v \in V, \{u, v\} \notin E\})$ obtaining all the maximal cliques in the aforementioned time.

Another, more recent, algorithm that uses a similar recursive approach is the one by Makino and Uno [19], that has a combinatorial version with delay bounded by $\mathcal{O}(\Delta^4)$ where Δ is the maximum degree of the graph in input, and another version based on matrix multiplication which achieves a delay of $\mathcal{O}(|V|^{2.37})$ time. These delays sum up to, respectively, a $\mathcal{O}(\alpha\Delta^4)$ time and a $\mathcal{O}(\alpha\,|V|^{2.37})$ time algorithm.

Although these algorithms may seem appealing for their theoretical complexity lower than Bron-Kerbosch-based ones they suffer from their space requirements. It is known, in fact, that both Tsukiyami et al. and Makino-Uno algorithms require $\mathcal{O}(|V|^2)$ additional space [8], while all of the three versions of the Bron-Kerbosch technique presented in this Chapter take only $\mathcal{O}(n + d\Delta)$ space, which is linear in the size of the input graph

[1] A *Maximal Independent Set* in a graph is a subset of its vertices such that there are no edges between any pair of them, and no other vertex can be added to the set without violating this property.

and its degeneracy, thus better suited for large graphs. Moreover, since the number α of maximal cliques to be enumerated is not know *a priori* we cannot assume that these algorithms are faster alternatives over the others that follow the Bron-Kerbosch scheme.

Chapter 3

Extending Eppstein's Algorithm

In this chapter we extend the Eppstein et al. algorithm to efficiently solve Problem 1.4, which we recall it is the enumeration of all the maximal $k_{\geq}$-cliques. The method that we use as a baseline for our study is the straightforward adaptation of the algorithms presented in Chapter 2: we let the algorithm by Eppstein et al. complete its execution over a graph and whenever a new clique is found, i.e., when both P and X are empty, we check if the clique found (contained in R) has at least k vertices. If this is not the case the clique is discarded and no counter is incremented. Our goal, however, is to exploit as much as possible the structure and the properties of the problem to get a noticeable speedup by cutting branches of the search tree that will lead to cliques with less than k vertices. Therefore we develop strategies to further prune the search space of the algorithm, combining the information we can extract from its working sets P, R, X and the value of k.

It is important for these heuristics to be as efficient as possible, in order to properly take advantage of the reduction of work they offer. We discuss two main heuristics, a simple one based on the size of the sets P and R, and one based on the structure of the induced subgraph $G[P]$ at each recursive call. In what follows, the procedure Tomita is intended to be the one called by Algorithm 2.3 at line 5 with the additional parameter k to keep track of the clique size.

3.1 Size Heuristic

Our first heuristic comes directly from the need to keep only cliques larger than k vertices, in fact the following Proposition holds.

Proposition 3.1. *A necessary condition for a maximal clique C of at least k vertices to be discovered by Algorithm 2.3 is that* Tomita *is first called with $|P| \geq k-1$.*

Proof. Suppose that $|P| < k-1$. Tomita adds vertices to a clique only by moving them from P to R, and the size of P never increases during the recursion because of the intersection operations. Let y be the maximum possible clique size with the given P and R. We have $y = |P| + |R| < k - 1 + 1 = k$, the inequality coming from our hypothesis on

P. So it will not be possible to complete the execution of TOMITA with $|R| \geq k$, even by taking all vertices in P. ∎

Recall that P is first populated with the neighbors of the vertex being analyzed (from which comes the -1 in the inequality) that follow it in the degeneracy order. Using this fact we can entirely skip a vertex if it has not enough following neighbors to make up a clique of at least k vertices.

We can make a step further and prove the following, stronger, theorem.

Theorem 3.1. *Along the path followed by* TOMITA *in the search tree leading to a $k_{\geq}$-clique, it holds that $|P| + |R| \geq k$*

Before giving the proof to this theorem, we have to prove two additional results. To lighten the notation, we will refer with P_h to the cardinality of set P at level h of the recursion tree, and similarly we will call R_h the cardinality of R at the same level h. We assume that the path has length t and we call x_i the number of vertices removed from P by the intersection operation at stage i, thus $x_i \geq 1$ for all i because we always remove v from P.

Lemma 3.1. *If $P_t = 0$, then for all $h = 1, 2, \ldots, t$ $P_h = \sum_{i=h+1}^{t} x_i$*

Proof. Since $0 = P_t = P_0 - \sum_{i=1}^{t} x_i$, we have that

$$0 = P_0 - \sum_{i=1}^{t} x_i = P_0 - \sum_{i=1}^{h} x_i - \sum_{i=h+1}^{t} x_i.$$

Now note that $P_h = P_0 - \sum_{i=1}^{h} x_i$, so it is

$$0 = P_h - \sum_{i=h+1}^{t} x_i \Rightarrow P_h = \sum_{i=h+1}^{t} x_i.$$

∎

Lemma 3.2. *If $P_t = 0$, then for all $h = 1, 2, \ldots, t$, $P_h \geq t - h$.*

Proof. For the sake of contradiction, assume that $P_h < t - h$. This means that $t - h = P_h + r$ for some $r > 0$ and, by Lemma 3.1:

$$0 = P_h - \sum_{i=h+1}^{t} x_i = P_h - \sum_{i=1}^{t-h} x_i = P_h - \sum_{i=1}^{P_h+r} x_{i+h} = P_h - \sum_{i=1}^{P_h} x_{i+h} - \sum_{i=1}^{r} x_{P_h+i}$$

Since this holds for every i and $x_i \geq 1$, we have:

$$0 = P_h - \sum_{i=1}^{P_h} x_{i+h} - \sum_{i=1}^{r} x_{P_h+i} \leq P_h - \sum_{i=1}^{P_h} 1 - \sum_{i=1}^{r} 1 = P_h - P_h - \sum_{i=1}^{r} 1 = -\sum_{i=1}^{r} 1.$$

But $r > 0$, leading to a contradiction. So it is $P_h \geq t - h$. ∎

We now have all the tools to prove Theorem 3.1.

Proof of Theorem 3.1. Suppose to be at the root of the subtree originated from the invocation of TOMITA with a vertex u in R. By Proposition 3.1 it must be that $P_0+R_0 \geq k$, with $R_0 = 1$. Notice also that, since the algorithm discovers a $k_{\geq}$-clique, $R_t \geq k$ and $P_t = 0$ because both P and X must be empty. We then prove that for all $h = 1, 2, \dots, t$, $P_h + R_h \geq k$.

Assume, for the sake of contradiction, that $P_h + R_h < k$ for some h. Applying Lemma 3.2 we get

$$k > P_h + R_h \geq t - h + R_h.$$

By construction of TOMITA we know that for each h, $R_h = R_0 + h$, so it is

$$k > t - h + R_0 + h = t + R_0 = R_t$$

which is a contradiction, since we assumed that the clique discovered at step t was of size greater than k. This concludes the proof.

■

This result gives us the possibility to stop the recursion whenever $|P| + |R|$ fails to reach at least k, meaning that the vertex being moved to R has too few neighbors to contribute in a clique of size at least k. Combined with the above Proposition 3.1, we get a very fast and intuitive method to cut useless branches, in fact both of these checks can be done in $\mathcal{O}(1)$ time.

The modified procedures are listed in Algorithm 3.1 and 3.2. Note that, at implementation time, it is advisable to repeat the check on $|P| + |R|$ when forming $P \cap N(v)$, to avoid doing an extra recursive call that would terminate immediately.

Algorithm 3.1 Eppstein et al.'s algorithm modified according to our Size Heuristic

```
1: function EPPSTEIN(G, k)
2:     for all v_i ∈ V in a degeneracy ordering of G do
3:         P ← N(v_i) ∩ {v_{i+1}, v_{i+2}, ..., v_{|V|-1}}
4:         if |P| + 1 < k then
5:             continue
6:         end if
7:         X ← N(v_i) ∩ {v_0, v_1, ..., v_{i-1}}
8:         TOMITA(P, {v_i}, X, k)
9:     end for
10: end function
```

3.2 Edge Count Heuristic

The previous heuristic, in some sense, represents just the natural way of adapting the algorithms of Chapter 2 to list only $k_{\geq}$-cliques, but we want to dig deeper in the proper-

Algorithm 3.2 Tomita et al.'s algorithm modified according to our Size Heuristic

```
1: function TOMITA(P, R, X, k)
2:     if P = ∅ and X = ∅ then
3:         R is a maximal clique
4:         return
5:     end if
6:     if |P| + |R| < k then
7:         return
8:     end if
9:     u ← a vertex in P ∪ X with maximum |P ∩ N(u)|
10:    for all v ∈ P \ N(u) do
11:        TOMITA(P ∩ N(v), R ∪ {v}, X ∩ N(v), k)
12:        P ← P \ {v}
13:        X ← X ∪ {v}
14:    end for
15: end function
```

ties that we can exploit to further cut the number of recursive calls the algorithms have to make.

While we speak primarily of recursive calls, we know that they are not the only thing that takes time to be done, in fact a very costly operation lies in the intersection of P and X with the neighborhood of a vertex. The authors of [9] took this problem seriously and developed an elegant implementative solution which we will present later in Chapter 4, for the time being we just need to say that we are going to attach our heuristic to two distinct moments during the execution of the algorithm: one is during the pivoting phase and the other one is during the computation of the intersection $P \cap N(pivot)$.

This second strategy focuses more on the internal structure of $G[P]$, particularly how, and how many edges connect the vertices belonging to P at any stage of the algorithm. A trivial property of a k-clique is that a complete subgraph of k vertices in an undirected graph $G = (V, E)$ has $k(k-1)/2$ edges. An immediate consequence of this property gives us a necessary condition on P to check, but before stating it we need to prove the following:

Claim 3.1. *Let $G = (V, E)$ be a graph. A set $C \subseteq V$ is a clique in G if and only if for all $C' \subset C$, C' is a clique.*

Proof. $\Rightarrow$: This is trivial because by definition of clique every $v \in C$ is connected to all other vertices of C, hence any possible subset of C is a clique.

