Editing to (P5, C5)-free Graphs - a Model for Community Detection?

Bachelor Thesis of

Philipp Schoch

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisor:	Michael Hamann

Time Period: 22nd June 2015 – 21st October 2015

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 21st October 2015

Abstract

We study the following method of detecting communities in a graph G:

Find a minimal set F of edges to delete from and edges to add to the graph G, such that the resulting graph G_F after applying these edits does not contain any P_5 or C_5 as an induced subgraph. Then use the connected components of the edited graph G_F as communities for the initial graph G.

The first of the two steps is called the (P_5, C_5)-free editing problem. The set F is called an editing set. The idea stems from [NG13], who did the same thing with P_4s and C_4s instead of P_5s and C_5s.

We also look at a variation of the method where only deletes of edges but no inserts of edges are allowed. That is we look for a solution of the (P_5, C_5)-free deletion problem instead of the (P_5, C_5)-free editing problem.

In order to find out whether this is a practical method for community detection we have to address two issues:

1. the computational complexity of the (P_5, C_5)-free editing problem and of the (P_5, C_5)-free deletion problem
2. find out whether the resulting connected components actually define meaningful communities, when applied to graphs that represent real world data

Our main contribution to the first issue is that we prove the NP-completeness of (P_5, C_5)-free editing and (P_5, C_5)-free deletion by generalizing existing NP-completeness proofs for similar problems. Furthermore we discuss two algorithms that can be used in order to find an exact or a heuristic solution to the (P_5, C_5)-free editing or the (P_5, C_5)-free deletion problem. These algorithms where derived by adapting the algorithms used by [NG13].

We then implement these algorithms in order to find editing sets for some real world graphs. When applying these algorithms we found that the heuristic performs much worse for (P_5, C_5)-free editing and (P_5, C_5)-free deletion than it does for (P_4, C_4)-free editing. But by combining the heuristic algorithm with the exact algorithm we still found some reasonable (P_5, C_5)-free editing/deletion sets for these real world graphs.

We visualized the graphs together with the found editing/deletion sets in order to analyse the quality of the resulting connected components/communities. Thereby we contribute to the second of the above two questions.

We found that the resulting communities are good on some graphs but on other graphs the size of the found communities is too big. These identified communities which are too big often contain the union of several underlying ground truth communities. As we cannot distinguish these underlying communities, we get less information about the graph than we would get by using a different algorithm, which also tells us the border between these underlying communities.

Deutsche Zusammenfassung

Diese Bachelorarbeit beschäftigt sich mit einem bestimmten Ansatz für die Aufgabe der Community Erkennung in Graphen. Community Erkennung bedeutet, dass man versucht für einen gegebenen Graphen eine Partitionierung der Knotenmenge in

Teilmengen zu finden, mit der Eigenschaft, dass die Knoten innerhalb der Teilmengen stärker miteinander verbunden sind als mit den Knoten aus den anderen Teilmengen. Das heißt, es gibt mehr Kanten innerhalb der einzelnen Teilmengen als zwischen den Teilmengen. Diese Teilmengen heißen *Communities*. Diese stärkere Verbindung zwischen Knoten derselben Community kann dabei auf eine Gemeinsamkeit oder eine gemeinsame Funktion in dem Netzwerk hinweisen, die sich alle Knoten aus derselben Community miteinander teilen.

Es wurden schon sehr viele verschiedene Ansätze entwickelt um die Aufgabe der Community Erkennung zu lösen. Ein relativ neuer Ansatz stammt von [NG13]. Um die Communities in einem Graphen G zu finden, geht man mit der Methode von [NG13] so vor:

Finde eine minimale Menge F von Kanten, die man von dem Graphen G entfernen oder zu dem Graphen G hinzufügen muss, so dass der daraus resultierende Graph G_F keinen P_4 und keinen C_4 als induzierten Subgraphen enthält. Benutze dann die Zusammenhangskomponenten von G_F als Communities für G.

Der erste der beiden Schritte bedeutet, dass wir das sogenannte (P_5, C_5)-free Editing Problem lösen. Die Menge F wird Editing Set genannt.

[NG13] haben diese Methode auf einige Graphen angewendet, die Daten aus der realen Welt repräsentierten. Dabei haben sie einige sinnvolle Communities gefunden.

Wir wollen uns hier mit einer Abwandlung dieser Methode beschäftigen, in der man nur verlangt, dass der modifizierte Graph G_F keine P_5s und keine C_5s enthält. Außerdem werden wir auch noch untersuchen was passiert, wenn man nur das Entfernen von Kanten erlaubt aber nicht das Einfügen von neuen Kanten. Das heißt wir werden nicht nur das (P_5, C_5)-free Editing Problem sondern auch das (P_5, C_5)-free Deletion Problem behandeln.

Um herauszufinden, ob dies ein sinnvoller Ansatz ist um Communities zu finden müssen wir zwei Fragestellungen bearbeiten:

1. Wie aufwändig ist es das (P_5, C_5)-free Editing Problem oder das (P_5, C_5)-free Deletion Problem zu lösen?
2. Werden die resultierenden Zusammenhangskomponenten tatsächlich sinnvolle Communities repräsentieren, wenn man die Methode auf reale Daten anwendet?

Unser wichtigster Beitrag zu der ersten Fragestellung ist es, dass wir die NP-Vollständigkeit des (P_5, C_5)-free Editing Editing Problems und des (P_5, C_5)-free Deletion Problems beweisen. Dazu verallgemeinern wir bereits existierende NP-Vollständigkeitsbeweise für ähnliche Probleme.

Außerdem diskutieren wir einige Algorithmen, die man verwenden kann um das (P_5, C_5)-free Editing und das (P_5, C_5)-free Deletion Problem zu lösen. Wir haben hierzu die von [NG13] verwendeten Algorithmen an unser Problem angepasst. Einer dieser Algorithmen berechnet eine exakte Lösung, wobei die Laufzeit exponentiell in der Anzahl der benötigten Editieroperationen ist, die man auf den Graphen anwenden muss. Die Laufzeit ist aber fixed-parameter tractable. Der andere Algorithmus ist eine Heuristik, die nur polynomielle Zeit braucht. Dafür gibt es aber keine Garantie, dass die Anzahl der von der Heuristik angewandten Editieroperationen minimal ist.

Wir haben diese Algorithmen auf einige Graphen angewandt, die Daten aus der realen Welt repräsentieren. Dabei ist uns aufgefallen, dass der heuristische Algorithmus beim (P_5, C_5)-free Editing Problem viel schlechtere Ergebnisse liefert als beim (P_4, C_4)-free Editing Problem. Trotzdem gelang es uns einige brauchbare (P_5, C_5)-free Editing

Sets und (P_5, C_5)-free Deletion Sets zu berechnen, indem wir den heuristischen Algorithmus mit dem exakten Algorithmus kombiniert haben.

Wir haben die Graphen dann zusammen mit den gefundenen Editing / Deletion Sets visualisiert um die Qualität der resultierenden Communities zu untersuchen. Dabei leisten wir also einen Beitrag zu der zweiten der oben genannten Fragestellungen.

Es stellt sich heraus, dass die resultierenden Communities auf manchen Graphen sinnvoll sind. Aber auf anderen Graphen dagegen gibt es das Problem, dass oft mehrere Communities miteinander verschmolzen werden. Eine von uns gefundenen Community ist hier manchmal die Vereinigung von mehreren zugrundeliegenden tatsächlichen Communities. Diese zugrundeliegenden Communities, die hier verbunden wurden, möchten wir aber oftmals voneinander unterscheiden können, was so aber nicht möglich ist.

Contents

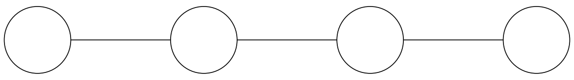
1. Introduction

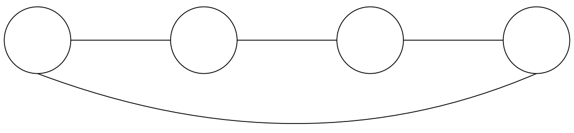
Graphs representing real world data often have a certain structure, where the edges are not distributed completely equally over the graph. Instead there is a high concentration of edges within special groups of vertices and low concentration of edges between these groups. This structure often indicates common properties and/or a common function or role inside of the network shared by vertices belonging to the same group. These groups of vertices are called *communities*. For example in a social network the communities might represent families, friendship circles, groups of people living in the same village, groups of people sharing a common interest and so on. However apart from social networks there are many other types of networks that contain a community structure, for example biological networks such as protein-protein interaction networks, the World Wide Web and so on.

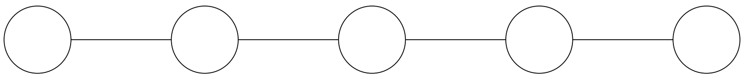
The problem of identifying these communities using only the information contained in the graph topology is called *community detection*. This task is not trivial as there are still edges between different communities, there are only statistically fewer of them than inside of the communities. Furthermore note that often the underlying correct solution which we want to detect is not sharply defined. Some vertices may be related to different tasks or roles in the network simultaneously. And the tasks may also be related to each other. For example assume we want to cluster web pages according to the topic that they deal with. Sometimes the web pages are related to more than one topic. And also some of the topics are related to each other. Therefore sometimes it is not clear whether a certain information contained in a web page belongs more to some topic A or more to some other topic B. Community detection is an important issue in many different disciplines like sociology, biology, computer science and so on. There has been much research in this area and many different methods and algorithms have been proposed to solve this task. In this bachelor thesis we study a variation of a new method proposed by [NG13]. They introduced a method of community detection primarily for the context of social networks that works as follows:

Assume we have a graph G. In order to apply the definition of [NG13] one has to find a closest (P_4, C_4)-free graph G_F. (P_4, C_4)-free means that G_F does not contain any P_4 or C_4 as an induced subgraph, where P_4 and C_4 are the graphs depicted in Figure 1.1a and Figure 1.1b. By “closest” (P_4, C_4)-free graph we mean that the set F of edges that one has to delete from or add to G in order to get G_F has a minimal cardinality. Now [NG13] defined the communities in G to be the connected components of G_F.

The problem of finding a closest (P_4, C_4)-free graph G_F - or the corresponding *editing set* F which transforms G into G_F - is called the (P_4, C_4)-free editing problem. A more formal definition of (P_4, C_4)-freeness and of editing problems is given in Section 1.2.


(a) P_4


(b) C_4


(c) P_5

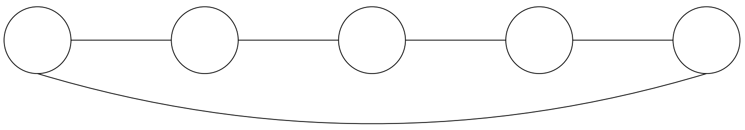

(d) C_5

Figure 1.1.: P_k and C_k for $k = 4$ and $k = 5$

As the (P_4, C_4)-free editing problem is NP-complete, it is usually not feasible to calculate the exact editing set. But using a heuristic approach [NG13] were still able to test their definition on some graphs, which represented real world data. In doing so they found that their definition actually identified sensible communities.

In the end they proposed relaxing the (P_4, C_4)-freeness condition to a (P_5, C_5)-freeness condition as a topic for further research. In this bachelor thesis we do that. Additionally we also study what happens if we allow only edge deletions when modifying the graph G to a (P_5, C_5)-free graph G_F. That is we also look at the (P_5, C_5)-free deletion and not only at the (P_5, C_5)-free editing problem.

[CK90] shows that each connected (P_4, C_4)-free graph contains a clique X, such that every vertex, which does not belong to X is adjacent to a vertex in X. Such a clique is called a dominating clique. Therefore we know that the connected components of a close (P_5, C_5)-free graph G_F of some graph G each contain a dominating clique. This "strong cohesion" in these components can be seen as a justification of the idea to use these components as communities for the graph G. Especially if G_F is the solution of the deletion problem and not the editing problem, then we know that these dominating cliques are also dominating cliques for the communities in G and not only in G_F. So by solving the (P_5, C_5)-free deletion problem we get communities that each contain a dominating clique.

In this bachelor thesis we want to find out whether solving the (P_5, C_5)-free editing or the (P_5, C_5)-free deletion problem is a suitable approach to identify communities in a graph.

To do so we need to answer two questions:

1. What is the computational complexity of the (P_5, C_5)-free editing and the (P_5, C_5)-free deletion problem? Is it at all feasible to edit a graph G of reasonable size to a closest or at least to a close (P_5, C_5)-free graph?
2. Do the connected components of the resulting graph G_F actually define meaningful communities for graphs that represent real world data?

Our main contribution to the first question is a NP-completeness proof for the (P_5, C_5)-free editing and the (P_5, C_5)-free deletion problem. We derived this proof by generalizing existing proofs of the NP-completeness of the (P_4, C_4)-free editing problem and of some other related problems. Furthermore we theoretically discuss two algorithms that can be used to calculate close (P_5, C_5)-free graphs. They are similar to the algorithms used in [NG13]. One of those two algorithms solves the (P_5, C_5)-free editing and the (P_5, C_5)-free deletion problem exactly. Its running time is exponential in the number k of needed edits, but it is still fixed-parameter tractable. The other algorithm is a heuristic algorithm, that needs only polynomial time. But we have no theoretic guarantee that the resulting graph is actually a *closest* (P_5, C_5)-free graph.

We implemented these algorithms and applied them to some real world graphs. Thereby we supplement the theoretical analysis of their running time by some practical studies and in the case of the heuristic we also evaluate the size of the calculated editing sets. Finally we combined both algorithms in order to calculate heuristic solutions of the (P_5, C_5)-free editing and the (P_5, C_5)-free deletion problem for the same real world graphs that were also used by [NG13]. We visualized the results in order to analyse the quality of the resulting connected components. Thereby we contribute to the second of the above mentioned questions.

The thesis is structured as follows:

Section 1.1 presents existing results that are related to the topic of this thesis. Section 1.2 of this chapter defines and explains terms and notations that are used in this thesis.

In Chapter 2 we prove the NP-completeness of the (P_5, C_5)-free editing problem by generalizing existing NP-completeness results by [EMC88], [LWGC12] and [NG13]. After that we show that the prove can be generalized even more to show the NP-completeness of other problems like for example the (P_5, C_4, C_5)-free editing problem or the (P_6, C_6)-free editing problem. Finally we show that the deletion versions of these problems are also NP-complete.

Chapter 3 discusses the two aforementioned algorithms that can be used to calculate close (P_5, C_5)-free graphs. Section 3.1 makes some notes on the data structure that we used to represent graphs. Section 3.2 and Section 3.3 introduce a P_5, C_5 counting algorithm and a P_5, C_5 search algorithm that are needed as subroutines for the two editing algorithms. Finally Section 3.4 discusses the exact editing algorithm. And Section 3.5 discusses the heuristic editing algorithm. For both editing algorithms we also show a modified version that solves the (P_5, C_5)-free deletion problem in the corresponding chapters.

Chapter 4 discusses the results of implementing the algorithms described in Chapter 3 and applying them to some real world graphs. We did that for two reasons. First we wanted to evaluate the algorithms themselves as means to solve the (P_5, C_5)-free editing and the (P_5, C_5)-free deletion problem. And second we wanted to make some conclusions about (P_5, C_5)-free editing in general, regarding the quality of the communities induced by a close (P_5, C_5)-free graph. We did the first task mainly in Sections 4.1, 4.2 and 4.3. Section 4.1 evaluates the heuristic editing algorithm. Section 4.2 tests an attempt to optimize the running time of the exact algorithm. It turns out that the running time of the algorithm can actually be reduced by more than a half if the optimization is used correctly. Section 4.3 describes how we combined the heuristic algorithm with the exact algorithm in order to improve the result calculated by the heuristic algorithm. Finally the second of the two tasks described above - analysing the quality of the community structure yielded by (P_5, C_5)-free editing and (P_5, C_5)-free deletion - is done in Section 4.4. To do so we visualised the graphs together with the applied edge edits and the resulting communities.

1.1. Related Work

On the topic of (P_5, C_5)-free graphs we want to mention two interesting characterizations. To do so we need a give a few definitions: A subset $X \subseteq V(G)$ of vertices of some graph G is called a *dominating set*, if each vertex $v \in V(G) \setminus X$ of G which does not belong to X is adjacent to some vertex in X. A *connected* dominating set is a dominating set X such that the subgraph $G[X]$ induced by X is connected. A *dominating clique* is a dominating set X, such that the subgraph $G[X]$ induced by X is a clique.

[BT90] shows that a graph G is (P_5, C_5)-free if and only if every connected induced subgraph contains a dominating clique. Note that this statement also contains the previously mentioned result of [CK90], which says that each connected (P_5, C_5)-free graph contains a dominating clique.

[Zve03] shows that a graph G is (P_5, C_5)-free if and only if for each connected induced subgraph the minimum size of a *connected* dominating set is *not* bigger than the minimum size of *any* dominating set.

Unfortunately it is difficult (or impossible) to use these characterizations in order to design an algorithm that checks efficiently whether a graph G is (P_5, C_5)-free. Even if we found an algorithm that checks efficiently whether G contains a dominating clique, doing that for each connected induced subgraph of G would presumingly be much harder. Note that both characterizations contain this "for each induced subgraph"-part. Therefore we do not use these characterizations in the algorithms described in Chapter 3. Instead we use a more straightforward approach in order to search for P_5s and C_5s.

There has also been much research on the topic of edge modification problems, not only in the context of (P_5, C_5)-freeness. For example [BBD06, p. 1825] contains a table showing the complexity status of the Π-edge editing and the Π-edge deletion problem for many graph classes Π. [Cai96] shows that the edge editing problem is fixed-parameter tractable for graph classes, which can be characterized by a finite set of forbidden induced subgraphs.

Important for our NP-completeness proof in Chapter 2 are the papers of [EMC88], [LWGC12] and [NG13]. [EMC88] shows that the (P_l)-free deletion problem is NP-complete for $l \geq 3$. [LWGC12] uses the same reduction as [EMC88] in the case of $l = 4$ to show that the (P_4, C_4)-free deletion problem is NP-complete. Finally [NG13] further generalizes this proof. They still use the same reduction as in [LWGC12], to show that the (P_4, C_4)-editing problem is NP-complete. However we did not find any paper that generalizes [EMC88]s completeness results also for the case of $l = 5$. So we do this generalization by ourselves in Chapter 2.

1.2. Notation and Basic Definitions

A *graph* $G = (V, E)$ is a (finite) set V of *vertices* and a set E of *edges*. An edge $e = \{u, v\}$ is a connection between two vertices $u, v \in V$. We denote the set of vertices of G by $V(G) := V$ and the set of its edges by $E(G) := E$. For each vertex $u \in V$ let $N(u) := \{v \in V \mid \{u, v\} \in E\}$ denote the set of neighbours of u. In this bachelor thesis by the word graph we refer to an undirected, simple graph (except when explicitly noted otherwise). A simple graph is a graph in which there are no two different edges between the same two vertices u and v and in which a vertex must not be adjacent to itself.

A graph H is called a *subgraph* of some graph G, if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Of course an edge $\{u, v\}$ of G may only appear in a subgraph H if its endpoints appear in H, too, that is $u, v \in V(H)$. If H contains all edges of G fulfilling this requirement, then H is called an *induced* subgraph of G. Note that an induced subgraph can be specified solely by giving its vertex set. Given a set $S \subseteq V(G)$ of vertices of G we denote the induced subgraph of G with the vertex set S by $G[S]$.

A *path* $p := (v_1, \ldots, v_n)$ from some vertex $u := v_1$ to some vertex $v := v_n$ in a graph G is a series of vertices $v_1, \ldots, v_n \in V(G)$ such that $\{v_i, v_{i+1}\} \in E(G)$ for each $1 \leq i < n$. A *connected component* of some graph G is a subset $X \subseteq V(G)$ of vertices such that for each pair of vertices $u, v \in X$ there exists a path from u to v, but there is no path from any $u \in X$ to any $w \in V(G) \setminus X$. If there is a path between each pair of vertices in G, then G is called *connected*. That is G is called connected if it consists of one connected component only.

A graph G, in which there is an edge between each pair of vertices $u, v \in V(G)$, is called a clique. For example Figure 1.2 depicts a clique containing 6 vertices. The clique containing p vertices is sometimes called K_p. Thus Figure 1.2 depicts a K_6.

For $k \in \mathbb{N}$ we define P_k to be the graph consisting of a simple path containing k vertices. And C_k shall be the graph consisting of a simple circle containing k vertices. More precisely P_k consists of the vertices $\{v_1, v_2, \ldots, v_k\}$ and edges $\{\{v_i, v_{i+1}\} \mid 1 \leq i < k\}$. And C_k consists of the vertices $\{v_1, v_2, \ldots, v_k\}$ and edges $\{\{v_i, v_{i+1}\} \mid 1 \leq i < k\} \cup \{\{v_k, v_1\}\}$. As an example 1.1c depicts a P_5 and 1.1d depicts a C_5.

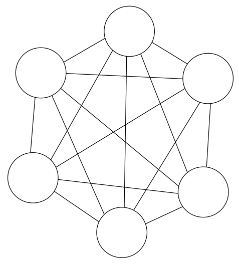


Figure 1.2.: a clique

We shall now introduce the notion of (P_5, C_5) freeness, (P_4, C_4) freeness, (P_5, C_4, C_5) freeness and so on. Let H be some graph. A graph G is called H-free if it does not contain H as an induced subgraph. More precisely G is called H-free if it does not contain an induced subgraph that is isomorphic to H. Two graphs H and H' are called isomorphic to each other if there is a bijective mapping $f : V(H) \mapsto V(H')$ such that $\{u, v\} \in E(H)$ if and only if $\{f(u), f(v)\} \in E(H')$ for all $u, v \in V(H)$. That is H and H' are isomorphic if H' is basically equal to H except that some vertices have been renamed in H'.

Let $H_1, \ldots, H_p$ be some graphs. $(p \in \mathbb{N}, p \geq 2)$ Then G is called $(H_1, \ldots, H_p)$-free if G does not contain an induced subgraph that is isomorphic to one of the graphs $H_1, \ldots, H_p$. In other words G is simultaneously H_1-free, H_2-free, …and H_p-free. In this paper we are primarily interested in (P_5, C_5)-freeness. In this case we have $p := 2$, $H_1 := P_5$ and $H_2 := C_5$.

Let us now introduce the notion of edge modification problems. Let $G = (V, E)$ be some graph. An *edge editing set* for G is a set F containing edges that shall be added to G and edges that shall be deleted from G. More formally $F \subseteq \{\{u, v\} \mid u, v \in V\}$ is a set of unordered pairs of vertices in V. And the result of applying F to G is denoted by $G_F = (V, E \triangle F)$. Here $E \triangle F := (E \setminus F) \cup (F \setminus E)$ denotes the symmetric difference of E and F. That is the edges in $F \cap E$ are removed from G and the edges in $F \setminus E$ are added to G. If F contains edge deletions only - that is $F \subseteq E$ - then F is also called an *edge deletion set*.

We will use the term “edge modification problem” as a generic term for edge editing and for edge deletion problems. Let Π be some graph property. The Π (edge) editing (search) problem and the Π (edge) deletion (search) problem are defined as follows:

Π - edge editing (search) problem

Instance A graph G.

Question Find an editing set $F \subseteq \{\{u, v\} \mid u, v \in V\}$ such that G_F satisfies Π and $|F|$ is minimal.

Π - edge deletion (search) problem

Instance A graph G.

Question Find a deletion set $F \subseteq E$ such that G_F satisfies Π and $|F|$ is minimal.

We are primarily interested in the (P_5, C_5)-free editing and the (P_5, C_5)-free deletion problem where Π is the property of being (P_5, C_5)-free.

The notion of NP-completeness is usually only defined for decision problems, where the solution to an instance of the problem is either “yes” or “no”. Therefore as we want to talk about NP-completeness in Chapter 2 we need to define a decision version of these problems:

Π - edge editing *decision* problem

Instance A pair (G, k) consisting of a graph G and an integer $k \geq 0$.

Question Is there an editing set $F \subseteq \{\{u, v\} \mid u, v \in V\}$ such that G_F satisfies Π and $|F| \leq k$?

Π - edge deletion *decision* problem

Instance A pair (G, k) consisting of a graph G and an integer $k \geq 0$.

Question Is there a deletion set $F \subseteq E$ such that G_F satisfies Π and $|F| \leq k$?

When talking about a Π edge editing problem it should be clear from the context whether we refer to the Π edge editing *search* problem or to the Π edge editing *decision* problem. In the context of NP-completeness we usually refer to the decision problem, otherwise we usually refer to the search problem. As this bachelor thesis does not discuss *vertex* deletion problems there is no risk of confusion when we use only the term Π deletion problem instead of Π *edge* deletion problem. Similarly we will often refer to the Π *edge* editing problem just by Π editing problem.

2. NP Completeness

In this bachelor thesis we are primarily interested in the (P_5, C_5)-free editing *search* problem and the (P_5, C_5)-free deletion *search* problem.

But as the notion of NP-completeness is usually only defined for decision problems we will prove the NP-completeness of the (P_5, C_5)-free editing *decision* problem and the (P_5, C_5)-free deletion *decision* problem (and some more related decision problems) here.

However note that if the decision version is not polynomial time solvable then neither is the search version. Assuming the search version were polynomial time solvable, then given an instance (G, k) of the decision version we might just calculate a minimal edge editing set F and then check whether $|F| \leq k$. So when we show the "hardness" of the decision version, then the search version is "hard", too.

Note that here in Chapter 2 by the (P_5, C_5)-free editing problem and the (P_5, C_5)-free deletion problem and so on we implicitly refer to the corresponding decision problem and not the search problem.

We will start in Section 2.1 by proving the NP-completeness of the (P_5, C_5)-free editing problem by generalizing the related NP-completeness proofs mentioned in Section 1.1. Then as an extra bonus in Section 2.2 we will show that the proof can be generalized even more in order to show the NP-completeness of related editing problems like for example the (P_5, C_4, C_5)-free editing problem or the (P_6, C_6)-free editing problem. Finally in Section 2.3 we will show that the corresponding deletion problems are NP-complete, too.

2.1. NP Completeness of (P_5, C_5)-free Editing

In this section we show that the P_5, C_5-free editing problem is NP-complete. Remember the definition of this problem from Section 1.2:

(P_5, C_5)**-free editing decision problem**

Instance A pair (G, k) consisting of a graph $G = (V, E)$ and an integer $k \geq 0$.

Question Is there an edge editing set $F \subseteq \{\{u, v\} \mid u, v \in V\}$ such that G_F is (P_5, C_5)-free and $|F| \leq k$?

As noted in Section 1.1 this proof is based on related NP-completeness proofs from [EMC88], [LWGC12] and [NG13].

The reduction that we use in the following Theorem 2.1 equals the one from [EMC88] for the case of $l = 5$.[1] Only the naming of the components of the graph is a bit different from [EMC88] and corresponds more to [NG13] and [LWGC12]. The argument why this reduction is still correct for our problem is in most parts based on [NG13] and [LWGC12]. (Although these two papers only discuss P_4 and (P_4, C_4)-free editing)

Theorem 2.1. *The (P_5, C_5)-free editing problem is NP-complete.*

Proof. The containedness is clear. We show its hardness by giving a reduction from the Exact 3-Cover decision problem which is defined as follows:

Exact 3-Cover

Instance A pair (S, C) consisting of a set $S = \{s_1, s_2, \dots, s_n\}$ of elements and a collection $C = \{S_1, S_2, \dots, S_m\}$ of subsets of S with $|S_i| = 3$.

Question Is there a subcollection $T \subseteq C$ such that S is the disjoint union of all 3 sets in T. That is $S := \bigcup_{T_i \in T} T_i$ and $T_i \cap T_j = \emptyset$ for each $T_i, T_j \in T$ with $T_i \neq T_j$.

Note that the number of subsets contained in a proper subcollection T is $t := \frac{n}{3}$. (Let us assume n is a multiple of 3, else the 3-cover instance would be trivially false.)

Given an instance (S, C) of Exact 3-Cover we construct an instance (G, k) of (P_5, C_5)-free editing editing as follows: First we add a clique with the vertices $S = \{s_1, s_2, \dots, s_n\}$ to G. Then we set the maximum number of edits to $k := 3(m - t) \cdot r + (r - 3t)$ where $r := \binom{n}{2}$ is the number of edges in S. For each $S_i \in S$ we add three cliques X_i, Y_i and Z_i to G such that $|X_i| = r$ and $|Y_i| = |Z_i| = 3k$. The newly added cliques shall be disjoint from each other and from the rest of G. Finally we add all possible edges between S_i and X_i, between X_i and Y_i and between Y_i and Z_i to G. Figure 2.1 depicts the graph G for the example that $S = \{s_1, \dots, s_6\}$ and $C = \{S_1, S_2, S_3\}$ with $S_1 := \{s_1, s_2, s_3\}, S_2 := \{s_3, s_4, s_5\}$ and $S_3 := \{s_4, s_5, s_6\}$. Note that Figure 2.1 shows which of the vertices in S are adjacent to which of the other cliques and which of the other cliques are adjacent to each other. But the number of depicted edges between one pair of cliques or one pair of a clique and an s_i is too low due to lack of space. For example there should actually be $\binom{6}{2} = 15$ instead of just four edges between s_1 and X_1.

