Algorithms for Finding Highly Connected Subgraphs

Bachelorarbeit

vorgelegt
der Fakultät IV – Elektrotechnik und Informatik,
Fachgebiet Algorithmik und Komplexitätstheorie,
der Technischen Universität Berlin

von Christopher Hannusch

Berlin, den 02.05.2017

Erstgutachter: Prof. Dr. Rolf Niedermeier
Zweitgutachter: Prof. Dr. Stephan Kreutzer
Betreuer: Manuel Sorge und Hendrik Molter

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Die selbstständige und eigenhändige Anfertigung versichert an Eides statt:

Ort, Datum Unterschrift

Zusammenfassung

HIGHLY CONNECTED SUBGRAPH ist ein Problem aus der Familie der dichten Teilgraphprobleme und ist NP-schwer. Gegeben einen Graph und eine natürliche Zahl k, soll ein Teilgraph mit exakt k Knoten gefunden werden, in dem jeder Knoten mindestens $\lfloor \frac{k}{2} \rfloor + 1$ Nachbarn hat. Um dies zu erreichen, entwickeln und optimieren wir Algorithmen und testen ihre Performance in der Praxis. Von den drei getesteten Algorithmen löst der Algorithmus für den Parameter degeneracy nicht nur die meisten Instanzen schneller als die anderen beiden Algorithmen für die Parameter h-Index und Kantenisolierung, sondern auch schneller als die NP-Schwere dieses Problems nahelegen würde.

Zusätzlich dazu erforschen wir einige Eigenschaften von HIGHLY CONNECTED SUBGRAPH, hauptsächlich wie viele aufeinanderfolgende (bezüglich der Anzahl Knoten in der Lösung k) „Nein“-Instanzen zwischen zwei „Ja“-Instanzen von HIGHLY CONNECTED SUBGRAPH liegen können. Wir haben herausgefunden, dass dies mindestens fünf sein können. Wir haben außerdem obere Schranken für k basierend auf der degeneracy beziehungsweise dem h-Index eines Graphen gefunden oder optimiert.

Abstract

HIGHLY CONNECTED SUBGRAPH is a problem from the family of dense subgraph problems and is NP-hard. Given a graph and a natural number k, we want to find a subgraph with exactly k vertices such that all vertices in this subgraph have at least $\lfloor \frac{k}{2} \rfloor + 1$ neighbors. In order to do this, we design and optimize some algorithms and test their performance in practice. Of the three tested algorithms, the algorithm for parameter degeneracy solves most instances not only faster than the other algorithms for parameter h-index and edge isolation, but also significantly faster than the NP-hardness of this problem would suggest.

Additionally, we research on some properties of the highly connected subgraph problem, mainly how many consecutive (in terms of solution size k) “no”-instances can be between two “yes”-instances of HIGHLY CONNECTED SUBGRAPH. We found out that this number is at least five. We also found or optimized upper bounds for k based on the degeneracy and the h-index of a graph.

Contents

1 Introduction **5**
1.1 Related work . . . 5
1.2 Our contributions . . . 6

2 Preliminaries **6**
2.1 Graph theory . . . 6
2.1.1 Graph parameters . . . 7
2.1.2 Highly connected graphs . . . 8
2.2 Decision and optimization problems . . . 8
2.3 Search-tree algorithms . . . 9
2.4 Parameterized complexity . . . 10

3 Properties of highly connected subgraphs **11**
3.1 Gaps . . . 11

4 Algorithms **13**
4.1 General concept and properties of algorithms . . . 13
4.2 Algorithm for parameter "degeneracy" . . . 14
4.3 Algorithm for parameter "h-Index" . . . 16
4.4 Algorithm for parameter "edge isolation" . . . 17
4.5 Reduction, pruning and branching rules . . . 18
4.5.1 Reduction rules . . . 18
4.5.2 Pruning rules . . . 18
4.5.3 Branching rules . . . 21

5 Implementation details and experimental results **24**
5.1 Implementation details . . . 24
5.2 Test data and environment . . . 24
5.3 Test results . . . 25
5.4 Conclusion . . . 30

6 Summary and outlook **30**

Literature **31**

1 Introduction

The problem of finding relatively dense clusters[1] or communities in graphs occurs in many natural applications, for example in biology [SUS07] and social networks. As an example of an application in a social network, one could be interested in groups of people where each person knows many of the others in the group, making it possible to show an advertisement to only a few persons per group. Assuming they tell their friends about it, this would probably help to minimize marketing costs and maximize profit.

As a special case of dense clusters, highly connected graphs are graphs where each vertex is a neighbor of more than half of all vertices [HS00]. In this thesis, we present some algorithms to find highly connected subgraphs (HCS), an NP-hard problem [HKS15, Theorem 1], and give results on their performance in practice.

Formally, the problem is defined as follows:

Highly Connected Subgraph
Input: An undirected graph $G = (V, E)$ and a $k \in \mathbb{N}$.
Question: Is there a subgraph $G' = (V', E')$ of G such that $|V'| = k$ and $\forall v \in V' : \deg(v) \geq \lfloor \frac{k}{2} \rfloor + 1$?

Here, $\deg(v)$ denotes the degree of the vertex v, that is the number of neighbors of v.

1.1 Related work

The paper "Finding highly connected subgraphs" [HKS15] and the PhD thesis by Manuel Sorge [Sor17, Section 7] directly cover the problem of finding highly connected subgraphs and contain summaries of related work this subsection is partly based on.

Hartuv and Shamir developed an algorithm that splits a graph into highly connected components [HS00]. In an article of Falk Hüffner et al. [Hü+14], an algorithm is presented that tries to minimize the number of edges that have to be deleted in order to obtain such highly connected components.

Cliques[2] are highly connected subgraphs and therefore HCS can be seen as a relaxation of the clique problem [PYB13]. The problem of finding subgraphs of order k such that each vertex has degree at least $k - s$, also called s-plexes, is similar to both Clique and Highly Connected Subgraph [MNS09]. Furthermore, 0.5-quasi-cliques, where each vertex has degree at least $\frac{k-1}{2}$, are very similar to highly connected subgraphs [LW08].

For the example problem of marketing in social networks mentioned above, where the graph changes over time as people become friends and unfriend each other, there is a paper about enumerating cliques in such graphs [Him+16].

The density of a graph is defined as $\frac{2 \cdot |E|}{|V| \cdot (|V| - 1)}$ [KS15], so instead of requiring a minimum degree like in the highly connected subgraph problem, the average degree of the vertices is more important in some of the dense subgraph problems.

[1] A set of vertices that is connected by many edges.
[2] Subgraphs where each vertex is adjacent to each other vertex.

1.2 Our contributions

We contributed to the research on highly connected subgraphs mainly by optimizing algorithms that solve this problem (Section 4) and testing how they perform in practice (Section 5). The algorithm for parameter degeneracy (Section 4.2) performs best in practice, compared to the algorithm for parameter h-index and edge isolation. Additionally, all of the algorithms perform significantly better than expected considering their complexities.

We also researched whether the existence of a highly connected subgraph implies the existence of highly connected subgraphs of other orders (Section 3.1). If there is a highly connected subgraph with an even number of vertices k, then also a highly connected subgraph of order $k-1$ exists. However, the non-existence of highly connected subgraphs of at least five consecutive orders does not imply the non-existence of larger highly connected subgraphs, which makes the optimization problem of finding the largest highly connected subgraph more complex in terms of practical running time.

Additionally, we found or optimized some upper bounds for the order k of highly connected subgraphs based on some parameters of the graph, namely the degeneracy ($k \leq 2d - 1$, Theorem 3.2) and the h-index ($k \leq 2h - 1$, Theorem 3.3).

2 Preliminaries

In this section we will present the theoretical background including definitions and notations concerning graph theory (Section 2.1), decision and optimization problems (Section 2.2), search-tree algorithms (Section 2.3), and parameterized complexity (Section 2.4).

2.1 Graph theory

In the following we present all necessary graph theoretical definitions used in this thesis. We use a notation similar to the one used in the book “Graph theory” of Reinhard Diestel [Die00].

Definition 2.1 (Graph)**.** Let V be a set and E be a set of subsets of size two of V. Then $G = (V, E)$ is a *graph*, where V is the set of *vertices* and E is the set of *edges*.

Notation. Let $G = (V, E)$ be a graph. For the *order* of G we write $n := |V|$ and for the number of edges we write $m := |E|$.

Definition 2.2 (Neighborhood and degree)**.** Let $G = (V, E)$ be a graph, $v \in V$ and $V' \subseteq V$.

- The (open) neighborhood $N_G(v) := \{u \in V \mid \{u, v\} \in E\}$ is the set of vertices that are adjacent to v.
- $N_G[v] := N_G(v) \cup \{v\}$ is the closed neighborhood of v.

- The (open) neighborhood of a set $N_G(V') := \cup_{v \in V'} N_G(v) \backslash V'$ consists of all neighbors of vertices in the set without the set itself.
- The closed neighborhood of a set is consistently defined as $N_G[V'] := N_G(V') \cup V'$.
- The degree $\deg_G(v)$ is the number of neighbors of v, that is $\deg_G(v) := |N_G(v)|$.

We typically omit the subscript G if it is not ambigious.

Definition 2.3 (Subgraph)**.** Let $G = (V, E)$ be a graph. A graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$ is a *subgraph* of G.

Definition 2.4 (Induced subgraph)**.** Let $G = (V, E)$ be a graph and $G' = (V', E')$ be a subgraph of G. We call $G' = G[V']$ an *induced subgraph* of G if and only if $\forall u, v \in V' : \{u, v\} \in E \iff \{u, v\} \in E'$. This means that G' contains all edges between vertices in V' that are in G.

Notation. Let $G = (V, E)$ be a graph, $v \in V$ and $V' \subseteq V$. As an abbreviation, we write $G - v := G[V \backslash \{v\}]$ and $G - V' := G[V \backslash V']$.

Definition 2.5 (Path)**.** Let $G = (V, E)$ be a graph and $u, v \in V$. A *path* P from $v_0 = u$ to $v_l = v$ is a subgraph of G such that $V(P) := \{v_0, \dots, v_l\}$ and $E(P) := \{\{v_0, v_1\}, \dots, \{v_{l-1}, v_l\}\}$. The set $\mathrm{path}(u, v)$ is the set of all paths from u to v. The *length* of P is $\mathrm{len}(P) := |V(P)| - 1$.