$\Leftarrow$: Suppose now that any subset of C is a clique. Since C is a subset of itself, C is a clique. ■

Proposition 3.2. *At any stage of* TOMITA*, if the induced subgraph $G[P]$ has less than $\frac{(k-|R|)(k-|R|-1)}{2}$ edges, then any $G[C \cup R]$ with $C \subseteq P$ does not contain a k-clique.*

Proof. Suppose that $C \cup R$ forms a clique in G with $C \subseteq P$ and that $|P| >= t, |R| = q, t + q = k$. By Claim 3.1 we know that any subgraph of a clique is a clique itself, and since $C \subseteq C \cup R$, C forms a clique in G. However, by hypothesis, $G[P]$ has less than $(k - q)(k - q - 1)/2$ edges, thus it cannot contain any clique of $k - q$ vertices, leading to a contradiction: $C \cup R$ was not a clique. ■

We can use this fact when computing $P' = P \cap N(v)$: we count the edges of the induced subgraph $G[P']$ and if they are strictly less than $(k - |R|)(k - |R| - 1)/2$ we can safely discard P' and proceed with the next iteration of the for loop, effectively skipping the corresponding (and subsequent) recursive calls. With the appropriate data structures this check can be done in $\mathcal{O}(|P'|)$, more details on this are given later in Section 4.1.

In a similar fashion there is another check we can do on P, this time involving the degree of each vertex belonging to it. We will exploit the following simple property.

Proposition 3.3. *In order for a graph $G = (V, E)$ to contain a clique on k vertices, there must exist k vertices with degree of at least $k - 1$.*

Proof. Follows immediately from the definition of clique in which every vertex must be adjacent to the other $k - 1$. ■

We can then plug this result into our heuristic by doing the following observation.

Observation 3.1. If $G[P]$ does not contain at least $k - |R|$ vertices with degree at least $k - |R| - 1$, then $G[P \cup R]$ does not contain a clique of (at least) k vertices.

We can check this condition also in $\mathcal{O}(|P|)$ time, details on how are given later in Section 4.1.

The pseudocode integrated with the full set of optimization is shown in Algorithm 3.3 and 3.4. We can see that it is convenient to perform every check of both heuristics whenever the sets change in order to cut the most recursive calls possible, since even the tiniest of changes in P can lead to a significant change in the structure of $G[P]$.

Note that the checks implemented in the EPPSTEIN procedure are important because they can cut a whole subtree by skipping a vertex entirely. The ones included in TOMITA, on the other hand, allow for an early stopping of the growth of the same subtree in two ways. One way, which is preferable, is to directly exclude a vertex from the list of candidates $P \setminus N(u)$, by checking the condition on the number of "good neighbors" every vertex on that list has. This will avoid the work on computing P' and, of course, subsequent recursive calls. We can say that in the recursion tree we avoid the creation of a new node. On the other hand, it can happen that a recursive call is instantiated and then immediately closed. This depends deeply on the structure of the graph and cannot be easily inferred until P' is formed.

Algorithm 3.3 Eppstein et al. algorithm modified according to both our heuristics

```
1: function EPPSTEIN(G, k)
2:     for all v_i ∈ V in a degeneracy ordering of G do
3:         P ← N(v_i) ∩ {v_{i+1}, v_{i+2}, ..., v_{|V|-1}}
4:         if |P| + 1 < k then
5:             continue
6:         else if G[P] contains less than (k − |R|)(k − |R| − 1)/2 edges then
7:             continue
8:         else if G[P] contains less than k − |R| vertices with degree ≥ k − |R| − 1 then
9:             continue
10:        end if
11:        X ← N(v_i) ∩ {v_0, v_1, ..., v_{i-1}}
12:        TOMITA(P, {v_i}, X, k)
13:    end for
14: end function
```

3.3 Other Approaches

The previous methods are fast and easy to implement, but there are several other techniques that in principle could be exploited. While their cost is fairly low in absolute terms, when plugged into the algorithms they may lead to a degradation of performance. Additionally, in our experiments, they didn't provide a significant decrease in the number of recursive calls when paired with the strategies explained above. However, we believe they are important to give a more complete view of what can be done and what we have tested during the development of our strategy.

3.3.1 Chromatic Number

One of the traditional property of graphs is the *Chromatic Number*, defined as the minimum number of colors needed so that every adjacent vertex is colored differently, and denoted with $\chi(G)$, where $G = (V, E)$ is the usual graph definition. A graph with no edges, for example, has $\chi(G) = 1$, while a bipartite graph is the only graph that has $\chi(G) = 2$ and planar graphs have $\chi(G) \leq 4$. The connection between $\chi(G)$ and our problem is given by the following property.

Proposition 3.4. *Let $G = (V, E)$ be a graph and $K \subseteq V$ a clique of k vertices. Then $\chi(G) \geq k$.*

Proof. Since every vertex of a clique is adjacent to each other, the only way to color the clique vertices is to use a different one for each vertex, thus k colors. Since K represents a subgraph of G, we have that the complete G must have a chromatic number of at least k. ■

Computing $\chi(G)$ for a generic graph is NP-Hard [15], thus many lower and upper bounds have been designed to circumvent this limit, and we can use them to get an

Algorithm 3.4 Tomita et al. algorithm modified according to both our heuristics

```
1: function TOMITA(P, R, X, k)
2:     if P = ∅ and X = ∅ then
3:         return                                              ▷ R is a maximal clique
4:     end if
5:     if |P| + |R| < k then
6:         return
7:     else if G[P] contains less than (k − |R|)(k − |R| − 1)/2 edges then
8:         return
9:     else if G[P] contains less than k − |R| vertices with degree ≥ k − |R| − 1 then
10:        return
11:    end if
12:    u ← a vertex in P ∪ X with maximum |P ∩ N(u)|
13:    Remove from P \ N(u) all vertices that have less than k − |R| − 1 neighbors in
       P with degree ≥ k − |R| − 2
14:    for all v ∈ P \ N(u) do
15:        P′ ← P ∩ N(v)
16:        R′ ← R ∩ N(v)
17:        Repeat all previous checks on P′ and R′
18:        if All checks pass then                             ▷ Same checks of lines 5-11
19:            TOMITA(P′, R′, X ∩ N(v), k)
20:        end if
21:        P ← P \ {v}
22:        X ← X ∪ {v}
23:    end for
24: end function
```

insight on what the set P has to offer at any stage of the modified Eppstein et al. algorithm, in particular:

Claim 3.2. *If $\chi(G[P]) < k - |R|$, then $G[C \cup R]$ does not contain a $k_{\geq}$-clique for any $C \subseteq P$.*

Proof. Using the contrapositive of Proposition 3.4 we know that all cliques in $G[P]$ are of size strictly smaller than $k - |R|$. Consider now vertices in R that form a clique in G by definition of R itself: when attaching these vertices to the maximum clique of P (call it C) the size of the latter is bounded above by $k - |R| + |R| = k$. We conclude that the maximum size possible of any clique in $G[P \cup R]$ is $k - 1$ and the thesis follows. ■

This result gives a straightforward method to stop the recursion whenever $\chi(G[P])$ becomes strictly smaller than $k - |R|$. As pointed above, the exact computation of the chromatic number is too costly to afford, thus we must resort to using an upper bound, which does not hinder the correctness of Claim 3.2. We need to compute this upper bound along with the other checks already present in Algorithm 3.4 at every stage of

Algorithm 3.5 Computing η, an upper bound for $\chi(G)$.

```
1: function ComputeEta(G = (V, E))
2:     Compute the degree δ_i ∀v_i ∈ V
3:     for i ← 0 to |V| do
4:         tab ← new array with δ(u) for all u ∈ N(v_i)
5:         Sort tab in non-increasing order
6:         ρ_i ← 0, stable ← false, j ← 0
7:         while !stable and j ≤ δ_i do
8:             if tab[j] > ρ_i then
9:                 ρ_i ← ρ_i + 1
10:            else
11:                stable ← true
12:            end if
13:            j ← j + 1
14:        end while
15:    end for
16:    Sort ρ in non-increasing order
17:    η ← 0, stable ← false, i ← 0
18:    while !stable and i ≤ |V| do
19:        if ρ_i ≥ η then
20:            η ← η + 1
21:        else
22:            stable ← true
23:        end if
24:        i ← i + 1
25:    end while
26:    return η
27: end function
```

the recursion in a way that is both fast and as accurate as possible. Our experiments were conducted using one out of three upper bounds proposed in [22], defined by the theorem below.

Theorem 3.2 (Theorem 3 of [22])**.** *For any simple, undirected graph $G = (V, E)$, $\chi(G) \leq \eta$, where η is the largest number of nodes with a degree $\delta \geq \eta$ that are adjacent to at least $\eta - 1$ other vertices, each with a degree of at least $\eta - 1$.*

The authors of [22] have shown that η is the tightest upper bound out of a set of eight different bounds considered, and it can be computed in $\mathcal{O}(\max\{|E|\log|E|, |V|\log|V|\}$ steps, a fairly low complexity speaking in absolute terms, with Algorithm 3.5.

While this approach is appealing from a theoretical point of view, our experiments showed a degradation of performance with little benefits in terms of recursive calls saved when paired with the other heuristics proposed above. The loss in performance is, alongside the higher complexity with respect to our $\mathcal{O}(|P|)$, mainly due to the number

of sort operations that have to be done each time. Also from the implementation point of view, our heuristics integrated better with the existing code, allowing them to be computed almost for free in practice.

3.3.2 Clique Number

Another property of graphs is the so-called *clique number*, defined as the cardinality of the maximum clique in G. More formally:

Definition 3.1 (Clique Number)**.** Let $G = (V, E)$ be a graph. The number of vertices in the largest maximal clique of G is called the *clique number* of G, and it is denoted by $\omega(G)$.

As for the chromatic number, the clique number is hard to compute, thus it is necessary to use bounds once more. The idea behind the use of the clique number is very similar to that of the chromatic number.

Claim 3.3. *If $\omega(G[P]) < k - |R|$, then $G[C \cup R]$ does not contain a $k_{\geq}$-clique, for any $C \subseteq P$.*

Proof. If the clique number for $G[P]$ is less than $k - |R|$, by definition it means that the vertices in P can be part of a clique of at most size $k - |R| - 1$. Adding the vertices of R that form a clique of $|R|$ size, we get a clique of strictly less than $k - |R| + |R|$ vertices, hence for any $C \subseteq P$, $G[C \cup R]$ cannot contain a clique of more than k vertices. ■

We shall then compute an upper bound on $\omega(G[P])$ at each stage of the algorithm, and whenever this bound is less than $k - |R|$, we can stop recurring.