Given an instance (S, C) of Exact 3-Cover the corresponding instance (G, k) of (P_5, C_5)-free editing is polynomial in the size of the instance (S, C). Also the calculation of (G, k) from (S, C) is not very complicated and is obviously feasible in polynomial time.

Now let us show that (S, C) is a yes-instance of Exact 3-Cover if and only if the corresponding instance (G, k) is a yes-instance of (P_5, C_5)-free editing, so we know that this is a proper reduction from Exact 3-Cover to (P_5, C_5)-free editing:

$\Rightarrow$ Assume there is a solution $T \subseteq C$ to the Exact 3-Cover instance (S, C), that is S is the disjoint union of the 3-sets in T. We construct a proper edge editing set F to the (P_5, C_5)-free editing instance (G, k) as follows: For each $S_i \in C \setminus T$ F deletes all edges between S_i and X_i. These are $3 \cdot r$ deletions for each such S_i. So altogether we get $3(m - t) \cdot r$ deletions since there are $m - t$ such sets in $C \setminus T$. For each pair of vertices $s_i, s_j \in S$ F deletes the edge between them if s_i and s_j do not belong to the same set $T_p \in T$. Let G_F be the graph that results from applying F to G and let $G_F[S]$ be the induced subgraph of G_F with the vertices in S. $G_F[S]$ will consist of t triangles with each

[1] Actually [EMC88] uses a slightly confusing notation where he proves NP-completeness of P_{l+1} free editing for $l \geq 2$. So unlike in Section 1.1 of this thesis in [EMC88] l denotes a number that is the size of the forbidden subgraph minus one. Therefore if you want to compare our proof to [EMC88]s, note that our reduction actually equals [EMC88]s reduction for the case of $l = 4$.

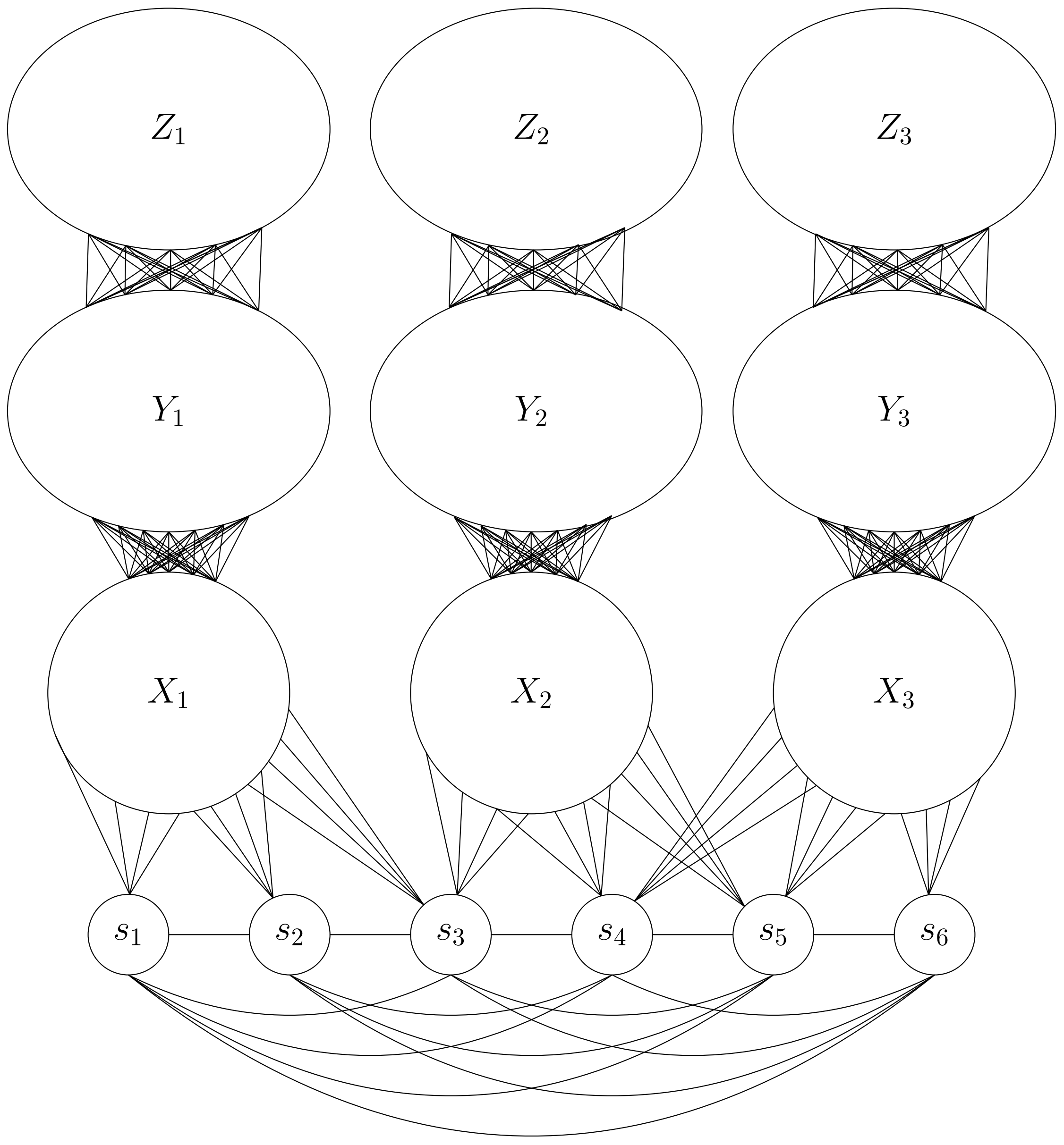


Figure 2.1.: Reduction for $S = \{s_1, \ldots, s_6\}$ and $C = \{S_1, S_2, S_3\}$ with $S_1 := \{s_1, s_2, s_3\}, S_2 := \{s_3, s_4, s_5\}$ and $S_3 := \{s_4, s_5, s_6\}$

three edges. So there will be $3t$ edges left between vertices of S, while there were initially r edges. Therefore we have $r - 3t$ edge deletions between vertices of S. Adding this together we get $|F| = 3(m-t) \cdot r + (r - 3 \cdot t) = k$.

Now let us prove that there is no P_5 and no C_5 in the resulting graph G_F: Note that each connected component in G_F consists either of three cliques X_i, Y_i, Z_i or four cliques S_i, X_i, Y_i, Z_i. And remember that for two cliques $K, K' \in \{S_i, X_i, Y_i, Z_i\}$ $(K \neq K')$ either each vertex in K is adjacent to each vertex in K' or there is absolutely no edge between K and K'. Now suppose in one such connected component there are five nodes v_1, v_2, v_3, v_4, v_5 that induce a P_5 or a C_5. Since there are less than five cliques, two of those vertices are in the same clique. Without loss of generality suppose $v_1, v_2 \in K$. Note that one of the other vertices v_3, v_4, v_5 is adjacent to v_1 or v_2. Otherwise the induced subgraph would not be connected and could therefore not be a P_5 or a C_5. Without loss of generality suppose v_3 is adjacent to v_1 or v_2. This is only possible if v_3 lies in a clique that is adjacent to K or it lies in K too. In both cases v_3 is not only adjacent to one of the vertices v_1, v_2 but to

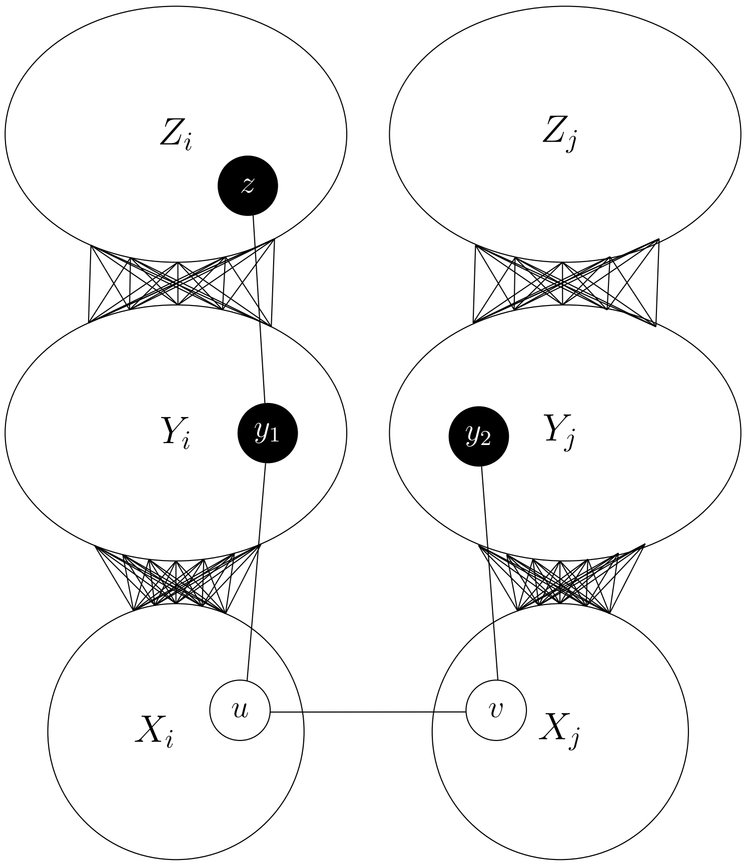


Figure 2.2.: Illustration of the proof of Lemma 2.3. Unaffected vertices are coloured black.

both of these vertices simultaneously. But v_1 and v_2 are also adjacent to each other, so v_1, v_2, v_3 induce a C_3. This is a contradiction since neither a P_5 nor a C_5 contain a C_3 as a subgraph.

$\Leftarrow$ Assume we have an edge editing set F for G such that $|F| \leq k$. Further assume that G_F is (P_5, C_5)-free. Remember G_F denotes the graph that we get by applying F to G. Let us show that in that case we can find a solution to our instance (S, C) of 3 cover. The idea is to show that the connected components in the subgraph $G_F[S]$ of G_F induced by S represent elements of C. To do so we need to prove some more helping lemmas first:

Let us call a vertex $v \in V$ *affected* if at least one of its incident edges was modified by F, that is added or deleted. Otherwise a vertex is called unaffected.

Lemma 2.2. *Each Y_i and each Z_i contain an unaffected vertex.*

Proof. Since each edge in F affects only two vertices, there are at most $2k$ affected vertices. But each clique Y_i and Z_i contains $3k > 2k$ vertices. □

Lemma 2.3. *G_F contains no edge from X_i to X_j. $(i \neq j)$*

Proof. Assume there is an edge $\{u, v\}$ in G_F with $u \in X_i$, $v \in X_j$. By Lemma 2.2 we can find unaffected vertices $z \in Z_i, y_1 \in Y_i$ and $y_2 \in Y_j$. Now z, y_1, u, v, y_2 induces a P_5 as depicted in Figure 2.2. But that is a contradiction, since we assumed that G_F is $(P_5, C_5)-$free. (Note that there can be no edges other than $\{u, v\}$ added or deleted by F between those five vertices, since it would contain at least one unaffected vertex.) □

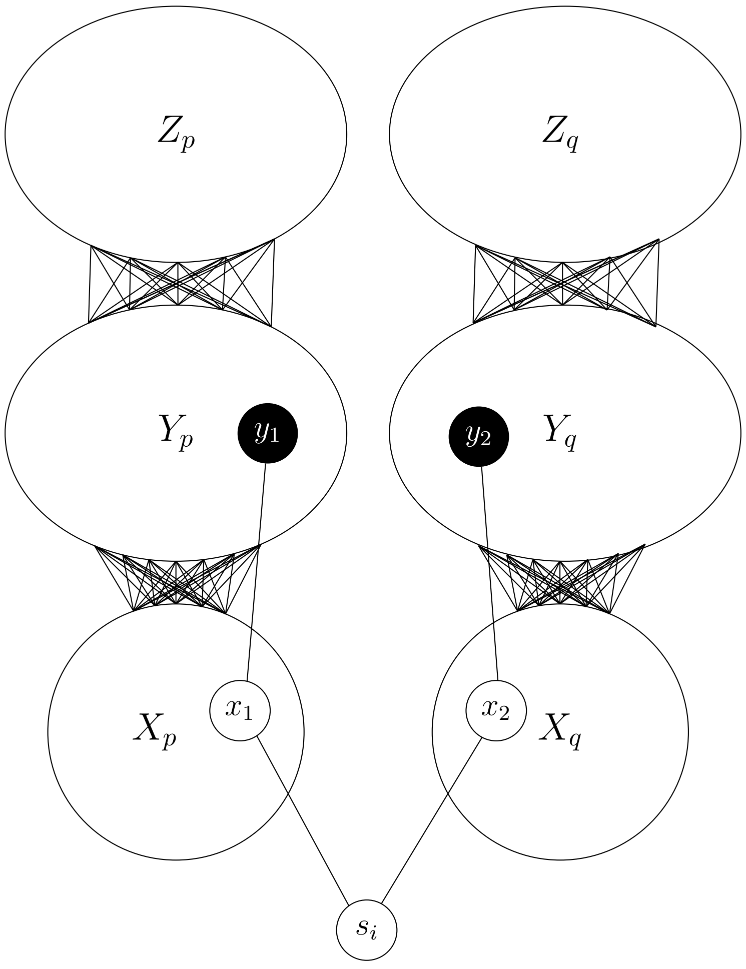


Figure 2.3.: Illustration of the proof of Lemma 2.4. Unaffected vertices are coloured black.

Lemma 2.4. *Every vertex s_i in S is adjacent to* at most one *X_j in G_F. (Let us say s_i is adjacent to X_j, if there is at least one x in X_j, such that s_i is adjacent to x)*

Proof. Assume there is an s_i that is adjacent to $x_1 \in X_p$ and $x_2 \in X_q$ simultaneously. $(p \neq q)$ By Lemma 2.2 we can find unaffected vertices $y_1 \in Y_p$ and $y_2 \in Y_q$. (See Figure 2.3) The edges (y_1, x_1) and (x_2, y_2) cannot be deleted by F since they contain an unaffected vertex. F cannot add the edge (x_1, x_2) by Lemma 2.3 and it cannot add any other edge that would destroy the P_5 induced by y_1, x_1, s_i, x_2, y_2, since it would contain an unaffected vertex. That is a contradiction since we assumed that G_F contains no P_5. □

Lemma 2.5. *Every vertex s_i in S is adjacent to* exactly one *X_j in G_F. And it was adjacent to that X_j in G already.*

Proof. There are $3rm$ edges between $\bigcup X_i$ and S in G. Because of Lemma 2.4 there may be at most $n \cdot r = 3tr$ of these edges left in G_F. So in order to fulfil Lemma 2.4 we need at least $3rm - 3tr = 3(m-t)r$ edge deletions between $\bigcup X_i$ and S. Assume that there is at least one $s \in S$ that has been completely separated from each X_i to which it was initially connected in G. In order to fulfil Lemma 2.4 and that assumption simultaneously r additional edge deletions are required. So the number of edge deletions $3(m-t)r + r$ would be greater than $k = 3(m-t) \cdot r + (r - 3t)$, which is a contradiction. □

Lemma 2.6. *If s_i is adjacent to X_p and s_j is adjacent to X_q in G_F for some $p \neq q$, then F must have deleted the edge $\{s_i, s_j\}$.*

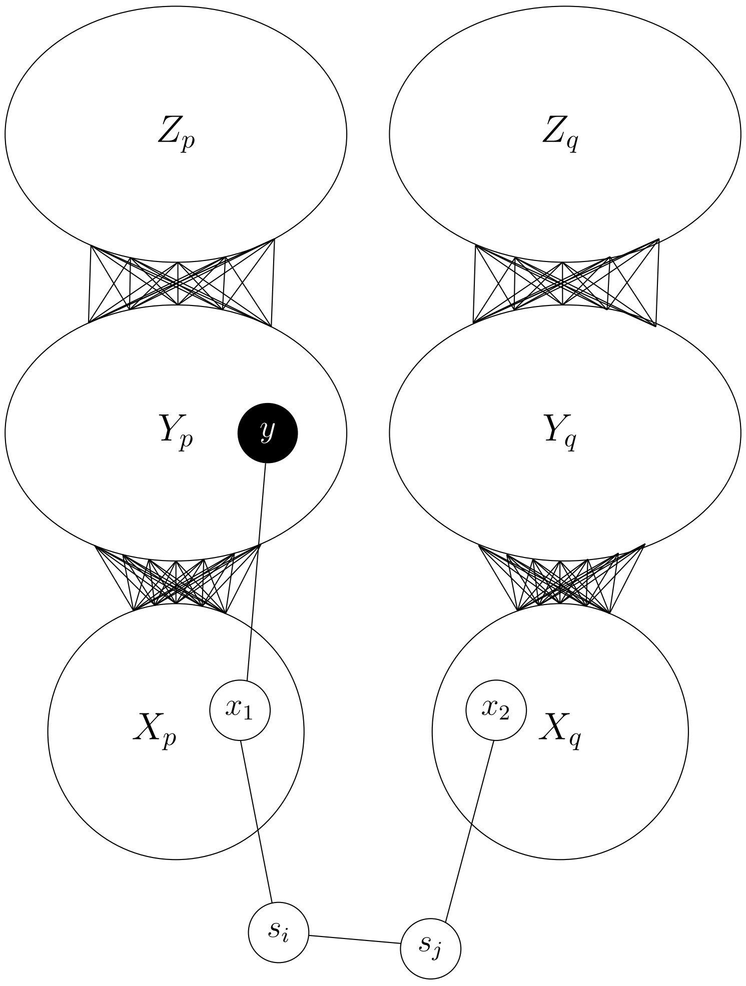


Figure 2.4.: Illustration of the proof of Lemma 2.6. Unaffected vertices are coloured black.

Proof. Let s_i, s_j be some vertices in S such that s_i is adjacent to some $x_1 \in X_p$ and s_j is adjacent to some $x_2 \in X_q$ ($p \neq q$). Now assume that F had not deleted the edge (s_i, s_j).

By Lemma 2.2 we find an unaffected vertex $y \in Y_p$. Let us show that F cannot destroy the P_5 induced by y, x_1, s_i, s_j, x_2, which is depicted in Figure 2.4. F cannot delete the edge (y, x_1) since y is an unaffected vertex. The edges (x_1, s_i), (s_i, s_j) and (s_j, x_2) exist in G_F by our initial assumption. F cannot add the edge (x_1, x_2) by Lemma 2.3, and it cannot add the edges (x_1, s_j) or (s_i, x_2) by Lemma 2.4. Any other edge addition that could destroy the P_5 would contain the unaffected vertex y, which is not possible. ☐

Lemma 2.7. *Each connected component in $G_F[S]$ is a subset of some $S_i \in C$. (Remember that $G_F[S]$ denotes the subgraph of G_F induced by S.)*

Proof. From Lemma 2.5 we know that each $s \in S$ is adjacent to one X_i set in G_F to which it was already connected in G. From Lemma 2.6 we know that the nodes of one connected component in $G_F[S]$ are all connected to the same X_i. Thus they belong to the same $S_i \in C$. ☐

Lemma 2.8. *The subgraph $G_F[S]$ of G_F is a disjoint union of cliques of size* exactly *three. (also called C_3) Therefore each connected component of $G_F[S]$ is not a proper subsets of some $S_i \in C$ but equal to some $S_i \in C$.*

Proof. From Lemma 2.7 we know that S consists of connected components of size *at most* three, as each S_i has size three.

As noted in the proof of Lemma 2.5 F contains at least $3(m-t)r$ edge deletions between $\bigcup X_i$ and S. So there may be at most $k - 3(m-t)r = r - 3t$ edge deletions in S.

The number of edges in a graph is the half of the sum of the degrees of all vertices appearing in the graph. If we assume that $G_F[S]$ consists of cliques of size 3, then all vertices would have degree 2. Therefore the number of edges left in S would be $\frac{3t \cdot 2}{2} = 3t$. Thus F would need to delete $r - 3t$ edges in S.

Now assume that $G_F[S]$ does not consist of cliques of size three only, but that there are also P_3s or cliques of size one or two. Note that in that case still all vertices in $G_F[S]$ had a degree ≤ 2 but now some vertices would have a degree < 2. So the sum of degrees is lower than in the previously discussed case. Therefore the number of edges remaining in S is smaller, too. Thus the number of deletions in S has to be even bigger than $r - 3t$. But we previously showed that this is not possible. So we have a contradiction. □

Lemma 2.9. *There is a solution to the instance (S, C) of the Exact 3-Cover problem (induced by the solution to the instance (G, k) of (P_5, C_5)-free editing).*

Proof. From 2.8 we know that each connected component in $G_F[S]$ corresponds to some $S_i \in C$. And obviously S is the disjoint union of these connected components/elements of C. □

This completes the proof of the equivalence of the instances (S, C) of Exact 3-Cover and (G, k) of (P_5, C_5)-free editing and therefore also completes the proof of Theorem 2.1.

□

2.2. Generalizing the NP-Completeness Proof

Theorem 2.1 can be generalized even more. The reductions used in the following generalization of Theorem 2.1 are still the same as in [EMC88].

Theorem 2.10. *Let $p \in \mathbb{N}_0$, $l \geq 4$ and $4 \leq \mu[1] < \cdots < \mu[p] \leq l$:*

The $(P_l, C_{\mu[1]}, \ldots, C_{\mu[p]})$-free editing problem is NP-complete.

Before we prove this theorem let us first make sure we understand what this theorem says:

First note that in the case of $p = 1$ we also refer to the $(P_l, C_{\mu[1]})$ free editing problem by $(P_l, C_{\mu[1]}, \ldots, C_{\mu[p]})$-free editing problem, although this notation suggests that there would be more than one forbidden induced subgraph. Furthermore in the case of $p = 0$ we also refer to the P_l-free editing problem by $(P_l, C_{\mu[1]}, \ldots, C_{\mu[p]})$-free editing problem. This might seem a bit confusing but it saves us tedious case distinctions.

Here are some examples of problems that are NP-complete according to Theorem 2.10:

For the case of $l := 5, p := 1, \mu[1] = 5$ we know that the (P_5, C_5)-free editing problem is NP-complete. So Theorem 2.10 is a generalization of Theorem 2.1.

For the case of $l := 5, p := 0$ we know that P_5-free editing is NP-complete. Remember that [EMC88] proved the NP-completeness only for the deletion version of this problem.

For the case of $l := 6, p := 1, \mu[1] := 6$ we know that (P_6, C_6)-free editing is NP-complete.

For the case of $l := 5, p := 2, \mu[1] := 4, \mu[2] := 5$ we know that (P_5, C_4, C_5)-free editing is NP-complete. This problem was also mentioned in [NG13] as another possible relaxation of the (P_4, C_4) freeness condition, which they focused on.

For the case of $l := 8, p := 3, \mu[1] := 4, \mu[2] := 6, \mu[3] := 7$ we know that (P_8, C_4, C_6, C_7)-free editing is NP-complete. This is just a random example invented only to illustrate Theorem 2.10.

Note that each P_L and each C_L with $L > l$ contains a P_l. Therefore it would not make any sense to use multiple paths of different size or circles of size bigger than the forbidden path as forbidden induced subgraphs. An interesting problem however that is not covered by Theorem 2.10 would be using only circles as forbidden induced subgraphs.

Proof. Again the containedness is clear. To show the hardness we use a reduction from Exact 3-Cover. Let (S, C) be an instance of Exact 3-Cover. The corresponding instance (G, k) of the $(P_l, C_{\mu[1]}, \ldots, C_{\mu[p]})$-free editing problem is constructed as follows: As in the proof of Theorem 2.1, we start with G containing the elements of S as vertices, such that S induces a clique in G. And again we define:

$$r := \binom{n}{2}$$
$$k := 3(m - t) \cdot r + (r - 3t)$$

For each $S_i \in S$ we add $l - 2$ (disjoint) cliques $Q_i[1], \ldots, Q_i[l-3], X_i$ to G such that $|Q[1]| = \cdots = |Q[l-3]| = 3k$ and $|X_i| = r$. And we add all possible edges between S_i and X_i, between X_i and $Q_i[l-3]$ and between $Q_i[j]$ and $Q_i[j-1]$ for $2 \leq j \leq l-3$ to G. For simplicity we may also sometimes denote X_i by $Q_i[l-2]$ and S_i by $Q_i[l-1]$. Note that for $l = 5$ the reduction is the same as in the proof of Theorem 2.1 except that Z_i is called $Q_i[1]$ and Y_i is called $Q_i[2]$.

Figure 2.5 depicts this reduction for a simple example instance of Exact 3-Cover .

Again this reduction is obviously computable in polynomial time. Now let us show that (S, C) is a yes-instance of Exact 3-Cover if and only if (G, k) is a yes-instance of $(P_l, C_{\mu[1]}, \ldots, C_{\mu[p]})$-free editing :

$\Rightarrow$ Assume there is a solution $T \subseteq C$ to the Exact 3-Cover instance (S, C). That is S is the disjoint union of the 3-sets in T. As a solution for the (G, k) instance of the $(P_l, C_{\mu[1]}, \ldots, C_{\mu[p]})$-free editing problem, we use the exact same editing set F as in the proof of Theorem 2.1. As we already showed there $|F| = k$. Let us show that G_F does not contain any of the forbidden induced subgraphs. Note that each connected component in G_F consists either of $l-2$ cliques $X_i, Q_i[l-3], \ldots, Q_i[1]$ or $l-1$ cliques $S_i, X_i, Q_i[l-3], \ldots, Q_i[1]$. For two cliques $K, K' \in \{S_i, X_i, Q_i[l-3], \ldots, Q_i[1]\}$ $(K \neq K')$ either each vertex in K is adjacent to each vertex in K' or there is absolutely no edge between K and K'. Assume there is a forbidden induced subgraph in such a connected component induced by some $V' \subseteq V(G)$. If there were $v_1, v_2 \in V'$ that belong to the same clique $(v_1, v_2 \in K \in \{S_i, X_i, Q_i[l-3], \ldots, Q_i[1]\})$, then this would lead to an induced C_3 in $G_F[V']$ as in the proof of proof of Theorem 2.1. Therefore there may be no two such vertices, since none of the forbidden induced subgraphs contains a C_3. It immediately follows that $G_F[V']$ is no P_l as there are only $l-1$ or $l-2$ cliques in each connected component but P_l has l vertices. Assume $G_F[V']$ is isomorphic to some $C_{\mu[q]}$. Let j_1 be the index of the outermost clique that contains a vertex from V', that is $j_1 := \min\{j \mid Q_i[j] \cap V' \neq \emptyset\}$. The vertex v that lies in $Q_i[j_1]$ has no neighbours in $Q_i[j_1 - 1]$ (or $Q_i[j_1 - 1]$ does not even exist if $j_1 = 1$). As there may be at most one vertex in one clique, there may be no neighbour of

v in $Q_i[j_1]$ and at most one neighbour in $Q_i[j_1+1]$. Thus v has degree one in the induced $C_{\mu[q]}$. This is a contradiction as $C_{\mu[q]}$ does not contain a degree one node.

$\Leftarrow$ Assume there is an editing set F with $|F| \le k$ such that G_F is $(P_l, C_{\mu[1]}, \ldots, C_{\mu[p]})$-free. Let us show that in this case there is a solution to the Exact 3-Cover -instance (S, C), that was reduced to (G, k). In order to do so we have to prove lemmas analogue to the Lemmas 2.2 to 2.8 in the proof of Theorem 2.1.

Lemma 2.11. *Each $Q_i[j]$ with $1 \le j \le l-3$ contains an unaffected vertex.*

Proof. Again we have $|Q_i[j]| = 3k > 2k$ as in the proof of Theorem 2.2. □

Lemma 2.12. *G_F contains no edge from X_i to X_j. ($i \ne j$)*

Proof. Assume there is an edge $\{u, v\}$ in G_F with $u \in X_i$, $v \in X_j$. By Lemma 2.11 we can find unaffected vertices $y_1 \in Q_i[1], \ldots, y_{l-3} \in Q_i[l-3]$ and $z \in Q_i[l-3]$. Now $y_1, \ldots, y_{l-3}, u, v, z$ induces a P_l. But that is a contradiction, since we assumed that G_F is $(P_l, C_{\mu[1]}, \ldots, C_{\mu[p]})$-free. (Note that there can be no edges other than $\{u, v\}$ added or deleted by F between those l vertices, since it would contain at least one unaffected vertex.) □

Lemma 2.13. *Every vertex s_i in S is adjacent to* at most one *X_j in G_F. (Let us say s_i is adjacent to X_j, if there is at least one x in X_j, such that s_i is adjacent to x)*

Proof. Assume there is an s_i that is adjacent to $x_1 \in X_a$ and $x_2 \in X_b$ simultaneously. ($a \ne b$) By Lemma 2.2 we can find unaffected vertices $y_1 \in Q_a[1], \ldots, y_{l-3} \in Q_a[l-3]$. The edges (y_q, y_{q+1}) for $1 \le q \le l-4$ and (y_{l-3}, x_1) cannot be deleted by F since they contain one or two unaffected vertices. F cannot add the edge (x_1, x_2) by Lemma 2.12 and it cannot add any other edge that would destroy the P_l induced by $y_1, \ldots, y_{l-3}, x_1, s_i, x_2$, since it would contain an unaffected vertex. That is a contradiction since we assumed that G_F contains no P_l. □

Lemma 2.14. *Every vertex s_i in S is adjacent to* exactly one *X_j in G_F. And it was adjacent to that X_j in G already.*

Proof. The proof is exactly the same as in Lemma 2.4. □

Lemma 2.15. *If s_i is adjacent to X_a and s_j is adjacent to X_b in G_F for some $a \ne b$, then F must have deleted the edge $\{s_i, s_j\}$.*

Proof. Let s_i, s_j be some vertices in S such that s_i is adjacent to some $x_1 \in X_a$ and s_j is adjacent to some $x_2 \in X_b$ ($a \ne b$). Now assume that F had not deleted the edge (s_i, s_j).