Definition 2.6 (Connected components)**.** Let $G = (V, E)$ be a graph. G is *connected* if $\forall u, v \in V : \mathrm{path}(u, v) \neq \emptyset$. A *connected component* of G is a maximum induced subgraph H of G such that H is connected.

Definition 2.7 (Diameter)**.** Let $G = (V, E)$ be a graph. The *diameter* of G is $\max\{\min\{len(P) \mid P \in \mathrm{path}(u, v)\} \mid u, v \in V\}$ if G is connected, else it is ∞.

Definition 2.8 (Cut and minimum cut)**.** Let $G = (V, E)$ be a graph. A *cut* C is a subset of E such that there exist $A, B \subset V$ with $A \cup B = V$ and $A \cap B = \emptyset$ and $\forall a \in A : \forall b \in B : \forall P \in \mathrm{path}(a, b) : E(P) \cap C \neq \emptyset$. A *minimum cut* of a graph G is the smallest cut $C \subseteq E$.

This basically means that C splits the vertices into two subsets that are only connected by edges in C. Note that the definition of minimum cut also works for graphs that are not connected. In this case, $C = \emptyset$.

2.1.1 Graph parameters

There are some graph parameters that are useful for complexity analysis of some of the graph algorithms presented in this thesis.

Definition 2.9 (Degeneracy)**.** Let $G = (V, E)$ be a graph. The *degeneracy* $d \in \mathbb{N}$ is the smallest natural number such that there is an ordering (an antisymmetric, transitive and total relation) $\preceq \subseteq V \times V$ such that $\forall v \in V : |\{u \mid \{u, v\} \in E \wedge u \preceq v\}| \leq d$. This means that every vertex v has at most d neighbors that are earlier in the ordering [LW70].

One can compute the degeneracy by repeatedly removing one of the vertices with the lowest degree until the graph is empty. Let v_i be the vertex removed in the ith iteration and $G_i := G - \{v_1, \dots, v_{i-1}\}$. The degeneracy then is $\max\{\deg_{G_i}(v_i) \mid 1 \leq i \leq n\}$. This computation can be done in $\mathcal{O}(n^2)$ time: Computation of the degree is doable in $\mathcal{O}(n + m)$ time and therefore also in $\mathcal{O}(n^2)$ time and finding and removing of the vertex with the lowest degree and updating the degree of the other vertices is in $\mathcal{O}(n)$ time and is repeated $\mathcal{O}(n)$ times. We will use this approach later in one of the search-tree algorithms for solving HCS.

The h-index, named after Jorge E. Hirsch, was originally defined as the largest number h of publications of a scientist that were cited at least h times [Hir05]. However, this can be seen as a graph problem and therefore we define the h-index on graphs as follows:

Definition 2.10 (H-index)**.** Let $G = (V, E)$ be a graph. The *h-index* $h \in \mathbb{N}$ is the largest natural number such that $|\{v \in V \mid \deg(v) \geq h\}| \geq h$. This means there are h vertices with degree at least h.

One can compute the h-index as follows: Let $v_1, \dots, v_n$ be ordered such that $\forall i, j \in \{1, \dots, n\} : i < j \to \deg(v_i) \geq \deg(v_j)$. The h-index then is $i - 1$, where i is the smallest number such that $\deg(v_i) < i$. This computation can be done in $\mathcal{O}(n+m)$ time since computing the degree is in $\mathcal{O}(n + m)$ time and sorting can be done using bucket sort in $\mathcal{O}(n)$ time.

2.1.2 Highly connected graphs

In the following we formally define highly connected graphs and subgraphs.

Definition 2.11 (Highly connected)**.** Let $G = (V, E)$ be a graph. G is *highly connected* if $\forall v \in V : \deg(v) \geq \lfloor \frac{|V|}{2} \rfloor + 1$. A highly connected subgraph of order $k \in \mathbb{N}$ of G is an induced subgraph with exactly k vertices that is highly connected.

Notation. We say that W is a *solution* of the highly connected subgraph problem if $G[W]$ is a highly connected subgraph.

2.2 Decision and optimization problems

A *decision problem* formally is the problem to decide whether a word is contained in a language. More abstract, this means answering a question (e.g. “Is there a solution of size k?”) for a given input instance (the “word”), where the word is in the language if and only if the answer is “yes”. For example, one can decide whether there is a highly connected subgraph H of order exactly k of a graph G. Here, (G, k) is the instance of the problem and H is the solution. In some cases, we are only interested in the existence of a solution, not the solution itself, meaning we only require an algorithm to output “yes” or “no”.

An *optimization problem* is the problem of finding the smallest or largest solution to a problem or just the corresponding input instance. Typically, we want to optimize a natural number, e.g. finding the largest k such that there is a highly connected subgraph.

Both types of problems are strongly related to each other. By solving the decision problem or the optimization problem, one can often solve the other problem. For example, if one can find the maximum independent set[3], then one can also answer "yes" to the question whether the graph has an independent set of at most that maximum size and "no" for larger independent sets. And the other way around, one can find the maximum by iterating from 0 to $|V|$ and outputting i as the maximum size of an independent set if there is an independent set of size i but none of size $i+1$.

However, the approach of solving the decision problem by finding the optimum value as described above for Independent Set does not work for all decision problems and HCS is one of these. A graph can have no highly connected subgraph of order k, but one of order $k-1$ and one of order $k+1$. For more details on this, see Section 3.1.

2.3 Search-tree algorithms

A search tree is a tree where the vertices represent *partial solutions* to a given problem, also called solution candidates, and the edges to children of a vertex represent different decisions one can make in order to find a solution. In our case, this is for example adding or not adding a vertex to a solution candidate. By recursively branching into all the children, a search-tree algorithm can find a solution if one exists, or output that there is no solution.

There can be rules to cut some children off from the tree depending on the current state. If branching into a child is guaranteed to not find a solution, the algorithm does not need to branch into that child. The rules to determine this are called *pruning rules*. Rules that define the structure of a search tree by generating the children of a vertex are called *branching rules*. We will see both a simple and a more advanced branching rule in Section 4.5.3. There are also *reduction rules* that can reduce the size of an instance of a problem before or during the actual search-tree algorithm is executed.

Definition 2.12 (Reduction rule)**.** Let S be the set of instances of our problem and $g : S \to \mathbb{N}$ determine the size of an instance. A *reduction rule* is a function $\mathrm{rr} : S \to 2^S$ such that there exists a solution for $s \in S$ if and only if there exists a solution for an $s' \in \mathrm{rr}(s)$ and $\sum_{s' \in \mathrm{rr}(s)} g(s') \leq g(s)$.

A reduction rule reduces the size of an instance or splits it into several smaller instances. For example, we can split a graph into connected components for our HCS problem. The size of an instance of HCS could be for example $n+m$.

Definition 2.13 (Pruning rule)**.** Let S be the set of partial solutions of our problem and $g : S \to \mathbb{N}$ the number of search tree vertices that are descendents of the vertex

[3] A set of vertices that are pairwise not connected.

corresponding to $s \in S$. A *pruning rule* is a function $\mathrm{pr} : S \to S$ such that there exists a solution for $s \in S$ if and only if there exists a solution for $\mathrm{pr}(s)$ and such that $g(\mathrm{pr}(s)) \leq g(s)$.

Definition 2.14 (Branching rule)**.** Let S be the set of partial solutions of our problem. A *branching rule* is a function $\mathrm{branch} : S \to 2^S$ such that there exists a solution for $s \in S$ if and only if there exists an $s' \in \mathrm{branch}(s)$ such that there is a solution for s'.

Note that our definitions of these different types of rules also include the definition of their correctness. However, they do not include the complexity. By these definitions, an algorithm that solves a problem is also a reduction rule, pruning rule, and branching rule. So, in order to make sense, the rules should have a lower time complexity than the algorithm itself and should not solve the problem in general. For the HCS problem, we only allow reduction, pruning and branching rules with polynomial time complexity. We explicitly do not include this in the definitions since there may be problems where rules with higher complexity may be beneficial.

Some rules are cheap in terms of complexity and running time in practice, some are expensive. Therefore, there often is a trade-off between having a larger search tree and doing more work per vertex to reduce the size of the search tree. For example, it may be beneficial to apply some pruning rules only in every second level of the search tree.

2.4 Parameterized complexity

In contrast to analyzing the complexity of an algorithm based only on the size of the input, in parameterized complexity we consider other parameters or properties of the input, for example the degeneracy of a graph.

There is a class FPT of problems that are fixed-parameter tractable, meaning their complexity is upper-bounded by $f(p) \cdot \mathrm{poly}(|X|)$, where p is a parameter of the instance, f is a computable function, and $|X|$ is the size of instance X. For example, Vertex Cover[4] is in FPT for parameter solution size k. One could recursively pick an arbitrary edge and branch on the two incident vertices, which implies a time complexity of $\mathcal{O}(2^k \cdot \mathrm{poly}(n))$.

The class XP contains all problems that are solvable in polynomial time in the size of the input for a fixed parameter p. Clearly, FPT $\subseteq$ XP. It is currently unknown whether FPT $\subset$ XP or FPT $=$ XP, but it is widely believed that XP contains problems that are not in FPT.

Since the running time is polynomial if the parameter is fixed, problems in FPT can often be solved efficiently for small parameters, but problems in XP $\setminus$ FPT can still have high complexities even for small fixed parameters. For example, a problem that can be solved in $\mathcal{O}(2^p \cdot n^2)$ time is efficiently solvable for $p = 4$, but a problem in $\mathcal{O}(n^p)$ time may be impractical to solve for large n even if p is as small as 4.

For more details on parameterized complexity and algorithms, refer to the literature [Cyg+15].

[4] Is there a set of vertices V' of size k such that every edge is incident to at least on vertex in V'?

3 Properties of highly connected subgraphs

Highly connected subgraphs have some interesting properties. Some of them are important for finding efficient algorithms and for proving their correctness, like the following one, which is essential for the algorithm for the parameter "degeneracy" in Section 4.2.

Theorem 3.1 (Diameter, [HS00, p. 177]). *The diameter of a highly connected graph is at most two.*

Proof. Any two vertices u and v have at least $\lfloor \frac{n}{2} \rfloor + 1$ neighbors in the graph. Since there are only $n - 2$ other vertices in the graph, it follows that $N(v) \cap N(u) \neq \emptyset$. Therefore, there exists a vertex w that is neighbor of both vertices and the path $(\{u, w, v\}, \{\{u, w\}, \{w, v\}\})$ has length two. Since this is true for any two vertices, it follows that a highly connected graph has a diameter of at most two. $\square$

The next two properties give us an upper bound for the order of highly connected subgraphs in a given graph.