This time the bound we used is purely analytical, and very easy to compute.

Lemma 3.3 ([1])**.** *For any simple, undirected graph $G = (V, E)$, it holds that*

$$\omega(G) \leq \frac{3 + \sqrt{9 - 8(|V| - |E|)}}{2}.$$

Although computable in $\mathcal{O}(1)$ time, this bound is practically loose and has given almost no benefits during our experiments, so we decided not to use it. There exist other bounds for $\omega(G)$ that are more accurate, but are based on the spectral analysis of A, the adjacency matrix associated with G. However, since we are dealing with very large graphs, handling matrices of up to millions of entries is not convenient both for time and space concerns.

3.3.3 Degeneracy and Second Edge Count Heuristic

We know from Chapter 1 that a graph with degeneracy d can only contain cliques of size up to $d + 1$, thus we could apply this reasoning also to $G[P]$ with the following observation.

Claim 3.4. *If $G[P]$ has degeneracy strictly less than $k - |R| - 1$, then $G[C \cup R]$ cannot contain a $k_{\geq}$-clique for any $C \subseteq P$.*

Proof. Same reasoning as for Claim 3.3. ■

What we need to do then is to apply Algorithm 1.1 to the vertices belonging to P at each recursive step and stop the process if $G[P]$ is found to have a degeneracy less than $k - |R| - 1$. While the algorithm for computing the degeneracy value of a graph takes linear time in the size of the subgraph considered and therefore seems appealing, it is more expensive than the other heuristics adopted with their $\mathcal{O}(|P|)$ cost, and again the loss in performance was not amortized by a significant decrease in the number of recursive calls during our experiments.

We can also think of a more involved version of our *edge count* heuristic, i.e., one that digs deeper into the structure of $G[P]$ in order to cut more useless subtrees.

Observation 3.2. Suppose that v is the vertex currently being moved from P to R. If there is a clique involving $R \cup \{v\}$, then there exists a vertex in $P \cap N(v)$ such that it is neighbor to other $k - |R| - 2$ neighbors of v in P.

This search is appealing because, along with a way to cut useless branches if such a vertex is not found in P, it also provides a good vertex to be added to R at the next recursion level of the algorithm, i.e. one with many neighbors in P. Unfortunately, this technique comes with a drawback. In fact, applying this strategy would require $\mathcal{O}(|P \cap N(v)|^3) = \mathcal{O}(d^3)$ time using basic adjacency lists, a cost too high to afford at each iteration. Despite this we implemented this strategy using ad hoc data structures like hash tables in order to answer queries of adjacency fast, but as expected the impact on time performance was significant, therefore we chose not to continue on this path.

3.3.4 Erdös and Turán Theorems

So far we have discussed necessary conditions, i.e., conditions that gave us a way to cut useless branches of the search tree that the Bron-Kerbosch algorithm has to visit in order to enumerate all cliques (of arbitrary size). In this subsection we want to give a glance at another type of conditions, the sufficient ones. Due to their nature they cannot be used to shrink the search space we have to visit, but they might be used in future algorithms or future improvements of Eppstein et al.'s algorithm in appropriate ways. Nevertheless we want to briefly present them for completeness' sake.

Erdös and Turán have conducted extensive research on graph theory and proved many results about their properties, among which we find the existence of complete subgraphs (cliques) in arbitrary graphs. The first result that we are going to discuss is the following theorem by Turán.

Theorem 3.3 (Turán [2])**.** *Let $G = (V, E)$ be a simple, undirected graph with $|V| = n$ and let $r \in \mathbb{N}$ be a positive integer such that $r \leq n$. Suppose that $n \equiv l \pmod{r-1}$. Then at most one of these can hold:*

1. $|E| > \left|n^2 - l^2\right| \cdot \frac{r-2}{2(r-1)} + \binom{l}{2}$
2. *G does not contain a clique of r vertices.*

This is very powerful, since we have a way to tell if a graph *surely* has a clique of any given size r, in fact, since only one of the two statements can be true at a time if the number of edges is larger than that threshold then we know that a r-clique surely exists. As we said above, this is a sufficient condition and the theorem does not tell us anything about the non-existence of r-cliques in G. This is because if the $|E|$ threshold is not met, nothing can be said on the second condition due to the fact that they are not mutually exclusive, so there can exist a graph in which the number of edges is less than or equal to the threshold while at the same time it has a r-clique. In fact it is known that such a graph exists, and an example is reported in [2].

Turán's Theorem can be reformulated in different ways, one for instance is an easier-to-read version, and one is rephrased to use the edge density of a (undirected) graph, defined as $\rho_2(G) = 2|E|/\binom{|V|}{2}$. We state these two variants below.

Theorem 3.4 (Turán)**.** *Let $G = (V, E)$ be a simple, undirected graph that does not contain a $r + 1$ clique, with $r \in \mathbb{N}, r < |V|$. Then $|E| \leq \left(1 - \frac{1}{r}\right) \frac{|V|^2}{2}$.*

Theorem 3.5 (Turán)**.** *Let $G = (V, E)$ be a simple, undirected graph and $r \in \mathbb{N}, r \leq |V|$. If $\rho_2(G) > 1 - \frac{1}{r-1}$, then G contains a r-clique.*

Theorem 3.5 has a stronger version given by Erdös:

Theorem 3.6 (Erdös-Turán)**.** *Let $G = (V, E)$ be a simple, undirected graph and $r \in \mathbb{N}, r \leq |V|$. If $\rho_2(G) > 1 - \frac{1}{r-1}$, then G contains $\Omega((|V|/(r-1))^{r-2})$ r-cliques.*

What we know now is that if G is sufficiently dense then not only it will contain a r-clique, it will have a high number of r-cliques. Again, this result cannot be used to our aim, but it can instead serve as a "guide" through the search tree, to drive the algorithm towards paths that will generate many desired cliques. We remark, however, that the other subtrees where the hypotheses are not verified must be checked too in our case.

Successive results established a connection between the presence of a clique and the degree of vertices in G, such as the following theorem.

Theorem 3.7 (Zarankiewicz [2])**.** *Let $G = (V, E)$ be a simple, undirected graph and let $r \in \mathbb{N}$ be a positive integer such that $r \leq |V|$. Then at most one of these can hold:*

1. $\min_{v \in V} \delta(v) > \left[|V| \frac{r-2}{r-1}\right]$
2. *G does not contain a clique of r vertices.*

As with Turán's theorem, we have again conditions that are not mutually exclusive, but this time the structure of G is being used in a deeper fashion with little time cost ($\mathcal{O}(|V|)$ complexity for verifying the first point).

The last theorem that we present here is due to Andrásfai, Erdös and Sós, and it binds together, conceptually, all the previous results plus a condition on the chromatic number.

Theorem 3.8 (Andrásfai, Erdös and Sós [2])**.** *Let $G = (V, E)$ be a simple, undirected graph and let $r \in \mathbb{N}$ be a positive integer such that $3 \leq r \leq |V|$. Then at most two of the following can hold:*

1. $\chi(G) \geq r$
2. $\min_{v \in V} \delta(v) > |V| \frac{3r-7}{3r-4}$
3. *G does not contain a clique of r vertices.*

The reasoning behind the application of this statement is the same as all previous theorems: if the first two conditions are met, then G must contain at least one clique of r vertices, because the third statement must be false. However we have to deal with the computation of the chromatic number, so we shall use instead a lower bound that can be efficiently computed. An example of a simple, fast to compute, lower bound is given in [4] and it involves only the degree of the vertices of G. Clearly there are tighter, and more recent, lower bounds but the one of [4] makes for a good starting point.

Chapter 4

Evaluation and Experiments

This chapter presents the experiments that we conducted in order to assess the quality of the strategies described in Chapter 4. We integrated them directly into the C++ implementation by Darren Strash on GitHub[1] of the Eppstein et al.'s algorithm, and run extensive experiments.

We first give the implementation details, then we describe the dataset that we used in Section 4.2, and finally we proceed to present and discuss the results obtained.

4.1 Implementation Details

While the pseudocode of the Eppstein et al.'s algorithm is quite simple, implementing it in C++ real code is challenging and it requires a careful attention to the details to actually get the good performances promised. For this reason we preferred to work and extend the code given by the authors and publicly available on GitHub, available under the GNU Public License (GPL) v3. We briefly discuss the details of the strategy they chose to adopt with a focus on the pivot search and the intersection operations. Then we proceed to explain how we integrated our heuristics of Chapter 3 into their code. A comprehensive look of Eppstein et al. implementation is given in Section 4 of [9].

The two most expensive operations of the algorithm are the pivot selection and the intersection of P and X with $N(v)$. One could implement the former with a simple scan of the neighbors of each $P \cup X$ vertex, counting how many of them are in P but this would make the whole algorithm require $\mathcal{O}(d^2|V|3^{d/3})$, a factor of d slower than the claimed running time in Theorem 2.5.

With this in mind, the authors have made use of additional data structures that requires only $\mathcal{O}(|V| + |E|)$ additional space in total. First, we have a reverse lookup table that allows us to know to which set does a vertex v belong at any given time during the execution. This is implemented with a direct addressed table, so it is an array of $|V|$ positions. Secondly, the sets P, R and X are represented using a single $|V|$ array, which is subdivided into three regions using three indices: set X occupies the

[1] `https://github.com/darrenstrash/quick-cliques`

Algorithm 4.1 Choosing the pivot vertex from $P \cup X$

```
 1: function ChoosePivot(A)        ▷ We assume A is the array storing the modified
    adjacency lists
 2:     max ← 0
 3:     pivot ← A[beginX]
 4:     for all i ∈ [beginX, beginR) do
 5:         j ← 0
 6:         while j < |A[i]| do
 7:             if A[i] ∉ P then              ▷ Checked with the reverse lookup table
 8:                 break
 9:             end if
10:             j ← j + 1
11:         end while
12:         if j > max then
13:             max ← j
14:             pivot ← A[i]
15:         end if
16:     end for
17:     return pivot
18: end function
```

first positions of the array until the first vertex of P, which then occupies the "internal" positions until vertices in R that find place in the last portion of the table. The address of this array is passed to every recursive call and every other procedure that works with these sets, along with the reverse lookup table and with a third data structure. The most important data structure is what represents the *modified adjacency lists* of the induced subgraph $G[P \cup X]$. They act like standard adjacency lists, but they are kept sorted in a way such that the neighbors in P of each vertex are always at the beginning of the lists, an invariant that is particularly useful during the pivot selection phase. Since the sets X and P are stored, in this order, contiguously in the same array, the pivot can be easily selected with Algorithm 4.1 which has linear cost in the size of the induced subgraph $G[P \cup X]$. In particular, because of the aforementioned invariant, it is possible to stop iterating over an adjacency list whenever we encounter a neighbor that is not in P.