By Lemma 2.11 we find unaffected vertices $y_1 \in Q_a[1], \ldots, y_{l-3} \in Q_a[l-3]$. Let us show that F cannot destroy the P_l induced by $y_1, \ldots, y_{l-3}, x_1, s_i, s_j, x_2$:

F cannot delete the edges (y_q, y_{q+1}) for $1 \le q \le l-4$ and (y_{l-3}, x_1) since the y_q-s and y_{l-3} are unaffected vertices. The edges (x_1, s_i), (s_i, s_j) and (s_j, x_2) exist in G_F by our initial assumption. F cannot add the edge (x_1, x_2) by Lemma 2.12, and it cannot add the edges (x_1, s_j) or (s_i, x_2) by Lemma 2.13. Any other edge addition that could destroy the P_l would contain one of the unaffected vertices $y_1, \ldots, y_{l-3}$, which is not possible. □

Lemma 2.16. *Each connected component in $G_F[S]$ is a subset of some $S_i \in C$. (Remember that $G_F[S]$ denotes the subgraph of G_F induced by S.)*

Proof. This follows from Lemmas 2.14 and 2.15 as in the proof of 2.7. □

Lemma 2.17. *The subgraph $G_F[S]$ of G_F is a disjoint union of cliques of size* exactly *three. Therefore each connected component of $G_F[S]$ is not a proper subsets of some $S_i \in C$ but equal to some $S_i \in C$.*

Proof. The proof is the same as for Lemma 2.8. □

Lemma 2.18. *There is a solution to the instance (S, C) of the Exact 3-Cover problem (induced by the solution to the instance (G, k) of $(P_l, C_{\mu[1]}, \dots, C_{\mu[p]})$-free editing).*

Proof. See the proof of Lemma 2.9. □

This completes the proof of the equivalence of the instances (S, C) of Exact 3-Cover and (G, k) of $(P_l, C_{\mu[1]}, \dots, C_{\mu[p]})$-free editing and therefore also completes the proof of Theorem 2.10.

□

2.3. NP-Completeness of the Deletion Version

Here we show that the deletion versions of the problems discussed in the previous section are NP-complete, too. Especially this includes the NP-completeness of the (P_5, C_5)-free deletion problem.

Theorem 2.19. *Let $p \in \mathbb{N}_0$, $l \geq 4$ and $4 \leq, \mu[1] < \dots < \mu[p] \leq l$: The $(P_l, C_{\mu[1]}, \dots, C_{\mu[p]})$-free deletion problem is NP-complete.*

Note that in the case of $p = 0$ this theorem was already proved in [EMC88].

Proof. The containedness is clear.

We show that the exact same reduction that was used in the proof of Theorem 2.10 is also a correct reduction to prove the hardness of the deletion problem.

Let (S, C) be some instance of Exact 3-Cover and let (G, k) be the corresponding instance of $(P_l, C_{\mu[1]}, \dots, C_{\mu[p]})$-free editing / deletion.

We show that (S, C) is a yes instance of Exact 3-Cover if and only if (G, k) is a yes-instance of $(P_l, C_{\mu[1]}, \dots, C_{\mu[p]})$-free deletion:

$\Rightarrow$ Assume (S, C) has a solution $T \subseteq C$. Note that the editing set F that we constructed from T in the proofs of Theorems 2.1 and 2.10 does not contain any edge insertions. So F is also a correct solution for the deletion problem.

$\Leftarrow$ Assume (G, k) is a yes-instance of the $(P_l, C_{\mu[1]}, \dots, C_{\mu[p]})$-free deletion problem. That is there is a deletion set $F \subseteq E(G)$ with $|F| \leq k$ such that G_F is $(P_l, C_{\mu[1]}, \dots, C_{\mu[p]})$-free. Note that a correct edge deletion set is also a correct editing set. Therefore (G, k) is also a yes-instance of the $(P_l, C_{\mu[1]}, \dots, C_{\mu[p]})$-free editing problem. But in that case we have already proved in the proof of Theorem 2.10 that (S, C) is a yes-instance of Exact 3-Cover.

□

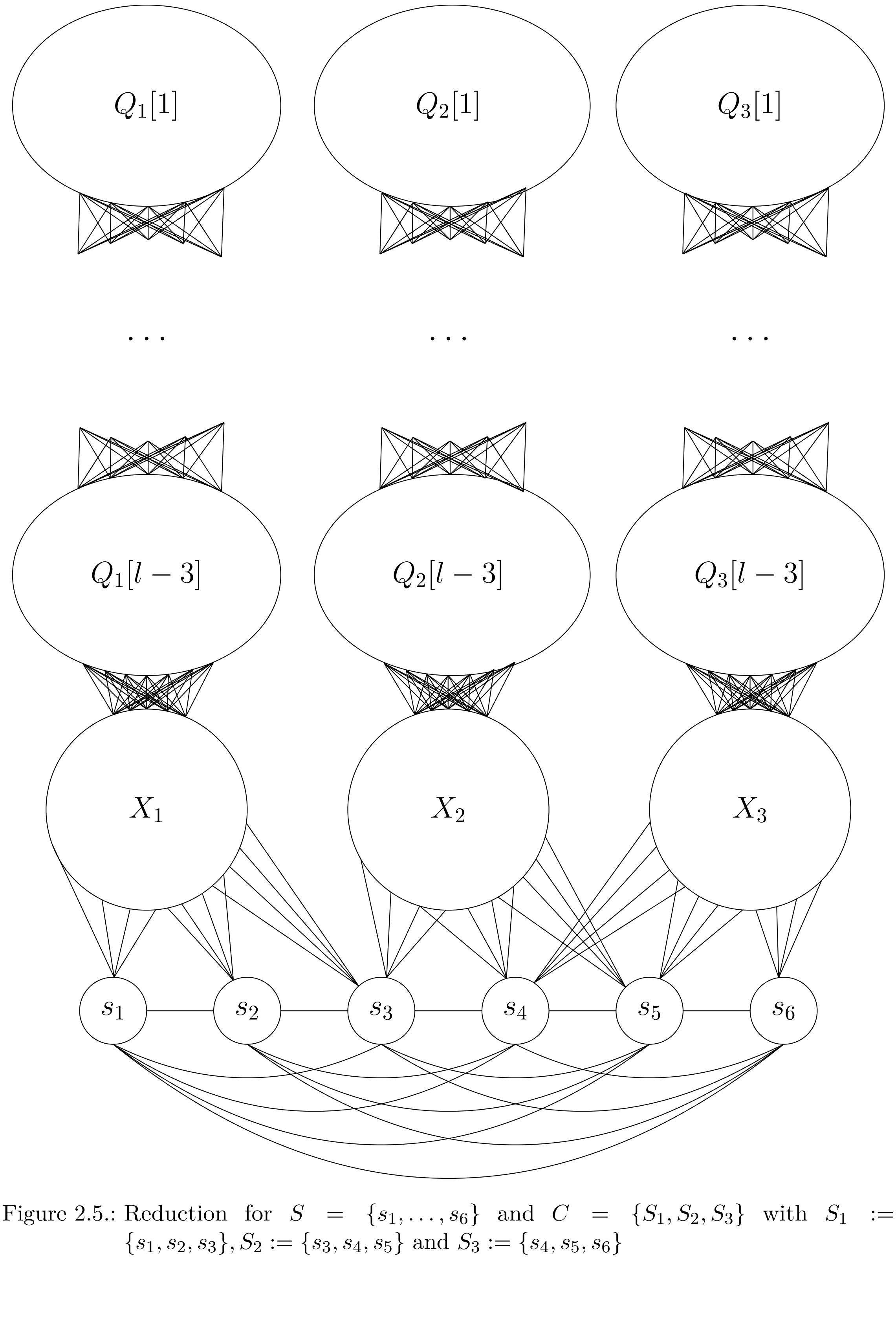


Figure 2.5.: Reduction for $S = \{s_1, \ldots, s_6\}$ and $C = \{S_1, S_2, S_3\}$ with $S_1 := \{s_1, s_2, s_3\}, S_2 := \{s_3, s_4, s_5\}$ and $S_3 := \{s_4, s_5, s_6\}$

3. Algorithms

In this chapter we propose algorithms that can be used in order to calculate close (P_5, C_5)-free graphs. We also discuss the correctness and theoretical bounds of the running time here.

In the subsequent Chapter 4 we further evaluate the running time and - in the case of the heuristic algorithms - also the quality of the results of these algorithms by practical case studies.

3.1. Graph Representation

In order to prove the claimed running time of the algorithms that are presented in the following sections, it is necessary to assume that we can check whether two given vertices $u, v \in V(G)$ of some graph G are adjacent in $O(1)$ time. To do so we need an adjacency matrix representation of the graph.

On the other hand however we need to assume that we can iterate over the neighbours of some given vertex v in $O(d(v))$ time. That is as the body of a loop of the form of Algorithm 3.1 is executed $d(v)$ times anyway in the asymptotic notation the time needed to calculate $N(v)$ can be neglected. However for that second assumption we need an adjacency list representation of the graph. Therefore when dealing with some graph we want to assume that we simultaneously have an adjacency matrix representation and an adjacency list representation of the graph in the memory.

To prove the running time of the editing algorithms it is convenient to assume that we can add and delete an edge $\{u, v\}$ in time $O(1)$.

It is obviously possible to update the adjacency matrix in $O(1)$ time when adding or deleting an edge. However we need to keep the adjacency list representation up to date as well. Adding an edge $\{u, v\}$ is no problem. We can add the vertex u to the head of the adjacency list of v and add the vertex v to the head of the adjacency list of u in $O(1)$ time. However when deleting an edge $\{u, v\}$ we first have to find the edge u in the adjacency

Algorithm 3.1: NEIGHBOURHOOD LOOP

```
1 forall $u \in N(v)$ do
2     do something with $u$
```

list of v and find the edge v in the adjacency list of u. To do so we may need $O(d(v))$ and $O(d(u))$ time. As a solution to this problem we can use a special adjacency matrix $A = (a_{i,j})$. Let $i[u]$ and $i[v]$ denote the indices of some vertices u and v. For a given edge $\{u, v\}$ we can not only save the boolean value true in the locations $a_{i[u],i[v]}$ and $a_{i[v],i[u]}$. Instead we can additionally save the location of the vertex v in the adjacency list of u in $a_{i[u],i[v]}$ and save the location of the vertex u in the adjacency list of v in $a_{i[v],i[u]}$. With the help this kind of adjacency matrix we can delete edges in $O(1)$ time.

So in the following sections of this chapter we assume that the following operations are feasible in $O(1)$ time:

- checking whether two vertices $u, v \in V(G)$ are adjacent
- adding an edge $\{u, v\}$
- deleting an edge $\{u, v\}$
- finding the next neighbour u of some vertex v before one execution of the body of a loop that iterates over $N(v)$ as in Algorithm 3.1

3.2. P5, C5 Count

In this section we introduce an algorithm that counts the total number of P_5s and C_5s in a graph G in time $O(m \cdot d^3) \subseteq O(m \cdot d^2 + \#P5 + \#C5)$. Here d denotes the maximal vertex degree in G and $\#P5$ and $\#C5$ denote the actual number of P_5s and C_5s in G.

Such an algorithm is needed in Section 3.5 to implement a greedy heuristic for the (P_5, C_5)-free editing search problem. The counting algorithm CountP5C5 that we introduce in this section can also easily be modified to a search algorithm. This search algorithm is needed for the exact (P_5, C_5)-free editing algorithm in Section 3.4. The search algorithm needs only $O(m \cdot d^2)$ time.

The algorithm CountP5C5 uses a subroutine CountP5C5Local(G,v). The subroutine CountP5C5Local(G,v) searches the neighbourhood $N(v)$ of v and also the neighbours of the neighbours of v for vertices that induce a P_5 or a C_5 together with v. One call to this subroutine does not find all P_5s and C_5s in G. It only counts exactly the P_5s that contain v as a middle vertex and the C_5s that contain v at any place. Therefore CountP5C5 has to call this subroutine multiple times. One time for each vertex $v \in V(G)$. CountP5C5 then sums up the number of vertices found in each of these calls to the subroutine.

A call to CountP5C5Local(G, v) iterates over quadruples (v_1, v_2, v_4, v_5) such that these vertices together with v induce either a P_5 or a C_5 and the vertices appear in the same order as in Figure 3.1 on this P_5 or C_5. To do so, CountP5C5Local iterates over pairs of neighbours $v_2, v_4 \in N(v)$ of v such that v_2, v, v_4 induce a P_3 with v as the middle vertex. For each of these pairs (v_2, v_4) it calculates the set $V_1(v_2, v_4) \subseteq N(v_2)$ of vertices that could take the place of v_1 in Figure 3.1. It also calculates a set $V_5(v_2, v_4) \subseteq N(v_3)$ of vertices that could take the place of v_5 in Figure 3.1. For each pair v_1, v_5 of vertices $v_1 \in V_1(v_2, v_4)$ and $v_5 \in V_5(v_2, v_4)$ it checks whether they are adjacent or not. That is it checks whether the found subgraph v_1, v_2, v, v_4, v_5 is a P_5 or a C_5. It then increments the according counter. Note that a more naive version of CountP5C5Local might recalculate $V_5(v_1, v_2, v_4)$ for each found candidate of v_1. Our version of CountP5C5Local may be faster as it recalculates $V_5(v_2, v_4)$ only one time for each pair v_2, v_4.

A more formal pseudocode description of CountP5C5 and its subroutine is given in Algorithm 3.3 and Algorithm 3.2.

Let us now prove the correctness and the running time of CountP5C5:

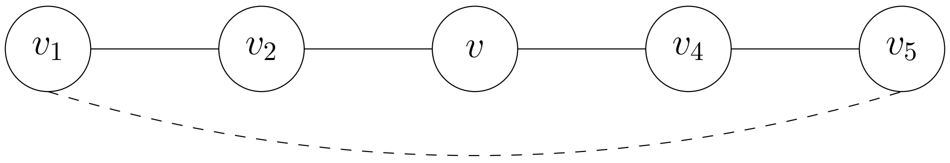


Figure 3.1.: Subgraph found by CountP5C5Local(G, v)

Algorithm 3.2: CountP5C5Local

```
Input: Graph $G = (V, E)$, node $v \in V$
Output: number of $P_5$s in $G$ containing $v$ as their middle vertex, number of $C_5$s in
        $G$ containing $v$ at any place.
1  P5count ← 0
2  C5count ← 0
3  forall $\{v_2, v_4\} \in N(v)$ with $v_2 \neq v_4$ do
4      if $v_2$ is not adjacent to $v_4$ then
           // v2, v, v4 = p3
5          $V_1 \leftarrow \emptyset$
6          forall $v_1 \in N(v_2)$ do
7              if $v_1$ is not adjacent to $v$ or $v_4$ then
                   // v1, v2, v, v4 = p4
8                  $V_1 \leftarrow V_1 \cup \{v_1\}$
9          $V_5 \leftarrow \emptyset$
10         forall $v_5 \in N(v_4)$ do
11             if $v_5$ is not adjacent to $v$ or $v_2$ then
                   // v2, v, v4, v5 = p4
12                 $V_5 \leftarrow V_5 \cup \{v_5\}$
13         forall pairs $(v_1, v_5)$ where $v_1 \in V_1, v_5 \in V_5$ do
14             if $v_5$ is adjacent to $v_1$ then
15                 C5count ← C5count + 1
16             else
17                 P5count ← P5count + 1
18 return P5count, C5count
```

Theorem 3.1. *CountP5C5Local is correct. That is, for a graph G and a vertex $v \in V(G)$, CountP5C5Local(G, v) returns the number of induced P_5s in G that contain v as the* middle *vertex. Furthermore CountP5C5Local(G, v) returns the number of induced C_5s in G that contain the vertex v at* any *place.*

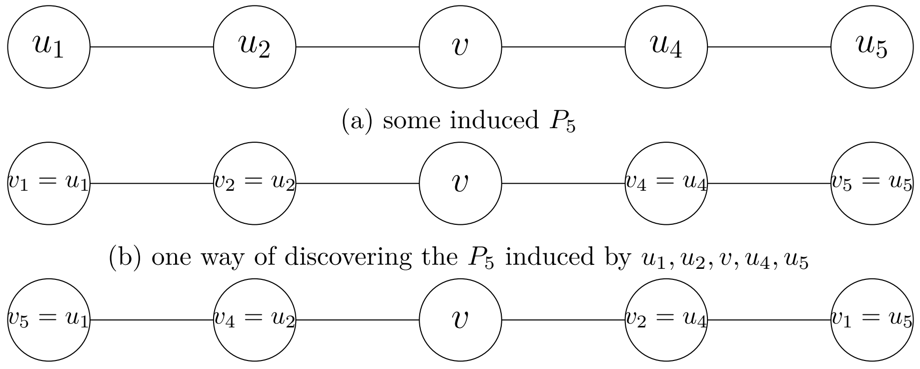
Proof. Let $\#P_5$ be the actual number of induced P_5s containing the vertex v as the "middle"-vertex and let $P5count$ be the number counted by the algorithm.

1) $P5count \geq \#P_5$:

We have to show that line 17 is executed at least once for each induced P_5 that contains v as "middle" vertex. That is we do not forget to count any of the P_5s. Let $G[V']$ be some of these P_5s induced by some subset $V' := \{u_1, u_2, v, u_4, u_5\}$, where v is the middle vertex. Without loss of generality assume the vertices u_i in $G[V']$ are enumerated as in Figure 3.2a. The body of the outermost loop in CountP5C5Local will be executed once for either $v_2 = u_2$ and $v_4 = u_4$ or for $v_2 = u_4$ and $v_4 = u_2$.

Algorithm 3.3: CountP5C5

```
    Input: Graph $G = (V, E)$
    Output: number of $P_5$s in $G$, number of $C_5$s in $G$
 1  P5count ← 0
 2  C5count ← 0
 3  P5countSum ← 0
 4  C5countSum ← 0
 5  forall $v \in V$ do
 6      P5count, C5count ← CountP5C5Local($G$, $v$)
 7      P5countSum ← P5countSum + P5count
 8      C5countSum ← C5countSum + C5count
 9  numberOfP5s ← P5countSum
10  numberOfC5s ← C5countSum/5
11  return numberOfP5s, numberOfC5s
```



(a) some induced P_5

(b) one way of discovering the P_5 induced by u_1, u_2, v, u_4, u_5

(c) another way of discovering the P_5 induced by u_1, u_2, v, u_4, u_5

Figure 3.2.: two different ways of discovering a P_5

Let us consider the former case first: As $v_2 = u_2$ is not adjacent to $v_4 = u_4$ the body of the if clause containing the three inner loops will be executed. The body of the first inner loop will some time be executed with $v_1 = u_1$ as u_1 is adjacent to $v_2 = u_2$. Note that the condition of the if clause will be fulfilled as $v_2 = u_2$ is not adjacent to v or $v_4 = u_4$. So u_1 will be added to V_1. Similarly in the second inner loop u_5 will be added to V_5.

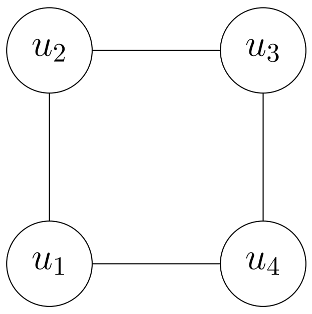
Since $u_1 \in V_1$ and $u_5 \in V_5$ the body of the third inner loop will be executed with $v_1 = u_1$ and $v_5 = u_5$. As $v_1 = u_1$ is not adjacent to $v_5 = u_5$ line 17 will be executed, with $v_1 = u_1, v_2 = u_2, v, v_4 = u_4, v_5 = u_5$ as in Figure 3.2b.

Now assume that instead the outer loop had chosen $v_2 = u_4$ and $v_4 = u_2$. Then in a similar way we can show that line 17 will be executed with $v_1 = u_5, v_2 = u_4, v_4 = u_2, v_5 = u_1$ as in Figure 3.2c.

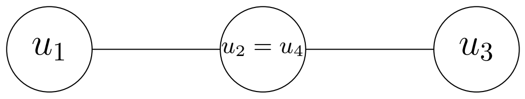
In any case line 17 is executed with $\{v_1, v_2, v, v_4, v_5\} = \{u_1, u_2, v, u_4, u_5\}$. Therefore for each induced P_5 (with “middle” vertex v) there is at least on execution of line 17. Thus $P5count \geq \#P_5$.

2) $P5count \leq \#P_5$:
We have to show two things. First that we do not count any subgraph that is not a P_5. And second that we do not count any subgraph twice.

2.1) Assume v_1, v_2, v, v_4, v_5 are the nodes of some execution of line 17. We must show that $V' := \{v_1, v_2, v, v_4, v_5\}$ actually induces a P_5. From the header of the outer loop we know



(a) C_4



(b) or P_3 ?

Figure 3.3.: How a badly designed C_4 counting algorithm might mistake a P_3 for a C_4.

that v_2 and v_4 are adjacent to v. By the header of the third inner loop we know that $v_1 \in V_1$ and $v_5 \in V_5$. By the construction of V_1 we know that v_1 must have been added to V_1 in line 8 in the first inner loop. By the header of this first inner loop we know that v_1 is adjacent to v_2. Similarly v_5 must have been added to V_5 in line 12 in the second inner loop. By the header of this second loop we know that v_5 is adjacent to v_4.

Let us now show that $G[V']$ contains no other edges except the mentioned edges $\{v_1, v_2\}$, $\{v_2, v\}$, $\{v, v_4\}$, $\{v_4, v_5\}$. By the if clause inside the first inner loop we know that v_1 is not adjacent to v or v_4. Similarly by the if clause inside the second loop we know that v_5 is not adjacent to v_2 and v. By the if clause starting in line 4 we know that v_2 is not adjacent to v_4. As the condition of the if clause starting in line 14 failed, we know that v_1 is not adjacent to v_5.

Note that these are all possible pairs of edges that must not be adjacent.

Now the edges are correct, but we also want to show that none of the vertices v_1, v_2, v, v_4, v_5 are identical. As an example why this is important consider a badly designed C_4 search algorithm that finds vertices u_1, u_2, u_3, u_4, where the edges $\{u_1, u_2\}, \{u_2, u_3\}, \{u_3, u_4\}, \{u_4, u_1\}$ exist. But u_1 is not adjacent to u_3 and u_2 is not adjacent to u_4. The algorithm might think this is a C_4 as depicted in Figure 3.3a but it might actually be a P_3 as in Figure 3.3b.

From line three we know that $v_2 \neq v_4$. In this bachelor thesis we only consider simple graphs. That is there may be no selfadjacent vertices. Thus $v_1 \neq v_2$, $v_2 \neq v$, $v \neq v_4$ and $v_4 \neq v_5$. Since $v_1 \notin N(v_4)$ but $v \in N(v_4)$, we have $v_1 \neq v$. Similarly we can use the following implications:

- $v_1 \in N(v_2)$ but $v_4 \notin N(v_2) \quad \Rightarrow \quad v_1 \neq v_4$
- $v_1 \in N(v_2)$ but $v_5 \notin N(v_2) \quad \Rightarrow \quad v_1 \neq v_5$
- $v_2 \in N(v_1)$ but $v_5 \notin N(v_1) \quad \Rightarrow \quad v_2 \neq v_5$
- $v \in N(v_2)$ but $v_5 \notin N(v_2) \quad \Rightarrow \quad v \neq v_5$

Now we know that v_1, v_2, v, v_4, v_5 induce a P_5 as in Figure 3.1. (without the dashed line)

2.2) Let $v_1^{(i)}, v_2^{(i)}, v_4^{(i)}, v_5^{(i)}$ be the values represented by the corresponding variables v_1, v_2, v_4, v_5 at the time when the algorithm hits line 17 for the i-th time. Let $V^{(i)} := \{v_1^{(i)}, v_2^{(i)}, v_4^{(i)}, v_5^{(i)}\}$ and $V^{(j)} := \{v_1^{(j)}, v_2^{(j)}, v_4^{(j)}, v_5^{(j)}\}$ for some $i \neq j$. We have to show $V^{(i)} \neq V^{(j)}$. We consider two cases:

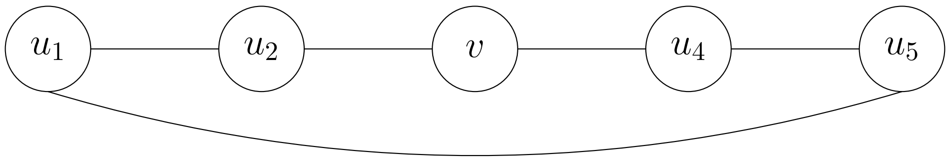


Figure 3.4.: some induced C_5

Case 1: i and j belong to the same execution of the outer loop: In this case we have $v_2^{(i)} = v_2^{(j)}$ and $v_4^{(i)} = v_4^{(j)}$. As $i \neq j$ we must be in different executions of the third inner loop. Thus $v_1^{(i)} \neq v_1^{(j)}$ and/or $v_5^{(i)} \neq v_5^{(j)}$. We assume the former, i.e. $v_1^{(i)} \neq v_1^{(j)}$, the other case works analogue. As $v_2^{(j)}, v_4^{(j)} \in N(v)$ but $v_1^{(i)} \notin N(v)$ we have $v_1^{(i)} \neq v_2^{(j)}, v_4^{(j)}$. As $v_5^{(j)} \in N(v_4^{(j)}) = N(v_4^{(i)})$ but $v_1^{(i)} \notin N(v_4^{(i)})$ we have $v_1^{(i)} \neq v_5^{(j)}$. Thus $v_1^{(i)} \notin V^{(j)}$ and $V^{(i)} \neq V^{(j)}$.

Case 2: i and j belong to different executions of the outer loop: In this case we have $\{v_2^{(i)}, v_4^{(i)}\} \neq \{v_2^{(j)}, v_4^{(j)}\}$. Thus $v_2^{(i)} \notin \{v_2^{(j)}, v_4^{(j)}\}$ and/or $v_4^{(i)} \notin \{v_2^{(j)}, v_4^{(j)}\}$. We assume the former case, the latter works analogue. As $v_2^{(i)} \in N(v)$ but $v_1^{(j)}, v_5^{(j)} \notin N(v)$ we have $v_2^{(i)} \notin \{v_1^{(j)}, v_5^{(j)}\}$. Thus $v_2^{(i)} \notin V^{(j)}$ and $V^{(i)} \neq V^{(j)}$.

Now we have shown that $p5count = \#P_5$. Showing that the number $\#C_5$ of induced C_5s in G is equal to the corresponding number $c5count$ that was returned by the algorithm, works similarly.

1) $C5count \geq \#C_5$:

We have to show that line 15 is executed at least once for each induced C_5 that contains v at any place. Let $G[V']$ be some of these P_5s induced by some subset $V' := \{u_1, u_2, v, u_4, u_5\}$. Without loss of generality assume that the vertices are enumerated as in Figure 3.4. Similar to the P_5 case we can show that the body of the third inner loop will at some point be executed with either $v_1 = u_1, v_2 = u_2, v, v_4 = u_4, v_5 = u_5$ or $v_1 = u_5, v_2 = u_4, v_4 = u_2, v_5 = u_1$. As u_1 is adjacent to u_5 the condition of the if clause in the third inner loop is fulfilled, so line 15 will be executed.

2) $C5count \leq \#C_5$:

Again we have to show two things. First that we do not count any subgraph that is not a C_5. And second that we do not count any subgraph twice.

2.1) Assume v_1, v_2, v, v_4, v_5 are the nodes of some execution of line 15. We have to show that these vertices induce a C_5 as in Figure 3.1 (including the dashed line). As the condition in the if-clause succeeded we know that v_1 is adjacent to v_5. The proof of the existence of the other needed edges, of the non-existence of edges that are not present in Figure 3.1 and of the inequalities between the vertices work exactly as in the P_5 case.

2.2) Showing that no C_5 is counted twice works exactly the same way as in the P_5 case.