Theorem 3.2 (Degeneracy and subgraph order, [HKS15]). *The order of a highly connected subgraph is at most $2d - 1$, where d is the degeneracy of the graph.*

Proof. From the definition of the degeneracy it follows that in any subgraph there is a vertex that has degree at most d. Therefore, $d \geq \lfloor \frac{k}{2} \rfloor + 1$, where k is the order of the highly connected subgraph. Since $d = \lfloor \frac{2d-1}{2} \rfloor + 1$, $k = 2d - 1$ is the maximum possible order of a highly connected subgraph. $\square$

Theorem 3.3 (h-index and subgraph order). *The order of a highly connected subgraph is at most $2h - 1$, where h is the h-index of the graph.*

Proof. For a subgraph to be highly connected there have to be k vertices of degree at least $\lfloor \frac{k}{2} \rfloor + 1$. Setting $k := 2h - 1$ gives us a minimum degree of h. According to the definition of h-index, a graph with an h-index of h can have $2h - 1$ or more vertices of degree h. This is also the maximum since larger k would require $k > h$ vertices of degree larger than h and therefore the h-index of the graph would be larger than h, which is a contradiction. $\square$

3.1 Gaps

Another property of highly connected subgraphs is the fact that - for example - there can be highly connected subgraphs of orders 3 and 5, but none of order 4 in a given graph. The following property is helpful if we know that a highly connected subgraph of an even order exists.

Theorem 3.4 (Relation between even and odd orders). *If a graph G has a highly connected subgraph of order $k = 2n$ for $n \in \mathbb{N}$, then G also has a highly connected subgraph of order $k - 1$.*

Proof. Let H be a highly connected subgraph with an even number of vertices. Then it holds that $\lfloor \frac{|H|}{2} \rfloor + 1 = \lfloor \frac{|H|-1}{2} \rfloor + 2$. Since every vertex in H has degree $\lfloor \frac{|H|}{2} \rfloor + 1$, removing an arbitrary vertex v from H reduces the degree of all other vertices by at most one. Therefore, $H - v$ is highly connected. $\square$

Note that this also implies that if there is no highly connected subgraph of an odd order k, then there is no highly connected subgraph of order $k + 1$.

Now we formally define gaps between orders for which highly connected subgraphs exist.

Definition 3.5 (Gap)**.** Let G be a graph and $k_1, k_2 \in \mathbb{N}$ with $k_2 \geq k_1 + 2$. We say that there is a *gap* of size $k_2 - k_1 - 1$ if there exist highly connected subgraphs of G of order k_1 and k_2, but for all $k', k_1 < k' < k_2$, there is no highly connected subgraph of G of order k'.

Note that it follows from Theorem 3.4 that k_2 is always an odd number, since if it was an even number, then there would also be a highly connected subgraph of order $k_2 - 1$.

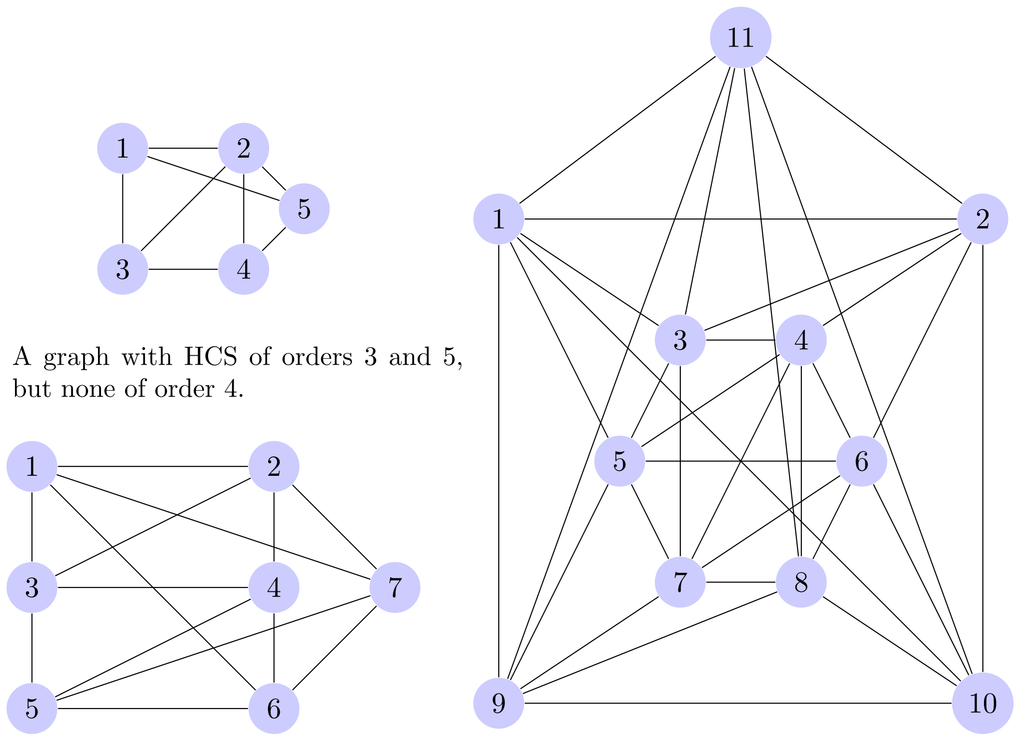
Gaps of size one are easy to find in theory and do exist in some graphs in practice. It is currently unknown how large gaps can be or if there even is a limit. If there is a limit ℓ, one can use this for determining if there is a highly connected subgraph of a given order k: If there are no highly connected subgraphs of order $k - \ell - 1 \leq k' < k$, then there is no highly connected subgraph of order k.

Research on this topic resulted in finding graphs with gaps of size five. Therefore, this approach may not be very helpful in practice, even if there is a limit, since directly finding a highly connected subgraph of order k may be faster than searching for subgraphs of at least six orders smaller than k. However, when looking for highly connected subgraphs of all possible orders, one would be able to stop searching after no highly connected subgraphs were found for more than the gap size limit consecutive orders.

The graph (a) displayed in Figure 1 is a simple example of a graph with a gap of size one. The more complicated graph (b) has a gap of size three since it has highly connected subgraphs of orders 3 and 7, but none of orders 4, 5, and 6. The graph (c) on the right is an example of a graph with a gap of size five.

The test whether highly connected subgraphs of orders 3 to $|V|$ exist was done using one of the algorithms for the highly connected subgraph problem. For graph (a), this can however be easily seen since the set $\{1, 2, 3\}$ induces a highly connected subgraph of order 3 and the whole graph is highly connected, but removing one arbitrary vertex always results in a graph where at least one vertex has less than $\lfloor \frac{4}{2} \rfloor + 1$ neighbors.

The graphs (a) and (b) were found by adding edges and vertices to graphs that were almost highly connected. Graph (c) was obtained by starting with the vertices 1 to 10 arranged in two columns, then for vertically and horizontally "adjacent" vertices an edge was added. Then diagonal edges were added similar to graph (b) and the rest was just



(a) A graph with HCS of orders 3 and 5, but none of order 4.

(b) A graph with HCS of orders 3 and 7, but none of orders 4, 5, and 6.

(c) A graph with HCS of orders 3, 4, 5, and 11, but none of orders 6 to 10.

Figure 1: Examples for graphs with gaps.

adding edges until every vertex has degree six by cleverly guessing and avoiding patterns that would add highly connected subgraphs.

4 Algorithms

In this section we will present some algorithms for finding highly connected subgraphs. We will also show reduction, pruning and branching rules to potentially optimize the running time of the algorithms in practice. For implementation details and actual performance in practice, see Section 5.

The algorithms for the parameters degeneracy and edge isolation have already been presented in the cited documents, while the algorithm for parameter h-index and the reduction, pruning, and branching rules were newly developed by us.

4.1 General concept and properties of algorithms

The algorithms presented in this section are search-tree algorithms that take a graph and a natural number k as input and in each node of the search tree, they have a set of vertices that they have chosen to be in a possible solution and a set of possible vertices to choose next in order to obtain a solution.

Notation. Let $G = (V, E)$ be a graph and k be the order of the highly-connected subgraph. Then $W \subseteq V$ with $|W| \leq k$ is the set of vertices the algorithm has currently chosen and $X \subseteq (V \backslash W)$ is the set of candidate vertices which might be put into W.

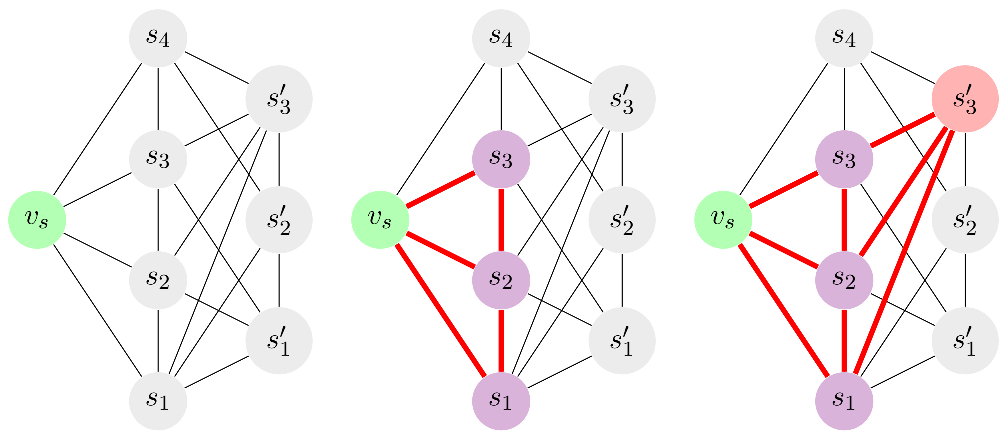
An algorithm may add vertices to W or X which are in none of these two sets, for example the neighbors of vertices it has added to W. A property that implies that this does not happen is needed for some of the pruning and branching rules to be applicable.

Definition 4.1 (Non-growing branch)**.** An algorithm is in a *non-growing branch* $(W, X, \ldots)$ if for all nodes $(W', X', \ldots)$ in the subtree of the search tree with root $(W, X, \ldots)$ it holds that $W' \subseteq (W \cup X)$ and $X' \subseteq X$.

Whether a branch is non-growing can be determined by analyzing the algorithm and may depend not only on the location in the code, but also on some parameters. We will see an example of this in the algorithm for parameter degeneracy in Section 4.2.