When a vertex v is moved to the R set, we must perform the intersection of P and X with the neighborhood of that vertex. This can be done in a simple and efficient way using the above array that represents the three sets: we scan it and swap vertices in $X \cap N(v)$ with those at the right boundary of the X region, where the P one begins, while vertices in $P \cap N(v)$ are kept at the beginning of the P region (to its left). Figure 4 shows an example of this strategy.

With this strategy we can reuse the same array at each recursive call without having to create and populate a new one every time, provided that when the recursion unfolds we move back to P those vertices that were moved to X from deeper calls. This comes with practically no cost since we have the list of vertices we have iterated on during the

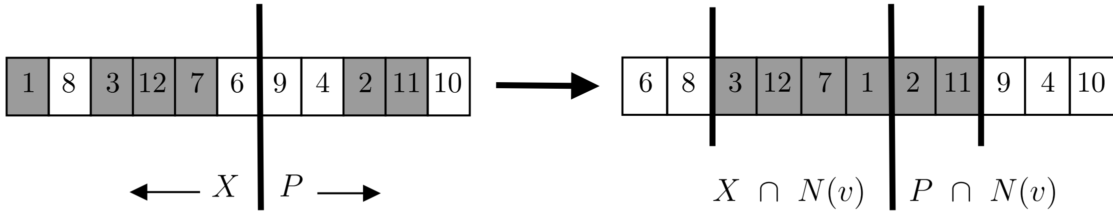


Figure 4: An example of the array used to represent the sets X and P contiguously. Highlighted vertices represent the neighbors of v. The array on the right has been rearranged to exclude non-highlighted vertices in preparation to the next recursive call. Figure reconstructed by the one originally found in [9].

for loop, and few swaps will suffice to restore the previous state of the sets. An important thing to be noted is also that the modified adjacency lists are kept sorted with respect to the presence or not in the current P by moving the correct vertices to the beginning and not with a classical sort operation. It happens, in fact, that when recurring the number of neighbors in P for each vertex diminishes so that lesser swaps will be required the deeper we go in the recursion tree. In Figure 5 we can see this strategy at work: notice how the search for a neighbor in P of any vertex can stop as soon as we exit the grey region (i.e. we encounter a vertex not in P) of its adjacency list.

We can now proceed to explain how we modified the code in order to include our strategy. The check on the size of P and R given by Theorem 3.1 is pretty straightforward: at the outermost level of the recursion, corresponding to the for loop of Algorithm 3.1, we check if v_i has enough neighbors to its right (i.e. that follow it in the degeneracy ordering). If this is not the case, we completely skip this vertex and continue with the next. At the beginning of each recursive call, and also immediately after the call to the procedure that moves a candidate vertex to R and performs the intersection of its neighborhood with P and X, we check if $|P| + |R| < k$ and we either close the recursive call or we move the vertex from R to X and proceed with the next candidate.

We plugged the Edge Count heuristic into three different regions of code. First is the time when sets P and X are formed for each vertex in the degeneracy order, after having checked if it has at least $k-1$ neighbors on its right. Then we apply the heuristic during the pivoting phase, and we remove from the list of candidates to iterate through, namely $P \setminus N(pivot)$, those vertices that do not have enough neighbors with a high degree in $G[P]$, according to our Observation 3.1. Finally, we do another check each time a vertex is moved to R from the list of candidate, thus after computing the intersection operation on P to see if that vertex leads to a configuration of $G[P]$ that cannot contain a clique of the required size.

These checks, as outlined in Section 3.2, can be done in $\mathcal{O}(|P|)$ time by making use of the aforementioned modified adjacency lists developed by the authors of [9] in the following way. Thanks to this particular data structure that is shared among all the functions of the code, it is sufficient to loop through the portion of array that covers the P set and check how many neighbors are also in P: this way we can do both parts

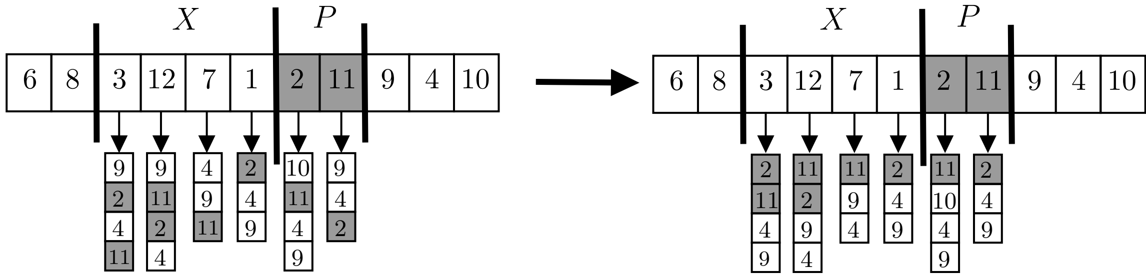


Figure 5: The modified adjacency list representation used by Eppstein et al. in their code. Each vertex in $P \cup X$ has an array of neighbors in P. When an intersection operation is done in preparation for a recursive call, vertices in $P \cap N(pivot)$ are swapped at the beginning of each list to speed up successive computations. Figure rebuilt from [9].

of the heuristic at the same time, i.e., we can count the number of vertices with high degree in P and also the total number of edges of $G[P]$. This is done in a stand-alone fashion, after the rest of the considered function, in the first two cases (pivoting and filling P and X for the first time), and in an incremental way during the moving to R of a vertex. In fact, since this action causes the update of P, we do the edge counting while rearranging the modified adjacency list for each vertex, and whenever we finish this action for a vertex we immediately check if it has enough neighbors in P with high degree. All of this introduces minimal overhead: the only parts that actually consumes $\Theta(|P|)$ time are the first two, while the third, during the moving of a vertex to R comes actually for free since it just suffices to increment two counters in the body of the existing loop that updates P, one for the edges, one for the high-degree neighbors spending $\mathcal{O}(1)$ additional time.

We also have inserted a simple timeout feature that exploits Unix signals, in particular SIGALRM, in order to have the possibility to stop the algorithm execution after a chosen amount of time and get the results that it was able to compute in the meantime, then exit cleanly.

Finally, the experiments were conducted on a commodity Linux machine equipped with a dual core Intel processor @ 2.3GHz and 4GB of memory, running Ubuntu 18.04 LTS. The machine was used solely for this purpose and no program other than the default processes was running at the same time. The code was compiled with `CLANG++` version 8, with the `-O3` optimization flag.

4.2 Dataset

The dataset we used is an extension of the one used by Eppstein et al. in [9], which in turn was the same dataset used by Tomita et al. for testing their algorithm in [24] with the only change in the random graphs that were not available and thus were regenerated with the same parameters. The dataset used in [9] is publicly available on Darren

Strash's personal website[2] and it consists in the following family of graphs:

- *BioGRID*: The Biological General Repository for Interaction Datasets data, version 3.0.65. Consists of protein-protein interaction networks with up to thousands of vertices.
- *MarkNewmann*: A database of mainly social networks curated by Newmann.
- *Pajek*: Large social and bibliographic networks.
- *Snap*: A sample of graphs from the Stanford Large Network Dataset Collection. It includes road networks, a co-purchasing network from Amazon.com data, social networks, email networks, a citation network, and two Web graphs.
- *DIMACS*: A dataset used for a DIMACS challenge, it contains algorithmically generated graphs specifically designed for testing clique-finding algorithms.
- *Random*: Tomita et al. generated some random graphs, i.e. Erdös-Renyi graphs, with variable edge probabilities that were not directly available, thus Eppstein et al. generated other random graphs with the same parameters.

For details and pointers to explanations of the single datasets, refer to [9]. We also have generated additional synthetic datasets to better understand the behavior of the algorithm equipped with our heuristic.

- *Loose*: Graphs containing a fixed number of cliques of a given size, loosely connected to each other. Given the number of nodes n and the clique size q, $\lfloor n/q \rfloor$ isolated cliques of q nodes are generated. Then each of the n nodes is connected to a number of neighbors chosen uniformly at random between 0 and $q-2$.
- *Regular*: Random graphs where each node has exactly d neighbors. Generated using an implementation of Kim and Vu algorithm [16].
- *Scalefree*: Graphs generated using a modification of the Barabási-Albert algorithm by Holme and Kim [13] in which there is a fixed probability p such that after the insertion of an edge, another edge may be added to form a triangle. This allows us to get more clustered graphs, i.e., with a higher average clustering coefficient[3] with respect to the standard Barabási-Albert algorithm.
- *Complementary*: Very dense graphs with a degeneracy value d close to $|V|$, used as "stress tests" for the algorithm. They are obtained by picking the complementary graph of small, sparse (i.e. low degeneracy), real-world networks. The complementary graph is defined, given a graph $G = (V, E)$, as $\overline{G} = (V, \overline{E} = \{\{u, v\} \mid u, v \in V, \{u, v\} \notin E\})$

[2] `http://www.ics.uci.edu/~dstrash/data.tar.gz`

[3] Given a vertex, its clustering coefficient is defined as the number of links between its neighbors divided by the maximum possible number of those links. The *average clustering coefficient* for a graph is defined as the average of this measure over all verticess of the graph.

4.3 Experimental Study and Discussion

In this section we present and discuss the experiments we conducted in the environment defined above.

Before we start the discussion we point out that, as noted in Chapter 3, we also implemented all the techniques listed as "other approaches" in Section 3.3, from the computation of the degeneracy of each induced subgraph $G[P]$ to the upper bounds on the chromatic number and clique number, to the second edge count heuristic using hash tables. However, the results we obtained showed little to no improvement both on the number of recursive calls saved and on the time spent. Sometimes the overhead introduced was high enough to drastically worsen the performances with respect to the baseline algorithm by Eppstein et al. For these reasons we chose not to report their results in this thesis, focusing only on the parts that produced actual improvements over the starting algorithm.