$\square$

Theorem 3.2. *CountP5C5 is correct. That is, for a graph G, CountP5C5(G) returns the total number of induced P_5s and the total number of induced C_5s in G.*

Proof. As a P_5 has only one "middle" vertex v_3 it will be counted only one time when CountP5C5Local(G, v) is called in line 6 with $v := v_3$. As a C_5 contains five vertices v_1, v_2, v_3, v_4, v_5 it will be counted five times, that is one time for each of the calls to CountP5C5Local(G, v) in which $v \in \{v_1, v_2, v_3, v_4, v_5\}$. Therefore after the execution of the loop in CountP5C5 we have $P5CountSum = \#P5$ and $C5CountSum = \#C5 \cdot 5$. Here $\#P5$ and $\#C5$ denote the actual numbers of P_5s and C_5s in the graph. We get the correct number of $C5$s in line 11 when we divide by 5. □

Theorem 3.3. *Let $t(n, m, d, \#P_5, \#C_5)$ be the maximum running time that is needed to execute P5C5Count(G) on a graph $G = (V, E)$ with $|V| = n$ maximum degree $d := \max_{v \in V} d(v)$ and in which the actual number of P_5s and C_5s is $\#P_5$ and $\#C_5$.*
Then:

$$t \in O(m \cdot d^2 + \#P5 + \#C5) \subseteq O(m \cdot d^3)$$

Proof. Obviously the majority of the running time is the time spent in the calls to the subroutine CountP5C5Local. Note that considering the discussion in Section 3.1 we can assume that one single execution of one line of code takes always only $O(1)$ time. Let t_s be the number of times that the subroutine CountP5C5Local is called during one execution of CountP5C5. Let t_o be the total number of times that the body of the outer loop is executed, summed up over all calls to the subroutine. Let t_{i1} (t_{i2}, t_{i3}) be the total number of times that the body of the first (second, third) inner loop is executed, summed up over all calls to the subroutine.

Obviously we have

$$t \in O(t_s + t_o + t_{i1} + t_{i2} + t_{i3})$$

as the lines of code outside of the outer loop of CountP5C5Local are executed t_s times, the ones inside the outer but outside of the inner loop are executed t_o and the lines of code in the inner loops are executed t_{i1}, t_{i2} or t_{i3} times.

We have

$$\begin{aligned} t_s &= n \\ t_o &= \sum_{v \in V} \binom{d(v)}{2} = \sum_{v \in V} \frac{d(v)(d(v)-1)}{2} \\ &\leq \sum_{v \in V} \frac{d(v) d}{2} = \left(\frac{1}{2} \sum_{v \in V} d(v)\right) \cdot d \\ &= m \cdot d \end{aligned}$$

In each of the t_o executions of the outer loop, the first and second loops are executed $d(v_2) \leq d$ and $d(v_4) \leq d$ times. Therefore

$$t_{i1}, t_{i2} \leq t_o \cdot d \in O(m \cdot d^2)$$

Note that the sum of the numbers of P5count and C5count returned by the subroutine in line 6 of the main routine is equal to the number of executions of the third inner loop of the subroutine. And these values are summed up over all executions of the subroutine. So in the end we have $t_{i3} = p5countSum + c5CountSum$. From the correctness proof of the algorithm we know that $p5countSum = \#P5$ and $c5CountSum = 5 \cdot \#C5$. Therefore

$$t_{i3} = \#P5 + 5 \cdot \#C5 \in O(\#P5 + \#C5)$$

On the other hand we know that $|V_1| \leq d(v_2) \leq d$ and $|V_5| \leq d(v_4) \leq d$. And in each of the t_o executions of the outer loop the last inner loop is executed $|V_1| \cdot |V_2| \leq d^2$ times. Therefore

$$t_{i3} \leq t_o \cdot d^2 = m \cdot d^3$$

Note that first upper bound is asymptotically at least as sharp as the second one since

$$\#P5 + 5 \cdot \#C5 = t_{i3} \leq m \cdot d^3$$

□

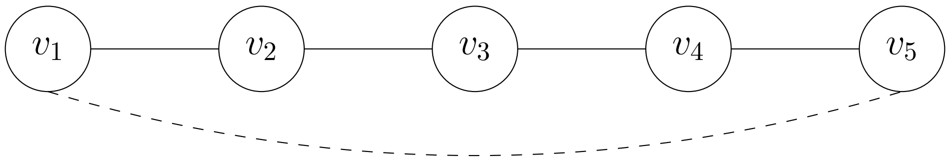


Figure 3.5.: result of FindP5C5(G) (if there is at least one induced P_5 or C_5 in G)

3.3. P5C5Search

We can modify the P5C5Count algorithm in order to return a set of vertices that induce a P5 or a C5.

Definition 3.4. *Let FindP5C5(G) be a subroutine that takes a given graph $G = (V, E)$ as input and does the following*

- *if G contains at least one induced P_5 and/or C_5, then FindP5C5(G) returns the boolean value true and five vertices $v_1, v_2, v_3, v_4, v_5 \in V$ that induce either a P_5 or a C_5. And the vertices shall be enumerated as in Figure 3.5.*
- *if G contains no induced P_5 and no induced C_5, FindP5C5(G) returns false.*

Theorem 3.5. *FindP5C5 can be implemented in time $O(m \cdot d^2)$*

Proof. We implement FindP5C5(G) such that it does the following:

- start executing the algorithm CountP5C5 on the input graph G
- when the subroutine CountP5C5Local hits line 15 or 17 for the first time then
  - abort the execution of both the subroutine CountP5C5Local and also of the CountP5C5 algorithm.
  - return the value true, and the set V' consisting of the vertices that were represented by the variables v_1, v_2, v, v_4, v_5 of the algorithm CountP5C5Local at the time when it hit line 15 or 17.
- if CountP5C5 finishes without ever hitting lines 15 or 17 of the subroutine CountP5C5Local then return false

We have shown in the proof of Theorem 3.2 that when line 15 or 17 of the subroutine is hit, then the vertices represented by the variables v_1, v_2, v, v_4, v_5 actually induce a P_5 or a C_5 as depicted in Figure 3.1. And we have also shown that if there is an induced P_5 or C_5, then it will be found. So if line 15 or 17 is never reached, then there is actually no P_5 or C_5 in G. Thus we know that this implementation of FindP5C5 returns the correct result.

The proof of the running time works the same way as in the proof of Theorem 3.3. Again the running time is in $O(t_s + t_o + t_{i1} + t_{i2} + t_{i3})$ where $t_s, t_o, t_{i1}, t_{i2}, t_{i3}$ are defined as in the proof of Theorem 3.3. However there is one important difference here: As we abort the calculation as soon as the body of the third inner loop of CountP5C5Local is hit the first time we have

$$t_{i3} \in O(1)$$

□

3.4. Exact Algorithm

In this section we introduce an editing algorithm EditToP5C5Free that solves the (P_5, C_5)-free editing decision problem. And as an extra bonus if the given graph G can be edited to a (P_5, C_5)-free graph in less than or equal to k steps then EditToP5C5Free(G,k) will also calculate such a close (P_5, C_5)-free graph G_F. However it is only guaranteed that $|F| \leq k$ but not necessarily $|F| = min!$ as required by the (P_5, C_5)-free editing search problem. To solve the search problem we can call EditToP5C5Free multiple times with increasing values of k until it finds a solution for the first time.

In Theorem 3.9 below we prove that our algorithm needs $O(9^k \cdot m \cdot n^2)$ time. Therefore while the (P_5, C_5)-free editing problem is NP-complete as shown in Chapter 2, it is at least fixed parameter tractable. Generally [Cai96] shows that if a hereditary graph property Π can be characterized by a finite set of forbidden subgraphs, then the Π-editing problem is fixed parameter tractable.

The algorithm proposed by [Cai96] is described in Algorithm 3.4. The only difference is that [Cai96] describes the algorithm for the general case of an arbitrary set of forbidden subgraphs and not just P_5s and C_5s. Also [Cai96] describes the algorithm more informally without using pseudocode.

We use an optimized version of Algorithm 3.4 which is described in 3.5.

Let us first try to understand the original version Algorithm 3.4:

The idea is that if the graph G contains a forbidden induced subgraph $G[V']$ induced by some $V' \subseteq V$, then a proper editing set F will always contain at least one edge addition or deletion in this subgraph $G[V']$. Therefore we iterate over all edges $e := \{u, v\}$ of the forbidden induced subgraph found in line 1. See lines 7 - 12. For each edge we check whether there is a editing set F with $e \in F$ such that G_F is (P_5, C_5)-free and $|F| \leq k$.

To check whether there is such a solution F containing the edge e, we first edit the edge e. That is we delete it if it exists and we add the edge e to G if the vertices u and v were not adjacent in G before. Then we recursively try to find at most $k - 1$ further edits to edit the resulting graph to a (P_5, C_5)-free graph.

If we fail to find a proper editing set starting with the edge e we shall not forget to undo the changes done to G in this failed attempt before we try another edge. We do that in line 12.

The recursion ends when we have edited G to a (P_5, C_5)-free graph (line 2- 3) or when we have no more edits left (line 4 - 5).

Now let us discuss the two optimizations that we applied to Algorithm 3.4 in Algorithm 3.5:

If the found forbidden subgraph $G[V']$ is a P_5 then adding the edge $\{v_1, v_5\}$ to G would only turn $G[V']$ into a forbidden induced C_5. Remember the vertices $v_i \in V'$ are enumerated as in Figure 3.5 by FindP5C5. By adding the dashed line we turn a P_5 into a C_5. To destroy this C_5 we will have to edit another edge e. But e would have destroyed the P_5 anyway. Therefore adding the edge $\{v_1, v_5\}$ was basically useless. Therefore we will not try to edit the edge $\{v_1, v_5\}$.

If the found forbidden induced subgraph is a C_5 then there are other problems of Algorithm 3.4 that we want to avoid in Algorithm 3.5. If we try to destroy a C_5 using a deletion then we turn that C_5 into a P_5. A later recursion that tries to destroy this P_5 will try to add edges that would have destroyed the initial C_5 anyway. Together with one of these edge additions the initial deletions are useless. Another problem is the one depicted in Figure 3.6.

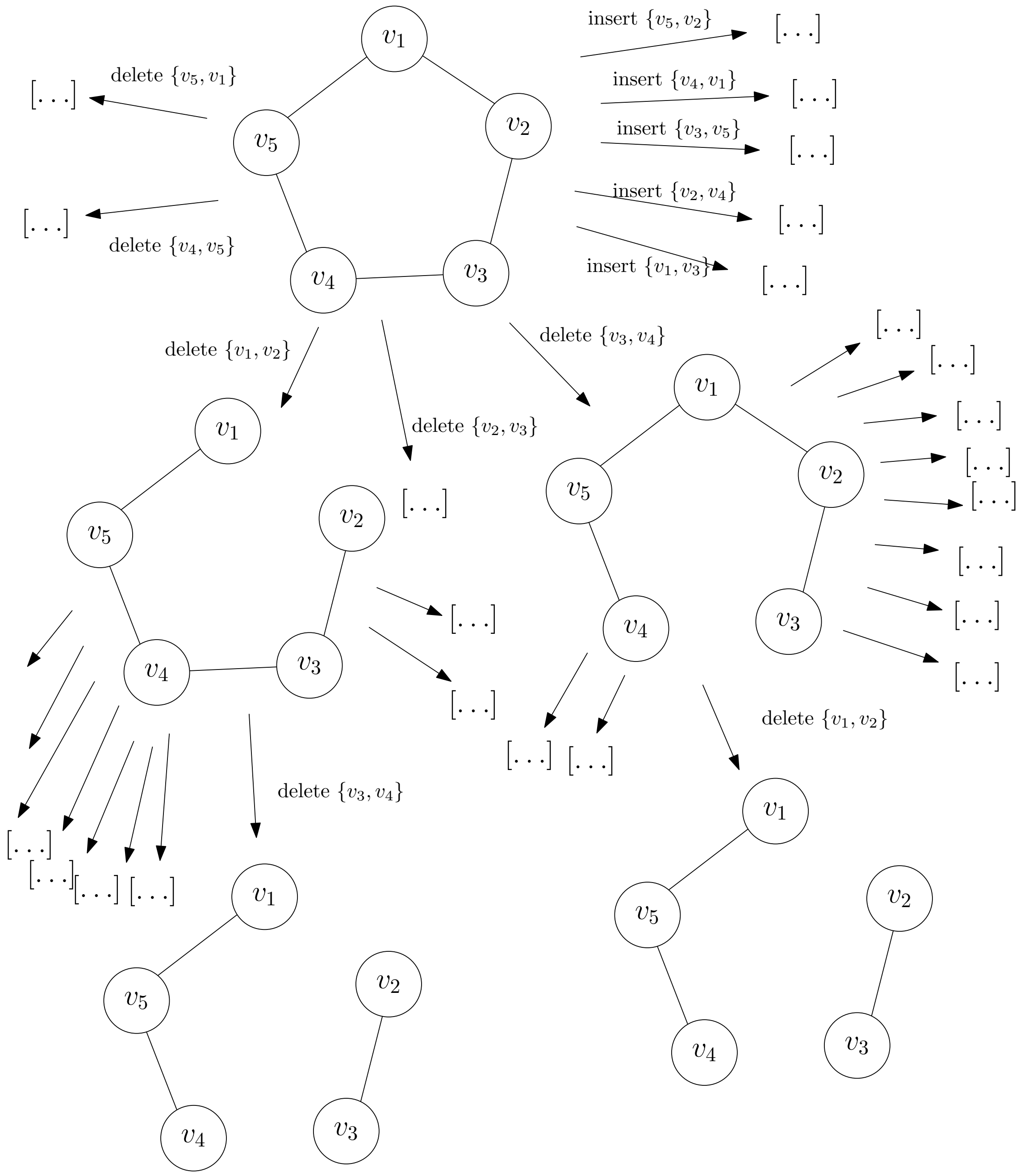


Figure 3.6.: redundant calculations in Algorithm 3.4

Algorithm 3.4: EditToP5C5freeCai

```
Input: Graph $G = (V, E)$, number $k$
Output: boolean value s indicating whether G can be edited to a $(P_5, C_5)$-free
        Graph in at most k steps
1  $found, V' \leftarrow$ FindP5C5($G$)
2  if $found$ = $false$ then
3      return true
4  else if $k$ = $0$ then
5      return false
6  else
7      forall $e := \{u, v\} \subseteq V'$ do
8          $G \leftarrow G_{\{e\}}$
9          $s \leftarrow$ EditToP5C5freeCai($G, k-1$)
10         if $s$ = $true$ then
11             return true
12         $G \leftarrow G_{\{e\}}$
13     return false
```

Here Algorithm 3.4 tries to destroy a C_5 induced by some vertices v_1, v_2, v_3, v_4, v_5. Let A be the recursive call to EditToP5C5FreeCai that is triggered after deleting the edge $\{v_1, v_2\}$. Let B be the recursive call to EditToP5C5FreeCai that is triggered after deleting the edge $\{v_3, v_4\}$. Now let us assume that for example in recursion A and in recursion B both times FindP5C5 returns the remaining P_5 induced by the same vertices - and not some other 5-tupel of vertices that also induce a P_5 or a C_5. Now when recursion A deletes the edge $\{v_3, v_4\}$ it ends up in the same situation as recursion B when it deletes the edge $\{v_1, v_2\}$. So recursion A and recursion B will trigger two further recursive calls that try to edit the exact same graph. This leads to redundant calculations that need unnecessary running time. To prevent these problems in the optimized version of the algorithm when destroying a C_5 using deletions we do both deletions in the same (recursive) call to EditToP5C5Free and not in different recursive calls .

Let us now prove that this optimized version of the algorithm is still correct:

Theorem 3.6. *Algorithm 3.5 is correct. In other words assume we call*

$$s \leftarrow EditToP5C5Free(G, k)$$

on some graph G. Let $G^{(1)}$ be the graph represented by G before the call to EditToP5C5Free(G, k) and let $G^{(2)}$ be the graph represented by G after the call to EditToP5C5Free(G,k). Then:

1. *if the algorithm returns $s = true$, then we have $G^{(2)} = G^{(1)}_F$ for some editing set F with $|F| \leq k$. Furthermore $G^{(2)}$ is (P_5, C_5)-free.*
2. *if the algorithm returns $s = false$, then $G^{(1)}$ can not be edited to a (P_5, C_5)-free graph in $\leq k$ steps. Furthermore we have $G^{(2)} = G^{(1)}$. That is, G will not be modified by the algorithm if there is no solution.*

Proof. Induction base $k = 0$:

Note that for $k = 0$ the algorithm will either end in line 3 or line 4, depending on whether $found$ will be set to true in line 1.

Algorithm 3.5: EditToP5C5free

```
Input: Graph $G = (V, E)$, number $k$
Output: boolean value s indicating whether G can be edited to a $(P_5, C_5)$-free
        Graph in at most k steps
$found, V' \leftarrow$ FindP5C5($G$)
if $found$ = $false$ then
    return true
else if $k = 0$ then
    return false
else if $G[V']$ is a $P_5$ then
    $v_1, v_5 \leftarrow$ first and last vertex of the $P_5$
    forall $e := \{u, v\} \subseteq V', e \neq \{v_1, v_5\}$ do
        $G \leftarrow G_{\{e\}}$
        $s \leftarrow$ EditToP5C5free($G, k-1$)
        if $s$ = $true$ then
            return true
        $G \leftarrow G_{\{e\}}$
    return false
else
    // $G[V']$ is a $C_5$
    if $k \geq 2$ then
        forall $e_1, e_2 \in E(G[V']), e_1 \neq e_2$ do
            $G \leftarrow G_{\{e_1,e_2\}}$
            $s \leftarrow$ EditToP5C5free($G, k-2$)
            if $s$ = $true$ then
                return true
            $G \leftarrow G_{\{e_1,e_2\}}$
    forall $e := \{u, v\} \subseteq V', \{u, v\} \notin E(G[V'])$ do
        $G \leftarrow G_{\{e\}}$
        $s \leftarrow$ EditToP5C5free($G, k-1$)
        if $s$ = $true$ then
            return true, $G'$
        $G \leftarrow G_{\{e\}}$
    return false
```

1) If the algorithm returns $s = true$, then found must be false such that the algorithm ended in line 3. So $G^{(1)}$ is already (P_5, C_5)-free. The algorithm does not modify G before returning true in line 3. Therefore we have $G^{(2)} = G^{(1)} = G^{(1)}_F$, where the editing set $F = \emptyset$ has size $0 = k$.

2) If the algorithm returns $s = false$ then this has happened in line 5. As the algorithm does not modify G before line 5 we have $G^{(2)} = G^{(1)}$. And obviously as found was set to true G is not (P_5, C_5)-free and therefore not editable to a (P_5, C_5)-free graph in zero steps.

Induction hypothesis:

The algorithm fulfils properties 1 and 2 if it is called with $k-1$. If $k \geq 2$ it also fulfils properties 1 and 2 when called with $k-2$.

Induction step $k-2, k-1 \rightarrow k$

1) Assume the algorithm returns $s = true$. Then the algorithm ended in line 3, 12, 21 or 27. If it ended in line 3 then G was already (P_5, C_5)-free and the algorithm leaves G unchanged so $G^{(2)} = G^{(1)} = G^{(1)}_{\emptyset}$ as in the induction base.

If it ended in line 12 or 27, then the most recent recursive call to EditToP5C5Free(G, k-1) has returned true. Let G' be the graph represented by G right before the last recursive call. Note that only the last recursive call returned true, since the algorithm returns immediately as soon as a recursive call is successful. The recursive calls except from the last one, which returned true, did not change the graph G by the induction hypothesis. All edits done in line 18 are undone in line 22. Except from the last edit e^* all edits made in lines 9 and 24 are undone in line 13 and 28. Therefore we have $G' = G^{(1)}_{\{e^*\}}$. By the induction hypothesis, as the last recursion returned true, it has edited G' to some (P_5, C_5)-free graph $G^{(2)} = G'_{F'}$, where $|F'| \leq k-1$. Now we have $G^{(2)} = G'_{F'} = G^{(1)}_F$ where $F := F' \triangle \{e*\}$. That is either $F := F' \cup \{e\}$ or $F := F' \setminus \{e\}$. In both cases we have $|F| \leq |F'| + 1 \leq (k-1) + 1 = k$. Remember the symmetric difference $\triangle$ is introduced in Section 1.2.

If the algorithm ended in line 21 then we can similarly show that the last recursion EditToP5C5Free(G, k-2) edited G from some $G' = G^{(1)}_{\{e_1, e_2\}}$ to some (P_5, C_5)-free $G'_{F'}$ where $|F'| \leq k-2$. And we have $G^{(2)} = G'_{F'} = G^{(1)}_F$, where $F := F' \triangle \{e_1, e_2\}$. Note that we have $|F| \leq |F'| + |\{e_1, e_2\}| \leq k$.

2) Assume the algorithm returns $s = false$. In this case none of the recursive calls has returned true and therefore by induction hypothesis none of them has modified G. Also all modifications to G in lines 9 are undone in line 13, the ones in line 18 are undone in line 22 and the ones in line 24 are undone in line 28. So $G^{(1)} = G^{(2)}$.

Let us also show that $G^{(1)}$ can not be edited to a (P_5, C_5)-free graph in $\leq k$ steps: Assume there was an editing set F with $|F| \leq k$ such that $G^{(1)}_F$ is (P_5, C_5)-free. As the algorithm returned true we did not end in line 3, therefore in line 1 we must have found a P_5 or a C_5 $G^{(1)}[V']$ in $G^{(1)}$ induced by some set $V' \subseteq V$.

Assume $G^{(1)}[V']$ is a P_5. Let v_1, v_5 be the first and the last vertex of this P_5. Because $G^{(1)}_F$ is (P_5, C_5)-free the graph $G^{(1)}_F[V']$ - induced by V' *after* applying F - is neither a P5 nor a C5. Therefore there must be an edge $e* := \{u, v\} \in F$ such that $\{u, v\} \subset V', \{u, v\} \neq \{v_1, v_5\}$. Note that if $\{v_1, v_5\}$ was the only pair of vertices of V' that belongs to F then $G^{(1)}_F[V']$ would be a C_5. Now as the loop in lines 8-14 iterates through all of these edges sometime line 9 will edit the edge $e* \in F$. Then EditToP5C5Free(G, k-1) is called recursively on $G = G' = G^{(1)}_{\{e*\}}$. Since $(G')_{F \setminus \{e*\}} = G^{(1)}_F$ is (P_5, C_5)-free and $|F \setminus \{e\}| = |F| - 1 \leq k-1$, G' is editable to a (P_5, C_5)-free graph in $\leq k-1$ steps. By induction hypothesis the recursion must return true, so we end in line 12 returning true. This is a contradiction to our initial assumption that the algorithm returns false.

Now assume $G^{(1)}[V']$ is a C_5. If F contains some edge addition $e* := \{u,v\} \subseteq V', e* \notin E(G^{(1)})$, then the loop in lines 23-28 will sometime edit this edge. Then analogue to the P_5 case, we can show that the recursion in line 25 must return true such that the algorithm ends in line 27 and returns true. So we have the same contradiction.

Now assume F contains no edge addition $e* := \{u,v\} \subseteq V', e* \notin E(G^{(1)})$. If F contained only zero or one edge deletions of edges in our induced C_5-subgraph $G^{(1)}[V']$, then $G^{(1)}_F[V']$ would still be a C_5 or a P_5. So instead there must be at least two edits $e_1, e_2 \in F$ that are deletions of edges in $G^{(1)}[V']$. Since we have $k \geq |F| \geq 2$ now we know that the loop in lines 17-22 will be executed. As this loop iterates over all such pairs of edges, it will eventually do a recursion EditToP5C5Free(G, k-2) with $G := G' := G^{(1)}_{\{e_1,e_2\}}$. As $G'_{F \setminus \{e_1,e_2\}} = G^{(1)}_F$ is (P_5, C_5)-free and $|F \setminus \{e_1, e_2\}| = |F| - 2 \leq k - 2$, G' is editable to a (P_5, C_5)-free graph in $\leq k - 2$ steps. Therefore by the induction hypothesis the recursion should return true, such that we end in line 21 returning true. Again a contradiction to our initial assumption, that the algorithm returned false. □

Theorem 3.7. *For each $k \in \mathbb{N}$ let t_k be the maximum number of times that the routine FindP5C5(...) is called, when executing EditToP5C5Free(G, k). Let t_k also include the calls to FindP5C5(...) in the recursive calls to EditToP5C5Free(...).*

Then:

- $t_0 = 1$
- $t_1 = 10$
- $t_k = 1 + \max\{9t_{k-1}, \quad 5t_{k-1} + 10t_{k-2}\}$ *for* $k \geq 2$

Proof. $t_0 = 1$:

If the algorithm is executed with $k = 0$ then it will call FindP5C5(...) one time in line one. After that it will return in line 3 or 5.

$t_1 = 10$: The algorithm executes FindP5C5(...) one time in line 1 and then possibly some more times in the recursive calls to EditToP5C5Free(G,0). There are three possibilities. If no P_5 or C_5 is found, then there will be no recursion. So the total number of calls to FindP5C5(...) is 1.

If a P_5 is found then the body of the loop from lines 8-13 will be executed for vertex pairs in the subgraph induced by V'. Each time it will make one recursive call to EditToP5C5Free(G,0). There are $\binom{|V'|}{2} = \binom{5}{2} = 10$ such vertex pairs. As the loop body is not executed for $\{v_1, v_5\}$ there will be only 9 recursions, each causing one call to FindP5C5(...). So the total number of calls to FindP5C5(...) is $1 + 9 = 10$.

If a C_5 is found then the body of the loop from lines 23-28 will be executed for non-edge vertex-pairs in V'. Each execution of the loop body causes one recursive call to EditToP5C5Free(G,0). There are 10 vertex pairs. As the loop body is only executed for possible vertex additions but not for the 5 possible vertex deletions, there will be only 5 recursions, each causing one call to FindP5C5(...). So the total number of calls to FindP5C5(...) is $1 + 5 = 6$.

So $t_1 = \max\{1, 10, 6\} = 10$.

$t_k = 1 + \max\{9t_{k-1}, \quad 5t_{k-1} + 10t_{k-2}\}$:

The algorithm executes FindP5C5(...) one time in line 1 and then possibly some more times in the recursive calls to EditToP5C5Free(G,k-1) or EditToP5C5Free(G,k-2). Again there are three possibilities:

If no P_5 or C_5 is found, then there will be no recursion. So the total number of calls to FindP5C5(...) is 1.

If a P_5 is found then the loop from lines 8-13 will do 9 recursive calls to EditToP5C5Free(G,k-1). The number 9 can be derived as in the proof for $t_1 = 10$. Let $r_1, \dots, r_9$ be the number of calls to FindP5C5(...) in each of these recursions. Then the total number of calls to FindP5C5(...) is $1 + r_1 + \dots + r_9$. This value is maximal for $r_1 = \dots = r_9 = t_{k-1}$. So the maximum value in case a P_5 was found is $1 + 9t_{k-1}$.

Now supposes a C_5 is found. Then the loop from lines 23-28 will do 5 recursive calls to EditToP5C5Free(G,k-1). The number 5 can be derived as in the proof for $t_1 = 10$. Let $r_1, \dots, r_5$ be the number of calls to FindP5C5(...) in each of these recursions.

As $k \geq 2$ also the loop in lines 17-22 will be executed. It will do one recursion to EditToP5C5Free(G,k-2) for each pair of edges in the found C_5. As there are 5 edges there are $\binom{5}{2} = 10$ such pairs. Let $r'_1, \dots, r'_{10}$ be the number of calls to FindP5C5(...) in each of these recursions to EditToP5C5Free(G,k-2). Then the total number of calls to FindP5C5(...) is $1 + r_1 + \dots + r_5 + r'_1 + \dots + r'_{10}$. This value is maximal for $r_1 = \dots = r_5 = t_{k-1}$ and $r'_1 = \dots = r'_{10} = t_{k-2}$. So the maximum value in case a C_5 was found is $1 + 5t_{k-1} + 10t_{k-2}$. The total maximum value is therefore $\max\{1, 1 + 9t_{k-1}, 1 + 5t_{k-1} + 10t_{k-2}\} = 1 + \max\{9t_{k-1}, 5t_{k-1} + 10t_{k-2}\}$.

□

Theorem 3.8. *Let t_k be defined as in Theorem 3.7 for $k \in \mathbb{N}$, then:*

$$t_k = \frac{1}{8}(9^{k+1} - 1)$$

Proof. Induction base $k = 0, k = 1$:

$$\frac{1}{8}(9^{0+1} - 1) = 1 = t_0$$
$$\frac{1}{8}(9^{1+1} - 1) = 10 = t_1$$

Induction hypothesis:

$t_{k'} = \frac{1}{8}(9^{k'+1} - 1)$ for all $0 \leq k' < k$

Induction step:

$$\begin{aligned} 5t_{k-1} + 10t_{k-2} &=_{Ind.Hyp.} \frac{5}{8}(9^k - 1) + \frac{10}{8}(9^{k-1} - 1) \\ &= \frac{1}{8}(5 \cdot 9^k - 5) + \frac{1}{8}(10 \cdot 9^{k-1} - 10) \\ &= \frac{1}{8}(55 \cdot 9^{k-1} - 15) \\ &< \frac{1}{8}(81 \cdot 9^{k-1} - 9) \\ &= \frac{9}{8}(9^k - 1) =_{Ind.Hyp.} 9t_{k-1} \end{aligned}$$

Thus:

$$\begin{aligned} t_k &= 1 + \max\{9t_{k-1}, \quad 5t_{k-1} + 10t_{k-2}\} \\ &= 1 + \frac{9}{8}(9^k - 1) \\ &= \frac{8}{8} + \frac{9^{k+1}}{8} - \frac{9}{8} \\ &= \frac{1}{8}(9^{k+1} - 1) \end{aligned}$$

□

Theorem 3.9. *EditToP5C5Free(G, k) can be executed in time*

$$t \in O(\min(\; 9^k \cdot m \cdot (d+k)^2, \quad 9^k \cdot m \cdot n^2 \;))$$

on a graph $G = (V, E)$ where $n := |V|$, $m := |E|$ and $d := \max_{v \in V} d(v)$ is the maximum degree of G.