4.2 Algorithm for parameter "degeneracy"

Algorithm 1 and its complexity is based on the fact that a vertex v with the lowest degree in a graph G has at most d neighbors, where d is the degeneracy of G, and when it is removed, another vertex v' with minimum degree in $G - v$ has at most d neighbors and so on.



(a) Step 1: Select vertex with minimum degree (e.g. v_s). **(b)** Step 2: Select neighbors S of v_s. **(c)** Step 3: Select vertices from $N(S) \backslash N[v_s]$ (only one in this case).

Figure 2: Visualization of algorithm for parameter "degeneracy" for $k = 5$. The vertices s_1 to s_4 are direct neighbors of v_s and s'_1 to s'_3 are neighbors of neighbors of v_s.

Algorithm 1: Algorithm for parameter “degeneracy”. [Sor17, p. 141]

```
Input: A graph $G = (V, E)$ with degeneracy $d$ and a $k \in \mathbb{N}$.
Output: True, if there is a highly connected subgraph of $G$ of order $k$, false
        otherwise.
begin
    while $G$ is not empty do
        $v_s :=$ a vertex of minimum degree in $G$
        foreach $S \in \text{subsets}(v_s, N(v_s), v_s, i)$, $\lfloor \frac{k}{2} \rfloor + 1 \leq i \leq min(|N(v_s)|, k-1)$ do
            if branch_recursive($S \cup v_s, N(S) \backslash N[v_s]$) $= true$ then
                return $true$
        $G := G - v_s$
    return $false$

Function subsets($W, X, v_s, i$)
    if $i = 0$ then
        return $\{W \backslash v_s\}$
    /* Apply pruning rules here.  Non-growing branch if and only if
       $|W| + i = k$.                                                    */
    $B :=$ branch($W, X$)
    return $\{\text{subsets}(W', X', v_s, i - (|W'| - |W|)) \mid (W', X') \in B\}$

Function branch_recursive($W, X$)
    if $|W| = k$ then
        return $true$ if $G[W]$ is highly connected, $false$ otherwise
    /* Apply pruning rules here.  This is always a non-growing
       branch.                                                           */
    $B :=$ branch($W, X$)
    foreach $(W', X') \in B$ do
        if branch_recursive($W', X'$) $= true$ then
            return $true$
    return $false$
```

The algorithm picks a vertex v_S with minimum degree in G and for all subsets S of the neighborhood of v_S, the algorithm tests for all subsets S' of the neighborhood of S excluding the closed neighborhood of v_S whether $G[\{v_S\} \cup S \cup S']$ is highly connected. The sets S and S' are determined using branching and pruning rules. This algorithm is visualized in Figure 2 for parameter $k = 5$.

The function subsets returns all subsets of $N(v_s)$ of size i if called with parameters $(v_s, N(v_s), v_s, i)$ and if $i \leq |N(v_s)|$, given that the branching rule is correct and only adds at most i vertices to W.

The time complexity of this algorithm is $\mathcal{O}(2^d \cdot n^{d+\mathcal{O}(1)})$ [Sor17, p. 140] and it is therefore in XP. Since $|N(v_S)| \leq d$, there are $\mathcal{O}(2^d)$ subsets of $N(v_S)$. There are n vertices in the graph, all of them are picked once to be v_S. For each of these $\mathcal{O}(2^d \cdot n)$ subsets S, there are less than n vertices in $N(S) \backslash N[v_S]$. Combined with the fact that $k \leq 2 \cdot d - 1$ (Theorem 3.2), this gives us $\mathcal{O}(n^d)$ subsets of the neighborhood. Finally, we need $\mathcal{O}(n^{\mathcal{O}(1)})$

time to test whether a subset induces a highly connected subgraph.

To show the correctness of this algorithm, it suffices to show that it tests all possible subgraphs of order k that can be highly connected. From Theorem 3.1 follows that we only have to consider neighbors and neighbors of neighbors of a vertex. Since the algorithm does exactly that, it is obviously correct.

4.3 Algorithm for parameter "h-Index"

The following algorithm is based on the assumption that vertices with high degree are more likely to be in a highly connected subgraph. Therefore, it prefers the h vertices with highest degree when branching over the neighbors of already chosen vertices, where h is the h-index of the graph.

Algorithm 2: Algorithm for parameter "h-index".

```
Input: A graph $G = (V, E)$ and a $k \in \mathbb{N}$.
Output: True, if there is a highly connected subgraph of $G$ of order $k$, false
        otherwise.
begin
    while $G$ is not empty do
        $v_1, \dots, v_n$ := vertices in $G$, ordered from highest to lowest degree
        $h :=$ h-index of $G$
        if $h = 0$ then
            return $false$
        $V_h := \{v_1, \dots, v_h\}$
        if $\text{br}(G[N[N[v_1]]], V_h, \{v_1\}, N(v_1)) = true$ then
            return $true$
        $G := G - v_1$
    return $false$

Function $\text{br}(G, V_h, W, X)$
    if $|W| = k$ then
        return $true$ if $G[W]$ is highly connected, $false$ otherwise
    /* Apply pruning rules here.  Never a non-growing branch.      */
    if $X = \emptyset$ then
        return $false$
    if $X \cap V_h \neq \emptyset$ then
        Pick a vertex $v \in X \cap V_h$.
    else
        Pick a vertex $v \in X$.
    return $\text{br}(G, V_h, W \cup \{v\}, N(W \cup \{v\})) \vee \text{br}(G - v, V_h, W, X \setminus \{v\})$
```

Algorithm 2 iteratively picks and removes the vertex v_1 with highest degree until the h-index h of the graph is zero or the graph is empty. In each iteration, it computes the set V_h of the h vertices with highest degree in G and recursively picks a vertex v out of the neighbors X in $G[N[N[v_1]]]$ of the picked vertices W and tests whether there is a solution with or without this vertex. If the intersection of X and V_h is not empty, v is picked out of this intersection.

This algorithm is obviously correct: If there is no highly connected subgraph of order k, it can only output $false$. If there is one, then it will find it because all subsets of V that can be a solution are tested. Only vertices in $N[N[v_1]]$ have to be considered for the diameter of a highly connected subgraph is at most two (Theorem 3.1).

The time complexity of this algorithm is trivially $\mathcal{O}(n^{2\cdot h-1+\mathcal{O}(1)})$ due to the fact that $k \leq 2 \cdot h - 1$ (Theorem 3.3) and it is therefore in XP.

A potential optimization for this algorithm is to use the branch_recursive function of the degeneracy algorithm for $k - |W| < \lfloor \frac{k}{2} \rfloor + 1$, since in this case every vertex in any possible highly connected subgraph has to be in $N[W]$ (if it was not, then it could not have $\lfloor \frac{k}{2} \rfloor + 1$ neighbors).

4.4 Algorithm for parameter "edge isolation"

There is a single-exponential fixed-parameter algorithm for the parameter edge isolation γ, which is the size of the cut between the set W that induces the highly connected subgraph $G[W]$ and the set $V \backslash W$, that runs in $\mathcal{O}(4^{\gamma} \cdot n^2 + (k \cdot n + \gamma) \cdot n \cdot m)$ time [HKS15, p. 7]. We use an implementation of this algorithm by Falk Hüffner for comparison of the running time in practice. Note that γ is likely to be large in some real-world applications and that we have to set γ to a potentially large value, but smaller than m, since this algorithm is designed to solve the following problem:

Isolated Highly Connected Subgraph
Input: An undirected graph G and $k, \gamma \in \mathbb{N}$.
Question: Is there a highly connected subgraph of G of order k that is γ-isolated?

The algorithm iteratively picks one vertex $v \in V$, sets $W := \{v\}$, recursively branches into the two cases of adding a vertex out of $N(W)$ to W and removing it from the graph, and if there was no solution containing v, it removes v from G. It stops branching when $|W| = k$ or the number of edges in the original input graph from the set W to already removed vertices exceeds γ. This limits the height of the search tree to at most $k + \gamma$ [HKS15, p. 7].

However, most of the optimizations for this more specific problem do not work for the problem Highly Connected Subgraph since we have to set $\gamma := \min(m - \lceil \frac{(\lfloor \frac{k}{2} \rfloor + 1) \cdot k}{2} \rceil, k \cdot (n - k))$.

Note that this algorithm does not use the fact that the diameter of a highly connected subgraph is at most two, meaning it may consider vertices even if the longest of the shortest paths from them to the vertices in W is longer than two.

For more details about the algorithm and its reduction rules, refer to the cited document or Section 7.4.2. of "Be Sparse! Be Dense! Be Robust! Elements of Parameterized Algorithmics" [Sor17].

4.5 Reduction, pruning and branching rules

In this subsection, we will present some rules that may reduce the running time of the algorithms in practice. Not all rules are applicable in general since some of them require non-growing branches (Definition 4.1).

4.5.1 Reduction rules

In this subsection, we present a simple reduction rule based on the fact that the minimum cut of a highly connected subgraph is at least $\lfloor \frac{k}{2} \rfloor + 1$.

Reduction Rule 4.2. If the size of the minimum cut $|C|$ of a graph is smaller than $\lfloor \frac{k}{2} \rfloor + 1$, then split the graph into subgraphs $G[A]$ and $G[B]$ (see Definition 2.8) and repeat this recursively for both subgraphs until no minimum cut of size smaller than $\lfloor \frac{k}{2} \rfloor + 1$ exists or the set has less than k vertices. Then call the algorithm for each of these subgraphs.

Proof of correctness. The size of a minimum cut of a highly connected graph is at least $\lfloor \frac{k}{2} \rfloor + 1$. Therefore, a graph with a smaller minimum cut cannot be highly connected and we can safely split it into two subgraphs.

We now need to show that this reduction also reduces the size of our instance as required by the definition of reduction rules. If we use $n + m$ as a measurement for the size of an instance, then the sum of the sizes of the two subgraphs is clearly smaller than the size of the input instance.

Finally, the time complexity is polynomial since minimum cut can be solved in $\mathcal{O}(n \cdot m + n^2 \cdot log(n))$ time with the Stoer-Wagner algorithm [SW97] for edge-weighted undirected graphs - in our case, we can simply set each edges weight to 1 - and the minimum cut can be applied less than n times for a graph with n vertices since its vertices can be split into at most n distinct subsets.

□

Note that this rule also splits the graph into connected components for the number of edges between them is zero, so putting the connected components in either A or B with at least one connected component per set gives a minimum cut of size zero.

Also note that there is an obvious reduction rule that returns $false$ - or, formally, the empty set - if the order of the graph is strictly smaller than k.