We give our results both in tabular and graphic form, showing the comparison among the three versions of the algorithm (no heuristic, size heuristic, size and size plus edge count heuristic) with respect to the time spent, in seconds, and the total number of recursive calls made. More in detail, we show:

1. The size of the input graph, namely the number of vertices and edges.
2. The degeneracy value of the graph.
3. The value of k (minimum maximal clique size) and the number of $k_{\geq}$-cliques found.
4. The time, in seconds, spent by: Eppstein et al.'s algorithm with no heuristic, Eppstein et al.'s algorithm equipped with our Size Heuristic, Eppstein et al.'s algorithm equipped with both Size and Edge Count Heuristics. These are averaged over five executions.
5. The number of recursive calls made by each of the aforementioned algorithms.

The first two points of the list describe the dataset giving information about its composition, and can be seen in Table 4.1 and Table 4.2.

Dataset	$\|V\|$	$\|E\|$	d
`BioGRID-Yeast`	6008	156945	64
`Marknewman-Astro`	16706	121251	56
`Pajek-Daysall`	13308	148035	73
`Random-100` ($p = 0.9$)	100	4473	81
`Random-500` ($p = 0.5$)	500	62571	225
`Random-1000` ($p = 0.3$)	1000	149998	266
`Snap-BerkStan`	685231	6649470	201
`DIMACS-Johnson-16-2-4`	120	5460	91

Table 4.1: Parameters of a subset of the graph dataset used by Eppstein et al. in [9].

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	d
`Complementary-chicagoRegional`	1468	1075480	1455
`Complementary-hamsterster`	2427	2927320	2315
`Complementary-mayaan`	1227	749743	1198
`Loose-26183-505` (clique size 505)	26183	12915990	738
`Loose-1234-503` (clique size 503)	1234	439836	593
`Loose-5783-407` (clique size 407)	5783	149998	584
`Regular-582` (582-regular graph)	17391	5060781	582
`Scalefree-2847-1245` ($p = 0.8$)	2847	1994490	1002
`Scalefree-1574-210` ($p = 0.8$)	1574	282833	208

Table 4.2: Parameters of a subset of our synthetic dataset.

We can see how the dataset used in [9] is generally made up of smaller and sparser graphs than the ones found in our synthetic dataset, but this will not always result in worse performances in all the graphs from the latter. In fact their particular structural properties will affect the performances of all the three variants of the algorithm. We may now proceed to discuss the results found in Table 4.3 and Table 4.4 and plotted in Figure 6 and Figure 7.

4.3.1 Recursive Calls Analysis

We begin by focusing on the number of recursive calls made. This number is the largest among all graphs for the pure version of Eppstein et al. algorithm, as it blindly scans the whole graph in input with no particular strategy. When heuristics come into play, however, this number does not immediately decrease for small values of k: this is to be expected because the heuristics were thought for finding large cliques and thus they can hardly be applied in cases with k close to two, the same implicit k that the Eppstein et al. algorithm works with.

The impact of our work becomes noticeable with the increase of k and the consequent decrease of cliques to be found: the number of recursive calls cut by the heuristics is large when comparing our Edge Count heuristic with the baseline algorithm, sometimes even zero recursive calls are sufficient to correctly conclude that there are no cliques of size greater than the given parameter k to be found. In other cases some recursive calls are still needed before terminating the execution, but this number is very low.

We can also see how the size heuristic alone is not sufficient to cut a considerable amount of calls and that the most savings come from our other one, which exploits more the internal structure of the subgraphs analyzed. This remains true even for the extreme case of the *Complementary* dataset, when the timeout is reached the highest number of cliques found is still given by the size plus edge count heuristic version along with the most number of recursive calls made (note that when a timeout occurs, it is better to have done more work, in contrast to when there is not a timeout). All this together confirms the soundness and the efficacy of our strategy.

Dataset	k	Cliques Found			Time (s)			Recursive Calls		
		Standard	S	S + E	Standard	S	S + E	Standard	S	S + E
BioGRID-Yeast	5	**619 933**			**1.32**	1.35	1.36	1 874 276	1 833 200	**1 753 254**
BioGRID-Yeast	20	**6 370**			1.32	0.79	**0.40**		265 474	**71 177**
BioGRID-Yeast	35	**0**			1.32	0.42	**0.15**		56 848	**722**
BioGRID-Yeast	50	**0**			1.32	0.27	**0.12**		20 134	**0**
Marknewman-Astro	4	**9 639**			**0.14**	0.15	0.15	66 080	60 694	**57 999**
Marknewman-Astro	30	**73**			0.14	0.07	**0.07**		3 165	**2 597**
Marknewman-Astro	57	**1**			0.14	**0.05**	0.05		**56**	**56**
Pajek-Daysall	10	**1 975 616**			**2.46**	2.95	3.06	53 992 739	5 189 077	**5 063 541**
Pajek-Daysall	28	**42**			2.46	1.10	**0.55**		294 531	**65 808**
Pajek-Daysall	50	**0**			2.46	0.34	**0.13**		16 718	**0**
Random-100 ($p = 0.9$)	8	**240 998 654**			**186.74**	224.13	239.64	576 466 196		**576 466 196**
Random-100 ($p = 0.9$)	25	**70 519 913**			186.74	203.66	**176.65**		408 577 577	**282 803 234**
Random-100 ($p = 0.9$)	34	**0**			186.74	41.76	**16.13**		16 255 377	**4 181 492**
Random-500 ($p = 0.5$)	5	**103 686 974**			**221.25**	247.14	245.86	289 325 735		**289 325 735**
Random-500 ($p = 0.5$)	12	**9 951**			221.25	286.39	**137.05**		121 023 924	**40 087 887**
Random-500 ($p = 0.5$)	15	**0**			221.25	173.11	**96.47**		64 369 156	**16 061 815**
Random-1000 ($p = 0.3$)	5	**15 662 448**			**40.61**	43.59	42.12	40 746 381	40 746 375	**40 723 178**
Random-1000 ($p = 0.3$)	9	**409**			40.61	39.18	**28.53**		21 326 017	**7 034 006**
Random-1000 ($p = 0.3$)	15	**0**			40.61	33.43	**20.10**		9 400 567	**1 785 889**
Snap-BerkStan	10	**457 914**			11.52	11.68	**10.25**	7 278 400	5 236 605	**2 962 145**
Snap-BerkStan	60	**3 717**			11.52	5.08	**4.53**		704 991	**151 160**
Snap-BerkStan	100	**2 208**			11.52	4.13	**4.02**		149 837	**114 213**
Snap-BerkStan	202	**0**			11.52	2.10	**2.07**		601	**1**
DIMACS-Johnson-16-2-4	5	**2 027 025**			**5.92**	6.32	6.52	14 258 174		**14 258 174**
DIMACS-Johnson-16-2-4	8	**2 027 025**			5.92	6.29	**5.88**		14 258 174	**9979328**
DIMACS-Johnson-16-2-4	9	**0**			5.92	6.10	**5.05**		12 194 939	**3 875 598**
DIMACS-Johnson-16-2-4	50	**0**			5.92	0.15	**0.02**		8 942	**642**

Table 4.3: Results for some of the graphs in the dataset used by Eppstein et al. [9], with varying minimum maximal clique size k (small, medium, large/very large for each graph). The time is averaged over five consecutive executions. `Standard` denotes the Eppstein et al. baseline algorithm as explained at the beginning of Chapter 3. `S` stands for our variant with the Size Heuristic, while `S + E` denotes our variant with both Size and Edge Count Heuristic. Marked in bold are the best performances for each row.

4.3.2 Individual Datasets Analysis

Speaking of the actual time spent, in seconds, the results are still good even if they do not directly reflect the big changes that happen in the recursion tree explored by the algorithms. In fact, there are cases such as the `Snap-Berkstan` graph where the number of recursive calls is considerably smaller with the heuristics, but the time spent is not as smaller with respect to the base version or even the size heuristic. To the best of our knowledge, this happens because the recursive calls cut by our pruning strategies correspond mainly to the final nodes of the branches in the recursion tree followed by the algorithm, i.e., to those calls that would have been closed anyways after few steps of computation. Still, the saving in time is not negligible and it follows the expected trend: for lower values of k it is not uncommon to see that the performances even worsen a little because our strategy, not being able to cut anything, turns into unnecessary overhead, but for higher k the standard algorithm (and the size heuristic) is completely outclassed.

After this general discussion, we now give some comments to the specific datasets, starting with the one from [9]. In this dataset we find many real-world graph that have