Proof. If we consider the time of one call to EditToP5C5Free without the time needed for the recursive calls, then the majority of this time is the time spent in the subroutine FindP5C5 in line 1. Note that the bodies of the loops contained in this algorithm are executed only a constant number of times. And considering the discussion in Section 3.1 we know that each line of code can be executed in $O(1)$ time except line 1 and the recursive calls. Note that to check whether $G[V']$ is a P_5 in line 6 we only need to check in $O(1)$ time whether v_1 and v_5 are adjacent to each other.

So the asymptotic running time of the algorithm is determined by the time needed for the calls to the FindP5C5 subroutine. According to Theorem 3.5 each of the t_k calls to this subroutine needs $O(\tilde{m} \cdot \tilde{d}^2)$ time. Where $\tilde{d}$ is the maximum degree and $\tilde{m}$ is the maximum number of edges in any of the edited versions of G that occur during the execution of the algorithm. So the total asymptotic running time is in $O(t_k \cdot \tilde{m} \cdot \tilde{d}^2)$. Obviously $\tilde{d} \leq d + k$ and $\tilde{m} \leq (m + k)$ as there are at most k edges added to G. Note that in the case of $k \geq m$ the solution to the editing problem is trivial. We can just delete all of the edges instead of running a complicated algorithm in order to find a proper editing set. Therefore we assume that EditToP5C5Free is called with values $k \leq m$ only. Thus $\tilde{m} \leq 2m$. So the total asymptotic running time is $O(t_k \cdot m \cdot (d+k)^2)$. On the other obviously we can also use the number of vertices n as an upper bound for $\tilde{d}$ instead of $d + k$. Thus we get an asymptotic running time in $O(t_k \cdot m \cdot n^2)$. Now we only need to insert the actual value of t_k from Theorem 3.8. □

Algorithm 3.5 can also be modified to a deletion algorithm as described in Algorithm 3.6. Similar to the editing version we can show that DeleteToP5C5Free(G, k) returns true if and only if there is an deletion set $F \subseteq E(G)$ such that $|F| \leq k$ and G_F is (P_5, C_5)-free. Also if there is such a solution then DeleteToP5C5Free will actually edit G to a (P_5, C_5)-free graph G_F with some $|F| \leq k$.

Let us now look at the running time of the deletion version.

Theorem 3.10. *For each $k \in \mathbb{N}$ let t^*_k be the maximum number of times that the routine FindP5C5(...) is called, while executing DeleteToP5C5Free(G, k). Then:*

$$t^*_k = \frac{1}{3}(4^{k+1} - 1)$$

Algorithm 3.6: DeleteToP5C5free

```
Input: Graph $G = (V, E)$, number $k$
Output: boolean value s indicating whether G can be edited to a $(P_5, C_5)$-free
        Graph in at most k steps using deletions only
1  $found, V' \leftarrow$ findP5orC5($G$)
2  if $found = false$ then
3      return true
4  else if $k = 0$ then
5      return false
6  else if $G[V']$ is a $P_5$ then
7      $v_1, v_5 \leftarrow$ first and last vertex of the $P_5$
8      forall $e \in E(G[V'])$ do
9          $G \leftarrow G_{\{e\}}$
10         $s \leftarrow$ DeleteToP5C5free($G, k - 1$)
11         if $s = true$ then
12             return true
13         $G \leftarrow G_{\{e\}}$
14     return false
15 else
       // $G[V']$ is a $C_5$
16     if $k \geq 2$ then
17         forall $e_1, e_2 \in E(G[V']), e_1 \neq e_2$ do
18             $G \leftarrow G_{\{e_1,e_2\}}$
19             $s \leftarrow$ DeleteToP5C5free($G, k - 2$)
20             if $s = true$ then
21                 return true
22             $G \leftarrow G_{\{e_1,e_2\}}$
23     return false
```

Proof. By a similar argument as in Theorem 3.7 we get the recursive formula:

- $t_0^* = 1$
- $t_1^* = 5$
- $t_k^* = 1 + \max\{4t_{k-1}^*, \quad 10t_{k-2}^*\}$ for $k \geq 2$

Now we can prove the explicit formula by induction: Induction base $k = 0, k = 1$:

$$\frac{1}{3}(4^{0+1} - 1) = 1 = t_0^*$$

$$\frac{1}{3}(4^{1+1} - 1) = 5 = t_1^*$$

Induction hypothesis:

$t_{k'}^* = \frac{1}{3}(4^{k'+1} - 1)$ for all $0 \leq k' < k$

Induction step:

$$\begin{aligned} 10t_{k-2}^* &=_{Ind.Hyp.} \frac{10}{3}(4^{k-1} - 1) \\ &= \frac{1}{3}(10 \cdot 4^{k-1} - 10) \\ &< \frac{1}{3}(16 \cdot 4^{k-1} - 4) \\ &= \frac{4}{3}(4^k - 1) =_{Ind.Hyp.} 4t_{k-1}^* \end{aligned}$$

Thus:

$$\begin{aligned} t_k^* &= 1 + \max\{4t_{k-1}^*, \quad 10t_{k-2}^*\} \\ &= 1 + \frac{4}{3}(4^k - 1) \\ &= \frac{3}{3} + \frac{4^{k+1}}{3} - \frac{4}{3} \\ &= \frac{1}{3}(4^{k+1} - 1) \end{aligned}$$

□

Theorem 3.11. *DeleteToP5C5Free(G, k) can be executed in time*

$$t \in O(4^k \cdot m \cdot d^2)$$

on a graph $G = (V, E)$ where $n := |V|$, $m := |E|$ and $d := \max_{v \in V} d(v)$ is the maximum degree of G.

Proof. Similar to the proof of Theorem 3.9 we can show that $t \in O(t_k^* \cdot \tilde{m} \cdot \tilde{d}^2)$. Here $\tilde{d}$ is the maximum degree and $\tilde{m}$ is the maximum number of edges in any of the edited versions of G that occur during the execution of the algorithm. As the deletion of edges does not increase the number of edges or the degree of any vertex, we have $\tilde{m} \leq m$ and $\tilde{d} \leq d$. □

There is another possible optimization to the editing algorithm that was not applied in Algorithm 3.5. In order to describe this better let $R := (V_R, E_R)$ be a directed graph that represents the tree of recursive calls to the editing algorithm when it is applied to some graph G. So V_R shall contain the initial call r_1 to EditToP5C5Free and all the recursive calls r_i to EditToP5C5Free that were triggered by r_1 directly or indirectly. Furthermore the edge (r_i, r_j) shall exist in R if r_j was directly triggered by the (recursive) call r_i. If there is a path from some $r_1 \in V_R$ to some $r_2 \in V_R$ then we call r_1 an ancestor of r_2 and r_2 a descendant of r_1. Now having introduced this formalism let us get back to the optimization of the editing algorithm.

In a recursive call $r \in V_R$ actually it is unnecessary to try to undo an edit made by any of its ancestors in the tree R. To know which pairs of vertices were already edited the algorithm could always label an edited vertex pair before doing a recursive call. After the recursive call when it undoes the edit it should also remove the label again. Then in the loops in lines 7 - 14, 17 - 22 and 23 - 29 the algorithm could skip labelled edges and only iterate over unlabelled edges. As it is difficult to tell how often the call to FindP5C5 in line will return a subgraph that contains an already labelled edge, it is also difficult to analyse theoretically how much this optimization improves the running time of the algorithm. But it is likely that it improves the running time most times in practice.

Note that this optimization can only be used for the editing algorithm but not for the deletion algorithm. The deletion algorithm cannot undo edits anyway. As it does not do inserts at all, it will especially not insert previously deleted edges.

When we use this optimization the number of needed recursive calls depends of course on the number of already labelled vertex pairs in the subgraph returned by FindP5C5. Therefore, as there are usually many P_5s and C_5s in the graph with presumably different numbers of labelled vertex pairs, we might wonder if we should specifically look for a forbidden subgraph that maximizes the number of labelled vertex pairs instead of just taking an arbitrary one. Similar to the routine FindP5C5 we could also modify the counting algorithm CountP5C5 to another search algorithm FindMostLabeledP5C5 that returns the most labelled induced P_5 or C_5 instead of just a random one. However such a routine could not abort the calculation as soon as the first induced P_5 or C_5 is found like FindP5C5. Instead it has to look at each one of them in order to find the best one. Therefore the running time of FindMostLabeledP5C5 is in $O(m \cdot d^3)$ like the running time of CountP5C5 and not in $O(m \cdot d^2)$ like the running time of FindP5C5.

The question is whether the presumably decreased number of recursive calls when using FindMostLabeledP5C5 instead of FindP5C5 outweighs the increased running time needed for each single recursive call. The practical results of Section 4.2 suggest that a combination of both approaches that uses FindMostLabeledP5C5 in "higher" recursive calls and FindP5C5 in "deeper" recursive calls might be the best solution.

3.5. Heuristic (P_5, C_5)-Free Editing

Unfortunately the running time of the exact editing algorithm is infeasible for most of the graphs that we study in Chapter 4. Therefore we use a heuristic approximation similar to the one used in [NG13]. The heuristic is described in Algorithm 3.7. In each step we test every possible edge-deletion and every possible edge addition. Then we greedily chose the edge edit that resulted in the lowest number of remaining forbidden induced subgraphs ($\#P_5$s + $\#C_5$s).

Note that lines one and two of Algorithm 3.7 are only needed in the case that G is already (P_5, C_5)-free. In that case we just want to return an empty editing set and not unnecessarily go through one iteration of the while-loop. If we know G is not already (P_5, C_5)-free we could also remove line one and two and use a do-while-loop.

A similar heuristic can also be used for the deletion problem see Algorithm 3.8.

Algorithm 3.7: HeurEditToP5C5Free

```
    Input: Graph $G = (V, E)$
    Output: a (hopefully small) editing set $F$ such that $G_F$ is $(P_5, C_5)$-free
 1  $\mathsf{P5count}, \mathsf{C5count} \leftarrow$ countP5C5 (G)
 2  $\mathsf{minCount} \leftarrow \mathsf{P5count} + \mathsf{C5count}$
 3  while $\mathsf{minCount} > 0$ do
 4      $\mathsf{minCount} \leftarrow \infty$
 5      forall $e := \{u, v\} \subseteq V(G)$ and $e \notin F$ do
 6          $\mathsf{P5count}, \mathsf{C5count} \leftarrow$ countP5C5( $G_{F \cup \{u,v\}}$)
 7          if $\mathsf{P5count} + \mathsf{C5count} < \mathsf{minCount}$ then
 8              $\mathsf{bestE} \leftarrow e$
 9              $\mathsf{minCount} \leftarrow \mathsf{P5count} + \mathsf{C5count}$
10      $F \leftarrow F \cup \{\mathsf{bestE}\}$
11  return F
```

Algorithm 3.8: HeurDeleteToP5C5Free

```
    Input: Graph $G = (V, E)$
    Output: a (hopefully small) deletion set $F \subseteq E(G)$ such that $G_F$ is $(P_5, C_5)$-free
 1  $\mathsf{P5count}, \mathsf{C5count} \leftarrow$ CountP5C5 (G)
 2  $\mathsf{minCount} \leftarrow \mathsf{P5count} + \mathsf{C5count}$
 3  $F \leftarrow \emptyset$
 4  while $\mathsf{minCount} > 0$ do
 5      $\mathsf{minCount} \leftarrow \infty$
 6      forall $e \in E(G_F)$ do
 7          $\mathsf{P5count}, \mathsf{C5count} \leftarrow$ CountP5C5( $G_{F \cup \{u,v\}}$)
 8          if $\mathsf{P5count} + \mathsf{C5count} < \mathsf{minCount}$ then
 9              $\mathsf{bestE} \leftarrow e$
10              $\mathsf{minCount} \leftarrow \mathsf{P5count} + \mathsf{C5count}$
11      $F \leftarrow F \cup \{\mathsf{bestE}\}$
12  return F
```

Theorem 3.12. *Let $G = (V, E)$ be some graph, and let $n := |V|, m := |E|$ and let $d := \max_{v \in V} d(v)$.*

HeurEditToP5C5Free(G) can be executed in time

$$O(\min(\quad k \cdot n^2 \cdot (m + k) \cdot (d + k)^3, \quad k \cdot n^5 \cdot (m + k) \quad))$$

where k is the number of edits in the editing set that will be found by HeurEditToP5C5Free.

HeurDeleteToP5C5Free(G) can be executed in $O(k_{del} \cdot m^2 \cdot d^3)$ time, where k_{del} is the number of deletes in the deletion set that will be found by HeurDeleteToP5C5Free.

Proof. The single execution of one line of code in HeurEditToP5C5Free except for the call to CountP5C5 in line 6 can be executed in $O(1)$ time. Note that we can check whether

$e \in F$ in line 5 in $O(1)$ time if we keep an adjacency matrix representation of the graph (V, F) in memory. Note that no line of code is executed more often than the call to CountP5C5 in line 6. Therefore the majority of the running time of HeurEditToP5C5Free is the time spent in the subroutine CountP5C5.

This subroutine is called $k \cdot n^2$ times. The while loop is executed k times and each time the inner forall loop is executed n^2 times. Each execution of the $k \cdot n^2$ calls to CountP5C5 needs time in $O(\tilde{m} \cdot \tilde{d}^3)$ by Theorem 3.3. Here let $\tilde{m}$ and $\tilde{d}$ denote the maximum number of edges and the maximum degree in any of the modified versions of G that we get during the execution of CountP5C5. Thus $t \in O(k \cdot n^2 \cdot \tilde{m} \cdot \tilde{d}^3$. Using $\tilde{m} \leq m + k$ and $\tilde{d} \leq d + k$ we get the first argument of the min function as upper bound for the running time of HeurEditToP5C5Free. If we use $\tilde{k} \leq n$ instead of $\tilde{k} \leq d + k$ then we get the other running time bound for HeurEditToP5C5Free.

Similar to the editing version in the deletion version the majority of the running time is spent in the subroutine CountP5C5. Here the subroutine is only called $O(k_{del} \cdot \tilde{m})$ times where $\tilde{m}$ is the maximum number of edges in any of the modified versions of G. Each execution of the $O(k_{del} \cdot \tilde{m})$ calls to the subroutine takes $O(\tilde{m} \cdot \tilde{d}^3)$ time. As in the editing case $\tilde{d}$ denotes the maximum degree appearing in any of the edited versions of G. Note however that as we only use deletions we have $\tilde{m} \leq m$ and $\tilde{d} \leq d$. Multiplying this yields the desired term for the running time of HeurDeleteToP5C5Free.

□

4. Experimental Evaluation

In this chapter we discuss the results of applying the heuristic and the exact editing and deletion algorithms of Chapter 3 to some real world graphs. In Sections 4.1 to 4.3 we first evaluate the performance of the algorithms only as means to solve the (P_5, C_5)-free editing and the (P_5, C_5)-free deletion problem. In Section 4.4 we finally evaluate the results in the context of community detection by visualising the graphs together with the calculated editing sets. We use the same graphs that were used in [NG13] in order to study (P_4, C_4)-free editing. Table 4.1 summarizes some general information about the graphs. For each graph it shows the number of vertices, the number of edges, the maximal degree $\max_{v \in V} d(v)$ appearing in the graph and the initial number of P_5s and C_5s.

We implemented the algorithms in C++. We ran the algorithms on a computer with an Intel(R) Core(TM) i7-2600K CPU with 3.4 GHz. The running times that are given in some of the tables of this chapter refer to an execution this machine using no parallelization.

Table 4.1.: general graph info

name	#vertex	#edge	max degree	#P5	#C5
karate	34	78	17	1583	20
lesmis	77	254	36	8497	62
dolphins	62	159	12	6189	142
grass	75	113	17	1657	1
football	115	613	12	120326	1232

4.1. Heuristic Algorithms

Table 4.3 and Table 4.2 show the size of the resulting editing sets when applying the heuristic editing and the heuristic deletion algorithm to the graphs. The tables also show the needed running time. The results of the editing version are rather disappointing in case of the graphs karate and lesmis. While we do not know what is the size of the actual best editing set, we know that 18 edits for karate and 293 edits for lesmis is too much. Every deletion set is also an editing set. Therefore the solutions with 13 and 153 edits found by the heuristic deletion algorithm are also a better solutions for the editing problem than the solutions found by the actual editing algorithm. Even more disappointing [BHSW15] found a (P_4, C_4)-free editing set for lesmis consisting of only 60 edits. As each P_5 and each C_5 contains a P_4 we know that a (P_4, C_4)-free graph is also (P_5, C_5)-free. Thus there is a solution with only 60 edits while the heuristic needed 293 edits. Interestingly the (P_4, C_4)-free editing solution containing only 60 edits was calculated using the same heuristic approach. As this heuristic works so well in the case of (P_4, C_4)-free editing, the disappointing result in the case of (P_5, C_5)-free editing is rather surprising.

When we look at the number of remaining P_5s and C_5s after each edit applied to the graph lesmis by the heuristic editing algorithm, we also get the impression that something the heuristic does not perform very well here. This is documented in Table 4.4. The algorithm often ends up in situations where none of the edits actually improves the number of remaining P_5s and C_5s. The column type lists the types of edits that were applied by the heuristic (deletion or insert). The entries in the columns P_5 and C_5 contain the remaining number of P_5s and C_5s after applying the edit in the column type in the corresponding row. A column of the form (ix)del x y means that the heuristic applied i deletions here and the first one decreased the number of P_5s and C_5s to x and y. The other $i - 1$ deletions left the number of P_5s and C_5s unchanged. In other words this line is equal to i consecutive lines containing the entries del x y. A line of the form (ix)ins x y is defined analogously. Note that after reaching a number of 168 remaining P_5s and 0 remaining C_5s the heuristic applied 154 edits that caused no further improvements!

Such a strange phenomenon did not occur in the other four graphs. In these graphs each step of the heuristic decreased the number of remaining forbidden induced subgraphs at least a little bit. If the interested reader wants to compare this by himself he can find tables analogue to Table 4.4 in the appendix.

Table 4.2.: heuristic deletion results

name	#deletions	needed time
karate	13	0.24 s
lesmis	153	16.50 s
dolphins	62	3.28 s
grass_web	43	0.61 s
football	190	8 min

Table 4.3.: heuristic editing results

name	#deletions	#inserts	sum	needed time
karate	10	8	18	2.33 s
lesmis	66	227	293	810.36 s
dolphins	48	6	54	43.69 s
grass_web	19	9	28	14.06 s
football	181	9	190	approx. 95 min

4.2. Exact Algorithm

Remember that in the end of Section 3.4 we proposed an optimized version of Edit-ToP5C5Free, that in a recursive call r skips other recursive calls belonging to vertex pairs that were already edited by ancestors of r. The question was whether we should still use the subroutine FindP5C5 in this optimized version of Algorithm 3.5. Or should we instead use the subroutine FindMostLabeledP5C5 which increases the running time per recursive call but presumably decreases the total number of needed recursive calls?

Table 4.4.: lesmis editing log

type	P5	C5	type	P5	C5	type	P5	C5	type	P5	C5
del	7366	59	del	1965	3	del	570	0	del	144	0
ins	6854	32	ins	1826	2	del	536	0	(15x)ins	144	0
del	6449	25	del	1690	0	del	502	0	(5x)del	144	0
del	6021	18	ins	1568	0	del	468	0	ins	120	0
del	5729	15	ins	1462	0	del	430	4	del	120	0
del	5447	13	del	1368	0	del	343	2	(15x)ins	120	0
ins	5158	13	ins	1280	0	del	242	0	(4x)del	120	0
del	4903	6	del	1194	0	del	222	0	ins	96	0
del	4651	6	ins	1114	0	del	202	0	del	96	0
del	4324	6	del	1042	0	ins	188	0	(15x)ins	96	0
del	4102	5	del	985	0	ins	176	0	(3x)del	96	0
ins	3917	5	del	928	0	ins	169	0	ins	72	0
ins	3649	5	del	871	0	(3x)del	168	0	del	72	0
ins	3453	5	ins	826	0	(23x)ins	168	0	(15x)ins	72	0
del	3230	5	ins	785	0	del	168	0	(2x)del	72	0
del	2949	5	del	745	0	(83x)ins	168	0	(2x)ins	48	0
del	2796	3	ins	706	0	del	168	0	ins	0	0
del	2648	3	del	672	0	(38x)ins	168	0			
del	2508	3	del	638	0	(6x)del	168	0			
del	2187	3	del	604	0	ins	144	0			

We shall now argue why a combination of both approaches might also be a good solution. To understand that assume we are in some recursive call r_1 of the form EditToP5C5Free(G, k). Further assume we skip a recursive call r_2 that would have been (directly) triggered by r_1 if the corresponding vertex pair had not been labelled. Then we will not only save the running time that is needed by r_2 directly but we will also skip the execution of the descendants of r_2. By descendants we mean the descendants in the tree R of recursive calls as defined in Section 3.4. Note that the size of this skipped subtree consisting of r_2 and its descendants is small if k is small. For example for $k = 1$ it consists only of r_2 itself. As r_2 is called with $k := 0$ in this case it will not trigger further recursive calls. However if k is big then the skipped subtrees will presumably be big too, and skipping them will save a lot of running time. So finding induced P_5s or C_5s with a high number of labelled vertex pairs that allow us to skip such subtrees is much more important for "higher" recursive calls with a big value of k than for "deeper" recursive calls with a low value of k. Therefore we defined a threshold value *thres*. And in our optimized implementation of EditToP5C5Free we shall use the algorithm FindMostLabeledP5C5 in "higher" recursive calls with $k \geq thres$ and we shall use FindP5C5 in "deeper" recursive calls with $k < thres$.

Now in order to find out what value we should use as threshold *thres* we ran the algorithm EditToP5C5Free(G, k) several times, where G is the karate-graph and $k = 9$. The algorithm did not succeed, so there is no solution with 9 steps. Each time we ran the algorithm we used a different value of thres. The results are shown in Tables 4.5 and 4.6.

The tables contain the total numbers of recursive calls needed for each value of *thres* in the row "rec. Count". Furthermore the tables contain some statistics specific for each "recursion depth". More precisely for the sets of recursive calls with the same value of $k := i$. The row "i) rec. Count" contains the total number x_i of recursive calls with $k := i$. As the algorithm did not succeed each of these x_i recursive calls must have found either a P_5 or a C_5. The row "i) num. P5s" contains the percentage p_i of how many of these x_i recursive calls found a P_5. The row “i -> i-1 (p5)” shows the average number of children in R of these $p_i\% \cdot x_i$ recursive calls. The remaining $100\% - p_i\%$ of the x_i recursive calls found a C_5. These recursive calls can trigger further recursive calls with $k = i - 1$ after destroying the C_5 by an edge addition and they can also cause recursive calls with $k = i - 2$ after destroying the C_5 using two edge deletions. The average number of children with $k = i - 1$ of each of these $(100\% - p_i\%) \cdot x_i$ recursive calls is documented in the row “i->i-1 (c5)”. The average number of children with $k = i - 2$ of these $(100\% - p_i\%) \cdot x_i$ recursive calls is documented in the row “i->i-2 (c5)”.

Note that the number of children of each recursive call finding a P_5 would be 9 in the unoptimized version of EditToP5C5Free, that does not label already edited edges. The number of “$i - 1$-children” of the recursive calls finding a C_5 would be 5. The number of “$i - 2$” children would be 10. Note that the numbers of children are lower in Tables 4.5 and 4.6 as there are apparently often already labelled vertex pairs in the found forbidden subgraphs.

Note total number of recursive calls in the second row of Tables 4.5 and 4.6 decreases when we decrease the value of *thres*. That is as expected the number of recursions is lower if we use the algorithm FindMostLabeledP5C5 instead of the algorithm FindP5C5 more often. But is it actually worth the higher running time of the algorithm FindMostLabeledP5C5? To find that out the first row of the table shows the total needed running time for each value of thres. Note that in a recursive call with $k = 0$ FindMostLabeledP5C5 has no advantage over FindP5C5 as for $k = 0$ there will be no further recursive calls anyway. Therefore we know without practical experiments that $thres = 0$ will not be better than $thres = 1$. Note that $thres = 0$ corresponds to the case that we exclusively use FindMostLabeledP5C5 and $thres = 10 = k + 1$ correspond to the case that we exclusively use FindP5C5. Using FindMostLabeledP5C5 exclusively needs more than 6100 seconds and using FindP5C5 exclusively needs 1300 seconds. However when we combine both approaches with $thres = 3$ we need only 470 seconds in this example. So this experiment suggests that a combination of both approaches is actually better than using one of the two search algorithms exclusively. Thereby we can reduce the running time by more than a half in this example.

4.3. Combining Exact and Heuristic Algorithms

Unfortunately the exact algorithms DeleteToP5C5Free and EditToP5C5Free are too slow to calculate an editing set or deletion set on their own for the graphs that we study here. However we can still use them to optimize the editing sets found by the heuristic algorithms.

The idea is as follows: Assume we have an editing set H calculated by some heuristic for a graph G such that G_H is (P_5, C_5)-free. To check whether H is an optimal editing set, and to find a better editing set if H is not, in theory we should call EditToP5C5Free$(G, |H| - 1)$. However, as the running time is exponential in $k := |H| - 1$ and $|H| - 1$ might be a big number, the calculation may be infeasible. So instead we first apply some but not all of the heuristic edits to G yielding a graph $G_{H \setminus S}$. This graph $G_{H \setminus S}$ contains all edits of H except the ones in some subset $S \subseteq H$. Then we apply the exact algorithm. That is we call EditToP5C5Free$(G_{H \setminus S}, |S| - 1)$. This calculation is feasible if $|S|$ is small enough. If we find a subset $S \subseteq H$ such that EditToP5C5Free$(G_{H \setminus S}, |S| - 1)$ returns true, then the algorithm has edited $G' := G_{H \setminus S}$ to some (P_5, C_5)-free graph $G'_F = G_{(H \setminus S) \triangle F}$ with $|F| \leq |S| - 1$. Thus we have found a better editing set $H_2 := (H \setminus S) \triangle F$. Note that if F has undone edits in H then we cannot just define H_2 as the union of $(H \setminus S)$ and F, since H shall not apply these undone edits. Instead we used the symmetric difference defined in Section 1.2. Also note that in any case $|H_2| \leq |H|$. After we optimized H to H_2 we can then look for subsets of H_2 to further optimize H_2 in the same way.