4.5.2 Pruning rules

In this section, we will present some pruning rules for search-tree algorithms for finding highly connected subgraphs.

The following pruning rule removes a vertex v from X if it is connected to less vertices in the partial solution W than necessary to have $\lfloor \frac{k}{2} \rfloor + 1$ neighbors in a subgraph of order k, even if all other added vertices are neighbors of v.

Pruning Rule 4.3. Remove $v \in X$ from G if it is adjacent to strictly less than $\lfloor \frac{k}{2} \rfloor + 2 - k + |W|$ vertices in W.

Proof of correctness. Let v be a vertex that is adjacent to less than $\lfloor \frac{k}{2} \rfloor + 2 - k + |W|$ vertices in W. If v is in a highly connected subgraph of order k, it has at least $\lfloor \frac{k}{2} \rfloor + 1$ neighbors. If there is a solution that is a superset of W, then let $W' \supseteq W$ be any solution of size k. Then v can be adjacent to at most $k - |W| - 1$ vertices in $W' \backslash W$. So v is adjacent to strictly less than $\lfloor \frac{k}{2} \rfloor + 2 - k + |W| + (k - |W| - 1) = \lfloor \frac{k}{2} \rfloor + 1$ vertices in W', but every vertex in $G[W']$ has degree at least $\lfloor \frac{k}{2} \rfloor + 1$, which is a contradiction. Else, if there is no solution that is a superset of W, then removing any vertex from X won't give us any additional solution. Therefore, the algorithm finds a solution after applying this pruning rule if and only if it finds a solution without applying this pruning rule.

The time complexity of this rule is $\mathcal{O}((n+m) \cdot log(n))$ since for all $\mathcal{O}(n)$ vertices $v \in X$, for the neighbors of v ($\mathcal{O}(m)$ in total) has to be tested whether they are in W. The test whether a set contains an element is in $\mathcal{O}(log(n))$. Removing $\mathcal{O}(n)$ vertices from a graph is in $\mathcal{O}((n+m) \cdot log(n))$. ☐

Pruning Rule 4.4. If $|W \cup X| < k$ and the algorithm is in a non-growing branch, then discard the current branch.

This rule is obviously correct, therefore a formal proof is omitted. The time complexity of this rule is $\mathcal{O}(1)$ if the sizes of the sets are stored as variables, which can be done easily without adding any time complexity to the operations that modify the sets.

The next rule tests whether all vertices $v \in W$ have enough neighbors to be in a highly connected subgraph if all $k - |W|$ vertices that may be added to W are neighbors of v.

Pruning Rule 4.5. If there is a vertex $v \in W$ such that $|N(v) \cap W| < \lfloor \frac{k}{2} \rfloor + 1 - k + |W|$, then discard the current branch.

Proof of correctness. Let $v \in W$ be a vertex such that $|N(v) \cap W| < \lfloor \frac{k}{2} \rfloor + 1 - k + |W|$. Let W' be a solution of size k. Then there are $k - |W|$ vertices in $W' \backslash W$ and $\lfloor \frac{k}{2} \rfloor + 1 - k + |W| + (k - |W|) = \lfloor \frac{k}{2} \rfloor + 1$. If v is adjacent to all of the vertices in $W' \backslash W$, then $|N(v) \cap W'| = |N(v) \cap W| + k - |W| < \lfloor \frac{k}{2} \rfloor + 1$. So W cannot be a subset of any solution of size k.

The running time of this rule is clearly in $\mathcal{O}(n + m \cdot log(n))$ and the proof is equivalent to the proof for the complexity of Pruning Rule 4.3, except that we do not have to remove the vertices from G. ☐

The following pruning rule tests if all vertices in W can have at least $\lfloor \frac{k}{2} \rfloor + 1$ neighbors based on $W \cup X$.

Pruning Rule 4.6. If there is a vertex $v \in W$ such that $|N(v) \cap (X \cup W)| < \lfloor \frac{k}{2} \rfloor + 1$ and the algorithm is in a non-growing branch, then discard the current branch.

Proof of correctness. Any solution $W' \supset W$, if one exists, is a subset of $W \cup X$, so v has to have at least $\lfloor \frac{k}{2} \rfloor + 1$ neighbors in $W \cup X$.

The time complexity of this pruning rule is $\mathcal{O}(n + m \cdot log(n))$. The proof is equivalent to the one for Pruning Rule 4.5. ☐

The next pruning rule removes vertices whose open or closed neighborhood is a subset of the open or closed neighborhood of a vertex v, if v is not part of any solution, which is trivially known when branching into $W \cup \{v\}$ resulted in no solution.

Pruning Rule 4.7. Let $v \notin W$ be a vertex. If there is no solution $W' \supset W$ such that $v \in W'$, then remove all vertices $u \in X$ with the following property from the graph: $((N(u) \cap Y) \subseteq (N(v) \cap Y)) \vee (N[u] \cap Y) \subseteq (N[v] \cap Y))$. Here, Y denotes the set of all vertices that can be in X or W in the descendents of the current search-tree vertex and is at most $X \cup W$ for non-growing branches. Else, Y is at most V.

Proof of correctness. We will prove the correctness by contradiction. Let us assume there is no solution $W' \supseteq W \cup \{v\}$, but a solution $W'' \supseteq W \cup \{u\}$ and $N(u) \subseteq N(v)$. Obviously, we can just replace u by v in W'' to obtain a solution $W' \supseteq W \cup \{v\}$.

Now let us assume the same, but with $N[u] \subseteq N[v]$ instead of $N(u) \subseteq N(v)$. Because $N[u] \subseteq N[v]$, there is an edge between u and v. However, since v is not in the solution, this edge is not in the highly connected subgraph induced by W''. So we can remove this edge from the graph G and then $N(u) \subseteq N(v)$, which we have already proved.

We now have to prove that, given that the algorithm is in a non-growing branch, we can use the intersection of the neighborhoods with $X \cup W$ instead of the neighborhoods itself. Since $X \cup W$ contains all vertices that may be in a solution $W' \supset W$, we can simply call the HCS algorithm with $G[X \cup W]$, which effectively does the same as the intersection with $X \cup W$.

Finally, the time complexity of the rule is clearly polynomial. A more detailed analysis and an idea for an algorithm that computes this rule can be found in the following text. □

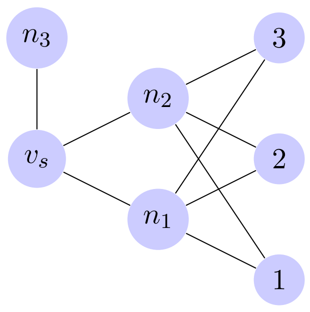


Figure 3: A graph with vertices n_1 and n_2 that share the same open neighborhood.

Considering the degeneracy-algorithm and the graph displayed in Figure 3, let us assume that n_1 was picked first as neighbor of v_s. Clearly, no highly connected subgraph containing these two vertices exists. Instead of picking n_2 next, we can remove it because it has the same open neighborhood as n_1. Additionally, we can remove n_3 since its neighborhood is a subset of the neighborhood of n_1.

The else-case - where the algorithm is not in a non-growing branch - is easier and faster to compute, since one has to compute the subset relations between the vertices only at the beginning and when the graph is permanently modified, for example in the outer loop of the degeneracy algorithm where the vertex with the lowest degree is removed. However, in this case the pruning rule may not remove many vertices in real-world applications.

The first case additionally requires to update the relations when vertices are removed from the set of candidates X or from the graph. This can be done by storing which vertices have to be removed from the graph such that $N(u) \subseteq N(v)$ or $N[u] \subseteq N[v]$, which are just the vertices in $N(u)\backslash N(v)$ or $N[u]\backslash N[v]$, respectively. When a vertex is removed from the graph or from X, the pairs to be added to the relations can be determined by a simple lookup.

More formally, we need for each relation a function relation_pairs : $V \to 2^{V^2}$ that maps a vertex v to all the pairs (u, w) that are not in the relation because of v. If the number of vertices in $N(u)\backslash N(w)$ or $N[u]\backslash N[w]$, respectively, is zero after v is removed, then we can add (u, w) to the relation. In practice, one could store the pairs that are not in the relation and the number of vertices mentioned before together and let the function return pointers to these locations, where the number effectively is a reference count. We additionally need a function that maps a vertex v to all locations of pairs that contain v - including pairs that are already in the relation -, which is used to update the relations when v is removed.

With these optimizations, the update is in $\mathcal{O}(n \cdot log(n))$ since $|\,\text{relation\_pairs}(v)| \in \mathcal{O}(n)$ for any vertex v, the number of pairs containing v is also in $\mathcal{O}(n)$, and insertion and deletion of elements in an AVL tree, which can be used as underlying data structure for a relation, is in $\mathcal{O}(log(n))$ [Sed83].

Note that there may be a similar pruning rule using graph isomorphism that removes a vertex u if there is no solution $W' \supset W$ such that $v \in W'$ and $v \notin W$ and there is an automorphism $h : V \to V$ such that $h(v) = u$. However, such a pruning rule would not have polynomial time complexity unless graph isomorphism is in P.

4.5.3 Branching rules

The branching rules discussed in this section take a pair (W, X) as input, where W is the set of chosen vertices and X the set of candidate vertices, and compute new pairs of the same type, which can be interpreted as children in the search tree.

Branching Rule 4.8 (Default branching rule)**.** Pick a vertex v out of X and return the two partial solutions with and without v, as shown in Algorithm 3.

Algorithm 3: A simple branching rule.

```
Function branch($W, X$)
    Pick a $v \in X$.
    return $\{(W, X\backslash\{v\}), (W \cup \{v\}, X\backslash\{v\})\}$.
```

Proof of correctness. If there is no solution for the instance (W, X), then there clearly is neither a solution for $(W \cup \{v\}, X\backslash\{v\})$ that additionally contains an arbitrary $v \in X$ nor a solution for $(W, X\backslash\{v\})$ that does not contain v. If there is no solution that contains $v \in X$ and no solution that does not contain v, there is also no solution to (W, X). The time complexity is clearly polynomial. $\square$

One can potentially optimize this branching rule by returning $\{(W \cup \{v_k\}, X\backslash\{v_1, \ldots, v_k\}) \mid k \in \{1, \ldots, |X| - k + |W| + 1\}\}$ for an arbitrary subset $\{v_1, \ldots, v_{|X|-k+|W|+1}\} \subseteq X$. This is the same as recursively applying the branching rule to $(W, X\backslash\{v\})$ and cutting off $(W, X\backslash\{v\})$ when $|X|$ is smaller than $k - |W|$.