Dataset	k	Cliques Found			Time (s)			Recursive Calls		
		Standard	S	S + E	Standard	S	S + E	Standard	S	S + E
Complementary-chicagoRegional	111	15 094 937 264	14 676 387 568	**16 162 533 712**	†	†	†	30 189 875 810	26 883 210 138	**30725084366**
Complementary-chicagoRegional	515	15 165 583 992	14 651 236 337	**16 213 618 235**	†	†	†	30 331 169 270	26 856 200 332	**30 479 084 162**
Complementary-chicagoRegional	1121		**0**		†	†	†	30 438 492 122	1 133 687 306	**1 445 534 016**
Complementary-hamsterster	111	15 172 537 653	14 692 821 989	**15 975 551 808**	†	†	†	30 345 076 544	30 656 710 884	**26 812 583 628**
Complementary-hamsterster	414	15 176 328 928	14 711 600 320	**16 158 958 167**	†	†	†	30 352 659 090	30 850 961 742	**26 999 043 908**
Complementary-hamsterster	818		0		†	†	†	30 329 478 375	2 506 269 042	**1 851 645 852**
Complementary-mayaan	111	3 485 086 276	3 535 835 040	**4 025 410 672**	†	†	†	9 865 749 327	10 248 057 386	**9 604 069 278**
Complementary-mayaan	313	3 483 052 730	3 526 226 488	**4 005 075 948**	†	†	†	9 859 932 155	10 234 489 793	**9 598 151 073**
Complementary-mayaan	616		**0**		†	†	†	9 859 823 722	179 370 427	**160 908 796**
Loose-26183-505 (clique size 505)	9		**873 242**		1 045.29	951.06	**737.41**	68 161 576	54 116 374	**43 776 391**
Loose-26183-505 (clique size 505)	109		**51**			879.13	**87.32**		6 285 944	**45 192**
Loose-26183-505 (clique size 505)	505		**51**			636.35	**37.55**		429 877	**25755**
Loose-26183-505 (clique size 505)	609		**0**			429.75	**23.90**		2 756 134	**0**
Loose-1234-503 (clique size 503)	9	914 132 676	**987 029 970**	986 147 542	†	†	†	3 364 597 622	3 614 362 422	**3 621 247 946**
Loose-1234-503 (clique size 503)	109	40	3069	**19 089**	†	†	†	3 334 908 278	36 748 721	**26 896 324**
Loose-1234-503 (clique size 503)	503	0	**2**	**2**	†	41.44	**2.52**	3 345 929 121	46 971	**2 446**
Loose-1234-503 (clique size 503)	509		**0**		†	14.89	**0.43**	3 334 495 382	14 364	**0**
Loose-5783-407 (clique size 407)	9		**551 238**		334.53	295.93	**248.61**	74 168 195	42 963 062	**30 372 009**
Loose-5783-407 (clique size 407)	109		**14**			141.12	**12.90**		1 009 489	**12 918**
Loose-5783-407 (clique size 407)	407		**14**			90.42	**5.41**		622 367	**5 698**
Loose-5783-407 (clique size 407)	409		**0**			46.95	**2.32**		613062	**0**
Regular-582 (582-regular graph)	2		**22 340 874**		130.90	122.90	**112.00**		**36 897 737**	
Regular-582 (582-regular graph)	7		**0**			121.94	**98.16**	36 897 737	28 936 478	**4 486 320**
Regular-582 (582-regular graph)	42		**0**			111.35	**19.72**		4 699 724	**0**
Scalefree-2847-1245 ($p = 0.8$)	65	6 331 424	6 343 848	**7 706 721**	†	†	†	162 506 651	162 823 804	**194 652 711**
Scalefree-2847-1245 ($p = 0.8$)	505	147 118	908 444	**1 782 305**	†	†	†	163 813 494	50 403 562	**75 735 221**
Scalefree-2847-1245 ($p = 0.8$)	925	0	16 909 236	**17 024 224**	†	†	**2 223.76**	162 909 127	283 087 112	**243 085 158**
Scalefree-2847-1245 ($p = 0.8$)	985	0	**1 328**	**1 328**	†	199.39	**47.32**	163 578 238	120 980	**44 808**
Scalefree-1574-210 ($p = 0.8$)	25		**11 845 446**		360.25	249.53	**202.86**	337 426 268	106 210 744	**89 546 136**
Scalefree-1574-210 ($p = 0.8$)	85		**1 163 359**			59.29	**37.21**		11 261 004	**9 949 324**
Scalefree-1574-210 ($p = 0.8$)	165		**0**			5.57	**0.51**		69 192	**299**

Table 4.4: Results for some of the graphs in our synthetic dataset, with varying minimum maximal clique size k (small, medium, large/very large for each graph). The time is averaged over five consecutive executions. `Standard` denotes the Eppstein et al. baseline algorithm as explained at the beginning of Chapter 3. `S` stands for our variant with the Size Heuristic, while `S + E` denotes our variant with both Size and Edge Count Heuristic. Marked in bold are the best performances for each row. † denotes a timeout.

degeneracy $d \ll |V|$, hence they are highly sparse. This is correctly exploited by the basic algorithm, as it has very good performances on most graphs and the time speedup of the heuristics is marginal (e.g. *BioGRID*, *Marknewman*, *Pajek*) but still noticeable. Despite this, the number of recursive calls, as pointed out above, is instead largely shrunk as k increases.

The most interesting set of graphs contained in this dataset is the *Random* one: they have a number of cliques which is proportional, as expected, to the probability used for adding edges to them. This number is particularly high for lower values of k even if the graph is relatively small in terms of number of vertices, but it rapidly goes towards zero. Regarding the time spent we note how the two heuristics give a boost to performances, especially when there are no cliques to be found. The reason why so much time is spent on these Erdös-Renyi graphs despite their little size compared to the others, lies behind the structure of this model: since edges are added following a uniform probability distribution, the resulting graphs will have vertices with approximately the same number of neighbors. If the probability of adding edges is sufficiently high, like in the case of `Random-100-0.9`, the number of maximal cliques will be close to the maximum. Since every vertex has, on average, the same degree t, there will be a point in which $t+1$ cliques are not possible. This will cause the heuristics to fail more often than on the other kind of graphs, because little cutting can be done when every vertex has the same, high,

number of neighbors. On the other hand, however, when k reaches the $t + 1$ threshold, almost every vertex will be cut away, leading to a massive improvement with respect to the standard version.

Moving to our synthetic dataset we can see more in detail how the structure of graphs influences the performance of this clique enumeration technique.

The dataset *Loose* represent, to some extent, the other extreme with respect to the Random one, i.e., these graphs are designed specifically to highlight the power of the heuristics since every vertex there has the most neighbors in the clique to which it is assigned and significantly less neighbors within the rest of the graph. The Edge Count heuristic captures exactly this fact because many vertices of these graphs will not have enough neighbors in the P set (especially ones in the "border" of a clique) and thus their corresponding recursive calls will terminate earlier with respect to the other versions. Notice also that the size heuristic alone is not sufficient for this graphs, because again vertices on the border of a clique will satisfy the inequality $|P| + |R| \geq k$ for a long time during the execution due to the high number of neighbors each node has in its clique.

Regular graphs are an interesting case because they contain very many cliques of small size despite having high degeneracy (which is in turn equal to the degree of the vertices in a regular graph). In particular, the following statement on $\omega(G)$ is true for every d-regular graph.

Theorem 4.1. *In a d-regular graph $G = (V, E)$ with $|V| = n$, the maximum possible size of any clique is either $\lfloor n/2 \rfloor$ or n.*

Proof. Suppose that $G = (V, E)$ has a clique A of a vertices, and let B denote the other $V \setminus A$ vertices, with $b = |B| = n - a$. Without loss of generality, suppose that $a > b$.

Since A is a clique, every vertex of it is connect to other $a - 1$ vertices in G, and to $d - (a - 1)$ vertices of B, thus the number of edges connecting vertices from A to B is $e = a(d - (a - 1))$.

With a similar reasoning, every vertex of B is connected to other $b - 1$ neighbors in the same set, thus $e \geq b(d(b - 1))$ and by substitution we get:

$$b(d(b-1)) \leq a(d(a-1))$$

By subtracting the left-hand side from the right-hand side we then have:

$$\begin{aligned}0 \leq (ad - a^2 + a) - (bd - b^2 + b) &= (ad - bd) - (a^2 - b^2) + (a - b) = \\ &= (a - b)(d - a - b + 1)\end{aligned}$$

Since $a > b$, it follows that $d \geq a + b - 1 = n - 1$, so G is the complete graph on n vertices. From this reasoning follows the thesis, since if any d-regular graph has a clique of more than $\lfloor n/2 \rfloor$ vertices, then it is the complete graph. ■

We can see from the results we obtained that this bound is often loose, as every regular graph we generated had cliques of only up to ten vertices. While this is a bad configuration for our heuristics, we observed that there is a significant gap between our

two heuristic versions and this may be due to the degeneracy order calculated at the beginning of the procedure, as this ordering affects the composition of the P set for each node which is in turn used by the Edge Count heuristic, thus different orderings may have provided different results in the discrepancy between the timing of two heuristics.

Regarding the *Scalefree* configuration, the closest among our models to real-world graphs, we observe that although the graphs we tested are again relatively small in terms of vertices (less than 3000), the degeneracy is high and the algorithms have often reached the one-hour time limit. This is to be expected since these graphs were designed to have a particularly high average clustering coefficient, and thus a very large amount of cliques inside. We can see how our Edge Count heuristic was able to provide better results, sometimes reaching a number of cliques discovered of one order of magnitude more with respect to the version with only the Size heuristic.

Finally, the *Complementary* dataset pushed the algorithms to their limits, making them always reach the timeout threshold. What we can learn from this dataset is that even under conditions that were not taken into consideration when designing the algorithm and the heuristics (e.g. degeneracy close the number of vertices, extremely dense graphs) the performances are quite good, and again the heuristics do not introduce significant overhead and are able to reduce the number of recursive calls made while getting a higher number of cliques found in the same amount of time.

The take-home message that these experiments give to us is that even if a graph is considered sparse in terms of degeneracy and we have an algorithm that has this measure as a parameter of its computational complexity, the internal structure of the input still plays an important role in the resolution of this enumeration problem, and should be further investigated.