Table 4.5.: performance of EditToP5C5Free and number of recursive calls (part 1)

thres	1	2	3	4	5
time (secs)	6100	950	470	500	620
rec. Count	37 647 916	37 621 624	59 564 403	84 156 500	98 239 534
9) rec. Count	1	1	1	1	1
9) num. P5s	100%	0%	100%	100%	100%
9 -> 8 (p5)	9	-	9	9	9
9-> 8 (c5)	-	5	-	-	-
9-> 7 (c5)	-	10	-	-	-
8) rec. Count	9	5	9	9	9
8) num. P5s	67%	80%	44%	78%	67%
8-> 7 (p5)	8.00	8.00	8.00	8.14	8.00
8-> 7 (c5)	4.33	5.00	4.40	4.50	4.33
8-> 6 (c5)	8.67	6.00	8.40	8.00	8.67
7) rec. Count	61	47	54	66	61
7) num. P5s	80%	89%	93%	83%	90%
7-> 6 (p5)	7.43	7.52	7.52	7.51	7.53
7-> 6 (c5)	4.58	4.40	4.75	4.45	4.67
7-> 5 (c5)	6.42	7.00	4.50	6.82	4.50
6) rec. Count	445	344	437	478	468
6) num. P5	87%	92%	88%	88%	86%
6-> 5 (p5)	7.15	7.16	7.17	7.20	7.13
6-> 5 (c5)	4.48	4.50	4.37	4.48	4.36
6-> 4 (c5)	5.37	4.82	5.43	4.71	5.58
5) rec. Count	3 098	2 423	3 001	3 366	3 180
5) num. P5	88%	88%	88%	88%	87%
5-> 4 (p5)	6.85	6.87	6.86	6.88	6.85
5-> 4 (c5)	4.32	4.49	4.36	4.44	4.42
5-> 3 (c5)	4.66	4.14	4.47	4.20	4.27
4) rec. Count	20 593	16 067	19 974	22 429	21 168
4) num. P5	88%	89%	88%	87%	90%
4-> 3 (p5)	6.64	6.64	6.64	6.66	8.62
4-> 3 (c5)	4.30	4.40	4.35	4.43	4.78
4-> 2 (c5)	4.15	3.82	3.96	3.69	7.17
3) rec. Count	132 828	103 942	128 819	144 666	176 086
3) num. P5	89%	89%	88%	92%	88%
3-> 2 (p5)	6.47	6.49	6.48	8.56	8.44
3-> 2 (c5)	4.27	4.36	4.33	4.70	4.77
3-> 1 (c5)	3.78	3.54	3.65	7.25	7.60
2) rec. Count	837 768	656 204	811 258	1 205 496	1 423 504
2) num. P5	89%	89%	92%	87%	85%
2-> 1 (p5)	6.34	6.35	8.54	8.35	8.32
2-> 1 (c5)	4.24	4.35	4.67	4.72	4.76
2-> 0 (c5)	3.46	3.24	7.31	7.59	7.53
1) rec. Count	5 179 068	4 063 350	6 718 488	9 586 452	11 219 093
1) num. P5	89%	88%	82%	81%	80%
1-> 0 (p5)	6.23	8.45	8.28	8.16	8.13
1-> 0 (c5)	4.24	4.62	4.75	4.73	4.74
1->-1 (c5)	0.00	0.00	0.00	0.00	0.00
0) rec. Count	31 474 045	32 779 241	51 882 362	73 193 537	85 395 964

Table 4.6.: performance of EditToP5C5Free and number of recursive calls (part 2)

thres	6	7	8	9	10
time (secs)	790	990	1200	1300	1300
rec. Count	126 550 382	141 262 068	157 986 259	176 078 647	179 697 059
9) rec. Count	1	1	1	1	1
9) num. P5s	100%	100%	100%	100%	100%
9 -> 8 (p5)	9	9	9	9	9
9-> 8 (c5)	-	-	-	-	-
9-> 7 (c5)	-	-	-	-	-
8) rec. Count	9	9	9	9	9
8) num. P5s	89%	78%	78%	100%	100%
8-> 7 (p5)	8.13	8.00	8.00	8.67	8.89
8-> 7 (c5)	4.00	4.00	4.00	-	-
8-> 6 (c5)	10.00	10.00	10.00	-	-
7) rec. Count	69	64	64	78	80
7) num. P5s	86%	92%	95%	95%	89%
7-> 6 (p5)	7.54	7.47	8.75	8.76	8.73
7-> 6 (c5)	4.60	4.40	5.00	5.00	4.89
7-> 5 (c5)	5.90	6.20	6.33	8.25	8.22
6) rec. Count	501	483	569	668	664
6) num. P5	88%	89%	87%	86%	89%
6-> 5 (p5)	7.17	8.76	8.66	8.66	8.53
6-> 5 (c5)	4.60	4.91	4.88	4.79	4.87
6-> 4 (c5)	4.62	7.43	7.88	8.10	8.03
5) rec. Count	3 497	4 053	4 673	5 465	5 458
5) num. P5	91%	87%	86%	83%	87%
5-> 4 (p5)	8.62	8.58	8.53	8.49	8.44
5-> 4 (c5)	4.81	4.83	4.88	4.86	4.81
5-> 3 (c5)	7.06	7.51	7.72	7.68	8.04
4) rec. Count	29 199	33 179	37 948	43 810	44 181
4) num. P5	88%	85%	84%	82%	86%
4-> 3 (p5)	8.47	8.47	8.42	8.37	8.35
4-> 3 (c5)	4.80	4.82	4.83	4.80	4.74
4-> 2 (c5)	7.58	7.66	7.74	7.76	7.83
3) rec. Count	236 479	266 853	302 792	346 285	351 526
3) num. P5	86%	84%	83%	82%	85%
3-> 2 (p5)	8.36	8.34	8.29	8.24	8.22
3-> 2 (c5)	4.78	4.79	4.78	4.76	4.70
3-> 1 (c5)	7.61	7.76	7.72	7.72	7.77
2) rec. Count	1 884 178	2 112 515	2 380 388	2 697 320	2 748 886
2) num. P5	83%	82%	82%	81%	83%
2-> 1 (p5)	8.22	8.19	8.15	8.10	8.08
2-> 1 (c5)	4.73	4.73	4.73	4.71	4.65
2-> 0 (c5)	7.48	7.53	7.48	7.47	7.48
1) rec. Count	14 624 188	16 346 945	18 343 948	20 590 473	21 018 940
1) num. P5	79%	79%	79%	79%	80%
1-> 0 (p5)	8.03	8.01	7.97	7.91	7.89
1-> 0 (c5)	4.70	4.69	4.68	4.66	4.62
1->-1 (c5)	0.00	0.00	0.00	0.00	0.00
0) rec. Count	109 772 261	122 497 966	136 915 867	152 394 538	155 527 314

Assume $H := \{e_1, \dots, e_{|H|}\}$ where the e_is are enumerated in the order as they were applied by the heuristic algorithm. The subsets that we tried to use to apply the described optimization method, were consecutive subsets of the form

$$S := S_{i,k+1} := \{e_j \mid i \leq j \leq i + k + 1\}$$

We started with $i := |H| - (k + 1)$. After each failed attempt we decreased i by about half of the size of the subset. When for some subset the above described optimization method succeeded and gave us a new editing set H_2 then we started again with $i := |H_2| - (k + 1)$. If we reached $i := 0$ and also there was no success with $S_{0,k+1}$ then we incremented the value of k and started the same procedure again. Initially we started with $k := 1$ and we stopped when we the running time needed after incrementing k another time would exceed a certain threshold.

By a similar method we also tried to optimize the deletion sets found by the heuristic deletion algorithm using DeleteToP5C5Free. The results of optimizing the deletion sets to obtain better deletion sets can be found in Table 4.7. The results of optimizing the editing sets can be found in Table 4.8.

As every deletion set is also an editing set we can also use the *deletion* sets found by the heuristic deletion algorithms as a starting point of the optimization in order to find better *editing* sets. Since the heuristic deletion algorithm performed actually better than the heuristic editing algorithm as seen in Section 3.5, this might sometimes even be a better starting point. The results of optimizing the *deletion* sets to better *editing* sets can be found in Table 4.9.

The second column of Tables 4.7 4.8 and 4.9 shows the number of edits needed by the initial editing set which was calculated by the heuristic algorithm. The third column shows the number of edits needed by the final optimized editing set H'. An entry of the form $x+y = z$ means that the editing set contains x deletions and y inserts and that this makes a total of z edits. The fourth column shows the maximums size $|S| = k+1$ of subsets which we used for the optimization in the above described way. Note that we only stopped the optimization process at points in time when the most recent optimization attempts failed for each of the subsets $S_{i,k+1} \in H'$, where $i = |H|-(k+1), |H|-(k+1)-\lfloor k+1 \rfloor, |H|-(k+1)-2\lfloor k+1 \rfloor, \dots, 0$. So while we don't know if the whole optimized editing (deletion) set H' is minimal, we know at least that these subsets $S_{i,k+1}$ are minimal editing (deletion) sets for the graphs $G_{H' \setminus S_{i,k+1}}$ that remain after applying the edits in $H' \setminus S_{i,k+1}$.

Note that in the case of the graph karate in Table 4.7 the value in column four is the equal to the size of the whole deletion set that we found. This means that an optimization attempt with the non-proper subset $S = H'$ did not find any better solution. In other words we verified that the found solution for the deletion problem is actually minimal in the case of karate.

Table 4.7.: heuristic deletion + exact deletion

name	initial #deletions	optimized #deletions	#verified edits	exact time
karate	13	13	13	97.6 s
lesmis	153	50	13	619.8 s
dolphins	62	57	14	917.2 s
grass_web	43	32	13	710.4 s
football	190	186	14	1156.5 s

Table 4.8.: heuristic editing + exact editing

name	initial #edit	optimized #edits	#verified edits	exact time
karate	10 + 8 = 18	6 + 9 = 15	10	928.1 s
lesmis	66 +227 =293	33 + 19 = 52	9	425.3 s
dolphins	48 + 6 = 54	48 + 6 = 54	10	645.1 s
grass_web	19 + 9 = 28	19 + 8 = 27	9	568.8 s
football	181 + 9 =190	179 + 7 = 186	9	303.0 s

Table 4.9.: heuristic deletion + exact editing

name	initial #deletions	optimized #edits	#verified edits	exact time
karate	13	13 + 0 = 13	10	801.4 s
lesmis	153	39 + 10 = 49	9	298.8 s
dolphins	62	50 + 5 = 55	10	775.1 s
grass_web	43	17 + 8 = 25	9	322.3 s
football	190	186 + 0 = 186	9	322.6 s

4.4. Community Structure

In this section we will visualize the editing sets calculated in Section 4.3. In doing so we will try to find out whether the connected components of the resulting graphs after applying the editing sets actually represent a sensible grouping of the vertices into communities. We will also look at the differences between the grouping gained by (P_5, C_5)-free editing and by (P_5, C_5)-free deletion. Therefore we will always look at a deletion set and at an editing set. (except when the best editing set is equal to the best deletion set as in the case of karate) Note that we have two editing sets for each graph - one obtained by combining heuristic editing with exact editing and one by combining heuristic deletion with exact editing. We will always look at the smaller editing set. While we calculated the editing sets using our own implementation of the methods described in the previous chapters, for the visual representation and for the layout of the graphs we used the help of a program called visone[1].

Our implementation of the editing algorithms wrote the resulting graph to a file using the DOT format. This file could than be opened by visone. The dot file contained the information which of the vertices were adjacent in the original graph, which of the edges in the original graph were deleted by the calculated editing set and between which vertices the editing set deleted edges. With the help of visone we added some additional information like for example the ground truth communities in the case of the graphs karate and football and we presented these informations visually for example by using colours. We also used colours to distinguish deleted and inserted and remaining (not edited) edges.

Furthermore as mentioned above we needed to construct a layout for each graph. That is we needed to place the vertices at sensible positions on the plane such that the resulting connected components after applying the editing set can easily be identified by the reader. We also tried to order the connected components in a way such that components with many deleted edges between them are placed more closely to one another. Furthermore we tried to minimize the number of intersections between edges and between edges and vertices. We used the help of an algorithm offered by visone, but we also did many manual optimizations by dragging vertices with the mouse pointer around.

[1]http://visone.info/index.html

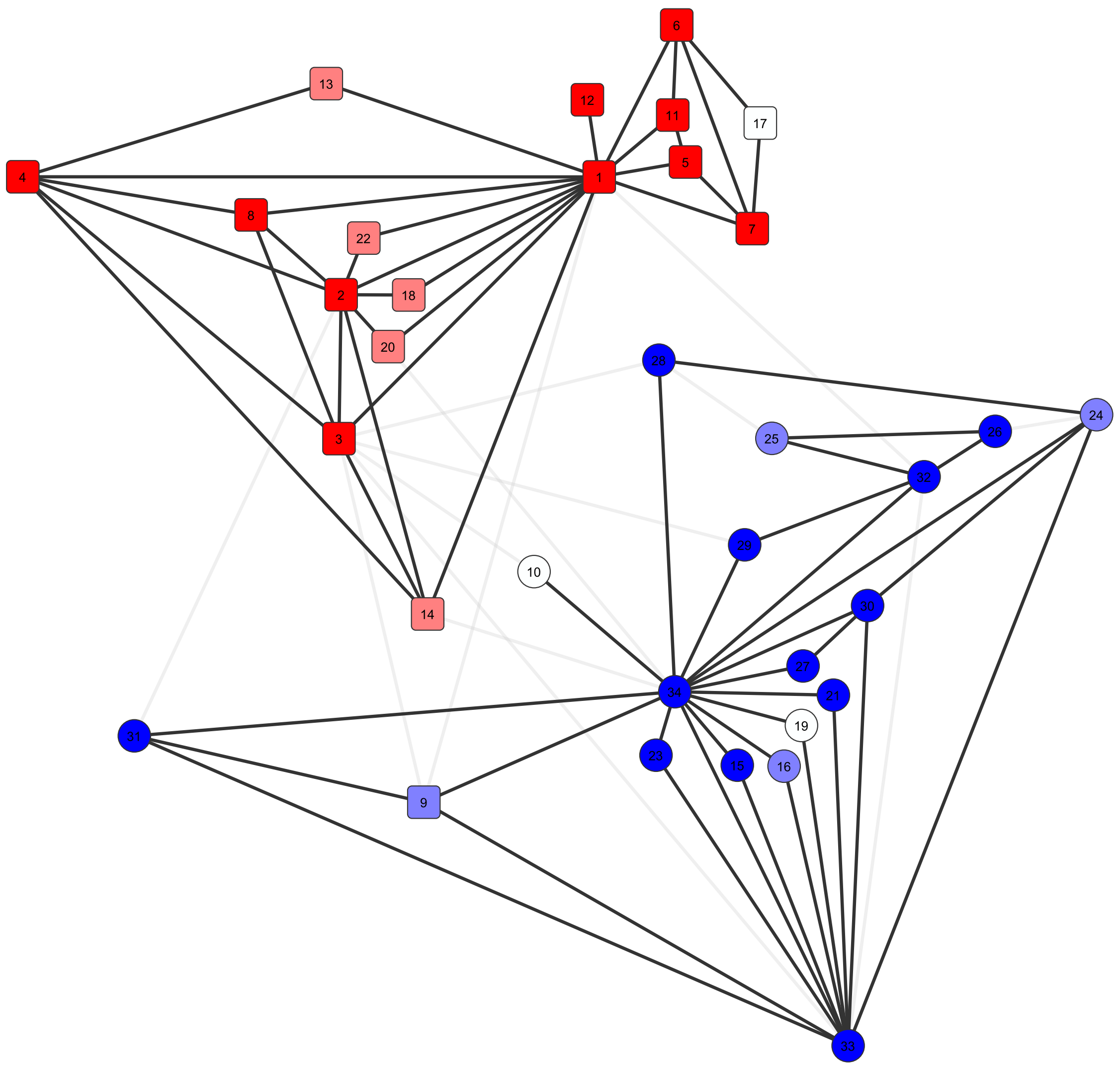


Figure 4.1.: karate heuristic deletion, could not be optimized by exact deletion or exact editing algorithm

4.4.1. Zachary's karate club (short karate)

For the graph karate the editing set $F_{del,edit}$ found by the combination of the heuristic *deletion* algorithm and the exact editing algorithm is smaller than $F_{edit,edit}$ the one found by the combination of the heuristic *editing* algorithm and the exact editing algorithm. It is 13 vs. 15, see Table 4.9 and Table 4.8. Furthermore we can see that the values of the second and the third column in Tables 4.7 and 4.9 are equal, they are both 13. That means that the deletion set F_{del} found by the heuristic deletion algorithm could not be optimized by either the exact editing algorithm nor by the exact deletion algorithm. Thus we have $F_{del,edit} = F_{del} = F_{del,del}$. That means the best editing set that we found for karate is equal to the best deletion set.

Figure 4.1 shows the graph karate together with this best deletion set which is also the best editing set. The corresponding deleted edges are coloured in a brighter colour. Before we explain the meaning of the colouring of the vertices in Figure 4.1 we first need to understand where this graph stems from.

The initial karate graph - as it looks before applying the deletion set - stems from [Zac77]. It shows the interactions between students of an university karate club. The club suffered from a conflict between the club president represented by vertex 34 and the karate instructor who corresponds to vertex number 1. Eventually the conflict led to a split of the club into two karate clubs. The graph represents the situation shortly before the split. [Zac77] showed that by looking for a minimal cut in a weighted version of the graph one could group the club's members into supporters of the club president and supporters of the karate instructor regarding the conflict. Thereby one could almost accurately have predicted which members would stay in the karate club led by the president and which will go to the newly formed organization headed by the instructor after the split.

In Figure 4.1 we coloured the supporters of the instructor (vertex 1) red and the supporters of the club president (vertex 34) blue. Weak supporters are coloured in a brighter colour and members that were neutral to the conflict are coloured white. (See [Zac77, p. 465]) Members that joined the club of the instructor after the split are depicted by a rectangle, members that stayed with the president are depicted by a circle. Note that our algorithm almost correctly separated the "rectangles" from the "circles". The only problem is vertex 9. However while number 9 joined the club of the instructor he is actually a (weak) supporter of the club president. [Zac77] explains it like this: "[...] he was only three weeks away from a test for black belt (master status) when the split in the club occurred. Had he joined the officers' club [=the club of the club president, vertex 34] he would have had to give up his rank and begin again in a new style of karate with a white (beginner's) belt, since the officers had decided to change the style of karate practiced in their new club. Having four years of study invested in the style of Mr. Hi [=the instructor, vertex 1], the individual could not bring himself to repudiate his rank and start again." So one could argue that vertex 9 is not actually misclassified as the student actually wanted to join the club of the instructor but just could not do so for technical reasons. [NG13] also classified vertex 9 as belonging to vertex 34 using (P_4, C_4)-free editing. And [GN02] did so as well using a different clustering method. These papers however did not mention the vertex 9 to be misclassified. Also [Zac77]s min-cut method puts vertex 9 into one group with vertex 34 (using a weighted version of the graph). Interestingly our method did not misclassify vertex 3, which was misclassified in [GN02] and [NG13].

So as a summary we conclude that the (P_5, C_5)-free editing was very successful on the graph karate.

4.4.2. Les Misérables (short lesmis)

Les Misérables is a 19-th century novel by Victor Hugo. [KKK93] constructed a network where the vertices are characters of the novel. Two vertices are adjacent if the corresponding characters exist in a chapter together. We call this network short "lesmis".

For this graph the editing set found by the combination of the heuristic deletion algorithm and the exact editing algorithm $F_{del,edit}$ was better than $F_{edit,edit}$ - the one found by the combination of the heuristic editing algorithm and the exact editing algorithm. (See Tables 4.8 and 4.9) Therefore we will concentrate on $F_{del,edit}$ here. Furthermore the best editing set that we found is actually better than the best deletion set $F_{del,del}$ that we found. The former one needs only 49 edits while the latter one needs 50 deletes. (see Tables 4.7 and 4.9). Therefore it makes sense to compare the communities obtained by the deletion version with the ones obtained by the editing version here.

Figure 4.2 and Figure 4.3 both depict the graph lesmis. Figure 4.2 shows the edges that are deleted by $F_{del,del}$ in a brighter colour. Figure 4.3 shows the edges that are deleted by $F_{del,edit}$ in a brighter colour. Edges inserted by $F_{del,edit}$ are coloured red. For lack of space we did not write down the real names of the characters represented by the vertices. Instead we just enumerated the vertices. You can find the real names of the characters in Table A.5 in the appendix.

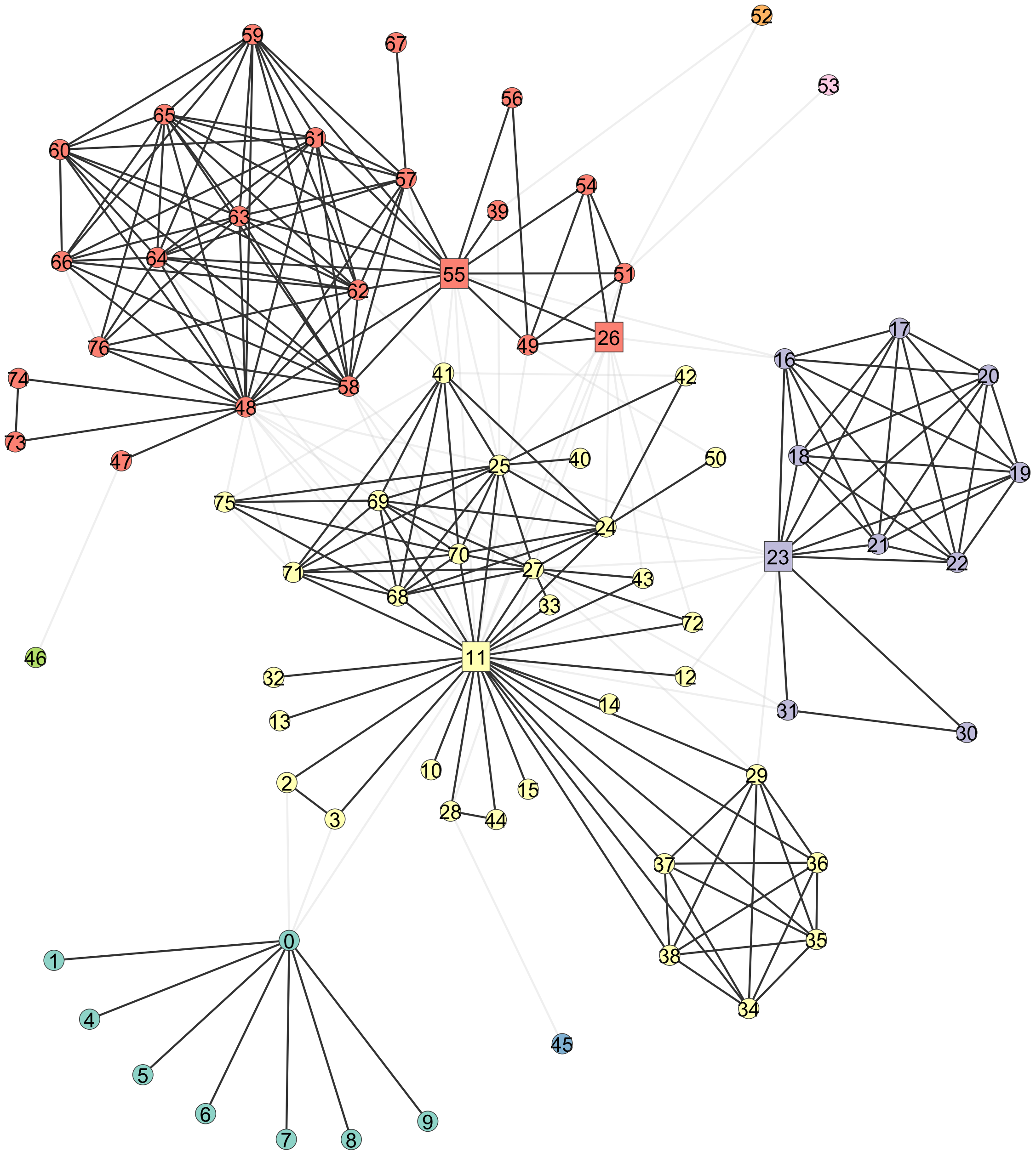


Figure 4.2.: lesmis heuristic deletion + exact deletion

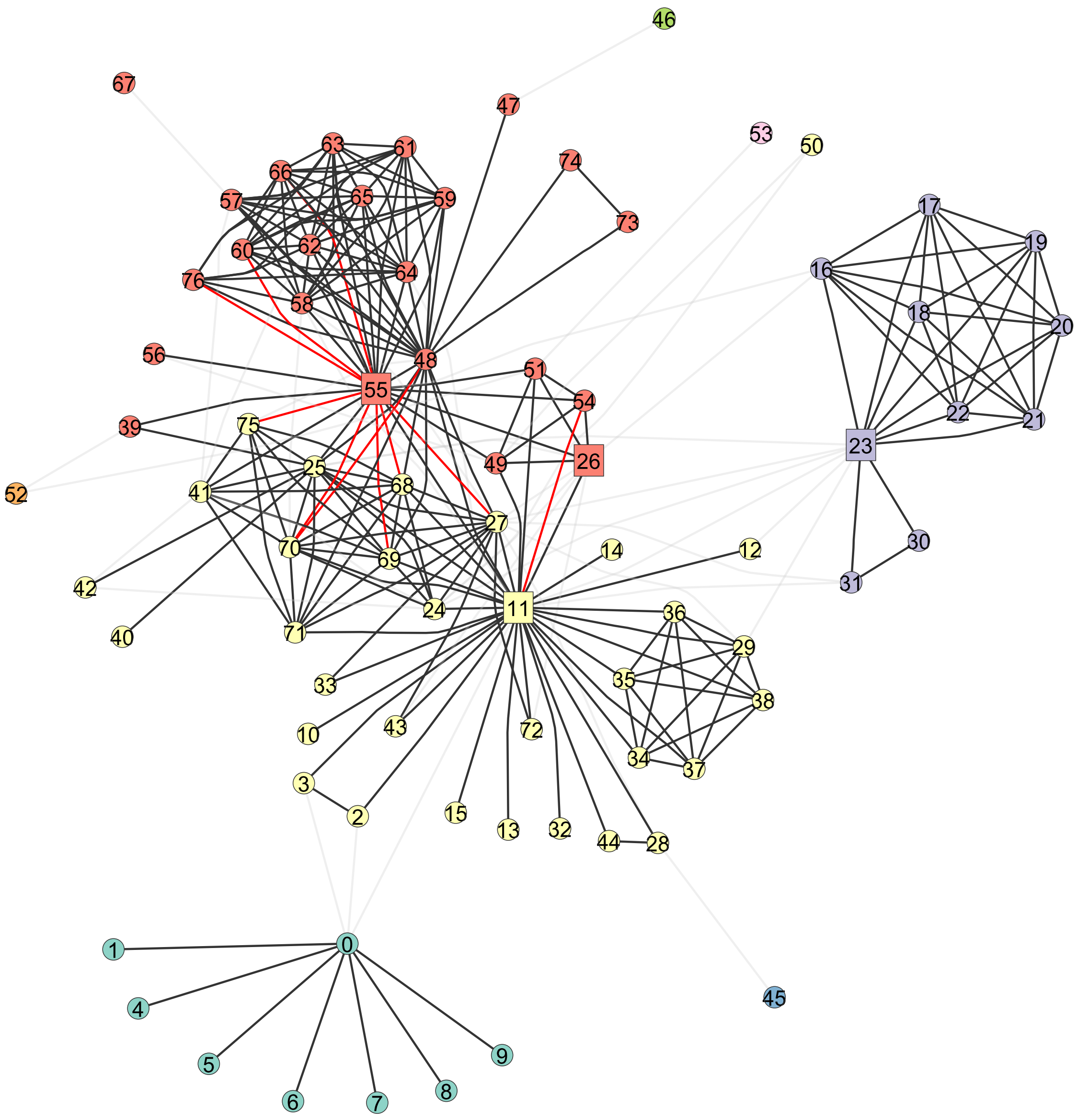


Figure 4.3.: lesmis heuristic deletion + exact editing

The ground-truth communities are not known for this graph. Thus unlike for the karate graph the colours that we used for the vertices do not indicate any ground-truth communities. We just used this colouring in order to compare the deletion version Figure 4.2 and the editing version Figure 4.3. In Figure 4.2 we coloured the vertices according to their connected component after applying $F_{del,del}$. Then in Figure 4.3 we coloured the vertices in the same colour as in Figure 4.2. Using this colouring if one looks closely, one can see that vertex 67 in the top left corner of Figure 4.3 does not belong to the same connected component as it does in Figure 4.2. You can see this as it is not connected to the other vertices of the same red colour by any remaining edge. By "remaining" edge we mean an edge which is not deleted by the editing set, that is it is not coloured in the brighter grey colour. Also vertex 50 on the right side is not connected to the yellow component in Figure 4.3. Except for vertex 67 and vertex 50 the big connected component of the editing version is the union of the red and the yellow components of the deletion version. The remaining components are the same for the editing and for the deletion version.

The novel Les Misérables consists of 5 parts. Four of them are named after a character. We depicted the corresponding four vertices by rectangles. Note that the editing algorithm strongly increased the degree of vertex 55 (see the red inserted edges). So in some sense we can say that the editing algorithm noticed that vertex 55 is important, though we did not tell it that vertex 55 actually appears in the title of one of the books.

When analysing the results of Figure 4.2 and Figure 4.3 we found that there are other equivalent solutions to the (P_5, C_5)-free editing problem that induce slightly different communities. By equivalent solution we mean a (P_5, C_5)-free editing set that contains an equal number of edits. So apparently the theoretical possibility, that the communities obtained by (P_5, C_5)-free editing are not completely well defined, actually occurs quite often in practice. We found that (at least) vertices 2, 3, 12 and 47 can as well be assigned to a different community. For example look at the vertices 2 and 3 on the lower left side of the yellow coloured main character 11 in Figure 4.2. If we delete the edges $\{2, 11\}$ and $\{3, 11\}$ then we don't need to delete the edges $\{2, 0\}$ and $\{3, 0\}$ anymore as the vertex 0 is already disconnected from the yellow group. Note that the newly formed component consisting of vertices 0, its neighbours 1 and 2 - 9 and the newly added vertices 2 and 3 does not contain any P_5 or C_5. A P_5 requires 3 vertices of a degree greater than one. And a C_5 requires even more of them. Only vertices 0, 2 and 3 have a degree greater than 2 in this newly formed community. But they already induce a C_3. Obviously as the yellow group was (P_5, C_5)-free before it will still be (P_5, C_5)-free without vertices 2 and 3. (If we take away some vertices of any (P_5, C_5)-free graph the resulting graph will still be (P_5, C_5)-free. In other words (P_5, C_5) freeness is a hereditary property.) So vertices 2 and 3 may as well be assigned to the cyan community instead of the yellow community. Note that the same changes can be made in the case of the editing version Figure 4.3 as well. In a similar way the reader may prove to himself that we can delete the edge $\{11, 12\}$ instead of $\{12, 23\}$. Thereby we can assign vertex 12 to the violet community instead of the yellow one. If we delete edge $\{47, 48\}$ instead of $\{47, 46\}$ then we can group vertex 37 together with vertex 46 instead of the red community. These changes to vertices 12 and 47 work in the deletion version Figure 4.2 and in the editing version Figure 4.3 as well.