The following branching rule chooses a vertex v out of W and adds neighbors of v until v has $\lfloor \frac{k}{2} \rfloor + 1$ neighbors in W. The vertex v is chosen such that it has the smallest number of possibilities to choose from its neighbors. If this is not possible because all vertices in W already have enough neighbors, it falls back to default branching with the optimization mentioned above.

Branching Rule 4.9. Pick a vertex v out of W such that the number of possibilities to add $\lfloor \frac{k}{2} \rfloor + 1 - |N(v) \cap W|$ neighbors of v is minimal, then pick the neighbors of v, as shown in Algorithm 4.

Algorithm 4: A more advanced branching rule.

```
Function branch($W, X$)
    $W' := \{w \in W \mid |N(w) \cap W| < \lfloor \frac{k}{2} \rfloor + 1\}$
    if $W'$ is non-empty then
        $Y := \arg\min_{v \in W'} \binom{|N(v) \cap X|}{\lfloor \frac{k}{2} \rfloor + 1 - |N(v) \cap W|}$
        Pick a vertex $v \in Y$ such that $|N(v) \cap X|$ is maximal.
        $Z := N(v) \cap X$
        $r := \lfloor \frac{k}{2} \rfloor + 1 - |N(v) \cap W|$
    else
        $Z := X$
        $r := 1$
    return $\{(W \cup \{v_k\}, X\backslash\{v_1, \ldots, v_k\}) \mid k \in \{1, \ldots, |Z| - r + 1\}\}$ for an
     arbitrary subset $\{v_1, \ldots, v_{|Z|-r+1}\} \subseteq Z$.
```

This branching rule requires that Pruning Rule 4.6 was already applied to W and X to assure the validity of the parameters for the binomial coefficient.

W' is the set of vertices in W that do not already have $\lfloor \frac{k}{2} \rfloor + 1$ neighbors in W. Y is the set of vertices $w \in W$ with the smallest number of possibilities to add vertices $w_1, ..., w_{\lfloor \frac{k}{2} \rfloor + 1 - |N(w) \cap W|} \in X \cap N(w)$ to W such that $|N(w) \cap (W \cup \{w_1\} \cup \ldots \cup \{w_{\lfloor \frac{k}{2} \rfloor + 1 - |N(w) \cap W|}\})| = \lfloor \frac{k}{2} \rfloor + 1$. Z is the set of neighbors we will choose vertices from and r is the number of vertices we want to add.

Proof of correctness. If W' is empty, then we simply fall back to Branching Rule 4.8, recursively applied to the first pair of the result of Branching Rule 4.8.

For the case where W' is non-empty, we basically pick a vertex v out of W'. If there is a solution $W'' \supset W$, then v has at least $\lfloor \frac{k}{2} \rfloor + 1$ neighbors in W''. Therefore, we can safely test all subsets of size $r = \lfloor \frac{k}{2} \rfloor + 1 - |N(v) \cap W|$ of neighbors of v and there is at least one subset S such that $S \cup W \subseteq W''$. If there is no solution $W'' \supset W$, then the branching rule obviously cannot add an additional solution because all returned solution candidates are supersets of W.

The time complexity for this rule is clearly polynomial, with the computation of W', Y, and Z being in $\mathcal{O}((n+m) \cdot log(n))$ time (refer to the proof for Pruning Rule 4.3) and the time complexity to compute the returned set being in $\mathcal{O}(n \cdot log(n))$. □

In this simple version of the branching rule we only add one vertex in each step and then recompute Y and Z. However, this recomputation may not be necessary in the $r-1$ following steps. In case $r > 1$, one can set r to $r-1$ in the next step because the vertex v from the previous step may still be one of the vertices with the lowest number of possibilities to add its neighbors. However, the added neighbor can have less possibilities to add its neighbors to W. Therefore, in Section 5.3 it will be tested which of the two variants perform better in practice. We call the simple version Branching Rule 4.9.1 and the modified version 4.9.2.

This branching rule may significantly decrease the number of possibilities that have to be checked. Let us assume that we have to add r neighbors of a vertex v and there are $b \geq r$ neighbors of v in X. There are $\binom{b}{r}$ possibilities to choose these r neighbors. We choose the vertex with the lowest number of possibilites first. So instead of checking $\binom{|X|}{k-|W|}$ possible solutions that contain W as a subset, we pick v_j, $r \leq j \leq b$, out of these b vertices and for v_j we choose $r-1$ vertices out of $\{v_1, ...v_{j-1}\}$ and for all of these combinations, we choose $k - |W| - r$ vertices out of $\{v_{j+1}, ..., v_b\} \cup (X \backslash N(v))$. This gives us the formula $\sum_{i=r}^{b} \binom{i-1}{r-1} \cdot \binom{|X|-i}{k-|W|-r}$. However, this is the number of possibilities if we apply the branching rule once for r steps and then switch back to the default branching rule. Analyzing this is relatively complex and does not tell us much about the performance in practice, therefore a more detailed analysis is omitted.

There is an optimization for both branching rules that forces that there are enough vertices left in the candidate set X such that $|X \cup W| \geq k$ in the next step or after the r vertices are added to W, respectively. For example, if for Branching Rule 4.9 $Z = X$ and $r = 1$, then in the next step there is a branch with $X = \emptyset$, which cannot be a solution if $|W| \neq k$. So we can safely remove some vertices from Z for Branching Rule 4.9 or force X to be of a minimum size for Branching Rule 4.8. For Branching Rule 4.8, we simply return $\{(W \cup \{v\}, X \backslash \{v\})\}$ if $|X| + |W| = k$. For Branching Rule 4.9, we remove $\delta := k - |W| - r - |X \backslash Z|$ vertices from Z if $\delta > 0$. These changes to the branching rules are obviously correct since we only discard branches that cannot lead to a solution.

5 Implementation details and experimental results

In this section, we give some details on the implementation (Section 5.1), the test data and environment (Section 5.2), and present the results of our tests (Section 5.3).

5.1 Implementation details

In this subsection, we will give some details on the implementation of the algorithms and the rules.

All code was written in C++, with exception of some code used by the implementation by Falk Hüffner, which was written in C. For information about the complexity of operations on C++ containers, refer to Table 1. The code was compiled using the Microsoft compiler included in Microsoft Visual Studio 2013.

The source code is available on the website of the Chair of Algorithmics and Complexity Theory (http://www.akt.tu-berlin.de/menue/software/).

The graph used by the algorithms for parameters degeneracy and h-index was implemented as a set of vertices, where each vertex has its own set of neighbors.

We avoided copying whole containers as much as possible because we assumed it to be very costly due to cache misses - since more space would be needed - and the overhead of the copying itself, so almost all operations on containers were accesses and updates of elements, insertions, and deletions and these updates were undone on returning from a function.

For the implementation of Pruning Rule 4.3 the number of neighbors in W of vertices in X was stored together with these vertices in a std::map and updated each time a vertex was added to or removed from W. For Pruning Rule 4.5, Pruning Rule 4.6, and Branching Rule 4.9 the number of neighbors of chosen vertices in the set of chosen vertices, that is $|N(v) \cap W|$ for $v \in W$, was stored together with these vertices and updated each time W was modified. The number $|N(v) \cap X|$ for $v \in W$ used in Pruning Rule 4.6 and Branching Rule 4.9 was stored and updated in a similar way. The optimized version of Branching Rule 4.8 was used and was implemented by simply iterating over the set of candidates X and only considering vertices that are greater than the current vertex based on the ordering of the C++ std::map - the keys are ordered ascendingly according to the C++ standard and iteration over a std::map is in this order, too.

We implemented the optimization for the h-index algorithm that called the branch_recursive function of the degeneracy algorithm as mentioned in Section 4.3.

5.2 Test data and environment

We used graphs from the article “A Graph Modification Approach for Finding Core-Periphery Structures in Protein Interaction Networks” by Sharon Bruckner et al. [BHK15]. The graphs vary in all parameters and include both sparse and dense graphs. The set of test graphs is split into a set of 30 small to medium graphs with up to 1045 vertices

Container \ Operation	Insertion	Search	Deletion
std::vector	$\mathcal{O}(1)$*	$\mathcal{O}(n)$	$\mathcal{O}(n)$
std::set	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$
std::map	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$

Table 1: Complexities of operations on C++ containers according to the C++ documentation (* means average case).

and 4590 edges and a set of two large graphs with more than 2000 vertices and 20000 edges. The exact numbers are shown together with the results in Tables 2 and 3.

The performance tests were done using an Intel Core i7-3930K processor with 3.2 GHz. Multiple tests were run in parallel to simulate real-world applications where often multiple tasks are executed concurrently and therefore there may be for example more cache misses in the shared cache than in a single-threaded test. This also has the effect that simultaneous multithreading is used like in most real-world applications. We spawned at most 11 threads since this processor supports 12 and we wanted to have some reserves such that unrelated tasks do not interfer with our tests. Each algorithm was given a time limit of ten minutes per instance of a graph G and a subgraph order k.

The results may differ slightly in a single-threaded test. However, this was not feasible in our case since the total running time would have been several days.

5.3 Test results

We ran several tests for comparison of the algorithms and the pruning and branching rules. We did not test Reduction Rule 4.2 and Pruning Rule 4.7 because they do not look promising since the conditions needed to apply them successfully may occur very rarely in practice. In the comparison tests of the algorithms, for the algorithms for parameters d and h Pruning Rules 4.3, 4.4, 4.5, and 4.6 were applied and Branching Rule 4.9.2 was used. The algorithm for parameter γ was used with all its optimizations that have been implemented by Falk Hüffner.

Table 2 shows several parameters of the graphs and the number of instances for which the algorithms ran more than one or ten minutes per graph. The degeneracy algorithm was slow only for one of these graphs, namely Graph 2181, which has a degeneracy of 41. Consistently with the time complexity of this algorithm, which is $\mathcal{O}(2^d \cdot n^{d+\mathcal{O}(1)})$, the degeneracy appears to be the most important factor for the running time. However, this algorithm performs much better than one would expect when only considering its complexity.

For the h-index algorithm, the h-index and the number of vertices and edges appear to be almost equally important.

The running time of the algorithm for the parameter edge isolation may also depend on more parameters than only n and m. Considering only the number of vertices and edges, the results are inconsistent: Graph 3676 has more vertices and edges than graph 502, but the edge isolation algorithm performs better for it. The lower degeneracy of graph 3676 may explain this behaviour, but this makes estimating the running time based on the graph parameters even harder.