4.3.3 Results Visualization

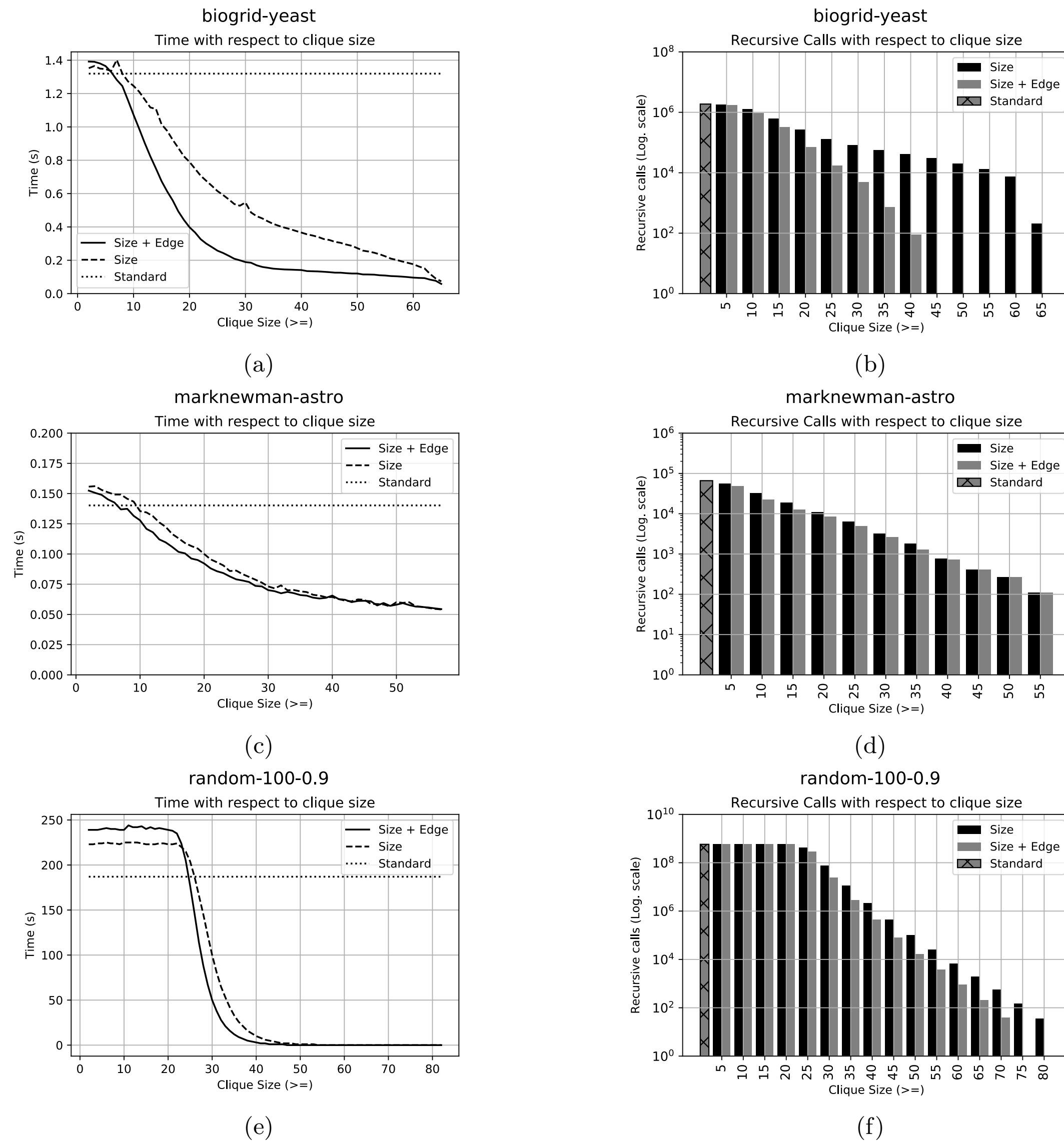


Figure 6: Results for some of the graphs contained in the Eppstein's dataset presented in Table 4.1. Left-hand plots show the time in seconds took by the algorithms, while plots on the right show the corresponding number of recursive calls.

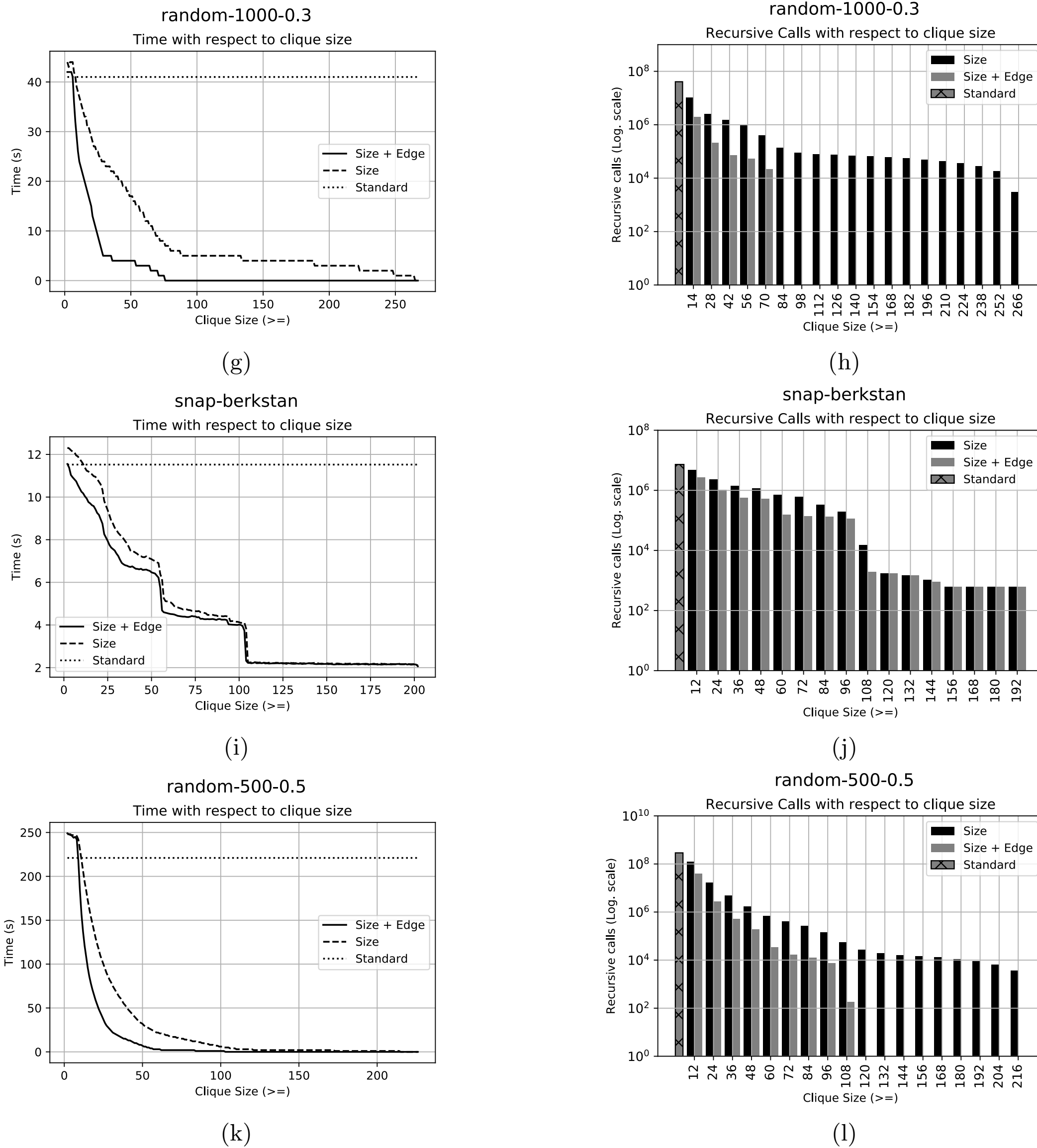


Figure 6: Results for some of the graphs contained in the Eppstein's dataset presented in Table 4.1. Left-hand plots show the time in seconds took by the algorithms, while plots on the right show the corresponding number of recursive calls.

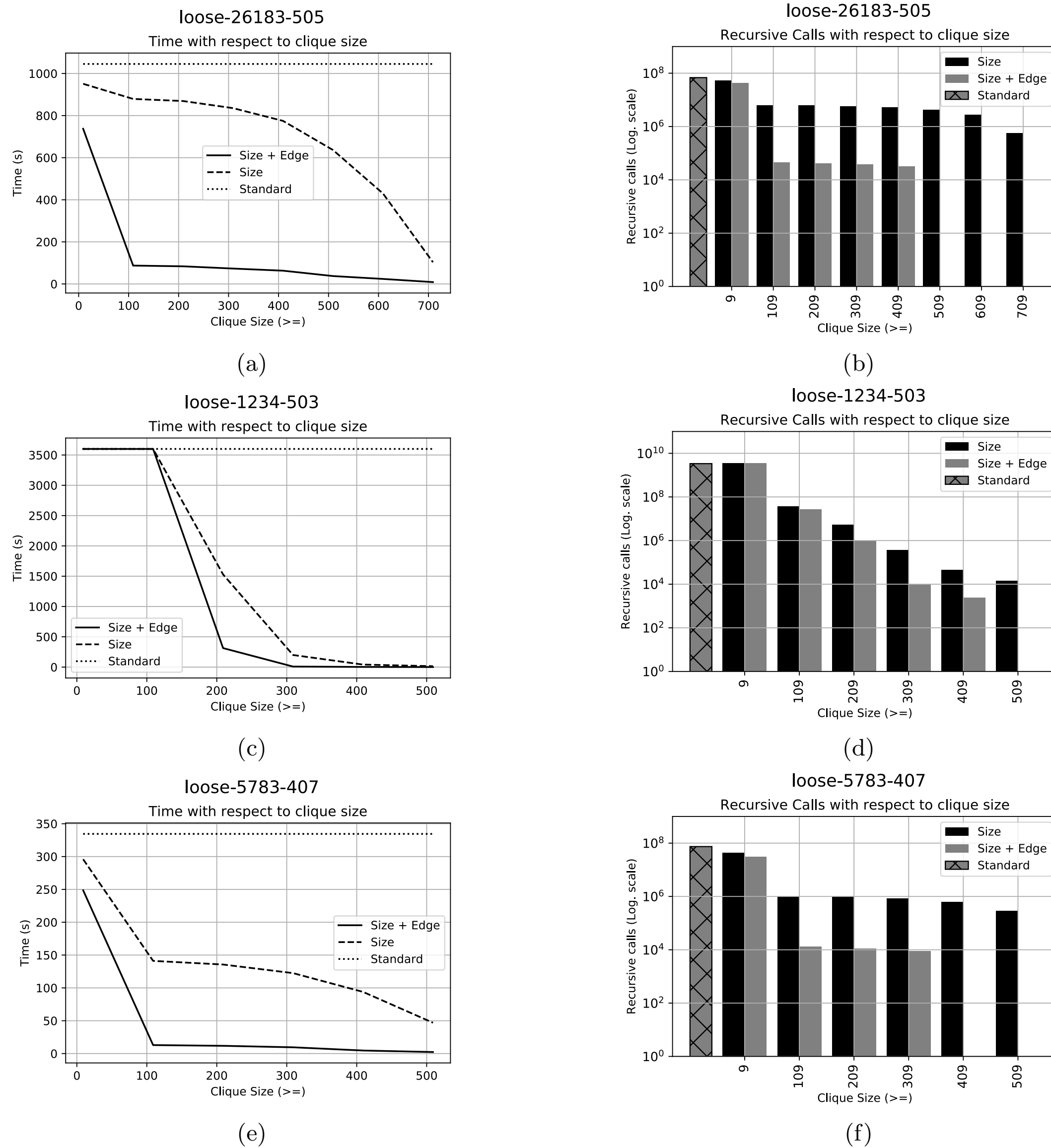
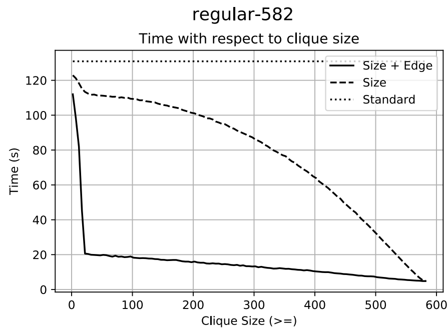
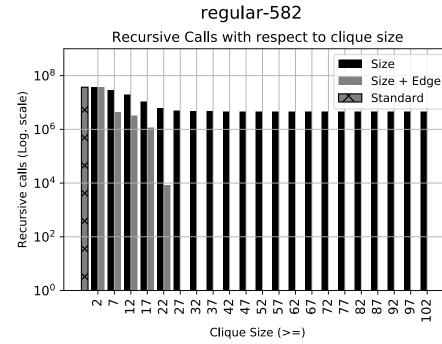