It is difficult to tell whether the communities detected by our algorithms are correct, as there are no known ground truth communities here. What might be a bit problematic in the case of the deletion version is the main character represented by vertex 26. It has six neighbours in the yellow community but only four neighbours in its own red community. So we might wonder if it would actually be better grouped together with the yellow community here. In the case of the editing version this problem does not exist as the yellow and the red community were merged into one community anyway.

However apart from that problem the communities look quite convincing. For example most of the vertices in the non-trivial communities except for vertex 26 have more neighbours that belong to the same community than neighbours that belong to different communities. A indication that it makes sense to assign these vertices to the community to which they were assigned here. By non-trivial communities we mean communities consisting of more than one vertex.

[NG13] noted that (P_4, C_4)-free editing correctly separated only unimportant characters into trivial groups. The vertices that were separated into single vertex groups by (P_5, C_5)-free editing and (P_5, C_5)-free deletion have a low degree. Furthermore they are not connected to any main character. These are both indications that they are unimportant too. Note that there are many degree one vertices connected to the main character vertex 11. Correctly none of them were isolated into trivial groups by either the deletion version nor by the editing version.

What could be criticized about the communities especially in case of the editing version is that they are too big. Therefore while they might be correct, they don't give us so much information as we might get from a more finely divided grouping. For example on the one hand it might be correct that a certain character in the big component of Figure 4.3 is more closely related to the main characters represented by vertices 55, 26 and 11 than he is related to vertex 23. However on the other hand we might also be interested in the question whether the character is more closely related to vertex 55 than he is related to vertex 11. Also note that the vertices 29, 36, 35, 34, 38 and 37 in the clique on the lower right side of the yellow community seem to have a stronger connection to each other and to vertex 11 than to the rest of the their community. The communities obtained by our approach do not tell us anything about this internal structure of the found communities.

As a summary we conclude that the results obtained by (P_5, C_5)-free deletion and (P_5, C_5)-free editing for the graph lesmis make sense but they give us too few information about the structure of the graph due to the big size of the identified communities.

4.4.3. Dolphins

The dolphins graph shows the social network of bottlenose dolphins living in some fjord in New Zealand. Two dolphins are joined by an edge if they were seen together a statistically significant number of times. It was created by [Lus03].

Figure 4.4 and Figure 4.5 both show the dolphin network. Figure 4.4 also shows the (P_5, C_5)-free deletion set that we found by combining the heuristic deletion and the exact deletion algorithm with each other. The deleted edges are depicted in a brighter colour. For the dolphins graph the best editing set that we found is actually the one that was calculated using the heuristic *editing* algorithm together with the exact editing algorithm. As shown in Table 4.8 and Table 4.9 it needed only 54 edits while the one which was calculated using the deletion heuristic needed 55 edits. Figure 4.5 depicts the former editing set. The deleted edges are coloured in a brighter colour and the inserted edges are coloured red.

As in the case of lesmis but unlike in the case of karate the colouring that we used for the vertices of dolphins does not represent any ground truth communities in the graph. It is only there to help compare the results of the deletion version with the results of the editing version. As in the case of lesmis the colours indicate the connected components of the deletion version. Using this colouring one can see that two of the three single vertices of the deletion version were grouped together with some of the larger communities in the editing version. Furthermore the vertex SN100 is in a different community in the editing version. Apart from that the communities are the same for both versions.

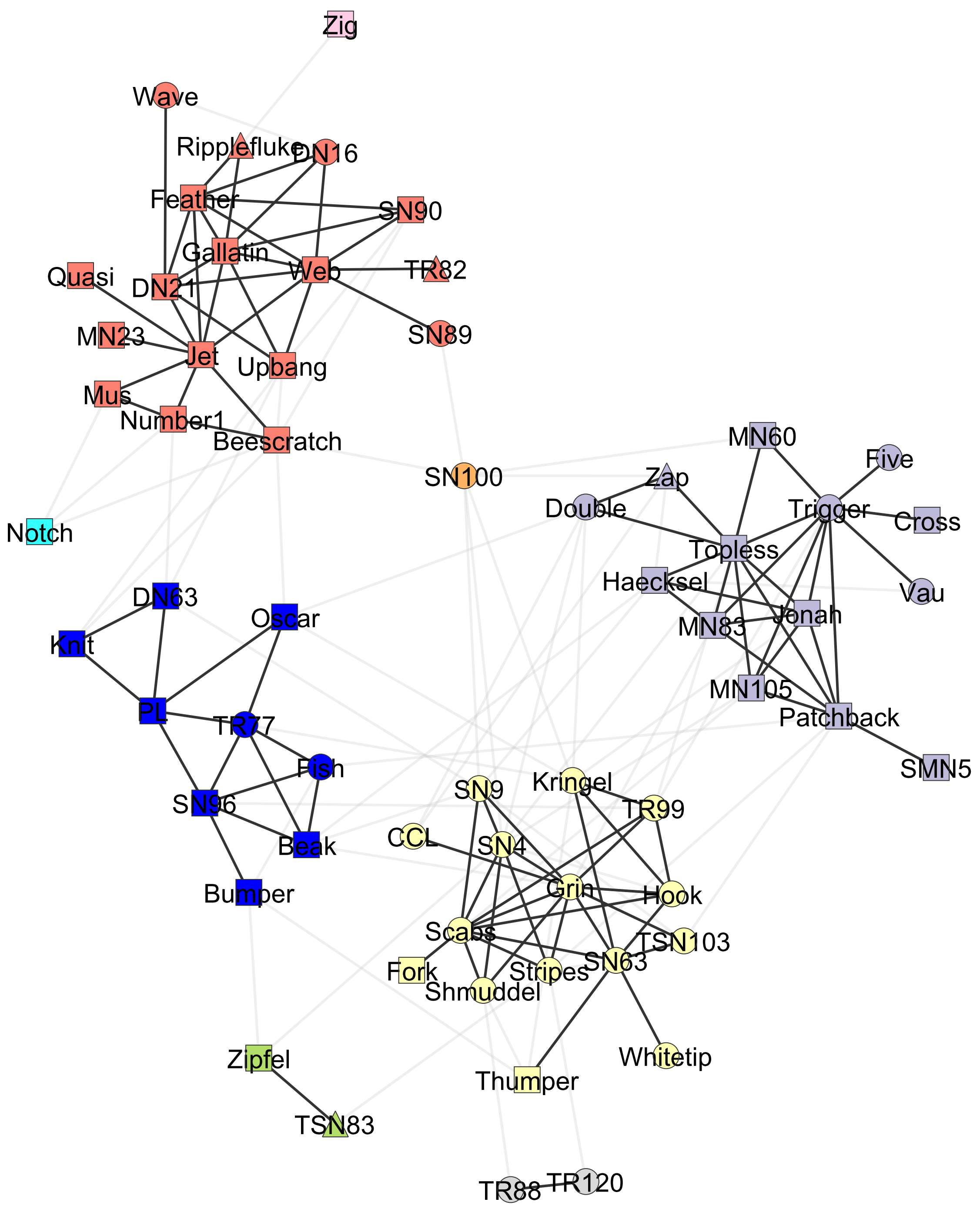


Figure 4.4.: dolphins heuristic deletion + exact deletion

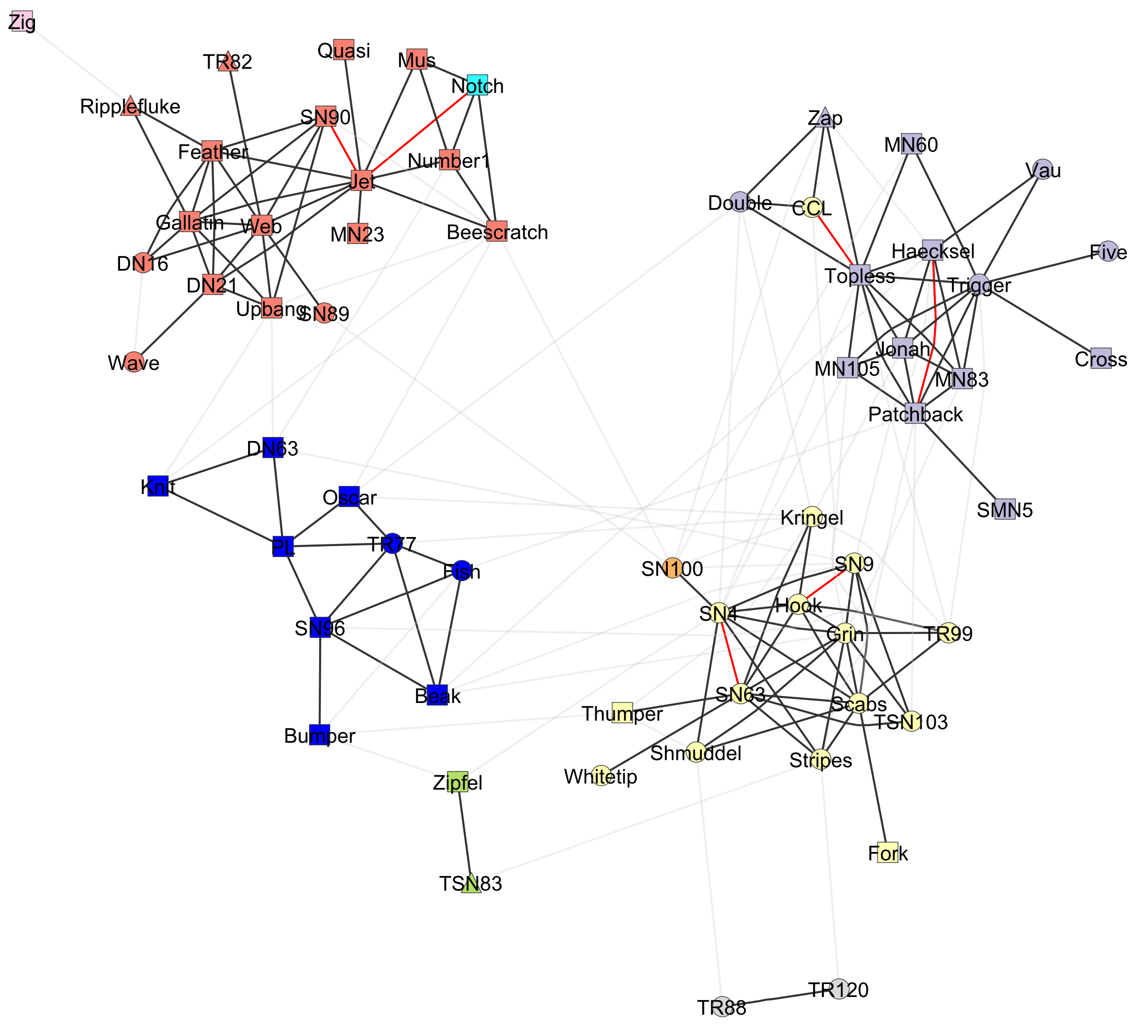


Figure 4.5.: dolphins heuristic editing + exact editing

Like [NG13] we depicted the females by circles and the male dolphins by a rectangles. Dolphins of unknown gender are depicted by triangles. Like [NG04] and [NG13] we also found that there is one community which consists mainly of females - the yellow one in our case.

As in the case of lesmis we found that there are a vertices which can equally well be assigned to different communities. For example the vertex Bumper on the lower side of the blue community. If we remove the edge from SN96 to Bumper then we do not need to delete the edge from Bumper to Zipfel. Therefore we get a similarly sized correct editing set or deletion set that assigns Bumper to the green community. Note that this is true for the deletion and for the editing version as well. In the case of the deletion version SN89 may be grouped together with SN100 instead of the red community. This ambiguity does not occur in the case of the editing version as SN100 is grouped together with the yellow group. Connecting SN89 with SN100 would cause a P_5 consisting of SN89, SN100, SN4 Scabs and Fork here. Furthermore the red vertices Mus, Number 1 and Beescratch which are adjacent to the blue vertex Notch can be assigned differently in the deletion version. If we delete the three edges from these three vertices to the vertex Jet then we will not need to delete three edges to the vertex Notch. Thereby we get a similarly sized editing set that assigns Mus, Number 1 and Beescratch to Notch instead of the red community. This ambiguity however does not occur in the editing version as Notch is connected to the red community here.

As in the case of lesmis we need to evaluate the community structure without having any ground truth communities to compare. For a large portion of the vertices the number of neighbours that lie in the same community is bigger than the number of neighbours that belong to different communities. This is an indication that they were assigned correctly. However there are also a few vertices that have many neighbours in other communities. We want to take a look some of those critical vertices.

The vertex double was assigned to the violet group by the deletion algorithm although it has only two neighbours in this group Tap and Topless. However it has three neighbours in the yellow group CCL, SN4 and Kringel. In the editing version CCL was moved to the violet clique. This justifies the assignment of Double a bit better as he has at least two neighbours in his community and only two neighbours in the yellow group. The vertex CCL itself has only one neighbour in the yellow group but two neighbours in the violet community. However on the other hand the one neighbour in the yellow group is more central two this group as indicated by its high degree of 10. Therefore both assignments of CCL make sense. Therefore we can not tell which assignment is better in this case the one of the deletion version or the one of the editing version. The vertex Knit on the left side of the blue has an equal number of neighbours in the red community as he has neighbours in his own blue community. The same holds for DN63.

The vertex Oscar in the blue group has only two neighbours in his own community but three neighbours in foreign communities. However note that each of the foreign neighbours belongs to a different community. Therefore the blue community is still the community with the maximum number of neighbours of Oscar. So Oscars assignment still makes sense.

Note that the algorithm also deleted edges between vertices of the same community. For example when we look at Kringel we might think at first sight that it has few connections to its own yellow community as there are only two remaining edges emanating from it. However if one looks closely one can see that two of the deleted edges emanating from Kringel also go to vertices of the same community in the deletion version. In the editing version three of the deleted edges go to vertices of the same component. In the deletion version four of the nine neighbours belong to the same community, in the editing version five of them belong to the same community. Also note that like in the case of Oskar the

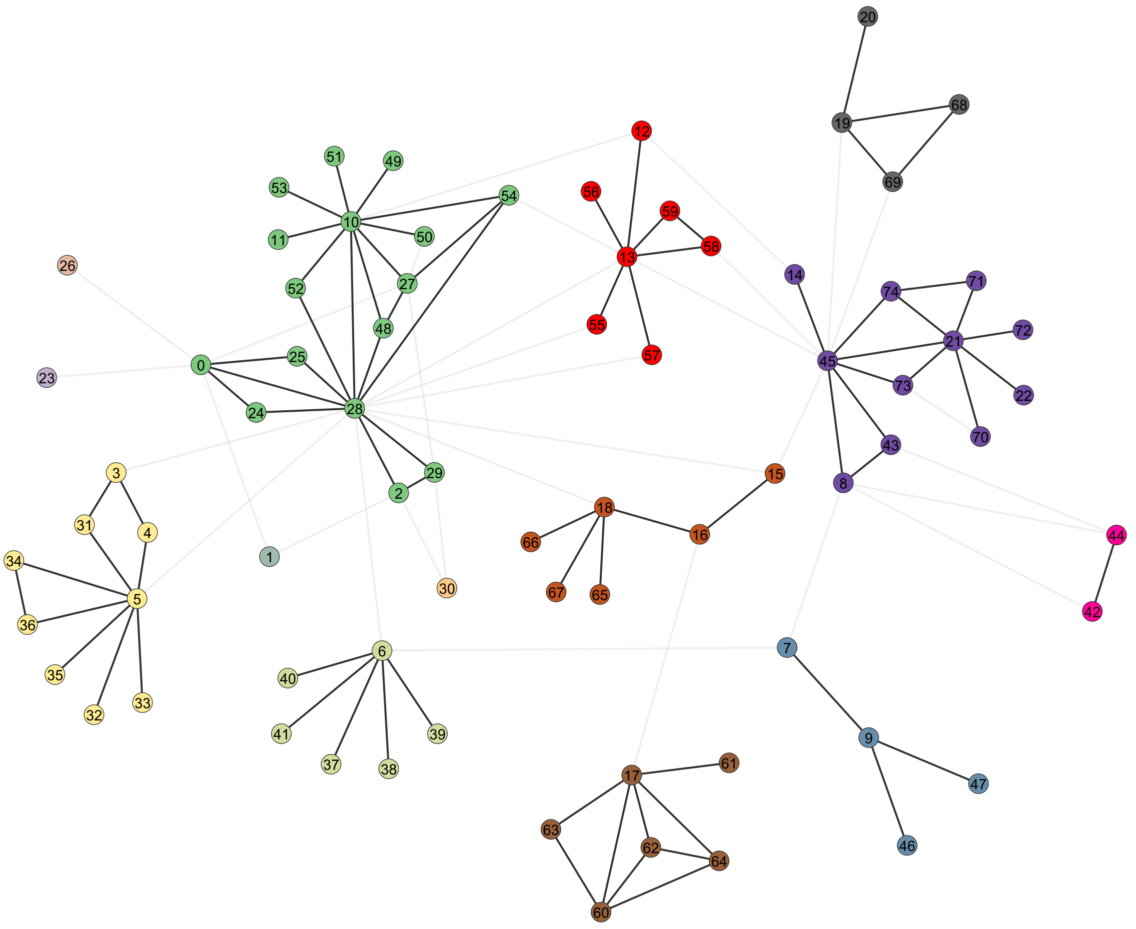


Figure 4.6.: grass_web heuristic deletion + exact deletion

foreign neighbours of Kringel do not all belong to the same foreign community. It is not 4 vs. 5 but 4 vs. 2 vs. 2. vs. 1 in the case of the deletion version. In the editing version it is 5 vs. 2 vs. 2. So assigning Kringel to the yellow community makes sense. If we analyse all vertices of the graph like this we get the impression that the communities defined by our algorithm are acceptable.

4.4.4. Grass_Web

This graph shows describes the food web between some grassland species. It was built in [DHC95].

Figure 4.6 and Figure 4.7 both show the graph grass_web. Figure 4.6 shows the (P_5, C_5)-free deletion set that we found using the combination of the heuristic deletion algorithm and the exact editing algorithm. The deleted edges are coloured in a brighter colour. The editing set that we found using the combination of the heuristic *deletion* algorithm and the exact editing algorithm is smaller than the one that we found using the combination of the heuristic *editing* algorithm and the exact editing algorithm. As shown in Table 4.8 and Table 4.7 the former one needs only 25 edits and the letter one needs 27 edits. Therefore we will concentrate on the former one here. It is shown in Figure 4.7. The corresponding deleted edges are coloured in a brighter colour and the inserted edges are coloured red.

Like for the graphs lesmis and dolphins the vertex colours are there to help compare the deletion and the editing version. Again the colours represent the communities found by the deletion version. Using this colouring we can see that the big community of Figure 4.7 contains the union of the “green” and the “red” community of Figure 4.6. Furthermore it

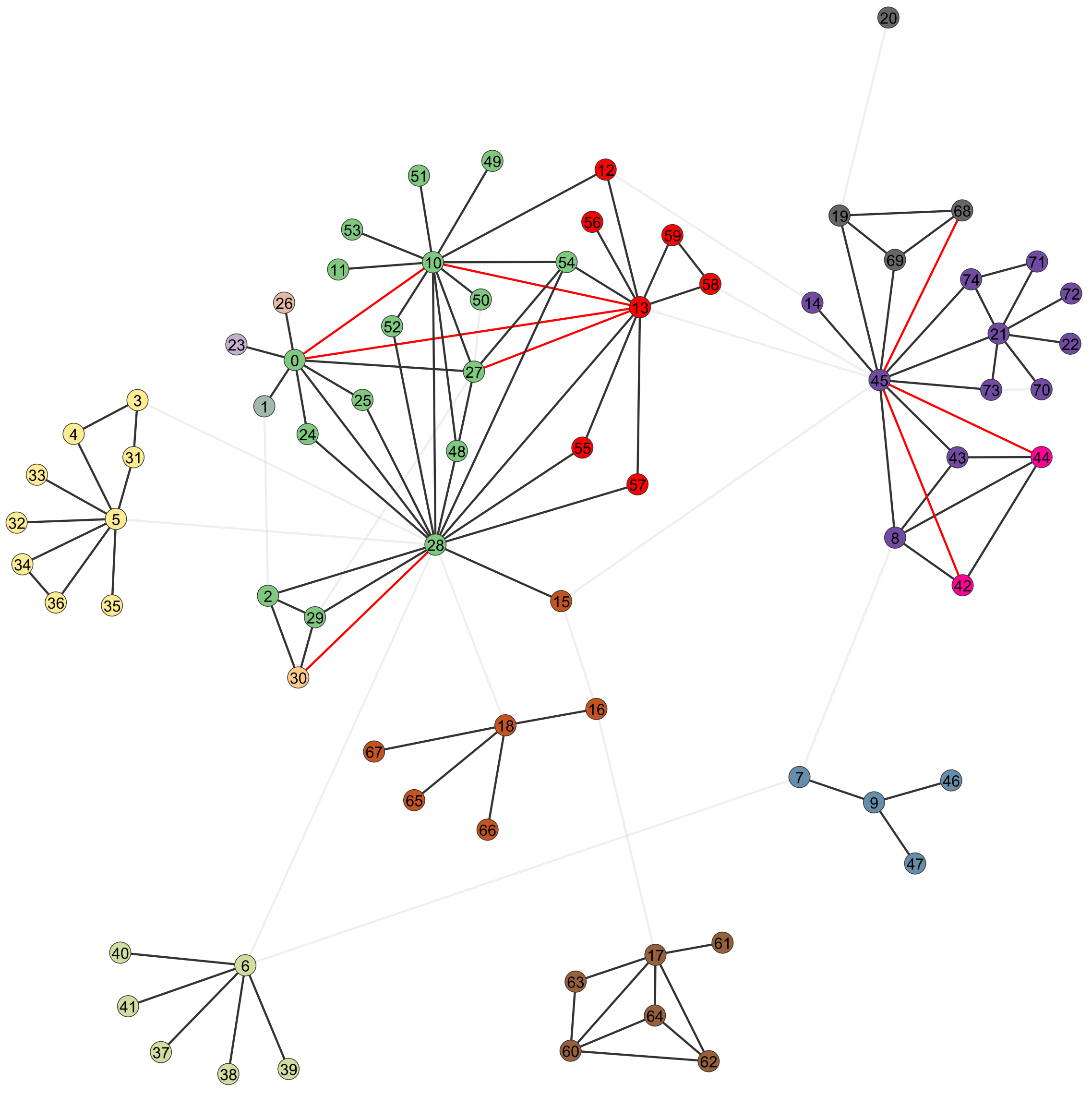


Figure 4.7.: grass_web heuristic deletion + exact edit

contains four vertices that were single vertex groups in Figure 4.6 and it contains vertex 15 which belonged to a different community in Figure 4.6. The second largest community of Figure 4.7 on top right corner almost corresponds to the union of three communities of Figure 4.6, except that vertex number 20 was separated into a new single vertex group.

Let us take a closer look on the above mentioned vertex 15, which was connected to the orange community in the deletion version but was connected to the green community in the editing version. Note that if we separated it from the green component and connected it to the orange component in the editing version as well, then the graph would still be (P_5, C_5)-free. In doing so we would need one additional deletion for the edge $\{28, 15\}$ but we would save the deletion of the edge $\{28, 15\}$. So the total number of edits would be the same. So we could equally well assign vertex 15 to the orange group in the editing version. In a similar way the reader may also prove to himself that we may also delete the edge $\{15, 16\}$ instead of the edge $\{28, 15\}$ in the deletion version. Thereby we could equally well assign vertex 15 to the green component in the deletion version. So the different assignment of vertex 15 in Figure 4.6 and in Figure 4.7 is actually arbitrary and does not really show a difference between the editing and the deletion version. Furthermore note that in both the deletion and in the editing version we might also separate vertex 15 from both the green and the orange community and instead leave the edge connecting it to the violet community. Note that vertex 15 would then be in a similar position as vertex 14 in the violet community. So if there was a P_5 (or C_5) in that community that includes vertex 15 and four other vertices, then the other four vertices had also induced a P_5 (or C_5) together with vertex 14. But this was not the case. So this community consisting of vertex 15 and the violet vertices would still be (P_5, C_5)-free. Furthermore note that when vertex 15 is not connected to the orange group then we could also connect vertex 16 to the brown group instead of the orange group. Another vertex that could have been assigned differently in both the editing and the deletion version is vertex 7 in the blue group. It could have been assigned to the beige group centred around vertex 6 as well. In the deletion version vertex 14 could have been assigned to the red community instead of the violet community. This different assignment however is not possible in the editing version. Here we would get P_5s for example the one induced by $14, 12, 10, 28, 29$. Furthermore in the deletion version the red vertices $12, 55$ and 57 could each have been assigned to the green community, too. Furthermore in the deletion version we might have separated vertex 0 from the green community. This would need 3 additional deletes. In exchange we might have left vertex 0 connected to the single vertices 1, 23 and 26.

We found out that the deletion set in Figure 4.6 is not optimal. There is one mistake at the lower half of the green community. When we delete the edges between the green vertices 2 and 28 and between 29 and 28 then we do not need to delete the edges $\{2, 1\}$, $\{2, 30\}$ and $\{29, 30\}$. So we save $3 - 2 = 1$ edits. There is another mistake at the violet community. If we separated vertices 8 and 43 from the violet community then we would need 2 additional deletes. However then we would not need to delete the edges between these two vertices and the pink community. Thereby we would save 3 deletes. Thus this would save us a total of $3 - 2 = 1$ deletes.

Except from all these ambiguities the results look quite convincing: Except for these above mentioned vertices whose assignment to the communities is ambiguous, most of the vertices have much more neighbours in their own communities than they have neighbours in foreign communities. This is true for both the deletion and the editing version. But the editing version is still better than the deletion version as there are fewer ambiguities than in the deletion version. Furthermore in the case of the editing version the total number of edges that connect different communities with each other is much lower than the number of edges inside of the communities.

As a summary we conclude that the result of the editing version looks very convincing. And the result of the deletion version is still acceptable but not as good as the editing version. Furthermore we found again that the theoretical possibility of not uniquely defined connected components actually appears quite frequent in practice. And apart from the community structure as a side result we found that there is room for improvement in the method that we used in order to calculate the deletion sets.

4.4.5. Football

This graph stems from [GN02]. The vertices in this graph represent teams of United States college football. Two teams are joined by an edge when they have played against each other during a particular football season. While according to [GN02] it is the 2000 season. [Eva10] writes that it actually appears to be the 2001 season. The teams are grouped into conferences. While most of the teams belong to one of these 11 conferences there are also a few independent teams. Games are more frequent between members of the same conference than between members of different conferences. A team plays about seven intraconference games and four interconference games in this season.

Interestingly for this graph the editing set $F_{del,edit}$ - calculated by the combination of the heuristic deletion algorithm and the exact editing algorithm - didn't use any insertions. Furthermore $F_{del,edit}$ is equal to $F_{del,del}$ - the deletion set found by the combination of the heuristic deletion and the exact deletion algorithm. We compared $F_{del,del}$ and $F_{del,edit}$ and found that the deletes are actually the same and it is not just the same number of deletes (i.e. $F_{del,del} = F_{del,edit}$ not only $|F_{del,edit}| = |F_{del,edit}|$). Figure 4.8 depicts the graph football together with this deletion set $F_{del,del} = F_{del,edit}$. The corresponding deleted edges are coloured in a brighter colour. The combination of the heuristic editing algorithm and the exact editing algorithm found another editing set $F_{edit,edit}$ of the same size. Figure 4.9 depicts this editing set. The corresponding deleted edges are coloured in a brighter colour. Note that there are also a few inserted edges, the ones coloured in red.

Note that unlike for the graphs lesmis, dolphins and grass_web, the colours of the vertices contain more information than just a help for the comparison between different editing/deletion sets. Here in Figure 4.8 and Figure 4.9 each vertex is coloured according to the conference to which it belongs. The grey vertices do not belong to any conference.

Note that in both versions none of the teams is separated from its original conference. However the grey-coloured vertices that do not belong to any conference were not recognized as single vertex communities. Instead they where grouped together with some other conferences. But a bigger problem is that both editing sets joined some pairs of conferences into one single community. Both editing sets connected the two conferences in the lower left corner. Additionally $F_{del,del} = F_{del,edit}$ connected the blue conference at the bottom with the orange conference at the top. And $F_{edit,edit}$ connected the orange conference to the beige one on its left side.