Graph No.	n	m	d	h	#Results $\geq$ 1 min			#Results $\geq$ 10 min		
					Alg. d	Alg. h	Alg. γ	Alg. d	Alg. h	Alg. γ
27	32	183	10	13	0	2	0	0	0	0
122	57	186	15	15	0	5	0	0	0	0
131	52	118	6	9	0	0	0	0	0	0
139	100	306	7	13	0	2	0	0	1	0
166	811	4590	11	35	0	10	12	0	10	8
329	104	281	9	12	0	5	0	0	1	0
398	83	818	19	27	0	4	20	0	4	19
447	43	244	12	15	0	4	0	0	4	0
462	57	398	14	20	0	6	7	0	6	3
472	32	184	11	13	0	4	0	0	1	0
480	30	164	11	12	0	5	0	0	0	0
502	46	518	20	24	0	6	12	0	6	3
775	61	354	11	16	0	2	6	0	2	0
776	47	291	10	15	0	0	3	0	0	0
777	40	265	10	15	0	0	0	0	0	0
778	38	137	7	10	0	0	0	0	0	0
781	33	49	3	5	0	0	0	0	0	0
790	38	45	2	4	0	0	0	0	0	0
910	31	71	5	6	0	0	0	0	0	0
932	30	106	8	10	0	0	0	0	0	0
1403	32	42	3	5	0	0	0	0	0	0
2181	150	2294	41	50	4	14	47	2	14	46
3674	1045	2907	12	26	0	12	7	0	12	3
3676	229	785	9	18	0	4	0	0	3	0
3677	415	2122	15	28	0	7	20	0	5	18
3682	61	226	10	12	0	5	0	0	1	0
3697	34	64	4	6	0	0	0	0	0	0
3700	33	26	2	3	0	0	0	0	0	0
3723	381	3294	21	42	0	27	33	0	25	32
3729	49	109	7	9	0	0	0	0	0	0
Total					4 (0.7%)	124 (22.1%)	167 (29.8%)	2 (0.4%)	95 (17.0%)	132 (23.6%)

Table 2: Parameters of test graphs and number of results that took longer than one or ten minutes to compute.

Graph No.	n	m	d	h	#Results $\geq$ 1 min			#Results $\geq$ 10 min		
					Alg. d	Alg. h	Alg. γ	Alg. d	Alg. h	Alg. γ
5634	2046	25023	30	81	21	44	52	15	43	52
5737	2198	21635	44	81	41	77	80	34	75	74
Total					62 (43.7%)	121 (85.2%)	132 (93.0%)	49 (34.5%)	118 (83.1%)	126 (88.7%)

Table 3: Parameters of large test graphs and number of results that took longer than one or ten minutes to compute.

The results for large graphs shown in Table 3 are consistent with the results and observations for the smallers graphs in Table 2, except for the h-index algorithm which appears to run faster for graphs with lower degeneracy since both graphs have the same h-index and roughly the same number of vertices and edges, but the h-index algorithm is slower for the graph with larger degeneracy.

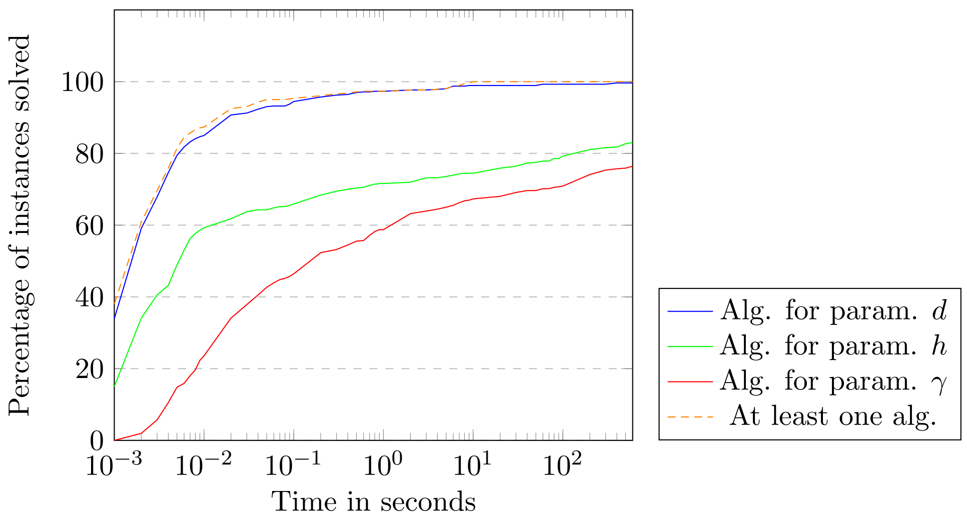


Figure 4: Comparison between algorithms for test graphs.

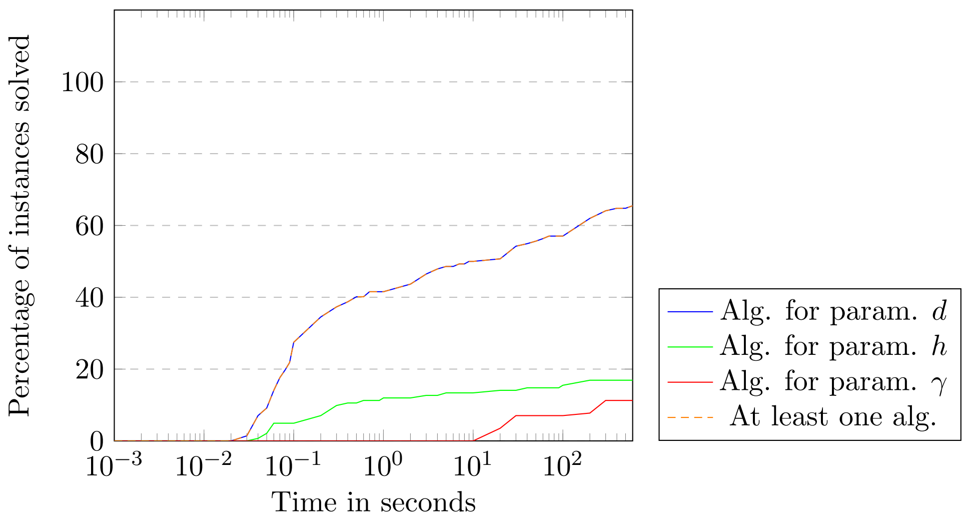


Figure 5: Comparison between algorithms for large test graphs.

Figure 4 shows the percentage of instances solved per algorithm, accumulated over time. Additionally, a line for "At least one algorithm" shows how many instances were solved by at least one algorithm, which implies that running all three algorithms in parallel would solve this instance within this time. The degeneracy algorithm performs much better than the other two algorithms in general, but some instances were solved faster by one of the other algorithms. About 34% of the instances were solved within one millisecond by the degeneracy algorithm and 85% within 10 milliseconds. The h-index algorithm solved about 20% less instances in the same time. The algorithm for parameter edge isolation needs relatively much time to solve even small instances, but solves only about 10% less instances than the h-index algorithm within 3 seconds.

For the large graphs from Table 3 the algorithm for parameter degeneracy clearly performs best in practice, as seen in Figure 5. All three algorithms are relatively slow even for small k. This may be due to the sorting by degree that has to be done before searching for highly connected subgraphs in the degeneracy and h-index algorithms. The edge isolation algorithm needs about one second to even find small highly connected subgraphs, while the other two algorithms find them within 30 to 40 milliseconds. Note that the degeneracy algorithm solves all instances faster than the other algorithms for these two graphs - in contrast to the smaller graphs, where it was slower on some instances.

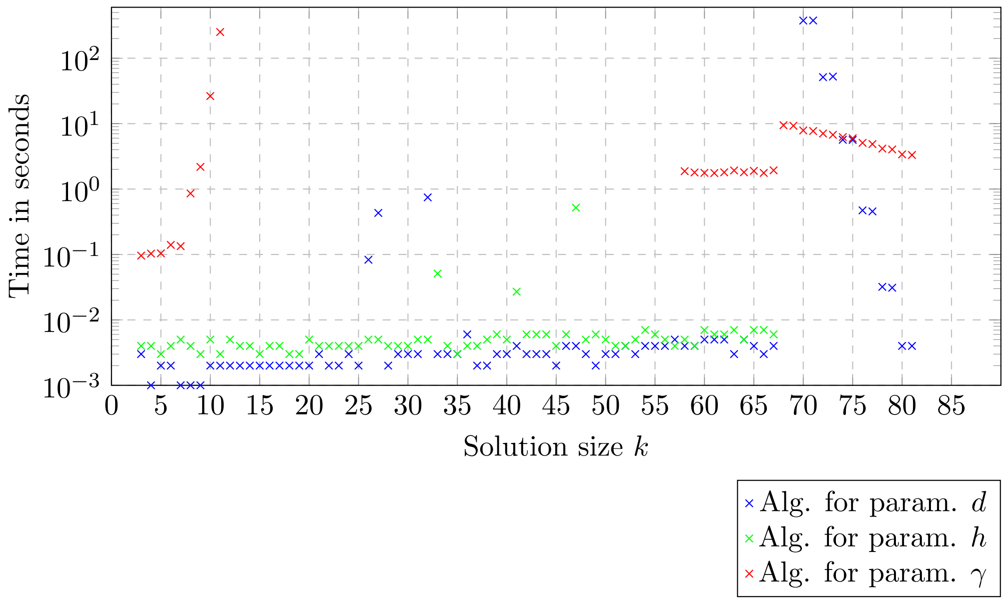


Figure 6: Duration for different orders k for graph 2181.

Figure 6 shows the relation between k and the running time for graph 2181. For the "yes"-instances $3 \leq k \leq 67$, the degeneracy and the h-index algorithm perform roughly the same, with the degeneracy algorithm being slightly faster in most of the instances. However, the algorithm for the parameter edge isolation performs far worse than the other two for these k. The h-index algorithm could not solve the other instances $68 \leq k \leq 81$, which are "no"-instances, within ten minutes, while the edge isolation algorithm solves them in under ten seconds per instance. Particularly interesting is the pattern in the running time of the algorithm for parameter degeneracy. For even k, the running time for k and $k + 1$ is about the same. This is most likely due to the fact that $\lfloor \frac{k}{2} \rfloor + 1 = \lfloor \frac{k+1}{2} \rfloor + 1$ for even k and therefore the pruning rules may remove almost the same vertices from the graph and so the search trees may look very similar for both k.