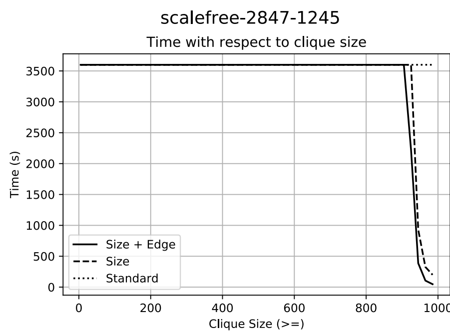
Figure 7: Results for some of the graphs contained in our synthetic dataset presented in Table 4.1. Left-hand plots show the time in seconds took by the algorithms, while plots on the right show the corresponding number of recursive calls.



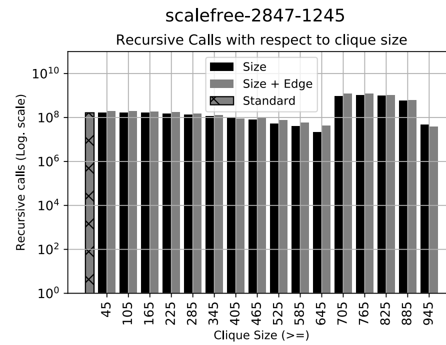
(g)



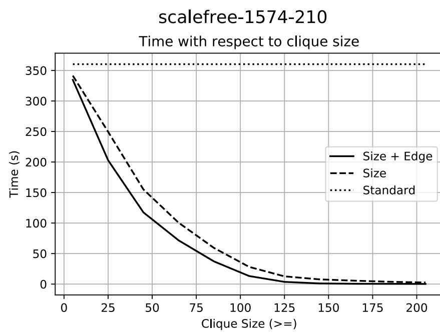
(h)



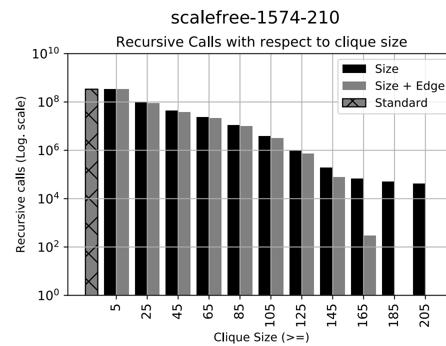
(i)



(j) Since timeouts occurred, in this plot the higher the better.



(k)



(l)

Figure 7: Results for some of the graphs contained in our synthetic dataset presented in Table 4.1. Left-hand plots show the time in seconds took by the algorithms, while plots on the right show the corresponding number of recursive calls.

Chapter 5

Conclusions and Future Work

We have presented and discussed the Bron-Kerbosch algorithm and two of its recent variations, namely the Tomita et al.'s and the Eppstein et al.'s, that have increasingly improved its performances using various strategies to cut the number of recursive calls needed by the algorithm to terminate. We then have developed additional strategies in order to further reduce the size of the search space by exploiting structural properties of the graphs combined with our aim of enumerating only the maximal cliques of size greater than a given threshold. An extensive experimental study was conducted in order to assess the quality of our heuristics and to see how different models of graphs influence the performances of all the algorithms.

The results that we obtained give a reason to continue working on this problem, as new and possibly different ideas with respect to the ones used by Eppstein et al. may acheive even better time bounds on real world graphs. There is also room for improvement in the field of dense graphs: while this kind of network is not as common as sparse ones, it may be interesting to explore in depth how these algorithms react and how they can be improved by exploiting more the internal structure of these graphs, since we have seen in the experiments that it represents a key factor for the performance of our methods.

From a more theoretical point of view we discussed also the output sensitivity of the Bron-Kerbosch algorithm and its variations, showing that at the time of writing only partial results on their (exponential) delay have been recently proven. However, if the algorithm by Eppstein et al. was proven to be output sensitive at least for some class of graphs that would give an explanation to its astounding performances on some datasets, since its running time would actually depend on the number of maximal cliques to be enumerated which is usually far from the exponential bound given by Moon and Moser [21, 8].

For the problem we attacked in this thesis, the enumeration of $k_{\geq}$-cliques, we believe that two approaches are possible. A natural way to further develop this work is to refine the strategies designed in Chapter 3, particularly the ones that were not adopted due to their poor cutting ability or too high cost to be applied with respect to the ones we chose to implement. In particular we need to explore more in detail the bounds that can be given to the clique number and to the chromatic number: nowadays the most tight

bounds that we have on both numbers come from the spectral analysis of the adjacency matrix of the graphs. As pointed out briefly in Chapter 3, just storing the adjacency matrix would require a huge amount of memory and a consequent large amount of time would be necessary to extract the information needed, so this kind of approach cannot be adopted in our context. Instead we shall develop new, tighter, combinatoric bounds by exploiting the structural properties of the graphs and of course we should also aim to prove new results regarding the existence of complete subgraphs within graphs with a given set of properties.

Another other way to continue working in this field is to design a completely new and different strategy to attack this enumeration problem. One instance of this approach could consist in using the results by Erdös and Turán presented in Section 3.3.4 to create an algorithm that is driven through the search space by those sufficient conditions and thus explore first the promising paths, then applying some kind of stopping condition to avoid exploring the rest of the useless paths. Theorem 3.5 suggests a way of proceeding in this sense that is to first explore the subgraphs having a density higher than the given threshold.

One could also choose to give up the exact enumeration and accept an approximation of the number of $k_{\geq}$-cliques, in exchange of a more favorable running time. As a final remark we note that all the problems and the work presented throughout this dissertation play a role in the theoretical field of complexity, so any step towards more efficient algorithms for NP-hard problems is a step towards the better understanding of the limits that complexity classes pose on the problems we can solve reasonably fast.

Bibliography

[1] A. T. Amin and S. L. Hakimi. Upper bounds on the order of a clique of a graph. *SIAM Journal on Applied Mathematics*, 22(4):569–573, 1972.

[2] B. Andrásfai, P. Erdös, and V. Sós. On the connection between chromatic number, maximal clique and minimal degree of a graph. *Discrete Mathematics*, 8(3):205 – 218, 1974.

[3] V. Batagelj and M. Zaveršnik. An o(m) algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 10 2003.

[4] J. Bondy. Bounds for the chromatic number of a graph. *Journal of Combinatorial Theory*, 7(1):96 – 98, 1969.

[5] C. Bron and J. Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, Sept. 1973.

[6] M. Chrobak and D. Eppstein. Planar orientations with low out-degree and compaction of adjacency matrices. *Theoretical Computer Science*, 86(2):243 – 266, 1991.

[7] A. Conte, T. De Matteis, D. De Sensi, R. Grossi, A. Marino, and L. Versari. D2k: Scalable community detection in massive networks via small-diameter k-plexes. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1272–1281, 2018.

[8] A. Conte, R. Grossi, A. Marino, and L. Versari. Sublinear-Space and Bounded-Delay Algorithms for Maximal Clique Enumeration in Graphs. *Algorithmica*, 82(6):1547–1573, June 2020.

[9] D. Eppstein, M. Löffler, and D. Strash. Listing all maximal cliques in large sparse real-world graphs. *ACM J. Exp. Algorithmics*, 18, Nov. 2013.

[10] P. Erdős and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[11] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75 – 174, 2010.

[12] S. Harenberg, G. Bello, L. Gjeltema, S. Ranshous, J. Harlalka, R. Seay, K. Padmanabhan, and N. Samatova. Community detection in large-scale networks: a survey and empirical evaluation. *WIREs Computational Statistics*, 6(6):426–439, 2014.

[13] P. Holme and B. J. Kim. Growing scale-free networks with tunable clustering. *Phys. Rev. E*, 65:026107, Jan 2002.

[14] P. Hui, E. Yoneki, S. Y. Chan, and J. Crowcroft. Distributed community detection in delay tolerant networks. In *Proceedings of 2nd ACM/IEEE International Workshop on Mobility in the Evolving Internet Architecture*, MobiArch '07, New York, NY, USA, 2007. Association for Computing Machinery.

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

[16] J. H. Kim and V. H. Vu. Generating random regular graphs. In *Proceedings of the thirty-fifth ACM symposium on Theory of computing - STOC '03*, page 213, San Diego, CA, USA, 2003. ACM Press.

[17] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*, pages 303–336. Springer, 2010.

[18] D. R. Lick and A. T. White. k-degenerate graphs. *Canadian Journal of Mathematics*, 22(5):1082–1096, 1970.

[19] K. Makino and T. Uno. New algorithms for enumerating all maximal cliques. In *Scandinavian workshop on algorithm theory*, pages 260–272. Springer, 2004.

[20] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, July 1983.

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

[22] M. Soto, A. Rossi, and Marc Sevaux. Three new upper bounds on the chromatic number. *Discrete Applied Mathematics*, 159(18):2281–2289, Dec. 2011.

[23] E. Tomita and A. Conte. Another time-complexity analysis for maximal clique enumeration algorithm CLIQUES. *Technical Report of IEICE COMP*, 1(1):1–8, 2020.

[24] 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.

[25] 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.

[26] W. W. Zachary. An information flow model for conflict and fission in small groups. *Journal of anthropological research*, 33(4):452–473, 1977.

[27] A. Zhang. *Protein Interaction Networks: Computational Analysis.* Cambridge University Press, 2009.