The graph football is the biggest one of the graphs that we studied. (see Table 4.1) Accordingly also the editing set and the deletion set is the biggest one for this graph. (see Tables 4.7, 4.9, 4.8) The question is whether our method of computing the editing sets performs worse on bigger graphs. Note for example that we tried to optimize subsets of the deletion sets of a maximum size of 9 or 10 for the editing sets of each graph. (Table 4.7) While in the case of the graph karate such a subset makes up $10/13 = 77\%$ of the whole editing set, in the case of football it is only $9/186 = 4.8\%$. Therefore intuitively we may expect that our method of calculating the editing sets yields suboptimal results for bigger graphs. Therefore the important question is if the disappointing results in the case of football are due to the method how we calculated the editing sets. Or are these problems inherent to (P_5, C_5)-free editing? Is it possible that the actual correct solution to the

(P_5, C_5)-free editing problem would separate all conferences from each other unlike our heuristic solution?

At least if we assume that the assignment of the independent teams to the conferences stays the same then the answer is no. Note that $F_{del,del}$ did not do any edits inside of the connected components. All the edits 186 in $F_{del,del}$ are therefore needed in order to separate the groups from each other. Furthermore in order to separate all conferences from each other the actual best editing set would need some additional deletes between the violet and the green conference and between the orange and the dark blue one. Therefore if the actual best editing set F_{best} would separate all conferences from each other we had $F_{best} \supsetneq F_{del,del}$. This is a contradiction since $|F_{best}| \leq |F_{del,del}|$.

The phenomenon that our approach merges several underlying ground truth communities into one community gives rise to the question if the results may still be useful if we want to learn something about the relationship between different communities. Assume we want to identify a hierarchical community structure. We then might first apply (P_5, C_5)-free editing. Then we might further subdivide the found communities into more subcommunities using a different community detection algorithm. This however only makes sense if the (P_5, C_5)-free editing algorithm does not merge the underlying subgroups in a completely random manner. Instead we hope that it merges different groups into one community only when there is some special connection between them.

To get an impression whether the merged conferences actually belong to each other in the case of the football graph we depicted the connections between the conferences in Figure 4.10. Here we represented each conference by only one vertex. The number on the vertices of Figure 4.10 represent the number of teams that belong to that conference. The number on an edge between two conferences A and B in Figure 4.10 represents the number of edges between those two conferences in the football graph. The size of the vertices and the edges correspond to the numbers the numbers that they are labelled with. Note that we did not include the vertices without conference in Figure 4.10 due to lack of space.

By looking at Figure 4.10 we can see that the merging of the dark blue and the orange community in the deletion version Figure 4.8 does not make much sense: There is actually only one edge between these two communities. We wonder why the algorithm did not instead merge the dark blue community for example with the green community to which it is connected by five edges.

Merging the orange community with the beige community as in the editing version Figure 4.9 makes more sense. The beige is one of the two conferences with the most connections to the orange one. (The other one is the red one.) We still may wonder why the beige community was not instead merged with the blue community on the left as there are even more connections. (7 vs 5)

Note that both the deletion version Figure 4.8 and the editing version Figure 4.9 merged the violet and the green conference. This actually makes sense as the green community has 8 connections to the violet conference but to each of the other conferences it has only less than five connections. Also the violet conference has only less than five connections to each of the other conferences. There is also some similarity in the neighbourhood of those two conferences. Both of them have many connections to the dark blue conference on the right of the green conference, to the pink conference and to the dark green conference. However there is also some difference as the green conference has more connections to the magenta conference and the violet conference has more connections to the dark red conference.

As a summary we conclude that the resulting communities are rather disappointing. Due to the merging of the conferences we can not separate all of them from each other. And presumingly this problem can not be solved by using a better (P_5, C_5)-free editing or a

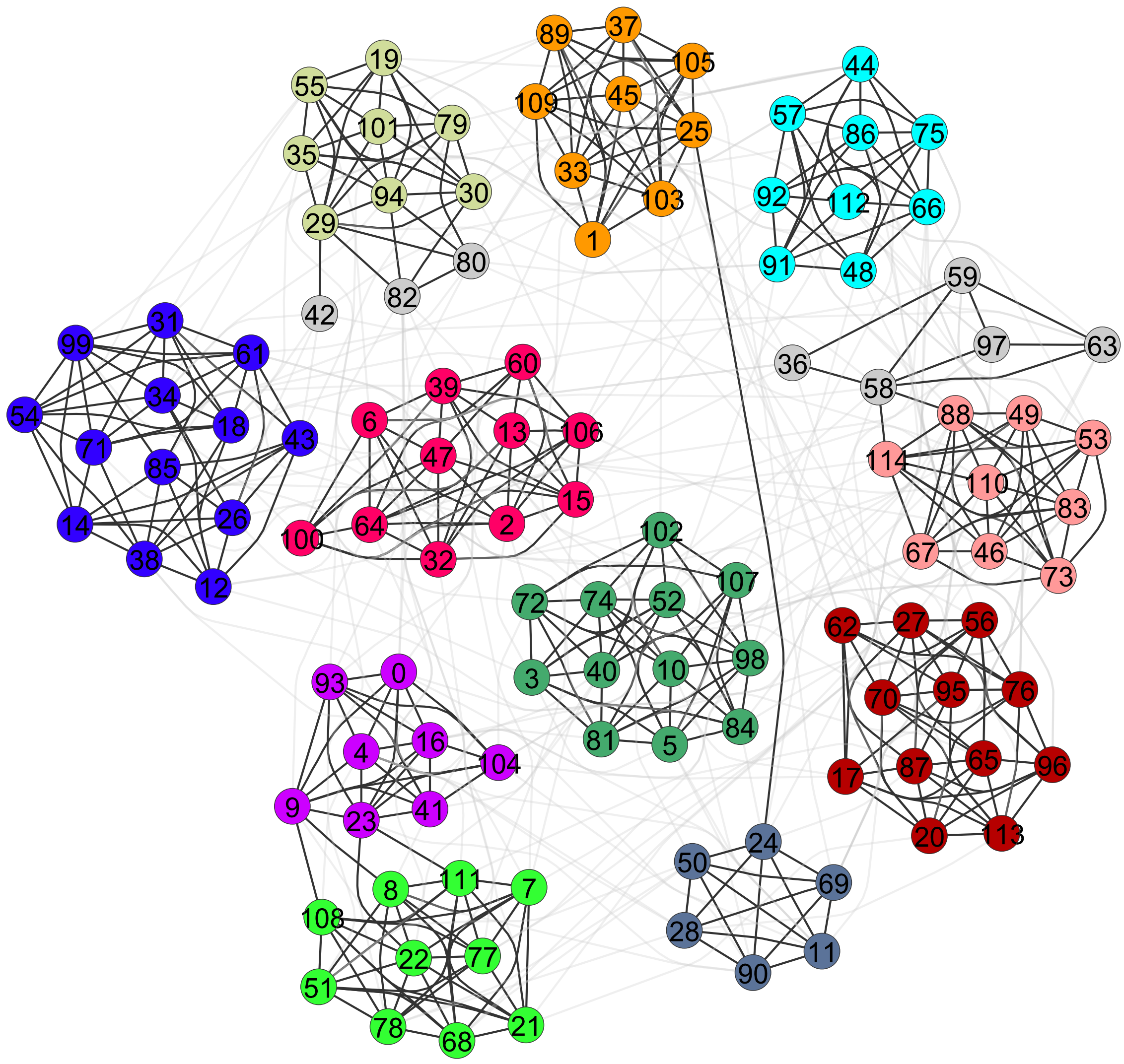


Figure 4.8.: football heuristic deletion + exact deletion = heuristic deletion + exact edit

better (P_5, C_5)-free deletion algorithm. Furthermore the merging of the communities is often rather random and does not even help us in the context of hierarchical clustering. We do not know whether at least the usefulness in the context of hierarchical clustering can be improved by using a better (P_5, C_5)-free editing and (P_5, C_5)-free deletion algorithm.

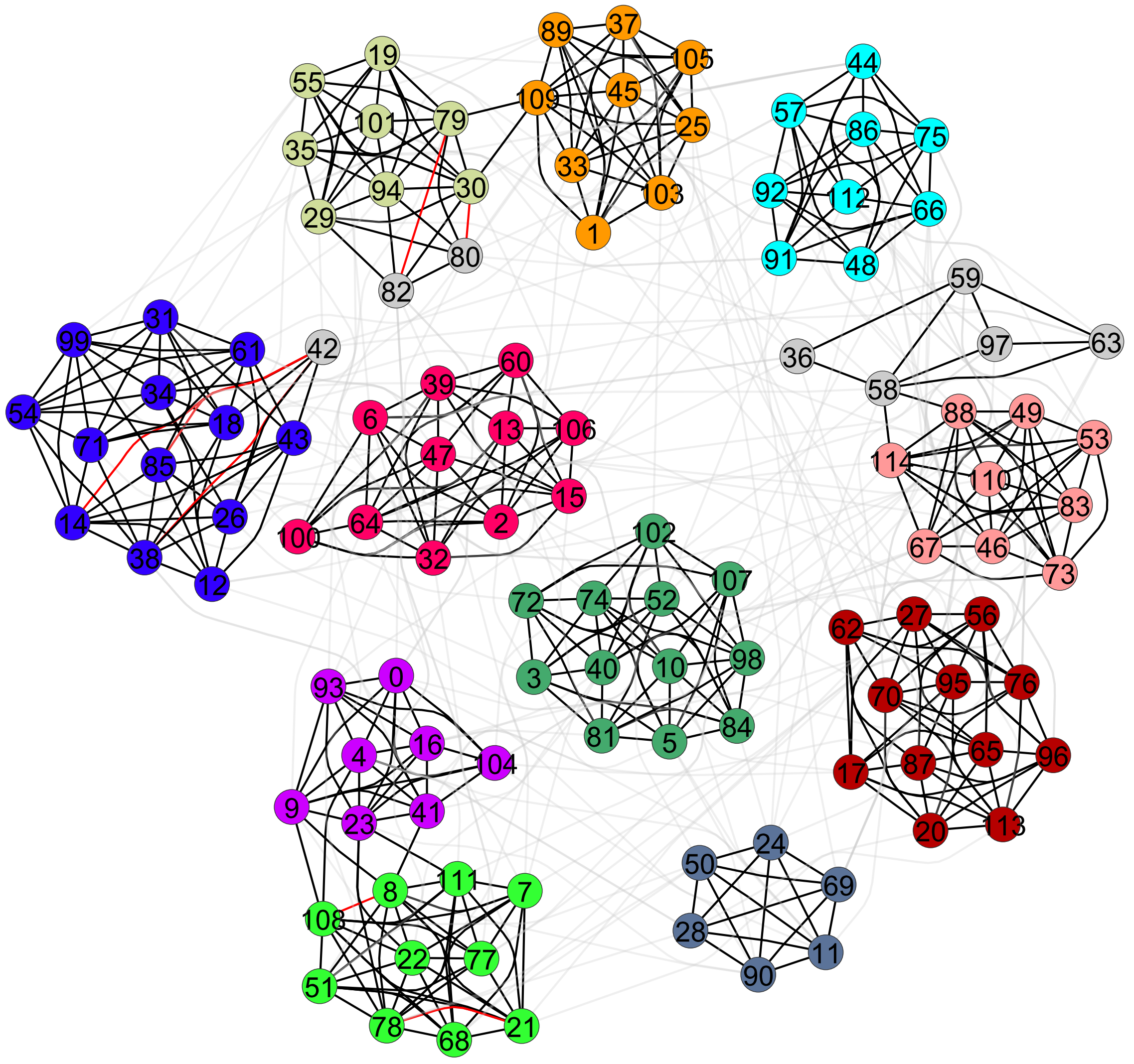


Figure 4.9.: football heuristic edit + exact edit

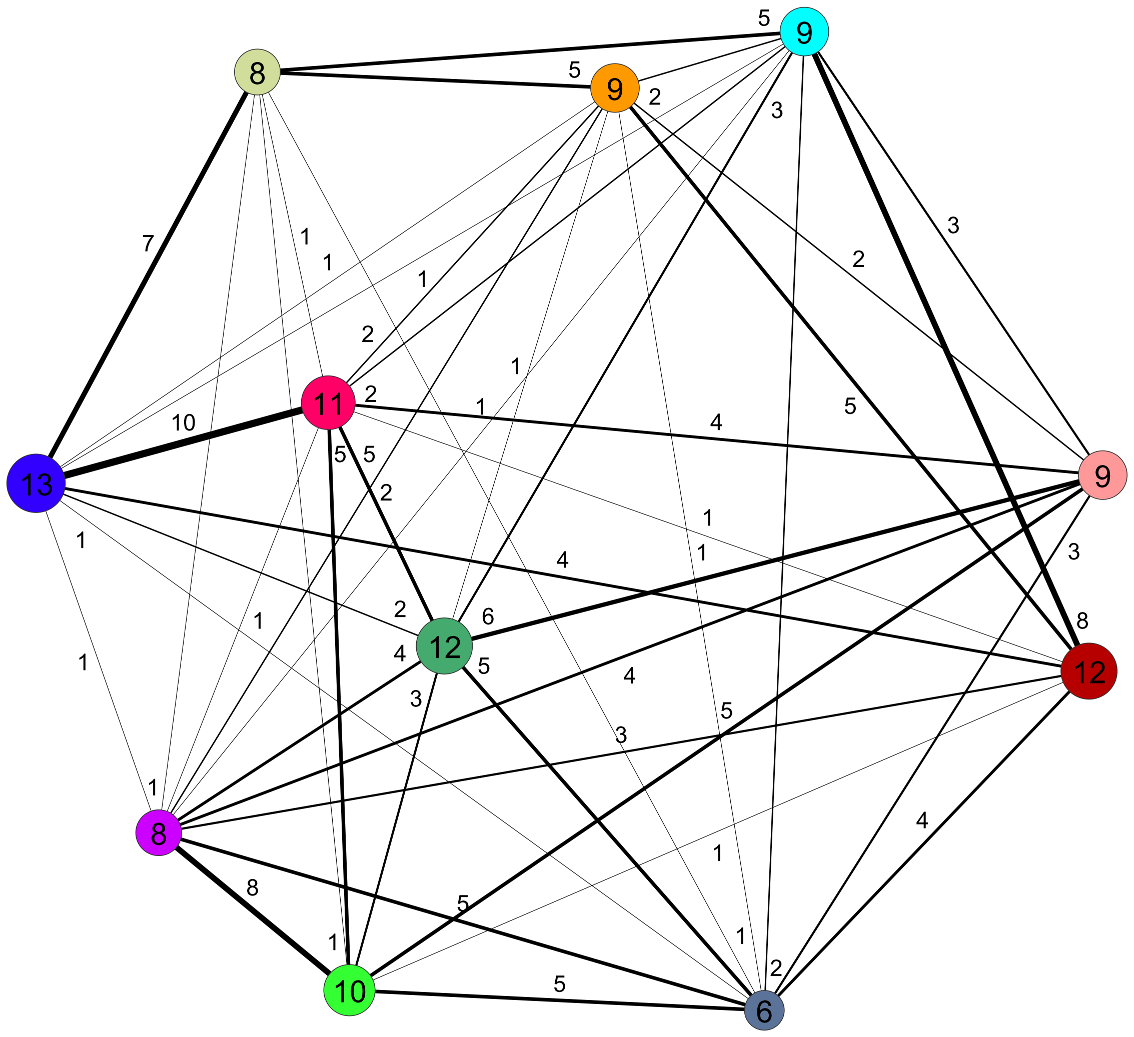


Figure 4.10.: football

5. Conclusion

In this bachelor thesis we have gained many theoretical and practical results regarding the question for the computational feasibility of the (P_5, C_5)-free editing and the (P_5, C_5)-free deletion problem and regarding the search for appropriate methods to solve these problems:

In our theoretical studies we have proven that the (P_5, C_5)-free editing and the (P_5, C_5)-free deletion problem and many more similar problems like (P_5, C_4, C_5)-free editing are NP-complete. To do so we generalized existing NP-completeness proofs.

We introduced a P_5 and C_5 counting algorithm and a P_5 and C_5 search algorithm, which was a modification of the counting algorithm. We needed these algorithms as subroutines for the (P_5, C_5)-free editing and the (P_5, C_5)-free deletion algorithms. We have shown that one can count the total number of P_5s and C_5s in a graph in $O(m \cdot d^3)$ time. Furthermore one can check whether a graph is (P_5, C_5)-free and return a P_5 or a C_5 if it is not in time $O(m \cdot d^2)$. Here and in the following sentences m denotes the number of edges in the graph and d is the maximum degree. We have adapted an exact editing algorithm, which was explained in [Cai96] and also used by[NG13], to the problem of (P_5, C_5)-free editing and (P_5, C_5)-free deletion. We also proposed some optimizations for this algorithm. We have shown that using this algorithm one can solve the (P_5, C_5)-free editing problem in $O(9^k \cdot m \cdot n^2)$ time and the (P_5, C_5)-free deletion problem in $O(4^k \cdot m \cdot d^2)$ time. Here k denotes the maximum number of allowed edits. When discussing an optimization of the exact editing algorithm we found that the goal of reducing the number of recursive calls and reducing the number of time needed per recursive call by the search for P_5s and C_5s conflict with each other. Furthermore we have described a heuristic for the (P_5, C_5)-free editing and the (P_5, C_5)-free deletion problem, which was also used by [NG13].

Following these theoretic studies we have also done many practical experiments: We found that the mentioned heuristic algorithm yields very bad results for the (P_5, C_5)-free editing problem. This was unexpected as it performed very good in the case of (P_4, C_4)-free editing in other papers. For the (P_5, C_5)-free deletion problem the results of the heuristic were better than for the editing version but still not very good in some cases.

We have also done some practical studies on the aforementioned conflicting goals when optimizing the running time of the exact editing algorithm. We showed that in order to optimize the running time finding a proper balance between both goals is better than completely concentrating on any one of them. By doing so one can reduce the running time by more than a half.

We have shown that while the heuristic algorithm yields often very bad results and the exact algorithm takes too much time for most graphs, we can still find some reasonable results by combining them. But still these results are presumingly not optimal. This is indicated by the fact that we found possible optimizations of the deletion set of one of the graphs when visualizing them.

We have also found practical results regarding the usefulness of the community structure induced by solutions to the (P_5, C_5)-free editing and the (P_5, C_5)-free deletion problem. These results were derived by visualizing the graphs together with the editing sets and the deletion sets calculated by the combination of the heuristic and the exact algorithms.

The (P_5, C_5)-free deletion and (P_5, C_5)-free editing method seems to work well on many graphs and not so well on other graphs. The main problem of this approach is that often the resulting communities are too big. The algorithm sometimes merges several underlying communities into one community. This phenomenon occurred in the graph football and also a bit in the graph lesmis.

Another problem - but less important than the merging problem - is that the communities are often not exactly well defined as there are often several solutions of equal size to the (P_5, C_5)-free editing and the (P_5, C_5)-free deletion problem.

5.1. Future Work

More future research is needed in order to find out how one can predict on which graphs the aforementioned merging phenomenon occurs. Thereby we might find out for which graphs (P_5, C_5)-free editing / deletion is a reasonable approach to identify the communities.

Another open question is whether the merging phenomenon if it occurs can be used in order to identify a hierarchical community structure. That is the question is whether two communities are merged together when they have something to do with each other or whether instead the communities are joined together randomly. In the case of the football graph we found that the algorithms merged together communities that obviously did not belong to each other especially in the deletion version. However we don't know if this result could be improved by a better algorithm or if it is inherent to the (P_5, C_5)-free editing/deletion approach. Furthermore the algorithm also merged two other communities where there was actually a strong connection between them. So it is too soon to already answer this question in this bachelor thesis.

Finally future research should try to find a better heuristic algorithm for the (P_5, C_5)-free deletion and especially for the (P_5, C_5)-free editing problem.

Bibliography

[BBD06] Pablo Burzyn, Flavia Bonomo, and Guillermo Durán. Np-completeness results for edge modification problems. *Discrete Applied Mathematics*, 154(13):1824 – 1844, 2006. Traces of the Latin American Conference on Combinatorics, Graphs and Applications A selection of papers from {LACGA} 2004, Santiago, Chile.

[BHSW15] Ulrik Brandes, Michael Hamann, Ben Strasser, and Dorothea Wagner. Fast quasi-threshold editing. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms - ESA 2015*, volume 9294 of *Lecture Notes in Computer Science*, pages 251–262. Springer Berlin Heidelberg, 2015.

[BT90] G. Bacsó and Zs. Tuza. Dominating cliques in p5-free graphs. *Periodica Mathematica Hungarica*, 21(4):303–308, 1990.

[Cai96] Leizhen Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Information Processing Letters*, 58(4):171 – 176, 1996.

[CK90] Margaret B. Cozzens and Laura L. Kelleher. Dominating cliques in graphs. *Discrete Mathematics*, 86(1–3):101 – 116, 1990.

[DHC95] Hassan Ali Dawah, Bradford A. Hawkins, and Michael F. Claridge. Structure of the parasitoid communities of grass-feeding chalcid wasps. *Journal of Animal Ecology*, 64(6):pp. 708–720, 1995.

[EMC88] Ehab S. El-Mallah and Charles J. Colbourn. Complexity of some edge deletion problems. *IEEE transactions on circuits and systems*, 35(3):354–362, 1988. cited By 19.

[Eva10] T S Evans. Clique graphs and overlapping communities. *Journal of Statistical Mechanics: Theory and Experiment*, 2010(12):P12037, 2010.

[GN02] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[KKK93] Donald Ervin Knuth, Donald Ervin Knuth, and Donald Ervin Knuth. *The Stanford GraphBase: a platform for combinatorial computing*, volume 37. Addison-Wesley Reading, 1993.

[Lus03] David Lusseau. The emergent properties of a dolphin social network. *arXiv preprint cond-mat/0307439*, 2003.

[LWGC12] Yunlong Liu, Jianxin Wang, Jiong Guo, and Jianer Chen. Complexity and parameterized algorithms for cograph editing. *Theoretical Computer Science*, 461(0):45 – 54, 2012. 17th International Computing and Combinatorics Conference (COCOON 2011).

[NG04] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69:026113, Feb 2004.

[NG13] James Nastos and Yong Gao. Familial groups in social networks. *Social Networks*, 35(3):439–450, 2013.

[Zac77] Wayne W. Zachary. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, 33(4):pp. 452–473, 1977.

[Zve03] Igor Edmundovich Zverovich. Perfect connected-dominant graphs. *Discuss. Math. Graph Theory*, 23(1):159–162, 2003.

Appendix

A. Appendix Section 1

Table A.1.: karate editing log

type	P5	C5
ins	1038	13
ins	801	12
ins	611	4
del	515	0
ins	415	0
del	323	0
ins	264	0
del	208	0
ins	157	0
del	114	0
ins	83	0
del	59	0
del	46	0
del	33	0
del	20	0
del	9	0
del	4	0
ins	0	0

Table A.2.: dolphins editing log

type	P5	C5	type	P5	C5	type	P5	C5	type	P5	C5
del	5754	134	del	2258	32	del	595	12	del	62	0
del	5384	132	del	2041	30	del	544	12	ins	50	0
del	5051	111	del	1915	22	del	496	12	del	39	0
del	4739	105	del	1763	21	del	449	12	del	28	0
del	4472	86	del	1630	21	del	372	12	ins	18	0
del	4202	79	del	1487	20	del	335	9	del	11	0
del	3983	73	del	1356	18	del	299	9	ins	6	0
del	3779	67	del	1249	15	ins	269	5	ins	2	0
del	3579	57	del	1140	15	del	239	5	del	0	0
del	3368	54	del	1040	15	ins	207	5			
del	3170	43	del	955	15	del	181	5			
del	2968	41	del	876	15	del	160	0			
del	2775	37	del	783	15	del	136	0			
del	2585	35	del	717	15	del	115	1			
del	2398	33	del	657	12	del	79	0			

Table A.3.: grass_web editing log

type	P5	C5
del	1372	1
del	1155	1
del	982	2
del	829	2
del	726	2
del	636	1
del	548	1
ins	470	1
del	408	0
del	346	0
del	291	0
del	242	0
del	201	0
ins	172	0
ins	145	0
ins	123	0
ins	106	0
ins	90	0
del	75	0
del	61	0
del	48	0
del	35	0
del	25	0
ins	17	0
del	9	0
del	5	0
ins	2	0
ins	0	0

Table A.4.: football editing log

type	P5	C5	type	P5	C5	type	P5	C5	type	P5	C5
del	118446	1211	del	54252	536	del	20137	209	del	3941	75
del	116690	1187	del	53309	510	del	19700	203	del	3769	75
del	114954	1162	del	52356	503	del	19260	192	del	3582	72
del	113261	1146	del	51424	494	del	18818	191	del	3330	72
del	111587	1131	del	50505	479	del	18309	190	del	3183	73
del	109928	1116	del	49583	477	del	17875	190	del	3014	67
del	108279	1110	del	48682	474	del	17449	188	ins	2876	63
del	106668	1077	del	47811	456	del	17027	178	del	2743	63
del	105054	1068	del	46934	451	del	16597	173	del	2586	63
del	103475	1068	del	46061	451	del	16181	173	del	2459	59
del	101919	1048	del	45189	451	del	15770	172	del	2333	55
del	100415	1030	del	44326	450	del	15300	171	del	2189	51
del	98915	1022	del	43330	448	del	14911	171	del	2060	51
del	97415	1006	del	42500	441	del	14530	171	del	1865	51
del	95973	976	del	41665	440	del	14045	170	del	1763	42
del	94536	965	del	40837	440	del	13664	170	del	1629	33
del	93128	941	del	40022	435	del	13281	170	del	1518	33
del	91739	925	del	39237	419	del	12910	170	del	1313	33
del	90346	917	del	38456	413	del	12549	170	del	1219	28
del	88957	910	del	37693	405	del	12130	170	del	1094	22
del	87604	894	del	36935	400	del	11779	162	ins	994	22
del	86260	872	del	36199	387	del	11435	148	del	907	22
del	84920	864	del	35479	384	del	11085	148	del	802	22
del	83600	854	del	34770	380	del	10747	137	del	720	21
del	82301	834	del	34075	375	del	10416	137	del	638	21
del	81004	820	del	33384	368	del	10095	137	del	564	20
del	79721	819	del	32702	367	del	9793	130	del	470	20
del	78458	802	del	32048	367	del	9498	125	del	408	20
del	77214	796	del	31415	363	del	9183	124	del	322	20
del	75970	789	del	30711	362	del	8882	123	del	273	20
del	74747	770	del	30089	356	del	8549	122	del	225	20
del	73553	759	del	29505	336	del	8260	117	del	177	20
del	72355	754	del	28929	323	del	7986	117	del	141	20
del	71192	744	del	28353	316	del	7661	117	ins	117	20
del	70048	724	del	27781	309	del	7392	117	ins	100	13
del	68925	705	del	27236	308	del	7130	117	ins	76	13
del	67812	679	del	26674	304	del	6855	117	del	54	13
del	66705	670	del	26148	288	del	6641	99	ins	41	4
del	65595	667	del	25610	287	del	6422	81	ins	30	0
del	64517	653	del	24976	286	del	6200	81	ins	21	0
del	63437	647	del	24451	278	del	5984	81	del	12	0
del	62382	624	del	23939	260	del	5729	81	del	4	0
del	61328	622	del	23432	260	del	5421	81	ins	0	0
del	60268	621	del	22932	260	del	5214	81			
del	59227	607	del	22442	255	del	5015	81			
del	58180	594	del	21978	225	del	4783	81			
del	57179	569	del	21497	225	del	4523	81			
del	56171	566	del	21036	225	del	4325	81			
del	55203	556	del	20588	212	del	4113	81			

Table A.5.: lesmis character names

vertex number	character name	vertex number	character name
0	Myriel	39	Pontmercy
1	Napoleon	40	Boulatruelle
2	MlleBaptistine	41	Eponine
3	MmeMagloire	42	Anzelma
4	CountessDeLo	43	Woman2
5	Geborand	44	MotherInnocent
6	Champtercier	45	Gribier
7	Cravatte	46	Jondrette
8	Count	47	MmeBurgon
9	OldMan	48	Gavroche
10	Labarre	49	Gillenormand
11	Valjean	50	Magnon
12	Marguerite	51	MlleGillenormand
13	MmeDeR	52	MmePontmercy
14	Isabeau	53	MlleVaubois
15	Gervais	54	LtGillenormand
16	Tholomyes	55	Marius
17	Listolier	56	BaronessT
18	Fameuil	57	Mabeuf
19	Blacheville	58	Enjolras
20	Favourite	59	Combeferre
21	Dahlia	60	Prouvaire
22	Zephine	61	Feuilly
23	Fantine	62	Courfeyrac
24	MmeThenardier	63	Bahorel
25	Thenardier	64	Bossuet
26	Cosette	65	Joly
27	Javert	66	Grantaire
28	Fauchelevent	67	MotherPlutarch
29	Bamatabois	68	Gueulemer
30	Perpetue	69	Babet
31	Simplice	70	Claquesous
32	Scaufflaire	71	Montparnasse
33	Woman1	72	Toussaint
34	Judge	73	Child1
35	Champmathieu	74	Child2
36	Brevet	75	Brujon
37	Chenildieu	76	MmeHucheloup
38	Cochepaille		