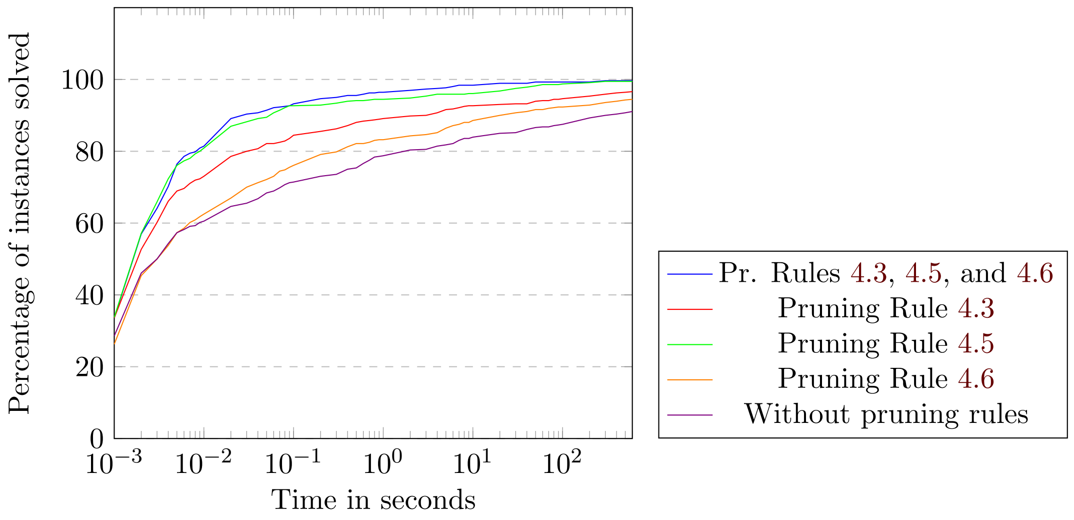


Figure 7: Comparison between Pruning Rules 4.3, 4.5, and 4.6 for the algorithm for parameter degeneracy. Pruning Rule 4.4 was always applied since it is computable with negligible overhead.

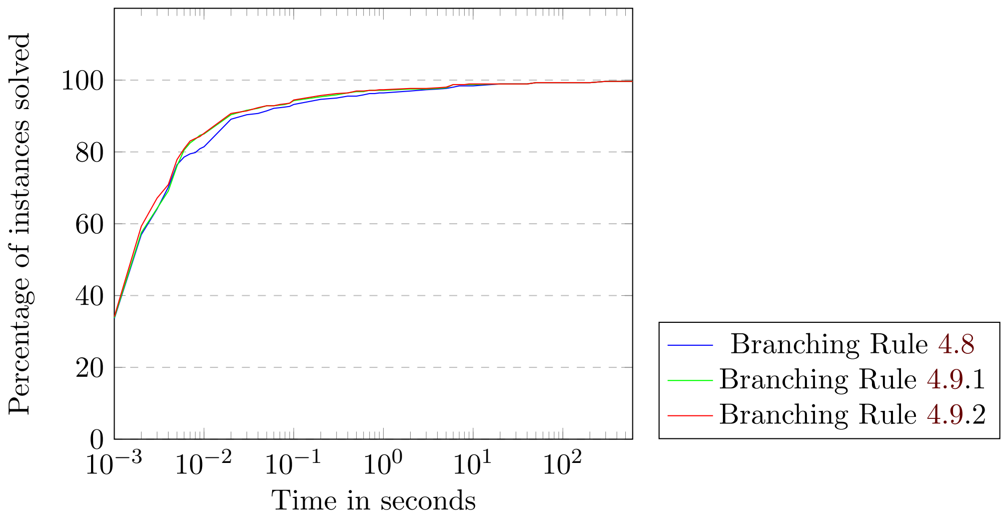


Figure 8: Comparison between Branching Rules 4.8, 4.9.1 and 4.9.2 for the algorithm for parameter degeneracy.

Figure 7 shows the differences in the running time of the algorithm for parameter degeneracy when different pruning rules were applied. Pruning Rule 4.4 was always applied since it has negligible overhead and is therefore not explicitly mentioned in the following.

Applying all three pruning rules performs best in general, but the performance is only slightly worse when only Pruning Rule 4.5 is applied. Pruning Rules 4.3 and 4.6 improved the performance not as much as Pruning Rule 4.5, but applying them made the algorithm solve several percent more instances in the same time compared to the algorithm without pruning rules.

Given this result, we applied these three pruning rules when we compared the different algorithms.

Figure 8 shows a comparison between the branching rules, again for the algorithm for parameter degeneracy. For this test, Pruning Rules 4.3 to 4.6 were applied. Clearly, there is not much difference between the running times of the different branching rules in this test. However, Branching Rule 4.9.2, where the possibilities were only recomputed when r neighbors have been added, is slightly faster than the other two in general. Remind that r is the number of neighbors of the picked vertex v that have to be added such that v has $\lfloor \frac{k}{2} \rfloor + 1$ neighbors in the partial solution.

5.4 Conclusion

The denegeracy algorithm performs best for almost all tested instances. This result is however not surprising since the complexity of this algorithm is lower than the complexities of the other two. Additionally, the algorithm for parameter edge isolation was originally designed for a more specialized problem and only a special case of this problem that is also almost the worst case solves the more general highly connected subgraph problem.

Pruning Rules 4.3 to 4.6 should always be applied and Branching Rule 4.9.2 should be used to get the best performance in practice.

The algorithms perform significantly better than expected considering their complexity. A possible explanation for this is that for vertices with relatively low degree the number of their neighbors of neighbors is likely small compared to the number of vertices n. Therefore, the number of possible subgraphs containing these vertices may often be significantly smaller than the upper bound for worst-case graphs used for the time complexity analysis. For dense graphs, it is likely that there are superpolynomial many subsets that induce a highly connected subgraph and therefore it is likely to find one of them relatively fast if the input is a "yes"-instance.

6 Summary and outlook

The results of this work are mainly that we found out that the algorithm for parameter degeneracy (Section 4.2) performs best in practice (Section 5.3), although it is relatively simple compared to the other algorithms for parameter h-index and edge isolation, and - as more theoretical results - that we found upper bounds for the order k of highly connected subgraphs (Section 3) and gaps of size five (Section 3.1).

The algorithms only solve the decision problem, meaning they stop after finding the first highly connected subgraph. One may be interested in finding all or distinct highly connected subgraphs. In the worst case, there are exponentially many highly connected subgraphs. However, finding distinct highly connected subgraphs only increases the running time by a linear factor since one can iteratively remove a highly connected subgraph from the graph and search for the next one in the modified graph. This approach would for example work for the marketing problem in social networks mentioned in the introduction.

The reduction and pruning rules (Section 4.5) may also be applicable for other algorithms and - with little modifications - for similar dense subgraph problems.

The existence of a limit of gap sizes is a very specific, yet interesting question that may be answered in future research and may lead to further understanding of graphs and graph problems.

Literature

[BHK15] S. Bruckner, F. Hüffner, and C. Komusiewicz. "A Graph Modification Approach for Finding Core–Periphery Structures in Protein Interaction Networks". In: *Algorithms for Molecular Biology* 10.16 (2015) (cit. on p. 24).

[Cyg+15] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized algorithms.* Vol. 3. Springer, 2015 (cit. on p. 10).

[Die00] R. Diestel. *Graph Theory.* Springer-Verlag New York, 2000 (cit. on p. 6).

[Him+16] A.-S. Himmel, H. Molter, R. Niedermeier, and M. Sorge. "Enumerating maximal cliques in temporal graphs". In: *Proceedings of the 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, (ASONAM '16).* IEEE Computer Society, 2016, pp. 337–344 (cit. on p. 5).

[Hir05] J. E. Hirsch. "An index to quantify an individual's scientific research output". In: *Proc. National Academy of Sciences* 102.46 (2005), pp. 16569–16572 (cit. on p. 8).

[HKS15] F. Hüffner, C. Komusiewicz, and M. Sorge. "Finding Highly Connected Subgraphs". In: *G.F. Italiano et al. (Eds.): SOFSEM 2015, LNCS* 8939 (2015), pp. 254–265 (cit. on pp. 5, 11, 17).

[HS00] E. Hartuv and R. Shamir. "A clustering algorithm based on graph connectivity". In: *Information Processing Letters* 76.4 (2000), pp. 175 –181 (cit. on pp. 5, 11).

[Hü+14] F. Hüffner, C. Komusiewicz, A. Liebtrau, and R. Niedermeier. "Partitioning biological networks into highly connected clusters with maximum edge coverage". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 11.3 (2014), pp. 455–467 (cit. on p. 5).

[KS15] C. Komusiewicz and M. Sorge. “An Algorithmic Framework for Fixed–Cardinality Optimization in Sparse Graphs Applied to Dense Subgraph Problems”. In: *Discrete Applied Mathematics* 193 (2015), pp. 145–161 (cit. on p. 5).

[LW08] G. Liu and L. Wong. “Effective Pruning Techniques for Mining Quasi–Cliques”. In: *Machine Learning and Knowledge Discovery in Databases, European Conference, ECML/PKDD 2008, Antwerp, Belgium, September 15-19, 2008, Proceedings, Part II.* 2008, pp. 33–49 (cit. on p. 5).

[LW70] D. R. Lick and A. T. White. “k-degenerate graphs”. In: *Canadian Journal of Mathematics* 22 (1970), pp. 1082–1096 (cit. on p. 7).

[MNS09] H. Moser, R. Niedermeier, and M. Sorge. “Algorithms and Experiments for Clique Relaxations—Finding Maximum s-Plexes”. In: *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA ’09).* Vol. 5526. LNCS. Springer, 2009, pp. 233–244 (cit. on p. 5).

[PYB13] J. Pattillo, N. Youssef, and S. Butenko. “On clique relaxation models in network analysis”. In: *Eur. J. Operational Research* 226.1 (2013), pp. 9–18 (cit. on p. 5).

[Sed83] R. Sedgewick. *Algorithms.* Addison-Wesley, 1983 (cit. on p. 21).

[Sor17] M. Sorge. “Be Sparse! Be Dense! Be Robust! Elements of Parameterized Algorithmics”. PhD thesis. Technische Universität Berlin, 2017 (cit. on pp. 5, 15, 17).

[SUS07] R. Sharan, I. Ulitsky, and R. Shamir. “Network-based prediction of protein function”. In: *Mol. Syst. Biol.* 3.88 (2007) (cit. on p. 5).

[SW97] M. Stoer and F. Wagner. “A Simple Min Cut Algorithm”. In: *Journal of the ACM* 44.4 (1997), pp. 585–591 (cit. on p. 18).